A Benchmark for Recipe Understanding in Artificial Agents

Jens Nevens[*], Robin De Haes[*], Rachel Ringe[†], Mihai Pomarlan[‡], Robert Porzel[†,◇], Katrien Beuls[△,◇], Paul Van Eecke[*,◇]

[*]Artificial Intelligence Laboratory, Vrije Universiteit Brussel
{jens, paul}@ai.vub.ac.be

[†]Digital Media Laboratory, University of Bremen
rringe@uni-bremen.de, porzel@tzi.de

[‡]Department of Applied Linguistics, University of Bremen
pomarlan@uni-bremen.de

[△]Faculté d'Informatique, Université de Namur
katrien.beuls@unamur.be

Abstract

This paper introduces a novel benchmark that has been designed as a test bed for evaluating whether artificial agents are able to understand how to perform everyday activities, with a focus on the cooking domain. Understanding how to cook recipes is a highly challenging endeavour due to the underspecified and grounded nature of recipe texts, combined with the fact that recipe execution is a knowledge-intensive and precise activity. The benchmark comprises a corpus of recipes, a procedural semantic representation language of cooking actions, qualitative and quantitative kitchen simulators, and a standardised evaluation procedure. Concretely, the benchmark task consists in mapping a recipe formulated in natural language to a set of cooking actions that is precise enough to be executed in the simulated kitchen and yields the desired dish. To overcome the challenges inherent to recipe execution, this mapping process needs to incorporate reasoning over the recipe text, the state of the simulated kitchen environment, common-sense knowledge, knowledge of the cooking domain, and the action space of a virtual or robotic chef. This benchmark thereby addresses the growing interest in human-centric systems that combine natural language processing and situated reasoning to perform everyday activities.

Keywords: benchmark, recipe execution, natural language understanding, situated reasoning

1. Introduction

Recipes are a type of procedural text that many people interact with in their daily lives. Because of this familiarity, the cooking domain is often used as a test bed to assess the ability of artificial agents to learn to perform everyday activities (Bollini et al., 2013; Kiddon et al., 2015; Jermsurawong and Habash, 2015). However, having agents understand how to execute recipes is a highly challenging endeavour due to the underspecified and grounded nature of recipe texts and the fact that recipe execution is a knowledge-intensive and precise activity. Indeed, recipe texts are grounded in the world since many co-references and anaphoric expressions in recipes do not have an antecedent in the recipe text itself, but rather refer to the result of the execution of earlier recipe steps. Consequently, such co-references cannot be resolved by reasoning over the recipe text alone, but also require to keep track of the state changes they entail in the kitchen environment. Recipe texts are also highly underspecified as information that seems trivial for humans preparing the dish is often completely left out or incomplete through the use of null arguments and ellipses (Kiddon et al., 2015; Ruppenhofer and Michaelis, 2010). However, for artificial agents, this information is crucial in order to carry out the cooking instructions correctly. Therefore, any missing information has to be derived from either common-sense knowledge or knowledge of the cooking domain and the cooking instructions need to be specified a level of precision that allows them to be executed in a simulated kitchen environment.

To illustrate these challenges, consider the following cooking instructions:

> Mix together flour, water, and eggs
> Roll the dough into small balls
> Place on a baking sheet

In the first instruction, the act of cracking the eggs before they can be mixed with flour and water is completely omitted from the recipe text. In this case, an artificial agent would need to combine situated reasoning, common-sense knowledge and knowledge of the cooking domain to derive that (i) eggs can be cracked, (ii) when cracked, eggs consist of egg whites and yolks, (iii) uncracked eggs would lead to egg shells when mixed, (iv) egg shells are

[◇] Joint last authors.

LREC-COLING 2024, pages 22–42
20-25 May, 2024.

not edible, and thus (v) the eggs should be cracked before going in the mixture. Similar examples include the peeling of some vegetables before they can be cut (like onions) or the default tool to use for carrying out certain cooking actions such as beating egg whites (namely a whisk) or cutting onions (namely a knife).

In the second instruction, the noun phrase *"the dough"* does not refer to an entity introduced earlier in the recipe text, but to the result of performing the first cooking action. Thus, to find this referent, the agent has to perform the mixing action and then use knowledge of the cooking domain, e.g. concerning the ingredients of the mixture and the mixture's consistency, to find out that this mixture can indeed be called dough, as opposed to other food products that are present in the kitchen.

The patient of the placing action in the third instruction is left out, but implicitly refers to the result of the previous cooking action. Here, one has to perform the rolling action and then reason over the recipe text, taking into account the sequential nature of cooking instructions as in Kiddon et al. (2015) and Jermsurawong and Habash (2015), to find that it is the resulting balls of dough that should be placed on a baking sheet.

The benchmark that we propose in this paper aims to address the challenges inherent to recipe execution. In particular, the goal is to assess the ability of an artificial agent to follow a recipe specified in natural language and prepare the dish in a simulated kitchen environment. Concretely, the benchmark comprises (i) a corpus of 30 recipes, (ii) a procedural semantic representation language specifying 38 cooking actions, (iii) qualitative and quantitative kitchen simulators, and (iv) a standardised evaluation procedure. The benchmark task consists in mapping a recipe expressed in natural language to a set of cooking actions that is precise enough to be executed in the kitchen simulator and thereby produces the desired dish. An example of such a mapping is presented in Figure 1. Crucially, our evaluation procedure does not compare the resulting set of cooking actions against a gold standard annotation. Instead, we evaluate how close the dish obtained by executing the cooking actions in the simulated kitchen resembles the gold standard dish. Solving this benchmark task thus requires to integrate reasoning over the recipe text, the state of the simulated kitchen, common-sense knowledge, knowledge of the cooking domain, and the action space of a virtual or robotic chef.

While the core task of this benchmark is similar to semantic parsing, it goes beyond that in two ways. First, in order to facilitate situated reasoning, the provided kitchen simulators can be accessed at any point in the process of mapping the recipe text to cooking actions. The use of kitchen simulators, rather than an actual robotic kitchen, allows to focus on a broader range of instructions and on the high-level logic underlying their execution, rather than on the lower-level robotic control systems needed to implement the task. Second, as the same dish can be prepared in many different ways, we compare the prepared dish against the gold standard dish, instead of comparing the resulting semantic specification directly. Given these reasons and the fact that this task cannot be cast to any other standard NLP task, no baseline results using off-the-shelf techniques are provided.

The benchmark does not take the form of a typical machine learning benchmark, as its goal is not to predict gold standard annotations directly. Instead, we provide the entire corpus of 30 recipes annotated with sets of cooking actions together with the ontology and the kitchen simulators as a testbench, without specific training and test sets. Participants to the benchmark are free to determine how much of this data is used for development, training, validation, setting hyperparameters, designing prompts, or whatever is necessary for their particular approach to tackling the benchmark. Next to the benchmark components that we provide, participants are encouraged to use external data, such as general knowledge graphs (e.g. Wikidata (Vrandečić and Krötzsch, 2014)), domain-specific ontologies (e.g. FoodOn (Dooley et al., 2018) or BAALL (Krieg-Brückner et al., 2021)) or large language models (e.g. GPT-3 (Brown et al., 2020)). Our benchmark is accessible online at `https://ehai.ai.vub.ac.be/recipe-execution-benchmark/`.

The remainder of this paper is structured as follows. Section 2 presents the different components of the benchmark in more detail. Section 3 provides an illustrative example of the benchmark task. An overview of related work is presented in Section 4. Finally, Section 5 provides a concluding discussion.

2. The Recipe Execution Benchmark

The recipe execution benchmark consists of four main components: a recipe corpus of 30 recipes (Section 2.3), a procedural semantic representation language specifying 38 cooking actions (Section 2.4), qualitative and quantitative kitchen simulators (Section 2.5), and a standardised evaluation procedure (Section 2.6). Before addressing those, we first elaborate on the kitchen state data structure (Section 2.1) and the cooking ontology (Section 2.2).

2.1. Kitchen States

The kitchen state is a central data structure in the recipe execution benchmark. It offers a complete

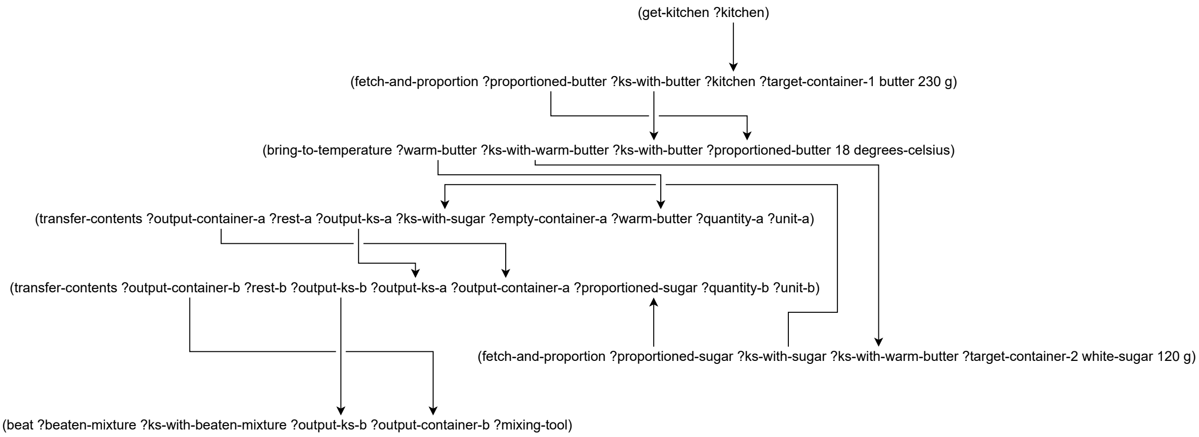

(get-kitchen ?kitchen)
(fetch-and-proportion ?proportioned-butter ?ks-with-butter ?kitchen ?target-container-1 butter 230 g)
(bring-to-temperature ?warm-butter ?ks-with-warm-butter ?ks-with-butter ?proportioned-butter 18 degrees-celsius)
(transfer-contents ?output-container-a ?rest-a ?output-ks-a ?ks-with-sugar ?empty-container-a ?warm-butter ?quantity-a ?unit-a)
(transfer-contents ?output-container-b ?rest-b ?output-ks-b ?output-ks-a ?output-container-a ?proportioned-sugar ?quantity-b ?unit-b)
(fetch-and-proportion ?proportioned-sugar ?ks-with-sugar ?ks-with-warm-butter ?target-container-2 white-sugar 120 g)
(beat ?beaten-mixture ?ks-with-beaten-mixture ?output-ks-b ?output-container-b ?mixing-tool)

Figure 1: A possible set of cooking actions for the cooking instructions *"230 grams of butter, room temperature. 120 grams of sugar. Beat the butter and the sugar together."* This set includes the cooking actions GET-KITCHEN, FETCH-AND-PROPORTION, BRING-TO-TEMPERATURE, TRANSFER-CONTENTS, and BEAT. Arrows between variables denote shared arguments between the cooking actions, pointing from output arguments to input arguments.

specification of the simulated kitchen, regardless of which simulator is used (cf. Section 2.5). The operationalisation of all cooking actions in the procedural semantic representation language (cf. Section 2.4) essentially boils down to manipulating the kitchen state in some way. Kitchen states, like all other entities in the simulated kitchen, can be represented as feature structures, i.e. (possibly nested) sets of attribute-value pairs.

The kitchen itself has an ambient temperature of 18°C. The main food preparation area is the counter top. Apart from this, the kitchen also has an oven, a stove, a microwave, a freezer (at -18°C), a fridge (at 5°C), a pantry and a kitchen cabinet. The ingredients are stored in the freezer, the fridge, and the pantry, while the kitchen cabinet is used to store all tools and utensils.

The execution of a recipe always starts from an initial kitchen state, containing all ingredients, utensils, and appliances that are necessary to be able to prepare the recipes provided with the benchmark. The initial kitchen state can be acquired by executing the cooking action GET-KITCHEN (cf. Section 2.4). A graphical overview of the initial kitchen state is provided in the technical appendix.

2.2. Cooking Ontology

Knowledge about the cooking domain is captured in an ontology that specifies the type system underlying the procedural semantic representation language, the kitchen states, and the kitchen simulators. Specifically, it defines kitchen states, ingredients, tools, modes, and units and relates them in a hierarchical structure. For instance, the ontology is used to indicate (i) that 'brown sugar' and 'white sugar' are types of 'sugar', (ii) that, among others, the freezer, the fridge, and the kitchen cabinet, but also bowls and cookie trays are types of containers, but the latter are transferable containers, while the former are not, and (iii) that certain objects are beatable, boilable, brushable, crackable, cuttable, mashable, meltable, perishable, shakeable, washable, etc. This ontology builds further on the SOMA ontology (Beßler et al., 2021), which aims to capture the physical and the social context of everyday activities and in itself extends the DUL ontology (Masolo et al., 2003). A full specification of the cooking ontology in YAML format is provided in the technical appendix.

2.3. Recipes

The benchmark dataset consists of 30 English recipes sourced from five different websites: AllRecipes.com (`https://www.allrecipes.com`), SimplyRecipes.com (`https://www.simplyrecipes.com`), Food.com (`https://www.food.com`, TheSpruceEats.com (`https://www.thespruceeats.com`), and Cooks.com (`https://www.cooks.com`). The recipes fall in two different categories: 15 recipes for preparing cookies and pastries, and 15 recipes for making salads. All recipes are relatively straightforward for a human to prepare, as they involve a limited number of ingredients, execution steps and required utensils. The recipe texts cover a variety of linguistic phenomena, such as co-references, anaphoric expressions, null arguments, ellipses, hyponyms, meronyms, etc.

The recipes are provided in XML format. For each recipe in the testbench, a set of actions that leads to a perfect end result is provided in the technical appendix. This is for illustrative purposes only, as there may exist other sets of actions that might lead to the same result.

2.4. Procedural Semantic Representation Language

For the purpose of this benchmark, we have designed a representation language in terms of procedural semantics (Woods, 1968; Winograd, 1972; Johnson-Laird, 1977), i.e. semantic representations that can be executed algorithmically. The procedural semantic representation language defines 38 cooking actions that can be executed in the simulated kitchen environments. Cooking actions are represented as predicates that can be declaratively combined by sharing their variable arguments (see Figure 1).

Cooking actions capture atomic operations that an agent should be able to perform in a kitchen setting. The 38 cooking actions that we provide are sufficient to cover at least all recipes in the benchmark. The actions can be grouped in six broad categories: (i) obtaining the kitchen state (GET-KITCHEN), (ii) location altering (e.g. FETCH-AND-PROPORTION and TRANSFER-CONTENTS), (iii) food combination (e.g. MIX, BEAT, SPREAD), (iv) food separation (e.g. SIFT, CRACK, CUT), (v) food manipulation (e.g. MELT, BOIL, WASH) and (vi) tool manipulation (e.g. COVER, LINE, GREASE). These actions are specified at a level of abstraction that allows to focus on the high-level logic of each action, rather than the lower-level robotic control systems needed to implement the action. The implementations are provided by the simulation environment that is used for evaluation (see Section 2.5).

Arguments of cooking actions are always typed according to the types defined in the cooking ontology (cf. Section 2.2). In general, types of arguments of cooking actions can be divided in five broad categories: (i) kitchen states, (ii) food, (iii) tools, (iv) mode specifiers (e.g. cutting patterns, arrangement patterns, etc.) and (v) quantities. Every cooking action has an at least an input kitchen state and an output kitchen state as its arguments. The input kitchen state is always specified after the output kitchen state. Arguments following the input kitchen state are considered input arguments, while arguments preceding the output kitchen state are considered output arguments. The technical appendix provides a detailed technical specification of all cooking actions.

2.5. Simulation

The kitchen simulator is used to execute cooking actions. The simulator has two modes through which actions can be performed: qualitative simulation and quantitative simulation. The qualitative simulation operates on the symbolic level, while the quantitative simulation contains a physics engine that is able to model the physical properties of ingredients and actions. Both simulation modes receive the same initial kitchen state (Section 2.1), have the same underlying ontology (Section 2.2), and implement the same cooking actions (Section 2.4).

The execution of individual cooking actions consists in finding bindings, i.e. computing values, for the output arguments given some values for the input arguments. A value can only be bound to an argument of a cooking action if the type of that value is the same or a subtype of the type specification of the argument (cf. Section 2.2). The input kitchen state of the cooking action represents the state of the kitchen before the action has been executed, while the output kitchen state is a copy of the input kitchen state where the effects of the cooking action have taken place. The remaining input and output arguments are the kitchen entities being manipulated or created by the cooking action.

Executing an entire set of cooking actions thus consists in finding the order in which individual cooking actions can be executed depending on the available bindings. The result is a list of bindings from all variable arguments in the provided cooking actions to specific kitchen entities, thereby grounding the actions in the simulation environment.

Some cooking actions implement a form of default reasoning on the level of their arguments. For instance, the last input argument of the cooking action BEAT is a ?BEATING-TOOL. If no value for this argument is provided, the cooking action can still be executed. It that case, a whisk will be fetched from the kitchen cabinet and used for the beating action. However, if the recipe specifies to beat egg whites with a fork, the aim should be to bind an instance of a fork to the ?BEATING-TOOL argument.

Temporal dependency relations between cooking actions are modelled by keeping track of the simulation time. Concretely, each cooking action specifies the time when the output arguments become available in the simulator based on the time of availability of its input arguments and the duration of the cooking action. Using this information, it can be discovered which cooking actions can be executed in sequence or in parallel.

The kitchen simulator is built using the Incremental Recruitment Language (IRL) system (Van den Broeck, 2008; Spranger et al., 2012; Nevens et al., 2019). IRL is a formalism for operationalising procedural semantics. It provides the necessary abstractions for defining structured representations of the environment (here, kitchen entities) and primitive operations (here, cooking actions), as well as for executing sets of primitive operations w.r.t. the environment in the manner described above. A key benefit of IRL is that it allows full control over the implementation of each cooking action. For the purpose of our kitchen simulator, it is used as an execution engine that delegates the execution of each cooking action to either the qualitative or quantitative simulation.

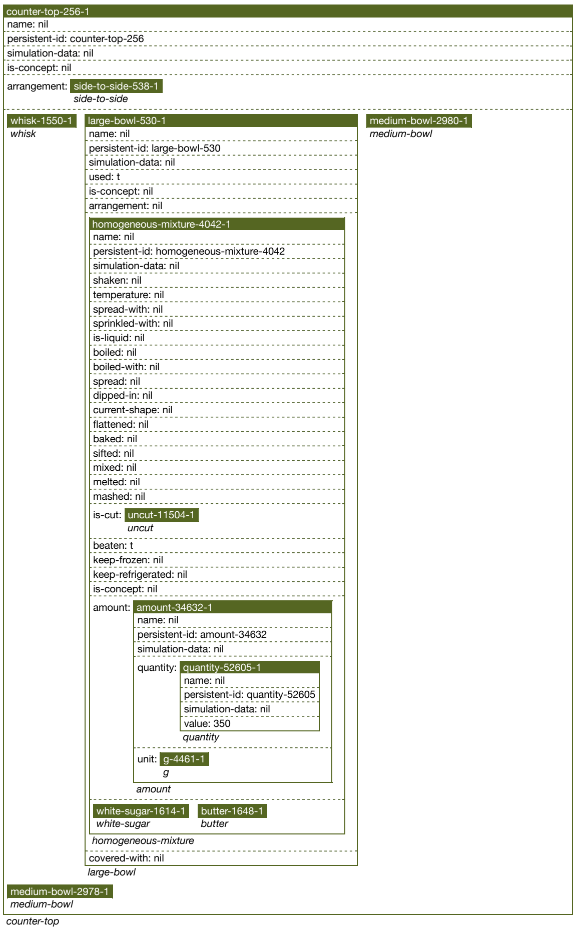
The kitchen simulators can be accessed through API calls. Each call should specify the execution mode, i.e. qualitative or quantitative, and the set of cooking actions to be executed. The resulting bindings are represented as a dictionary where the keys are the variables in the cooking actions and the values are objects from the simulation environment encoded in the JSON format.

2.5.1. Qualitative Simulation

The aim of the qualitative simulator is to model the kitchen environment on a qualitative level (see e.g. Kuipers (1994); Bratko (2012)), thereby abstracting away from the fine-grained, numerical details of what for example physics simulators offer. When a cooking action is executed in this symbolic kitchen environment, it causes one or more meaningful changes in the kitchen state. For instance, the cooking action (FETCH-AND-PROPORTION ?PROPORTIONED-BUTTER, ?KS-WITH-BUTTER, ?KITCHEN, ?TARGET-CONTAINER-1, BUTTER, 230, G) from Figure 1 finds an empty bowl in the kitchen cabinet and detects butter in the fridge, places both the bowl and the container of butter on the counter top, transfers 230g of that butter to the empty bowl, and puts the remainder of the butter back into the fridge. This leads to a new kitchen state (?KS-WITH-BUTTER) that is a copy of the previous kitchen state (?KITCHEN) with the following changes: (i) the butter container in the fridge contains 230g less butter, (ii) an empty bowl (?TARGET-CONTAINER-1) has been removed from the kitchen cabinet, and (iii) the counter top contains a bowl with 230g of butter (?PROPORTIONED-BUTTER). An example kitchen state obtained by executing all cooking actions from Figure 1 is provided in Figure 2. In this figure, only the counter top is visualised and only the large bowl with homogeneous mixture is fully expanded.

2.5.2. Quantitative Simulation

In contrast to the qualitative simulation, the quantitative simulation models the physical properties of ingredients and manipulation of objects in more detail. This simulation is built on top of the widely adopted PyBullet robotics simulator package (Coumans and Bai, 2016–2023). In turn, PyBullet uses OpenGL and benefits from – but does not require – hardware acceleration via GPUs. A screenshot of the kitchen environment in the qualitative simulation is provided in Figure 3.



```
counter-top-256-1
name: nil
persistent-id: counter-top-256
simulation-data: nil
is-concept: nil
arrangement: side-to-side-538-1
             side-to-side
whisk-1550-1   large-bowl-530-1                                     medium-bowl-2980-1
whisk          name: nil                                            medium-bowl
               persistent-id: large-bowl-530
               simulation-data: nil
               used: t
               is-concept: nil
               arrangement: nil
                 homogeneous-mixture-4042-1
                 name: nil
                 persistent-id: homogeneous-mixture-4042
                 simulation-data: nil
                 shaken: nil
                 temperature: nil
                 spread-with: nil
                 sprinkled-with: nil
                 is-liquid: nil
                 boiled: nil
                 boiled-with: nil
                 spread: nil
                 dipped-in: nil
                 current-shape: nil
                 flattened: nil
                 baked: nil
                 sifted: nil
                 mixed: nil
                 melted: nil
                 mashed: nil
                 is-cut: uncut-11504-1
                         uncut
                 beaten: t
                 keep-frozen: nil
                 keep-refrigerated: nil
                 is-concept: nil
                 amount: amount-34632-1
                         name: nil
                         persistent-id: amount-34632
                         simulation-data: nil
                         quantity: quantity-52605-1
                                   name: nil
                                   persistent-id: quantity-52605
                                   simulation-data: nil
                                   value: 350
                                   quantity
                         unit: g-4461-1
                               g
                         amount
                 white-sugar-1614-1   butter-1648-1
                 white-sugar          butter
                 homogeneous-mixture
               covered-with: nil
               large-bowl
medium-bowl-2978-1
medium-bowl
counter-top
```

Figure 2: Example representation offered by the qualitative kitchen simulator after executing the cooking actions specified in Figure 1.

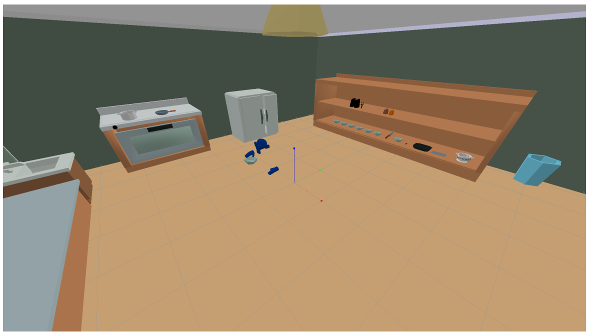


Figure 3: Screenshot of the qualitative kitchen simulation environment.

In executing cooking actions, lower-level details about the execution, such as where exactly to place an object, are decided by the simulation itself. Simulation in PyBullet covers rigid body physics. The simulation environment is three-dimensional, with objects being represented by low-polygon count meshes. Liquids are approximated by collections of particles. Other aspects, such as baking or cutting, are approximated via so-called custom dynamics.

These are scripts running in parallel to the simulation that check whether triggering motions occur. For cutting, for instance, a triggering motion is when the blade part of an object that can cut (determined via the ontology) approaches an object that is cuttable (determined via the ontology). As a result, the motion event triggers and the original object is replaced by a collection of separated parts.

Next to all object properties from the qualitative simulation, the kitchen states in the quantitative simulation keeps a full description of the physical state of objects in terms of position, orientation, linear and angular velocity, and any other state variables that are relevant to the custom dynamics implemented via scripting. The quantitative simulation thus complements the qualitative simulation in that it provides information about the influence of general physics on the kitchen state rather than about higher-level qualitative state changes.

2.6. Evaluation Procedure

The focus of our evaluation procedure lies on measuring the successful execution of the cooking actions obtained by processing the recipe text. We introduce a simulation-based metric called the dish approximation score (Section 2.6.1). Together with this metric, the quality of the obtained cooking actions can be gauged using a semantic similarity measure (Section 2.6.2) and the kitchen simulator's execution time (Section 2.6.3). Finally, we provide a graphical evaluation tool that allows participants of the benchmark to easily compute these metrics (Section 2.6.4).

2.6.1. Dish Approximation Score

The dish approximation score is inspired by existing metrics used to gauge the performance of compositional directives with non-reversible state changes (Shridhar et al., 2020). With this score, we aim to quantify how similar two dishes are independently from the steps involved in their preparation.

The dish approximation score is computed for all objects in the final kitchen state and the maximum score is returned. This is because the output argument of the final cooking action does not necessarily correspond to the final dish, as other operations might still be executed after the dish is ready, e.g. putting away utensils or cleaning up the kitchen. However, this approach yields satisfactory results since the final dish will always be present somewhere in the final kitchen state, e.g. one of the kitchen entities on the counter top.

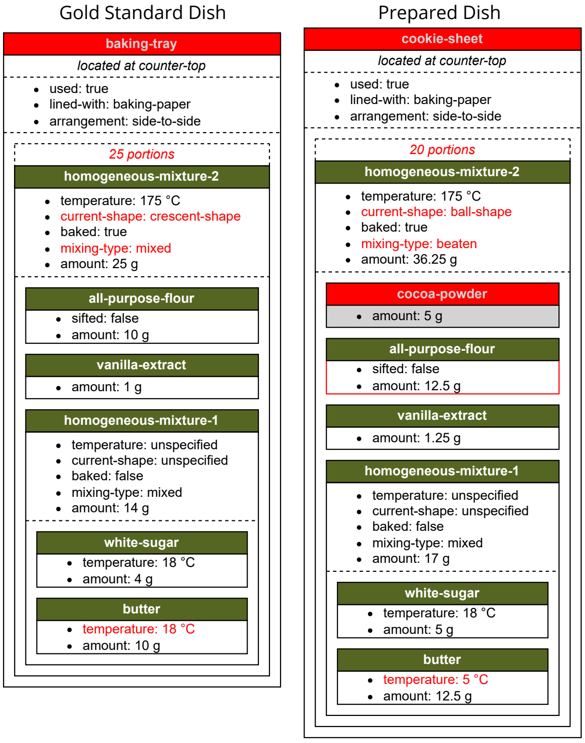


Figure 4: Illustrative example of what differences in attribute values negatively impact the dish approximation score (highlighted in red).

The dish approximation score is computed by comparing the attributes and values of a given kitchen entity and the gold standard dish (see Figure 4). The final score is a weighted sum where 2% of the points count towards the presentation (e.g. where the dish is located and how many portions are prepared) and 98% of the points count towards the contents. To compare the dish's contents, we compare the food products the dish is made of and in which quantities. Each food product may be composed of a combination of other food products. These are first decomposed until the base ingredients are reached. Base ingredients are the food products that cannot be further decomposed in terms of the level of granularity of the benchmark. For instance, in Figure 4, the HOMOGENEOUS-MIXTURE-2 in the prepared dish consists of COCOA-POWDER, ALL-PURPOSE-FLOUR, VANILLA-EXTRACT, and HOMOGENEOUS-MIXTURE-1, which itself is composed of WHITE-SUGAR and BUTTER.

The computation of the dish approximation score starts from the base ingredients. Concretely, for every base ingredient in the gold standard dish, we look up the most similar base ingredient in the prepared dish. The score for this base ingredient is the number of attributes and values both ingredients have in common, divided by the total number of attributes and values of the gold standard ingredient. In addition to the properties of the base ingredients, we also check the sequence of intermediate food products they are used in, and how many properties these intermediate food products have in common. Gold standard ingredients that are missing in the prepared dish or prepared ingredients that are in excess are given a similarity score of zero. The similarity scores are again combined by a weighted sum, where 60% counts towards the similarity of the base ingredients and 40% counts towards the similarity of the intermediate food products. All weights involved in computing the dish approximation score were chosen to reflect human intuition after small-scale experiments with the 30 benchmark recipes. Figure 4 highlights differences in attribute values that negatively influence the dish approximation score. Note that the entities shown in Figure 4 are for illustrative purposes and that the actual entities used in the simulation environments generally have more properties.

2.6.2. Smatch Score

A direct comparison of the cooking actions performed by an artificial agent with some example annotation can already give a first indication of how closely the end result will resemble the gold-standard dish. To this end, we provide a modified version of the Smatch score (Cai and Knight, 2013) that first converts the cooking actions from the procedural semantic representation language to triples before computing the maximum of F-scores. However, as the Smatch score is based purely on the semantic structure, its value decreases whenever some cooking actions are performed in a different order even though the end result remains the same. As this frequently occurs in recipes, the value of the Smatch score should always be interpreted as an indication together with the other metrics.

2.6.3. Recipe Execution Time

The recipe execution time measures the number of timesteps that are needed to execute a set of cooking actions in terms of the kitchen simulator's internal clock. This metric can be used to gauge cooking efficiency, which could be an indication of insight into the recipe. However, a lower execution time does not necessarily indicate better performance on the benchmark as it can be caused by insufficient understanding of the recipe text, thereby performing less or incorrect cooking actions. Thus, the execution time should always be interpreted in light of the other evaluation metrics.

2.6.4. Evaluation Tool

The benchmark comes with an evaluation tool that can be used to obtain the aforementioned metrics. The evaluation tool expects a single input file that specifies the unique identifier of a recipe (e.g. #ALMOND-CRESCENT-COOKIES) followed by the set of all cooking actions that is obtained by processing the entire recipe text. Each cooking action must be specified on a separate line. Through the use of recipe identifiers, the same input file can be used to evaluate multiple recipes. The evaluation tool produces a single file containing the evaluation metrics for each recipe on a separate row. The evaluation tool also allows to visualise the entire simulation process on a web interface.

3. Illustrative Example

Figure 5 provides an illustrative example of one possible approach for tackling the benchmark task. In this approach, the recipe text is processed line per line, interleaved with the execution of the resulting cooking actions in the qualitative or quantitative kitchen simulator. This way, the approach makes maximum use of the kitchen simulators to facilitate situated reasoning over the recipe. At the start of the recipe, a binding of the initial kitchen state to the variable ?INITIAL-KS is obtained by executing the action GET-KITCHEN. The kitchen state ?INITIAL-KS together with any other sources of information, such as the cooking ontology, the action space of the artificial chef, general knowledge graphs, large language models, etc., can be used to map the instruction *"230 grams of butter, room temperature"* to a set of possible cooking actions. In this example, this results in the cooking actions FETCH-AND-PROPORTION and BRING-TO-TEMPERATURE. Note the use of the variable ?INITIAL-KS in the action FETCH-AND-PROPORTION. This action thus operates over the initial kitchen state. Next, these two cooking actions are executed using either the qualitative or quantitative kitchen simulator. This results in a new kitchen state ?KS-WITH-BUTTER, which includes a bowl with 230 grams of butter at room temperature ?WARM-BUTTER. Now, this new kitchen state, the bowl of butter and any other information can be used in processing the next instruction of the recipe, yielding another set of cooking actions, which is then executed, and so on. All kitchen entities in the final kitchen state ?KS-6 are compared against the gold standard dish and the kitchen entity with the highest dish approximation score is returned.

Given this example approach, we draw the attention to two aspects that are crucial when tackling the recipe execution benchmark. First, there is the challenge of mapping the instructions from the recipe text to the appropriate set of cooking actions. This requires reasoning over the recipe text, the current state of the simulated kitchen, commonsense and domain-specific knowledge and the action space of the virtual or robotic chef. For example, in order to beat the butter and sugar together, these ingredients need to be combined in a single container. From the previous kitchen state, it can be derived that there are bowls of proportioned sugar and butter on the counter top. Hence, two

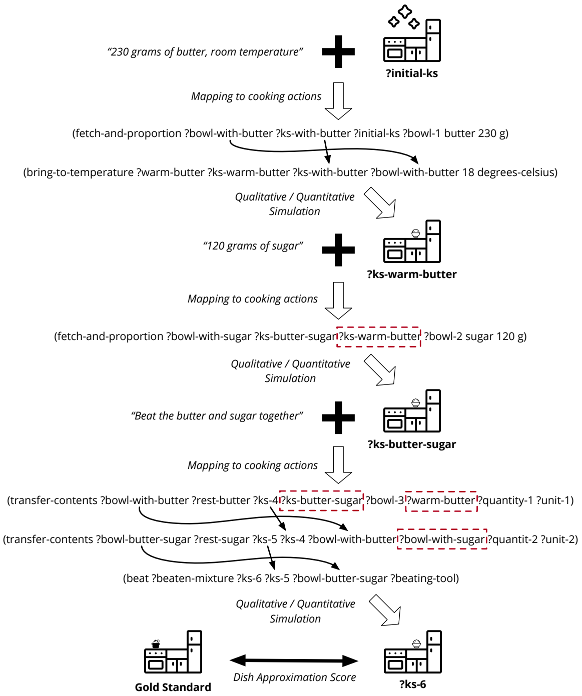

"230 grams of butter, room temperature"
?initial-ks
Mapping to cooking actions
(fetch-and-proportion ?bowl-with-butter ?ks-with-butter ?initial-ks ?bowl-1 butter 230 g)
(bring-to-temperature ?warm-butter ?ks-warm-butter ?ks-with-butter ?bowl-with-butter 18 degrees-celsius)
Qualitative / Quantitative Simulation
"120 grams of sugar"
?ks-warm-butter
Mapping to cooking actions
(fetch-and-proportion ?bowl-with-sugar ?ks-butter-sugar ?ks-warm-butter ?bowl-2 sugar 120 g)
Qualitative / Quantitative Simulation
"Beat the butter and sugar together"
?ks-butter-sugar
Mapping to cooking actions
(transfer-contents ?bowl-with-butter ?rest-butter ?ks-4 ?ks-butter-sugar ?bowl-3 ?warm-butter ?quantity-1 ?unit-1)
(transfer-contents ?bowl-butter-sugar ?rest-sugar ?ks-5 ?ks-4 ?bowl-with-butter ?bowl-with-sugar ?quantit-2 ?unit-2)
(beat ?beaten-mixture ?ks-6 ?ks-5 ?bowl-butter-sugar ?beating-tool)
Qualitative / Quantitative Simulation
Gold Standard
Dish Approximation Score
?ks-6

Figure 5: Illustrative example of one possible approach for tackling the benchmark task.

TRANSFER-CONTENTS actions are required to transfer both sugar and butter in a new bowl before the BEAT action can be executed. Second, notice how the processing of individual utterances results in sets of cooking actions that re-use the same variables. These variables are highlighted in red in Figure 5. Concretely, the variable ?KS-WITH-WARM-BUTTER in the second network was introduced in the first network, while ?KS-BUTTER-SUGAR, ?WARM-BUTTER, and ?BOWL-WITH-SUGAR are variables in the third network originating from the first and second network. This re-use of variables is essential for carrying out the cooking actions correctly, as, for example, a kitchen state with the right amount of butter and sugar is required for processing and executing the instruction *"Beat the butter and sugar together"*. Indeed, in terms of language processing, the use of *"the butter and sugar"* indicates that these referents were introduced before, while in terms of execution, these entities need to be found in the current kitchen state. In general, the re-use of variables allows cooking actions to refer back to previous cooking actions or their execution results (as in *"the dough"* in Section 1). From these examples, it is clear that the proposed benchmark task goes beyond semantic parsing tasks. In this benchmark, the processing of the recipe text can only yield correct networks of cooking actions when combined with information from previous recipe instructions, their execution in the simulated kitchen, common-sense and domain-specific knowledge, and the action space of the virtual or robotic chef. The main challenge of the benchmark thus lies in the integration of these various components and reasoning over the different sources of knowledge.

An additional, interactive example can be accessed via `https://ehai.ai.vub.ac.be/demos/recipe-understanding/`. The semantic parsing part is here operationalised using the Fluid Construction Grammar framework (Steels, 2004; van Trijp et al., 2022; Beuls and Van Eecke, 2023, 2024).

4. Benchmarks for Recipe Execution

The aim of the recipe execution benchmark proposed is this paper is to assess the ability of an artificial agent to understand and reason about natural language recipes. Closely related to this aim are the tasks of semantic parsing of recipe texts (Section 4.1) and the robotic execution of recipes (Section 4.2).

4.1. Semantic Parsing for Recipes

There exists a wide variety of semantic annotation schemes that aim to address the challenges commonly found in the semantic parsing of recipe texts.

The Carnegie Mellon University Recipe Database (CURD) (Tasse and Smith, 2008) consists of 260 English recipes annotated with the Minimal Instruction Language for the Kitchen (MILK) annotation scheme. MILK consists of 12 high-level cooking actions expressed as first-order logic predicates. The execution of these operations modifies a symbolic kitchen state that keeps track of ingredients and tools. Tasse and Smith (2008) report only preliminary results on parsing the recipe text into the correct sequence of predicates, without predicting the predicates' arguments. More recently, LLMs were used for tackling this particular semantic parsing task (Cohen and Mooney, 2023).

The Simplified Ingredient Merging Map in Recipes annotation scheme (SIMMR) uses the MILK annotations of the CURD dataset to generate a dependency tree that related recipe instructions to either previous instructions or ingredients (Jermsurawong and Habash, 2015). It thereby offers a more high-level, but also more coarse-grained representation of the structure of recipes compared to MILK.

With the aim of moving away from a domain-specific annotation scheme, the Recipe Instruction Semantics Corpus (RISeC) (Jiang et al., 2020) annotates the CURD recipes with a frame-based annotation scheme, using PropBank frames (Kingsbury and Palmer, 2002). Co-reference links across sentences as well as natural language description of implicit references to earlier concepts are added. BERT-based models are used for entity recognition, relation extraction, and zero anaphora identification tasks.

Kiddon et al. (2015) annotated recipes using so-called action graphs. These graphs are composed of predicates that share some of their arguments. The predicates are tied to verbs in the recipe, while the arguments are tied to food items and locations. They annotated a corpus of 2456 recipes. The authors propose an unsupervised machine learning approach using EM algorithms.

Finally, recipes are annotated using flow graphs. Based on initial work by Mori et al. (2012), the Recipe Flow Graph (r-FG) Corpus (Mori et al., 2014) consists of 266 Japanese recipes annotated with predicate-argument structures. These graphs combine syntactic relations (e.g. subj, d-obj, i-obj) with cooking-specific relations (e.g. food part-of, tool complement) to represent the structure of a recipe in a single graph. Later, an English Recipe Flow Graph Corpus of 300 recipes was released by Yamakata et al. (2020) and semantic parsers for this corpus have recently been presented by Donatelli et al. (2021) and Fan and Hunter (2023). Multi-modal annotations using recipe flow graphs are also being provided by grounding the nodes of the graph in image data using bounding boxes (Nishimura et al., 2020) or image pairs (Shirai et al., 2022).

4.2. Systems for Recipe Execution

Systems for recipe execution combine robotic manipulation, computer vision, knowledge representation and reasoning into an integrated approach.

Neural Process Networks (Bosselut et al., 2018) use a neural network architecture to integrate cooking actions and entities. They model the understanding of a recipe as finding a sequence of kitchen state changes induced by executing actions on kitchen entities in simulation. This simulation is performed by updating and tracking a set of a priori specified actions and entity embeddings that encode information along six relevant dimensions, namely location, cookedness, temperature, composition, shape and cleanliness.

The MIT BakeBot project (Bollini et al., 2013) operationalises the execution of different recipes by a physical robot. From a dataset of recipes annotated with sequences of states and actions, BakeBot first learns a policy in simulation. To cook a dish, BakeBot then uses this policy to map the recipe instructions to a sequence of states and actions, after which they are executed on a physical robot. However, each action first needs to be translated to multiple, low-level motion operations. The instructions that the robot can execute are rather limited due to the need to implement them in physical reality and some actions require human assistance to be executed.

Moving away from cooking instructions, the CRAM cognitive architecture presented by Beetz et al. (2011, 2023) enables a variety of physical robots to complete everyday manipulation tasks, such as setting the table. The central principle of their architecture is the ability to propagate information both bottom-up, from action and perception modules, and top-down, from high-level planning and meta-level reasoning components to a central declarative reasoning system.

Finally, Höffner et al. (2022) focus on everyday activity commands in the context of household robotics. They have designed a processing pipeline consisting of components for semantic parsing, language grounding, and simulation, with an overarching ontological framework. However, their pipeline currently only handles single directives in isolation.

5. Conclusion

We have presented a novel benchmark that has been designed as a test bed for evaluating whether artificial agents are able to understand how to perform everyday activities in the cooking domain. Understanding how to execute recipes is a highly challenging endeavour due to the underspecified and grounded nature of recipe texts and the fact that recipe execution is a knowledge-intensive and precise activity. The benchmark task that we propose consists in mapping recipes specified in natural language to a set of cooking actions that is concrete and detailed enough to be executed in a simulated kitchen environment, while also producing the desired dish. To this end, we provided (i) a corpus of 30 recipes, (ii) a procedural semantic representation language of 38 cooking actions, (iii) qualitative and quantitative kitchen simulators, and (iv) a standardised evaluation procedure. To overcome the challenges inherent to recipe execution, the mapping of recipe texts to cooking actions requires to integrate reasoning over the recipe text, the state of the simulated kitchen, common-sense knowledge, knowledge of the cooking domain, and the action space of a virtual or robotic chef. This benchmark distinguishes itself from other semantic parsing and recipe execution benchmarks in that (i) kitchen simulators can be used at any time to facilitate situated reasoning and (ii) evaluation is not in terms of a gold-standard annotation, but in terms of the quality of the prepared dish. The benchmark thereby addresses the growing interest in human-centric systems that combine natural language processing and situated reasoning to perform everyday activities (Steels, 2023; Verheyen et al., 2023).

6. Acknowledgements

The research reported on in this paper received funding from the EU's H2020 RIA programme under grant agreement no. 951846 (MUHAI), the Research Foundation Flanders (FWO) through a postdoctoral grant awarded to PVE (grant no. 76929), and from the Collaborative Research Center (SFB) 1320 EASE – Everyday Activity Science and Engineering, University of Bremen (www.easecrc.org), sub-project P01 "Embodied Semantics for the Language of Action and Change".

7. Bibliographical References

Michael Beetz, Gayane Kazhoyan, and David Vernon. 2023. The CRAM cognitive architecture for robot manipulation in everyday activities. *arXiv preprint arXiv:2304.14119*.

Michael Beetz, Ulrich Klank, Ingo Kresse, Alexis Maldonado, Lorenz Mösenlechner, Dejan Pangercic, Thomas Rühr, and Moritz Tenorth. 2011. Robotic roommates making pancakes. In *Proceedings of the 11th IEEE-RAS International Conference on Humanoid Robots*, pages 529–536, New York, NY, USA. IEEE.

Daniel Beßler, Robert Porzel, Mihai Pomarlan, Abhijit Vyas, Sebastian Höffner, Michael Beetz, Rainer Malaka, and John Bateman. 2021. Foundations of the socio-physical model of activities (soma) for autonomous robotic agents. In *Formal Ontology in Information Systems*, pages 159–174, Amsterdam, Netherlands. IOS Press.

Katrien Beuls and Paul Van Eecke. 2023. Fluid Construction Grammar: State of the art and future outlook. In *Proceedings of the First International Workshop on Construction Grammars and NLP (CxGs+NLP, GURT/SyntaxFest 2023)*, pages 41–50. Association for Computational Linguistics.

Katrien Beuls and Paul Van Eecke. 2024. Construction grammar and artificial intelligence. In Mirjam Fried and Kiki Nikiforidou, editors, *The Cambridge Handbook of Construction Grammar*. Cambridge University Press, Cambridge, United Kingdom. Forthcoming.

Mario Bollini, Stefanie Tellex, Tyler Thompson, Nicholas Roy, and Daniela Rus. 2013. Interpreting and executing recipes with a cooking robot. In Jaydev P. Desai, Gregory Dudek, Oussama Khatib, and Vijay Kumar, editors, *Experimental Robotics: The 13th International Symposium on Experimental Robotics*, pages 481–495. Springer-Verlag, Heidelberg, Germany.

Antoine Bosselut, Omer Levy, Ari Holtzman, Corin Ennis, Dieter Fox, and Yejin Choi. 2018. Simulating action dynamics with neural process networks. In *6th International Conference on Learning Representations (ICLR 2018)*, page 10.

Ivan Bratko. 2012. *Prolog Programming for Artificial Intelligence (Fourth Ed.)*. Pearson Education, Harlow, United Kingdom.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*, pages 1877–1901, Red Hook, NY, USA. Curran Associates Inc.

Shu Cai and Kevin Knight. 2013. Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 748–752. Association for Computational Linguistics.

Vanya Cohen and Raymond Mooney. 2023. Using planning to improve semantic parsing of instructional texts. In *Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations (NLRSE)*, pages 47–58. Association for Computational Linguistics.

Erwin Coumans and Yunfei Bai. 2016–2023. Pybullet: A Python module for physics simulation for games, robotics and machine learning. http://pybullet.org.

Lucia Donatelli, Theresa Schmidt, Debanjali Biswas, Arne Köhn, Fangzhou Zhai, and Alexander Koller. 2021. Aligning actions across recipe graphs. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6930–6942. Association for Computational Linguistics.

Damion M. Dooley, Emma J. Griffiths, Gurinder S. Gosal, Pier L. Buttigieg, Robert Hoehndorf, Matthew C. Lange, Lynn M. Schriml, Fiona S.L. Brinkman, and William W. L. Hsiao. 2018. FoodOn: a harmonized food ontology to increase global food traceability, quality control and data integration. *npj Science of Food*, 2(1):23.

Yi Fan and Anthony Hunter. 2023. Understanding the cooking process with english recipe text. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 4244–4264. Association for Computational Linguistics.

Sebastian Höffner, Robert Porzel, Maria M. Hedblom, Mihai Pomarlan, Vanja Sophie Cangalovic, Johannes Pfau, John A. Bateman, and Rainer Malaka. 2022. Deep understanding of everyday activity commands for household robots. *Semantic Web*, 13(5):895–909.

Jermsak Jermsurawong and Nizar Habash. 2015. Predicting the structure of cooking recipes. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 781–786. Association for Computational Linguistics.

Yiwei Jiang, Klim Zaporojets, Johannes Deleu, Thomas Demeester, and Chris Develder. 2020. Recipe instruction semantics corpus (RISeC): Resolving semantic structure and zero anaphora in recipes. In *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing*, pages 821–826. Association for Computational Linguistics.

Philip N. Johnson-Laird. 1977. Procedural semantics. *Cognition*, 5(3):189–214.

Chloé Kiddon, Ganesa Thandavam Ponnuraj, Luke Zettlemoyer, and Yejin Choi. 2015. Mise en place: Unsupervised interpretation of instructional recipes. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 982–992. Association for Computational Linguistics.

Paul R. Kingsbury and Martha Palmer. 2002. From treebank to propbank. In *Proceedings of the 3rd International Conference on Language Resources and Evaluation*, pages 1989–1993, Paris, France. European Language Resources Association (ELRA).

Bernd Krieg-Brückner, Serge Autexier, and Mihai Pomarlan. 2021. The baall ontology-configuration of service robots, food, and diet. In *FOIS 2021 Ontology Showcase, held at FOIS 2021 - the 12th International Conference on Formal Ontology in Information Systems*. CEUR Workshop Proceedings.

Benjamin Kuipers. 1994. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. MIT Press, Cambridge, MA, USA.

Claudio Masolo, Stefano Borgo, Aldo Gangemi, Nicola Guarino, and Alessandro Oltramari. 2003. Wonderweb deliverable d18-ontology library (final). Technical report, National Research Council-Institute of Cognitive Science and Technology. IST Project 2001-33052 WonderWeb: Ontology Infrastructure for the Semantic Web.

Shinsuke Mori, Hirokuni Maeta, Yoko Yamakata, and Tetsuro Sasada. 2014. Flow graph corpus from recipe texts. In *Proceedings of the 9th International Conference on Language Resources and Evaluation (LREC)*, pages 2370–2377, Paris, France. European Language Resources Association (ELRA).

Shinsuke Mori, Tetsuro Sasada, Yoko Yamakata, and Koichiro Yoshino. 2012. A machine learning approach to recipe text processing. In *Proceedings of the 1st Workshop on Cooking with Computers*, pages 1–6, Paris, France. Centre national de la recherche scientifique (CNRS).

Jens Nevens, Paul Van Eecke, and Katrien Beuls. 2019. A practical guide to studying emergent communication through grounded language games. In *AISB 2019 Symposium on Language Learning for Artificial Agents*, pages 1–8. AISB.

Taichi Nishimura, Suzushi Tomori, Hayato Hashimoto, Atsushi Hashimoto, Yoko Yamakata, Jun Harashima, Yoshitaka Ushiku, and Shinsuke Mori. 2020. Visual grounding annotation of recipe flow graph. In *Proceedings of the 12th International Conference on Language Resources and Evaluation*, pages 4275–4284, Paris, France. European Language Resources Association (ELRA).

Josef Ruppenhofer and Laura A Michaelis. 2010. A constructional account of genre-based argument omissions. *Constructions and frames*, 2(2):158–184.

Keisuke Shirai, Atsushi Hashimoto, Taichi Nishimura, Hirotaka Kameko, Shuhei Kurita, Yoshitaka Ushiku, and Shinsuke Mori. 2022. Visual recipe flow: A dataset for learning visual state changes of objects with recipe flows. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 3570–3577, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. 2020. ALFRED: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10737–10746, New York, NY, USA. IEEE.

Michael Spranger, Simon Pauw, Martin Loetzsch, and Luc Steels. 2012. Open-ended procedural semantics. In Luc Steels and Manfred Hild, editors, *Language Grounding in Robots*, pages 153–172. Springer, New York, NY, USA.

Luc Steels. 2004. Constructivist development of grounded construction grammar. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pages 9–16.

Luc Steels. 2023. Conceptual foundations for human-centric AI. In *Human-Centered Artificial Intelligence (ACAI 2021)*, pages 8–35, Cham, Switzerland. Springer.

Dan Tasse and Noah A. Smith. 2008. SOUR CREAM: Toward semantic processing of recipes. Technical Report CMU-LTI-08-005, Carnegie Mellon University, Pittsburgh, PA, USA.

Wouter Van den Broeck. 2008. Constraint based compositional semantics. In *Proceedings of the 7th International Conference on the Evolution of Language (EVOLANG7)*, pages 338–345. World Scientific.

Remi van Trijp, Katrien Beuls, and Paul Van Eecke. 2022. The FCG Editor: An innovative environment for engineering computational construction grammars. *PLOS ONE*, 17(6):e0269708.

Lara Verheyen, Jérôme Botoko Ekila, Jens Nevens, Paul Van Eecke, and Katrien Beuls. 2023. Neurosymbolic procedural semantics for reasoning-intensive visual dialogue tasks. In *Proceedings of the 26th European Conference on Artificial Intelligence (ECAI 2023)*, pages 2419–2426, Amsterdam, Netherlands. IOS Press.

Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85.

Terry Winograd. 1972. Understanding natural language. *Cognitive Psychology*, 3(1):1–191.

William A. Woods. 1968. Procedural semantics for a question-answering machine. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, pages 457–471, New York, NY, USA.

Yoko Yamakata, Shinsuke Mori, and John A. Carroll. 2020. English recipe flow graph corpus. In *Proceedings of the 12th International Conference on Language Resources and Evaluation*, pages 5187–5194, Paris, France. European Language Resources Association (ELRA).

Appendix A. Initial Kitchen State

Figure 6 gives a detailed graphical overview of the initial kitchen state. The ingredients can be found in the freezer, the fridge, and the pantry in the specified quantities. The kitchen cabinet contains all cooking utensils.

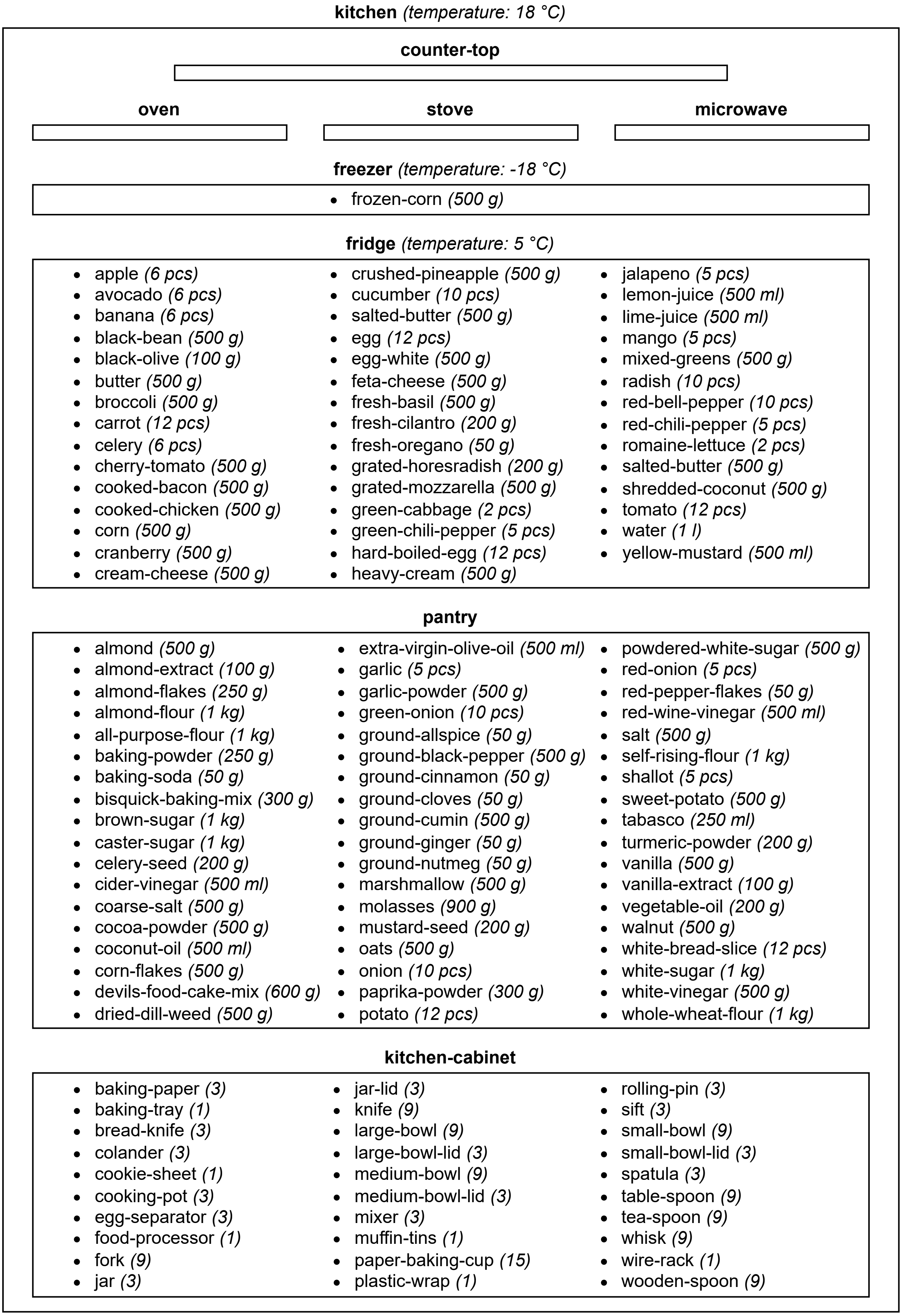
Appendix B. Cooking Ontology

The cooking ontology in YAML format can be found in the supplementary materials accompanying this paper. The cooking ontology underlies the procedural semantic representation language, the kitchen states, and both kitchen simulators. As such, it needs to be consulted from both the qualitative simulation, implemented in Common Lisp using Incremental Recruitment Language (IRL), as well as the quantitative simulation, implemented using PyBullet. The YAML format was chosen for the cooking ontology as it is relatively straightforward to edit by hand and parsers for most mainstream programming languages, including Common Lisp and Python, are available. The cooking ontology defined for this benchmark builds further on the SOMA ontology which in itself extends the DUL ontology.

Appendix C. Recipe Corpus

A data dump of the recipe corpus can be found in the supplementary materials accompanying this paper. Here, we briefly explain the data format of the corpus. We chose to use a consistent and structured XML format to represent the recipes in our benchmark. Each recipe XML file consists of four components. First, a unique identifier for the recipe is included. Second, the recipe title is specified. This could be useful as contextual information since it often states the type of dish that should be prepared. Third, the recipe file lists all ingredients that are necessary to prepare the dish. Finally, the recipe file contains a list of instructions that should be executed in order to prepare the dish. An excerpt of the XML format for the Almond Crescent Cookies recipe is shown in Listing 1.

The gold-standard annotations for all recipes are found in separate files. These files contain the recipe identifier, followed by one primitive operation per line. However, these annotations are only meaningful as a whole due to common recipe idiosyncrasies, such as missing steps, ellipses and contextual references. An excerpt of the gold-standard annotation for the Almond Crescent Cookies recipe is shown in Listing 2.



kitchen *(temperature: 18 °C)*

counter-top

oven

stove

microwave

freezer *(temperature: -18 °C)*

- frozen-corn *(500 g)*

fridge *(temperature: 5 °C)*

- apple *(6 pcs)*
- avocado *(6 pcs)*
- banana *(6 pcs)*
- black-bean *(500 g)*
- black-olive *(100 g)*
- butter *(500 g)*
- broccoli *(500 g)*
- carrot *(12 pcs)*
- celery *(6 pcs)*
- cherry-tomato *(500 g)*
- cooked-bacon *(500 g)*
- cooked-chicken *(500 g)*
- corn *(500 g)*
- cranberry *(500 g)*
- cream-cheese *(500 g)*
- crushed-pineapple *(500 g)*
- cucumber *(10 pcs)*
- salted-butter *(500 g)*
- egg *(12 pcs)*
- egg-white *(500 g)*
- feta-cheese *(500 g)*
- fresh-basil *(500 g)*
- fresh-cilantro *(200 g)*
- fresh-oregano *(50 g)*
- grated-horesradish *(200 g)*
- grated-mozzarella *(500 g)*
- green-cabbage *(2 pcs)*
- green-chili-pepper *(5 pcs)*
- hard-boiled-egg *(12 pcs)*
- heavy-cream *(500 g)*
- jalapeno *(5 pcs)*
- lemon-juice *(500 ml)*
- lime-juice *(500 ml)*
- mango *(5 pcs)*
- mixed-greens *(500 g)*
- radish *(10 pcs)*
- red-bell-pepper *(10 pcs)*
- red-chili-pepper *(5 pcs)*
- romaine-lettuce *(2 pcs)*
- salted-butter *(500 g)*
- shredded-coconut *(500 g)*
- tomato *(12 pcs)*
- water *(1 l)*
- yellow-mustard *(500 ml)*

pantry

- almond *(500 g)*
- almond-extract *(100 g)*
- almond-flakes *(250 g)*
- almond-flour *(1 kg)*
- all-purpose-flour *(1 kg)*
- baking-powder *(250 g)*
- baking-soda *(50 g)*
- bisquick-baking-mix *(300 g)*
- brown-sugar *(1 kg)*
- caster-sugar *(1 kg)*
- celery-seed *(200 g)*
- cider-vinegar *(500 ml)*
- coarse-salt *(500 g)*
- cocoa-powder *(500 g)*
- coconut-oil *(500 ml)*
- corn-flakes *(500 g)*
- devils-food-cake-mix *(600 g)*
- dried-dill-weed *(500 g)*
- extra-virgin-olive-oil *(500 ml)*
- garlic *(5 pcs)*
- garlic-powder *(500 g)*
- green-onion *(10 pcs)*
- ground-allspice *(50 g)*
- ground-black-pepper *(500 g)*
- ground-cinnamon *(50 g)*
- ground-cloves *(50 g)*
- ground-cumin *(500 g)*
- ground-ginger *(50 g)*
- ground-nutmeg *(50 g)*
- marshmallow *(500 g)*
- molasses *(900 g)*
- mustard-seed *(200 g)*
- oats *(500 g)*
- onion *(10 pcs)*
- paprika-powder *(300 g)*
- potato *(12 pcs)*
- powdered-white-sugar *(500 g)*
- red-onion *(5 pcs)*
- red-pepper-flakes *(50 g)*
- red-wine-vinegar *(500 ml)*
- salt *(500 g)*
- self-rising-flour *(1 kg)*
- shallot *(5 pcs)*
- sweet-potato *(500 g)*
- tabasco *(250 ml)*
- turmeric-powder *(200 g)*
- vanilla *(500 g)*
- vanilla-extract *(100 g)*
- vegetable-oil *(200 g)*
- walnut *(500 g)*
- white-bread-slice *(12 pcs)*
- white-sugar *(1 kg)*
- white-vinegar *(500 g)*
- whole-wheat-flour *(1 kg)*

kitchen-cabinet

- baking-paper *(3)*
- baking-tray *(1)*
- bread-knife *(3)*
- colander *(3)*
- cookie-sheet *(1)*
- cooking-pot *(3)*
- egg-separator *(3)*
- food-processor *(1)*
- fork *(9)*
- jar *(3)*
- jar-lid *(3)*
- knife *(9)*
- large-bowl *(9)*
- large-bowl-lid *(3)*
- medium-bowl *(9)*
- medium-bowl-lid *(3)*
- mixer *(3)*
- muffin-tins *(1)*
- paper-baking-cup *(15)*
- plastic-wrap *(1)*
- rolling-pin *(3)*
- sift *(3)*
- small-bowl *(9)*
- small-bowl-lid *(3)*
- spatula *(3)*
- table-spoon *(9)*
- tea-spoon *(9)*
- whisk *(9)*
- wire-rack *(1)*
- wooden-spoon *(9)*

Figure 6: Graphical overview of the initial kitchen environment.

Listing 1: Excerpt of Almond Crescent Cookies recipe file

```
<recipe>
    <id>almond-crescent-cookies</id>
    <title>Almond Crescent Cookies</title>
    <ingredients>
        <ingredient>
            <utterance>
                230 grams butter, room temperature
            </utterance>
        </ingredient>
        ...
    </ingredients>
    <instructions>
        <instruction>
            <utterance>
                Beat the butter and the sugar together until light and
                    fluffy.
            </utterance>
        </instruction>
        ...
    </instructions>
</recipe>
```

Listing 2: Excerpt of Almond Crescent Cookies annotation file

```
#almond-crescent-cookies
(get-kitchen ?kitchen)
(fetch-and-proportion ?proportioned-butter ?ks-with-butter ?kitchen ?
   target-container-1 butter 230 g)
(bring-to-temperature ?warm-butter ?ks-with-warm-butter ?ks-with-
   butter ?proportioned-butter ?room-temp-quantity ?room-temp-unit)
(fetch-and-proportion ?proportioned-sugar ?ks-with-sugar ?ks-with-warm
   -butter ?target-container-2 white-sugar 120 g)
...
```

Appendix D. Cooking Actions

The cooking actions of the procedural semantic representation language, their intended meaning, and simulator-specific implementation choices are listed here in alphabetical order. The number after the name of the cooking action indicates its arity.

bake/9

- Arguments:
  ?baked-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-bake*, *?oven*, *?time-value*, *?time-unit*, *?temperature-value*, *?temperature-unit*

- Intended Meaning:
  Obtain *?baked-thing* by baking *?thing-to-bake* in *?oven* at the temperature specified by *?temperature-value* and *?temperature-unit* for the duration specified by *?time-value* and *?time-unit*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:
  - *?oven* defaults to the closest unused oven in the kitchen
  - *?temperature-value* and *?temperature-unit* default to the temperature of *?oven* (only possible in case *?oven* is specified)

- Constant Arguments:
  - *?time-value* and *?temperature-value* must be numerical values
  - *?time-unit* must be *hour* or *minute*
  - *?temperature-unit* must be *degrees-celsius*

boil/8

- Arguments:
  ?boiled-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-boil*, *?stove*, *?heating-setting*, *?time-value*, *?time-unit*

- Intended Meaning:
  Obtain *?boiled-thing* by boiling *?thing-to-boil* on the *?stove* at the heating setting specified by *?heating-setting* for the duration specified by *?time-value* and *?time-unit*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:

  – *?stove* defaults to the closest unused stove in the kitchen

  – *?heating-setting* defaults to *medium-heat*

  – *?time-value* and *?time-unit* default to 30 minutes

- Constant Arguments:

  – *?heating-setting* must be *low-heat*, *medium-heat*, *medium-high-heat*, *high-heat*

  – *?time-value* and *?temperature-value* must be numerical values

  – *?time-unit* must be *hour* or *minute*

beat/5

- Arguments:
  ?beaten-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-beat*, *?beating-tool*

- Intended Meaning:
  Obtain *?beaten-thing* by beating *?thing-to-beat* using *?beating-tool*. Beating can be seen as a more intense form of mixing which adds some air bubbles during the combination process. This form of mixing also leads to a homogeneous result as individual components are not kept intact. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:

  – *?beating-tool* defaults to the closest unused whisk in the kitchen

bring-to-temperature/6

- Arguments:
  ?thing-at-desired-temperature, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-bring-to-temperature*, *?temperature-value*, *?temperature-unit*

- Intended Meaning:
  Obtain *?thing-at-desired-temperature* at the temperature specified by *?temperature-value* and *?temperature-unit* by waiting for *?thing-to-bring-to-temperature* to cool off or warm up by advancing towards the ambient temperature. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:

  – *?temperature-value* and *?temperature-unit* default to the current room temperature of the kitchen, which is around 18 °C.

- Constant Arguments:

  – *?temperature-value* must be a numerical value

  – *?temperature-unit* must be *degrees-celsius*

cover/5

- Arguments:
  ?covered-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-cover*, *?cover*

- Intended Meaning:
  Obtain *?covered-thing* by covering *?thing-to-cover* with the specified *?cover*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:

  – *?cover* defaults to an appropriate cover for the given *?thing-to-cover*, i.e., a *bowl-lid* for a *bowl*, a *jar-lid* for a *jar* or *plastic-wrap* for anything else.

cut/7

- Arguments:
  ?cut-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-cut*, *?cutting-pattern*, *?cutting-tool*, *?cutting-surface*

- Intended Meaning:
  Obtain *?cut-thing* by using *?cutting-tool* to cut *?thing-to-cut* on the *?cutting-surface* according to the specified *?cutting-pattern*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:

  – *?cutting-tool* defaults to the closest unused knife in the kitchen

  – *?cutting-surface* defaults to the closest unused cutting board in the kitchen

- Constant Arguments:
  - *?cutting-pattern* must be *chopped*, *finely-chopped*, *slices*, *fine-slices*, *squares*, *two-cm-cubes*, *halved*, *shredded*, *minced*, or *diced*

crack/5

- Arguments:
  ?container-with-whole-eggs, *?kitchen-state-out*, *?kitchen-state-in*, *?eggs-to-crack*, *?target-container-for-whole-eggs*
- Intended Meaning:
  Obtain *?container-with-whole-eggs* by cracking *?eggs-to-crack*, i.e., removing the egg shell from *?eggs-to-crack*, and dropping the egg contents in the container specified by *?target-container-for-whole-eggs*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?target-container-for-whole-eggs* defaults to the closest unused medium bowl in the kitchen

dip/5

- Arguments:
  ?dipped-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-dip*, *?dip*
- Intended Meaning:
  Obtain *?dipped-thing* by dipping *?thing-to-dip* into *?dip*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

drain/6

- Arguments:
  ?drained-thing, *?remaining-liquid*, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-drain*, *?draining-tool*
- Intended Meaning:
  Obtain *?drained-thing* by draining *?thing-to-drain* using *?draining-tool* leaving the remaining liquid in *?remaining-liquid*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?draining-tool* defaults to the closest unused colander in the kitchen

fetch/5

- Arguments:
  ?fetched-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-fetch*, *?quantity-to-fetch*
- Intended Meaning:
  Obtain *?fetched-thing* by locating one or more *?thing-to-fetch* objects in the kitchen and bringing it to a common work area such as a kitchen countertop. The exact number of objects to fetch is specified by *?quantity-to-fetch*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Constant Arguments:
  - *?quantity-to-fetch* must be a numerical value
  - *?thing-to-fetch* must be any transferable container or cooking utensil available in the kitchen environment.

fetch-and-proportion/7

- Arguments:
  ?fetched-and-proportioned-ingredient, *?kitchen-state-out*, *?kitchen-state-in*, *?target-container-for-proportioned-ingredient*, *?ingredient-to-fetch-and-proportion*, *?proportion-value*, *?proportion-unit*
- Intended Meaning:
  Obtain an amount of the food product *?fetched-and-proportioned-ingredient* by fetching an *?ingredient-to-fetch-and-proportion*, taking a portion from it specified by *?proportion-value* and *?proportion-unit* and placing this portion inside the container specified by *?target-container-for-proportioned-ingredient*. Ingredient leftovers are returned to their original location. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Constant Arguments:
  - *?ingredient-to-fetch-and-proportion* must be any ingredient that is mentioned in the ingredient list of a supported recipe. Ingredients should be specified by replacing all spaces in an ingredient name with the minus sign (-), e.g., ‘ground black pepper’ would become *ground-black-pepper*.
  - *?proportion-value* must be a numerical value
  - *?proportion-unit* must be *piece*, *g*, *teaspoon*, *tablespoon*, *l* or *ml*

flatten/5

- Arguments:
?flattened-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-flatten*, *?flattening-tool*
- Intended Meaning:
Obtain *?flattened-thing* by flattening *?thing-to-flatten* using *?flattening-tool*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?flattening-tool* defaults to the closest unused rolling pin in the kitchen

flour/5

- Arguments:
?floured-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-flour*, *?flour*
- Intended Meaning:
Obtain *?floured-thing* by flouring *?thing-to-flour* with *?flour*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?flour* defaults to 10 grams of all-purpose flour taken from the closest container with all-purpose flour

fry/8

- Arguments:
?fried-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-fry*, *?stove*, *?heating-setting*, *?time-value*, *?time-unit*
- Intended Meaning:
Obtain *?fried-thing* by frying *?thing-to-fry* on the *?stove* at the heating setting specified by *?heating-setting* for the duration specified by *?time-value* and *?time-unit*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?stove* defaults to the closest unused stove in the kitchen
  - *?heating-setting* defaults to *medium-heat*
  - *?time-value* and *?time-unit* default to 30 minutes
- Constant Arguments:
  - *?heating-setting* must be *low-heat*, *medium-heat*, *medium-high-heat*, *high-heat*
  - *?time-value* and *?temperature-value* must be numerical values
  - *?time-unit* must be *hour* or *minute*

get-kitchen/1

- Arguments:
?initial-kitchen-state
- Intended Meaning:
Obtain the initial state of the kitchen *?initial-kitchen-state*. This is expected to provide access to an environment model of the kitchen to provide contextual information needed for executing a recipe.
- Default Values:
  - *?initial-kitchen-state* defaults to the initial kitchen state in which a recipe will be executed. This argument is expected to be left to its default value in which case this primitive functions as a ‘getter’.

grease/5

- Arguments:
?greased-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-grease*, *?thing-to-grease*, *?grease*
- Intended Meaning:
Obtain *?greased-thing* by greasing *?thing-to-grease* with *?grease*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?grease* defaults to 10 grams of butter taken from the closest container with butter

grind/5

- Arguments:
?ground-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-grind*, *?grinding-tool*
- Intended Meaning:
Obtain *?ground-thing* by grinding *?thing-to-grind* using *?grinding-tool*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?grinding-tool* defaults to the closest unused food-processor in the kitchen

leave-for-time/6

- Arguments:
  ?cooled-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-cool*, *?time-value*, *?time-unit*
- Intended Meaning:
  Obtain *?cooled-thing* by waiting for the duration specified by *?time-value* and *?time-unit* to let *?thing-to-cool* cool off towards the ambient temperature. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Constant Arguments:
  - *?time-value* and *?temperature-value* must be numerical values
  - *?time-unit* must be *hour* or *minute*

line/5

- Arguments:
  ?lined-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-line*, *?lining*
- Intended Meaning:
  Obtain *?lined-thing* by lining *?thing-to-line* with *?lining*, e.g., lining a baking tray with some baking paper or lining muffin tins with paper baking cups. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?lining* defaults to the closest unused sheet of baking paper
- Constant Arguments:
  - *?lining* must be *baking-paper* or *paper-baking-cups*
  - *?thing-to-line* must be *baking-tray*, *cookie-sheet*, *pan* or *muffin-tins*

mash/5

- Arguments:
  ?mashed-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-mash*, *?mashing-tool*
- Intended Meaning:
  Obtain *?mashed-thing* by mashing up *?thing-to-mash* using *?mashing-tool*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?mashing-tool* defaults to the closest unused fork in the kitchen

melt/5

- Arguments:
  ?melted-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-melt*, *?melting-tool*
- Intended Meaning:
  Obtain *?melted-thing* by melting *?thing-to-melt* using *?melting-tool*. This melting tool could be any kind of heating appliance in the kitchen, ranging from a pan on the stove to a microwave. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?melting-tool* defaults to the closest unused microwave in the kitchen

mingle/5

- Arguments:
  ?mingled-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-mingle*, *?mingling-tool*
- Intended Meaning:
  Obtain *?mingled-thing* by mingling *?thing-to-mingle* using *?mingling-tool*. Mingling can be seen as a softer form of mixing in which the individual components are still kept intact during the combination process. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?mingling-tool* defaults to the closest unused wooden spoon in the kitchen

mix/5

- Arguments:
  ?mixed-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-mix*, *?mixing-tool*
- Intended Meaning:
  Obtain *?mixed-thing* by mixing *?thing-to-mix* using *?mixing-tool*. Mixing can be seen as a form of mixing that is intense enough to achieve a homogeneous mixture without being so intense that air bubbles are added during the mixing process. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?mixing-tool* defaults to the closest unused whisk in the kitchen

peel/6

- Arguments:
  ?peeled-thing, *?peel*, *?kitchen-state-out* *?kitchen-state-in*, *?thing-to-peel*, *?peeling-tool*

- Intended Meaning:
  Obtain *?peeled-thing* and its *?peel* by peeling *?thing-to-peel* using *?peeling-tool*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:

  – *?peeling-tool* defaults to the closest unused knife in the kitchen

portion-and-arrange/8

- Arguments:
  ?portions, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-portion*, *?portion-size-value*, *?portion-size-unit*, *?placement-pattern*, *?container-for-portions*

- Intended Meaning:
  Obtain *?portions* by portioning *?thing-to-portion* into portions that each have a size specified by *?portion-size-value* and *?portion-size-unit*. These portions are placed onto the container *?container-for-portions* following the *?placement-pattern*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:

  – *?placement-pattern* defaults to a pattern in which all portions are evenly spread out over the container

  – *?container-for-portions* defaults to the countertop of the kitchen

  – *?portion-size-value* and *?portion-size-unit* default to portion sizes that cause an equal division over the available tins (only possible in case *?container-for-portions* are muffin tins)

- Constant Arguments:

  – *?placement-pattern* must be *side-to-side*, *evenly-spread*, or *5-cm-apart*

preheat-oven/6

- Arguments:
  ?preheated-oven, *?kitchen-state-out*, *?kitchen-state-in*, *?oven*, *?temperature-value*, *?temperature-unit*

- Intended Meaning:
  Obtain *?preheated-oven* by changing the settings of the *?oven* to reach the temperature specified by *?temperature-value* and *?temperature-unit*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:

  – *?oven* defaults to the closest unused oven in the kitchen

- Constant Arguments:

  – *?temperature-value* must be a numerical value

  – *?temperature-unit* must be *degrees-celsius*

refrigerate/7

- Arguments:
  ?refrigerated-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-refrigerate*, *?refrigerator*, *?time-value*, *?time-unit*

- Intended Meaning:
  Obtain *?refrigerated-thing* by putting *?thing-to-refrigerate* inside *?refrigerator* for the duration specified by *?time-value* and *?time-unit*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:

  – *?refrigerator* defaults to the closest unused fridge in the kitchen

  – *?time-value* and *?time-unit* default to one hour

- Constant Arguments:

  – *?time-value* must be a numerical value

  – *?time-unit* must be *minute* or *hour*

seed/6

- Arguments:
  ?seeded-thing, *?seed*, *?kitchen-state-out* *?kitchen-state-in*, *?thing-to-seed*, *?seeded-tool*

- Intended Meaning:
  Obtain *?seeded-thing* and its *?seed* by seeding *?thing-to-seed* using *?seeding-tool*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:
  - *?seeding-tool* defaults to the closest unused knife in the kitchen

separate-eggs/8

- Arguments:
  ?egg-yolks, *egg-whites*, *?kitchen-state-out*, *?kitchen-state-in*, *?eggs*, *?container-for-yolks*, *?container-for-whites*, *?egg-separator*
- Intended Meaning:
  Obtain *?egg-yolks* and *egg-whites* by using an *?egg-separator* to separate separating the whole *?eggs* into the *?container-for-yolks* and *?container-for-whites* respectively. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?container-for-yolks* defaults to the closest unused stove in the kitchen
  - *?container-for-whites* defaults to the closest unused medium bowl in the kitchen (excluding the one found for *?container-for-yolks*)
  - *?egg-separator* defaults the closest unused egg separator in the kitchen

shake/4

- Arguments:
  ?shaken-thing, *?kitchen-state-out* *?kitchen-state-in*, *?thing-to-shake*
- Intended Meaning:
  Obtain *?shaken-thing* by shaking *?thing-to-shake* to mix its contents, which are generally liquids, until a homogeneous mixture is reached. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

shape/5

- Arguments:
  ?shaped-thing, *?kitchen-state-out* *?kitchen-state-in*, *?thing-to-shape*, *?shape*
- Intended Meaning:
  Obtain *?shaped-thing* by shaping *?thing-to-shape* into the shape specified by *?shape*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Constant Arguments:
  - *?shape* must be *ball-shape* or *crescent-shape*

sift/6

- Arguments:
  ?sifted-thing, *?kitchen-state-out* *?kitchen-state-in*, *?container-to-sift-into*, *?thing-to-sift*, *?sift*
- Intended Meaning:
  Obtain *?sifted-thing* by using *?sift* to sift *?thing-to-sift* into the container specified by *?container-to-sift-into*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?container-to-sift-into* defaults to the closest unused large bowl in the kitchen
  - *?sift* defaults to the closest unused sift in the kitchen

spread/6

- Arguments:
  ?thing-with-spread-on, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-spread-on*, *?thing-to-spread*, *?spreading-tool*
- Intended Meaning:
  Obtain *?thing-with-spread* by spreading *?thing-to-spread* on *?thing-to-spread-on* using *spreading-tool*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.
- Default Values:
  - *?spreading-tool* defaults to the closest unused spatula in the kitchen

sprinkle/5

- Arguments:
  ?thing-with-sprinkles-on, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-sprinkle-on*, *?sprinkles*
- Intended Meaning:
  Obtain *?thing-with-sprinkles-on* by sprinkling *?sprinkles* onto *?thing-to-sprinkle-on*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

transfer-contents/8

- Arguments:
  ?container-with-transferred-contents,

?container-with-rest-of-contents, *?kitchen-state-out*, *?kitchen-state-in*, *?container-to-transfer-contents-to*, *?container-with-contents-to-transfer*, *?value-of-transfer-amount*, *?unit-of-transfer-amount*

- Intended Meaning:
  Obtain *?container-with-transferred-contents* and *?container-with-rest-of-contents* by transferring an amount (specified by *?value-of-transfer-amount* and *?unit-of-transfer-amount*) of the container *?container-with-contents-to-transfer*'s contents into *?container-to-transfer-contents-to* leaving the remaining contents in *?container-with-rest-of-contents*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:
  - *?container-to-transfer-contents-to* defaults to the closest unused large bowl in the kitchen
  - *value-of-transfer-amount* and *unit-of-transfer-amount* default to an amount for which all contents are transferred, effectively emptying the original container

- Constant Arguments:
  - *?value-of-transfer-amount* must be a numerical value
  - *?unit-of-transfer-amount* must be *piece*, *g*, *teaspoon*, *tablespoon*, *ml*, or *percent*

transfer-items/6

- Arguments:
  ?transferred-items, *?kitchen-state-out*, *?kitchen-state-in*, *?items-to-transfer*, *?placement-pattern*, *?destination*

- Intended Meaning:
  Obtain *?transferred-items* by carefully transferring all items from *?items-to-transfer* to *?destination* and placing them there according to the pattern specified by *?placement-pattern*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

- Default Values:
  - *?placement-pattern* defaults to a pattern in which the available location is filled up from side to side by creating rows of items one at a time in which items are placed next to each other.

uncover/5

- Arguments:
  ?uncovered-thing, *?cover* *?kitchen-state-out*, *?kitchen-state-in*, *?covered-thing*

- Intended Meaning:
  Obtain *?uncovered-thing* and its prior *?cover* by removing the cover from *?covered-thing*. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.

wash/4

- Arguments:
  ?washed-thing, *?kitchen-state-out*, *?kitchen-state-in*, *?thing-to-wash*

- Intended Meaning:
  Obtain *?washed-thing* by rinsing off or washing *?thing-to-wash* with water. The arguments *?kitchen-state-in* and *?kitchen-state-out* represent the contextual situation before and after execution of this predicate.