Data & Knowledge Engineering 163 (2026) 102558

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

A commonsense reasoning framework for substitution in cooking

Antonis Bikakis [a], Aissatou Diallo [b], Luke Dickens [a], Anthony Hunter [b,*], Rob Miller [a]

[a] Dept of Information Studies, University College London, London, UK
[b] Dept of Computer Science, University College London, London, UK

ARTICLE INFO

Keywords:
Commonsense reasoning
Reasoning about instructions
Reasoning about cooking
Substitution
Repurposing
Inconsistency-directed reasoning

ABSTRACT

The ability to substitute some resource or tool for another is a common and important human ability. For example, in cooking, we often lack an ingredient for a recipe and we solve this problem by finding a substitute ingredient. There are various ways that we may reason about this. Often we need to draw on commonsense reasoning to find a substitute. For instance, we can think of the properties of the missing item, and try to find similar items with similar properties. Despite the importance of substitution in human intelligence, there is a lack of a theoretical understanding of the faculty. To address this shortcoming, we propose a commonsense reasoning framework for conceptualizing and harnessing substitution. In order to ground our proposal, we focus on cooking. Though we believe the proposal can be straightforwardly adapted to other applications that require formalization of substitution. Our approach is to produce a general framework based on distance measures for determining similarity (e.g. between ingredients, or between processing steps), and on identifying inconsistencies between the logical representation of recipes and integrity constraints that we use to flag the need for mitigation (e.g. after substituting one kind of pasta for another in a recipe, we may identify an inconsistency in the cooking time, and this is resolved by updating the cooking time).

1. Introduction

An important human ability is that of substitution: We have a task (doing or making something) for which we lack a resource. We solve the problem by repurposing an alternative to substitute for the missing thing. For example, we could have a recipe for bread that includes the ingredient butter. In fact, olive oil can be used as a substitute. Someone unaware of this may nonetheless be able to work it out by reasoning about the relevant common properties of butter and olive oil. Both are used in baking to add fat, both make a relatively small change to the flavour, and both are neutral in their savoury vs sweet impact. In order to mimic this capability with scalable computational reasoning, we will present a commonsense reasoning framework for substitution for cooking.

Whilst our focus in this paper is on cooking, we see that substitution is an important issue across the gamut of human activities from every day home life (e.g. cooking, gardening, DIY, first-aid, etc.), working life (e.g. farming, manufacturing, etc.), through to crisis management (e.g. dealing with the aftermath of earthquakes).

Within the cooking domain, there are various reasons for why there is a need to substitute ingredients including the following:

Problem of availability Substitution can be required when there is a lack of availability of an ingredient. Perhaps we forgot to buy the item when we last went shopping or perhaps our local shops do not stock the item or perhaps it is out of season. Also, scarcity, arising from fluctuations in supply, price and quality can mean that a planned or desired dish cannot be made

* Corresponding author.
E-mail address: anthony.hunter@ucl.ac.uk (A. Hunter).

https://doi.org/10.1016/j.datak.2026.102558
Received 15 February 2024; Received in revised form 9 December 2025; Accepted 19 January 2026
Available online 22 January 2026

exactly as intended. This either means that you do without the dish, that you make it with an overpriced or poor quality ingredient, or that you substitute that ingredient for another. This last option is such a common scenario that many recipes include notes on reasonable substitutions and some recipe websites have even taken to asking users to propose their own ideas about substitutions as a form of crowdsourcing.

Dietary constraints Another reason to replace one or more ingredients with appropriate substitutes in a dish is due to dietary constraints. The substituting ingredient here must satisfy competing demands: it must retain some properties of the substituted ingredient, omit others, and satisfy the preferences of the consumer. In order to make predictions here a detailed understanding of the properties of ingredients is needed, such as is captured in food knowledge graphs Some constraints will be hard/strict, particularly where they relate to such things as allergies. Other constraints will be softer, for instance, a vegetarian would not want their chips to be deep fried in lard (smoke point 188C), and a suitable substitute may demand a similar smoke point, but a number of substitutes are recommended under these circumstances each with slightly different smoke points, e.g. sunflower oil (smoke point 230C). As this may be a safety critical objective this also has implications about how substitution predictions are integrated with human processes. For instance, a food safety qualified individual may have to mediate between an automated decision-support system and the decision to substitute one ingredient for another. As a consequence, the reasoning/prediction process may need to be explained/interrogated and the underlying knowledge base or model edited/curated.

Environmental impact A third reason for substitution can arise when we take environmental considerations into account. For instance, we may want to substitute an ingredient because it is out of season (e.g. substituting broccoli for asparagus in pasta dishes), or because there is a desire to reduce the environmental impact of production (e.g. substituting bean burgers for meat burgers) or because there is a desire to reduce the environmental impact of transport (e.g. substituting tap water for bottled water).

The ability to successfully predict ingredient substitutions has commercial application too. For instance, a number of home delivery supermarkets now automatically substitute unavailable items from grocery orders (although not always successfully). Ingredient substitution is a challenging problem and isn't just about the properties of ingredients, as Brian O'Driscoll (an Irish rugby player) said: "Knowledge is knowing that a tomato is a fruit. Wisdom is knowing not to put it in a fruit salad".

Proposing substitutions for a given ingredient in a known recipe is a form of recommendation, but with additional complexity to other, more conventional recommender systems, e.g. for films, and this is itself an imperfect science. Film recommender systems typically offer a list of recommendations from which the user can select, and so only require a high chance that the user will find a reasonable suggestion within a list of items that the user can browse through, unlike the online grocery store that has just one opportunity to propose an appropriate substitution.

Even in food recommendation systems with a browsing feature, there are other additional complexities. Importantly, the recommendation for food substitution must not only satisfy the recipient's preferences but also work well in the context of the dish. Having said this, the other ingredients in the dish may provide important information about the user's current preferences. Finally, the use of different ingredients may require different processing requirements. To fully encompass and reason about the implications of a proposed substitution, an account of how this changes the recipe method is also needed. For instance, replacing white flour with brown in bread making can lead to longer rising and cooking times.

So we need to not only find substitutions but also identify mitigations for the recipe to still work. We regard this mitigation as **secondary substitutions**. So when we make a substitution, we need to check whether there is a need for secondary substitutions, and determine what those secondary substitutions are. Our approach to identifying the need for mitigation is to use integrity constraints so that a violation of an integrity constraint denotes the need for one or more secondary substitutions.

In many cases, when we make one or more substitutions, we change the recipe so that the final outcome of the cooking is somewhat different. This occurs even if we undertake secondary substitutions. So we need to consider what we regard as being acceptable for the revised result of the cooking. This in turn depends on the driver (the aim or motivation) for doing a substitution with the following being important types of driver.

Similarity to missing item(s). Here the aim is to find a substitute that is as close as possible in key respects to the missing item. For example, if we lack spaghetti, and we have other pastas in our cupboard including linguine, fusilli, penne, and lasagne, then linguine is the closest to spaghetti. If we lack multiple items, then for each missing item, we consider the nearest item independently of the other items. Continuing the above example, if we lacked spaghetti and sea salt, then we might replace the spaghetti by linguine and sea salt by mineral salt. Note, for this driver, we are not concerned with the effect on the recipe (the other ingredients or cooking actions), nor on the resulting dish. For example, if we are cooking chicken stew, and we lack chicken, the nearest substitute might be tofu, but using tofu might involve substantial changes to the recipe, as the cooking actions will need to be changed, and the resulting dish will differ noticeably from the original. Note, we will tend to discuss the substitution of individual items but this can be generalized to sets of missing items.

Similarity to original dish. Here the aim is to find a revised recipe that results in a dish that is as close as possible in key respects to the original dish. For example, suppose we lack eggs for making a cake, and we have two candidates for a substitute that are yoghurt and flax seeds. In many respects, yoghurt is more similar to eggs than flax seeds are to egg. Yoghurt, like eggs, is a high protein animal product whereas flax seeds are a high fat plant product. So yoghurt would be a better substitute for eggs if similarity to missing item is the driver. In contrast, if similarity to the original dish is the driver, then flax seeds are likely to be a better substitute for eggs.

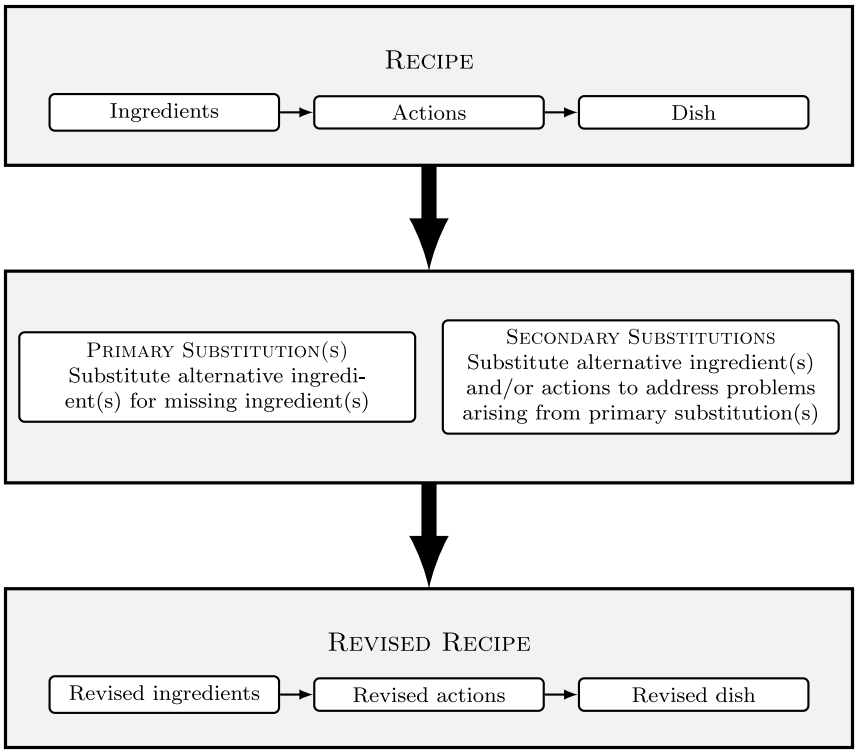

Fig. 1. Summary of four key drivers for substitutions in cooking: **Similarity to missing item(s)** means selecting primary substitutions to minimize the distance between each ingredient, and its corresponding ingredient in the revised recipe; **Similarity to original dish** means selecting substitutions to minimize the distance between the dish and the revised dish; **Minimal secondary substitutions** means after the primary substitutions have been chosen, minimal further changes are made to the ingredients, equipment, and actions; and **Improve specified properties of dish** means selecting substitutions to ensure that the revised dish meets specified properties.

Minimal secondary changes to recipe. Once we have identified the substitutes for the missing item(s), i.e. we have identified the primary substitution(s), we want to minimize the number of secondary substitutions to ingredients and/or the cooking steps. For example, in a recipe for meringue, it is possible to use aquafaba instead of egg white, and it can provide a meringue that is very similar to that obtainable with egg white. However, it involves a number of steps to prepare this as a substitute. So if our driver is to minimize the changes to the recipe, then we may choose an alternative substitute such as an artificial egg powder, which might involve fewer changes to the recipe, even if the resulting dish might be inferior to using aquafaba. As a special case we choose to only minimize the secondary substitutions of ingredients (i.e. we don't mind changing any cooking actions, but we don't want to change ingredients unless necessary) or to only minimize the secondary substitutions of cooking actions (i.e. we don't mind changing any ingredients, but we don't want to change any cooking actions unless necessary).

Improve specified properties of dish. Here the aim is to find a revised recipe that results in a dish that is reasonably close to the original but must fit in a revised category (e.g. change the recipe for chicken casserole so that it fits the category of being vegetarian necessitates replacing chicken as an ingredient), or improves the dish in a specified way (e.g. improved flavour, more spicy, less spicy, less salt, fewer calories, reduced saturated fat, etc.). So the aim is to make minimal substitutions to the recipe in order that the revised recipe satisfied the specified properties.

We summarise the difference between primary and secondary substitutions, and the four types of driver in Fig. 1

Since substitution is an important intelligent activity in cooking, and other domains, we need to better understand the underlying principles. We also need to develop technologies that are able to undertake substitution. For food, this can lead to better ways to use food. This can help us better enjoy food (by allowing us to cook food based on what we have available), reduce food waste (by finding uses for ingredients we have available), improve our health (by replacing less healthy ingredients), and make catering more efficient (by using ingredients that are more cost effective). Furthermore, in order to make these technologies usable, we need to develop technologies that are scalable and robust.

The proposal in this paper provides the first comprehensive framework for computational knowledge representation and reasoning with substitution. This gives us a clearer understanding of the nature of substitution. We have focused on the domain of cooking but the framework can be adapted to other domains where items are prepared through multi-step processes. Furthermore,

with this framework, we have practical technology for automated reasoning with recipes, that can support analysis of recipes, and automated identification of substitutes, where the substitutes are based on the underlying nature of the ingredients, equipment, or processes, within the overall recipe.

Candidates for substituting an ingredient or item of equipment or cooking step, are identified using a distance measure, so that the nearer a candidate is to what is being replaced, the better that candidate. However, such a substitution (a primary substitution) may cause an inconsistency with the integrity constraints and background knowledge that requires mitigation. We address the need for mitigation by secondary substitutions (one or more further substitutions) that restore consistency with the integrity constraints and background knowledge. This then raises different options for regarding a primary substitution acceptable since we may want the closest match for the missing item irrespective of the number and nature of the secondary substitutions, or may prefer to compromise on the primary substitution in order to limit the number or nature of the secondary substitutions.

In order to show how we can undertake substitutions in recipes, we need a formal language for recipes that capture the ingredients, the equipment, and the processes, that are required. We also need this formal language to capture background knowledge and integrity constraints. In this paper, we provide a simple language for this. It is based on classical propositional logic. Variables can be used and the formulae that contain them are treated as schema. Note that, it is not the aim of this paper to provide a comprehensive formalism for representing and reasoning with recipes. Rather, we just give a simple proposal that is sufficient for our needs for introducing the substitution framework. We believe that the substitutions framework presented here would be applicable for a wide range of formalisms for representing and reasoning with recipes including those based on event calculus (see for example [1–3]) or answer set programming (ASP) (see for example [4]). In parallel work, we investigate higher-level formalism for recipes than presented in this paper but which can be directly executed using an ASP solver [5].

We proceed as follows: In Section 2, we provide the definitions for a clausal logic that we will use in the rest of the paper; In Section 3, we provide a framework for representing and reasoning with recipes in classical propositional logic; In Section 4, we provide a framework for candidates for substitution in recipes based on distance functions; In Section 5, we provide a framework for syntactic operations for substitution in cooking sessions; In Section 6, we investigate the stages of substitution in recipes (i.e how a substitution make cause the need for further substitutions); In Section 7, we investigate different drivers for substitution in recipes; In Section 8, we investigate how the need for substitution can be viewed as the resolution of inconsistencies resulting from the violation of integrity constraints; In Section 9, we discuss the related literature; And in Section 10, we discuss our contributions and how they may be developed in further work.

2. Preliminaries

In this section, we provide a logical formalism that in subsequent sections we will use as the basis for representing and reasoning with recipes. For this, we will consider a language based on clauses that we introduce as follows where $\mathcal{C}$ is a set of constant symbols, $\mathcal{F}$ is a set of function symbols, $\mathcal{V}$ is a set of variable symbols, and $\mathcal{P}$ is a set of predicate symbols.

We form **terms** in the usual way: If $\alpha \in \mathcal{C} \cup \mathcal{V}$, then α is term, and if $\alpha \in \mathcal{F}$, and $\beta_1, \ldots, \beta_n$ are terms, then $\alpha(\beta_1, \ldots, \beta_n)$ is a term. For example, $\mathtt{X}$ is a variable, and therefore a term, and $\mathtt{flour}$ and $\mathtt{g}$ are constant symbols, and therefore terms. Using these terms, then $\mathtt{measure}(\mathtt{flour}, \mathtt{X}, \mathtt{g})$ is a term. We also use arithmetic operators such + and − as infix functions. For example, for variables $\mathtt{X}$ and $\mathtt{Y}$, $\mathtt{X} - \mathtt{Y}$ is a term.

If a term contains no variables, then it is a **ground term**. For example, $\mathtt{measure}(\mathtt{flour}, 250, \mathtt{g})$ is a ground term where $\mathtt{flour}$, 250, and $\mathtt{g}$ are constant symbols. Let $\mathcal{T}$ be the set of terms, and let $\mathcal{G}$ be the set of ground terms. We will use the policy that a variable in an example is represented by a string with an upper case first letter (e.g. $\mathtt{T1}$ is a variable symbol whereas $\mathtt{t1}$ is a constant symbol).

We form atoms in the usual way: If $\alpha \in \mathcal{P}$, and $\beta_1, \ldots, \beta_n \in \mathcal{T}$, then $\alpha(\beta_1, \ldots, \beta_n)$ is an atom. For example, $\mathtt{onhand}(\mathtt{count}(\mathtt{eggs}, \mathtt{Y}), \mathtt{T})$ is an atom, where $\mathtt{onhand}$ is a predicate symbol, $\mathtt{count}$ is a function symbol, $\mathtt{eggs}$ is a constant symbol, and $\mathtt{Y}$ and $\mathtt{T}$ are variables with $\mathtt{T}$ denoting a timepoint variable. If an atom contains no variables, then it is a **ground atom**. For example, $\mathtt{onhand}(\mathtt{count}(\mathtt{eggs}, 2), \mathtt{t23})$ is a ground atom where $\mathtt{eggs}$, 2, and $\mathtt{t23}$ are constant symbols, with the latter denoting a timepoint constant.

If δ is an atom, then δ is a **positive literal**, and $\neg\delta$ is a **negative literal**. A **literal** is either a positive literal or a negative literal. Let $\mathcal{A}$ denote the set of positive literals, $\mathcal{N}$ denote the set of negative literals, and let $\mathcal{L}$ denote the set of all literals. We assume that $\bot$ and $\top$ are atomic propositions which we will refer to as falsity and tautology (respectively). So $\neg\top$ is equivalent to $\bot$, and $\neg\bot$ is equivalent to $\top$.

A **clause** is of the form $\delta_1 \leftarrow \delta_2 \wedge \cdots \wedge \delta_n$ where δ_1 is a literal and each δ_i where $i \in \{2, \ldots, n\}$ are literals.

Example 1. If we consume an egg at a timepoint, then we have one less egg at the next timepoint where for a timepoint $\mathtt{T}$, the notation $\mathtt{T} + 1$ denotes the next timepoint (as explained in Section 3.3.

$$\mathtt{onhand}(\mathtt{count}(\mathtt{egg}, 2), \mathtt{T} + 1) \leftarrow \mathtt{onhand}(\mathtt{count}(\mathtt{egg}, 3), \mathtt{T}) \wedge \mathtt{consume}(\mathtt{count}(\mathtt{egg}, 1), \mathtt{T})$$

For a clause ϕ of the form $\delta_1 \leftarrow \delta_2 \wedge \cdots \wedge \delta_n$, let $\mathsf{Head}(\phi) = \psi_1$ and let $\mathsf{Tail}(\phi) = \{\delta_2, \ldots, \delta_n\}$. For a clause ϕ, if $\mathsf{Head}(\phi) = \bot$, then ϕ is an integrity constraint. And for a clause ϕ, if $\mathsf{Tail}(\phi) = \emptyset$, or $\mathsf{Tail}(\phi) = \top$, then ϕ is a literal. So a literal can be treated as special case of a clause.

We assume that the variables in a formula can only be instantiated by a ground term. A **grounding** is a pair (x/y) where x is a variable and y is a ground term. A **ground set** G is a set of groundings. For a formula δ, and a grounding set G, $\mathsf{Ground}(\delta, G)$ is the formula obtained by replacing each variable x by term y for each grounding (x/y) in G. A formula is **ground** iff it contains no variables. The first clause in Example 2 is a clause that is not ground, and the second clause in Example 2 is a ground clause. We generalize the application of a grounding set G to a set of formulae Γ as follows: $\mathsf{Ground}(\Gamma, G) = \{\mathsf{Ground}(\delta, G) \mid \delta \in \Gamma\}$.

Example 2. Consider the following clause δ and grounding set $G = \{(\mathtt{Y}/1000), (\mathtt{X}/500), (\mathtt{T}/\mathtt{t7})\}$. Here X, Y and T are variables and 1000, 500 and t are constants, where T is a variable for a timepoint and t is a specific timepoint.

$$\begin{aligned}&\mathtt{onhand}(\mathtt{measure}(\mathtt{flour}, \mathtt{X}-\mathtt{Y}, \mathtt{g}), \mathtt{T}+1)\\ &\quad\leftarrow \mathtt{onhand}(\mathtt{measure}(\mathtt{flour}, \mathtt{X}, \mathtt{g}), \mathtt{T})\\ &\quad\wedge \mathtt{consume}(\mathtt{measure}(\mathtt{flour}, \mathtt{Y}, \mathtt{g}), \mathtt{T})\end{aligned}$$

The result of $\mathsf{Ground}(\delta, G)$ is the following ground clause.

$$\begin{aligned}&\mathtt{onhand}(\mathtt{measure}(\mathtt{flour}, 1000\text{–}500, \mathtt{g}), \mathtt{t8})\\ &\quad\leftarrow \mathtt{onhand}(\mathtt{measure}(\mathtt{flour}, 1000, \mathtt{g}), \mathtt{t7})\\ &\quad\wedge \mathtt{consume}(\mathtt{measure}(\mathtt{flour}, 500, \mathtt{g}), \mathtt{t7})\end{aligned}$$

In order to reason with a set of literals and clauses, we use the classical propositional consequence relation. To use it, we assume that all the formulae are ground.

3. Representing recipes

Based on the logic we presented in the previous section, we provide a language for representing and reasoning with recipes. We provide a number of examples of recipes in the language. We do not advocate a specific set of terms, atoms, or clauses. Rather, we leave the exact choice of formulae to the user. Different choices can have different effects on the kinds of inferences we can draw from the knowledgebase.

3.1. Modelling dynamics

In this subsection, we outline how we will model dynamics, and we will provide the definitions in the following subsections. For this, we assume a **point-based representation of time**:

- We assume a sequence of **timepoints**. We will assume that the timepoints are in a linear sequence. Each timepoint is associated with a snapshot of the fragment of the world that we are interested in. Furthermore, we will assume we have enough timepoints to be able to describe the evolution of the cooking at a certain level of granularity. We will include in the background knowledge in the knowledgebase to axiomatize this, as explained later). Each timepoint has an associated set of facts that is true at that time point. So we can determine what facts are true or false at each timepoint, and we can see the evaluation of the facts over time.
- In addition to timepoints, we also need the notion of **durations**. A duration is part of the specification of an action. It says how long the action should be undertaken. When we specify an action, it might be assumed to be instantaneous (or at least of negligible time). For example, pouring 500 g of flour into a mixing boil will normally take negligible time. Really, we mean that we do not care how long the action is as it does not affect the outcome of the recipe. For instance, peeling 1 kg of potatoes might take a few minutes, but we can regard it as taking negligible time as the time taken is unimportant for the rest of the recipe. However, for some actions, the duration is important. For example, if we boil spaghetti, then the time is important. If it is too short, then the pasta is too hard, whereas if it is too long, then the pasta becomes mushy. So for each action, we associate a specific duration (e.g. 5 min), or duration range (e.g. 5–10 min). If the latter, then the intention is that the action is carried for a period within that range.

In order to relate timepoints and durations, we need to be able to consider the period of time elapsed between timepoints. This is so that we can represent the period of time that an action is actually undertaken. For instance, we might have the specification that an egg should be boiled for 4 min, and we have starting timepoint (e.g. $\mathtt{t}_{17}$) and ending timepoint e.g. $\mathtt{t}_{65}$ for boiling the egg. We then need to check that the time elapsed between the starting timepoint and the ending timepoint is 4 min (i.e. $\mathtt{t}_{65} - \mathtt{t}_{17}$ is (approximately) 4 min).

3.2. Logical language for recipes

We assume a recipe is executed over time. We will represent this by actions occurring over intervals (represented by starting and ending timepoints), and we will assume that for each action, there may be preconditions that hold before the action occurs, and postconditions that hold after the action has occurred. In the language we introduce below, we will use variables that can be instantiated with specific time points. We will also assume constraints on these variables. These will capture the temporal precedence of the time points.

3.2.1. Logical terms for recipes

We start by defining the types of term that are used to define nine types of atom, and then we use the atoms to define the clauses that are used to specify the processes.

- A **food term** is a term of the form $\alpha(\beta_1, \ldots, \beta_n)$ where α is a function symbol (of arity zero or more) denoting a food item, and $\beta_1, \ldots, \beta_n$ are ground terms denoting qualities of the item e.g. `1cm_pieces` and `lightly_fried`. Some examples of food terms are the following.

```
chopped_onions
chopped_onions(1cm_pieces)
fried_onions(1cm_pieces, lightly_fried)
```

- A **count term** is a term of the form $\texttt{count}(\alpha, \eta)$ where α is a food term, and η is a constant symbol denoting the quantity of the item or η is a variable symbol or η is of the form $\rho - \sigma$ or $\rho + \sigma$ where ρ and σ are variable symbols. We can use other arithmetic operators similarly. Some examples of count terms are the following.

```
count(onions, 2)
count(onions, X − Y)
```

- A **measure term** is a term of the form $\texttt{measure}(\alpha, \eta, \upsilon)$ where α is a food term, and η is a constant symbol denoting the quantity of the item or η is a variable symbol or η is of the form $\rho - \sigma$ or $\rho + \sigma$ where ρ and σ are variable symbols, and υ is the unit. We can use other arithmetic operators similarly. Some examples of measure terms are the following.

```
measure(chopped_onions, 50, g)
measure(chopped_onions, 50–75, g)
measure(chopped_onions(1cm_pieces), 50, g)
measure(fried_onions(1cm_pieces, lightly_fried), 50, g)
measure(flour(wholemeal), X − Y, g)
```

- An **equipment term** is a term of the form $\alpha(\beta_1, \ldots, \beta_n)$ where α is a function symbol (of arity zero or more) denoting an equipment item, and $\beta_1, \ldots, \beta_n$ are constant symbols denoting qualities or quantities of the item. Some examples of equipment terms are the following.

```
pan
bread_maker(500g_capacity)
knife
cheese_grater
```

- A **location term** is a term of the form $\alpha(\beta_1, \ldots, \beta_n)$ where α is a function symbol denoting a location preposition (e.g. `on`, `in`, `over`, etc.), and each β_i is a food term, a count term, an equipment term, or a location term (by recursion). So a location term is a prepositional phrase. Some examples of location terms are the following.

```
on(spaghetti, plate)
on(griddle)
in(oven)
on(plate)
in(pan)
```

- A **modification term** is a ground term that can be used to modify an action, and so it makes the action more specialized in the way it is executed (e.g. for the action `wisk`, the modifier could be `quickly` or `introducing_air`) or in the nature of the result of the action e.g. for the action `chop`, the modifier can be `into(cube)`, or `into(slices)`, or `into(1cm_pieces)`.

```
into(slices)
into(cubes)
into(1cm_pieces)
quickly
introducing_air
until(golden_brown)
until(crispy)
```

- A **duration term** is either a constant that denotes a time duration with a unit (e.g. `60secs`, `5 min`, `1 h`, `3days`, etc.) which we call a **fixed duration**, or an interval (e.g. `5--10 min`, `1--2 h`) which we call a **interval duration**. The informal meaning of an interval duration, say `5–10 min` is that the duration is between 5 min and 10 min.
- An **action term** is a term of the form $\alpha(\beta)$ or of the form $\alpha(\beta, \gamma_1, \ldots, \gamma_n)$ where α is a function symbol denoting an action in the form of a verb, and β is a food term or a count term or a measure term, and each γ_i is a location term or a duration term

or a modification term. Some examples of action terms are the following.

put(tomatoes, in(pan))
fry(chopped_onions)
fry(chopped_onions, 5 min)
fry(chopped_onions, 5 − 10 min)
fry(chopped_onions, in(pan), until(golden_brown))
chop(onions, into(1cm_pieces))

- A **temporal term** is a variable (that can be instantiated with a constant that denotes a timepoint), or is a constant that denotes a timepoint, as illustrated in Example 2.

In this paper, we limit ourselves to these terms, and some subsidiary terms that we will defined later as required. But this choice is only illustrative, and alternatives choices of terms could be used as our framework for substitution framework is agnostic about the choice of terms.

3.2.2. Logical atoms for recipes

Using the terms defined above, we define the following five types of atom where τ (and τ') is a temporal term. In examples, we use the variable symbol T perhaps with a number, e.g. T8, to denote variables for timepoints, and we use the constant symbol t perhaps with number, e.g. t6, to denote specific timepoints.

- A **do atom** is of the form $\texttt{do}(\alpha, \tau_1, \tau_2)$ where α is an action term and τ_1, τ_2 are temporal terms (constant or variable). Some examples of ground do atoms are the following.

do(chop(onion), t1, t1)
do(fry(onion, 5 min), t1, t2)
do(thaw(frozen_chopped_onion, 60 − 120 min), t3, t7)

- An **onhand atom** is of the form $\texttt{onhand}(\alpha, \tau)$ where α is a food or count term or measure term or equipment term and τ is a temporal term. Some examples of onhand atoms are the following.

onhand(chopped_onions, t)
onhand(measure(onion, 50, g), t)
onhand(count(eggs, 2), t)
onhand(measure(flour, 250, g), t)

- A **consume atom** is of the form $\texttt{consume}(\alpha, t)$. where α is a food or count or measure term and τ is a temporal term. Some examples of ground consume atoms are the following.

consume(chopped_onions, t)
consume(count(eggs, 2), t)
consume(measure(flour, 250, g), t)

- A **temporal atom** is the form $\texttt{before}(\tau_1, \tau_2)$ to denote that temporal term τ_1 occurs before temporal term τ_2, and $\texttt{before\_or\_equal}(\tau_1, \tau_2)$ to denote that temporal term τ_1 occurs before or at the same time as temporal term τ_2
- A **period atom** is the form $\texttt{sameperiod}(\delta, \tau_1, \tau_2)$ to denote that the duration term δ is less than or equal to the period from temporal term τ_1 to temporal term τ_2, For example, since the period from 10:46 to 10:51 is less than or equal to 5 min, the following holds.

sameperiod(5 min, 10 : 46, 10 : 51)

We have considered five key types of atom to give us the ability to capture a wide variety of information found in recipes, and thereby give us the ability to consider substitutions. However, this selection of atoms, and the terms that they incorporate, could be substantially revised depending on the needs for representing and reasoning with recipes.

3.2.3. Logical formulae for recipes

Based on the above atoms, we now define the clauses of the language. Recall that a clause is of the form $\delta_1 \leftarrow \delta_2 \wedge \cdots \wedge \delta_n$ where δ_1 is a literal and each δ_i where $i \in \{2, \ldots, n\}$ are literals. We will focus on specific kinds of clause including the following.

- A **preparation clause** which is of the form $\delta_1 \leftarrow \delta_2 \wedge \cdots \wedge \delta_n$ where δ_1 is an onhand atom and each δ_i such that $i \in \{2, \ldots, n\}$ is either a do atom or an onhand atom or a consume atom, and one of the atoms is a do atom.
- A **consumption clause** which is of the form $\delta_1 \leftarrow \delta_2 \wedge \cdots \wedge \delta_n$ where δ_1 is an onhand atom and each δ_i such that $i \in \{2, \ldots, n\}$ is either an onhand atom or a consume atom, and one of the atoms is a consume atom.
- A **ramification clause** which is of the form $\delta_1 \leftarrow \delta_2 \wedge \cdots \wedge \delta_n$ where δ_1 is a negated onhand atom and each δ_i such that $i \in \{2, \ldots, n\}$ is either an onhand atom or a do atom, and one of the atoms is a do atom.
- A **background clause** which is of the form $\delta_1 \leftarrow \delta_2 \wedge \cdots \wedge \delta_n$ where for each δ_i such that $i \in \{1, \ldots, n\}$ is an onhand atom.

- An **integrity constraint** which is of the form $\bot \leftarrow \delta_1 \wedge \cdots \wedge \delta_n$ where each δ_i is an onhand, or do atom or the negation of an onhand, or do atom (i.e. classical negation). Note, we will use $\bot$ to denote contradiction.

We will use the preparation and consumption clauses to specify the processes of recipes, and we will use integrity constraints for checking the consequences of a recipe are consistent. We will use the ramification, persistence, and background clauses to support both tasks. We will use Δ to denote the set of preparation, consumption, ramification, persistence, and background clauses required for a recipe, and Θ to denote a set of integrity constraints, required for a recipe. We provide examples of these clauses in the following.

Example 3 (*Preparation Clause*)**.** This clause has the precondition `onhand(have(onion),T1)`, the action `do(chop(onion),T2)`, and the postcondition `onhand(chopped_onions,T3)`.

```
onhand(chopped_onions,T3)
        ← do(chop(onion),T2,T3)
        ∧ onhand(onion,T1)
        ∧ before(T1,T2) ∧ before(T2,T3)
```

Example 4 (*Preparation Clause*)**.** Like the previous example, the following preparation clause is for the process of chopping onions. However, it is a more specific version.

```
onhand(count(chopped_onions,50,g),T5)
        ← do(chop(onion,into(1cm_pieces)),T3,T4)
        ∧ do(remove(skin,onion),T1,T2)
        ∧ onhand(count(onion,50,g),T0)
        ∧ sequence([T0,T1,T2,T3,T4,T5])
```

Example 5 (*Preparation Clause*)**.** Like the previous examples, the following preparation clause is for the process of chopping onions. However, it provides an alternative way of obtaining the desired result.

```
onhand(chopped_onions,T3)
        ← do(thaw(frozen_chopped_onion,60 − 120 min),T1,T2)
        ∧ onhand(frozen_chopped_onion,T0)
        ∧ before(T0,T1) ∧ before(T1,T2) ∧ before(T2,T3)
```

Example 6 (*Preparation Clause*)**.** This example shows that we can undertake an action for an unspecified period (i.e. there is not an explicit duration), but rather the action continues until another state condition is satisfied.

```
onhand(fried_onions,T4)
        ←  onhand(onion(golden_brown),T3)
        ∧ do(fry(chopped_onion),T2,T3)
        ∧ onhand(chopped_onion,T1)
        ∧ before(T1,T2) ∧ before(T2,T3) ∧ before(T3,T4)
```

We can consider a variant of the above where the atoms `onhand(onion(golden_brown),T3)` and `do(fry(chopped_onion),T2,T3)` are replaced in the above process clause by the following atom that qualifies the period of the frying `do(fry(chopped_onion,until(golden_brown)),T2,T3)`.

Each preparation clause can be viewed as capturing an input–output relationship where the input are the `onhand` atoms in the antecedent of the clause, and the output is the `onhand` atom in the head. So the input and output provide the pre-conditions and post-condition respectively for the process steps in the clause given by the `do` atoms.

Example 7 (*Consumption Clause*)**.** The following consumption clause specifies that if we have N items of type `X` (e.g. eggs) and `M` are consumed, then the number of them that are available is reduced to `N − M`. Note, `T + 1` denotes the next interval after `T`.

```
onhand(count(X,N − M),T + 1)
        ← onhand(count(X,N),T)
        ∧ consume(count(X,M),T)
```

Example 8 (*Consumption Clause*)**.** The following consumption clause specifies that if the quantity of flour by weight consumed is `X`, then the quantity of flour is reduced by `X`.

```
onhand(measure(flour,Y − X,g),T + 1)
        ← onhand(measure(flour,Y,g),T)
        ∧ consume(measure(flour,X,g),T)
```

Example 9 (*Ramification Clause*)**.** The following update clause specifies that if an item is moved from a position, then the original state does not hold.

$$\neg \texttt{onhand(on(X, Y), T + 1)} \leftarrow \texttt{do(move(X, Z), T)} \wedge (\texttt{Z} \neq \texttt{Y}) \wedge \texttt{ onhand(on(X, Y).T)}$$

We can use integrity constraints to ensure that we prohibit models with multiple quantities of available items.

Example 10 (*Integrity Constraint*)**.** In this example, we use integrity constraints to ensure that we do not have multiple numbers of an item X (e.g. eggs).

$$\bot \leftarrow \texttt{onhand(count(X, N), T)} \wedge \texttt{onhand(count(X, M), T)} \wedge \texttt{N} \neq \texttt{M}$$

Example 11 (*Integrity Constraint*)**.** We use integrity constraints to ensure that for a material Z (e.g. flour), there is only one quantity of that material (i.e. X = Y) available at any point in time where is U is the unit for the measure.

$$\bot \leftarrow \texttt{onhand(measure(Z, X, U), T)} \wedge \texttt{onhand(measure(Z, Y, U), T)} \wedge \texttt{X} \neq \texttt{Y}$$

Example 12 (*Integrity Constraint*)**.** We use integrity constraints to ensure the timing of actions are consistent with the time required for the action. For the constraint below, we assume instant noodles only take 2 min to cook and so if a recipe cooks them for a shorter or longer period (e.g. do(boil(instant_noodles, 10 min), T1, T2)), then this is treated as inconsistent (e.g. since 10 min $\neq$ 2 min holds).

$$\bot \leftarrow \texttt{do(boil(instant\_noodles, D), T1, T2)} \wedge \texttt{D} \neq 2\ \texttt{min}$$

Example 13 (*Integrity Constraint*)**.** We use integrity constraints to ensure that an action forces some fluents to no longer hold. If X is in Y at time T, and X is poured onto Z, then X is no longer in Y at time T + 1, and so assuming that it is still in Y at time T + 1 causes an inconsistency.

$$\bot \leftarrow \texttt{onhand(in(X, Y), T + 1)} \wedge \texttt{onhand(in(X, Y), T)} \wedge \texttt{do(pour(X), onto(Z), T)}$$

Example 14 (*Integrity Constraint*)**.** We use integrity constraints to ensure that there is at most one action A undertaken at each point in time.

$$\bot \leftarrow \texttt{do(A, T}_1\texttt{, T}_2\texttt{)} \wedge \texttt{do(A}'\texttt{, T}'_1\texttt{, T}'_2\texttt{)} \wedge \texttt{A} \neq \texttt{A}' \wedge \texttt{before\_or\_equal(T}_1\texttt{, T}'_1\texttt{)} \wedge \texttt{before\_or\_equal(T}'_1\texttt{, T}_2\texttt{)}$$

Recipes often express the same information in different ways. For example, an ingredient may be described by a count (e.g., "one 250 g packet of butter") or by a measure (e.g., "250 g of butter"), and quantities may appear in multiple units. To allow the reasoning process to handle such variations, we include background clauses that encode simple ontological relationships (e.g., alternative names, broader/narrower ingredient types, unit conversions, etc.), so that the system can infer the intended equivalences whenever needed.

Example 15 (*Background Clause*)**.** We use clauses like the following to represent equivalence between a count term and measure term.

$$\texttt{onhand(measure(butter, 250, g), T)} \leftarrow \texttt{onhand(count(250\_g\_packet\_of\_butter, 1), T)}$$

Example 16 (*Background Clause*)**.** We use clauses like the following to represent an equivalence between measures.

$$\texttt{onhand(measure(X, 500, g), T)} \leftarrow \texttt{onhand(measure(X, 0.5, kg), T)}$$

Example 17 (*Background Clause*)**.** We use clauses like the following to represent availability of pan of capacity X if we have a pan of greater capacity.

$$\texttt{onhand(sauce\_pan(capacity(X)), T)} \leftarrow \texttt{onhand(sauce\_pan(capacity(Y)), T)} \wedge \texttt{X} \leq \texttt{Y}$$

For reasoning with time, we assume that before_or_equal is a linear ordering relation over timepoints. So it is reflexive (i.e. for all timepoints τ, before_or_equal(τ, τ) holds), antisymmetric (i.e. for all timepoints τ and τ', if before_or_equal(τ, τ') and before_or_equal(τ', τ) hold, then $\tau = \tau'$), transitive (i.e. for all timepoints τ, τ', and τ'', if before_or_equal(τ, τ') and before_or_equal(τ', τ'') hold, then before_or_equal(τ, τ'') holds), and linearly connected (i.e. for all timepoints τ and τ', before_or_equal(τ, τ') or before_or_equal(τ', τ) holds).

Now we define before in terms of before_or_equal: For all timepoints τ and τ', before(τ, τ') holds iff before_or_equal(τ, τ') holds and before_or_equal(τ', τ) does not hold.

Example 18 (*Background Clause*)**.** We can use the following clauses to reason with time.

$\texttt{before(T1, T3)} \leftarrow \texttt{before(T1, T2)} \wedge \texttt{before(T2, T3)}$
$\texttt{before(T1, T2)} \leftarrow \texttt{sequence([T1, T2, T3, \ldots, T}_\texttt{n}\texttt{])}$
$\texttt{before(T1, T2)} \leftarrow \texttt{before\_or\_equal(T1, T2)} \wedge \neg\texttt{before\_or\_equal(T2, T1)}$

$\texttt{sequence([])}$
$\texttt{sequence([T2, \ldots, Tn])} \leftarrow \texttt{sequence([T1, T2, \ldots, T}_\texttt{n}\texttt{])}$
$\texttt{sequence([T1, T2, \ldots, Tn])} \leftarrow \texttt{before(T1, T2)} \wedge \texttt{sequence([T2, \ldots, T}_\texttt{n}\texttt{])}$

$\texttt{before\_or\_equal(T, T)}$
$\texttt{before\_or\_equal(T1, T2)} \leftarrow \texttt{before(T1, T2)}$
$\texttt{before\_or\_equal(T1, T3)} \leftarrow \texttt{before\_or\_equal(T1, T2)} \wedge \texttt{before\_or\_equal(T2, T3)}$

The examples of clauses presented in this section are meant to be indicative. For specific recipes and for specific reasoning tasks, we may wish to adapt them.

3.3. Logical reasoning with recipes

In order to reason with kinds of clauses described in the previous subsections, we require the definition for the subsequent timepoint for a timepoint: For all timepoints $\tau \in T$, let $\tau + 1$ denote $\tau' \in T$ such that there is no τ'' where τ is before τ'' and τ'' is before τ'. For instance, suppose we have the sequence of timepoints $[\texttt{t1}, \texttt{t7}, \texttt{t9}]$, where $\texttt{t1}$ is before $\texttt{t9}$, $\texttt{t1}$ is not before $\texttt{t7}$ and $\texttt{t7}$ is not before $\texttt{t1}$, then $\texttt{t1} + 1 = \texttt{t9}$.

3.3.1. Cooking sessions

In order to deal with actions and consumables, we introduce the notion of assumptions as follows. These capture the ingredients and actions that we would require in order to use the process rules.

Definition 1. A set of **assumptions** is a set of ground do, onhand, and consume, atoms.

We can select a set of assumed actions for the rules in a knowledgebase. For instance, if $\psi_0 \leftarrow \psi_1 \wedge \cdots \wedge \psi_n$ is in Δ and $\phi \in \{\psi_1, \ldots, \psi_n\}$ and (ϕ is of the form $\texttt{consume}(\alpha, \tau)$ or ϕ is of the form $\texttt{do}(\alpha, \tau, \tau')$ ϕ is of the form $\texttt{onhand}(\alpha, \tau)$).

Example 19. Consider the recipe in Fig. 2. The following is a set of assumptions for ingredients.

$\texttt{onhand(boiling\_water, t1)}$
$\texttt{onhand(spaghetti, t1)}$
$\texttt{onhand(chopped\_onions, t1)}$
$\texttt{onhand(tomatoes, t1)}$
$\texttt{onhand(mixed\_herbs, t1)}$

The following is a set of assumptions for action. Note, in general, we do not prohibit multiple actions at the same timepoint, though later we will discuss how we can use an integrity constraint to ensure that only one action is undertaken at each timepoint.

$\texttt{do(pour(tomato\_sauce, onto(on(spaghetti, plate))), t11)}$
$\texttt{do(pour(spaghetti, plate), t10)}$
$\texttt{do(boil(spaghetti, 10 - 14 min), t3, t9)}$
$\texttt{do(put(spaghetti, in(pan2)), t2)}$
$\texttt{do(pour(boiling\_water, in(pan2)), t1)}$
$\texttt{do(simmer(pan, 5 min), t5, t7)}$
$\texttt{do(put(mixed\_herbs, in(pan)), t4)}$
$\texttt{do(put(tomatoes, in(pan)), t4)}$
$\texttt{do(fry(chopped\_onions, 4 - 5 min), t2, t3)}$
$\texttt{do(put(chopped\_onions, in(pan)), t1)}$

We also require the assumption $\texttt{sequence([t1, t2, t3, t4, t5, t6, t7, t8, t9, t10])}$ for timepoints.

We now define a cooking session as recipe (i.e. a set of preparation, consumption, ramification, persistence, and background clauses), a set of assumptions, a set of integrity constraints, and an atom (that is intended to denote a food item).

Definition 2. A **cooking session** is a tuple (Δ, Γ, Θ) where Δ is a recipe (i.e. a set of preparation, consumption, ramification, persistence, and background clauses), Γ is a set of assumptions (i.e. availability of specific set of ingredients, equipment, and actions), Θ is a set of integrity constraints and background knowledge (i.e. background clauses).

Example 20. Let Δ be the four preparation clauses in Fig. 2, let Γ be the assumptions given in Example 19, and let $\Theta = \emptyset$.

```
onhand(on(spaghetti_pomodoro,plate),T3)
      ← do(pour(tomato_sauce,on(spaghetti,plate)),T2,T2)
      ∧ onhand(tomato_sauce,T1)
      ∧ onhand(on(spaghetti,plate),T1)
      ∧ before(T1,T2) ∧ before(T2,T3)

onhand(on(spaghetti,plate),T3)
      ← do(pour(cooked(spaghetti),plate),T2,T2)
      ∧ onhand(cooked_spaghetti,T1)
      ∧ before(T1,T2) ∧ before(T2,T3)

onhand(cooked(spaghetti),T5)
      ← onhand(softness(spaghetti,al_dente),T4)
      ∧ do(boil(spaghetti,10-14min),T3,T4)
      ∧ do(put(spaghetti,in(pan2)),T2,T2)
      ∧ do(pour(boiling_water,in(pan2)),T1,T1)
      ∧ onhand(boiling_water,T1)
      ∧ onhand(spaghetti,T1)
      ∧ sequence([T1,T2,T3,T4,T5])

onhand(tomato_sauce,T8)
      ← do(simmer(mix,in(pan),5min),T6,T7)
      ∧ do(put(mixed_herbs,in(pan)),T5,T5)
      ∧ do(put(tomatoes,in(pan)),T4,T4)
      ∧ do(fry(chopped_onions,4-5min),T2,T3)
      ∧ do(put(chopped_onions,in(pan)),T1,T1)
      ∧ onhand(chopped_onions,T1)
      ∧ onhand(tomatoes,T1)
      ∧ onhand(mixed_herbs,T1)
      ∧ sequence([T1,T2,T3,T4,T5,T6,T7,T8])
```

Fig. 2. A set of preparation clauses for preparing a simple pasta dish. The recipe is split into four rules.

So for a cooking session (Δ, Γ, Θ), Δ provides the processing steps for the recipe, Γ provides the assumed ingredients and actions, and Θ provides constraints to check that these work in an acceptable way and background knowledge that may be required to support reasoning with the recipe. We separate Δ from Θ because later we will want to update Δ with substitutions for ingredients, equipment, and processing steps, but leave Θ unchanged.

Later we will consider how a cooking session can be considered with respect to a specific food item π. So if π, say $\texttt{onhand(baked(victoria\_sponge),t)}$ is the intended result of a recipe at time $\texttt{t}$, then we can use logical consequence as defined in the next subsection to show that π follows from a cooking session.

3.3.2. Consequence relation

To reason with a recipe, we use the classical propositional logic consequence relation $\vdash$. For a cooking session (Δ, Γ, Θ), a ground literal α, and a grounding set G, we denote that α is a classical propositional inference from $\mathsf{Ground}(\Delta \cup \Gamma \cup \Theta, G)$ by $(\Delta, \Gamma, \Theta) \vdash \phi$. Note, we could equivalently treat all formulae with variables as being universally quantified formulae where each variable is in the scope of a universal quantifier. Then we could use first-order predicate logic for determining whether α is an inference. However, using grounding offers a simple solution for implementation since we can use automated reasoning such as SAT solvers with the propositional logic formulae.

Example 21. Consider the following clause.

```
onhand(cooked(boiled_egg),T2)
      ← do(boil(egg),T1)
      ∧ consume(count(egg,1),T1)
      ∧ onhand(count(egg,1),T1)
      ∧ before(T1,T2)
```

Suppose Δ contains the above clause and Γ contains the following atoms.

```
onhand(count(egg,1),t1)
consume(count(egg,1),t1)
do(boil(egg),t1)
```

Then we obtain inferences that include the following:

$(\Delta, \Gamma, \Theta) \vdash \texttt{onhand(count(egg, 1), t1)}$
$(\Delta, \Gamma, \Theta) \vdash \texttt{consume(count(egg, 1), t1)}$
$(\Delta, \Gamma, \Theta) \vdash \texttt{do(boil(egg), t1)}$
$(\Delta, \Gamma, \Theta) \vdash \texttt{onhand(cooked(boiled\_egg), t2)}$

If we also assume the clause in Example 7, then we also have the following inference.

$(\Delta, \Gamma, \Theta) \vdash \texttt{onhand(count(egg, 0), t2)}$

There are many ways that we can specify an inconsistent set of formulae using this language. The following is an example based on an integrity constraint.

Example 22. The atoms $\texttt{onhand(count(egg, 1), T)}$ and $\texttt{onhand(count(egg, 6), T)}$ are inconsistent with the integrity constraint. So if Γ contains the above atoms, and Θ contains the following integrity constraint, we obtain $(\Delta, \Gamma, \Theta) \vdash \bot$.

$$\bot \leftarrow \texttt{onhand(count(egg, 6), T)} \wedge \texttt{onhand(count(egg, 1), T)}$$

In order to ensure the consumption of the same resource at the same time is blocked, we can use integrity constraints of the following form.

Example 23. The atoms $\texttt{do(use(count(egg, 1)), t}_1\texttt{)}$ and $\texttt{do(use(count(egg, 6)), t}_2\texttt{)}$ are inconsistent with the following integrity constraint. So if Γ contains the above atoms, and Θ contains the following integrity constraint, we obtain $(\Delta, \Gamma, \Theta) \vdash \bot$.

$$\bot \leftarrow \texttt{consume(count(X, N), T)} \wedge \texttt{consume(count(X, M), T)} \wedge \texttt{N} \neq \texttt{M}$$

We can also use integrity constraints to ensure that the consumption of resources sums to not more than the original availability of the resource in the assumptions.

Example 24. The following integrity constraint is for the case when there are two assumptions that a resource is consumed, and there is an inconsistency if the sum of these is greater than the available resource.

$$\begin{aligned}\bot \leftarrow\ & \texttt{consume(count(X, N1), T2)} \\ & \wedge\ \texttt{consume(count(X, N2), T1)} \\ & \wedge\ \texttt{onhand(count(X, M), T0)} \\ & \wedge\ \texttt{before(T0, T1)} \wedge \texttt{before(T0, T2)} \\ & \wedge\ \texttt{N1} + \texttt{N2} > \texttt{M}\end{aligned}$$

We can further restrict the consumption of items by introducing an integrity constraint that blocks the consumption of the same item in multiple actions in multiple locations at the same time.

To reason with a recipe, we can execute it. In other words, we can simulate the sequence of steps involved in using the recipe. The ingredients gives the details of the starting state, and then the preparation clauses give us the details of the subsequent states.

Definition 3. Let Δ be a set of clauses, and Γ be a set of assumptions. The **execution function**, denoted Execute, is defined as follows.

$$\textsf{Execute}(\Delta, \Gamma, \Theta) = \{\phi \mid (\Delta, \Gamma, \Theta) \vdash \phi\}$$

Example 25. Consider Δ containing the following clause

$$\begin{aligned}&\texttt{onhand(hard\_boiled\_egg, t6)} \\ &\quad \leftarrow\ \texttt{do(put(egg, in(egg\_cup)), t5)} \\ &\quad \wedge\ \texttt{do(boil(water, in(pan), 3 - 5 min), t3, t4)} \\ &\quad \wedge\ \texttt{do(pour(water, into(pan)), t2)} \\ &\quad \wedge\ \texttt{do(put(egg, in(pan)), t1)} \\ &\quad \wedge\ \texttt{onhand(egg, t1)} \\ &\quad \wedge\ \texttt{onhand(measure(water, 500, ml), t1)} \\ &\quad \wedge\ \texttt{sequence([t1, t2, t3, t4, t5])}\end{aligned}$$

plus the following assumptions Γ.

$\texttt{do(put(egg, in(egg\_cup)), t5)}$	$\texttt{do(boil(water, in(pan), 3 - 5 min), t3, t4)}$
$\texttt{do(pour(water, into(pan)), t2)}$	$\texttt{do(put(egg, in(pan)), t1)}$
$\texttt{onhand(egg, t1)}$	$\texttt{onhand(measure(water, 500, ml), t1)}$

From this, we have the execution $\textsf{Execute}(\Delta, \Gamma, \Theta) = \Gamma \cup \{\texttt{onhand(hard\_boiled\_egg, t6)}\}$.

The output of a cooking session is defined as the items onhand that are not assumed (i.e. not ingredients) and that are available at the last point in time.

Definition 4. For a cooking session (Δ, Γ, Θ), the **dishes** function is defined as follows.

$$\begin{aligned}\mathsf{Dishes}(\Delta, \Gamma, \Theta) = \{\alpha \mid \mathtt{onhand}(\alpha, t) \in \mathsf{Execute}(\Delta, \Gamma, \Theta) \\ \text{and for all } \mathtt{onhand}(\alpha', t') \in \mathsf{Execute}(\Delta, \Gamma, \Theta), t \geq t'\}\end{aligned}$$

Example 26. Consider the cooking session in Example 25. From this, we have $\mathtt{hard\_boiled\_egg} \in \mathsf{Dishes}(\Delta, \Gamma, \Theta)$.

When we have a cooking session (Δ, Γ, Θ), there is an intended output π and we would want that to be in $\mathsf{Dishes}(\Delta, \Gamma, \Theta)$. As we will investigate later, if π is not in $\mathsf{Dishes}(\Delta, \Gamma, \Theta)$, then we may seek a π' in $\mathsf{Dishes}(\Delta, \Gamma, \Theta)$ that is similar to π.

4. Candidates for substitution

We are interested in substituting one ingredient for another. For example, swapping an onion for a shallot in a recipe for a sauce. However, we are also interested in substituting intermediate food items in a recipe such as chopped shallots for chopped onions. Furthermore, we are interested in substituting equipment and processes. For example, instead of the process of kneading flour, water, and yeast, for 10 min to make dough for bread, we can mix the ingredients in a bowl, and leave it overnight.

We use a distance measure to compare ingredients. So for ingredients α and β, $d(\alpha, \beta)$ denotes the distance between α and β. The closer the distance $d(\alpha, \beta)$, the better one would be a substitute for another. So $d(\alpha, \beta) = 0$ means α and β would be perfect substitutes for each other. Since it is a distance measure, it is always the case that $d(\alpha, \alpha) = 0$, and so any ingredient is a perfect substitute for itself.

We are also interested in substitutions of equipment, of actions, and of chunks of recipes. These means that we need to not only substitute logical terms but also clauses. So we need to also consider distance between formulae ϕ and ψ, and hence consider distance of the form $d(\phi, \psi)$ which denotes the distance between ϕ and ψ.

There are various ways we can define a distance measure for finding substitutions. We give some options in the rest of this section.

4.1. Distance measures based on role

Given a set of recipes Δ, we can group them in (overlapping subsets) by **recipe type** (e.g. `breads`, `cakes`, `biscuits`, `baked goods`, `roasts`, `pasta dishes`, `stews`, `soups`, `etc.`). These are based on the nature of the final food item produced by the recipe.

Each ingredient in recipe has a role. The are many ways that roles can be described (see for example www.foodafactoflife.org.uk) Some examples of roles are `thicken`, `bind`, `flavour`, `shorten`, `sweeten`, `make_flaky_pastry`, `glaze`, `denature`, `retain_moisture`, `aerate`, `gel` and `raise`.

A **role function** ρ_t, for a recipe type t, takes an ingredient, and returns the roles of the ingredient in the recipe type. The following are some examples of role functions.

- $\rho_{\mathtt{baked\_goods}}(\mathtt{butter}) = \{\mathtt{binds(flour)}, \mathtt{flavours(buttery)}\}$
- $\rho_{\mathtt{baked\_goods}}(\mathtt{milk}) = \{\mathtt{binds(flour)}, \mathtt{flavours(buttery)}, \mathtt{flavours(milky)}\}$
- $\rho_{\mathtt{baked\_goods}}(\mathtt{honey}) = \{\mathtt{sweetens}, \mathtt{flavours(honey)}\}$
- $\rho_{\mathtt{fish\_cake}}(\mathtt{egg}) = \{\mathtt{binds(fish\_and\_potato)}\}$

We can use the role function to define a role distance function which is based on Jaccard distance. Essentially, the more properties that are in common, the closer the two ingredients.

Definition 5. For food items α and β, the **role distance function**, denoted $d_t(\alpha, \beta)$, is defined for recipe type t

$$d_t(\alpha, \beta) = 1 - \frac{|\rho_t(\alpha) \cap \rho_t(\beta)|}{|\rho_t(\alpha) \cup \rho_t(\beta)|}$$

Example 27. For the following specifications of role, $d_{\mathtt{baked\_good}}(\mathtt{butter}, \mathtt{milk}) = 2/3$

$$\begin{aligned}&\rho_{\mathtt{baked\_goods}}(\mathtt{butter}) = \{\mathtt{binds(flour)}, \mathtt{flavours(buttery)}\}\\ &\rho_{\mathtt{baked\_goods}}(\mathtt{milk}) = \{\mathtt{binds(flour)}, \mathtt{flavours(buttery)}, \mathtt{flavours(milky)}\}\end{aligned}$$

We can use NLP methods to extract role functions from recipes and other text sources. Recipe types can be learned by using hierarchical clustering and/or obtained from ontologies. And/or assignment of types to a recipe can be undertaken with ML classification methods.

4.2. Distance measures based on word embeddings

A distance measure can be defined based on a word embedding such as the general purpose word embeddings Word2Vec [6] or Glove [7], or the specialized word embedding Food2Vec which is a pre-trained word embeddings for cooking substitution [8]. For food items α and β, and a word embedding, the distance function $d(\alpha, \beta)$ is the cosine similarity between α and β in the word embedding.

Word embeddings for food could be improved by considering the following options: (**Analogical reasoning with word embeddings**) For example, determine vector for `choppedshallot` from vectors for `chopped`, and `shallot`, using for instance methods for combining word-embeddings for individual words, to give the word-embedding for a compound (for a review, see [9,10]); (**Word embeddings trained by recipe type**) For example, train the word embedding only with recipes for baked goods, and so ingredients that are near each other are more likely to be substitutes which can be represented as $d_{\texttt{baked\_goods}}(\texttt{olive\_oil}, \texttt{butter})$; (**Word embeddings trained by role(s)**) Train the word embedding to minimize the distance between two ingredients for a specific role, and so for example $d^{\texttt{binding}}_{\texttt{baked\_goods}}(\texttt{olive\_oil}, \texttt{butter})$ is low whereas $d^{\texttt{binding}}_{\texttt{baked\_goods}}(\texttt{egg}, \texttt{cheese})$ is high. We can also consider combining distance by reducing each distance to unit interval and then using t-norms For more information on developing distance measures in the food domain see [11].

4.3. Distance measures based on operational knowledge

We can specify distance measures from knowledge of cooking and/or we can identify common substitutions for ingredients or steps in recipes using text mining. For example,

- 1 cup of buttermilk can be substituted by 1 cup yoghurt OR 1 tablespoon lemon juice or vinegar plus enough milk to make 1 cup
- 1 cup of mayonnaise can be substituted by 1 cup sour cream OR 1 cup plain yoghurt
- 1 whole egg can be substituted by half a banana mashed with 1/2 teaspoon baking powder

There are numerous resources on options for substitutions (e.g. substitutions to transfer a dish into vegan dish [12]) that can be used as the basis of specifying distance measures. Alternatively, text mining can be used to identify substitutions that are commonly used in recipes online.

Example 28. Suppose we have the ingredient $\phi = \texttt{onhand}(\texttt{measure}(\texttt{chopped\_onions}, 100\ \texttt{g}), \texttt{T})$, and we have the following clause ψ, then we can specify $d(\phi, \psi)$ to be low.

$$\begin{aligned}&\texttt{onhand}(\texttt{chopped\_onions}, \texttt{T4})\\ &\quad \leftarrow \texttt{do}(\texttt{chop}(\texttt{onion}), \texttt{T2}, \texttt{T3})\\ &\quad \wedge\ \texttt{onhand}(\texttt{onion}, \texttt{T1})\\ &\quad \wedge\ \texttt{before}(\texttt{T1}, \texttt{T2}) \wedge \texttt{before}(\texttt{T2}, \texttt{T3})\end{aligned}$$

Example 29. Suppose we need the ingredient $\phi = \texttt{onhand}(\texttt{maple\_syrup}, \texttt{T})$, and we have the following clause ψ, then we can specify $d(\phi, \psi)$ to be low.

$$\begin{aligned}&\texttt{onhand}(\texttt{maple\_syrup}, \texttt{T8})\\ &\quad \leftarrow\ \texttt{do}(\texttt{boil}(\texttt{pan}, 5\ \texttt{min}), \texttt{T6}, \texttt{T7})\\ &\quad \wedge\ \texttt{do}(\texttt{put}(\texttt{measure}(\texttt{vanilla\_extract}, 10, \texttt{g}).\texttt{in}(\texttt{pan})), \texttt{T5})\\ &\quad \wedge\ \texttt{do}(\texttt{put}(\texttt{measure}(\texttt{water}, 200, \texttt{ml}).\texttt{in}(\texttt{pan})), \texttt{T4})\\ &\quad \wedge\ \texttt{do}(\texttt{put}(\texttt{measure}(\texttt{white\_sugar}, 100, \texttt{g}).\texttt{in}(\texttt{pan})), \texttt{T3})\\ &\quad \wedge\ \texttt{do}(\texttt{put}(\texttt{measure}(\texttt{brown\_sugar}, 200, \texttt{g}).\texttt{in}(\texttt{pan})), \texttt{T2})\\ &\quad \wedge\ \texttt{do}(\texttt{put}(\texttt{measure}(\texttt{butter}, 20, \texttt{g}), \texttt{in}(\texttt{pan})), \texttt{T1})\\ &\quad \wedge\ \texttt{sequence}([\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}, \texttt{T6}, \texttt{T7}, \texttt{T8}])\end{aligned}$$

We can also specify a distance measure between sets of atoms or sets of clauses as we illustrate in the next example.

Example 30. If Φ is the set of literals $\{\texttt{onhand}(\texttt{measure}(\texttt{egg\_white}, 100, \texttt{ml}), \texttt{T1})\}$ and Ψ is the following set of literals then we can specify $d(\Phi, \Psi)$ to be low.

$$\{\texttt{do}(\texttt{whip}(\texttt{aquafaba}, 10\ \texttt{min}), \texttt{T1}), \texttt{onhand}(\texttt{measure}(\texttt{aquafaba}, 100, \texttt{ml}), \texttt{T1})\}$$

It is also possible to construct distance measures from knowledge graphs of food (such as FoodOn) based on ranking of ingredients using a range of scoring functions [13].

We explain how we can use these different kinds of distance measure in substitution in the next section.

5. Substitution operations

As we have discussed in previous sections, a recipe is represented by a set of assumed ingredients and equipment Γ, and a set of clauses that capture the processes for producing the intended food item to be output Δ, together with clauses for background knowledge and integrity constraints Θ. Note, we separate the integrity constraints and background knowledge from the process clauses because we do not want to make substitutions into the constraints and background, rather we want to focus substitutions on the process clauses and assumptions,

There are **three levels of substitution** that we can make as follows. We explain them informally here, and then formalize them in the subsequent subsections.

Term-level substitution Here we substitute a term in a formula or set of formulae. For example, we can swap `count(apple, 3)` for `count(pear, 3)` in a recipe for a cake. We use term substitution to replace specific ingredients and equipment in process clauses and assumptions, and to amend specific actions in process clauses (for example, we can swap the term `boil(fusilli, 12 min)` for the term `boil(fusilli, 10 min)` in the condition of a process clause `do(boil(fusilli, 10 min), T)`),

Condition-level substitution Here we substitute an atom or conjunction of atoms in a formula or set of formulae. For example, we can swap the atom `do(boil(fusilli, 12 min), T)` for the atom `do(boil(fusilli, 10 min), T)` in a recipe for a pasta dish. We can generalize to a conjunction of atoms $\phi_1 \wedge \cdots \wedge \phi_m$, and $\phi'_1 \wedge \cdots \wedge \phi'_n$, and so we can replace $\phi_1 \wedge \cdots \wedge \phi_m$ by $\phi'_1 \wedge \cdots \wedge \phi'_n$.

Clause-level substitution Here we substitute a set of clauses for a set of clauses. This allows for more substantial changes to a recipe that allow steps of a recipe to be added or removed.

Note, there is some overlap in what these types of substitution can do, depending on the clauses in recipe. So it may be the case that replacing a term or replacing an atom have the same effect. For instance, it may be the case that swapping in term `boil(fusilli, 12 min)` or swapping in the atom `do(boil(fusilli, 12 min), T)` have the same effect.

We formalize these three levels of substitution (i.e. term substitution, condition substitution, and clause substitution) in the following subsections.

5.1. Term substitution

We will start with a distance measure d, and a threshold τ, a **substitution candidate**, denoted $[\alpha/\beta]$, holds for α and β (where α and β are food items, or equipment, or processes) iff we have $d(\alpha, \beta) \leq \tau$.

Definition 6. For a term, literal or clause ϕ, the **term substitution** of term α by term β, is $\phi[\alpha/\beta]$

- If ϕ is a ground term, and $\phi = \alpha$, then $\phi[\alpha/\beta]$ is β
- If ϕ is a constant symbol, $\phi \neq \alpha$, then $\phi[\alpha/\beta]$ is ϕ
- If ϕ is a term or atom of the form $\alpha(\gamma_1, \ldots, \gamma_n)$, then $\phi[\alpha/\beta]$ is $\beta(\gamma_1[\alpha/\beta], \ldots, \gamma_n[\alpha/\beta])$
- If ϕ is $\neg\psi$, then $\phi[\alpha/\beta]$ is $\neg(\psi[\alpha/\beta])$
- If ϕ is $\psi_1 \leftarrow \psi_2 \wedge \cdots \wedge \psi_n$, then $\phi[\alpha/\beta]$ is $\psi_1[\alpha/\beta] \leftarrow \psi_2[\alpha/\beta] \wedge \cdots \wedge \psi_n[\alpha/\beta]$

For a set of formulae $\Xi = \{\psi_1, \ldots, \psi_n\}$, $\Xi[\alpha/\beta]$ is $\{\psi_1[\alpha/\beta], \ldots, \psi_n[\alpha/\beta]\}$

In the following example, we swap an ingredient that is represented by a constant symbol. So we replace the type of ingredient, but not the quantity required.

Example 31. Revisiting the previous example, consider the following substitution of an atom with the substitution candidate `[butter/margarine]`.

$$\texttt{onhand(measure(butter, 20, g), t1)[butter/margarine]} = \texttt{onhand(measure(margarine, 20, g), t1)}$$

In the example in Fig. 3, we consider substituting an ingredient that is represented by a ground term, and we replace both the type of food and the quantity required.

We can handle equipment in the same way as food items. So we can use distance both for types of equipment and capacities of equipment.

Example 32. We can define a distance measure, so that for instance `sauce_pan(500 ml)` is close to `sauce_pan(600 ml)`, and `sauce_pan(500 ml)` is close to `frying_pan(500 ml)`, but `sauce_pan(500 ml)` is not close to `sauce_pan(1000 ml)` since if we require a 1000 ml pan for a recipe, a 500 ml pan may be too small.

The following are some basic properties that one may consider holding for a notion of substitution. They may appear to be desirable for our substitution operator, but as we show in the following proposition, only the first holds. The others do not hold for term substitution because when we substitute, we can in a sense lose distinctions between items, and these are irreversible.

Consider the following preparation clause ϕ.

$$\begin{aligned}
&\texttt{onhand}(\texttt{measure}(\texttt{bread}, 500, \texttt{g}), \texttt{t7}) \\
&\quad \leftarrow \texttt{do}(\texttt{switch_on}(\texttt{bread_maker}, \texttt{programme_3}), \texttt{t6}) \\
&\quad \wedge \texttt{do}(\texttt{put}(\texttt{water}, \texttt{in}(\texttt{bread_maker})), \texttt{t5}) \\
&\quad \wedge \texttt{do}(\texttt{put}(\texttt{flour}, \texttt{in}(\texttt{bread_maker})), \texttt{t4}) \\
&\quad \wedge \texttt{do}(\texttt{put}(\texttt{yeast}, \texttt{in}(\texttt{bread_maker})), \texttt{t3}) \\
&\quad \wedge \texttt{do}(\texttt{put}(\texttt{rape_seed_oil}, \texttt{in}(\texttt{bread_maker})), \texttt{t2}) \\
&\quad \wedge \texttt{consume}(\texttt{measure}(\texttt{water}, 330, \texttt{ml})), \texttt{t1}) \\
&\quad \wedge \texttt{consume}(\texttt{measure}(\texttt{flour}, 500, \texttt{g})), \texttt{t1}) \\
&\quad \wedge \texttt{consume}(\texttt{measure}(\texttt{yeast}, 5, \texttt{g})), \texttt{t1}) \\
&\quad \wedge \texttt{consume}(\texttt{measure}(\texttt{butter}, 30, \texttt{g})), \texttt{t1}) \\
&\quad \wedge \texttt{onhand}(\texttt{measure}(\texttt{water}, 330, \texttt{ml})), \texttt{t1}) \\
&\quad \wedge \texttt{onhand}(\texttt{measure}(\texttt{flour}, 500, \texttt{g})), \texttt{t1}) \\
&\quad \wedge \texttt{onhand}(\texttt{measure}(\texttt{yeast}, 5, \texttt{g})), \texttt{t1}) \\
&\quad \wedge \texttt{onhand}(\texttt{measure}(\texttt{butter}, 30, \texttt{g})), \texttt{t1}) \\
&\quad \wedge \texttt{sequence}([\texttt{t1}, \texttt{t2}, \texttt{t3}, \texttt{t4}, \texttt{t5}, \texttt{t6}, \texttt{t7}])
\end{aligned}$$

So $\phi[\alpha/\beta]$ results in the following clause where $\alpha = \texttt{measure}(\texttt{butter}, 30, \texttt{g})$ and $\beta = \texttt{measure}(\texttt{rape\_seed\_oil}, 25, \texttt{ml})$.

$$\begin{aligned}
&\texttt{onhand}(\texttt{measure}(\texttt{bread}, 500, \texttt{g}), \texttt{t7}) \\
&\quad \leftarrow \texttt{do}(\texttt{switch_on}(\texttt{bread_maker}, \texttt{programme_3}), \texttt{t6}) \\
&\quad \wedge \texttt{do}(\texttt{put}(\texttt{water}, \texttt{in}(\texttt{bread_maker})), \texttt{t5}) \\
&\quad \wedge \texttt{do}(\texttt{put}(\texttt{flour}, \texttt{in}(\texttt{bread_maker})), \texttt{t4}) \\
&\quad \wedge \texttt{do}(\texttt{put}(\texttt{yeast}, \texttt{in}(\texttt{bread_maker})), \texttt{t3}) \\
&\quad \wedge \texttt{do}(\texttt{put}(\texttt{rape_seed_oil}, \texttt{in}(\texttt{bread_maker})), \texttt{t2}) \\
&\quad \wedge \texttt{consume}(\texttt{measure}(\texttt{water}, 330, \texttt{ml})), \texttt{t1}) \\
&\quad \wedge \texttt{consume}(\texttt{measure}(\texttt{flour}, 500, \texttt{g})), \texttt{t1}) \\
&\quad \wedge \texttt{consume}(\texttt{measure}(\texttt{yeast}, 5, \texttt{g})), \texttt{t1}) \\
&\quad \wedge \texttt{consume}(\texttt{measure}(\texttt{rape_seed_oil}, 25, \texttt{ml})), \texttt{t1}) \\
&\quad \wedge \texttt{onhand}(\texttt{measure}(\texttt{water}, 330, \texttt{ml})), \texttt{t1}) \\
&\quad \wedge \texttt{onhand}(\texttt{measure}(\texttt{flour}, 500, \texttt{g})), \texttt{t1}) \\
&\quad \wedge \texttt{onhand}(\texttt{measure}(\texttt{yeast}, 5, \texttt{g})), \texttt{t1}) \\
&\quad \wedge \texttt{onhand}(\texttt{measure}(\texttt{rape_seed_oil}, 25, \texttt{ml})), \texttt{t1}) \\
&\quad \wedge \texttt{sequence}([\texttt{t1}, \texttt{t2}, \texttt{t3}, \texttt{t4}, \texttt{t5}, \texttt{t6}, \texttt{t7}])
\end{aligned}$$

Similarly, for the ingredients that are assumed in Γ.

$$\begin{aligned}
&\texttt{onhand}(\texttt{measure}(\texttt{water}, 330, \texttt{ml})), \texttt{t1}), \quad \texttt{onhand}(\texttt{measure}(\texttt{flour}, 500, \texttt{g})), \texttt{t1}), \\
&\texttt{onhand}(\texttt{measure}(\texttt{yeast}, 5, \texttt{g})), \texttt{t1}), \quad \texttt{onhand}(\texttt{measure}(\texttt{butter}, 30, \texttt{g})), \texttt{t1})
\end{aligned}$$

The term substitution $\Gamma[\alpha/\beta]$ results in the following.

$$\begin{aligned}
&\texttt{onhand}(\texttt{measure}(\texttt{water}, 330, \texttt{ml})), \texttt{t1}), \quad \texttt{onhand}(\texttt{measure}(\texttt{flour}, 500, \texttt{g})), \texttt{t1}), \\
&\texttt{onhand}(\texttt{measure}(\texttt{yeast}, 5, \texttt{g})), \texttt{t1}), \quad \texttt{onhand}(\texttt{measure}(\texttt{rape_seed_oil}, 25, \texttt{ml})), \texttt{i1})
\end{aligned}$$

Fig. 3. Example of substitution for a clause in a recipe and a set of assumption.

Proposition 1. *The term substitution operation satisfies reflexivity, but not transitivity, associativity, nor reversibility.*

- Reflexivity: $\phi[\alpha/\alpha] = \alpha$
- Transitivity: $(\phi[\alpha/\beta])[\beta/\gamma] = \phi[\alpha/\gamma]$
- Associativity: $(\phi[\alpha/\beta])[\gamma/\delta] = (\phi[\gamma/\delta])[\alpha/\beta]$
- Reversibility: $(\phi[\alpha/\beta])[\beta/\alpha] = \phi$

Proof (*Reflexivity*)**.** Follows directly from definitions. (Transitivity) For a counterexample, let $\phi = \texttt{r(a, b)}$, $\alpha = \texttt{a}$, $\beta = \texttt{b}$, and $\gamma = \texttt{c}$. So $(\phi[\alpha/\beta])[\beta/\gamma] = \texttt{r(c, c)}$ and $\phi[\alpha/\gamma] = \texttt{r(c, b)}$. (Associativity) For a counterexample, let $\phi = \texttt{r(a, b)}$, $\alpha = \texttt{a}$, $\beta = \texttt{b}$, $\gamma = \texttt{b}$ and $\delta = \texttt{a}$.

So $(\phi[\alpha/\beta])[\gamma/\delta] = \mathtt{r(a,a)}$ and $(\phi[\gamma/\delta])[\alpha/\beta] = \mathtt{r(b,b)}$. (Reversibility) For a counterexample, let $\phi = \mathtt{r(a,b)}$, $\alpha = \mathtt{a}$, and $\beta = \mathtt{b}$. So $(\phi[\alpha/\beta])[\beta/\alpha] = \mathtt{r(a,a)}$. □

Definition 7. We define the Subterms function as follows where it returns a multiset, using square braces to denote the multiset and $\oplus$ to denote multiset union.

- If ϕ is a constant symbol or a variable symbol, then $\mathsf{Subterms}(\phi) = [\phi]$.
- If ϕ is a term of the form $\alpha(\beta_1, \ldots, \beta_n)$, then

$$\mathsf{Subterms}(\phi) = [\alpha(\beta_1, \ldots, \beta_n)] \oplus \mathsf{Subterms}(\beta_1) \oplus \cdots \oplus \mathsf{Subterms}(\beta_n)$$

- If ϕ is an atom of the form $\alpha(\beta_1, \ldots, \beta_n)$, then

$$\mathsf{Subterms}(\phi) = \mathsf{Subterms}(\beta_1) \oplus \cdots \oplus \mathsf{Subterms}(\beta_n)$$

- If ϕ is a literal of the form $\neg\psi$, then $\mathsf{Subterms}(\phi) = \mathsf{Subterms}(\psi)$.
- If ϕ is a clause of the form $\psi_1 \leftarrow \psi_2 \wedge \cdots \wedge \psi_n$, then

$$\mathsf{Subterms}(\phi) = \mathsf{Subterms}(\psi_1) \oplus \cdots \oplus \mathsf{Subterms}(\psi_n)$$

Example 33. For the clause below, the set of subterms is {meringues, bake(tray, 120C, 1.75 h), tray, 120C, 1.75 h, pour(contents(bowl), on(tray)), contents(bowl), on(tray), bowl, tray, T6, T5, T4}.

```
onhand(meringues, T6)
        ← do(bake(tray, 120C, 1.75 h), T5)
        ∧ do(pour(contents(bowl), on(tray)), T4)
```

The following definition of isomorphic specifies that two clauses are isomorphic iff they have the same number of conditions and for each condition and for the head, the corresponding atom in the other clause is isomorphic, and two atoms are isomorphic if they have the same number of terms.

Definition 8. For formulae ϕ and ψ, ϕ is **isomorphic** with ψ iff (1) ϕ is a clause of the form $\phi_1 \leftarrow \phi_2 \wedge \cdots \wedge \phi_n$ and ψ is a clause of the form $\psi_1 \leftarrow \psi_2 \wedge \cdots \wedge \psi_n$ and for each $i \in \{1, \ldots, n\}$, ϕ_i is isomorphic with ψ_i; Or (2) ϕ is an atom of the form $\alpha(\beta_1, \ldots, \beta_m)$ and ψ is an atom of the form $\alpha'(\beta'_1, \ldots, \beta'_m)$.

Example 34. The clauses in Examples 7 and 8 are isomorphic, and the clauses in Fig. 3 are isomorphic.

The following result shows that we can turn one formula into another using term substitution when all the subterms are unique, and the two formulae are isomorphic. For this, we use the notion of a syntax tree. Each formula can equivalently be represented by a **syntax tree** where the subformulae and terms are subtrees: For each node in the tree, we label it with a formula, subformula, term or subterm. Each leaf is labelled with constant or variable symbol. If a node is labelled with a (sub)term $\alpha(\beta_1, \ldots, \beta_n)$, then the children are labelled with the subterms β_1 to β_n. If a node is labelled with an atom $\alpha(\beta_1, \ldots, \beta_n)$, then the children are labelled with the terms β_1 to β_n. If a node is labelled with a literal $\neg\alpha$, then the child is labelled with the atom α. If a node is labelled with a clause $\psi_1 \leftarrow \psi_2 \wedge \cdots \wedge \psi_n$, then the children are labelled with the literals ψ_1 to ψ_n. We assume that a syntax tree for a formula ϕ labelling the root is the smallest tree that satisfies the above constraints. Finally, a syntax tree T for ϕ has **unique leaves** iff n and n' are leaves in T and $n \neq n'$ and β labels n and β' labels n' then $\beta \neq \beta'$.

Proposition 2. *For clauses ϕ and ψ, if ϕ has syntax tree T and T has unique leaves, and ϕ is isomorphic to ψ, then there is $[\alpha_1/\beta_1], \ldots, [\alpha_n/\beta_n]$ such that*

$$\phi([\alpha_1/\beta_1], \ldots, [\alpha_n/\beta_n]) = \psi$$

Proof. Since ϕ and ψ are isomorphic, they have identical syntax trees. Let T_ϕ be the syntax tree for ϕ and let T_ψ be the syntax tree for ψ. And let L_ϕ be the set of labels for T_ϕ (i.e. $L_\phi = \{\delta | n$ is a node in T_ϕ and n is labelled with $\delta\}$). Since T_ϕ is uniquely labelled, there is a surjective function f from L_ϕ to L_ψ. Therefore, for each $\alpha \in L_\phi$, there is a $\beta \in L_\psi$, where $f(\alpha) = \beta$. Therefore, f can be represented by a sequence of term substitutions $[\alpha_1/\beta_1], \ldots, [\alpha_n/\beta_n]$ such that $\phi([\alpha_1/\beta_1], \ldots, [\alpha_n/\beta_n]) = \psi$. □

Another way to constrain term substitution is to only introduce terms that do not already appear in the formula which we define as follows.

Definition 9. Let $\Sigma = \langle \Pi_1, \ldots, \Pi_n \rangle$ be a substitution tuple where each Π_i is a term substitution. Σ is **independent** iff for each i, where Π_i is of the form $[\alpha_i/\beta_i]$, and for each j, where $i \neq j$ and Π_j is of the form $[\alpha_j/\beta_j]$, $\alpha_i \neq \alpha_j$ and $\alpha_i \neq \beta_j$.

Proposition 3. *If the substitution tuple of term substitutions $\Sigma = \langle [\alpha_1/\beta_1], \ldots, [\alpha_i/\beta_i] \rangle$ is independent, then the sequence of substitutions is associative (i.e. $\phi\Sigma = \phi\Sigma'$, for any formula ϕ, and any permutation Σ' of Σ).*

Proof. Assume Σ is independent. So for each $[\alpha_i/\beta_i]$ in Σ, there is no $[\alpha_j/\beta_j]$ such that $\alpha_i = \alpha_j$, or $\alpha_i = \beta_j$. So if $[\alpha_i/\beta_i]$ is applied to $(\ldots(\phi[\alpha_1/\beta_1])\ldots)[\alpha_{i-1}/\beta_{i-1}]$, then the following holds

$$\alpha_i \notin \mathsf{Subterms}((\ldots(\phi[\alpha_1/\beta_1])\ldots)[\alpha_{i-1}/\beta_{i-1}]) \setminus \mathsf{Subterms}(\phi)$$

but it may be the case that $\alpha_i \in \mathsf{Subterms}(\phi)$. So $\beta_i \in \mathsf{Subterms}(\phi\Sigma)$ iff $\alpha_i \in \mathsf{Subterms}(\phi)$. So $\beta_i \in \mathsf{Subterms}(\phi\Sigma')$ iff $\alpha_i \in \mathsf{Subterms}(\phi)$. So $\phi\Sigma = \phi\Sigma'$. □

To conclude, term substitution replaces specific ingredients and equipment in process clauses in Δ and in assumptions in Γ, and it replaces specific actions in process clauses in Δ.

5.2. Condition substitution

Now we consider the second level of substitution which is for replacing conditions in a clause as defined next.

Definition 10. For a clause ϕ of the form $\psi_1 \leftarrow \psi_2 \wedge \cdots \wedge \psi_m$, the **condition substitution** of the set of literals Φ, by the set of literals Ψ, is $\phi[\Phi/\Psi]$ defined as follows where $\{\psi_2', \ldots, \psi_n'\}$ is $(\{\psi_2, \ldots, \psi_m\} \setminus \Phi) \cup \Psi$.

$$\phi[\Phi/\Psi] = \psi_1 \leftarrow \psi_2' \wedge \cdots \wedge \psi_n'$$

If $\mathsf{Tail}(\phi) = \Phi$, then let $\phi[\Phi/\emptyset] = \mathsf{Head}(\phi) \leftarrow \top$. Also, for a set of formulae $\Xi = \{\phi_1, \ldots, \phi_n\}$, let $\Xi[\Phi/\Psi]$ be $\{\phi_1[\Phi/\Psi], \ldots, \phi_n[\Phi/\Psi]\}$.

Using condition substitution, we can handle complex substitutions such as ingredient plus action as illustrated by the following example.

Example 35. Consider the need to substitute the following conditions

$$\phi_1 = \texttt{onhand(measure(egg\_white, 100, ml), T1)}$$
$$\phi_2 = \texttt{do(put(measure(egg\_white, 100, ml), in(bowl)), T1)}$$

in the following process clause ψ.

```
onhand(meringues, T6)
        ← do(bake(tray, 120C, 1.75 h), T5)
        ∧ do(pour(contents(bowl), on(tray)), T4)
        ∧ do(use_electric_mixer(bowl, 15 min), T3)
        ∧ do(put(measure(castor_sugar, 50, g), in(bowl)), T2)
        ∧ do(put(measure(egg_white, 100, ml), in(bowl)), T1)
        ∧ onhand(measure(castor_sugar, 50, g), T1)
        ∧ onhand(measure(egg_white, 100, ml), T1)
        ∧ sequence([T1, T2, T3, T4, T5, T6])
```

Let $\phi_1' = \texttt{onhand(measure(aquafaba), 100, ml), T1)}$, and $\phi_2' = \texttt{do(whip(aquafaba, 10 min), T1)}$. So we can undertake the condition substitution $\psi[\{\phi_1, \phi_2\}/\{\phi_1', \phi_2'\}]$ to give the following revised process clause.

```
onhand(meringues, T6)
        ← do(bake(tray, 120C, 1.75 h), T5)
        ∧ do(pour(contents(bowl), on(tray)), T4)
        ∧ do(use_electric_mixer(bowl, 15 min), T3)
        ∧ do(put(measure(castor_sugar, 50, g), in(bowl)), T2)
        ∧ onhand(measure(castor_sugar, 50, g), T1)
        ∧ do(whip(aquafaba, 10 min), T1)
        ∧ onhand(measure(aquafaba, 100, ml), T1)
        ∧ sequence([T1, T2, T3, T4, T5, T6])
```

When comparing term substitution with condition substitution, we see that term substitution is across all of the recipe. Every occurrence of the term in the recipe is substituted. But we may want to only substitute in parts of the recipe. For example, we may want to replace egg for making the base of the cake, but use egg in the cream filling for the cake. We can do this by just substituting in individual formulae using condition substitution. So we can focus on individual clauses using condition substitutions. To support this, there is an advantage in splitting a recipe into a larger number of shorter process clauses as this will allow more targeted substitutions.

Proposition 4. *The condition substitution operation satisfies reflexivity, but not transitivity, associativity, nor reversibility.*

- Reflexivity: $\phi[\Phi/\Phi] = \phi$
- Transitivity: $(\phi[\Phi_1/\Phi_2])[\Phi_2/\Phi_3] = \phi[\Phi_1/\Phi_3]$
- Associativity: $(\phi[\Phi_1/\Phi_2])[\Phi_3/\Phi_4] = (\phi[\Phi_3/\Phi_4])[\Phi_1/\Phi_2]$

- Reversibility: $(\phi[\Phi_1/\Phi_2])[\Phi_2/\Phi_1] = \phi$

Proof (*Reflexivity*)**.** Follows directly from the definition. (Transitivity) For counterexample, let ϕ be $\psi \leftarrow \psi'.$, let $\Phi_1 = \emptyset$, $\Phi_2 = \{\psi'\}$, and $\Phi_3 = \emptyset$. So $(\phi[\Phi_1/\Phi_2])[\Phi_2/\Phi_3] = \psi \leftarrow \top$ and $\phi[\Phi_1/\Phi_3] = \psi \leftarrow \psi'$. (Associativity) For counterexample, let ϕ be $\psi \leftarrow \psi.$, let $\Phi_1 = \emptyset$, $\Phi_2 = \{\psi'\}$, $\Phi_3 = \{\psi'\}$, and $\Phi_4 = \emptyset$. So $(\phi[\Phi_1/\Phi_2])[\Phi_3/\Phi_4] = \psi \leftarrow \top$, and $(\phi[\Phi_3/\Phi_4])[\Phi_1/\Phi_2] = \psi \leftarrow \psi'$. (Reversibility) For counterexample, let ϕ be $\psi \leftarrow \psi.$, let $\Phi_1 = \emptyset$, and $\Phi_2 = \{\psi'\}$. So $(\phi[\Phi_1/\Phi_2])[\Phi_2/\Phi_1] = \psi \leftarrow \top$. □

Proposition 5. *For clauses ϕ and ψ, if* $\mathsf{Head}(\phi) = \mathsf{Head}(\psi)$*, then there are sets Φ and Ψ such that $\phi[\Phi/\Psi] = \psi$.*

Proof. Let ϕ be $\delta \leftarrow \delta_1 \wedge \cdots \wedge \delta_n$ and let ϕ be $\delta \leftarrow \delta'_1 \wedge \cdots \wedge \delta'_m$. If we let $\Phi = \{\delta_1 \ldots \delta_n\}$, and we let $\Psi = \{\delta'_1 \ldots \delta'_m\}$, then $\phi[\Phi/\Psi] = \psi$. □

Since a recipe is composed of one or more process clauses, if we want to add or delete ingredients and/or equipment, or we want to add or delete actions, then we can use condition substitution.

5.3. Clause substitution

The third level of substitution is clause substitution which allows for substitution of large components of recipes as defined below.

Definition 11. For a set of clauses Δ, the **clause substitution** of the set of clauses Φ, by the set of clauses Ψ, denoted $\Delta[\Phi/\Psi]$, is defined as $\Delta[\Phi/\Psi] = (\Delta \setminus \Phi) \cup \Psi$.

Example 36. Consider the recipe $\Delta = \{\phi_1\}$ where ϕ_1 is the following clause. Also suppose we lack a `beef_patty` (i.e. `onhand(count(beef_patty, 1), T1)` is not in our assumptions).

```
onhand(beef_burger, T4)
        ← do(put(beef_patty, in(burger_bap), T3)
        ∧ do(cook(burger_bap, on(griddle), 8 min), T2)
        ∧ do(cook(beef_patty, on(griddle), 5 min), T1)
        ∧ onhand(count(burger_bap, 1), T1)
        ∧ onhand(count(beef_patty, 1), T1)
        ∧ sequence([T1, T2, T3, T4])
```

If we have the necessary ingredients, then we can add the following process clause ϕ_2 to the recipe. In other words, we can make the clause substitution $\Delta[\emptyset/\{\phi_2\}]$ which results in $\{\phi_1, \phi_2\}$.

```
onhand(count(beef_patty, 1), T6)
        ← do(mix(bowl), T5)
        ∧do(put(minced_meat, in(bowl)), T4)
        ∧do(put(chopped_onion, in(bowl)), T3)
        ∧do(put(egg, in(bowl)), T2)
        ∧do(put(pepper, in(bowl)), T1)
        ∧ sequence([T1, T2, T3, T4, T5, T6])
```

We also use clause substitutions to effect the condition substitutions on a set of clauses. A condition substitution is of the form $\phi[\Phi/\Psi]$ which specifies that the conditions of clause ϕ to be updated. However, this then needs to be done on the set of clauses. We do this by embedding the condition substitution within a clause substitution as follows: $\Delta[\{\phi\}/\{\phi[\Phi/\Psi]\}]$. So this says that ϕ is replaced by $\phi[\Phi/\Psi]$ in Δ.

Example 37. Returning to Example 35 where the condition substitution is $\psi[\{\phi_1, \phi_2\}/\{\phi'_1, \phi'_2\}]$. Suppose $\Delta = \{\psi\}$. So the condition substitution can be embedded in the clause substitution as follows.

$$\Delta[\{\psi\}/\{\psi[\{\phi_1, \phi_2\}/\{\phi'_1, \phi'_2\}]\}]$$

Proposition 6. *For a set of clauses Δ, the clause substitution operation satisfies empty and reflexivity but not transitivity, associativity, nor reversibility.*

- Empty: $\Delta[\Delta/\emptyset] = \emptyset$
- Reflexivity: $\Delta[\Phi/\Phi] = \Delta$
- Transitivity: $(\Delta[\Phi_1/\Phi_2])[\Phi_2/\Phi_3] = \Delta[\Phi_1/\Phi_3]$
- Associativity: $(\Delta[\Phi_1/\Phi_2])[\Phi_3/\Phi_4] = (\Delta[\Phi_3/\Phi_4])[\Phi_1/\Phi_2]$
- Reversibility: $(\Delta[\Phi_1/\Phi_2])[\Phi_2/\Phi_1] = \Delta$

Proof. Recall $\Delta[\Phi/\Psi] = (\Delta \setminus \Phi) \cup \Psi$. (Empty) $\Delta[\Delta/\emptyset] = (\Delta \setminus \Delta) \cup \emptyset = \emptyset$. (Reflexivity) $\Delta[\Phi/\Phi] = \Delta = (\Delta \setminus \Phi) \cup \Phi = \Delta$. (Transitivity) Consider the counterexample where $\Delta = \{\phi\}$, $\Phi_1 = \emptyset$, $\Phi_2 = \{\phi\}$, and $\Phi_3 = \emptyset$. So $(\Delta[\Phi_1/\Phi_2])[\Phi_2/\Phi_3] = \emptyset$ whereas $\Delta[\Phi_1/\Phi_3] = \Delta$. (Associativity) Consider the counterexample where $\Delta = \emptyset$, $\Phi_1 = \emptyset$, $\Phi_2 = \{\phi\}$, $\Phi_3 = \{\phi\}$, and $\Phi_4 = \emptyset$. So $(\Delta[\Phi_1/\Phi_2])[\Phi_3/\Phi_4] = \emptyset$, whereas $(\Delta[\Phi_3/\Phi_4])[\Phi_1/\Phi_2] = \{\phi\}$. (Reversibility) Consider the counterexample where $\Delta = \{\phi\}$, $\Phi_1 = \emptyset$, and $\Phi_2 = \{\phi\}$. So $(\Delta[\Phi_1/\Phi_2])[\Phi_2/\Phi_1] = \emptyset$ whereas $\Delta = \{\phi\}$. □

So clause substitution is a straightforward way to add and delete clauses. For instance, to add intermediate steps as in Example 36. We can use it directly or use as a vehicle for effecting condition substitutions as in Example 37.

5.4. Sequences of substitutions

In the previous subsections, we have introduced three types of substitution, namely term substitutions, condition substitutions, and clause substitutions. As we will see in the next section, we often need to consider sequences of substitutions in order to obtain a satisfactory recipe.

In order to use all three types of substitution in a sequence, we first need to generalize term substitutions and condition substitutions to sets of clauses.

- if Π is a term substitution of the form $[\alpha/\beta]$, then $\Delta\Pi = \{\phi\Pi \mid \phi \in \Delta\}$.
- if Π is a condition substitution of the form $[\Phi/\Psi]$, then $\Delta\Pi = \{\phi[\Phi/\Psi] \mid \phi \in \Delta\}$.

As we define next, a substitution tuple is a list of zero or more substitutions where each substitution is a term substitution, condition substitution, or clause substitution. We illustrate this in Fig. 4.

Definition 12. A **substitution tuple** is an n-tuple $\langle \Pi_0, \ldots, \Pi_n \rangle$ where $n \geq 0$ and each Π_i is a term, condition, or clause, substitution.

Using a substitution tuple, we can make a sequence substitution which is the first substitution in the list applied to the knowledgebase, and then with the result of this substitution, the second substitution in the list applied, and so on.

Definition 13. Let Δ be a set of formulae, and let $\Sigma = \langle \Pi_0, \ldots, \Pi_n \rangle$ be a substitution tuple. A **sequence substitution** is

$$\Delta\Sigma = (\ldots((\Delta\Pi_0)\Pi_1)\ldots)\Pi_n$$

If the substitution tuple is empty (i.e. $\Sigma = \langle\rangle$), then $\Delta\Sigma = \Delta$. Also, we can concatenate sequence substitution (i.e. $(\Delta\Sigma_1)\Sigma_2 = \Delta(\Sigma_1 + \Sigma_2)$ where $\Sigma_1 + \Sigma_2$ is the concatenation of substitution tuples Σ_1 and Σ_2).

Following from the results in the previous subsections, we have reflexivity (i.e. if every substitution in a substitution tuple Σ is reflexive, then $\Delta\Sigma = \Delta$). However, as for the individual types of substitution, we do not have transitivity, associativity, or reversibility, in general.

The following result shows that we could just use a single clause substitution instead of a substitution sequence since the single clause substitution gives exactly the required revised knowledgebase. In other words, there is a clause substitution Π such that for any recipes Δ and Δ', $\Delta\Pi = \Delta'$. However, in general, we do not want to use a single clause substitution because we require: (**dynamic substitutions**), i.e. we want to be able infer the required substitutions from other knowledge as required by the context (whereas using a single clause substitution means that we have these substitutions known in advance); (**explainable substitutions**) i.e. we want to be able give the substitutions as minimal changes to the recipe and where these are clearly expounded in terms of the exact changes to the ingredients and steps in the recipe and why (whereas just using a single clause substitution would be like saying that we reject the original recipe and replace it with a new recipe).

Proposition 7. *For all substitution tuples $\Sigma = \langle \Pi_1 \ldots \Pi_n \rangle$, there is a clause substitution Π^* such that $\Delta\Sigma = \Delta\Pi^*$.*

Proof. Let Δ^* be the knowledgebase resulting from the sequence substitution $\Delta\Sigma$. Now let Π^* be $[\Delta/\Delta^*]$. So $\Delta\Pi^* = \Delta^*$. □

For explainable substitutions, we want to make minimal changes to a knowledgebase. For instance, if we need to change one ingredient from α to β in a clause in a recipe, then a single term substitution $[\alpha/\beta]$ might be sufficient, and so doing this, we clearly see what has changed, whereas for a clause substitution that changes Δ into Δ', we do not know exactly what has changed without looking at all the clauses in Δ' and have changed from Δ.

So if we need minimal changes, then we need to consider how we can define these. We start with the following subsidiary definitions which provide a partitioning of the substitutions in a substitution tuple.

- $\mathsf{TermSubs}(\Sigma) = \{\Pi \in \Sigma \mid \Pi \text{ is a term substitution }\}$
- $\mathsf{ConditionSubs}(\Sigma) = \{\Pi \in \Sigma \mid \Pi \text{ is a condition substitution }\}$
- $\mathsf{ClauseSubs}(\Sigma) = \{\Pi \in \Sigma \mid \Pi \text{ is a clause substitution }\}$

We now consider three criteria for measuring substitutions below that we will use as the basis of identifying a notion of minimality for substitution tuples. We explain these as follows: the length is simply the number of substitutions in the substitution tuple; the number of conditions revised is the number of clauses that have conditions amended; and the number of clauses revised is the number of clauses that have been removed plus the number of clauses that have been added.

We start with the recipe $\Delta = \{\phi\}$ where ϕ is defined as follows that is for cooking spaghetti.

$$\begin{aligned}\phi = \quad & \texttt{onhand(cooked(spaghetti),T4)} \leftarrow \\ & \texttt{do(boil(spaghetti,10min),T3)} \\ & \wedge\ \texttt{do(put(spaghetti,in(pan)),T2)} \\ & \wedge\ \texttt{do(pour(boiling\_water,in(pan)),T1)} \\ & \wedge\ \texttt{sequence}([\texttt{T1},\texttt{T2},\texttt{T3},\texttt{T4}])\end{aligned}$$

First we consider a primary substitution because we lack spaghetti (using term substitution). Let $\alpha = \texttt{spaghetti}$ and $\beta = \texttt{fusilli}$.

$$\begin{aligned}\phi[\alpha/\beta] = \quad & \texttt{onhand(cooked(fusilli),T4)} \leftarrow \\ & \texttt{do(boil(fusilli,10min),T3)} \\ & \wedge\ \texttt{do(put(fusilli,in(pan)),T2)} \\ & \wedge\ \texttt{do(pour(boiling\_water,in(pan)),T1)} \\ & \wedge\ \texttt{sequence}([\texttt{T1},\texttt{T2},\texttt{T3},\texttt{T4}])\end{aligned}$$

Now we consider a secondary substitution (using condition substitution) to deal with change in cooking time for the substituted pasta. Let $\alpha' = \texttt{do(boil(fusilli,10min),T3)}$ and $\beta' = \texttt{do(boil(fusilli,12min),T3)}$.

$$\begin{aligned}(\phi[\alpha/\beta])[\alpha'/\beta'] = \quad & \texttt{onhand(cooked(fusilli),T4)} \leftarrow \\ & \texttt{do(boil(fusilli,12min),T3)} \\ & \wedge\ \texttt{do(put(fusilli,in(pan)),T2)} \\ & \wedge\ \texttt{do(pour(boiling\_water,in(pan)),T1)} \\ & \wedge\ \texttt{sequence}([\texttt{T1},\texttt{T2},\texttt{T3},\texttt{T4}])\end{aligned}$$

We conclude with another primary substitution because we also lack a pan (using term sibstitution). Let $\alpha'' = \texttt{pan}$ and $\beta'' = \texttt{iron\_casserole\_dish}$.

$$\begin{aligned}((\phi[\alpha/\beta])[\alpha'/\beta'])[\alpha''/\beta''] = \quad & \texttt{onhand(cooked(fusilli),t4)} \leftarrow \\ & \texttt{do(boil(fusilli,12min),T3)} \\ & \wedge\ \texttt{do(put(fusilli,in(iron\_casserole\_dish)),T2)} \\ & \wedge\ \texttt{do(pour(boiling\_water,in(iron\_casserole\_dish)),T1)} \\ & \wedge\ \texttt{sequence}([\texttt{T1},\texttt{T2},\texttt{T3},\texttt{T4}])\end{aligned}$$

Fig. 4. Example with substitution tuple involving primary and secondary substitutions. Let $\Pi_0 = [\alpha/\beta]$, $\Pi_1 = [\alpha'/\beta']$, and $\Pi_2 = [\alpha''/\beta'']$. So $\langle \Pi_0, \Pi_1, \Pi_2 \rangle$ is a substitution tuple.

- Σ has **length** n iff Σ is an n-tuple.
- Σ has n **conditions revised** iff

$$|\{\phi \in \Delta \mid \phi[\Phi/\Psi] \neq \phi \text{ and } [\Phi/\Psi] \in \mathsf{ConditionSubs}(\Sigma)\}| = n$$

- Σ has n **clauses revised** iff

$$|((\Delta \setminus \bigcup_{[\Phi/\Psi]\in\mathsf{ClauseSubs}(\Sigma)} \Phi) \cup (\bigcup_{[\Phi/\Psi]\in\mathsf{ClauseSubs}(\Sigma)} \Psi))| = n$$

In the following definition, we specify that substitution sequence Σ is smaller than substitution sequence Σ' iff one of the following three options holds: (Option 1) Σ has fewer clause revisions than Σ'; (Option 2) Σ and Σ' have equal clause revisions and Σ has fewer condition revisions than Σ'; (Option 3) Σ and Σ' have equal clause revisions and Σ and Σ' have equal conditions revisions and Σ is shorter than Σ'.

Definition 14. For substitution tuples Σ and Σ', let Σ have length n_{length}, have $n_{conditions}$ conditions revised, and have $n_{clauses}$ clauses revised, and let Σ' have length n'_{length}, have $n'_{conditions}$ conditions revised, and have $n'_{clauses}$ clauses revised. Σ is **smaller** than Σ' iff one of the following three options holds:

- (Option 1) $n_{clauses} < n'_{clauses}$;
- (Option 2) $n_{clauses} = n'_{clauses}$ and $n_{conditions} < n'_{conditions}$;
- (Option 3) $n_{clauses} = n'_{clauses}$ and $n_{conditions} = n'_{conditions}$ and $n_{length} < n'_{length}$.

The smaller-than-or-equal relation is defined as follows: Σ is **smaller-than-or-equal** to Σ' iff Σ is smaller than Σ' or $n_{clauses} = n'_{clauses}$ and $n_{conditions} = n'_{conditions}$ and $n_{length} = n'_{length}$.

The empty sequence $\langle\rangle$ is smaller than any non-empty sequence, and it constitutes the minimal element in the posit containing all substitution tuples.

Proposition 8. *The smaller-than-or-equal relation is a partial ordering (reflexive, anti-symmetric, and transitive) over the set of all substitution tuples.*

Proof. Since each substitution tuple is characterized in terms of three numbers (n_{length}, $n_{conditions}$ and $n_{clauses}$), and the definition of the relation is in terms of the relative magnitude of these numbers, we satisfy the reflexive, anti-symmetric, and transitive conditions. □

In the rest of the paper, we will investigate in detail how we use substitutions, and how we meet our desiderata of dynamic and explainable substitutions.

6. Stages of substitution

We now consider how we identify and apply substitutions. As we introduced earlier, a **primary substitution** is a substitution that has been undertaken because we lack some food item or equipment, or possibly we lack the ability or desire to carry out some action, whereas a **secondary substitution** is a substitution that has been carried out to deal with problems raised by the primary substitution. We give an example of primary and secondary substitutions in Fig. 4.

We can substitute intermediate products in a recipe (e.g. `sliced_carrot` for `diced_carrot`) as shown in Example 38. This involves swapping the intermediate as a primary substitution, and then change the process clause as secondary substitution. This also needs an integrity constraint that captures sliced carrot requires a slice action in recipe.

Example 38. Consider a substitution of `sliced_carrot` for `diced_carrot` in the following clause.

$$\texttt{onhand(diced\_carrot, T3)} \leftarrow \texttt{do(dice(carrot), T2)} \wedge \texttt{onhand(carrot, T1)}$$

This gives the following clause

$$\texttt{onhand(sliced\_carrot, T3)} \leftarrow \texttt{do(dice(carrot), T2)} \wedge \texttt{onhand(carrot, T1)}$$

The following integrity constraint then identifies that we cannot obtain `sliced_carrot` from the action `do(dice(carrot))`.

$$\bot \leftarrow \texttt{onhand(sliced\_carrot, T3)} \wedge \texttt{do(dice(carrot), T2)}$$

The secondary substitution that involves changing `do(dice(carrot), T2)` to `do(slice(carrot), T2)` results in the following clause.

$$\texttt{onhand(sliced\_carrot, T3)} \leftarrow \texttt{do(slice(carrot), T2)} \wedge \texttt{onhand(carrot, T1)}$$

We now consider the steps required for substitution starting with the item to be substituted which raises the primary substitution, and then potentially secondary substitutions. We formalize these in the following sections.

6.1. Primary substitutions

Primary substitution in this section is about making a substitution for something that is missing (equipment/ingredient) or the lack of desire or ability to carry out a specific action. For this, we use the following definition to identify candidates for substitution.

Definition 15. Let (Δ, Γ, Θ) be a cooking scenario. let α be the item that is to be replaced, let d is a distance measure, and ω is a threshold value. Note, we formalize primary substitutions as term substitutions.

$$\text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega, \Theta) = \{[\alpha/\beta] \mid d(\alpha, \beta) \leq \omega\}$$

Example 39. Consider the clause ψ defined as follows that is for cooking spaghetti.

$$\begin{aligned}\psi = \quad & \texttt{onhand(cooked(spaghetti), T4)} \leftarrow \\ & \texttt{do(boil(spaghetti, 10 min), T3} \\ & \wedge\ \texttt{do(put(spaghetti, in(pan)), T2)} \\ & \wedge\ \texttt{do(pour(boiling\_water, in(pan)), T1)} \\ & \wedge\ \texttt{sequence([T1, T2, T3, T4])}\end{aligned}$$

Also suppose we have `fusilli`, `tagliatelle`, `penne`, `linguine`, `canneloni`, `lasagne`, `rigitoni` and `farfalle` as alternatives to `spaghetti`, with the following distance measures.

$$\begin{aligned} &d(\texttt{spaghetti}, \texttt{fusilli}) = 0.65 && d(\texttt{spaghetti}, \texttt{tagliatelle}) = 0.44 \\ &d(\texttt{spaghetti}, \texttt{penne}) = 0.67 && d(\texttt{spaghetti}, \texttt{linguine}) = 0.21 \\ &d(\texttt{spaghetti}, \texttt{canneloni}) = 0.88 && d(\texttt{spaghetti}, \texttt{lasagne}) = 0.98 \\ &d(\texttt{spaghetti}, \texttt{rigitoni}) = 0.63 && d(\texttt{spaghetti}, \texttt{farfalle}) = 0.52 \end{aligned}$$

So if we set $\omega = 0.45$, then we obtain the following

$$\text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega, \Theta) = \{[\texttt{spaghetti/tagliatelle}], [\texttt{spaghetti/linguine}]\}$$

For a choice of missing ingredient α, recipe Δ, and assumptions Γ, and d being the role distance function, if $\omega = 0$, then $\text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega)$ will contain the substitutions $[\alpha/\beta]$ where β has the same roles as α, and if $\omega = 1$, then $\text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega)$ will contain all possibles substitutions $[\alpha/\beta]$ where α and β are ingredients (i.e. anything can be a substitution). For most word embeddings d, one would expect that $\text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega)$ will just be the substitution $[\alpha/\alpha]$ (as for most word embeddings, and for almost all pairs of words α and β, there is non-zero distance between them).

If we are missing multiple ingredients, then we make multiple substitutions. For instance, if we miss two ingredients, then we find the best substitute for the first missing ingredient, and then find the best substitute for the second. But, since each substitution is with respect to a missing ingredient and not the other ingredients, the sequence for this does not matter.

Example 40. Consider the clause ψ defined as follows that is for cooking spaghetti.

$$\begin{aligned}\psi = \quad & \texttt{onhand(cooked(spaghetti), T4)} \leftarrow \\ & \texttt{do(boil(spaghetti, 10 min), T3)} \\ & \wedge\ \texttt{do(put(spaghetti, in(pan)), T2)} \\ & \wedge\ \texttt{do(pour(boiling\_water, in(pan)), T1)} \\ & \wedge\ \texttt{sequence([T1, T2, T3, T4])}\end{aligned}$$

Suppose we lack `spaghetti` and we lack `boiling_water`, then finding the substitute for `spaghetti` is not affected by what we use to substitute for `boiling_water` and vice versa.

Because the choice of each substitution is made with respect to the original ingredients, the other primary substitutions do not interfere with each other, and so we have the following result.

Proposition 9. *If* $\beta_1 \in \text{Substitutions}(\alpha_1, \Delta, \Gamma, d, \omega, \Theta)$*, and* $\beta_2 \in \text{Substitutions}(\alpha_2, \Delta, \Gamma, d, \omega, \Theta)$*, then* $(\Delta[\alpha_1/\beta_1])[\alpha_2/\beta_2] = (\Delta[\alpha_2/\beta_2])[\alpha_1/\beta_1]$.

Proof. Follows directly from associativity for term substitution (Proposition 1). □

The Substitutions function is monotonic in the threshold. So for and α, Δ, Γ, and d, if $\omega \leq \omega'$, then we have the following.

$$\text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega, \Theta) \subseteq \text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega', \Theta)$$

Similarly, the Substitutions function is monotonic in Δ, and in Γ. Hence, if $\Delta \subseteq \Delta'$ and $\Gamma \subseteq \Gamma'$, then we have the following.

$$\text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega, \Theta) \subseteq \text{Substitutions}(\alpha, \Delta', \Gamma', d, \omega, \Theta)$$

In addition, the Substitutions function is monotonic in the scope of the distance function. In other words, if $\{\beta \mid d(\alpha, \beta)\} \subseteq \{\beta \mid d'(\alpha, \beta)\}$, then we have the following.

$$\text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega, \Theta) \subseteq \text{Substitutions}(\alpha, \Delta, \Gamma, d', \omega, \Theta)$$

However, the process of substituting for an ingredient, equipment, or action, does not take account of how a substitution might fundamentally change the final dish that is produced by a recipe. For example, suppose we have a recipe Δ for `chickensupreme`, α is `count(chicken(breast, skin_on), 2)`, and β is `count(can(butter_beans, 400 g), 2)`. The recipe $\Delta[\alpha/\beta]$ is not really for chicken supreme, but rather for a dish that we might call bean supreme.

A consequence of this issue (i.e. that a substitution does not necessarily change the name of the final food item being produced) means that we do not change the properties of the item being produced. Continuing the example of substituting chicken for butter beans in the recipe for chicken beans, if we do not change the name of the final item (in this case, chicken supreme), the properties of the final food item will be unchanged. So even if we have substituted butter beans, and thereby have a vegetarian dish, this will be recognized as property of the dish since the name of the dish has not changed. We will deal with issue in the next subsection using a secondary substitution.

6.2. Secondary substitutions

One or more secondary substitutions are required when a primary substitution causes the resulting recipe to be inconsistent with the integrity constraints and background knowledge.

Example 41. In a recipe for spaghetti bolognese, we can substitute spaghetti with fusilli: But fusilli only takes 12 min to cook. So if we do this substitution, then it is essential to substitute the cooking time. To do this, we use integrity constraints (which we can extract from recipes). For example,

$$\bot \leftarrow \texttt{do(boil(fusilli, } \delta\texttt{), T)} \wedge \delta \neq \texttt{12 min}$$

So continuing Fig. 4, if we have say `do(boil(fusilli, 10 min), i3)` $\in \Gamma$, then with the integrity constraint, we obtain the inference $\bot$ (i.e. an inconsistency). Hence, mitigation is required.

In order to identify the need for mitigation, we need to identify any inconsistencies between the atoms that arise in the execution. A mitigation is required if the following holds.

$$\bot \in \text{Execute}(\Delta, \Gamma, \Theta)$$

As defined later in this subsection, a mitigation is one or more substitutions that ensures the execution is consistent with the integrity constraints.

Definition 16. A **mitigation** for a cooking session (Δ, Γ, Θ) is a substitution tuple Σ s.t. for some set of assumptions Γ', $\bot \notin \text{Execute}(\Delta\Sigma, \Gamma', \Theta)$. A **minimal mitigation** is a mitigation Σ for (Δ, Γ, Θ) and no Σ' that is a mitigation for (Δ, Γ, Θ) such that Σ' is smaller than Σ.

Example 42. Continuing Example 41, we can use the substitution $[\alpha/\beta]$ as a mitigation, where $\alpha = \texttt{do(boil(fusilli, 10 min), T)}$ and $\beta = \texttt{do(boil(fusilli, 12 min), T)}$. This will ensure the conditions $\bot \notin \text{Execute}(\Delta[\alpha/\beta], \Gamma', \Theta)$ and $\phi \in \text{Execute}(\ldots(\Delta[\alpha/\beta], \Gamma', \Theta))$ are satisfied. Furthermore, this mitigation is a minimal mitigation.

Proposition 10. *If (Δ, Γ, Θ) is a cooking session, and $\bot \notin \text{Execute}(\Delta, \Gamma, \Theta)$, and for some output $\pi \in \text{Dishes}(\Delta, \Gamma, \Theta)$, it is the case that $\pi \in \text{Execute}(\Delta, \Gamma, \Theta)$, then the empty substitution tuple (i.e. $\langle\rangle$) is the minimal mitigation for (Δ, Γ, Θ).*

Proof. Because $\bot \notin \text{Execute}(\Delta, \Gamma, \Theta)$, and output $\pi \in \text{Execute}(\Delta, \Gamma, \Theta)$, the substitution tuple $\langle\rangle$ is trivially a mitigation for (Δ, Γ, Θ). Since there is no substitution tuple that is smaller than $\langle\rangle$, $\langle\rangle$ is the minimal mitigation for (Δ, Γ, Θ). □

As shown by the following result, it is not necessarily the case that we can find a secondary substitution that resolves the inconsistencies (or if we resolve it, we introduce another).

Proposition 11. *For a cooking session (Δ, Γ, Θ), it is not necessarily the case that there is a substitution tuple Σ such that Σ is a mitigation for (Δ, Γ, Θ).*

Proof. If for all Σ and Γ', it is the case that $\bot \in \text{Execute}(\Delta\Sigma, \Gamma', \Theta)$ holds, then the there is no mitigation for this cooking session. This can arise for instance if we have an integrity constraint $\bot \leftarrow \phi$ for each onhand atom ϕ in the language. □

Whilst the above proof is based on an extreme situation, it shows that in order to find a mitigation, there is a need to consider how the space of solutions is constrained by the cooking session including its integrity constraints and background knowledge. Furthermore, to show that it is normally possible to find mitigations, we consider the following property of Θ that allows us to allow get mitigations.

Definition 17. For a set of integrity constraints and background knowledge Θ is **sensible** iff there is a Δ and Γ such that $\bot \notin \text{Execute}(\Delta, \Gamma, \Theta)$ and $\text{Dishes}(\Delta, \Gamma, \Theta) \neq \emptyset$.

Proposition 12. *Let (Δ, Γ, Θ) is a cooking session. If Θ is sensible, then there is a substitution tuple Σ such that Σ is a mitigation for (Δ, Γ, Θ).*

Proof. Assume Θ is sensible. So there is a Δ' and Γ' such that $\bot \notin \text{Execute}(\Delta', \Gamma', \Theta)$ and $\text{Dishes}(\Delta', \Gamma', \Theta) \neq \emptyset$. So there is a substitution tuple Σ such that $\Delta\Sigma = \Delta'$. So Σ is a mitigation for (Δ, Γ, Θ). □

We now return to the issue of a substitution that can result in a final food item that has fundamentally changed key properties. We illustrate how we can view this as the need for a mitigation, and how it can be addressed using secondary substitution.

Example 43. Consider a recipe Δ for chicken supreme, where π is `chickensupreme`, α is `count(chicken(breast, skin_on), 2)`, and β is `count(can(butter_beans, 400 g), 2)`. Now consider the following integrity constraint.

$$\bot \leftarrow \texttt{onhand(prepared(chicken\_supreme)}, \tau) \wedge \neg \texttt{use(chicken)}$$

where `use(chicken)` is defined as holding when there is an onhand atom holding in the first interval for butter beans in some form. A mitigation could be to substitute `chicken_supreme` with `bean_supreme`.

Note, the substitution in the above example is more than changing the name of the dish. The name connects the output of the recipe to properties of that output. So for instance, the food item `chicken_supreme` has properties such as `meat_based_food`, and `chicken_dish`, whereas the food item `bean_supreme` has properties such as `vegetarian_dish`. Both would have properties such as `contains_cream` and `contains_garlic`.

7. Drivers for substitution

We consider four drivers for substitution as summarized in Table 1. These are similarity to missing item, similarity to original dish, minimal change to recipe, and improvement to final dish. We investigate these approaches in the following subsections.

Table 1
Four drivers for substitution in cooking.

Type of driver	Problem	Solution
Similarity to missing item	Unavailable item (ingredient or equipment) or action	Minimize change to ingredient or equipment by minimizing distance between original and its substitute
Similarity to original dish	Unavailable item (ingredient or equipment) or action	Do primary and secondary substitutions to produce a dish as similar as possible to original dish.
Minimal secondary change to recipe	Unavailable item (ingredient or equipment) or action	Minimize change to recipe by minimizing number of secondary substitutions
Improve specified properties of dish	Change final food output to improve its properties	Do primary and secondary substitutions to produce a dish satisfying specified properties.

7.1. Substitution for similarity to unavailable item or action

The following set is the set of candidates that are closest to the unavailable item (ingredient or equipment) or action. So an arbitrary member of this set is chosen as the primary substitution.

$$\mathsf{BestFirst}(\alpha, \Delta, \Gamma, d, \omega, \Theta) = \{\beta \in \mathsf{Substitutions}(\alpha, \Delta, \Gamma, d, \omega, \Theta) \mid \text{for all } \beta' \in \mathsf{Substitutions}(\alpha, \Delta, \Gamma, d, \omega, \Theta)\ d(\alpha, \beta) \leq d(\alpha, \beta')\}$$

Then any necessary secondary substitutions are undertaken to ensure consistency. So the substitution tuple $\langle \Pi_1, \Pi_2, \ldots, \Pi_k \rangle$ is a substitution for **similarity to missing item** α in Δ, where $\Pi_1 = [\alpha/\beta]$ and $\beta \in \mathsf{BestFirst}(\alpha, \Delta, \Gamma, d, \omega, \Theta)$ and $\langle \Pi_2, \ldots, \Pi_k \rangle$ is a minimal mitigation.

Example 44. We start with a cooking session (Δ, Γ, Θ) for cooking spaghetti where $\Delta = \{\phi\}$ where $\phi \in \Delta$ is defined as follows.

$$\begin{aligned}\phi = \quad & \texttt{onhand(cooked(spaghetti), T4)} \leftarrow \\ & \texttt{do(boil(spaghetti, 10 min), T3)} \\ & \wedge\ \texttt{do(put(spaghetti, in(pan)), T2)} \\ & \wedge\ \texttt{do(pour(boiling\_water, in(pan)), T1)} \\ & \wedge\ \texttt{sequence([T1, T2, T3, T4])}\end{aligned}$$

First we consider a primary substitution. Let $\alpha = \texttt{spaghetti}$ and let the nearest ingredient be $\beta = \texttt{linguine}$.

$$\begin{aligned}\phi[\alpha/\beta] = \quad & \texttt{onhand(cooked(linguine), T4)} \leftarrow \\ & \texttt{do(boil(linguine, 10 min), T3)} \\ & \wedge\ \texttt{do(put(linguine, in(pan)), T2)} \\ & \wedge\ \texttt{do(pour(boiling\_water, in(pan)), T1)} \\ & \wedge\ \texttt{sequence([T1, T2, T3, T4])}\end{aligned}$$

Now we consider a secondary substitution to deal with change in cooking time for the substituted pasta. Let $\alpha' = \texttt{do(boil(linguine, 10 min), T3)}$ and $\beta' = \texttt{do(boil(linguine, 12 min), T3)}$.

$$\begin{aligned}(\phi[\alpha/\beta])[\alpha'/\beta'] = \quad & \texttt{onhand(cooked(linguine), T4)} \leftarrow \\ & \texttt{do(boil(linguine, 12 min), T3)} \\ & \wedge\ \texttt{do(put(linguine, in(pan)), T2)} \\ & \wedge\ \texttt{do(pour(boiling\_water, in(pan)), T1)} \\ & \wedge\ \texttt{sequence([T1, T2, T3, T4])}\end{aligned}$$

So the substitution tuple for similarity to missing item is $\langle [\alpha/\beta], [\alpha'/\beta'] \rangle$.

Downsides of substitution for similarity to missing item include: (**Potentially unnecessary secondary substitutions**) there may be one or more secondary substitutions that would be unnecessary if we compromised on the primary substitution; And (**Potentially substantially different output**) the substitution tuple is only guaranteed to provide a final food output with a recipe that is consistent with the integrity constraints and properties, and this final output might be quite different to the original intended output of the recipe. We address these issue with other kinds of substitution in the next two subsections.

7.2. Substitution for minimal change to recipe

Making a poor choice of primary substitution may cause more significant changes to a recipe when doing mitigation. To address this, we may compromise on the closeness of the primary substitution in order to limit the number or type of secondary substitutions.

Let $\Delta[\alpha/\beta]$ be the result of the primary substitution on the recipe. Let $\mathsf{Secondary}(\Delta, \alpha, \beta, \Theta)$ be the set of minimal secondary substitutions (i.e. each element is a minimal mitigation for $(\Delta[\alpha/\beta], \Gamma', \Theta)$ where Γ' is a set of assumptions) as defined below. So a minimal change to a recipe is the concatenation of the primary substitution and secondary substitution that is shortest, as defined next.

Definition 18. For a cooking session (Δ, Γ, Θ),

1. if $[\alpha/\beta] \in \text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega, \Theta)$
2. and $\Sigma \in \text{Secondary}(\Delta, \alpha, \beta, \Theta)$
3. and for all $[\alpha/\beta'] \in \text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega, \Theta)$ and for all $\Sigma' \in \text{Secondary}(\Delta, \alpha, \beta', \Theta)$,

$$\Sigma \text{ is shorter than } \Sigma'$$

then a **minimal change** to a recipe Δ is the substitution tuple $\langle \Pi_1, \ldots, \Pi_k \rangle$ where Π_1 is $[\alpha/\beta]$ and $\langle \Pi_2, \ldots, \Pi_k \rangle$ is Σ.

Example 45. Continuing Example 44, suppose the distance between `spaghetti` and `farfalle` is higher than between `spaghetti` and `linguine` but also suppose that the cooking time for `farfalle` is the same as `spaghetti`. So we have α = `spaghetti` and β = `farfalle` where

- $[\alpha/\beta] \in \text{Substitutions}(\alpha, \Delta, \Gamma, d, \omega, \Theta)$
- $\Sigma = \langle \rangle$ is the minimal mitigation.

Therefore, the minimal change to the recipe is $\langle [\alpha/\beta] \rangle$. In contrast, for β' = `fusilli`, any minimal mitigation is not equal to the empty tuple.

Whilst the above example would result in a dish that involves a minor compromise in the final dish, in general, trying to substitute an ingredient and then minimize the change to the recipe might result in a dish that is somewhat distant from the original dish. For example, we may lack a beef patty for a beefburger, and we may have the ingredients to make a beef patty (minced meet, chopped onion, egg, etc.) and we may also have a bean burger patty. If we wanted a substitution that minimizes the change to the recipe, then we would pick the bean burger patty, whereas we may prefer to make a more substantial change to the recipe to allow for making the beef burger patty. We consider how we might get the latter using the next method.

7.3. Substitution for similarity to original dish

In order to choose a sequence of substitutions that ensures similarity to an intended original dish π, we use integrity constraints that capture when a substitution causes the revised dish to deviate from the original dish. For example, if we substitute beans for beef in a burger, we lose the property that the burger is a meat dish, or that it has a meaty flavour. So we represent this as an integrity constraint as in Example 46.

When we have a violation of an integrity constraint that captures the deviation from the original intended dish, we have two options: (**rectification by change of dish name**) which involves taking the original name π of the intended dish and replacing it with a name π' that is consistent with the integrity constraints concerning the properties of the dish (e.g. changing the name from `beef_burger` to `bean_burger` when the revised recipe does not have meat); or (**rectification by change of recipe**) which involves changing the recipe so that it meats the basic requirements of the intended dish (e.g. for the intended dish `meat_burger`, changing the recipe so that it contains meat or something meat-flavoured). We capture both kinds of rectification in the following definition for substitution for similarity to original.

Definition 19. Let (Δ, Γ, Θ) be a cooking session for dish π and let $[\alpha/\beta]$ be a primary substitution. A substitution for **similarity to original dish** is a substitution tuple $\Sigma = \langle \Pi_1, \ldots, \Pi_k \rangle$ for ingredient substitution $\Pi_1 = [\alpha/\beta]$ and mitigation $\langle \Pi_2, \ldots, \Pi_k \rangle$ and output $\pi' \in \text{Dishes}(\Delta, \Gamma, \Theta)$ and distance measure d over dishes and threshold for similarity for dishes λ:

1. $\bot \notin \text{Execute}(\Delta\Sigma, \Gamma', \Theta)$
2. $\pi' \in \text{Dishes}(\Delta\Sigma, \Gamma', \Theta)$
3. $d(\pi, \pi') \leq \lambda$

In the above definition, either there is a π' in the final outcome where the distance between π and π' is below a threshold (rectification by change of name) or Σ modifies the recipe so that π is in the final output and $d(\pi, \pi) \leq \lambda$ for all λ (rectification by change of recipe). We illustrate the former in the first example below, and the latter in the subsequent two examples below.

Example 46. Consider a recipe Δ for a burger that involves cooking a beef patty and a bun on a griddle, and then putting the beef patty in the bun.

$$\begin{aligned}\phi = \ & \texttt{onhand(beef\_burger, 1), T7)} \\ & \leftarrow \texttt{do(put(beef\_patty, in(bun)), T6)} \\ & \wedge \texttt{do(grill(beef\_patty, 10 min), T3, T5)} \\ & \wedge \texttt{do(grill(bun, 2 min)), T2, T4)} \\ & \wedge \texttt{onhand(count(bun, 1), T1)} \\ & \wedge \texttt{onhand(count(beef\_patty, 1), T1)} \\ & \wedge \texttt{sequence([T1, T2, T3, T4, T5, T6, T7])}\end{aligned}$$

Now let $\alpha = \texttt{beef\_patty}$ and $\beta = \texttt{bean\_patty}$, and so we replace the beef patty with a bean patty. In this, the recipe $\Delta[\alpha/\beta]$ would result in $\bot \in \text{Execute}(\Delta[\alpha/\beta], \Gamma[\emptyset/\emptyset], \Theta)$ when Θ contains the following integrity constraint.

$$\bot \leftarrow \texttt{flavour}(\texttt{beef\_burger}, \texttt{meaty}) \wedge \neg\texttt{contains}(\texttt{bean\_burger}, \texttt{meat})$$

So the mitigation is to change the name to bean_burger (i.e. [beef_burger/bean_burger]). So we can use rectification by change of dish name. So the revised recipe is

$$\begin{aligned}\phi = \texttt{onhand}(\texttt{bean\_burger}, 1), \texttt{T7})\\ \leftarrow \texttt{do}(\texttt{put}(\texttt{bean\_patty}, \texttt{in}(\texttt{bun})), \texttt{T6})\\ \wedge\ \texttt{do}(\texttt{grill}(\texttt{bean\_patty}, 10\ \texttt{min}), \texttt{T3}, \texttt{T5})\\ \wedge\ \texttt{do}(\texttt{grill}(\texttt{bun}, 2\ \texttt{min}), \texttt{T2}, \texttt{T4})\\ \wedge\ \texttt{onhand}(\texttt{count}(\texttt{bun}, 1), \texttt{T1})\\ \wedge\ \texttt{onhand}(\texttt{count}(\texttt{bean\_patty}, 1), \texttt{T1})\\ \wedge\ \texttt{sequence}([\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}, \texttt{T6}, \texttt{T7}])\end{aligned}$$

Example 47. We revisit Example 46 concerning a recipe Δ for a burger that involves cooking a beef patty on a griddle. In contrast to Example 46 where the output is a bean_burger, we can add the following clause ϕ_2 to Δ

$$\begin{aligned}\phi = \texttt{onhand}(\texttt{count}(\texttt{beef\_patty}, 1), \texttt{T6})\\ \leftarrow \texttt{do}(\texttt{mix}(\texttt{bowl}), \texttt{T5})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{minced\_meat}, \texttt{in}(\texttt{bowl})), \texttt{T4})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{chopped\_onion}, \texttt{in}(\texttt{bowl})), \texttt{T3})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{egg}, \texttt{in}(\texttt{bowl})), \texttt{T2})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{pepper}, \texttt{in}(\texttt{bowl})), \texttt{T1})\\ \wedge\ \texttt{sequence}([\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}, \texttt{T6}])\end{aligned}$$

Therefore, $\Pi = [\emptyset/\{\phi_2\}]$, and so the resulting cooking session is $(\Delta\Pi, \Gamma', \Theta)$ for some set of assumptions Γ'. This solution would be obviate the inconsistency with the integrity constraint given in Example 46.

As we saw above, a primary substitution can change the properties of the intermediate (product). For example, if we substitute meat with tofu in a burger, the result could be less flavourful (which can be identified by an integrity constraints as above). From a culinary point of view, a solution is to first cook tofu in mushroom sauce.

Example 48. Consider a similar example to that of Example 46 concerning a recipe Δ for a burger that involves cooking a beef patty on a griddle. Here, the intended output from clause ϕ_1 below is burger_patty rather than beef_patty.

$$\begin{aligned}\texttt{onhand}(\texttt{count}(\texttt{burger\_patty}, 1), \texttt{T6})\\ \leftarrow \texttt{do}(\texttt{mix}(\texttt{bowl}), \texttt{T5})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{minced\_meat}, \texttt{in}(\texttt{bowl})), \texttt{T4})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{chopped\_onion}, \texttt{in}(\texttt{bowl})), \texttt{T3})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{egg}, \texttt{in}(\texttt{bowl})), \texttt{T2})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{pepper}, \texttt{in}(\texttt{bowl})), \texttt{T1})\\ \wedge\ \texttt{sequence}([\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}, \texttt{T6}])\end{aligned}$$

And suppose we want to make the primary substitution that replaces minced_meat with tofu. However, if we may also have the following integrity constraint, then we obtain an inconsistency

$$\bot \leftarrow \texttt{flavour}(\texttt{burger\_patty}, \texttt{meaty}) \wedge \neg\texttt{contains}(\texttt{burger\_burger}, \texttt{meat}) \wedge \neg\texttt{contains}(\texttt{burger\_burger}, \texttt{mushroom})$$

To ensure that the burger has a meaty taste, we could first marinade the tofu in mushroom. For this we could add the following clause ϕ_2 to Δ using clause substitution;

$$\begin{aligned}\texttt{onhand}(\texttt{mushroom\_flavoured\_tofu}), \texttt{T6})\\ \leftarrow \texttt{do}(\texttt{fry}(\texttt{frying\_pan}, 10\ \texttt{min}), \texttt{T5})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{tofu}, \texttt{in}(\texttt{frying\_pan})), \texttt{T4})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{mushrooms}, \texttt{in}(\texttt{frying\_pan})), \texttt{T4})\\ \wedge\ \texttt{sequence}([\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}, \texttt{T6}])\end{aligned}$$

And then update the clause ϕ_1 that resulted from the primary substitution as follows using condition substitution where the condition $\texttt{do}(\texttt{put}(\texttt{egg}, \texttt{in}(\texttt{bowl})), \texttt{T2})$ is removed and the condition $\texttt{do}(\texttt{put}(\texttt{mushroom\_flavoured\_tofu}, \texttt{in}(\texttt{bowl})), \texttt{T4})$ is added.

$$\begin{aligned}\texttt{onhand}(\texttt{count}(\texttt{burger\_patty}, 1), \texttt{T6})\\ \leftarrow \texttt{do}(\texttt{mix}(\texttt{bowl}), \texttt{T5})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{mushroom\_flavoured\_tofu}, \texttt{in}(\texttt{bowl})), \texttt{T4})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{chopped\_onion}, \texttt{in}(\texttt{bowl})), \texttt{T3})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{milk}, \texttt{in}(\texttt{bowl})), \texttt{T2})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{flour}, \texttt{in}(\texttt{bowl})), \texttt{T1})\\ \wedge\ \texttt{do}(\texttt{put}(\texttt{pepper}, \texttt{in}(\texttt{bowl})), \texttt{T1})\\ \wedge\ \texttt{sequence}([\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}, \texttt{T6}])\end{aligned}$$

So instead of using tofu, we use mushroom_flavoured_tofu, and provide a clause for producing this food item. The resulting recipe does not violate the integrity constraint.

In each of the above examples, we need to ensure that the distance between the intended dish π and the dish obtained by the revised recipe π' are sufficiently close. For Example 46, whether $d(\texttt{beef\_burger}, \texttt{bean\_burger}) \leq \lambda$ depends on the specification of the distance measure d and the subjective choice for λ. For Example 47, the intended dish and the dish obtained by the revised recipe are the same (i.e. beef_burger) and so the distance is zero. For Example 48, the intended dish and the dish obtained by the revised recipe are the same (i.e. burger_patty) and so the distance is zero, but we have seen how the integrity constraint can force the refinement of the recipe from using the ingredient tofu directly by updating the recipe to include a step for making mushroom_flavoured_tofu.

7.4. Substitution for improvement to dish

We now consider substitution to meet specific requirements. For example, changing a recipe so that it is vegetarian, gluten-free, or reduced calories. We will assume that we are dealing with binary categories. For example, a recipe for 200 g of digestive biscuits has the property of reduced calories if the number of calories is below 300 calories. Or we can reduce or remove salt from a recipe with a property that salt level is below a certain level.

In order to represent and reason with properties, we introduce a **property atom** as an atom of the form $\texttt{property}(\alpha, \beta)$ where α is a term denoting a food item, and β is a property of the food item. For example, $\texttt{property}(\texttt{porridge}, \texttt{gluten\_free})$ denotes that porridge is gluten-free. We define a **property clause** as a clause ϕ where $\mathsf{head}(\phi)$ is a property atom. The following is an example of a property clause.

$$\begin{aligned}&\texttt{property}(\texttt{biscuit\_mix}, \texttt{gluten\_free})\\ &\qquad\leftarrow \texttt{contains}(\texttt{biscuit}, \texttt{oat\_flour})\\ &\qquad\wedge \neg\texttt{contains}(\texttt{biscuit}, \texttt{plain\_flour})\end{aligned}$$

We use property atoms and property clauses in the following definition where we use substitutions to ensure that a property holds.

Definition 20. Let (Δ, Γ, Θ) be a cooking session. Let π be an intended dish. Let ψ be a property literal. Let Γ' be a set of assumptions. A substitution for improving a dish according to ψ is a substitution tuple $\Sigma = \langle \Pi_1, \ldots, \Pi_k \rangle$ such that

1. $\bot \notin \mathsf{Execute}(\Delta\Sigma, \Gamma' \cup \{\phi\}, \Theta)$
2. $\pi \in \mathsf{Dishes}(\Delta\Sigma, \Gamma', \Theta)$
3. $\psi \in \mathsf{Execute}(\Delta\Sigma, \Gamma', \Theta)$

Example 49. Consider a recipe which includes the following process clause that uses wheat flour.

$$\begin{aligned}&\texttt{onhand}(\texttt{biscuit\_mix}, \texttt{T6})\\ &\qquad\leftarrow \texttt{do}(\texttt{mix}(\texttt{bowl}), \texttt{T5})\\ &\qquad\wedge \texttt{do}(\texttt{add}(\texttt{measure}(\texttt{white\_caster\_sugar}, 200, \texttt{g}), \texttt{in}(\texttt{bowl})), \texttt{T4})\\ &\qquad\wedge \texttt{do}(\texttt{add}(\texttt{measure}(\texttt{unsalted\_butter}, 200, \texttt{g}), \texttt{in}(\texttt{bowl})), \texttt{T3})\\ &\qquad\wedge \texttt{do}(\texttt{add}(\texttt{count}(\texttt{egg}, 1), \texttt{in}(\texttt{bowl})), \texttt{T2})\\ &\qquad\wedge \texttt{do}(\texttt{add}(\texttt{measure}(\texttt{plain\_flour}, 400, \texttt{g}), \texttt{in}(\texttt{bowl})), \texttt{T1})\\ &\qquad\wedge \texttt{sequence}([\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}, \texttt{T6}])\end{aligned}$$

Suppose we have a Γ such that biscuit_mix is in $\mathsf{Execute}(\Delta, \Gamma, \Theta)$. If we also have the following clause in Δ, then we would get $\neg\texttt{property}(\texttt{biscuit\_mix}, \texttt{gluten\_free})$ in $\mathsf{Execute}(\Delta, \Gamma, \Theta)$.

$$\begin{aligned}&\neg\texttt{property}(\texttt{biscuit\_mix}, \texttt{gluten\_free})\\ &\qquad\leftarrow \texttt{contains}(\texttt{biscuit}, \texttt{plain\_flour})\\ &\qquad\wedge \neg\texttt{contains}(\texttt{biscuit}, \texttt{oat\_flour})\end{aligned}$$

Now suppose we make a substitution in the process clause where α is plain_flour and β is oat_flour. Then we would get $\texttt{property}(\texttt{biscuit\_mix}, \texttt{gluten\_free})$ in $\mathsf{Execute}(\Delta, \Gamma, \Theta)$ if we have the following clause in Θ.

$$\begin{aligned}&\texttt{property}(\texttt{biscuit\_mix}, \texttt{gluten\_free})\\ &\qquad\leftarrow \texttt{contains}(\texttt{biscuit}, \texttt{oat\_flour})\\ &\qquad\wedge \neg\texttt{contains}(\texttt{biscuit}, \texttt{plain\_flour})\end{aligned}$$

Example 50. Continuing Example 49, suppose we also want that biscuits are low fat, then we might have the property $\texttt{property}(\texttt{measure}(\texttt{biscuit\_mix}, 800\ \texttt{g}), \texttt{low\_calorie})$ defined by the following clause.

$$\begin{aligned}&\texttt{property}(\texttt{measure}(\texttt{biscuit\_mix}, 800, \texttt{g}), \texttt{low\_calorie})\\ &\qquad\leftarrow \texttt{contains}(\texttt{measure}(\texttt{biscuit\_mix}, 800, \texttt{g}), \texttt{measure}(\texttt{butter}, \texttt{X}))\\ &\qquad\wedge \texttt{lessthan}(\texttt{X}, 100, \texttt{g})\end{aligned}$$

The property can be satisfied with a term substitution of measure(unsalted_butter, 200, g) by measure(unsalted_butter, 100, g) giving the following updated version of the process clause.

onhand(biscuit_mix, T6)
← do(mix(bowl), T5)
∧ do(add(measure(white_caster_sugar, 200, g), in(bowl)), T4)
∧ do(add(measure(unsalted_butter, 100, g), in(bowl)), T3)
∧ do(add(count(egg, 1), in(bowl)), T2)
∧ do(add(measure(plain_flour, 400, g), in(bowl)), T1)
∧ sequence([T1, T2, T3, T4, T5, T6])

This method assumes that we turn continuous improvements into binary improvements. Also, the method does not take into account the trade-off of realizing improvements and minimizing changes. However, we could combine this method with previous methods to be able to take into account secondary substitutions with the aim of satisfying desired properties and minimizing secondary changes.

7.5. Comparison of approaches to substitution

The four drivers for substitution methods (as summarized in Table 1) have lead to methods that can be compared pairwise to show there are situations where each gives different results to the others.

Similarity to missing item *versus* minimal change to recipe These give different substitutions when the former picks the nearest ingredient or equipment to the missing item, whereas the latter picks the ingredient or equipment that requires a shorter mitigation. For example, if there are two choices β_1 and β_2 for the missing item α where $d(\alpha, \beta_1) < d(\alpha, \beta_2)$, and if we choose β_1 then we require the minimal mitigation Σ_1 and if we choose β_2 then we require the minimal mitigation Σ_2 where Σ_2 is shorter than Σ_1. So the similarity to missing item method picks β_1 whereas the minimal change to recipe method picks β_2.

Similarity to missing item *versus* similarity to original dish These give different substitutions when the former picks the nearest ingredient or equipment to the missing item, whereas the latter picks the ingredient or equipment that together with any mitigation results in a dish that is nearest to the original dish. For example, if there are two choices β_1 and β_2 for the missing item α where $d(\alpha, \beta_1) < d(\alpha, \beta_2)$, and if we choose β_1 then with the minimal mitigation Σ_1, we get dish π_1, and if we choose β_2 then with the minimal mitigation Σ_2, we get dish π_2, where $d(\pi, \pi_1) > d(\pi, \pi_2)$. So the similarity to missing item method picks β_1 whereas the similarity to original dish method picks β_2.

Similarity to missing item *versus* improve specified properties of dish These give different substitutions when the former picks the nearest ingredient or equipment to the missing item, whereas the latter picks the ingredient or equipment that as part of a substitution sequence leads to a dish with required properties. For example, if there are two choices β_1 and β_2 for the missing item α where $d(\alpha, \beta_1) < d(\alpha, \beta_2)$, and if we choose β_1 then with the minimal mitigation Σ_1, but we get a cooking session that does not satisfy the desired property ϕ, whereas if we choose β_2 then with the minimal mitigation Σ_2, we get a cooking session that does satisfy the desired property ϕ. So the similarity to missing item method picks β_1 whereas the improvement to dish method picks β_2.

Minimal secondary changes *versus* similarity to original dish These give different substitutions when the former picks the ingredient or equipment that requires a shorter mitigation, whereas the latter picks the ingredient or equipment that together with any mitigation results in a dish that is nearest to the original dish. For example, if there are two choices β_1 and β_2 for the missing item α, and if we choose β_1 then we require the minimal mitigation Σ_1 and if we choose β_2 then we require the minimal mitigation Σ_2 where Σ_1 is shorter than Σ_2. Also, if we choose β_1 then with the minimal mitigation Σ_1, we get dish π_1, and if we choose β_2 then with the minimal mitigation Σ_2, we get dish π_2, where $d(\pi, \pi_1) > d(\pi, \pi_2)$. So the minimal change to recipe method picks β_1, whereas the similarity to original dish method picks β_2.

Minimal secondary changes *versus* improve specified properties of dish These give different substitutions when the former picks the ingredient or equipment that requires a shorter mitigation, whereas the latter picks the ingredient or equipment that as part of a substitution sequence leads to a dish with required properties. For example, if there are two choices β_1 and β_2 for the missing item α, and if we choose β_1 then we require the minimal mitigation Σ_1, but we get a cooking session that does not satisfy the desired property ϕ, and if we choose β_2 then we require the minimal mitigation Σ_2, but we get a cooking session that does satisfy the desired property ϕ. Also suppose that Σ_1 is shorter than Σ_2. So the minimal change to recipe method picks β_1, whereas the improvement to dish method picks β_2.

Similarity to original dish *versus* improve specified properties of dish The former picks the ingredient or equipment that together with any mitigation results in a dish that is nearest to the original dish, whereas the latter picks the ingredient or equipment that as part of substitution sequence leads to a dish with required properties. For example, if there are two choices β_1 and β_2 for the missing item α, and if we choose β_1 then we require the minimal mitigation Σ_1, and we get dish π_1, but we get a cooking session that does not satisfy the desired property ϕ, and if we choose β_2 then we require the minimal mitigation Σ_2, and we get dish π_2, but we get a cooking session that does satisfy the desired property ϕ. Also suppose $d(\pi, \pi_1) < d(\pi, \pi_2)$. So the similarity to original dish method picks β_1, whereas the improvement to dish method picks β_2.

So the four methods give us a range of different options for how to do substitution. Therefore, it is necessary to know the driver for substitution for deciding which method to use as we can clearly see that the resulting recipe can be quite different to the original recipe.

8. Inconsistency-directed substitution

In this paper, we have presented secondary substitutions as being identified via inconsistencies with integrity constraints. In this section we consider types of integrity constraints, and therefore types of situation that require mitigation following primary substitution, and we consider how we can reason and resolve inconsistencies efficiently. Note, we could also make primary substitution inconsistency-directed. For this, we could include integrity constraints for the ingredients and equipment, and so if any are missing, then we infer an inconsistency. We leave consideration of inconsistency-directed primary substitutions and instead focus on inconsistency-directed secondary substitutions in this section.

8.1. Types of integrity constraints

Integrity constraints are central to identifying the need for mitigation. These integrity constraints conform to specific formats, and importantly, they may be generated automatically. We start by considering the following typology of integrity constraints, and then consider how we can generate them automatically.

Action duration Many actions have a specified duration (fixed or range). For example, the following is a duration range term $\texttt{boil(penne, 12 - 14 min)}$. However, it is possible that the specified duration differs from what is recorded in the knowledge graph about the ingredients and actions on them. For example, we may have the condition $\texttt{do(boil(penne, 2 min), T)}$ in a process clause, whereas the knowledge graph may have the tuple $\texttt{penne, boil\_time, 12 - 14 min}$. So we have the integrity constraint.

$$\bot \leftarrow \texttt{do(boil(penne, D), T)} \wedge \texttt{D} \neq \texttt{12 - 14 min}$$

Onhand quantity Many onhand atoms involve either count or measure terms. For example, $\texttt{onhand(count(eggs, 4), T)}$. So if we can infer multiple onhand atoms with the same food item but with a different count, then we have inconsistency.

$$\bot \leftarrow \texttt{onhand(count(X, Y), T)} \wedge \texttt{onhand(count(X, Z), T)} \wedge \texttt{Y} \neq \texttt{Z}$$

Similarly, if we can infer multiple onhand atoms for the same food item but with a different count, then we have inconsistency.

$$\bot \leftarrow \texttt{onhand(measure(X, Y), T)} \wedge \texttt{onhand(measure(X, Z), T)} \wedge \texttt{Y} \neq \texttt{Z}$$

The use of the above measure integrity constraints assumes that we have sufficient background knowledge to deal with different units.

Consume quantity As with onhand count/measure integrity constraints, we require integrity constraints to ensure the same food item is not consumed in different quantities at the same time. For example, the atoms $\texttt{consume(count(eggs, 4), t)}$ and $\texttt{consume(count(eggs, 3), t)}$ would imply an inconsistency.

$$\bot \leftarrow \texttt{consume(count(X, Y), T)} \wedge \texttt{consume(count(X, Z), T)} \wedge \texttt{Y} \neq \texttt{Z}$$

Consume availability If a consume atom specifies that a certain quantity of a food item is required, but the onhand atom specifies a different quantity, then we have an inconsistency.

$$\bot \leftarrow \texttt{consume(count(X, Y), T)} \wedge \texttt{onhand(count(X, Z), T)} \wedge \texttt{Y} \neq \texttt{Z}$$

Onhand location Onhand atoms can incorporate location features. For example, the onhand atom $\texttt{onhand(in(eggs, bowl), T)}$ specifies that the eggs are in the bowl. Various kinds of constraints are required to ensure that items are in the correct location (i.e conform to the change of location that follow from an action), and not in two places at the same time.

$$\bot \leftarrow \texttt{onhand(in(X, Y), T + 1)} \wedge \texttt{onhand(in(X, Y), T)} \wedge \texttt{do(pour(X, Z), T)} \wedge \texttt{Y} \neq \texttt{Z}$$

Parallel actions We can use integrity constraints to ensure that there is at most one action undertaken at each point in time.

$$\bot \leftarrow \texttt{do(X, T1, T2)} \wedge \texttt{do(Y, T3, T4)} \wedge \texttt{X} \neq \texttt{Y} \wedge \texttt{before(T1, T3)} \wedge \texttt{before(T3, T2)}$$

Preparation The following integrity constraint then identifies that we cannot have $\texttt{sliced\_carrot}$ if we do not have the action $\texttt{do(slice(carrot)}$ and we did not start with $\texttt{sliced\_carrot}$ as an initial ingredient. Here, we have assumed an extra kind of atom for the ingredients.

$$\begin{aligned}\bot \leftarrow\ & \texttt{onhand(sliced\_carrot, T3)} \\ & \wedge \neg\texttt{do(slice(carrot), T2)} \\ & \wedge \neg\texttt{ingredient(sliced\_carrot, T1)} \\ & \wedge \texttt{before(T1, T2)} \wedge \texttt{before(T2, T3)}\end{aligned}$$

Equipment For simple equipment, we may assume that we have an unlimited supply (i.e. more than would be required for any normal recipe). So we have many plates, bowls, etc. However, some equipment will be limited. For example, we might assume that we have one oven, one electric mixer, one bread making machine, one microwave, etc. So we may need constraints on actions that would use the same equipment for multiple tasks.

$$\begin{aligned}\bot \leftarrow\ & \texttt{do}(\texttt{mix\_with\_electric\_mixer}(\texttt{X},\texttt{D}),\texttt{T}) \\ & \wedge\ \texttt{do}(\texttt{mix\_with\_electric\_mixer}(\texttt{Y},\texttt{D}),\texttt{T}) \\ & \wedge\ \texttt{X} \neq \texttt{Y}\end{aligned}$$

We can define finer grained constraints to deal with more complex situations such as ovens where multiple items can go in if use the same temperature. So it two items are in the oven at the same time and they need different temperatures, then there is an inconsistency.

$$\begin{aligned}\bot \leftarrow\ & \texttt{do}(\texttt{put\_in\_oven}(\texttt{X},\texttt{H1},\texttt{D1}),\texttt{T1},\texttt{T2}) \\ & \wedge\ \texttt{do}(\texttt{put\_in\_oven}(\texttt{Y},\texttt{H2},\texttt{D2}),\texttt{T3},\texttt{T4}) \\ & \wedge\ \texttt{X} \neq \texttt{Y} \wedge \texttt{H1} \neq \texttt{H2} \wedge \texttt{before}(\texttt{T1},\texttt{T3}) \wedge \texttt{before}(\texttt{T2},\texttt{T3})\end{aligned}$$

Properties of food items Specific food items have specific properties. If we change the ingredients used to prepare a food item or we change the cooking steps, then we may change some of those specific properties. For instance, if we do not use meat in the preparation of a beef burger, then it may lose the property of having a meaty flavour.

$$\bot \leftarrow \texttt{flavour}(\texttt{beef\_burger},\texttt{beefy}) \wedge \neg\texttt{contains}(\texttt{beef\_burger},\texttt{beef})$$

As another example, if we do not use chicken in the preparation of chicken supreme, then the name is incorrect. This is captured by the following integrity constraint.

$$\bot \leftarrow \texttt{onhand}(\texttt{prepared}(\texttt{chicken\_supreme}),\tau) \wedge \neg\texttt{use}(\texttt{chicken})$$

In the next subsection, we will consider how we can group these different types of integrity constraint according to how we obtain them.

8.2. Generating integrity constraints

We group the integrity constraints in the previous section into three groups where each group indicates how we can obtain them.

Fixed group For this group, there is a fixed number of the integrity constraints irrespective of the language (i.e. the number of actions, ingredients, etc.). For instance, if we adopt the onhand integrity constraints, then there be two of them irrespective of the actions, ingredients, and equipment. Similarly, if we adopt the consume quantity integrity constraint, consume availability integrity constraint, and parallel actions integrity constraint, then there would be one each of them.

Pattern-based group For this group, there is a schema for the integrity constraint, and that this schema can be instantiated with the actions, equipment, ingredients, etc in the language. Members of this group include action duration integrity constraints, onhand location integrity constraints, preparation integrity constraints, and equipment integrity constraints. For each type of schema, the number of integrity constraints can be calculated. For instance, for the schema for the equipment integrity constraints, the number of these integrity constraints will equal the number of items of equipment in the language since each item of equipment gives an integrity constraint. Similarly, for the schema for the action duration integrity constraints, the number of these integrity constraints will equal the product of the number of items of actions, and the number of food items, in the language.

General group For this group, the format depends on the kind of knowledge available. Some of them could be obtained by a transformation from knowledge graphs as we explain below. Members of this group include onhand location integrity constraints and properties of food items integrity constraints. Furthermore, for this type of integrity constraint, there is a potentially open-ended number of them.

So some types of integrity constraint (namely, onhand quantity, consume quantity, consume availability, multiple keys, and parallel actions) are either as specified in the previous section or they can be adapted from them, whereas some types of integrity constraint are more challenging to develop as there may be a large number of them, and there may be a need to have adequate knowledge graphs, as well as methods for automatically generating them (action duration, onhand location, preparation, and properties of food items).

Example 51. Consider the example of an action duration integrity constraint given in the previous subsection. Assuming, we have a knowledge graph with an appropriate structure, we can automatically generate such integrity constraints. For instance, if we have relations of the form `cooking_duration` and `cooking_style`, and these appear in the following knowledgegraph triples, then we can generate the atoms $\texttt{do}(\texttt{boil}(\texttt{penne},\texttt{D}),\texttt{T})$ and $12-14$ `min`, and hence obtain the integrity constraint.

$$\begin{aligned}&(\texttt{penne},\texttt{cooking\_style},\texttt{boil}) \\ &(\texttt{penne},\texttt{cooking\_duration},12-14\ \texttt{min})\end{aligned}$$

We do not specify the structure of the knowledge graph here as this as is an open-ended task. However, we may assume that a knowledge graph has relations that capture knowledge including: **cooking styles** for an ingredient (e.g. `boil` for `pasta`, `toast` for `bread`); **cooking times** for an ingredient (e.g. `10-12 min` for `fusilli`); **result of an action** on an ingredient or intermediate food item (e.g. `sliced_carrot` is the result of the action `slice` on the ingredient `carrot` or `dough` is the result of the action `mix` on the ingredients `flour`, `yeast`, and `water`); **locations of ingredients** after actions (e.g. location of ingredient `salt` after action `pour` qualified by prepositional phrase `in_bowl` is `bowl`); **equipment availability** which provides local context constraints about equipment (e.g. the number of available `bread_making_machine` is 1, `microwave` is 0, and `mixing_bowl` is 4); **key ingredients** for a dish for example, `beef` and `bap` are key ingredients for the dish `beef_burger`); **descriptors** for ingredients and dishes (for example, the dish `beef_burger` has the type of descriptor `flavour` with the parameter `meaty`, the ingredient `honey` has the descriptor `flavour` with the parameter `sweet`, the dish `biscuit` with the parameter `texture` has the parameter `crumbly`, and the ingredient `oat_flour` with the parameter `allergy_information` has the parameter `gluten_free`). More general knowledge representation formalisms than a knowledge graph may also be required for generating integrity constraints concerning properties of food.

8.3. Localizing issues via minimal inconsistencies

Inconsistencies are central to identification of the need of mitigation, and furthermore, they point to the kind of mitigation. For example, if we have an inconsistency involving an integrity constraint of the action duration type, then we can try to fix the inconsistency by changing the duration of the action.

Definition 21. Let (Δ, Γ, Θ) be a cooking session. A **minimal inconsistent subset** of a cooking session is $\Phi \subseteq (\Delta \cup \Gamma \cup \Theta)$ such that $\Phi \vdash \bot$, and for all $\Phi' \subset \Phi$, $\Phi' \nvdash \bot$. Let $\mathsf{MinInc}(\Delta, \Gamma, \Theta)$ be the set of minimal inconsistent subsets of (Δ, Γ, Θ).

Example 52. Consider a cooking session (Δ, Γ, Θ) for a multi-layer cake with different cake mixes for each layer. Let Γ contain the following two atoms.

$$\texttt{do(mix\_with\_electric\_mixer(carrot\_cake\_mix, 4 min), t13, t15)}$$
$$\texttt{do(mix\_with\_electric\_mixer(sponge\_cake\_mix, 3 min), t12, t14)}$$

Let the following clause be in Θ. Hence, there is a $\Phi \in \mathsf{MinInc}(\Delta, \Gamma, \Theta)$ that contains exactly these three formulae.

$$\begin{aligned}\bot \leftarrow\ & \texttt{do(mix\_with\_electric\_mixer(X, D), T1, T2)}\\ & \wedge\ \texttt{do(mix\_with\_electric\_mixer(Y, D), T3, T4)}\\ & \wedge\ \texttt{X} \neq \texttt{Y} \wedge \texttt{before(T1, T3)} \wedge \texttt{before(T3, T2)}\end{aligned}$$

Proposition 13. *Let (Δ, Γ, Θ) be a cooking session. For each $\Phi \in \mathsf{MinInc}(\Delta, \Gamma, \Theta)$, $|\Phi \cap \Theta| = 1$.*

Proof. For $\Phi \in \mathsf{MinInc}(\Delta, \Gamma, \Theta)$, $(\Delta, \Gamma, \Theta) \vdash \bot$. Therefore, there is a $\phi \in \Phi$ such that $\mathsf{Head}(\phi) = \bot$ and for all $\psi \in \mathsf{Tail}(\phi)$, $(\Delta, \Gamma, \Theta) \vdash \psi$. Since there is no $\Phi' \subset \Phi$ such that $\Phi' \vdash \bot$, there is no $\phi' \in \Phi$ such that $\mathsf{Head}(\phi') = \bot$ and $\phi \neq \phi'$. Therefore, $|\Phi \cap \Theta| = 1$. □

Given the above result, we can classify inconsistencies according to the type of integrity constraint. In addition, inconsistencies can arise because the assumptions are problematic or because the recipe is problematic.

Recipe inconsistency: A recipe inconsistency arises whenever there are sufficient assumptions to give the intended dish π there is also an inconsistency just involving Δ. In this case, for all Γ, if $\pi \in \mathsf{Execute}(\Delta, \Gamma, \Theta)$, then $\bot \in \mathsf{Execute}(\Delta, \Gamma, \Theta)$. So the cause of the inconsistency is in Δ.

Process inconsistency: A process inconsistency arises whenever there are sufficient assumptions to give the intended dish π there is also an inconsistency involving an integrity constraint. In this case, for all Γ, if $\pi \in \mathsf{Execute}(\Delta, \Gamma, \Theta)$, then $\bot \in \mathsf{Execute}(\Delta, \Gamma, \Theta)$ and $\bot \notin \mathsf{Execute}(\Delta, \Gamma, \emptyset)$. So the cause of the inconsistency is between Δ and Θ.

Assumption inconsistency: This is an inconsistency involving the assumptions and so the inconsistency is for a specific cooking session (Δ, Γ, Θ). If $\bot \in \mathsf{Execute}(\Delta, \Gamma, \Theta)$ and there is a Γ' such that $\pi \in \mathsf{Execute}(\Delta, \Gamma', \Theta)$, and $\bot \notin \mathsf{Execute}(\Delta, \Gamma', \Theta)$. So whilst the session is inconsistency, there exists a set of assumptions, for which the intended item is produced and the session is consistent.

If a cooking session is inconsistent, there is always a substitution that makes it consistent and still allows the intended food item to be inferred. So any of the above three types of inconsistency can be fixed. The simple way to obtain this is to delete clauses using clause substitution, or alternatively add conditions to clauses using condition substitution.

Proposition 14. *Let (Δ, Γ, Θ) be a cooking session for π. If $\bot \in \mathsf{Execute}(\Delta, \Gamma, \Theta)$, then there is a substitution tuple $\Sigma = \langle \Pi_1, \ldots, \Pi_k \rangle$ such that $\bot \notin \mathsf{Execute}(\Delta\Sigma, \Gamma, \Theta)$*

Proof. A drastic solution is to remove all the clauses in Δ and add a clause ϕ where $\mathsf{Head}(\phi) = \pi$ to Δ, and replacing all the assumptions with the conditions for this clause. Hence, the revised cooking session is given by $(\Delta[\Delta/\{\phi\}], \Gamma[\Gamma/\mathsf{Tail}(\phi)], \Theta)$. So this revised cooking session can be chosen to be consistent and entail π. □

The above proof concerns a quite drastic situation, and normally it is not necessary for substitutions to be so drastic. For instance, using term substitution is likely to be less drastic by maintaining more of the original recipe. However, term substitution does not always resolve the problem as shown next.

Example 53. Consider the following clause in a recipe Δ. and Γ contain the conditions of this clause. Here, there are two knead actions in the clause.

```
onhand(raised_dough, T4)
        ← do(leave_to_raise(dough, 30 min), T3, T4)
        ∧ do(knead(dough, 5 min), T2, T3)
        ∧ do(leave_to_raise(dough, 90 min), T1, T2)
        ∧ do(knead(dough, 15 min), T0, T1)
        ∧ onhand(dough, T0)
        ∧ sequence([T0, T1, T2, T3, T4])
```

Suppose there is an integrity constraint that is violated (e.g. assume that dough should be kneaded for 10 min as given below).

$$\bot \leftarrow \texttt{do(knead(dough, D)} \wedge \texttt{D} \neq \texttt{10 min}$$

To fix the clause, we might not be able to just use a substitution where the term `knead(dough, 10 min)` for `knead(dough, 5 min)` since we would also swap `knead(dough, 10 min)` for `knead(dough, 15 min)` which may be incorrect for producing the desired food item. A solution would be to separate the actions so that for instance, we have the actions `first_knead` and `second_knead`. In this way, we could substitute for one action and not the other.

Proposition 15. *Let (Δ, Γ, Θ) be a cooking session. If $\bot \in \mathsf{Execute}(\Delta, \Gamma, \Theta)$, then there is not necessarily a substitution tuple $\Sigma = \langle \Pi_0, \Pi_1, \ldots, \Pi_k \rangle$ such that each Π_i is a term substitution and $\bot \notin \mathsf{Execute}(\Delta\Sigma, \Gamma', \Theta)$.*

Proof. Consider Example 53 as a counterexample. □

The above proof is based on showing that if the terms are the same in the conflicting atoms, then term substitution won't remove the conflict as both atoms will be the same. However, with condition substitution and clause substitution, the recipe can always be revised so that it is consistent with the integrity constraints.

Proposition 16. *Let (Δ, Γ, Θ) be a cooking session for the food item π. If $\pi \notin \mathsf{Execute}(\Delta, \Gamma, \Theta)$, then there is a substitution tuple $\Sigma = \langle \Pi_0, \Pi_1, \ldots, \Pi_k \rangle$ such that $\pi \in \mathsf{Dishes}(\Delta\Sigma, \Gamma', \Theta)$ and $\bot \notin \mathsf{Execute}(\Delta\Sigma, \Gamma', \Theta)$.*

Proof. We can choose these substitutions so that π follows from the revised session. For example, we can do a clause substitution $[\Delta/\{\phi\}]$ where $\mathsf{Head}(\phi) = \pi$, and $\mathsf{Tail}(\phi)$ is consistent with Θ. So $\pi \in \mathsf{Execute}(\Delta[\Delta/\{\phi\}], \Gamma[\Gamma \setminus \mathsf{Tail}(\phi)], \Theta)$ and $\bot \notin \mathsf{Execute}(\Delta[\Delta/\{\phi\}], \Gamma[\Gamma \setminus \mathsf{Tail}(\phi)], \Theta)$. □

We can have a trivial recipe that just says that we have the dish (such as Example 54).

Example 54. The following process clause ensures that the final food product i.e. `victoria_sponge` is always available.

```
onhand(baked(victoria_sponge), T) ← ⊤
```

We can stop this by insisting that there are ingredients that are not the final item. For instance, we could prohibit executions where the intended output is in the assumptions.

Example 55. The following process clauses is for a one-step recipe (i.e. from the cake mix, we get the cake). So this would be acceptable according to the above restriction.

```
onhand(baked(victoria_sponge), T)
        ← do(cook(cake_mix), T)
        ∧ onhand(cake_mix, T)
```

With our logic-based approach, we can specify a recipe in arbitrary detail. So we can choose the level of granularity that actions are described. At one extreme, we can put all the actions in a recipe into a single process clause, and at the other extreme, we can put each action in a separate clause. We illustrate these extremes in the following example.

Example 56. Consider the following process clause that involves three actions for preparing scone with clotted cream and jam. As a culinary note, this is the Devon method rather the Cornwall method for preparing scones.

```
onhand(prepared(scone_cream_and_jam), T)
        ← do(spread_jam(scone), T)
        ∧ do(spread_cream(scone), T)
        ∧ do(cut_in_two(scone), T)
        ∧ onhand(scone, T)
```

Th above process clause can be split into the following three process clauses with each containing one process clause.

```
onhand(prepared(scone_cream_and_jam), T)
        ← do(spread_jam(on(scone_halves(with(cream_on_top))), T)
        ∧ onhand(count(scone_halves(with(cream_on_top), 2), T)

onhand(count(scone_halves(with(cream_on_top), 2)), T)
        ← do(spread_cream(on(scone_halves)), T)
        ∧ onhand(count(scone_halves, 2), T)

onhand(count(scone_halves, 2), T)
        ← do(cut_in_two(scone), T)
        ∧ onhand(scone, T)
```

When a minimal inconsistent subset is the result of an action duration or parallel action or multiple key integrity constraints, then there is a substitution into the assumptions such that the minimal inconsistency is resolved and no new minimal inconsistency is produced by the substitution. Essentially, the assumptions are revised so that the timepoints used ensure that the action duration are sufficiently long, and actions are done sequentially.

Proposition 17. *Let (Δ, Γ, Θ) be a cooking session for dish π. If $\pi \in \mathsf{Dishes}(\Delta, \Gamma, \Theta)$, and $\Phi \in \mathsf{MinInc}(\Delta \cup \Gamma \cup \Theta)$, and there is a $\phi \in \Phi$ such that ϕ is the parallel action integrity constraint, then there is a set of assumptions Γ' such that $\Phi \notin \mathsf{MinInc}(\Delta \cup \Gamma' \cup \Theta)$ and $\pi \in \mathsf{Dishes}(\Delta, \Gamma', \Theta)$.*

Proof. Since, the overall recipe with timings can be represented by a proof tree, resolving the inconsistency is about making the actions sequential rather than parallel, and therefore making the satisfaction of each process clause sequential rather than parallel. Let ϕ be the parallel actions integrity constraint. Since ϕ is in a minimal inconsistent subset, there are assumptions $\mathtt{do}(\alpha, \tau_1, \tau_2)$ and $\mathtt{do}(\alpha', \tau_3, \tau_4)$ such that $\alpha \neq \alpha'$ and τ_1 is before τ_3 and τ_3 is before τ_2. So there is a revised set of assumptions such that $\mathtt{do}(\alpha, \tau_1, \tau_2)$ and $\mathtt{do}(\alpha', \tau'_3, \tau'_4)$ such that τ_2 is before τ_3. In order to ensure that π is maintained in the final output, this may require other actions to similarly have their timings separated. □

More generally, we can resolve the minimal inconsistencies that arise from substitutions. Whilst some substitutions might cause further minimal in consistences, if we assume that our set of integrity constraints and background knowledge is sensible (Definition 17), then by Proposition 12, we can always find a substitution tuple that is a mitigation.

9. Related work

The use of computational methods to solve problems in cooking has been the focus of several studies and projects in the last two decades. Some series of workshops have also been dedicated to this topic, such as the "Computer Cooking Contest" (CCC) from 2008 to 2017, the "Cooking with Computers" (CWC) workshops in 2012 and 2013 and the "Integrated Food Ontology Workshops" (IFOW) since 2020. The focus of CCC was specifically on the adaptation of recipes using Case-Based Reasoning (CBR) techniques, CWC was about the use of AI methods in cooking, while IFOW focuses on the development of vocabularies and data models for various aspects of food such as nutritional data, eating patterns, agricultural treatments, etc. Below, we discuss how our work relates to some of the studies presented in these forums, as well as to the broader literature in this area, focusing more on three topics: representation and reasoning with recipes; ontologies and knowledge graphs; and computational methods for substitution.

9.1. Representation and reasoning with recipes

Most of the CBR-based solutions presented in CCC rely on a structured (XML or ontology-based) representation of recipes. Each ingredient as well as the dish produced by the recipe are commonly represented as separate entities. This doesn't however hold for the preparation steps, which in most cases have the form of a single block of text. Because of that, their substitution methods do not take into account the adaptation of the cooking actions that may be required for processing the new ingredients. Two notable exceptions are the Taaable [14,15] and CookingCAKE systems [16]. The former uses a tree representation of recipes where nodes represent recipe ingredients or outputs of cooking actions and edges represent actions. They use this representation either to enable users to correct or complete recipe trees, which have been created from recipe texts using NLP, using a graph editing tool [15]; or to enable the textual adaptation of recipe preparations by identifying common sequences of actions applied to a single ingredient (they

call such sequences "prototypes") and finding, using Formal Concept Analysis, the prototype that best fits the given recipe [14]. Their tree representation is similar but less expressive than the recipe graphs that we propose (for example, it cannot model by-products of a cooking action), and they do not deal with some of the reasoning tasks that we address such as the comparison or composition of recipes. The RDF representation of recipes that they proposed more recently [17] captures other aspects of a recipe such as the type of the produced dish and the types and quantities of the ingredients, but not the cooking actions.

CookingCAKE is another CBR system aimed at the adaptation of recipes that is based on the representation of recipes as workflows and the use of Process-Oriented Case-Based Reasoning (POCBR) techniques. The adaptation methods they use are based on decomposing workflows into meaningful sub-components called workflow streams [16]. A workflow stream is a collection of connected tasks that create a new item. A workflow (modelling a recipe) can be adapted by using the workflow streams of other workflows (recipes) that produce the same item in a different manner, e.g., with other tasks (cooking actions) or data (ingredients). Their methods are based on similar ideas with ours, especially with respect to the composition of recipes and structural substitution. They do not, however, provide methods for validating the generated recipes. Moreover, being based on CBR, the proposed solution cannot easily be extended with other types of knowledge (e.g. commonsense knowledge, user preferences, etc.); on the other hand, declarative systems such as the one we present in this paper are much more modular and adaptable to new types of knowledge.

Other studies that represent recipes as workflows are [18–22]; their aims, however, are different from ours. Most of them focus on extracting workflow graphs from recipe texts using NLP [19,21,22]. The system described in [18] aims at facilitating the retrieval of recipes from workflow repositories, while [20] focuses on clustering recipes. Graphs for representing recipes were also used in [23,24]. Similarly to our recipe graphs, both ingredients and cooking actions are modelled as nodes; however, their graphs do not include intermediate products or by-products. Their aims were also different; [23] developed methods for retrieving recipes from the Web, while [24] aimed at the analysis of recipes for example to identify usage patterns of ingredients and cooking actions and to compare recipes.

A representation of recipes as plans was recently proposed in [25]. A recipe is represented as a sequence of steps, each of which corresponds to a cooking action and includes various parameters such as the input ingredients, the output of the action, allergen information, etc. A plan-based representation is indeed a promising alternative; however, reasoning on recipes is left by the authors for future work.

Whilst the primary of our paper is to investigate the notion of substitution, we have outlined how recipes can be modelled using classical logic. Our approach therefore differs from the proposals we have discussed in this subsection. Classical logic provides an expressive formalism for capturing key details of recipes such as dealing with time and quantities, and using clauses allows for a direct representation of the preconditions and actions of a recipe to be the preconditions of a clause, and the post-conditions to be the consequent of a clause. Another advantage of our approach is that we can ground the recipe steps to give a set of ground clauses which can then be used directly with a SAT solver to check for consistency or entailment.

9.2. Ontologies and knowledge graphs

The use of ontologies for formally representing food or recipes has recently been gaining interest. Many of the CBR-based systems that participated in CCC used a simple ontology of ingredients to identify possible ingredient substitutions [14,16,26–29]. Most of these ontologies were implemented in RDFS but, apart from the taxonomic (subclass) relations, they do not capture any other types of relations among the ingredients or between the ingredients and the cooking actions. They are not, therefore, able to support most of the reasoning tasks that we address in this paper.

A more comprehensive, general-purpose ontology for foods is the FoodOn ontology [30,31]. This has the form of a multi-faceted taxonomy organizing foods by source organism, harvest state, region of origin, colour, production process, chemical composition, physical state, etc. The ontology contains two elements, which could be used for modelling substitutions in cooking: a symmetric property called "has food substance analog"; and a set of classes (e.g. beverage analog, chocolate product analog, egg product analog, etc.), which are used to explicitly model food product analogs for various types of food. Both elements, however, are aimed at modelling generic food analogs, rather than recipe-specific substitutions of ingredients. Due to the small number of object properties, it cannot support reasoning over recipes. It nevertheless provides a very comprehensive classification of foods, which we plan to exploit in the future, for example, for developing the comestibles hierarchy.

An ontology design pattern for ingredient substitution in recipes was recently proposed in [32]. The model captures different aspects of a dish (diet, technology, tastiness). It models a recipe as a set of ingredients and a set of instructions, and ingredient substitution as transformations of these sets, which may be required to encompass ingredient change. Its value, however, is mostly representational, as it does not support reasoning over recipes and ingredient substitution.

FoodKG is a large-scale food knowledge graph, which integrates nutrition information, general food substitutions, recipe data and food taxonomies [33]. It describes each recipe as a set of ingredients, each of which is associated with a quantity and a unit of measurement. It does not, however, describe cooking actions or the sequence in which the ingredients are processed and cannot therefore support most of the reasoning tasks that we describe in this paper.

Our logical framework is ontology-agnostic but, being based on propositional logic, is fully compatible with ontologies. It can therefore be enriched with ontological knowledge available in the food ontologies and knowledge graphs discussed above. For example, background clauses (see Section 3.2.3) could be automatically populated with class-hierarchy information from FoodOn or other related ontologies, while property clauses (see Section 7.4) could be instantiated with nutritional, compositional, or flavour data available in FoodKG. Furthermore, ontology-based semantic similarity metrics can complement our distance functions (see Section 4.1), offering an alternative way to identify candidate substitutes using taxonomic proximity or shared properties. Exploring such integrations is part of our future work plans.

9.3. Substitution

There are a number of proposals that explore how substitutions could be found for ingredients in recipes. By analysing existing recipes, cooking actions and the ingredients that they are commonly applied to can be identified, and this can then be used to identify candidates for substitution [34]. More generally, large datasets of recipes can be analysed to determine whether an ingredient tends to be essential or can be dropped or added, whether the quantity of an ingredient can be modified, which ingredients tend to co-occur frequently, and user-generated suggestions for functionally equivalent ingredients, and for healthier variants of a recipe [35]. They can also be identified by combining explicit information about the ingredients in FoodKG, and implicit information from word embeddings [13]. A common approach followed by systems that use an ontology of ingredients is to pick substitute ingredients from classes that are close to the class of the original ingredient; for example, from a parent or a sibling class [14,16,26–29]. In some of these systems, the ingredient ontology is enriched with values on the subclass relations, which indicate how suitable it is to replace one type of ingredient with another [26,28]. None of these methods provide a formalism for representing or reasoning with recipes, but they could be used for finding candidates for substitution for use in our framework.

Finally, there are two other proposals that, although not specifically focused on cooking, could also be applied to this domain. The first one is a proposal for using a formalism to capture features of objects (namely, shape, material, and role of the object) and then reason with that knowledge to identify alternative uses [36]. Potentially, this logic-based approach could be adapted for recipes by perhaps drawing on the approach in our proposal. The second is a logic-based approach to activity recognition, based on a logic programming implementation of Event Calculus [37]. The focus of that work is on the representation and recognition of long-term activities as temporal combinations of short-term activities. Although their approach cannot be applied as it is to address ingredient substitution or any other reasoning tasks that are specific to recipes, mainly due to the lack of an explicit representation of comestibles and their relations with cooking actions, it does provide some ideas that we might implement in future extensions of this work, e.g. to model and reason with the temporal aspects of recipes.

10. Discussion

In this paper, we have proposed a commonsense reasoning framework for substitutions in cooking. Overall, our perspective has been to reduce the problem to a combination of using **distance measures** to compare food items (ingredients, intermediate items, and dishes), to compare equipment, and to compare processing steps, and using **inconsistency management** where integrity constraints are used to flag when mitigations are required, and inconsistency resolution is via second substitutions.

To develop our perspective, we have made the following contributions: (1) Representation of candidates for substitution using notions of distance between the original and possible substitutes; (2) Specification of substitution as one or more syntactic operations on the logical representation of recipes; (3) Conceptualization of the primary and secondary stages of substitution; (4) Methods for four types of driver for substitution (i.e. similarity to missing item; similarity to original dish; minimal change to recipe; improve final dish); And (5) Investigation of secondary substitutions as the identification and resolution of inconsistencies.

We see substitution as part of a broader range of abilities for repurposing. We can consider cases of both substitution, i.e. finding an alternative for a missing resource, and exploitation, i.e. identifying a new role for an existing resource. For a discussion of repurposing, see [38]. The human ability to repurpose objects and processes is universal, but it is not a well-understood aspect of human intelligence. Repurposing arises in everyday situations such as finding substitutes for missing ingredients when cooking, or for unavailable tools when doing DIY. It also arises in critical, unprecedented situations needing crisis management. After natural disasters and during wartime, people must repurpose the materials and processes available to make shelter, distribute food, etc.

In parallel work, we have developed a graph-based formalism for representation and reasoning with recipes. It incorporates subset of the features that we incorporate in this paper — essentially comestibles (ingredient, intermediate food items, and final products) and actions — but it can be directly encoded as a programme in answer set programming [5]. In future work, we will extend this computational approach with the features of the logic-based approach in this paper. Furthermore, there is a pipeline we could develop starting with recipes in free text that can be translated into graphical formalism, which can then reasoned with using ASP, and substitutions being identified using the framework proposed in this paper. Investigations into the NLP translation of into graphical formalism include those that focus on the explicit information in recipes [22,23,39,40], and an investigation into the implicit information in recipes, i.e the intermediate comestibles that arise in recipes that might not be explicitly mentioned in the recipe, [41].

Once we have a recipe represented in the logic (i.e. a specific set of clauses Δ), there are various properties that we can consider for it including the following.

Finite session This means that the execution of the cooking session results in a finite set of inferences. So (Δ, Γ, Θ) is a **finite session** iff $\mathsf{Execute}(\Delta, \Gamma, \Theta)$ is finite.

Finite recipe This means that there is no set of assumptions that would result in an infinite execution. So Δ is a **finite recipe** iff $\mathsf{Execute}(\Delta, \Gamma, \Theta)$ is finite for all finite Γ.

Viable session With respect to a specific food item to be prepared, a cooking session is viable means that the execution of the session would result in the food item being produced. So Δ is **viable session** for π iff $\pi \in \mathsf{Execute}(\Delta, \Gamma, \Theta)$.

Viable recipe With respect to a specific food item to be prepared, a recipe is viable means that there is a set of assumption that would result in an execution with food item being produced. So Δ is **viable recipe** for π iff there is a Γ s.t. $\pi \in \mathsf{Execute}(\Delta, \Gamma, \Theta)$.

Minimal recipe With respect to a specific food item to be prepared, a recipe is minimal means that any subset of the recipe would not produce the item. So Δ is **minimal recipe** for π iff for all $\Delta' \subset \Delta$ s.t. $\pi \notin \mathsf{Execute}(\Delta', \Gamma, \Theta)$.

Trivial session With respect to a specific food item to be prepared, a cooking session is trivia means that the item is assumed (i.e. it is given as an ingredient). So (Δ, Γ, Θ) is **trivial session** for π iff $\pi \in \Gamma$.

Consistent session This means that the execution of a cooking session is consistent. So (Δ, Γ, Θ) is **consistent session** iff $\bot \notin \mathsf{Execute}(\Delta, \Gamma, \Theta)$.

Consistent recipe This means that for all sets of assumptions that are consistent with the integrity constraints and background knowledge, the execution is consistent. So Δ is **consistent recipe** w.r.t. Θ iff for all consistent Γ, $\bot \notin \mathsf{Execute}(\Delta, \Gamma, \Theta)$.

We now consider specific examples of recipes, and how they can be analysed using the above properties. The following example illustrates failure of the finite property.

Example 57. The following preparation clauses can give an infinite execution since the egg can be moved from one plate to the other.

$$\begin{array}{ll}
\mathtt{onhand(on(egg, plate2), T_3)} & \mathtt{onhand(on(egg, plate1), T_3)} \\
\quad \leftarrow \mathtt{put(egg, on(plate2), T_2)} & \quad \leftarrow \mathtt{put(egg, on(plate1), T_2)} \\
\quad \wedge \mathtt{onhand(on(egg, plate1), T_1)} & \quad \wedge \mathtt{onhand(on(egg, plate2), T_1)} \\
\quad \wedge \mathtt{before(T_1, T_2)} \wedge \mathtt{before(T_2, T_3)} & \quad \wedge \mathtt{before(T_1, T_2)} \wedge \mathtt{before(T_2, T_3)}
\end{array}$$

An empty recipe (i.e. $\Delta = \emptyset$) does not give a viable session unless it is a trivial session. In general, minimality of assumptions is desirable, so that we minimize the ingredients and actions required, and minimality of process clauses in the recipe is desirable so that we minimize the ingredients consumed, the actions undertaken, and the production of unnecessary food items

The finite, viable, consistent, trivial, and minimal, properties capture simple and natural commonsense questions one might ask of a recipe. When one of these properties fails, they can be explained in terms of the specific clauses that cause the failure.

In future work, we will further investigate integrity constraints and background knowledge including generating integrity constraints automatically from knowledge graphs where there are potentially many triples in a knowledge graph and do we decide which pairs of triples can be used to make an integrity constraints; development of methods for generating inconsistency constraints and background knowledge from data; acquiring and representing negative properties (e.g. bread made with yeast but we don't want it to taste of yeast); consideration of refinement of the methods to allow for use of hard and soft constraints.

Also, in future work, we will consider how to reason about the persistence of ingredients using non-monotonic reasoning, and provide more detailed coverage of how to represent and reason about recipes including how to keep track of the quantities of intermediate products and by-products of cooking. Persistence is required when fluents that have been unchanged. In other words, if there is no consume atom for some quantity of that item, then the same quantity of the item is available at the next point in time. So if it cannot be inferred that the item has not been consumed, we can make the default inference that the item is still available to be consumed. This future work is likely to draw on formalisms for analogical reasoning [42–44], temporal reasoning [2,3,45], and commonsense reasoning [46–51].

CRediT authorship contribution statement

Antonis Bikakis: Writing – original draft, Formal analysis, Conceptualization. **Aissatou Diallo:** Formal analysis, Conceptualization. **Luke Dickens:** Writing – original draft, Formal analysis, Conceptualization. **Anthony Hunter:** Writing – original draft, Formal analysis, Conceptualization. **Rob Miller:** Writing – original draft, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Anthony Hunter reports financial support was provided by Leverhulme Trust. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors are grateful to the Leverhulme Trust for supporting the project *Repurposing of Resources: from Everyday Problem Solving through to Crisis Management* (2022–2025).

Data availability

No data was used for the research described in the article.

References

[1] Dalal Alrajeh, Rob Miller, Alessandra Russo, Sebastian Uchitel, Reasoning about triggered scenarios in logic programming, Theory Pract. Log. Program. 13 (4-5-Online-Supplement) (2013).

[2] Jiefei Ma, Rob Miller, Leora Morgenstern, Theodore Patkos, An epistemic event calculus for ASP-based reasoning about knowledge of the past, present and future, in: Proceedings of International Conference on Logic for Programming, vol. 26, 2013, pp. 75–87.

[3] Rob Miller, Murray Shanahan, Some alternative formulations of the event calculus, in: Computational Logic: Logic Programming and beyond, Springer, 2002, pp. 452–490.

[4] Vladimir Lifschitz, What is answer set programming? in: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008, AAAI Press, 2008, pp. 1594–1597.

[5] Antonis Bikakis, Aïssatou Diallo Luke Dickens, Anthony Hunter, Rob Miller, A graphical formalism for commonsense reasoning with recipes, 2023, CoRR, abs/2306.09042.

[6] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, Efficient estimation of word representations in vector space, CoRR 1301.3781 (2013).

[7] Jeffrey Pennington, Richard Socher, Christopher Manning, Glove: Global vectors for word representation, in: Proceedings of EMNLP'14, Association for Computational Linguistics, 2014, pp. 1532–1543.

[8] Chantal Pellegrini, Ege Özsoy, Monika Wintergerst, Georg Groh, Exploiting food embeddings for ingredient substitution, in: Proceedings of the 14th International Joint Conference on Biomedical Engineering Systems and Technologies, BIOSTEC 2021, Volume 5: HEALTHINF, SCITEPRESS, 2021, pp. 67–77.

[9] Corina Dima, Daniel de Kok, Neele Witte, Erhard Hinrichs, No word is an Island—A transformation weighting model for semantic composition, Trans. Assoc. Comput. Linguist. 7 (2019) 437–451.

[10] Vered Shwartz, A Systematic Comparison of English Noun Compound Representations, Technical Report, 2019, arXiv, 10.48550/ARXIV.1906.04772.

[11] Jason Youn, Tarini Naravane, Ilias Tagkopoulos, Using word embeddings to learn a better food ontology, Front. Artif. Intell. 3 (2020).

[12] Celine Steen, Joni-Marie Newman, The Complete Guide to Vegan Food Substitutions, Fair Winds Press, 2010.

[13] Sola S. Shirai, Oshani Seneviratne, Minor E. Gordon, Ching-Hua Chen, Deborah L. McGuinness, Identifying ingredient substitutions using a knowledge graph of food, Front. Artif. Intell. 3 (2021) 621766.

[14] Alexandre Blansché, Julien Cojan, Valmi Dufour-Lussier, Jean Lieber, Pascal Molli, Emmanuel Nauer, Hala Skaf-Molli, Yannick Toussaint, TAAABLE 3: Adaptation of ingredient quantities and of textual preparations, in: 18h International Conference on Case-Based Reasoning - ICCBR 2010, "Computer Cooking Contest" Workshop Proceedings, Alessandria, Italy, 2010.

[15] Valmi Dufour-Lussier, Florence Le Ber, Jean Lieber, Thomas Meilender, Emmanuel Nauer, Semi-automatic annotation process for procedural texts: An application on cooking recipes, in: Emmanuel Nauer Amélie Cordier (Ed.), Cooking with Computers Workshop - ECAI 2012, Montpellier, France, 2012.

[16] Gilbert Müller, Ralph Bergmann, Cookingcake: A framework for the adaptation of cooking recipes represented as workflows, in: Workshop Proceedings of ICCBR'15, in: CEUR Workshop Proceedings, vol. 1520, CEUR-WS.org, 2015, pp. 221–232.

[17] Emmanuelle Gaillard, Jean Lieber, Emmanuel Nauer, Adaptation of TAAABLE to the ccc'2017 mixology and salad challenges, adaptation of the cocktail names, in: Antonio A. Sánchez-Ruiz, Anders Kofod-Petersen (Eds.), Proceedings of ICCBR 2017 Workshops (CAW, CBRDL, PO-CBR), Doctoral Consortium, and Competitions Co-Located with the 25th International Conference on Case-Based Reasoning (ICCBR 2017), Trondheim, Norway, June 26-28, 2017, in: CEUR Workshop Proceedings, vol. 2028, CEUR-WS.org, 2017, pp. 253–268.

[18] Qihong Shao, Peng Sun, Yi Chen, WISE: a workflow information search engine, in: Proceedings of ICDE'09, IEEE Computer Society, 2009, pp. 1491–1494.

[19] Pol Schumacher, Mirjam Minor, Kirstin Walter, Ralph Bergmann, Extraction of procedural knowledge from the web: a comparison of two workflow extraction approaches, in: Proceedings of WWW'12, ACM, 2012, pp. 739–747.

[20] Ralph Bergmann, Gilbert Müller, Daniel Wittkowsky, Workflow clustering using semantic similarity measures, in: Proceedings of KI'13, in: Lecture Notes in Computer Science, vol. 8077, Springer, 2013, pp. 13–24.

[21] Liang-Ming Pan, Jingjing Chen, Jianlong Wu, Shaoteng Liu, Chong-Wah Ngo, Min-Yen Kan, Yugang Jiang, Tat-Seng Chua, Multi-modal cooking workflow construction for food recipes, in: Proceedings of ACM Multimedia'20, 2020, pp. 1132–1141.

[22] Yoko Yamakata, Shinsuke Mori, John Carroll, English recipe flow graph corpus, in: Proceedings of LREC'20, European Language Resources Association, 2020, pp. 5187–5194.

[23] Liping Wang, Qing Li, Na Li, Guozhu Dong, Yu Yang, Substructure similarity measurement in Chinese recipes, in: Proceedings of WWW'08, Association for Computing Machinery, 2008, pp. 979–988.

[24] Minsuk Chang, Leonore V. Guillain, Hyeungshik Jung, Vivian M. Hare, Juho Kim, Maneesh Agrawala, RecipeScape: An interactive tool for analyzing cooking instructions at scale, in: Proceedings of CHI'18, Association for Computing Machinery, 2018, pp. 1–12.

[25] Vishal Pallagani, Priyadharsini Ramamurthy, Vedant Khandelwal, Revathy Venkataramanan, Kausik Lakkaraju, Sathyanarayanan N. Aakur, Biplav Srivastava, A rich recipe representation as plan to support expressive multi modal queries on recipe content and preparation process, 2022, CoRR, arXiv:2203.17109.

[26] Juan DeMiguel, Laura Plaza, Belén Díaz-Agudo, Colibricook: A CBR system for ontology-based recipe retrieval and adaptation, in: Martin Schaaf (Ed.), ECCBR 2008, the 9th European Conference on Case-Based Reasoning, Trier, Germany, September 1-4, 2008, Workshop Proceedings, 2008, pp. 199–208.

[27] P. Javier Herrera, Pablo Iglesias, David Romero, Ignacio Rubio, Belén Díaz-Agudo, JaDaCook: Java application developed and cooked over ontological knowledge, in: Martin Schaaf (Ed.), ECCBR 2008, the 9th European Conference on Case-Based Reasoning, Trier, Germany, September 1-4, 2008, Workshop Proceedings, 2008, pp. 209–218.

[28] Régis Newo, Kerstin Bach, Alexandre Hanft, Klaus-Dieter Althoff, On-demand recipe processing based on CBR, in: Cindy Marling (Ed.), ICCBR-2010 Workshop Proceedings: Computer Cooking Contest Workshop, Karlsruhe, Germany, 2010, pp. 209–218.

[29] Sérgio Mota, Belén Díaz-Agudo, Acook: Recipe adaptation using ontologies, case-based reasoning systems and knowledge discovery, in: Emmanuel Nauer Amélie Cordier (Ed.), Cooking with Computers Workshop - ECAI 2012, Montpellier, France, 2012.

[30] Damion M. Dooley, Emma J. Griffiths, Gurinder Pal Singh Gosal, Pier Luigi Buttigieg, R. Hoehndorf, Matthew Lange, Lynn M. Schriml, Fiona S.L. Brinkman, William W.L. Hsiao, Foodon: a harmonized food ontology to increase global food traceability, quality control and data integration, NPJ Sci. Food 2 (2018).

[31] Damion Dooley, Magalie Weber, Liliana Ibanescu, Matthew Lange, Lauren Chan, Larisa Soldatova, Chen Yang, Robert Warren, Cogan Shimizu, Hande K. McGinty, William Hsiao, Food process ontology requirements, Semant. Web (2022) 1–32.

[32] Agnieszka Ławrynowicz, Anna Wróblewska, Weronika T. Adrian, Bartosz Kulczyński, Anna Gramza-Michał owska, Food recipe ingredient substitution ontology design pattern, Sensors 22 (3) (2022) 1095.

[33] Steven Haussmann, Oshani Seneviratne, Yu Chen, Yarden Ne'eman, James Codella, Ching-Hua Chen, Deborah L. McGuinness, Mohammed J. Zaki, FoodKG: A semantics-driven knowledge graph for food recommendation, in: Proceedings of ISWC'19, Springer-Verlag, 2019, pp. 146–162.

[34] Yuka Shidochi, Tomokazu Takahashi, Ichiro Ide, Hiroshi Murase, Finding replaceable materials in cooking recipe texts considering characteristic cooking actions, in: Proceedings of the ACM Multimedia'09 Workshop on Multimedia for Cooking and Eating Activities, Association for Computing Machinery, 2009, pp. 9–14.
[35] Chun-Yuen Teng, Yu-Ru Lin, Lada A. Adamic, Recipe recommendation using ingredient networks, in: Proceedings of ACM Web Science Conference, in: WebSci '12, Association for Computing Machinery, 2012, pp. 298–307.
[36] Ana-Maria Olteteanu, Zoe Falomir, Object replacement and object composition in a creative cognitive system. Towards a computational solver of the alternative uses test, Cogn. Syst. Res. 39 (2016) 15–32.
[37] Alexander Artikis, Marek J. Sergot, Georgios Paliouras, A logic programming approach to activity recognition, in: Ansgar Scherp, Ramesh C. Jain, Mohan S. Kankanhalli, Vasileios Mezaris (Eds.), Proceedings of the 2nd ACM International Workshop on Events in Multimedia, EIMM 2010, Firenze, Italy, October 25 - 29, 2010, ACM, 2010, pp. 3–8.
[38] Antonis Bikakis, Luke Dickens, Anthony Hunter, Rob Miller, Repurposing of resources: from everyday problem solving through to crisis management, 2021, CoRR, abs/2109.08425.
[39] Yi Fan, Anthony Hunter, Understanding the cooking process with english recipe text, in: Anna Rogers, Jordan L. Boyd-Graber, Naoaki Okazaki (Eds.), Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023, Association for Computational Linguistics, 2023, pp. 4244–4264.
[40] Aïssatou Diallo, Antonis Bikakis, Luke Dickens, Anthony Hunter, Rob Miller, Unsupervised learning of graph from recipes, 2024, arXiv:2401.12088.
[41] Aïssatou Diallo, Antonis Bikakis, Luke Dickens, Anthony Hunter, Rob Miller, Pizzacommonsense: Learning to model commonsense reasoning about intermediate steps in cooking recipes, 2024.
[42] Rogers Hall, Computational approaches to analogical reasoning: A comparative analysis, Artificial Intelligence 39 (1989) 39–120.
[43] Henri Prade, Gilles Richard, Analogical proportions: Why they are useful in AI, in: Zhi-Hua Zhou (Ed.), Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021, ijcai.org, 2021, pp. 4568–4576.
[44] Suryani Lim, Henri Prade, Gilles Richard, Using analogical proportions for explanations, in: Florence Dupin de Saint-Cyr, Meltem Öztürk-Escoffier, Nico Potyka (Eds.), Scalable Uncertainty Management - 15th International Conference, SUM 2022, Paris, France, October 17-19, 2022, Proceedings, in: Lecture Notes in Computer Science, vol. 13562, Springer, 2022, pp. 309–325.
[45] James F. Allen, Maintaining knowledge about temporal intervals, Commun. ACM 26 (11) (1983) 832–843.
[46] Gerd Brewka, Nonmonotonic Reasoning: Logical Foundations of Commonsense, Cambridge University Press, 1991.
[47] Ernest Davis, The naive physics perplex, AI Mag. 19 (4) (1998) 51–79.
[48] Erik T. Mueller, Commonsense Reasoning, Morgan Kaufmann, 2006.
[49] Ernest Davis, Logical formalizations of commonsense reasoning: A survey, J. Artificial Intelligence Res. 59 (2017) 651–723.
[50] Ernest Davis, Gary Marcus, Noah Frazier-Logue, Commonsense reasoning about containers using radically incomplete information, Artificial Intelligence 248 (2017) 46–84.
[51] Fabio Aurelio D'Asaro, Antonis Bikakis, Luke Dickens, Rob Miller, Probabilistic reasoning about epistemic action narratives, Artificial Intelligence 287 (2020) 103352.

Antonis Bikakis is an associate professor in the Dept of Information Studies, University College London, London, UK.

Aissatou Diallo is a research fellow in the Dept of Computer Science, University College London, London, UK.

Luke Dickens is an associate professor in the Dept of Information Studies, University College London, London, UK.

Anthony Hunter is a professor in the Dept of Computer Science, University College London, London, UK.

Rob Miller is an associate professor in the Dept of Information Studies, University College London, London, UK.