A Graphical Formalism for Reasoning about Substitution in Resource Transforming Procedures

ANTONIS BIKAKIS*, University College London, United Kingdom
FABIO AURELIO D'ASARO, Università del Salento, Italy
AISSATOU DIALLO, University College London, United Kingdom
LUKE DICKENS, University College London, United Kingdom
ANTHONY HUNTER, University College London, United Kingdom
ROB MILLER, University College London, United Kingdom

The ability to repurpose and substitute materials and resources when necessary is an important aspect of human reasoning and activity. In particular, substitution plays a vital role in resource consuming and artifact producing activities – purposeful, goal directed procedures that transform resources from raw materials into finished products, the descriptions of which we refer to here as *recipes*. To see this, consider how adaptable humans are when we encounter constraints, such as limited resources, when making, manufacturing and constructing. In spite of this there has been comparatively little work given to developing representations for substitution within such contexts in a formal reasoning framework. We address this gap by proposing a graphical formalisation that captures consumables and the actions on them in the form of labelled bipartite graphs. Using examples such as "do it yourself" (DIY) instructions, manufacturing processes and cooking recipes to illustrate, we then propose formal definitions for comparing recipes, for composing recipes from subrecipes, and for deconstructing recipes into subrecipes. We then introduce and compare two formal definitions for substitution which are required when there are missing consumables, or some actions are not possible, or because there is some need to change the final product. We illustrate how automated reasoning about recipes in this context may be achieved by implementing our definitions in answer set programming (ASP).

JAIR Associate Editor: Laura Giordano

JAIR Reference Format:
Antonis Bikakis, Fabio Aurelio D'Asaro, Aissatou Diallo, Luke Dickens, Anthony Hunter, and Rob Miller. 2025. A Graphical Formalism for Reasoning about Substitution in Resource Transforming Procedures. *Journal of Artificial Intelligence Research* 84, Article 1 (September 2025), 45 pages. DOI: 10.1613/jair.1.18606

1 Introduction

Many human activities require the ability to reason about physical materials and actions upon them. This includes the ability to repurpose and substitute materials and resources, and modify actions when necessary (Bikakis et al. 2021). In particular, substitution plays a vital role in resource consuming and artifact producing activities —

*Corresponding Author.

Authors' Contact Information: Antonis Bikakis, ORCID: 0000-0003-4162-1818, a.bikakis@ucl.ac.uk, University College London, London, United Kingdom; Fabio Aurelio D'Asaro, ORCID: 0000-0002-2958-3874, fabioaurelio.dasaro@unisalento.it, Università del Salento, Lecce, Italy; Aissatou Diallo, ORCID: 0000-0003-1556-2391, a.diallo@ucl.ac.uk, University College London, London, United Kingdom; Luke Dickens, ORCID: 0000-0003-0896-1407, l.dickens@ucl.ac.uk, University College London, London, United Kingdom; Anthony Hunter, ORCID: 0000-0001-5602-7446, anthony.hunter@ucl.ac.uk, University College London, London, United Kingdom; Rob Miller, ORCID: 0000-0002-2879-2025, r.s.miller@ucl.ac.uk, University College London, London, United Kingdom.

DOI: 10.1613/jair.1.18606

purposeful, goal directed procedures that transform resources from raw materials into finished products. For the purpose of this paper, we use the term "recipe" to refer to any series of procedural steps or set of instructions for producing material artifacts or products that involves the consumption of materials or physical resources. Examples of such are D.I.Y. instructions, industrial manufacturing processes and cooking recipes.

Having a precise formalism for representing recipes, and precise definitions for analysing and manipulating them, allows for automating the process of comparing and manipulating recipes. For instance, we can automatically determine whether a recipe is consistent, or incomplete, or what potential recipes are consistent with a set of constraints. Moreover, we can integrate representation and reasoning for recipes with automated planning systems, so that for instance a recipe is turned into a specific plan for execution (taking into account constraints on time and resources), and with robotic systems, so that for instance a recipe is executed by a physical robot to produce the required products.

To facilitate the above, we propose a high-level representation of recipes as labelled bipartite graphs where the first subset of nodes denotes the consumables involved in the recipe (input materials, intermediate items, final products and by-products) and the second subset of nodes denotes actions on those consumables. The edges reflect the (possibly partially ordered) sequence of steps taken in the recipe going from the input materials to products. The labelling on nodes is used to annotate the type of consumable or action for each node. We also show how this graphical representation can be implemented in answer set programming (ASP) to enable automation of the various reasoning tasks outlined above.

Note that our formalism allows for the explicit representation of intermediate steps and consumables that might otherwise be implicit in recipes. Taking the examples of cooking and D.I.Y. instructions (for which there is a wealth of examples on the web), recipes in text or pictures do not typically explicitly state all necessary actions or all the inputs and outputs of each action, instead much of this is implicit and based on commonsense understanding. Throughout the explanation of a recipe, pronouns and elliptic references are made to consumables and actions, under the assumption that common sense is enough for a reader to understand the sense of the recipe. For instance, it is common in a cooking recipe for bread to include a step where the flour, salt, water and yeast are mixed together, but the recipe may not explicitly state that the result of this step is a dough. Nonetheless, subsequent steps may still refer to a dough, so the implicit assumption is that it is clear to the reader that the output of the mixing action is dough. Commonsense knowledge has to be used to fill the gap. Using our formalism, these gaps in the text can be exposed.

By formalising recipes, we can also consider the nature of substitution in recipes (i.e., replacing consumables and actions, or even replacing subrecipes, within recipes). There are various reasons for why there might be a need to substitute input materials.

- The most obvious reason for substitution is when there is a **lack of availability** of an input material. For example, when pitching a tent, we may not have any metal tent pegs but can instead use wooden pegs cut from branches found nearby.
- Substitution is also required to replace one or more recipe elements with appropriate substitutes because of **individual constraints** on their usage. This could perhaps arise due to preferences or requirements of the consumer, manufacturer or supplier, and may relate to properties of the input materials, actions and/or products. For instance in cooking, when considering dietary constraints (e.g., because of allergies, calorie limits, or vegan considerations), the substituting ingredient must satisfy competing demands: it must retain some properties of the substituted ingredient (e.g., preserving flavour or consistency), while changing others (e.g., to be free from gluten, or lower in calories, or not derived from animals).
- A third reason for substitution can arise from **global constraints**. For example, we may want to take into consideration the cost of the recipe as a whole. One such cost could be in terms of environmental impact, which can arise from production, transport, packaging and/or other concerns.

From the above, we see that substitution in recipes raises challenges for a knowledge representation and reasoning formalism, as there is need to satisfy constraints as to what is a valid recipe, and this calls for non-trivial automated reasoning.

This general perspective we articulate on resource transforming procedures has seen little focused research, but in the culinary domain, there are other proposals for graphical representations of cooking recipes. These include those used as a target language for natural language processing (Dufour-Lussier et al. 2012; Mori et al. 2014; Schumacher et al. 2012; Yamakata et al. 2020), to support the retrieval of recipes from the Web (Wang et al. 2008) or workflow repositories (Shao et al. 2009), to identify usage patterns of ingredients and/or cooking actions (Bergmann et al. 2013; Blansché et al. 2010; Chang et al. 2018) or to adapt recipes to the user's requirements (Müller and Bergmann 2015). We discuss these proposals, together with other related work, in Section 12. However, none suffice for our present purposes. Most lack a formal semantics, and none systematically consider the inputs and outputs of each action. Consequently, they cannot support operations to compose recipes from atomic recipes, operations to deconstruct complex recipes, formal methods for analysing or comparing recipes, or formal methods for defining substitutions in recipes that span input materials, actions, and subrecipes.

The rest of the paper is organised as follows. Section 2 summarises the terminology and definitions from graph theory needed for our proposed framework and tools. Section 3 introduces *type hierarchies*, needed to express correspondences and relationships between different actions and those between consumables. Section 4 presents a representation of recipes as bipartite graphs, and Section 5 considers the validity or acceptability of such recipe graphs. Section 6 presents definitions for comparing recipes, and Section 7 presents definitions for composition of recipes from subrecipes. Section 8 defines a type of substitution based on changing the types of nodes within a recipe graph, and Section 9 discusses substitution based on changing the structure of the graph. Section 10 presents an implementation of our definitions as an answer set program (ASP). Section 11 discusses the broad applicability of our framework and presents and discusses examples within various domains. Finally, Section 12 discusses related work, and Section 13 reflects on our contributions, as well as suggesting directions for future work.

2 Background Terminology

In this section we recall some basic terminology and definitions from graph theory needed for the rest of the paper. We use the standard notion of a directed acyclic graph, i.e. a graph with directed edges in which there are no cycles. Formally, a **directed graph** is a pair (V, E) where V is a set of nodes and $E \subseteq V \times V$ is a binary relation over V. We say that there is a directed edge or **arc** from node a to node b whenever $(a, b) \in E$. A directed graph is **acyclic** iff the transitive closure E^+ of E is irreflexive, i.e. for all $v \in V$, $(v, v) \notin E^+$. We say there is a **path** from node a to node b whenever $(a, b) \in E^+$.

A directed graph is (weakly) connected iff there is a path between any two different nodes if we disregard the direction of the edges, and is strongly connected if there is a path in both directions between any two different nodes. Formally, a directed graph is **weakly connected** iff for any two nodes $a, b \in V$, $a \neq b$, then $(a, b) \in E_u^+$, where E_u is derived from E as follows: $(x, y) \in E_u$ iff $(x, y) \in E$ or $(y, x) \in E$.

A directed acyclic graph is **rooted** at node $r \in V$ iff for all $v \in V \setminus \{r\}$ the following conditions hold: (a) $(v, r) \notin E$; and (b) $(r, v) \in E^+$.

We also use the notion of a bipartite graph, i.e. a graph for which it is possible to partition the nodes into two disjoint sets such that there are no edges connecting any two nodes from the same set. Formally, a graph (V, E) is **bipartite** iff there are two sets $V_1, V_2 \subset V$, such that $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$, and for each $(u, v) \in E$ and $k \in \{1, 2\}$, $\{u, v\} \cap V_k \neq \emptyset$.

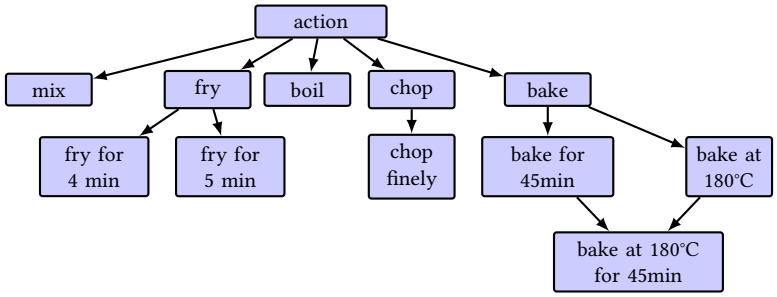


Fig. 1. An incomplete type hierarchy for action types. Each type is more specialized than its parent. The text in each node specifies the type, and may be written in different ways (e.g., "bake at 180°C for 45min" is equivalent to "bake at 356F for 45min").

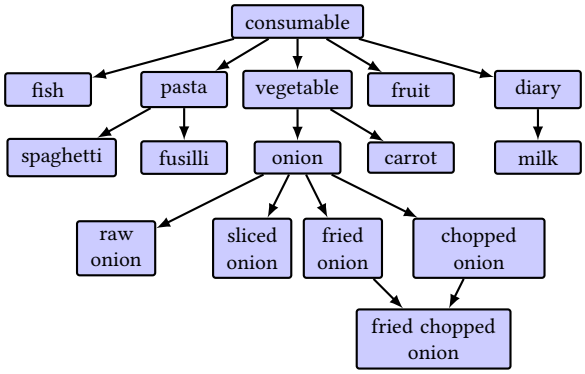


Fig. 2. An incomplete type hierarchy for consumable types. Each type is more specialised than its parent.

We use rooted directed graphs in Section 3 below to define type hierarchies of actions and consumables found in recipes, and we use weakly connected, acyclic, bipartite directed graphs in Section 4 to represent the structure of recipes.

3 Type Hierarchies

We focus on consumables and actions as the two basic entity types in a recipe. We treat actions as atomic so that the details of duration, modifiers, equipment, etc. are assumed to be indivisible parts of the action descriptor. Domains might include actions at a "high-level", as well as at a more specific or "lower-level" actions with more detailed descriptors. For instance, in the cooking domain high-level actions might include "mix" or "bake", while lower-level actions might be "mix in a blender until uniform" or "bake at 180°C for 45 minutes". Similarly, we treat consumables as atomic too. So in the case of the cooking domain, at a high-level we might have consumables such as "fish" or "fruit", while at a more detailed level, we might have "smoked salmon" or "cherry tomato". We also assume that consumables are infinitely available, in the sense that we do not consider quantities of them and so assume there is always as much as is needed for a recipe.

We assume a directed acyclic graph $(\mathcal{T}_\mathsf{A}, S_\mathsf{A})$, which represents the **action type hierarchy**, rooted at node "action" (e.g., see Figure 1). Each node $a \in \mathcal{T}_\mathsf{A}$ represents an action type. For each pair of nodes, t_1, t_2 such that

$(t_2, t_1) \in S_A^+$ (the transitive closure of S_A), we say that t_1 is a **subtype** of t_2. We write $t_1 \preceq t_2$ to denote that t_1 is a subtype of, or equal to, type t_2, (e.g.,in Figure 1, "finely chop" is a subtype of "chop"), and write $t_1 \simeq t_2$ to denote that either $t_1 \preceq t_2$ or $t_2 \preceq t_1$ holds (i.e they are on the same path). We assume a similar structure for the **consumable type hierarchy** $(\mathcal{T}_C, S_C)$ rooted at node "consumable" (e.g.,see Figure 2).

Action and consumable hierarchies can be encoded in any language that supports directed acyclic graphs. In the following sections, we do no not commit to a specific language, to keep the definitions language-independent. In Section 10, we implement the hierarchies in the same language (ASP) that we implement recipe graphs (described in the following section) to take advantage of its representation and reasoning capabilities. An alternative choice would be RDF, which is a standard model for directed graphs representing web data or an ontology language such as RDFS or OWL. Being compatible with all these languages, it is possible to automatically generate these hierarchies by importing data from existing taxonomies/ontologies. The cooking domain is particularly well provided for in this respect, with resources such as LanguaL[1], the Agrovoc Multilingual Thesaurus[2], FoodEx2[3], the FoodOn ontology (D. Dooley et al. 2022; D. M. Dooley et al. 2018), etc. There are also several ISO vocabulary standards that cover a wide range of fields and industries, including nanotechnology (80004-1:2023[4]), the leather industry (ISO 23718:2007[5]), the rubber industry (ISO 1382:2020[6]), the plastics industry (ISO 472:2013[7]), the steel industry (ISO 6929:2013[8]), etc.; and more generic vocabularies covering multiple domains such as the Global Product Classification[9].

3.1 Parameterised Action and Consumable Types

In this subsection, we briefly explain how we could introduce parameterised types as a form of syntactic sugar. This applies to both consumables and actions. Essentially, we can use a schema for a type with variables, and these variables need to be instantiated to give individual types. This would mean that we remain with a propositional language rather than a first-order language but it would allow us to more efficiently represent types.

For parameterised actions, we can introduce a value that provides more detail about the action. For example, for baking, we can have parameterised actions such as the following where X and Y are variables to be instantiated in the schema

- "bake for X minutes" where $X \in \{0, 1, 2, \ldots, 360\}$
- "bake at Y degrees Celsius" where $Y \in \{120, 125, 130, \ldots, 550\}$
- "bake at Y degrees for X minutes" where $X \in \{0, 1, 2, \ldots, 360\}$ and $Y \in \{120, 125, 130, \ldots, 550\}$

Examples of instantiations of these schemas, which are in the form of propositional atoms, are given below.

- "bake for 45 minutes"
- "bake at 220 degrees Celsius"
- "bake at 220 degrees Celsius for 45 minutes"

This can then be reflected in associated templated forms for child relationships, for instance "bake for X minutes" $\preceq$ "bake" and "bake at Y degrees for X minutes" $\preceq$ "bake for X minutes".

Parameterised actions could include consumable types, which might be useful for more coarsely grained recipes, such as the following:

[1] https://www.langual.org/default.asp
[2] https://agrovoc.fao.org/browse/agrovoc/en/
[3] https://www.efsa.europa.eu/en/data/data-standardisation
[4] https://www.iso.org/standard/79525.html
[5] https://www.iso.org/standard/67424.html
[6] https://www.iso.org/standard/68018.html
[7] https://www.iso.org/standard/44102.html
[8] https://www.iso.org/standard/52949.html
[9] https://www.gs1.org/standards/gpc

- "pour X over Y" where $X \in$ {"dessert sauce"} and $Y \in$ {"dessert"}
- "glaze X with Y then dust with Z" where $X \in$ {"unglazed cake"}, $Y \in$ {"glazing agent"} and $Z \in$ {"dusting agent"}

We can also consider parameterised consumables. Just as with action types, these could be value-based, such as "pastry rolled to thickness X", or they could involve one or more consumables. Most interestingly, if S is a set of consumable types, there are generic consumable types such as the following:

- mix_of_spices(S) where $|S| > 1$ and $S \subseteq \{t \in \mathcal{T}_C \mid t \prec$ "spice"$\}$
- mix_of_chopped_dried_fruit(S) where $|S| > 1$ and $S \subseteq \{t \in \mathcal{T}_C \mid t \prec$ "chopped dried fruit"$\}$

The syntactic sugar proposed for parameterised actions and consumables can be used directly with all the other definitions proposed in this paper. In other words, we use syntactic sugar in its conventional sense to describe a short-hand technique. These short-hand parameterised types (e.g., mix_of_spices(S)) represent a collection of grounded consumable types (e.g., mix_of_spices("Ground Black Pepper", "Ground Cumin"), mix_of_spices("Black Cardamom Seeds", "Amchar Powder"),...), and these are then used directly with the rest of the definitions of the framework.

The motivation for incorporating parameterised actions and consumables depends on how we intend to acquire and use the framework. If we wanted to take a conservative approach we could conceivably specify in advance all actions and consumables (or obtain them from source data). In contrast, if we wanted to identify new actions and consumables, then the parameterisation would allow us to instantiate them in novel ways. For instance, if we allowed the instantiation of X by "chocolate" and Y by "pasta", then we instantiate "pour X over Y" as "pour chocolate over pasta". Such a permissive instantiation would need to be checked by the user before being adopted but it may lead to some interesting novel creations. The acceptance of these novel creations will be managed by the concepts of acceptability tuples and valid recipes to be described in Section 5.

4 Recipes as Graphs

We represent the "structure" of recipes as bipartite graphs based on a finite set of nodes C, called consumable nodes, and a finite set of nodes $\mathcal{A}$, called action nodes, such that $C \cap \mathcal{A} = \emptyset$. For each action node, an incoming (respectively outgoing) node represents an input (respectively output) for the action.

Definition 1. *A* **recipe graph** *is a tuple (C, A, E) where: (1) $\emptyset \subset C \subseteq C$ and $\emptyset \subset A \subseteq \mathcal{A}$; (2) E is a set of arcs that is a subset of $(C \times A) \cup (A \times C)$; (3) $(C \cup A, E)$ is a weakly connected directed acyclic graph; (4) for all $n_a \in A$, there are arcs (n_c, n_a) and (n_a, n'_c) in E; and (5) for all $n_c \in C$, if $(n_a, n_c), (n'_a, n_c) \in E$, then $n_a = n'_a$.*

Intuitive descriptions of the conditions for a recipe graph are as follows: condition (1) – a recipe graph cannot be an empty graph; condition (2) – a recipe graph is bipartite; condition (3) – connectedness (with bipartness) ensures that all consumables are either inputs, intermediates or outputs and that all recipe subparts link together to form a whole, and acylicity enforces a natural partial ordering over the stages of the recipe (inputs processed via intermediates into outputs); condition (4) – for each action node, there is at least one incoming arc and at least one outgoing arc (actions transform one or more input consumables into one or more output consumables); and condition (5) – for each consumable node, there is at most one incoming arc (two or more actions cannot output the same consumable). This last condition is included because, given the granularity at which we want to reason, it is important that each sub-process that leads directly to the production of a consumable is regarded as atomic and described unambiguously in a single action descriptor. We shall see that this has benefits in understanding a recipe, and facilitates simpler definitions for analysing recipes, combining recipes, and substitution in recipes[10].

[10]It is also a practically viable condition to maintain. In cases where it would seem that two or more different actions appear to result in the same consumable, this can be managed by appropriate "subtyping" of that consumable. For example, in the steel making example in Section

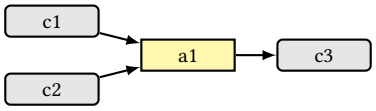


Fig. 3. An atomic recipe graph where $c1$, $c2$, and $c3$ are consumable nodes and $a1$ is an action node.

A recipe graph is not strongly connected otherwise it would be cyclic. Nodes are not necessarily unique to a single recipe graph. In other words, for distinct recipe graphs (C, A, E) and (C', A', E'), it is possible for $(C \cup A) \cap (C' \cup A') \neq \emptyset$. Definition 1's conditions 1), 2) and 4) together imply that the fewest number of action nodes a recipe can contain is one, and that these minimal recipes must have both incoming and outgoing consumable nodes[11]. This leads to the following definition of atomic recipe graphs.

Definition 2. *An* **atomic recipe graph** *is a recipe graph* (C, A, E) *where* A *contains exactly one action node.*

We give an example of an atomic recipe graph in Figure 3 and a non-atomic recipe graph in Figure 4.

A recipe (as illustrated in Figure 4) is a recipe graph together with a *typing function* (F in the definition below). This function assigns an action (respectively consumable) type to each action (respectively consumable) node, such that each type comes from the type hierarchy and no two consumable nodes get types from the same branch of the hierarchy.

Definition 3. *A* **recipe** R *is a tuple* (C, A, E, F) *where* (C, A, E) *is a recipe graph and* $F : C \cup A \rightarrow \mathcal{T}_C \cup \mathcal{T}_A$ *is a* **typing function** *that assigns a consumable (respectively action) type to each consumable (respectively action) node in* C *(respectively* A*) s.t. for all* $n, n' \in C$*, if* $F(n) \simeq F(n')$*, then* $n = n'$*. For a recipe* $R = (C, A, E, F)$*, we also define the following notation:* $\mathsf{Graph}(R) = (C, A, E)$*,* $\mathsf{Type}(R) = F$*,* $\mathsf{Nodes}(R) = C \cup A$*, and* $\mathsf{Arcs}(R) = E$*.*

In practice, we interpret a recipe as supporting any collection of concrete input ingredients that are subtypes of the consumable types that input to the recipe. For example, if we have a recipe that takes "raw prawns" as input, and applies the action "grill" to give output type "grilled prawns", then this recipe would apply to any subtype of "raw prawns", e.g., "raw tiger prawns". However, the recipe does not give information about what the more specific output type would be. For instance, we would not in this case be able to reason from the recipe alone that the recipe produced "grilled tiger prawns".

Note that Definition 3 requires that no two consumable nodes have types in the same branch of the hierarchy. This ensures no consumable is the result of one or more actions on itself. Otherwise, this might mean that an action did not change a consumable, even when certain subtypes were used as input. For example, for an action a with input consumable node c and output consumable node c', and using the type hierarchies in Figures 1 and 2, we do not allow assignment $F(c)$ = "fried onion", $F(a)$ = "fry", and $F(c')$ = "fried onion". Neither do we allow the assignment $F(c)$ = "onion", $F(a)$ = "fry", and $F(c')$ = "fried onion", as this supports the subtype "fried onion" for which the action does not change the consumable. Instead, we would need something like the assignment $F(c)$ = "raw onion", $F(a)$ = "fry", and $F(c')$ = "fried onion" where there is sufficient specificity in all consumables to indicate an effect of the action.

A node with no incoming edges, such as those labelled "uncooked spaghetti" or "cold pomodoro sauce" in the recipe shown in Figure 4, represents a consumable that has the role of an **input** for the recipe. A node with no outgoing edges represents a consumable that has role of being a **product** (the intended output such as a dish, or something that can be used as an ingredient for another recipe) or a **by-product** (i.e.,a supplementary

11.2 the "charge furnace" action has output "blast furnace waste gas" while the "refining" action has "refining waste gas" as output, both subtypes of "waste gas".

[11]As shown later, we can think of these minimal recipe graphs as the units from which larger recipe graphs are built.

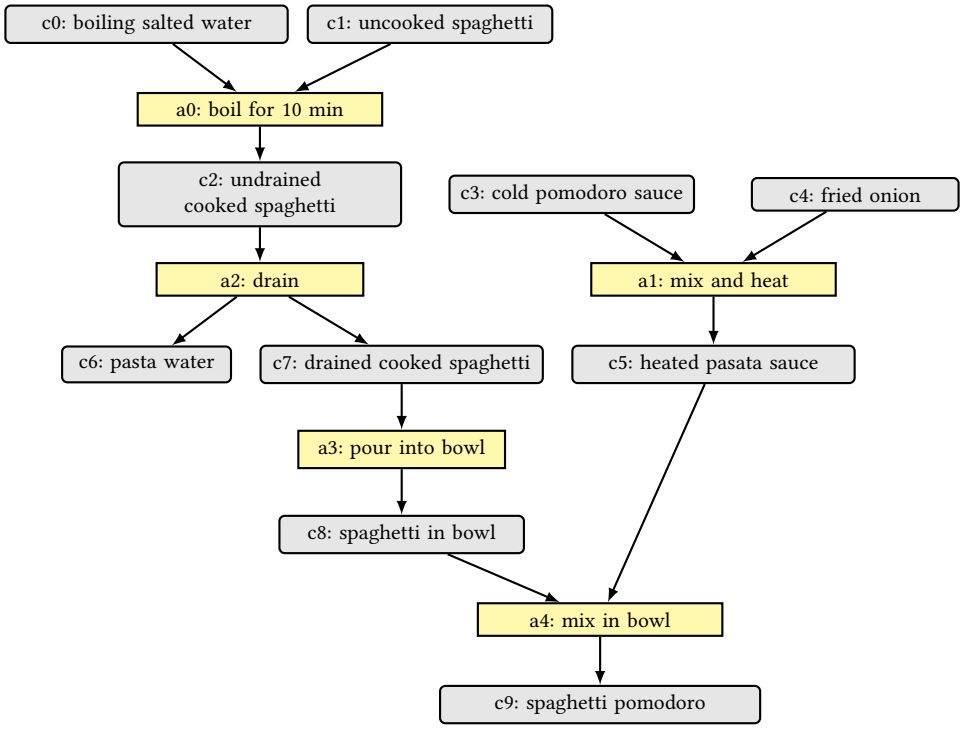


Fig. 4. A recipe R where for each node, its type is given after the colon. Let $\text{Graph}(R) = G$. So G is a non-atomic recipe graph. Also, $\text{Cons}(G) = \{\text{c0, c1, c2, c3, c4, c5, c6, c7, c8, c9}\}$, $\text{Acts}(G) = \{\text{a0, a1, a2, a3, a4}\}$, $\text{In}(G) = \{\text{c0, c1, c3, c4}\}$, $\text{Out}(G) = \{\text{c6, c9}\}$, and $\text{Mid}(G) = \{\text{c2, c5, c7, c8}\}$.

output) such as "pasta water" in the recipe in Figure 4. The capacity to express by-products is a key feature of our formalism and facilitates reasoning about potential reuse of secondary outputs from recipes – such reuse is defined as *exploitation* by Bikakis et al. 2021. A consumable node with both incoming and outgoing edges represents a consumable that is an **intermediate** item (i.e a consumable that is created and then used within the recipe). The consistent and explicit representation of intermediates is again a key feature of our formalism, and makes it possible to define a general form of substitution where whole subprocesses (subrecipes) can be swapped out for others, as long as the resulting recipe meets certain acceptability criteria (more about this in Section 9).

Given a recipe graph G, we can use different typing functions. The choice of typing function dictates what the recipe is, and, in part, its level of detail. For example, when considering an assignment to an action node using the hierarchy in Figure 1, the choice of which branch to use (from the root) will fundamentally affect the recipe, whereas going down the branch will affect the level of detail of the recipe. For example, for a node a1, typing functions F_1, F_2, and F_3, if $F_1(\text{a1})$ = "fry", $F_2(\text{a1})$ = "fry for 4 min", and $F_3(\text{a1})$ = "bake", then F_1 and F_2 are assigning actions where one is a subtype of the other, whereas F_3 is giving a completely different action.

Given Definition 1 the following properties hold for recipe graph (C, A, E) with node $n \in C \cup A$, where $\text{InDegree}(n)$ is the number of edges that impinge on n and $\text{OutDegree}(n)$ is the number of edges that exit n: (1) for all $n \in A$, $\text{InDegree}(n) > 0$ and $\text{OutDegree}(n) > 0$; (2) for all $n \in C$, $\text{InDegree}(n) \leq 1$; (3) there is an $n \in C$ s.t. $\text{InDegree}(n) = 0$; and (4) there is an $n \in C$ s.t. $\text{OutDegree}(n) = 0$.

The following notation will be useful: For a recipe R, we define $\mathsf{Acts}(R)$ to be the action nodes appearing in R; $\mathsf{In}(R)$ to be the consumable nodes that are the input nodes to R; $\mathsf{Out}(R)$ to be the consumable nodes that are the output nodes to R; $\mathsf{Mid}(R)$ to be the consumable nodes that are internal nodes in R; and $\mathsf{Cons}(R)$ to be the consumables nodes appearing in R. Therefore, $\mathsf{In}(R)$, $\mathsf{Mid}(R)$, $\mathsf{Out}(R)$ form a partition of $\mathsf{Cons}(R)$ (i.e. $\mathsf{In}(R) \cup \mathsf{Out}(R) \cup \mathsf{Mid}(R) = \mathsf{Cons}(R)$; $\mathsf{In}(R) \cap \mathsf{Out}(R) = \emptyset$; $\mathsf{In}(R) \cap \mathsf{Mid}(R) = \emptyset$; and $\mathsf{Out}(R) \cap \mathsf{Mid}(R) = \emptyset$). As an example of using these functions, see the caption of Figure 4. Finally, for a recipe R, where $n, n' \in \mathsf{Nodes}(R)$, we let $n \leq n'$ denote that either there is a path from n to n' in $\mathsf{Graph}(R)$ or $n = n'$. For example, for the recipe in Figure 4, $c_1 \leq c_7$, $c_3 \leq c_9$, and $c_5 \leq c_9$.

5 Acceptability Tuples and Valid Recipes

The definition of recipes is liberal as it does not constrain what would be a sensible recipe. As a starting point to addressing this issue, we can use existing recipes to determine what are valid inputs and outputs for each type of action. For example, for any action involving chopping or cutting, the input(s) have to be solid. As another example, if an input type is raw carrot, and the action type is chop, then the output type cannot be chopped onion. We formalise this notion of validity in the rest of this section. For this, we start with the notion of an acceptability tuple.

Definition 4. *Given consumable type hierarchy $(\mathcal{T}_\mathsf{C}, S_\mathsf{C})$ and action type hierarchy $(\mathcal{T}_\mathsf{A}, S_\mathsf{A})$, an* **acceptability tuple** *is a tuple (T_i, t, T_j), where $t \in \mathcal{T}_\mathsf{A}$ is an action type, and $T_i, T_j \subset \mathcal{T}_\mathsf{C}$ are non-empty sets of consumable types such that $T_i \cap T_j = \emptyset$ and such that for any $t_1, t_2 \in T_i \cup T_j$, if $t_1 \neq t_2$ then $t_1 \not\approx t_2$. The set of all acceptability tuples is denoted $\mathcal{U}(\mathcal{T}_\mathsf{C}, S_\mathsf{C}, \mathcal{T}_\mathsf{A}, S_\mathsf{A})$, or, where no ambiguity arises, simply as $\mathcal{U}$.*

The tuple (T_i, t, T_j) is intended to express that action t performed with consumable inputs T_i produces consumable outputs T_j. Clearly, some elements in $\mathcal{U}$, such as ({"hot water"}, "chop", {"roast chicken"}), will be non-sensical as regards reasonable representations of culinary inputs and outcomes of cooking actions. For the purposes of discussion, in the commentary below we will refer to such tuples as "non-valid", whereas tuples that fit with our commonsense notions within the given domain will be referred to as "valid". So "validity" implies that the action of the second argument, when applied to the set of inputs of the first argument would reasonably be expected to result in the set of products of the third argument. "Validity" in this context is obviously to some extent subjective, and will vary from individual to individual and from culture to culture. The definitions below are designed to accommodate this.

In the following, we assume that there is a particular starting set of valid acceptability tuples (a subset of $\mathcal{U}$), provided as part of the domain, the elements of which correspond to known and accepted outcomes when applying certain actions to input consumables. We call this starting set the set of *given acceptability tuples* and denote it U_0. Essentially, an acceptability tuple $(T_i, t, T_j) \in U_0$ captures a set of consumables T_i that within the domain of interest are known to be used with action t to give the consumables in T_j. The constraints on the types in the above definition are in line with our requirements on recipes that forbid a type from appearing alongside its subtypes, so that each element in U_0 corresponds to a possible atomic recipe.

Example 1. *Some examples of acceptability tuples that might be included in U_0 for the cooking domain include the following.*

({"loaf of bread"}, "cut", {"slice of bread"})
({"slice of bread"}, "put in toaster on medium", {"toast"})
({"raw carrot"}, "chop", {"chopped carrot"})
({"flour","eggs","milk"}, "mix thoroughly", {"batter"})
({"dried pasta", "hot water"}, "boil for 10min", {"cooked pasta", "pasta water"})
({"pasta", "pasta sauce"}, "mix and serve", {"pasta dish"})
({"fried onions", "tomatoes", "mixed herbs"}, "mix and heat", {"heated pasta sauce"})

Acceptability tuples in U_0 may come from direct knowledge, or, perhaps more usually, will have been derived from a corpus of (potentially complex) recipes known to be valid. Whatever the case, this set of given acceptability tuples may not provide complete coverage over the set of acceptability tuples that we want to consider as valid. Next, we consider how to apply rules in order to infer additional valid acceptability tuples from what is given, in order to broaden this coverage.

For instance, consider the given acceptability tuple

$$(\{\text{“flour”}, \text{“eggs”}, \text{“milk”}\}, \text{“mix thoroughly”}, \{\text{“pancake batter”}\})$$

and some additional type “plain flour” $\in \mathcal{T}_\mathsf{C}$ such that “plain flour” $\preceq$ “flour”, we might infer that

$$(\{\text{“plain flour”}, \text{“eggs”}, \text{“milk”}\}, \text{“mix thoroughly”}, \{\text{“pancake batter”}\})$$

is also a valid acceptability tuple. This is argued based on the fact that a cook can safely interpret a more general (input) ingredient as a more specific one, e.g. “flour” as “plain flour”, “prawns” as “king prawns”, and so on.

However, the suitability of such inference rules may depend on the domain, taste or other factors, and the development of improved sets of rules might represent a significant research effort in itself. To facilitate this, we provide a mechanism for inferring additional valid acceptability tuples based on a set of rules that can be tailored to need.

Definition 5. *For a given consumable type hierarchy $(\mathcal{T}_\mathsf{C}, S_\mathsf{C})$ and action type hierarchy $(\mathcal{T}_\mathsf{A}, S_\mathsf{A})$, an acceptability tuple inference rule, $\phi \subseteq 2^{\mathcal{U}} \times \mathcal{U}$, is a binary relation over the power set of acceptability tuples and the set of acceptability tuples*[12]*. For an arbitrary set of given acceptability tuples U_0 and a set Φ of acceptability tuple inference rules, we define the closure $\Phi^*(U_0)$ of U_0 with respect to Φ in the standard way as the fixed point (i.e. limit) of a series $\Phi^0(U_0), \Phi^1(U_0), \ldots$ of sets of acceptability tuples defined inductively as follows:*

$$\Phi^0(U_0) = U_0$$
$$\Phi^{n+1}(U_0) = \Phi^n(U_0) \bigcup_{\phi \in \Phi} \{r : \textit{exists } R \subseteq \Phi^n(U_0) \textit{ s.t. } (R, r) \in \phi\}$$

(Note that the existence of the fixed point is guaranteed since the series is non-decreasing and $\mathcal{U}$ is finite.) We refer to $\Phi^*(U_0)$ as the set of **valid acceptability tuples w.r.t. Φ and U_0**.

Example 2. *Here follow some example tuples, inferences, inference rules, etc. that illustrate the concepts presented in Definition 5. We begin by listing a small number of potentially useful and widely applicable inference rules, each applying to an arbitrary acceptability tuple $(T_i, t, T_j) \in \mathcal{U}$:*

- *ϕ_A encodes the example inference rule given informally above, that any element of the input set can be replaced by a more specific type, i.e.*

$$\phi_A = \{(\{(T_i, t, T_j)\}, ((T_i \setminus \{t_x\}) \cup \{t_y\}, t, T_j)) : t_x \in T_i, t_y \in \mathcal{T}_\mathsf{C}, t_y \preceq t_x\}$$

- *ϕ_B encodes that any consumable element of the output set can be replaced by a more general consumable type. This captures the notion that any atomic recipe output of a given type, is also a member of any ancestor type. Returning to the pancake batter acceptability tuple above, let*

$$u = (\{\textit{“flour”}, \textit{“eggs”}, \textit{“milk”}\}, \textit{“mix thoroughly”}, \{\textit{“pancake batter”}\}),$$

we could switch the output type to be simply “batter” to give inferred tuple u':

$$u' = (\{\textit{“flour”}, \textit{“eggs”}, \textit{“milk”}\}, \textit{“mix thoroughly”}, \{\textit{“batter”}\}),$$

The general rule is:

$$\phi_B = \{(\{(T_i, t, T_j)\}, (T_i, t, (T_j \setminus \{t_x\}) \cup \{t_y\})) : t_x \in T_i, t_y \in \mathcal{T}_\mathsf{C}, t_x \preceq t_y\}$$

[12]For those familiar with logical inference rules, the first argument can be thought of as the set of *premises* of the rule, and the second argument as the *conclusion*.

- ϕ_C *encodes that any action can be replaced by a more general action. This is informed by the fact that, in the cooking domain at least, there is a good deal of common sense required for a cook to interpret some proposed action in a recipe. Many given acceptability tuples will already be in terms of a more general action than might be available in the hierarchy, but it is reasonable to assume that a chef can appropriately interpret this. Thus our example tuple u might readily be replaced with*

$$u'' = (\{\text{“flour”}, \text{“eggs”}, \text{“milk”}\}, \text{“combine”}, \{\text{“pancake batter”}\}),$$

and the chef will know that here “combine” means the same as “mix thoroughly” which in turn might be read as “mix thoroughly until there are no lumps”. The general inference rule is:

$$\phi_C = \{(\{(T_i, t, T_j)\}, (T_i, t', T_j)) : t' \in \mathcal{T}_\mathsf{A}, t \preceq t'\}$$

With these three inference rules we can now construct an inference rule set $\Phi_{ex2} = \{\phi_A, \phi_B, \phi_C\}$*, and form the closure* $\Phi^*_{ex2}(U_0)$ *of a given set* U_0 *of acceptability tuples with respect to* Φ_{ex2} *according to Definition 5.*

Theoretically, according to rule ϕ_C, any action can be replaced by the root of the action hierarchy, “action". This ability to replace actions is an advantage; as we will see in later sections, it can support analysis of recipes, allow for comparison of recipes (to say whether one is equivalent to another, or one is a subrecipe of another), and allow for substitutions. Furthermore, being able to replace an action with a more general action, including the root of the action hierarchy, is not a problem in practice. First, whilst the definition allows this generalisation, it is not necessary to use this generalisation. In other words, different agents may prefer different levels of generality. For instance, an agent with a lot of knowledge about the actions, might be able to interpret what a more general description of that action might entail given the input and output, whereas an agent with less knowledge might want the more specific action so that they have more information about what is required. Second, when we consider substitution later in the paper, we do not replace actions unless they give some advantage in other respects. In other words, we view changes made to a recipe as having a cost, and so we aim to minimise this cost.

In the following definition, we describe when a recipe is valid with respect to a set of valid acceptability tuples (such as $\Phi^*(U_0)$). Essentially, for each action node in the recipe, we check that according to the labelling function, there is an acceptability tuple with the same types for the input and output consumables. For this, we introduce the following subsidiary definitions: $\mathsf{ActionInputs}(R, a) = \{c \mid (c, a) \in \mathsf{Arcs}(R)\}$ which gives the set of nodes in recipe R that have an arc to a, and $\mathsf{ActionOutputs}(R, a) = \{c \mid (a, c) \in \mathsf{Arcs}(R)\}$ which gives the set of nodes that have an arc from a.

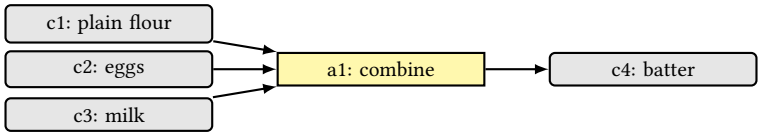
Definition 6. *Given a recipe* R *with* $\mathsf{Type}(R) = F$ *and a set of valid acceptability tuples* U*,* R *is* **valid** *w.r.t.* U *iff for all* $a \in \mathsf{Acts}(R)$*, the following holds:*

$$(\{F(c) \mid c \in \mathsf{ActionInputs}(R, a)\}, F(a), \{F(c) \mid c \in \mathsf{ActionOutputs}(R, a)\}) \in U$$

Example 3. *We continue with Example 2's inference rule set* $\Phi_{ex2} = \{\phi_A, \phi_B, \phi_C\}$*, and assume that the type hierarchies* $(\mathcal{T}_\mathsf{C}, S_\mathsf{C})$ *and* $(\mathcal{T}_\mathsf{A}, S_\mathsf{A})$ *are such that “pancake batter”*$\preceq$*“ batter”, “plain flour”*$\preceq$*“flour” and “mix thoroughly”*$\preceq$*“combine”. Then given*

$$U_0 = (\{\text{“flour”}, \text{“eggs”}, \text{“milk”}\}, \text{“mix thoroughly”}, \{\text{“pancake batter”}\})$$

the recipe represented by the following diagram is valid w.r.t. $\Phi^*_{ex2}(U_0)$*:*



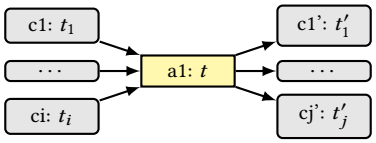


Fig. 5. An atomic recipe that corresponds with the acceptability tuple $(\{t_1, \ldots, t_i\}, t, \{t'_1, \ldots, t'_j\})$.

An atomic recipe can always be constructed from an acceptability tuple by identifying a recipe graph and a typing function such that each type in T_i is assigned to an input node to the action node, and each type in T_j is assigned to an output node from the action node, as illustrated in Figure 5. In the other direction, an acceptability tuple can be straightforwardly constructed from an atomic recipe. Indeed, atomic recipes extracted from a corpus of known recipes could be the primary source of given valid acceptability tuples in U_0. If enough recipes, from sufficient different sources, incorporate a specific atomic recipe, it may be regarded as a good candidate for inclusion in U_0.

5.1 Role and Acquisition of Acceptability Tuples

The role of acceptability tuples is to capture what users would find acceptable steps in a recipe. The concept of an acceptability tuple constitutes an achievable target for mining structured knowledge related to the task of substitution. In this subsection, we discuss potential directions that might be explored to acquire acceptability tuples from pre-existing sources. How we approach this issue will depend on the domain.

For some domains, such as in industrial processes, what would be acceptable would be relatively straightforward to establish. Established scientific and industrial knowledge can be used to specify general rules for acceptability tuples, though of course what is acceptable might still depend on the organization, industry best practices, or regulatory environments. So within a context, it may be straightforward to obtain the acceptability tuples, with all users within that context being comfortable with those acceptability tuples, while moving to another context (e.g. another organization, or another country) might mean that a different set of acceptability tuples is required.

For other domains, such as cooking, what users would find acceptable depends on their context and culture. For instance, in Thai cuisine, pla ra is often used instead of salt for seasoning dishes, but it is unlikely to be regarded as acceptable in seasoning for many people in other cultures. As another example, the use of ketchup as a pasta source might be acceptable for some people who just want quick and easy food, but it would probably be unacceptable to many people.

To be confident of the validity of a set of acceptability tuples in cooking for a set of users, the set of acceptability tuples can be constructed from established practice (for example, as manifested by recipes) within that context or culture. More specifically, if we were concerned with recipes for a particular set of users, i.e. a particular demographic, and we wanted to be conservative in the reasoning, we would use a set of acceptability tuples for that demographic. However, if we wanted to investigate novel recipes in a domain such as cooking, we could be more relaxed about the acceptability tuples, in particular to allow more general consumables and actions in each tuple, and thus explore more innovative recipes. For instance, more general patterns for acceptability tuples might possibly be obtained from, for example, manuals on use of flavours in cooking. Here, we could possibly identify general rules about the use of flavours, and deploy them for generating novel acceptability tuples.

Acquisition of acceptability tuples could possibly be undertaken from text mining of recipes. In the cooking domain, there are large corpora of recipes such as Recipe1M and within that there is a great deal of relevant information which could be mined, such as consumables (ingredients, intermediate products, and final products) and actions, and this information then harnessed to construct acceptability tuples.

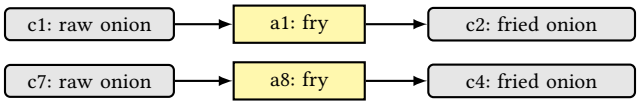


Fig. 6. The top and bottom recipes are equivalent.

An alternative approach to generating acceptability tuples is to use large language models (LLMs). In other work, we have developed a new corpus of cooking recipes enriched with descriptions of intermediate steps of the recipes that explicate the input and output for each step (Diallo et al. 2024). This information is essentially the information required for acceptability tuples. We have provided the data set together with a data collection process, and baseline models based on the LLMs T5 and GPT-3.5. This investigation has shown that LLMs are a promising approach to generating acceptability tuples. The focus has been cooking of Pizzas, but the lessons learnt with this subdomain appear generalisable.

6 Comparison of Recipes

Given a set of recipes, natural questions to ask include whether one recipe is a subrecipe of another recipe, whether two recipes are equivalent in some sense, and, if so, whether one recipe is a finer-grained, or a more specific, recipe than another. We now consider these questions. First, we introduce the following subsidiary functions that give the inputs and final products of a recipe R: $\mathsf{Inputs}(R) = \{F(n) \mid n \in \mathsf{In}(R)\}$, and $\mathsf{Outputs}(R) = \{F(n) \mid n \in \mathsf{Out}(R)\}$. Continuing the example in Figure 4, $\mathsf{Input}(R) =$ { boiling salted water, uncooked spaghetti, cold pomodoro sauce, fried onion } and $\mathsf{Output}(R) =$ { pasta water, spaghetti pomodoro }.

We use a standard notion of isomorphism to express that two recipe graphs are topologically identical: Recipes R_1 and R_2 are **isomorphic** iff there is a bijection $b : \mathsf{Nodes}(R_1) \rightarrow \mathsf{Nodes}(R_2)$, s.t. $(n, n') \in \mathsf{Arcs}(R_1)$ iff $(b(n), b(n')) \in \mathsf{Arcs}(R_2)$.

One recipe is a subrecipe of another recipe means that the former is contained in the latter.

Definition 7. *A recipe $R' = (C', A', E', F')$ is a* **subrecipe** *of recipe $R = (C, A, E, F)$, denoted $R' \sqsubseteq R$, iff $C' \subseteq C$, $A' \subseteq A$, $E' = E \cap ((C' \times A') \cup (A' \times C'))$, and for all $n \in C' \cup A'$, $F'(n) = F(n)$.*

Two recipes are equivalent if they are isomorphic and they employ the same set of acceptability tuples, as illustrated in Figure 6 and formalised in the following definition.

Definition 8. *Recipes R_1 and R_2 are* **equivalent**, *denoted $R_1 \equiv R_2$, iff R_1 and R_2 are isomorphic with bijection $b : \mathsf{Nodes}(R_1) \rightarrow \mathsf{Nodes}(R_2)$, and for all $n \in \mathsf{Nodes}(R_1)$, $F_1(n) = F_2(b(n))$, where $\mathsf{Type}(R_1) = F_1$ and $\mathsf{Type}(R_2) = F_2$.*

Two recipes are in-out aligned if they take the same input consumables and output the same products. They do not necessarily have to use the same methods to produce the outputs from the inputs (e.g.,for making a white loaf, one might use the normal process of mixing, kneading, and baking, but the other might use a bread making machine). For example, recipes in Figures 4 and 7 are in-out aligned.

Definition 9. *Recipe R_1 and R_2 are* **in-out aligned**, *denoted $R_1 \equiv_{io} R_2$, iff $\mathsf{In}(R_1) = \mathsf{In}(R_2)$, $\mathsf{Out}(R_1) = \mathsf{Out}(R_2)$, and for each $c \in \mathsf{In}(R_1) \cup \mathsf{Out}(R_1)$, $F_1(c) = F_2(c)$, where $\mathsf{Type}(R_1) = F_1$ and $\mathsf{Type}(R_2) = F_2$.*

When comparing free-text written recipes for the same dish, it is common for some to provide more intermediate steps than another. We capture this in the definition below for one recipe being finer-grained than another (e.g.,the recipe in Figure 4 is finer-grained than that in Figure 7).

Definition 10. *For recipes R_1 and R_2, R_1 is* **as fine grained as** *R_2, iff (1) $R_1 \equiv_{io} R_2$; and (2) there is a surjection $g : \mathsf{Nodes}(R_1) \rightarrow \mathsf{Nodes}(R_2)$ s.t. if $n \leq n'$ in R_1, then $g(n) \leq g(n')$ in R_2. R_1 is* **finer-grained** *than R_2 iff R_1 is as fine grained as R_2 but R_2 is not as fine grained as R_1.*

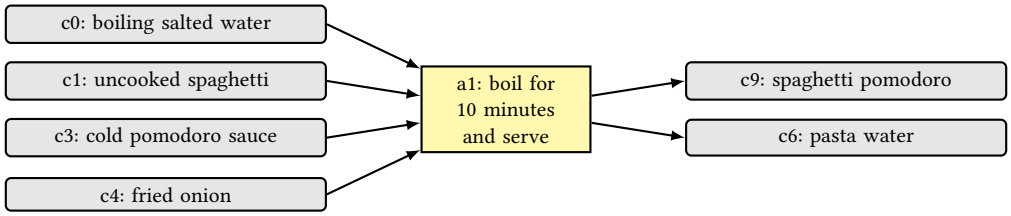


Fig. 7. This recipe is in-out aligned to the recipe given in Figure 4 as they have the same types for the input and output nodes. But the recipe in Figure 4 is finer-grained than this recipe.

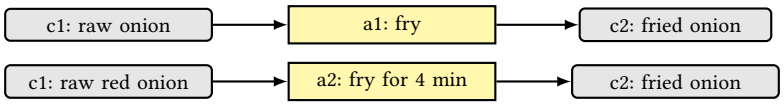


Fig. 8. The bottom recipe is more specific than the top recipe.

Condition 2 means that g is an order-preserving map. By Definition 10, a recipe R is as fine grained but not finer-grained than itself. Also, amongst an equivalence class of recipes as defined by the $\equiv_{io}$ relation, atomic recipes are the least fine grained (i.e.,for all recipes R, R', if $R \equiv_{io} R'$, and R is an atomic recipe, then R' is as fine grained as R).

We can compare recipes with respect to the type hierarchy. As defined below, and illustrated in Figure 8, this compares the specificity of the types assigned to two isomorphic graphs, and so contrasts with the above notion of granularity that compares the structures of the graphs. It provides a generalisation of the notion of equivalence.

Definition 11. *For recipes R_1 and R_2, R_1 is* **as specific as** *R_2, iff R_1 and R_2 are isomorphic with bijection b from $\mathsf{Nodes}(R_1)$ to $\mathsf{Nodes}(R_2)$ s.t. for each node $n \in \mathsf{Nodes}(R_1)$, $F(b(n)) \preceq F(n)$. R_1 is* **more specific** *than R_2 iff R_1 is as specific as R_2 but R_2 is not as specific as R_1.*

Whilst we have considered some ways of comparing recipes in this section, there are numerous further ways that we can formalise comparison of recipes that could be useful for applications. This includes representing groupings of recipes, grouped according to the input materials, the actions used, the subrecipes that they contain, and the products they output.

7 Composition of Recipes

It is natural to think of a recipe being composed of subrecipes. For instance, making a pasta dish with a sauce is composed of preparing the pasta and preparing the sauce in parallel, and then mixing them together. To investigate composition, we start with the following definition.

Definition 12. *For bipartite graphs $G_1 = (U_1, V_1, E_1)$ and $G_2 = (U_2, V_2, E_2)$, the* **bipartite union** *of G_1 and G_2 is $G_1 \uplus G_2 = (U_1 \cup U_2, V_1 \cup V_2, E_1 \cup E_2)$.*

Some simple observations are (1) $G \uplus G = G$; (2) $G_1 \uplus G_2 = G_2 \uplus G_1$; (3) if G_1 is a subgraph of G_2, then $G_1 \uplus G_2 = G_2$; and (4) if G_1 and G_2 are disjoint, then G_1 and G_2 are disjoint components in $G_1 \uplus G_2$, so that $G_1 \uplus G_2$ is not connected. These seem to be desirable properties, but unfortunately, as the example in Figure 9 illustrates, bipartite union does not meet our needs for composing recipes, since it can result in compositions that are bipartite graphs but not recipe graphs.

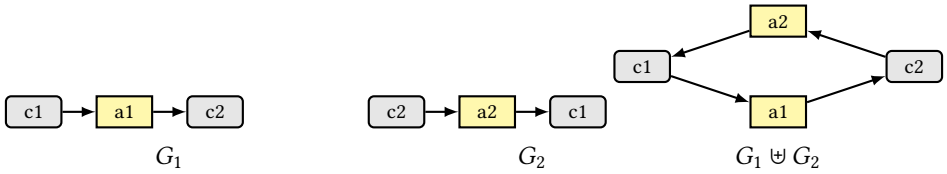


Fig. 9. Even though G_1 and G_2 are recipes graphs, the bipartite union of them, i.e.,$G_1 \uplus G_2$, violates the definition for a recipe graph (because of the acyclic condition).

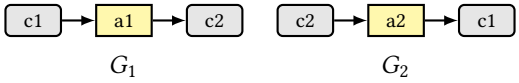


Fig. 10. For recipes R_1 and R_2, where $\mathsf{Graph}(R_1) = G_1$, and $\mathsf{Graph}(R_2) = G_2$, $R_1 \oplus R_2$ is undefined because of violation of Condition 4 in Definition 13.

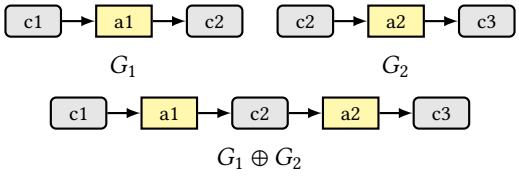


Fig. 11. For recipes R_1 and R_2, where $\mathsf{Graph}(R_1) = G_1$ (top left), and $\mathsf{Graph}(R_2) = G_2$ (top right), and any two typing functions $\mathsf{Type}(R_1) = F_1$ and $\mathsf{Type}(R_2) = F_2$, then $R_1 \oplus R_2$ is defined whenever $F_1(C_2) = F_2(C_2)$ and $F_1(C_1) \neq F_2(C_3)$ to give the graph $\mathsf{Graph}(R_1 \oplus R_2) = G_1 \oplus G_2$ (bottom).

For this reason, we use the following definition for composition, which we illustrate in Figures 10 and 11. Essentially, two recipes R_1 and R_2 can be composed if the output nodes of R_1 overlap with the input nodes of R_2, and no other nodes are common between the two recipes.

Definition 13. *If $R_1 = (C_1, A_1, E_1, F_1)$ and $R_2 = (C_2, A_2, E_2, F_2)$ are recipes such that (1) $\mathsf{Out}(R_1) \cap \mathsf{In}(R_2) \neq \emptyset$; (2) $\mathsf{Mid}(R_1) \cap \mathsf{Mid}(R_2) = \emptyset$; (3) $\mathsf{Acts}(R_1) \cap \mathsf{Acts}(R_2) = \emptyset$; (4) $\mathsf{Out}(R_2) \cap \mathsf{In}(R_1) = \emptyset$; (5) for all $n \in \mathsf{Out}(R_1) \cap \mathsf{In}(R_2)$, $F_1(n) = F_2(n)$; and (6) for all $n \in \mathsf{Cons}(R_1) \setminus \mathsf{Out}(R_1)$, and for all $n' \in \mathsf{Cons}(R_2) \setminus \mathsf{In}(R_2)$, $F_1(n) \neq F_2(n')$, then the* **composition** *of R_1 and R_2, denoted by $R_1 \oplus R_2 = (C, A, E, F)$, where $C = C_1 \cup C_2$; $A = A_1 \cup A_2$; $E = E_1 \cup E_2$; and $F = F_1 \cup F_2$, otherwise $R_1 \oplus R_2$ is undefined.*

We could generalise the definition by changing condition 5 to allow matching of subtypes rather than equality of types. We will investigate this more flexible way of combining recipes in future work.

Importantly, if the conditions of composition in Definition 13 are satisfied, then composition is a recipe.

Proposition 1. *For recipe graphs R_1 and R_2, if $R_1 \oplus R_2$ is defined then it is a recipe graph.*

Proof. Assume that R_1 and R_2 are recipes such that $R_1 \oplus R_2$ is defined. Then R_1 and R_2 satisfy conditions 1 to 5 in Definition 13. Since $C = C_1 \cup C_2$, $A = A_1 \cup A_2$, and $E = E_1 \cup E_2$, (C, A, E) is a bipartite graph. Since $\mathsf{Out}(R_1) \cap \mathsf{In}(R_2) \neq \emptyset$, the graph is connected. Since $\mathsf{Mid}(R_1) \cap \mathsf{Mid}(R_2) = \emptyset$, $\mathsf{Acts}(R_1) \cap \mathsf{Acts}(R_2) = \emptyset$, and $\mathsf{Out}(R_2) \cap \mathsf{In}(R_1) = \emptyset$, the graph is acyclic. Since for all $n \in \mathsf{Out}(R_1) \cap \mathsf{In}(R_2), F_1(n) = F_2(n)$, $F = F_1 \cup F_2$ is well-formed. So $R_1 \oplus R_2$ is a recipe. □

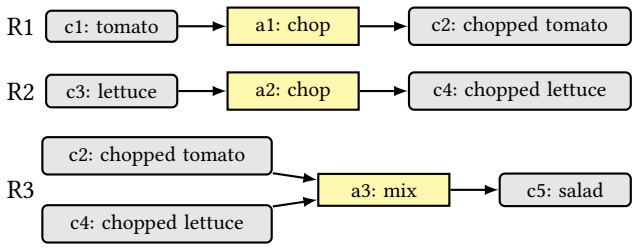


Fig. 12. For recipes $R1$, $R2$, and $R3$, $R1 \oplus (R2 \oplus R3)$ is a valid recipe but $(R1 \oplus R2)$ is undefined and so $(R1 \oplus R2) \oplus R3$ is also undefined. Hence, $\oplus$ is not associative.

As a consequence of the definition of composition, we get the following equivalences for the nodes.

Proposition 2. *For recipes R_1 and R_2, if $R_1 \oplus R_2$ is defined then the following hold: (1)* $\mathsf{In}(R_1 \oplus R_2) = (\mathsf{In}(R_1) \cup \mathsf{In}(R_2)) \setminus (\mathsf{Out}(R_1) \cap \mathsf{In}(R_2))$*; (2)* $\mathsf{Mid}(R_1 \oplus R_2) = (\mathsf{Out}(R_1) \cap \mathsf{In}(R_2)) \cup \mathsf{Mid}(R_1) \cup \mathsf{Mid}(R_2)$*; and (3)* $\mathsf{Out}(R_1 \oplus R_2) = (\mathsf{Out}(R_1) \cup \mathsf{Out}(R_2)) \setminus (\mathsf{Out}(R_1) \cap \mathsf{In}(R_2))$.

Unlike bipartite union, $\oplus$ is not commutative, nor associative (as illustrated in Figure 12). Furthermore, if $R_1 \oplus R_2$ is defined, then $R_2 \oplus R_1$ is undefined. Also if R_1 and R_2 are disjoint (i.e.,$\mathsf{Nodes}(R_1) \cap \mathsf{Nodes}(R_2) = \emptyset$), then both $R_1 \oplus R_2$ and $R_2 \oplus R_1$ are undefined. However, it is the case that if both $R_1 \oplus (R_2 \oplus R_3)$ and $(R_1 \oplus R_2) \oplus R_3$ are defined then they are equal.

Any recipe can be composed from one or more atomic recipes (i.e.,recipes with an atomic recipe graph). To show this, we use the Compose function which for a set of recipes gives the closure under the $\oplus$ operator. We define this as follows: For any set of recipes $\{R_1, \ldots, R_n\}$, $\mathsf{Compose}(\{R_1, \ldots, R_n\})$ is the smallest set such that

- $\{R_1, \ldots, R_n\} \subseteq \mathsf{Compose}(\{R_1, \ldots, R_n\})$
- for any $R, R' \in \mathsf{Compose}(\{R_1, \ldots, R_n\})$, if $R \oplus R'$ is defined then $R \oplus R' \in \mathsf{Compose}(\{R_1, \ldots, R_n\})$.

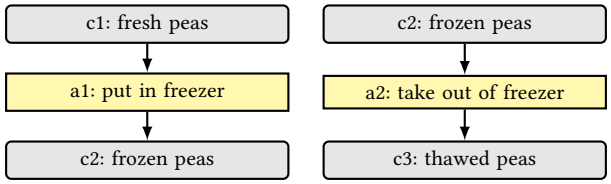
Proposition 3. *For any recipe R, there is a set of atomic recipes $\{R_1, \ldots, R_n\}$ s.t.*

$$R \in \mathsf{Compose}(\{R_1, \ldots, R_n\})$$

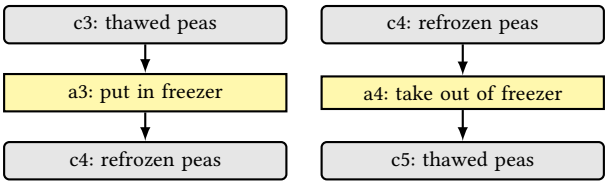
Proof. Since R is a recipe, it is a labelled connected acyclic bipartite graph. For each action node, the inputs and outputs correspond to an atomic recipe. Let $\{R_1, \ldots, R_n\}$ be the set of all such atomic recipes. Then by repeated application of Definition 13, $R \in \mathsf{Compose}(\{R_1, \ldots, R_n\})$. □

Only a finite number of recipe graphs can be composed from a finite set of atomic recipe graphs since infinite sequences cannot be formed (as shown in the proof of Proposition 4 below).

Example 4. *Consider the first two atomic recipes below. We can compose them so that we start with fresh peas and finish with thawed peas.*



We could extend the example with the following atomic recipe on the left to obtain a recipe for refrozen peas. But we would not be able to extend it further with the atomic recipe on the right because it would violate the conditions for composition (because two consumable nodes would have the same type).



Note, for this example, we assume that fresh peas, frozen peas, thawed peas, and refrozen peas are in different branches of the type hierarchy for consumables.

Proposition 4. *For any finite set of atomic recipes* $\{R_1, \ldots, R_n\}$, $|\mathsf{Compose}(\{R_1, \ldots, R_n\})| < |2^{\{R_1,\ldots,R_n\}}|$.

Proof. The proposition follows directly from the fact that, if the composition of two recipes is defined, it is unique (and therefore the step-wise composition of any number of recipes, if defined, is unique, irrespective of the order of composition), and from the fact that the "empty graph" is not a recipe. □

So far, we have considered how to compose subrecipes into recipes. Now, we consider the inverse process for turning recipes into atomic subrecipes. For a recipe R, the **decompose function**, denoted $\mathsf{Decompose}(R)$, returns the set $\{R' \mid R' \sqsubseteq R \text{ and } R' \text{ is atomic}\}$.

Proposition 5. *For any* R, *if* R *is a recipe, then* $R \in \mathsf{Compose}(\mathsf{Decompose}(R))$.

Proof. Since recipe graphs are finite, R must contain at least one action node whose outgoing arcs all lead to consumable products (i.e. members of $\mathsf{Out}(R)$). By definition of a recipe, removing such an action node together with its incoming and outgoing arcs and the consumable nodes the outgoing arcs lead to results in another well-formed (sub)recipe, which can then be subjected to an equivalent removal process. The repetition of this decomposition process until only a single atomic recipe graph remains can be represented in a binary tree with root R, whose leaves $\{R_1, \ldots, R_n\}$ are exactly all the members of $\mathsf{Decompose}(R)$, and such that all the leaves and non-leaves, including the root R, are members of $\mathsf{Compose}(\{R_1, \ldots, R_n\})$. □

So we have provided functions for composing and decomposing recipes. In this, we see that atomic recipes are the basic building blocks for recipes.

8 Type Substitution in Recipes

We now consider how we can substitute individual consumables and actions. In other words, how we can update the typing function.

A **binding** is a tuple of the form (n, t) where $n \in C \cup \mathcal{A}$ is a node and $t \in \mathcal{T}_\mathsf{C} \cup \mathcal{T}_\mathsf{A}$ is a type from a type hierarchy, such that if $n \in C$ then $t \in \mathcal{T}_\mathsf{C}$ and if $n \in \mathcal{A}$ then $t \in \mathcal{T}_\mathsf{A}$. A **substitution set** T is a set of bindings s.t. for all $(n, t), (n', t') \in T$, if $n = n'$, then $t = t'$ (i.e.,each binding refers to a different node), and if $t \simeq t'$, then $n = n'$ (i.e.,no two bindings refer to types on the same type path).

Definition 14. *Let* $R = (C, A, E, F)$ *be a recipe, and let* T *be a substitution set referring only to nodes in* $C \cup A$. *Let* R *and* T *be such that the following condition is met: there are no consumable types* $t_{c1}, t_{c2} \in \mathcal{T}_\mathsf{C}$ *and consumable nodes* $c_1, c_2 \in C$ *such that* $t_{c1} \simeq t_{c2}$, $F(c_1) = t_{c1}$, $(c_2, t_{c2}) \in T$ *and for all* t, $(c_1, t) \notin T$. *Then the* **substitution in** F **by** T, *denoted* $F \otimes T$, *is defined as follows for each* $n \in C \cup A$.

$$F \otimes T(n) = \begin{cases} t & \text{if } (n, t) \in T \\ F(n) & \text{otherwise} \end{cases}$$

Otherwise $F \otimes T$ *is undefined. For convenience, we let* $R \otimes T$ *denote the recipe* $R' = (C, A, E, F \otimes T)$.

Table 1. Consider a recipe graph G with nodes $\{c1, a1, c2, a2, c3\}$ and edges $(c1, a1), (a1, c2), (c2, a2), (a2, c3)$. The typing function F is given in the first row of the table. The second row is for the updated typing function F' which results from a primary substitution set $\{(c1, \text{raw onion})\}$. The third row is for the updated typing function F'' which results from a second substitution set $\{(c2,\text{chopped onion})\}$.

	c1	a1	c2	a2	c3
F	raw carrot	chop	chopped carrot	boil	soup
$F' = F \otimes \{(c1, \text{raw onion})\}$	raw onion	chop	chopped carrot	boil	soup
$F'' = F' \otimes \{(c2, \text{chopped onion})\}$	raw onion	chop	chopped onion	boil	soup

The condition in Definition 14 ensures that type substitution results in a new recipe, i.e. that the substitution does not result in two consumable nodes within the recipe graph whose types are on the same type path. For an illustration of type substitution, see Table 1, where the first row gives the assignment for the typing function F, and the second row gives the updated assignment after the substitution of "raw onion" for node c1.

Type substitutions allow any recipe to be turned into any other isomorphic recipe. But note that type substitution may cause a previously valid recipe to become invalid. We discuss how we deal with this issue after the following result.

Proposition 6. *For any recipes R and R', if* $\mathsf{Graph}(R)$ *and* $\mathsf{Graph}(R')$ *are isomorphic, then there is a substitution set T s.t. $F \otimes T = F'$ where* $\mathsf{Type}(R) = F$ *and* $\mathsf{Type}(R') = F'$.

Proof. Assume R and R' are isomorphic. So they have the same structure, though possibly the names of the nodes and their labelling are different. Let $b : \mathsf{Nodes}(R) \to \mathsf{Nodes}(R')$ be a bijection satisfying the constraint for isomorphic recipes (i.e. that $(n_1, n_2) \in \mathsf{Arcs}(R)$ iff $(b(n_1), b(n_2)) \in \mathsf{Arcs}(R')$), and let $T = \{(n, t) \mid F(n) \neq F'(b(n))$ and $F'(b(n)) = t\}$. So $F \otimes T = F'$. □

We now consider how the need for substitutions arises in practice. A **primary substitution** is a substitution that has been undertaken because we lack some input consumable or we are unable to do an action (perhaps because we lack required equipment or ability), whereas a **secondary substitution** is a substitution that has been carried out to deal with validity issues raised by the primary substitution. For example, suppose we lack fresh spaghetti for a cooking recipe that lists it as an ingredient. We could use dried spaghetti as a substitute for this missing consumable. This would be a primary substitution. However, the cooking time of fresh spaghetti is 3 minutes whereas the cooking time of dried spaghetti is 11 minutes. Assuming we have an appropriate set of valid acceptability tuples U, this would result in a violation of the validity constraint, and so we would need to substitute the action "boil spaghetti for 3 minutes" to "boil spaghetti for 11 minutes". This would be a secondary substitution which would be required for the recipe to regain validity.

A **primary substitution set**, denoted P, is a set of primary substitutions. For instance, suppose we have a recipe R, and we are missing input consumables $F(n_1), \ldots, F(n_k)$ where $n_1, \ldots, n_k$ are nodes in $\mathsf{In}(R)$, then we would need alternative consumables $t_1, \ldots, t_k$ to give the primary substitution set $\{(n_1, t_1), \ldots, (n_k, t_k)\}$. As defined next, a secondary substitution set is needed to fix any validity violations problems created by the primary substitution set. In other words, given a primary substitution set, we may in addition require a secondary substitution set to ensure that the resulting recipe is valid with respect to a set of valid acceptability tuples.

Definition 15. *A substitution set S is a* **secondary substitution set** *for recipe R w.r.t. a primary substitution set P and set of valid acceptability tuples U iff $P \cup S$ is a substitution set, and $R \otimes (P \cup S)$ is valid w.r.t. U.*

Note that in the above definition S cannot undo or modify the substitutions in P since this would violate the condition of $P \cup S$ being a substitution set. An illustration of the use of primary and secondary substitution sets is given in Table 1. The next proposition follows directly from Definition 14.

Proposition 7. *For a recipe R, where $\mathsf{Type}(R) = F$, the following hold: (Reflexivity) $F \otimes \{(n, F(n))\} = F$; (Associativity) if $n_1 \neq n_2$, then $(F \otimes \{(n_1, t_1)\}) \otimes \{(n_2, t_2)\} = (F \otimes \{(n_2, t_2)\}) \otimes \{(n_1, t_1)\}$ whenever both these substitution sequences are defined; (Reversibility) $(F \otimes \{(n, t)\}) \otimes \{(n, F(n))\} = F$; and (Empty) if $T = \{(n_1, t_1), \ldots, (n_i, t_i)\}$, and $n_1, \ldots, n_i \notin \mathsf{Nodes}(R)$, then $F \otimes T = F$.*

A desirable feature for a secondary substitution set is that it does not contain unnecessary substitutions. So, given a primary substitution set, we seek to identify minimal (with respect to set membership) secondary substitution sets such that the result of applying the primary and secondary substitutions is a recipe that is valid with respect to the acceptability set. This gives rise to the following definition:

Definition 16. *A* **substitution pair** *for a recipe R and acceptability set U is a tuple (P, S) where P (respectively S) is a primary (respectively secondary) substitution set, $P \cup S$ is a substitution set, $R \otimes (P \cup S)$ is valid w.r.t. U, and there is no $S' \subset S$ such that $R \otimes (P \cup S')$ is valid w.r.t. U.*

However, the above definition does not take into account the nature of individual substitutions. For instance, if we lack spaghetti for spaghetti bolognese, but we have tagliatelle and rice, either would be possible substitutes, but many would judge tagliatelle to be a much less drastic change to the recipe. To address issues such as this, we use a cost measure to compare consumables and actions. So for consumable or action types t_1 and t_2, $d(t_1, t_2)$ denotes the cost for swapping t_1 with t_2. The smaller the cost, the better one would substitute for another. So $d(t_1, t_2) = 0$ means t_2 would be a perfect substitute for t_1. We assume that it is always the case that $d(t, t) = 0$, i.e. any consumable or action is a perfect substitute for itself. It may also be appropriate in some cases to define d with respect to a recipe so that, for example, aquafaba might reasonably replace egg in baking but not in an omelette.

A cost measure can be a distance measure that has been based on a word embedding such as the general purpose word embeddings Word2Vec (Mikolov et al. 2013) or Glove (Pennington et al. 2014), or a word embedding specialised for the given domain. For instance in the cooking domain, we may wish to use Food2Vec which is a pre-trained word embedding for ingredient substitution (Pellegrini et al. 2021). For consumables t_1 and t_2, and a word embedding, the distance function $d(t_1, t_2)$ is the cosine similarity between t_1 and t_2 in the word embedding.

Alternatively, a cost measure can be derived from the type hierarchies, or from other knowledge graphs or ontologies. There are numerous such resources concerning options for substitutions (e.g. substitutions to transform a cooking recipe into a vegan alternative (Steen and Newman 2010)) that can be used as the basis of specifying cost measures. Alternatively, cost measures can be defined as a combination of word embeddings and ontological knowledge (Shirai et al. 2021). Furthermore, cost can be calculated so that generalisation is penalized. In other words, if we have the option of substituting t by t_1 or t_2, and t_1 is relatively similar to t and is as specialised in the type hierarchy as t, whereas t_2 is an ancestor of t and is therefore a more general type, then we may set d so that $d(t, t_1)$ is lower than $d(t, t_2)$, as for example when t is "carrot", t_1 is "parsnip", and t_2 is "vegetable".

We take the view that a cost measure is a more general notion than a distance measure. As suggested above, we can use distance measure as a cost measure, but we do not always want to restricted ourselves to distance measures. One reason for this is that a property of distance measures is that if $d(t_1, t_2) = 0$ holds, then t_1 and t_2 are indistinguishable, but we might want to express that some (distinguishable) substitutions are "perfect". Another property of distance measures is symmetry, i.e. for all t_1 and t_2, $d(t_1, t_2) = d(t_2, t_1)$. However, we may wish, for example, to distinguish between replacing a difficult action $t_{\texttt{diff}}$ by a simple action $t_{\texttt{simp}}$ and replacing a simple action by a difficult action. Here, we might want $d(t_{\texttt{diff}}, t_{\texttt{simp}})$ to be cheaper than $d(t_{\texttt{simp}}, t_{\texttt{diff}})$.

Another kind of flexibility we may wish to harness in the specification of a cost function is that of taking into account whether the substitution is of an input to the recipe, or an output of the recipe, or an intermediate step. In this case we can define three cost functions $d_{\mathtt{in}}$, $d_{\mathtt{mid}}$ and $d_{\mathtt{out}}$ where the cost can be different depending on whether nodes are input or output consumables rather than internal ones (actions would be covered by $d_{\mathtt{mid}}$). If a user did not wish to consider the cost of substituting intermediate consumables they could set $d_{\mathtt{mid}}(t', t) = 0$ for consumable types. Thus a cost measure can be engineered that meets the modelling and reasoning needs of the specific application.

Once we have a cost measure, one straightforward way of calculating the cost of a substitution set is using summation, as in the following definition.

Definition 17. *The* **cost** *of substitution pair (P, S) w.r.t. cost measure d and typing function F is*

$$\mathsf{Cost}_d(F, P, S) = \sum_{(n,t)\in P\cup S} d(F(n), t)$$

.

Note, however, that there are other simple alternatives for calculating the cost of a substitution set, such as *max* (i.e. take the cost $d(F(n), t)$ of the substitution $(n, t) \in P \cup S$ that is greater than or equal to the cost $d(F(n'), t')$ of any other substitution $(n', t') \in P \cup S$)). We will investigate such alternatives in future work.

Example 5. *Consider recipe R for a vegetable soup where* $\mathsf{Type}(R) = F$, $\{c1, c2, a1, a2\} \subset \mathsf{Nodes}(R)$, $F(\mathrm{c1})$ = "raw carrot", $F(\mathrm{c2})$ = "barley", $F(\mathrm{a1})$ = "chop", *and* $F(\mathrm{a2})$ = "soak". *For a substitution pair (P, S), suppose P =* {(c1, "raw onion"), (c2, "potato")}, *and S =* {(a1, "chop"), (a2, "peel and chop")}. *Then*

$$\begin{aligned}\mathsf{Cost}_d(F, P, S) = \quad & d(\text{"raw carrot"}, \text{"raw onion"})\\ & +d(\text{"barley"}, \text{"potato"})\\ & +d(\text{"soak"}, \text{"peel and chop"})\end{aligned}$$

Definition 18. *Let R be a recipe, P a primary substitution set for R, and U an acceptability set. A secondary substitution set S is a* **preferred secondary substitution set** *w.r.t. R, P, U and cost measure d iff for all substitution pairs (P, S') for R and U, $\mathsf{Cost}_d(\mathsf{Type}(R), P, S) \leq \mathsf{Cost}_d(\mathsf{Type}(R), P, S')$. In this case, (P, S) is a* ***type substitution*** *for R w.r.t. U.*

So type substitution allows us to update a typing function, and it can take account of the need for secondary substitutions. Furthermore, it can take into account how drastic the proposed changes are. However, it does not allow changes to the structure of the recipe graph – a possibility we consider in the next section.

9 Structural Substitution in Recipes

We now consider how we can represent a more general notion of substitution in recipes, by allowing a subgraph of a recipe to be replaced by a different subgraph. We start with three subsidiary definitions.

Definition 19. *For recipes R and R', if $R' \sqsubseteq R$, then the* **front set** *of R' w.r.t. R, denoted* $\mathsf{Front}(R, R')$, *is the set* $(\mathsf{Out}(R') \setminus \mathsf{Out}(R)) \cup (\mathsf{In}(R') \setminus \mathsf{In}(R))$. *Otherwise* $\mathsf{Front}(R, R')$ *is undefined.*

So the front set of a subgraph is the set of nodes that are in or out nodes in the subgraph but not in or out nodes in the enclosing graph.

Definition 20. *A recipe R' is an* **untrimmed subrecipe** *of a recipe R, denoted $R' \sqsubseteq^* R$, iff $R' \sqsubseteq R$ and for all $a \in \mathsf{Acts}(R')$, if (c, a) or (a, c) is in* $\mathsf{Arcs}(R)$, *then $c \in \mathsf{Nodes}(R')$.*

So an untrimmed subrecipe is such that for each action in the subgraph, all the consumable nodes that are connected in the original graph are in the subgraph. An example of trimmed and untrimmed subgraphs is given

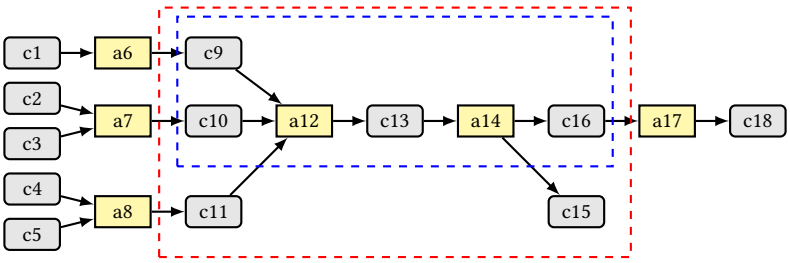


Fig. 13. Example of trimmed (blue, dashed, inner box) and untrimmed (red, dashed, outer box) subgraphs.

in Figure 13. In both Figure 14 and Figure 15, R_1 is an untrimmed subrecipe of R and R_2 is an untrimmed subrecipe of R'. When we consider structural substitution, if we attempt to find substitutes for trimmed subgraphs, then the result of the substitution may be an unconnected graph, as the new substituting subgraph might not connect to all the consumable nodes left in the recipe graph.

We also require the notion in the following definition of an existing subrecipe of a recipe being *parallel* with a candidate for its replacement. This captures whether the subgraph and its potential substitute would connect to the front nodes in the same direction. For example, in Figure 14, both R_1 (via $a2$) and R_2 (via $a9$) connect to the front node $c3$ in the same direction.

Definition 21. *Recipe R_1 is* **parallel** *to recipe R_2 w.r.t. recipe R iff for all $c \in \mathsf{Front}(R, R_1)$, the following hold: (1) for all $(c, a) \in \mathsf{Arcs}(R_1)$, there is a $(c, a') \in \mathsf{Arcs}(R_2)$; and (2) for all $(a, c) \in \mathsf{Arcs}(R_1)$, there is a $(a', c) \in \mathsf{Arcs}(R_2)$.*

We use the above Definitions 19, 20 and 21 as conditions in the following definition of structural substitution.

Definition 22. *Let R, R_1, and R_2, be recipes such that the following five conditions hold: (i) $\mathsf{Front}(R, R_1) \subseteq (\mathsf{In}(R_2) \cup \mathsf{Out}(R_2))$, (ii) R_1 is parallel with R_2 w.r.t. R, (iii) $R_1 \sqsubseteq^* R$, (iv) $(\mathsf{Nodes}(R) \setminus \mathsf{Nodes}(R_1)) \cap \mathsf{Nodes}(R_2) = \emptyset$, (v) for all $n \in (\mathsf{Cons}(R) \setminus \mathsf{Cons}(R_1))$, and for all $n' \in \mathsf{Cons}(R_2)$, if $F(n) \simeq F_2(n')$, then $n = n'$. The* **structural substitution** *of R_1 by R_2 in R, denoted $R[R_1/R_2]$, is defined as follows: $R[R_1/R_2] = (C', A', E', F')$ where*

(1) $C' = (\mathsf{Cons}(R) \setminus \mathsf{Cons}(R_1)) \cup \mathsf{Cons}(R_2)$
(2) $A' = (\mathsf{Acts}(R) \setminus \mathsf{Acts}(R_1)) \cup \mathsf{Acts}(R_2)$
(3) $E' = (\mathsf{Arcs}(R) \setminus \mathsf{Arcs}(R_1)) \cup \mathsf{Arcs}(R_2)$
(4) *for all $n \in C' \cup A'$, $F'(n) = F_2(n)$ if $n \in \mathsf{Nodes}(R_2)$, and $F'(n) = F(n)$ if $n \notin \mathsf{Nodes}(R_2)$.*

If conditions (i)–(v) do not hold then $R[R_1/R_2]$ is undefined.

So the nodes, arcs, and labelling, of the graph obtained by substitution are those of R minus those of R_1 and plus those of R_2 as illustrated in Figures 14 and 15. The five preconditions in Definition 22 can be explained as follows: (i) recall that the front of R_1 w.r.t. R is the set of nodes that are in or out nodes in R_1 but not in or out nodes in R, so requiring that this front must be a subset of the in and out nodes of R_2 ensures that there are sufficient replacement nodes in R_2 to reconnect R; (ii) this condition states that the connections R_2 makes to R are in the same direction as those that R_1 makes to R, thus ensuring that for each node in R_1 that goes to (respectively from) a node in the rest of R, the corresponding node in R_2 goes to (respectively from) a node in the rest of R; (iii) this condition states that R_1 is an untrimmed subrecipe of R, so ensuring that for each action in R_1, all the consumable nodes that are connected to that action in R are also connected to that action in R_1; (iv) this condition states that R_2 does not have any consumable nodes in common with R except those in R_1, thus ensuring that R_2 only connects with R via the same nodes as R_1; (v) this condition states that no consumable node in the remainder of R after R_1 has been removed is on the same type hierarchy path as any consumable

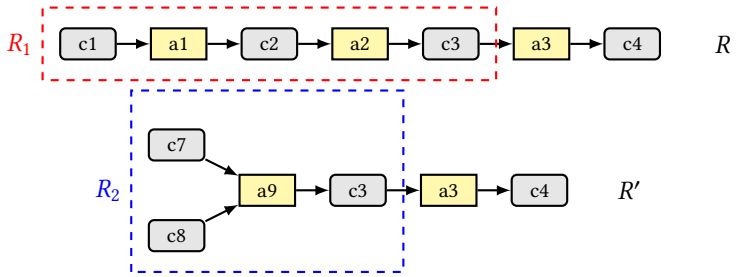


Fig. 14. Example of structural substitution. The top graph refers to recipe R, the subgraph in the top dashed box refers to R_1, the bottom graph refers to recipe R', and the subrecipe in the bottom dashed box refers to R_2. Hence $\text{Front}(R, R_1) = \{c3\} \subseteq (\text{In}(R_2) \cup \text{Out}(R_2))$, and conditions (ii)–(iv) of Definition 22 are also met. Suppose that condition (v) concerning the typing of consumable nodes is also met. Then $R' = R[R_1/R_2]$. Suppose further that R and R_2 are both acceptable recipes w.r.t. some set of acceptability tuples U, and that $F_1(c3) = F_2(c3)$ where $\text{Type}(R_1) = F_1$ and $\text{Type}(R_2) = F_2$. Then R' is also an acceptable recipe in which $F_2(c3)$ is produced from ingredients $F_2(c7)$ and $F_2(c8)$, as opposed to being produced from ingredient $F_1(c1)$.

node in R_2, thus ensuring that the revised typing function for R after R_2 has been inserted meets the necessary condition for typing functions stated in Definition 3 of a recipe. These five conditions ensure that the result of structural substitution is indeed a recipe, as shown in the following proposition.

Proposition 8. *For any recipes R, R_1 and R_2, if $R[R_1/R_2]$ is defined then it is a recipe.*

Proof. Assume conditions (i) to (v) of the definition for structural substitution hold. From (i) and parts 1 to 3 of the definition of structural substitution, we have for all $n \in \text{Nodes}(R) \setminus \text{Nodes}(R_1)$, and for all $n' \in \text{Nodes}(R_1)$, if $(n, n') \in \text{Arcs}(R)$, then $(n, n') \in \text{Arcs}(R[R_1/R_2])$, and if $(n', n) \in \text{Arcs}(R)$, then $(n', n) \in \text{Arcs}(R[R_1/R_2])$. So R connects with R_1 using the same arcs as $R[R_1/R_2]$ connects with R_2. From (ii), for all $c \in \text{Front}(R, R_1)$, if R_1 connects with c with an incoming (respectively outgoing) arc, then R_2 connects with c with an incoming (respectively outgoing) arc. So R_2 connects with the nodes in $\text{Front}(R, R_1)$ with arcs in the same direction as R_1, and furthermore from (iii), R_2 connects to all nodes in $\text{Front}(R, R_1)$. From (iv), the nodes in R_2 do not introduce any cycles with $\text{Nodes}(R) \setminus \text{Nodes}(R_1)$. From (v), and part 4 of the definition of structural substitution, F' is typing function for recipe according to Definition 3. Therefore, from these assumptions, $R[R_1/R_2]$ is a recipe. □

Provided a structural substitution into a valid recipe preserves the types of the (consumable) nodes it connects to, and provided the newly inserted sub-recipe is itself valid, the result of the substitution is also a valid recipe, as stated in the following proposition:

Proposition 9. *Let R, R_1 and R_2 be recipes that are valid w.r.t. some set of valid acceptability tuples U, such that $R[R_1/R_2]$ is defined and such that for all $n \in (\text{Nodes}(R) \cap \text{Nodes}(R_2))$, $\text{Type}(R)(n) = \text{Type}(R_2)(n)$. Then $R[R_1/R_2]$ is valid w.r.t. U.*

Proof. The proposition follows directly from parts 1 – 4 of Definition 22 and Definition 6 of a valid recipe. □

Structural substitution satisfies some simple properties including reflexivity (i.e.,$R[R_1/R_1] = R$), reversibility (i.e.,$(R[R_1/R_2])[R_2/R_1] = R$), and "totality", in the sense that $R[R/R'] = R'$. This last totality property implies that structural substitution subsumes type substitution, since it follows that for any type substitution pair (P, S), $R[R/(R \otimes (P \cup S))] = R \otimes (P \cup S)$. Structural substitution does not however satisfy transitivity, or associativity.

We can calculate the cost of any structural substitution, or sequence of structural substitutions, by using a distance measure. For a structural substitution $R[R_1/R_2]$, this could be a function of the graph edit distance

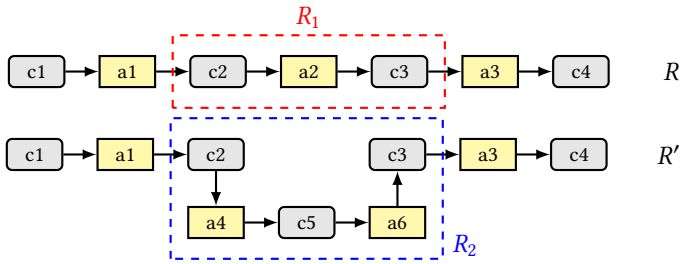


Fig. 15. Example of structural substitution. The top graph refers to recipe R, the subgraph in the top dashed box refers R_1, the bottom graph refers to recipe R', and the subrecipe in the bottom dashed box refers R_2. Hence, $\text{Front}(R, R_1) = \{c2, c3\} \subseteq (\text{In}(R_2) \cup \text{Out}(R_2))$, and conditions (ii)–(iv) of Definition 22 are also met. Suppose that condition (v) concerning the typing of consumable nodes is also met. Then $R' = R[R_1/R_2]$. Suppose further that R and R_2 are both acceptable recipes w.r.t. some set of acceptability tuples U, and that $F_1(c2) = F_2(c2)$ and $F_1(c3) = F_2(c3)$ where $\text{Type}(R_1) = F_1$ and $\text{Type}(R_2) = F_2$. Then R' is also an acceptable recipe in which $F_1(c3)$ is produced from $F_2(c2)$ using actions $F_2(a4)$ and $F_2(a6)$, as opposed to being produced using action $F_1(a2)$.

between R and $R[R_1/R_2]$, and the distance between the ingredients and actions in R and those in $R[R_1/R_2]$. We will investigate specific definitions in future work.

As with type substitutions, we can have structural substitutions that we regard as primary, in the sense that they are subgraphs that we want or need to replace. But these can also have knock-on effects that necessitate secondary substitutions. We define this below and illustrate in Figure 16. For the purposes of the definition, we assume that we have specified a numerical distance measure $d(R, R')$ between any two recipes R and R'.

Definition 23. *Let R_p be the recipe that is the result of applying the sequence $\sigma_p = [R_1/R'_1], \ldots, [R_j/R'_j]$ of structural substitutions to recipe R, and let R_s be a recipe that is the result of applying the sequence $\sigma_s = [R_{j+1}/R'_{j+1}], \ldots, [R_k/R'_k]$ of structural substitutions to R_p. Then σ_s is a* **secondary substitution sequence** *for recipe R w.r.t. primary substitution sequence σ_p and set of acceptability tuples U iff (i) R_s is acceptable w.r.t. U, (ii) for each atomic recipe R_a such that $R_a \sqsubseteq R_p$ and $R_a \not\sqsubseteq R$, then $R_a \sqsubseteq R_s$, and (iii) there is no alternative sequence σ'_s of structural substitutions that when applied to R_p would result in a recipe R'_s satisfying conditions (i) and (ii) and such that $d(R_p, R'_s) < d(R_p, R_s)$.*

Condition (ii) in Definition 23 is necessary to ensure that the secondary substitutions do not undo the effects of the primary substitutions, and condition (iii) is needed to ensure that the secondary substitution sequence represents a minimal 'fix' for the issues created by the primary substitution sequence.

10 Computational Recipes

Logic programming is a natural paradigm with which to represent the framework and definitions described in Sections 2 to 9 above, since it can then facilitate automated reasoning about individual or groups of recipes in a direct and transparent way. We illustrate by making available an Answer Set Programming (ASP) implementation of the majority of Definitions 1 to 23 written in the Potassco[13] system Clingo (Gebser et al. 2019), sufficient to perform a number of reasoning tasks including, for example, verification that a given labelled graph conforms to the definition of a recipe, verification that a recipe is valid w.r.t. a given set of acceptability tuples, and generation of preferred secondary substitution sets given a primary type substitution for a recipe. Included in the implementation is a portfolio of example recipes and types taken from the cooking domain. It is beyond the

[13] https://potassco.org/clingo/

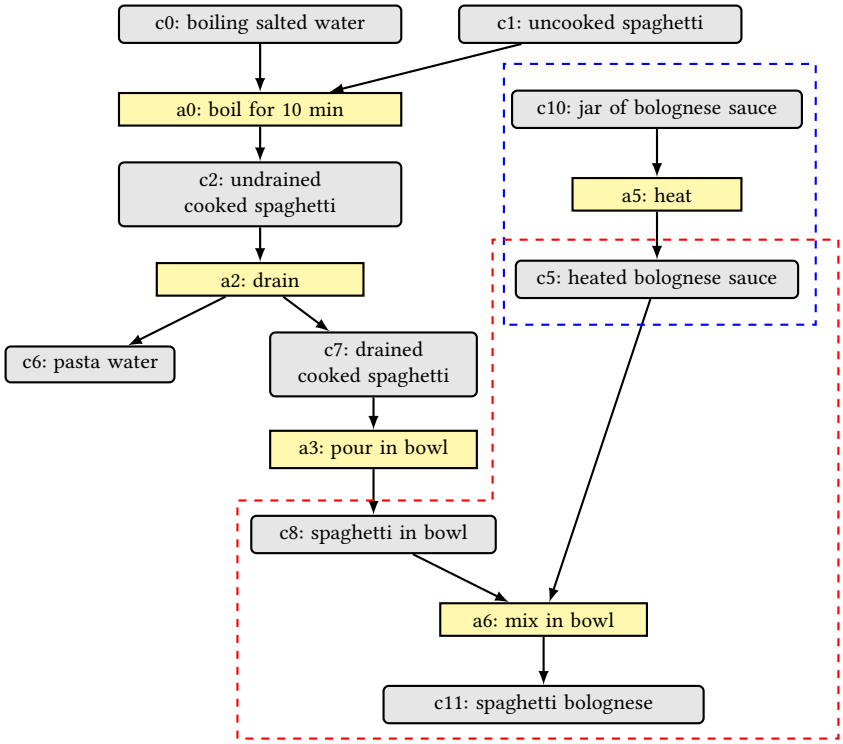


Fig. 16. Let R be the recipe given in Figure 4 with typing function F, and let $R_1 \sqsubseteq^* R$ be the recipe with graph given by the nodes $\{c3, c4, c5, a1\}$ in Figure 4. Let R_2 be the recipe corresponding to nodes within the blue box (the upper dashed box above). Let $R_1' \sqsubseteq^* R[R_1/R_2]$ be recipe with graph given by nodes $\{c5, c8, a4, c9\}$ and let R_2' be the recipe corresponding to nodes within the red box (lower dashed box above). Suppose $[R_1/R_2]$ is a primary substitution, and $[R_1'/R_2']$ is a secondary substitution, then the recipe above is $(R[R_1/R_2])[R_1'/R_2']$. Note that the type function for R_1 is F restricted to the nodes of the graph of R_1. The type function F_1' of R_1' matches the type function of $R[R_1/R_2]$ restricted to the graph of R_1'. More precisely, it matches F for $c8$, $a4$ and $c9$ but differs for $c5$, and so is defined as $F_1'(c8)$ = "spaghetti in bowl", $F_1'(c9)$ = "spaghetti pomodoro", $F_1'(a4)$ = "mix in bowl" and $F_1'(c5)$ = "heated bolognese sauce".

scope of this paper to give more than a brief overview of this implementation, but for more detail the interested reader may refer to the associated GitHub code repository[14], and in particular to the README file within the repository that explains the structure and different modes of usage of the code.

To give a flavour of the implementation, we describe here a module containing a domain-specific encoding of a recipe for spaghetti pomodoro (see Fig 4 at the end of Section 4), and the code modules containing domain-independent encodings of Definitions 1 (of a recipe graph) and 3 (of a recipe). The README file includes instructions on how to run these modules together within the Clingo environment to ascertain that the spaghetti pomodoro data structure does indeed conform to Definitions 1 and 3.

The top level ASP data structure for a recipe is given by the predicate `recipe/2`, where the first argument references the recipe graph and the second argument references the associated type function. Nodes and arcs

[14] https://github.com/lukedickens/asp_recipe_graphs

are identified as being within a recipe graph by the predicate `in/2`, and the typing of nodes is specified by the predicate `type_of/3`. For example, for spaghetti pomodoro we have:

```
recipe(rg_spaghetti_pomodoro,tf_spaghetti_pomodoro).
in(arcs(rg_spaghetti_pomodoro),arc(c(0),a(0))).
in(arcs(rg_spaghetti_pomodoro),arc(c(1),a(0))).
...
in(arcs(rg_spaghetti_pomodoro),arc(a(4),c(9))).
...
type_of(tf_spaghetti_pomodoro,c(0),"boiling salted water").
type_of(tf_spaghetti_pomodoro,c(1),"uncooked spaghetti").
type_of(tf_spaghetti_pomodoro,a(0),"boil for 10 min").
...
type_of(tf_spaghetti_pomodoro,a(4),"mix in bowl").
```

The action and consumable type hierarchies are declared with the predicate `child/2`:

```
child("spaghetti", "pasta").
child("uncooked spaghetti", "spaghetti").
...
child("boil for 10 min", "boil").
child("mix in bowl", "mix").
```

Domain independent definitions are implemented in a straightforward manner, largely in terms of ASP constraints. For example, Definition 1 of a recipe graph is coded as:

```
% C contains only consumable nodes:
:- in(c_nodes(RG),X), not is_c_node(X), recipe_graph(RG).
% A contains only action nodes:
:- in(a_nodes(RG),X), not is_a_node(X), recipe_graph(RG).
% E contains only arcs:
:- in(arcs(RG),X), not is_arc(X), recipe_graph(RG).

% Recipe graphs must have non empty consumable, action and arc sets
:- empty(c_nodes(RG)), recipe_graph(RG).
:- empty(a_nodes(RG)), recipe_graph(RG).
:- empty(arcs(RG)), recipe_graph(RG).

% Recipe graphs must not be cyclic
:- cyclic(RG), recipe_graph(RG).
% Recipe graphs must be connected
:- -connected(RG), recipe_graph(RG).
% Action nodes in recipe graphs must be properly connected
% (in and out going edges to consumable nodes)
:- -a_node_properly_connected(RG,a(N)), recipe_graph(RG).

% for each consumable node in a recipe graph, there is at
% most one incoming arc:
:- in(arcs(RG),arc(a(N1),c(N))), in(arcs(RG),arc(a(N2),c(N))),
   N1 != N2, recipe_graph(RG).
```

and Definition 3 of a recipe is coded as:

```
% the 3rd argument of type_of is a function of the first two arguments:
% range of function is the c_nodes and a_nodes of corresponding graph
```

```
1 { type_of(TF,c(N),Ctype) : consumable_type(Ctype) } 1 :-
    recipe(RG,TF), in(c_nodes(RG),c(N)).
1 { type_of(TF,a(N),Atype) : action_type(Atype) } 1 :-
    recipe(RG,TF), in(a_nodes(RG),a(N)).

% for each recipe, type_of is restricted to its nodes only:
:- type_of(TF,N,T), recipe(RG,TF), not in(nodes(RG),N).

% all the consumable node types in a recipe must be in different type paths:
:- recipe(RG,TF), in(c_nodes(RG),N1), in(c_nodes(RG),N2), N1 != N2,
   type_of(TF,N1,T1), type_of(TF,N2,T2), same_type_path(T1,T2).
```

where the auxiliary predicates used in these definitions such as `is_c_node/1`, `is_a_node/1`, `empty/1`, `connected/1`, `cyclic/1` and `same_type_path/2` are defined in a straightforward manner. Loading and running the modules containing the above code in Clingo produces a single answer set that includes the literal `recipe(rg_spaghetti_pomodoro,tf_spaghetti_pomodoro)`, thus confirming that the "spaghetti pomodoro" data structure does indeed satisfy all the constraints in the encodings of Definitions 1 and 3. The README file gives detailed instructions for this example. Other definitions in Sections 2 to 8 are implemented in a similar style. This example task (confirming a recipe structure) uses Clingo in theorem proving mode (using "cautious entailment" in ASP parlance) since it confirms a truth in all models (further remarks about this in the README file).

For some other computational tasks we instead want to use Clingo in a generative way, where an answer set can be considered in isolation as supplying a solution to a problem. An example is when we want the program to find sets of secondary type substitutions that give validity to a given primary type substitution. An illustration of this is given in the GitHub repository (and described in detail in the README file), in which the "uncooked spaghetti" ingredient is replaced with "uncooked fusilli". Using the encodings of the definitions in Section 8, Clingo is able to find an optimal (i.e. least cost) set of secondary type substitutions that restore validity to the new variation of the recipe in Figure 4. For this example, Clingo's `#maximize` operator is used as a particularly succinct way of providing a cost function – minimising the cost of the secondary type substitution by maximising the similarity between the original and altered recipes (see the file 'type_substitution.lp' in the GitHub repository for coding details).

To summarise the features of our implementation more generally, it includes:

- Syntax for declaring consumable and action type hierarchies as discussed in Section 3, and an extensive type hierarchy that defines consumables and actions based on real world recipes, including their sub-type relationships.
- Syntax for declaring a recipe graph as a bipartite graph of consumable and action nodes, and constraints to ensure it satisfies Definition 1, Section 3. A recipe graph, `recipe_graph(RG)`, will also be an atomic recipe graph, `atomic_recipe_graph(RG)`, if it additionally satisfies constraints corresponding to Definition 2.
- A type function definition which, for a given type function, maps graph nodes uniquely onto types.
- Syntax to declare recipes as required by Definition 3. A recipe, `recipe(RG,TF)`, combines a recipe graph, `RG`, and type-function, `TF`, and must satisfy all appropriate constraints, for example, that all nodes have appropriate types and no two consumable types are comparable.
- Syntax to allow a recipe, `recipe(RG,TF)`, to be defined as a given recipe, `given_recipe(RG,TF)` (in which case it is to be treated as background knowledge). Otherwise, it is simply a recipe. A recipe, `recipe(RG,TF)`, that is not a given recipe, is a candidate recipe, `candidate_recipe(RG,TG)`.
- A collection of example recipes expressed as given recipes and derived from real world recipes found online and manually encoded into ASP.

- Code to facilitate the declaration of acceptability tuples (see Definition 4) as a unique acceptability tuple key associated with a collection of input and output elements and a single action element. More precisely, `input_element_position(Key,CType,I)` associates acceptability tuple key `Key`, with consumable type `CType` as the `I`th such input type. Similarly for output types. Literal `action_element(Key,AType)` associates acceptability tuple key `Key` with the single action type `AType` for the tuple. An acceptability tuple can be declared a given acceptability tuple; if it is not then it is by default a candidate acceptability tuple.
- A facility for executing the code such that it extracts given acceptability tuples from given recipes and candidate acceptability tuples from candidate recipes.
- A facility for executing the code in an alternative mode such that it can evaluate a candidate recipe in terms of validity. This requires background knowledge in the form of given acceptability tuples (either extracted from given recipes or declared directly). A candidate recipe is found to be valid if all extracted candidate tuples match at least one valid acceptability tuple from the background knowledge.
- Where there is a given recipe `given_recipe(RG,TF)` defined, a facility to request a type substitution using a literal of the form `type_substitution(RG,TF,PartSpecTF)`, declaring primary substitutions in the form `primary_substitution((RG,TF,PartSpecTF),Node,Type)`. In this case, `given_recipe(RG,TF)` is treated as background knowledge. Subsequent execution then finds suitable types as completions of the partially specified type-function `PartSpecTF` and reports any that differ from type-function `TF`, that are not themselves primary substitutions, as secondary substitutions `secondary_substitution((RG,TF,PartSpecTF),N2,T2)`. (See Definitions 15 and 18 in Section 8.) Note that the user must typically provide sufficient additional given acceptability tuples (or other given recipes) such that a valid completion of `PartSpecTF` exists. If a type substitution is found then it has minimal cost with respect to the size of the secondary substitution set (assuming the cost is defined in terms of $0-1$ distance for types)[15].

Code supporting the comparison of recipes (Section 6), composition of recipes (Section 7), structural substitution (Section 9) and acceptability tuple inference (Definition 6 from Section 5) are planned for future iterations of the code base.

10.1 Experimental Evaluations

In this section we provide an analysis of the computational behavior of our approach under various scenarios. We first experiment with the computational cost of searching for all acceptability tuples represented by a recipe graph and the associated type function. We call this process recipe tuple discovery. Figure 17 demonstrates the relationship between the *number of discovered acceptability tuples* and the CPU time required for acceptability tuple discovery. Different choices of the clingo constant `max_input_set_size` are shown in different colours. For a fixed value of `max_input_set_size`, the trend appears linear as the number of tuples increases. A similar analysis is presented in Figure 18, which examines the dependence of CPU time on the *total number of input and output elements across all discovered tuples*. The trend appears again linear, modulo an increased standard deviation across different runs on the same recipe. Figure 19 explores the impact of the `max_input_set_size` on CPU time during acceptability tuple discovery. The constant `max_input_set_size` determines the maximum number on input elements that clingo will consider for any given acceptability tuple, and thus has a significant impact on the size of the search space. Here, results appear linear in log-scale, which highlights the exponential relationship between this parameter and CPU time. A similar constant `max_output_set_size` determines the maximum number of output elements in any tuple. We did not experiment with this as we expect its influence to closely mirror that of `max_input_set_size`.

[15]Recall from Section 8 that a more sophisticated approach might use path distance within the type hierarchy between the type in `TF` and the type in `PartSpecTF` as the contributing cost of an element within the secondary substitution set.

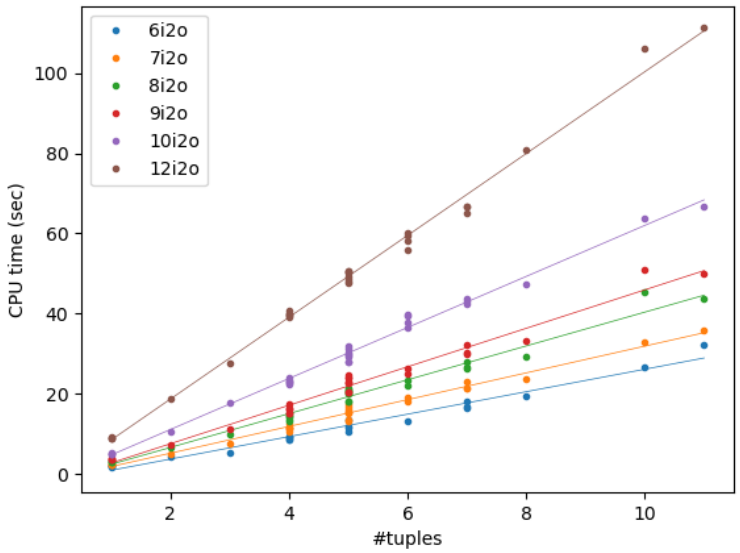


Fig. 17. Number of acceptability tuples versus CPU times for acceptability tuple discovery. Each point represents a single run of clingo using the default configuration, and applying acceptability tuple discovery on a single recipe, where the x-position show the number of discovered acceptability tuples (action nodes within the graph) and the y-position is the CPU time for the execution. Different lines correspond to different conditions based on the size of the allowed maximum size of the extracted input and output sets, for instance 6i2o means input sets upto size 6 and output sets up to size 2.

We then focus on type substitution. Figure 20 examines the dependence of *CPU time* on the *number of acceptability tuples provided during substitution*, whereas Figure 21 provides a complementary view by analysing the *solving time*, as reported by `clingo`, for the same substitution tasks. For each run of these experiments (a single point in the plots), a type substitution query is made on a given recipe, which forms part of the background knowledge. Additional background knowledge is provided in terms of given acceptability tuples. All experiments provide sufficient background knowledge to support a successful execution of the query, while additional given acceptability tuples simply enlarge the search space. The x-position of the points is determined by the total number of given acceptability tuples for the query, including those provided by the given recipe. We constrain the type hierarchy to those types that appear within the given recipe or acceptability tuples, as well as any ancestral types. The colour of each point corresponds to a type substitution for a specific given recipe with known solution.

Finally, Figure 22 investigates how solving time depends on different configurations of the `clingo` solver, again in log-scale. Note that different clingo configurations can be run via the `--configuration=<arg>` flag, where `arg` is one of `crumpy`, `frumpy`, `handy`, `jumpy`, `trendy`, and `tweety`. This is used to select predefined configurations that optimise the performance of the `clingo` solver for different types of problem instances. Each configuration adjusts internal solver parameters, heuristics, and strategies to improve solving time and resource usage based on the nature of the problem being solved, e.g., their size, simplicity, symmetry, etc.

These analyses collectively highlight key computational characteristics of our approach. Specifically, the experiments show how computational costs increase with parameters such as the number of acceptability tuples, input and output elements, and maximum input set size. Figures 17 and 18 show that the overall behavior of

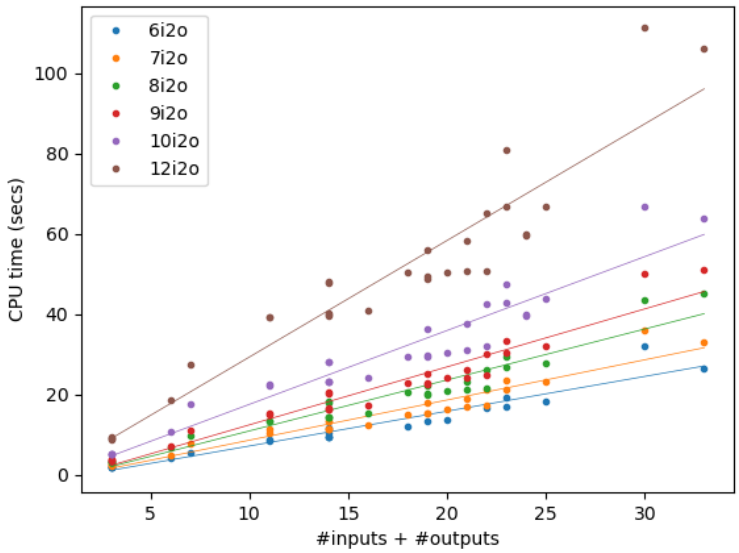


Fig. 18. Number of input and output elements versus CPU times for acceptability tuple discovery. Each point represents a single run of clingo using the default configuration, and applying acceptability tuple discovery on a single recipe, where the x-position show the sum of input and output elements for all discovered acceptability tuples and the y-position is the CPU time for the execution.

clingo seems to be linear in the number of tuples, and input and output elements. Nonetheless, the trend becomes exponential with respect to the maximum input set size as depicted in Figure 19. Interestingly, a comparison between solving and total CPU time for a varying number of acceptability tuples, as shown in Figures 20 and 21, seems to suggest that solving time, in the worst-case (`spaghetti_pomodoro`), increases more rapidly than total CPU time. This may indicate that for small-to-medium instances, grounding time dominates the total CPU time, but the fast growth of solving time highlights the importance of optimising the solving process, as this aspect could become predominant for larger instances. In this respect, we experimented with solving configurations as shown in Figure 22. We only show solving time as configurations do not affect grounding time. Except for `frumpy` (the most conservative solving strategy), most configurations appear to perform similarly. These findings demonstrate that the provided ASP implementation, although affected by high computational costs given its exact nature, is feasible for small to medium-scale problems. Furthermore, let us recall again that it provides a sound basis for evaluating approximate, more efficient methods in future work.

10.2 Discussion of Experimental Investigation

It can be seen from the above experiments that the proposed method for substitution, which involves exploring all possible substitutions for a consumable and their cascading effects, can become computationally prohibitive as the consumable type hierarchy and the set of acceptability tuples expand. To address this within ASP, one possible approach is to prioritise substitutions that are more likely to succeed. These could include common substitutions found in domain-specific datasets (e.g., Recipe1M) or resources like foodsubs.com, as well as recipe-specific solutions previously computed by the system during substitution tasks. In the current implementation, we use

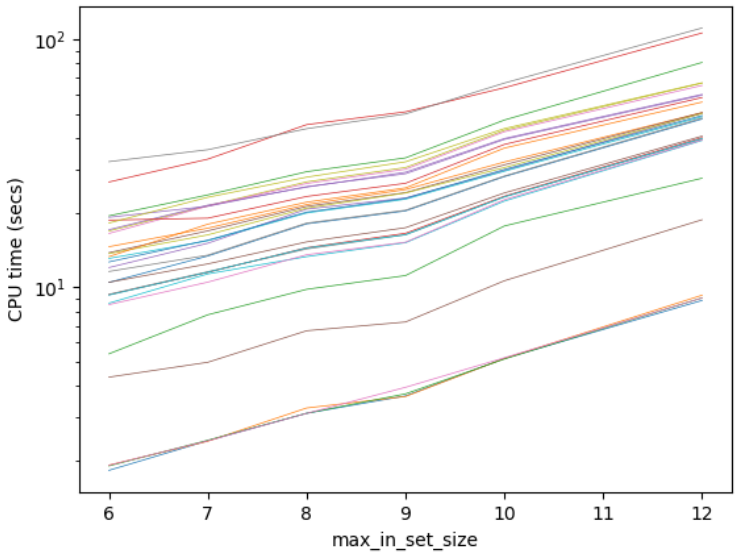


Fig. 19. CPU time dependence on maximum input set size for acceptability tuple discovery. Each line consists of a series of points each a single run of clingo using the default configuration, and applying acceptability tuple discovery on a single recipe. The x-position of each point shows the value of the `max_input_set_size` parameter for that run and the y-position is the CPU time for the execution. Y-axis is in log-scale to highlight the exponential relationship between independent and dependent variable.

custom methods to improve computation time, both for acceptability tuple discovery and type substitution; namely, we compress the type hierarchy only using types necessary for the provided recipes and/or tuples (this can have a large effect on grounding time). The empirical evaluation shows, in fact, that grounding time typically dominates solving time for small to medium problem instances. More generally, we reduced the number of non-ground variables in our domain-independent code as well, which has significantly decreased total computation time.

However, note that our motivation for providing an ASP implementation is to empirically test the theoretical definitions and demonstrate that we can run them using case study examples, not to provide an efficient or scalable implementation. In our opinion approaches to scalability, including those involving suboptimal solutions within ASP, are best addressed within the context of specific domains. As such these fall outside of the scope of our current work. In future work, we may explore different implementation approaches that are more scalable. This could involve leveraging specialised graph database systems like Neo4j or developing algorithms in more efficient programming languages, such as C++.

11 Domain Areas and Scope of the Framework

Thus far in this paper, we have presented examples for our substitution framework mainly in terms of cooking recipes. In this section, we broaden the discussion, and demonstrate that our framework can apply to a broad variety of domains. We illustrate this with a number of examples including those relating to DIY, gardening, industrial processing, and disaster management. We will show that these examples present the same essential

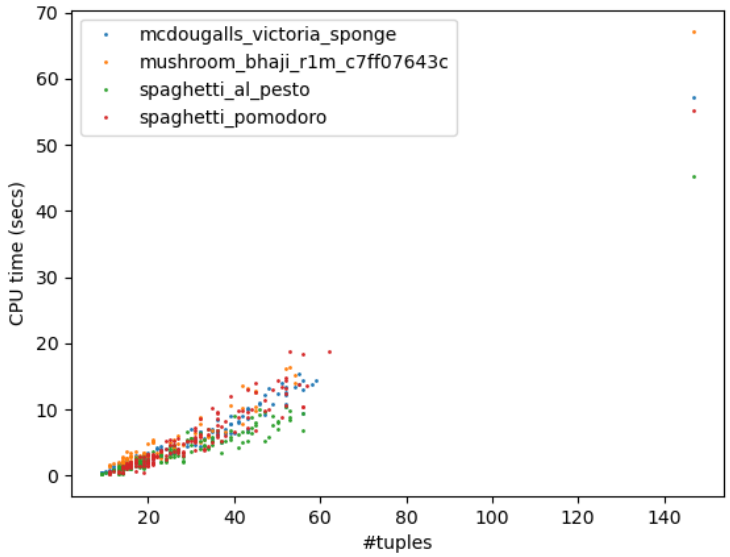


Fig. 20. CPU time dependence on number of given acceptability tuples for type substitution. Each point represents a single run of clingo using the default configuration, and applying type substitution for one of 4 different base conditions, but for different quantities of additional given acceptability tuples that do not influence the final type substitution and a type hierarchy limited to the types within the base recipe and all super-types. The x-position reflects the total number of given acceptability tuples for the associated query, including those within the given recipe. The y-position is the CPU time for the execution.

qualities so far discussed for recipes, namely the consumption (modification, combination and transformation) of consumables into different and new forms, with input, intermediate and outputs materials. We also indicate how the need for substitution naturally arises within each of these domains and some specific challenges that emerge when attempting to address particular domains.

11.1 Domestic and Small Scale Making

Consider first small scale procedures that might be carried out in a domestic environment or a workshop. This covers domains such as DIY, gardening, and other forms of manual making such as pottery. Here we present just three examples:

- **Building a shelf unit:** Figure 23 represents the construction of a simple open shelf unit made from planks of wood available at lengths of 1800mm. The wood is cut into the appropriate lengths, then joined into a frame, before the shelves being fixed internally within that frame.
- **Growing Tomatoes:** Figure 24 presents a recipe for growing and harvesting tomatoes from seeds. This involves first propagating the seedlings, potting these into individual pots, then hardening off the plants by leaving them outside under controlled conditions, before planting them out, allowing them to ripen and harvesting the resulting crop.

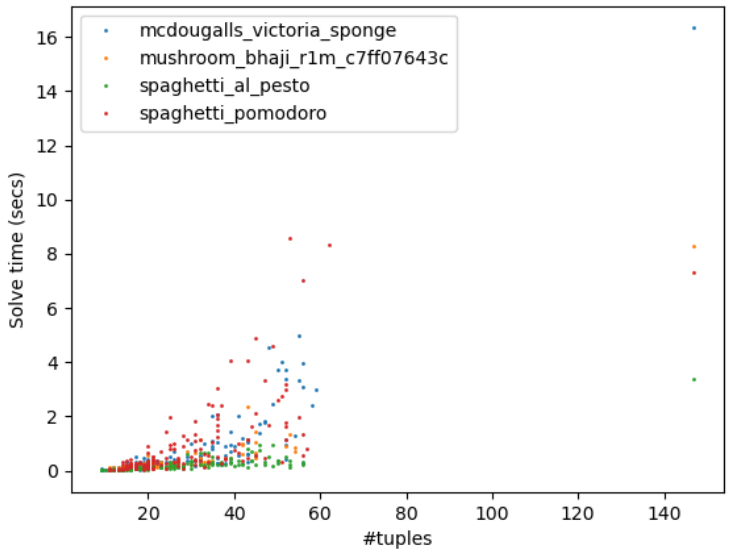


Fig. 21. Solving time dependence on number of given acceptability tuples for type substitution. Points as in Figure 20 but here y-position shows solving time as reported by clingo.

- **Making a ceramic bowl:** Figure 25 presents a recipe for making a ceramic bowl, including the shaping of the clay, as well as a number of subsequent steps required to prepare and condition the clay for the final product.

The need for substitution is apparent in each of these examples, just as it is in cooking. This could be when making do with limited or constrained materials, e.g., different length planks when making a shelf, and this may relate most simply to type substitution as described and illustrate throughout the document so far. Substitutions here can be more general too, such as adapting recipes to make similar items, e.g., building a cupboard rather than a shelf, growing cucumbers rather than tomatoes, or making a cup rather than a bowl, and reasoning about these recipes may be more generally achieved with structural substitution as the numbers of consumables or actions may change as may the structure of the graph. Finally, substitution may relate to reasoning about different recipes that achieve the same ends, e.g., a different cutting order for the shelf, or growing method for the tomatoes. Such alternative recipes with the same ends would be in-out aligned, with the original recipe, e.g., a method for growing tomatoes to maturity within a greenhouse, and again reasoning about these recipes would be a form of structural substitution.

11.2 Industrial Procedures

Here we show that our formalism can be applied to industrial resource transforming procedures just as it can be applied to more domestic procedures. One key difference for these recipes are that they often refer to a continuous pipeline through which resources are transformed, rather than a series of discrete steps that might be expected to be performed in isolation as in the cooking domain. Nonetheless, these industrial procedures can be readily captured by our framework. Here we give just two examples:

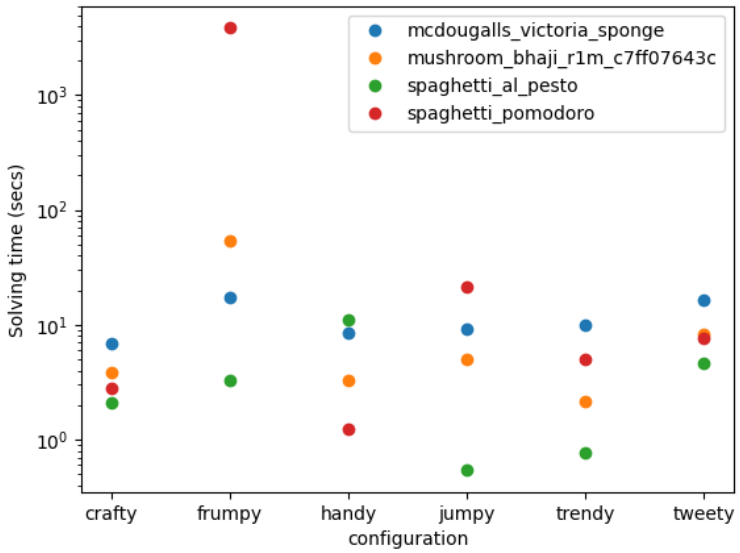


Fig. 22. Solving time dependence on clingo configuration. Points indicate a single run of type substitution on one of 4 cases with largest set of additional files. The y-position indicates solving time on a log scale.

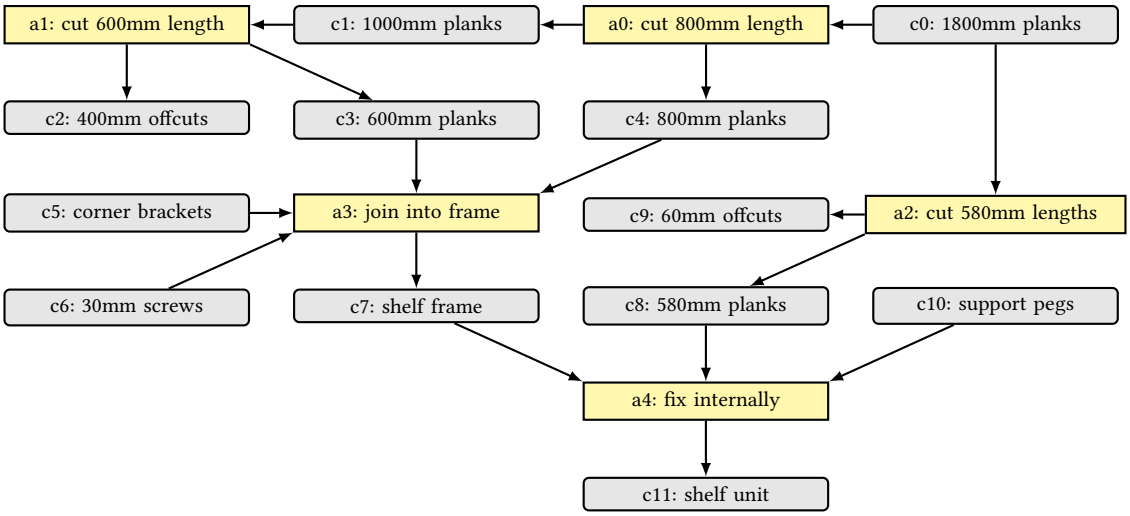


Fig. 23. A recipe for constructing a DIY shelf unit.

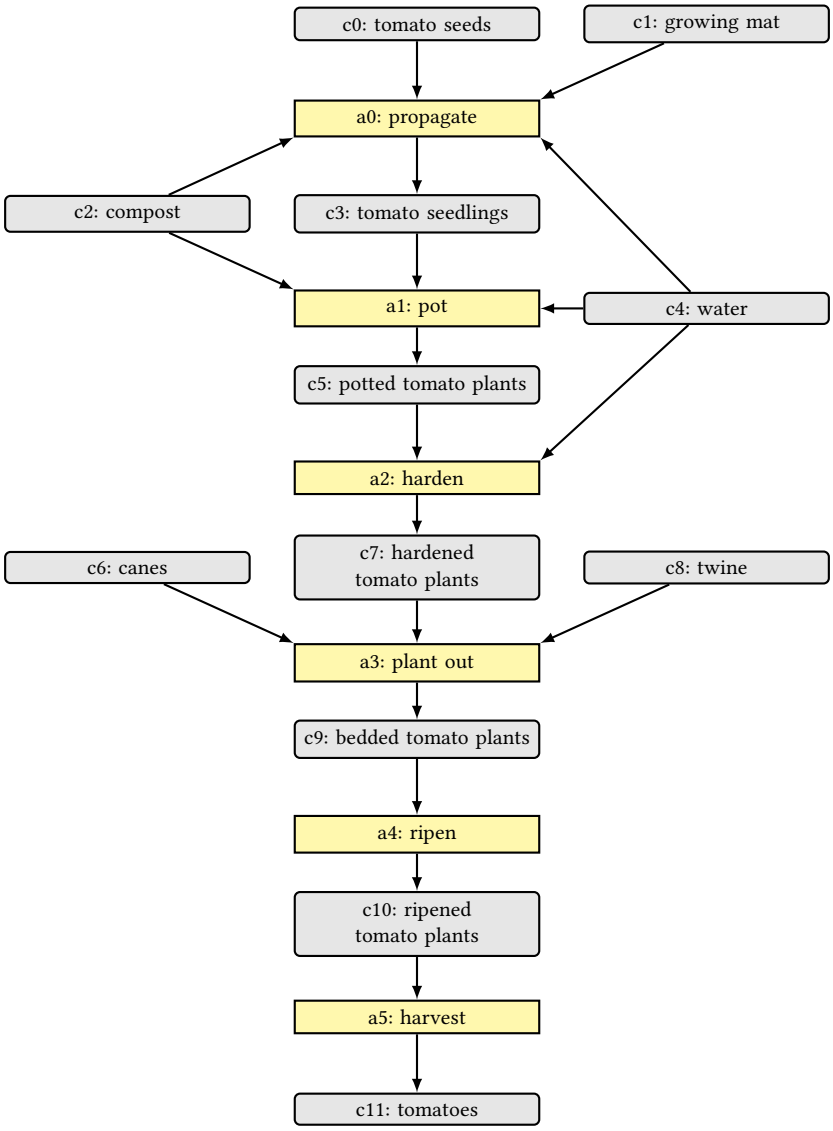


Fig. 24. A recipe for growing tomatoes from seeds.

- **Steel making:** Figure 26 presents a recipe of steel making and is a simplification of the Basic Oxygen Steelmaking (BOS) process as described at `http://britishsteel.co.uk/what-we-do/how-we-make-steel/`. In this recipe, coke, iron ore, limestone and sinter (an agglomeration of iron ore fines, fluxes such as limestone, and coke fines) charge a furnace which produces bloom – a mixture of molten iron and waste (slag). The iron is tapped off and then purified before being blasted with high purity oxygen to form steel.
- **Industrial water purification:** Figure 27 presents a typical industrial water purification recipe. Here untreated or raw water is first mixed with coagulants (here alum) and gently mixed (flocculation) to

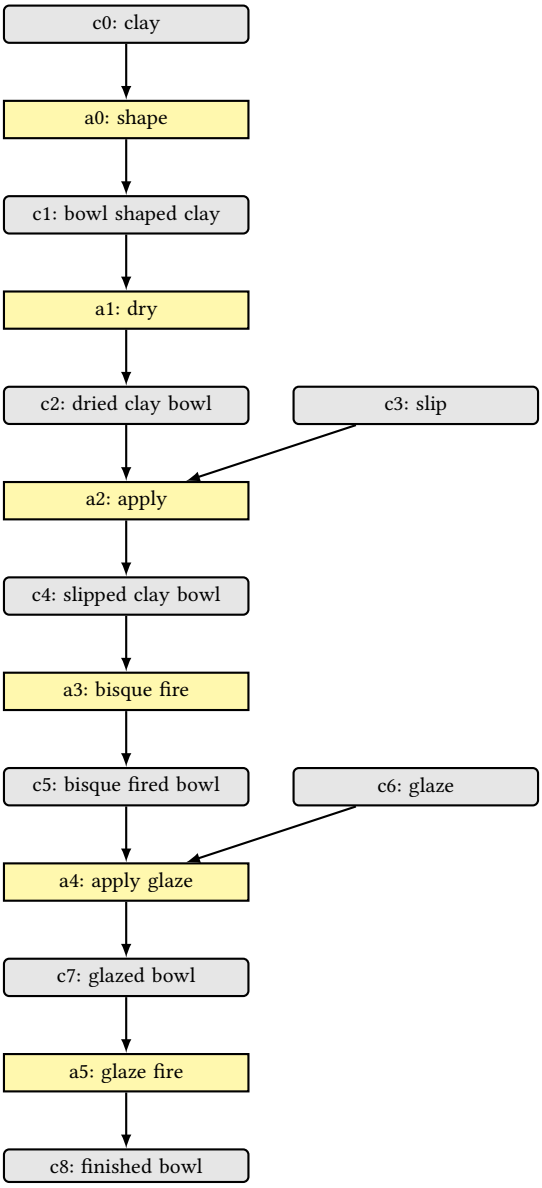


Fig. 25. A recipe for making a ceramic bowl.

encourage the formation of large suspended particles which can then be extracted by letting them settle out (sedimentation). The clarified water is then filtered and disinfected before it is ready to drink.

Due to the nature of these procedures, a great deal of investment has been put into developing appropriate alternative methods, and cost saving adaptations. As such, there are numerous known substitutions that would also lead to reasonable alternative recipes. For instance in the water purification process, alum can be substituted

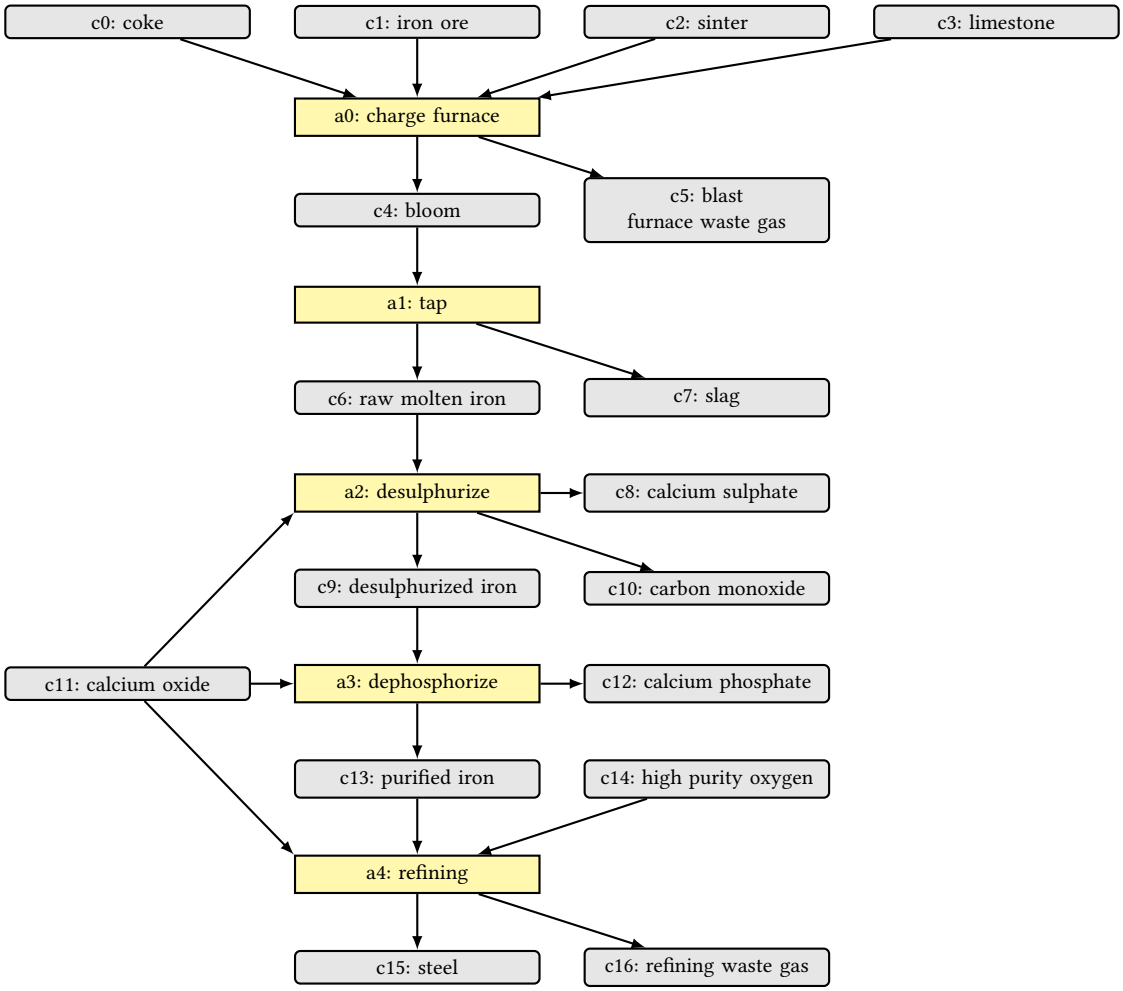


Fig. 26. A recipe for steel making.

with ferric chloride as a coagulant, and chlorine can be substituted with ozone as a disinfectant. These choices come with various implications in terms of costs, availability, quality of output and potential need for mitigation steps. For instance, in the case of industrial water purification, a more complex change would be to replace the single action of filtration with a two step recipe involving simple filtration followed by activated carbon filtration, which can improve the effectiveness of the later disinfection step, and an additional activated carbon filtration step can follow disinfection with carbon to improve taste and odour of the final product. Moreover, due to the scale at which these recipes are implemented, the implications of small differences in costs and required processing of by-products in this domain may have greater influence on decision making than in a domestic domain, and another avenue of future work might be to enrich the current framework with quantities and costs to facilitate such reasoning. Nevertheless, there may be good quality knowledge bases available to facilitate reasoning about substitutions in these domains.

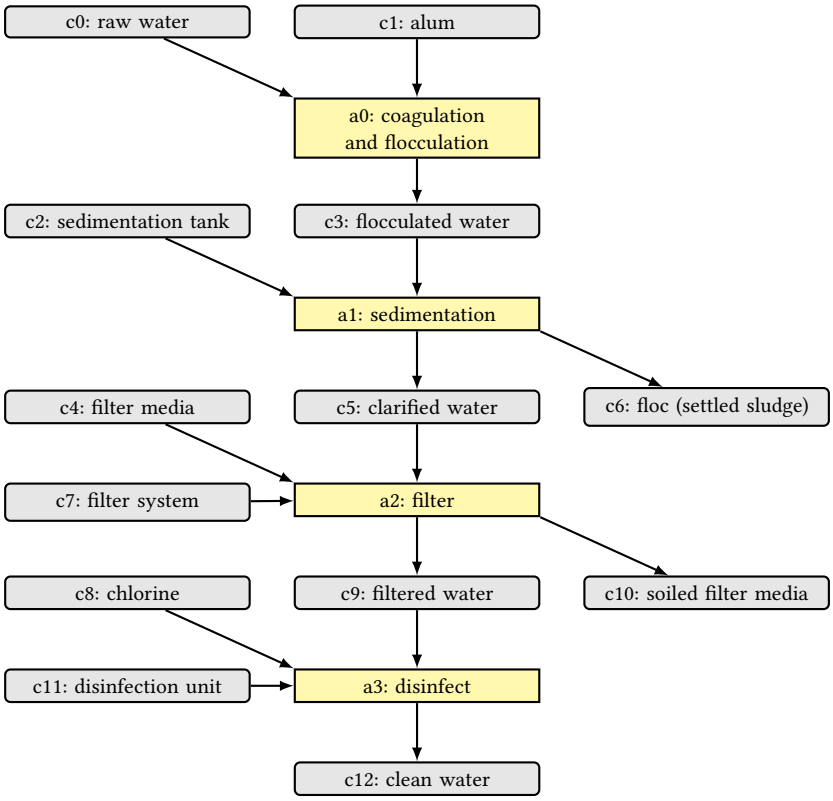


Fig. 27. A recipe for industrial water purification.

11.3 Crisis Response and Disaster Management

We present here examples of recipes more suited to a crisis response or disaster relief scenario. This domain is characterised by the need to be particularly creative or flexible in order to accommodate extreme and/or heterogeneous constraints that can arise after natural or man-made disasters, and where the need to preserve life may trump usual acceptability limitations. Here we present two simple representative examples:

- **Constructing a makeshift shelter:** Figure 28 represents the construction of a makeshift shelter, where only rope, tarpaulin and metal pegs are available.
- **Domestic water purification:** Figure 29 presents a recipe for water purification as might be required in a crisis situation such as after a natural disaster. This is somewhat analogous to the recipe shown in Figure 27 but here some of the steps are adapted, replaced and even omitted, and refer to actions that can be carried out by an individual without complex equipment.

Consider first the construction of a makeshift shelter, which might be already regarded as an alternative to (substitution in place of) a more conventional temporary structure constructed from a tent frame and tent shell. Nonetheless, there remain numerous opportunities for substitution. For instance, the recipe requires metal pegs, which might not be available. Most simply, wooden pegs might be used in their place, but other alternatives are available if ready made wooden pegs are also unavailable. This may require subrecipes to be added in place of the loose metal peg node to represent cutting wooden pegs from scavenged wood (a form of structural substitution).

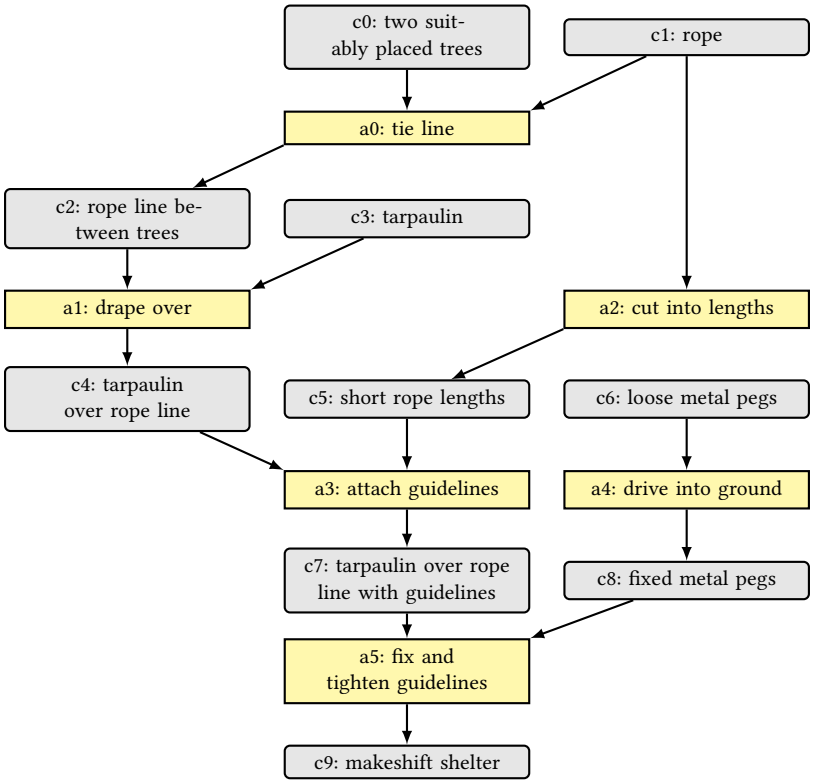


Fig. 28. A recipe for constructing a makeshift shelter in a survival scenario or crisis situation.

While wooden pegs may be suitable in a wooded environment with soft ground cover, alternative substitutions may be required due to the unsuitability of the ground for pegs, e.g., when on rock or concrete. In this case, heavy weights might be used. This could require some additional making steps too if ready made weights aren't available, such as the filling of bags with loose soil or sand.

As with the makeshift shelter, the water purification recipe may not be suitable in certain situations due to availability or other constraints. Note that the primary input consumable (raw water) and output (clean water) of this recipe matches that of the industrial water purification recipe from Figure 27. This presents opportunities to adapt knowledge from the industrial domain (which will be heavily researched) into the crisis response setting.

12 Related Work

In this section we discuss related work in two areas: representation and reasoning about recipes; and reasoning about substitution. Most of the approaches presented are from the culinary domain, but, as we have argued throughout this paper, these can be adapted to other goal-directed procedures that transform resources from raw materials into finished products.

12.1 Representation and Reasoning about Recipes

The "Computer Cooking Contest" (CCC), which ran from 2008 to 2017, focused on the the adaptation of cooking recipes using Case-Based Reasoning (CBR) techniques. Many of these approaches, for example (Blansché et al.

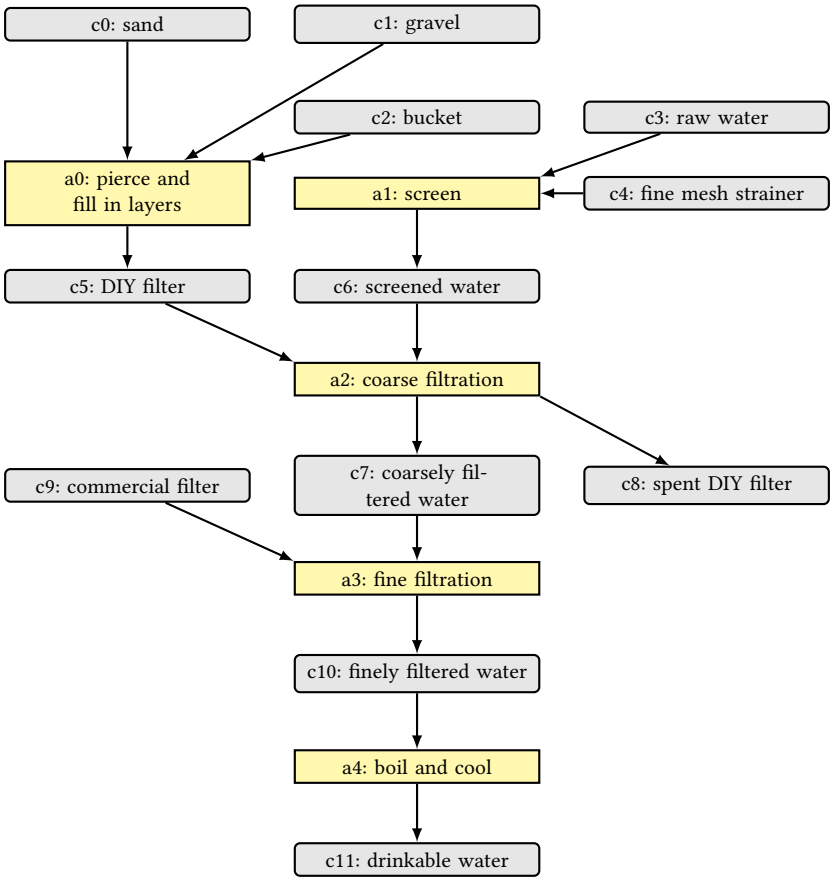


Fig. 29. A recipe for water purification in a crisis situation.

2010; DeMiguel et al. 2008; Herrera et al. 2008; Mota and Díaz-Agudo 2012; Müller and Bergmann 2015; Newo et al. 2010), rely on an ontology-based representation of recipes. Each ingredient as well as the final product of a recipe are commonly represented as separate entities. This does not however hold for the preparation steps, which in most cases have the form of a single block of text. Because of that, they address recipe adaptation taking into account only the ingredients of the recipe, but not the cooking actions used to process the ingredients. Two notable exceptions are the TAAABLE (Blansché et al. 2010; Dufour-Lussier et al. 2012) and CookingCAKE (Müller and Bergmann 2015) systems. The former uses a tree representation of recipes, where nodes represent recipe ingredients or outputs of cooking actions and edges represent cooking actions. It uses 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 (Dufour-Lussier et al. 2012); 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 (Blansché et al. 2010). Their tree representation is similar but less expressive than the recipe graphs that we propose (for example, it cannot model by-products of an action) and cannot support some of the reasoning tasks that we address such as the comparison or composition of recipes. The RDF representation of recipes proposed more recently by (Gaillard

et al. 2017) 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 based on the representation of recipes as workflows and the use of Process-Oriented Case-Based Reasoning (POCBR) techniques (Müller and Bergmann 2015). The adaptation methods it uses are based on decomposing workflows into meaningful sub-components called workflow streams. 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 (actions) or data (input materials). Its methods are based on similar ideas with ours, especially with respect to the composition of recipes and structural substitution. It does not, however, provide methods for validating 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.). 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 (Bergmann et al. 2013; Mori et al. 2014; Pan et al. 2020; Schumacher et al. 2012; Shao et al. 2009; Yamakata et al. 2020); their aims, however, are different from ours. Most of them focus on extracting workflow graphs from recipe texts using NLP techniques (Mori et al. 2014; Pan et al. 2020; Schumacher et al. 2012; Yamakata et al. 2020). The system described in (Shao et al. 2009) aims at facilitating the retrieval of recipes from workflow repositories, while (Bergmann et al. 2013) focuses on clustering recipes. Graphs for representing recipes were also used in (Chang et al. 2018; Wang et al. 2008). Similarly to our recipe graphs, they modelled both consumables and actions as nodes; however, their graphs do not include intermediate products or by-products. Their aims were also different; (Wang et al. 2008) developed methods for retrieving recipes from the Web, while (Chang et al. 2018) aimed at the analysis of recipes, for example to identify usage patterns of ingredients and cooking actions and to compare recipes.

FoodKG is a large-scale food knowledge graph, which integrates nutrition information, general food substitutions, recipe data and food taxonomies (Haussmann et al. 2019). It describes recipes as sets of ingredients, each 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.

A representation of recipes as plans was proposed in (Pallagani et al. 2022). 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 a promising alternative; however, reasoning about recipes is left by the authors for future work.

12.2 Reasoning about Substitution

There are a number of proposals that explore how substitutions could be found for ingredients in cooking recipes. For example, (Shidochi et al. 2009) describes a process of analysing existing recipes, cooking actions and the ingredients that they are commonly applied to, and how this analysis can then be used to identify candidates for substitution. 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 to collect user-generated suggestions for functionally equivalent ingredients, and for healthier variants of a recipe (Teng et al. 2012). Substitutions can also be identified by combining explicit information about the ingredients in FoodKG, and implicit information from word embeddings (Shirai et al. 2021). 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 (Blansché et al. 2010; DeMiguel et al. 2008; Herrera et al. 2008; Mota and Díaz-Agudo 2012; Müller and Bergmann 2015;

Newo et al. 2010). 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 (DeMiguel et al. 2008; Newo et al. 2010). 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.

An ontology design pattern for ingredient substitution in recipes was proposed in (Ławrynowicz et al. 2022). 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.

A formalism that captures features of objects (namely, shape, material, and role of the object) and enables reasoning with that knowledge to identify replacements or alternative uses of objects was proposed in (Olteteanu and Falomir 2016). This approach can be valuable when rich knowledge about the objects and their potential uses is available. However, it does not account for dynamic aspects of the problem, such as the transformation of an object, and for more sophisticated kinds of substitution such as structural substitution that we address in this paper.

Finally, a proposal that has some connections with our work is a logic-based approach to activity recognition based on a logic programming implementation of Event Calculus presented in (Artikis et al. 2010). 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 substitution or any other reasoning tasks that are specific to recipes, mainly due to the lack of an explicit representation of consumables and their relations with transforming actions, it does provide some ideas that we might implement in future extensions of this work, e.g., to model and reason about the temporal aspects of recipes.

12.3 Other Process Modelling Languages and Frameworks

There are other formalisms that address some of the needs that we are concerned with in this paper. There are a variety of process modelling formalisms such as Business Process Model and Notation[16] (BPNM) which provides a graphical notation for representing business processes. The notion of a recipe could be captured in this notation, which, however, does not provide methods for analysing recipes, including atomic recipes, isomporphic recipes and composition of recipes, connecting with type ontologies, and supporting different types of substitution.

Other kinds of processing modelling formalisms include process algebra (e.g.,(Hoare 1985; Milner 1989; Milner et al. 1992)) which aims to model interactions between processes in terms of message passing using a small set of primitives, and Petri nets (Petri 2005) which aim to model distributed systems in terms of a bipartite graphs with nodes for places and for transitions. These formalisms offer the ability to represent and reason with concurrent/distributed aspects of recipes (for instance, what parts of a recipe can be undertaken in parallel) but, compared to our approach, they have the same limitations as BPMN discussed above.

Recipes could be regarded as plans that involve resources (which we refer to as consumables) and actions. It is therefore possible to consider harnessing AI planning systems with deterministic models (see for example, (Ghallab et al. 2016)) to generate recipes. However, to do this, we would need to define the constraints on these plans so that they conform to our specification of a recipe, including the connection with the type hierachy. Furthermore, using AI planning systems does not provide methods for analysing recipes, including atomic recipes, isomorphic recipes, composition of recipes, nor methods for supporting different types of substitution. Hence, we may consider AI planning systems are complementary tools to what we have provided in this paper.

Another graph-theoretic approach to modeling resource transforming procedures is the P-graph framework (Friedler, Tarján, et al. 1992). It was originally developed for the modeling, analysis, and optimisation

[16] https://www.omg.org/spec/BPMN/2.0/PDF

of chemical processes, but it has also been applied to other domains such as energy conversion, supply chain management, waste and resource management and production lines (see (Friedler, Aviso, et al. 2019) for a comprehensive review). Similar to our model, P-graphs are bipartite graphs consisting of two types of nodes: operations and materials. However, our recipe graph model is more expressive, as it assigns types to the nodes. This enables more sophisticated reasoning for tasks such as recipe composition, comparison, validation and substitution. The wide range of applications of P-graphs suggests that our recipe model is applicable to any domain involving processes that transform resources into products. Moreover, the declarative implementation presented in Section 10 facilitates the integration of and reasoning with domain-specific expert knowledge, enabling a knowledge-based engineering methodology commonly used across a wide range of industries including aerospace, automotive, shipbuilding, electrical engineering and manufacturing (Quintana-Amate et al. 2015; Stjepandić et al. 2015).

We also see that abstraction, defined as the process of mapping a representation of a problem onto a new representation (Giunchiglia and Walsh 1992), is very relevant to modelling and reasoning with recipes. Our model allows for creating more abstract or concrete representations of a recipe by leveraging type hierarchies or the structure of recipe graphs. Specifically, for two recipes R_1 and R_2 where R_1 is finer-grained than R_2 (see Definition 10), R_2 can be seen as an abstraction of R_1: all actions and consumables in R_1 are mapped to corresponding elements in R_2, with no unmapped elements in R_2. For instance, the recipe in Figure 7 is an abstraction of the one in Figure 2. Definition 11 describes another form of abstraction, which uses the type hierarchies. Here, for two recipes R'_1 and R'_2, such that R'_1 is more specific than R'_2 (meaning that they both have the same structure, but the type of each node in R'_2 is the same with or higher in the type hierarchy than the type of the corresponding node in R'_1), R'_2 can be seen as an abstraction of R'_1. This notion of abstraction also underpins the validation of recipes (Proposition 9) and type substitution (Definition 14), both of which utilise type hierarchies. In future work, we plan to study more closely how our model relates to existing theories of abstraction in AI, such as those in (Banihashemi et al. 2017; Giunchiglia and Walsh 1992), and develop extensions of our model using such theories.

13 Conclusion and Future Work

As argued, numerous human activities involve reasoning about physical materials and actions upon them including the need to adapt procedures in the presence of constraints. This paper presents a framework for such resource transforming procedures, which we have referred to as recipes. Recipes might therefore refer to industrial or manufacturing processes, crafting or making techniques, DIY instructions, and cooking recipes. Here, we have provided a formalism based on labelled bipartite graphs for representing and reasoning with recipes, including a range of definitions for comparing recipes, and two ways of making substitutions in recipes, namely type substitution and structural substitution. Type substitution can be used to represent small changes in a recipe, such as a change in an individual consumable or action, and can also accommodate secondary adjustments needed as a consequence of the change. In contrast, structural substitutions allow for larger changes that can involve whole parts of the recipe being changed in one step, and can involve changes in the underlying graph shape.

Our formalisation can be used as a basis for automated analysis, comparison and manipulation of recipes. In particular, we have shown how we can implement it in ASP as a proof of concept. Using our implementation, we can determine whether a recipe is consistent, or incomplete, or what are possible recipes that are consistent with a set of constraints. We envisage implementations of our formalism being integrated, for example, with automated planning systems so that a recipe can be transformed into a specific plan for execution, taking into account constraints on time and resources, or with robotic systems so that a recipe can be executed by a physical robot to produce the required output.

At present we allow text about timings and locations in the type hierarchy. In future work, we plan to further develop the approach we present in Section 3.1 regarding parameterisation of actions to include consideration of

equipment and location, as well as timing of actions and quantities of consumables. This would allow us to draw these timings and locations out so that we can reason and analyse them. For example, if we have atomic recipe R_1 which takes 2 minutes and atomic recipe R_2 which takes 10 minutes, and we combine them to give $R_1 \oplus R_2$, then it would be desirable to determine that $R_1 \oplus R_2$ takes 12 minutes.

We also plan to further develop the concept of cost in determining better substitutions. We will investigate how substitution cost can be determined from ontological resources, and explore how to generalise the notion of cost so that it is calculated over equivalence classes of recipes. This would allow for better handling of substitution to generate the same recipe with an isomorphic graph. For example, if we have a cooking recipe for buttered toast with a consumable node $c1$ labelled with toast, and consumable node $c2$ labelled with butter, and we use type substitutions to have consumable node $c1$ labelled with butter, and consumable node $c2$ labelled with toast, then we may want the cost to be zero, since there is no material change to the recipe.

We will also investigate other possible implementations of the graphical formalism in the context of specific domains. In particular, we will explore the use of *RDF* and *SHACL* for the representation and validation of recipes, as this could enable easier integration of information from external resources. Examples of external resources from the realm of cooking include *FoodOn* and *FoodKG* (as discussed in more detail in Section 12). We will explore further how our implementations can incorporate techniques and frameworks for commonsense reasoning (see e.g. (Davis 2017)), since many of the domains mentioned here are ripe for utilising and developing this line of research.

Acknowledgments

This research is funded by the Leverhulme Trust via the *Repurposing of Resources: from Everyday Problem Solving through to Crisis Management* project (2022-2025). Fabio Aurelio D'Asaro acknowledges support from the project *Ethical Design for AI*, part of the Spoke 6 of the PNRR project FAIR (PE00000013, CUPH97G22000210007), funded by the European Union-NextGenerationEU and the Italian Ministry of University and Research (MUR).

References

A. Artikis, M. J. Sergot, and G. Paliouras. 2010. "A logic programming approach to activity recognition." In: *Proceedings of the 2nd ACM international workshop on Events in multimedia, EIMM 2010, Firenze, Italy, October 25 - 29, 2010*. Ed. by A. Scherp, R. C. Jain, M. S. Kankanhalli, and V. Mezaris. ACM, 3–8. doi:10.1145/1877937.1877941.

B. Banihashemi, G. D. Giacomo, and Y. Lespérance. 2017. "Abstraction in Situation Calculus Action Theories." In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. Ed. by S. Singh and S. Markovitch. AAAI Press, 1048–1055. doi:10.1609/AAAI.V31I1.10693.

R. Bergmann, G. Müller, and D. Wittkowsky. 2013. "Workflow Clustering Using Semantic Similarity Measures." In: *Proceedings of KI'13* (Lecture Notes in Computer Science). Vol. 8077. Springer, 13–24. doi:10.1007/978-3-642-40942-4_2.

A. Bikakis, L. Dickens, A. Hunter, and R. Miller. 2021. "Repurposing of Resources: from Everyday Problem Solving through to Crisis Management." *CoRR*, abs/2109.08425.

A. Blansché, J. Cojan, V. Dufour-Lussier, J. Lieber, P. Molli, E. Nauer, H. Skaf-Molli, and Y. Toussaint. July 2010. "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, (July 2010). https://inria.hal.science/inria-00526663.

M. Chang, L. V. Guillain, H. Jung, V. M. Hare, J. Kim, and M. Agrawala. 2018. "RecipeScape: An Interactive Tool for Analyzing Cooking Instructions at Scale." In: *Proceedings of CHI'18*. Association for Computing Machinery, 1–12. ISBN: 9781450356206.

E. Davis. 2017. "Logical formalizations of commonsense reasoning: A survey." *Journal of Artificial Intelligence Research*, 59, 651–723.

J. DeMiguel, L. Plaza, and B. Díaz-Agudo. 2008. "ColibriCook: A CBR System for Ontology-Based Recipe Retrieval and Adaptation." In: *ECCBR 2008, The 9th European Conference on Case-Based Reasoning, Trier, Germany, September 1-4, 2008, Workshop Proceedings*. Ed. by M. Schaaf, 199–208.

A. Diallo, A. Bikakis, L. Dickens, A. Hunter, and R. Miller. 2024. "PizzaCommonSense: A Dataset for Commonsense Reasoning about Intermediate Steps in Cooking Recipes." In: *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*. Ed. by Y. Al-Onaizan, M. Bansal, and Y. Chen. Association for Computational Linguistics, 12482–12496. https://aclanthology.org/2024.findings-emnlp.728.

D. Dooley et al.. 2022. "Food Process Ontology Requirements." *Semantic Web*, Preprint, 1–32.
D. M. Dooley, E. J. Griffiths, G. P. S. Gosal, P. L. Buttigieg, R. Hoehndorf, M. Lange, L. M. Schriml, F. S. L. Brinkman, and W. W. L. Hsiao. 2018. "FoodOn: a harmonized food ontology to increase global food traceability, quality control and data integration." *NPJ Science of Food*, 2.
V. Dufour-Lussier, F. Le Ber, J. Lieber, T. Meilender, and E. Nauer. Aug. 2012. "Semi-automatic annotation process for procedural texts: An application on cooking recipes." In: *Cooking with Computers workshop - ECAI 2012*. Ed. by E. N. Amélie Cordier. Montpellier, France, (Aug. 2012). https://inria.hal.science/hal-00735262.
F. Friedler, K. Tarján, Y. Huang, and L. Fan. 1992. "Graph-theoretic approach to process synthesis: axioms and theorems." *Chemical Engineering Science*, 47, 8, 1973–1988. doi:https://doi.org/10.1016/0009-2509(92)80315-4.
F. Friedler, K. B. Aviso, B. Bertok, D. C. Foo, and R. R. Tan. 2019. "Prospects and challenges for chemical process synthesis with P-graph." *Current Opinion in Chemical Engineering*, 26, 58–64. doi:https://doi.org/10.1016/j.coche.2019.08.007.
E. Gaillard, J. Lieber, and E. Nauer. 2017. "Adaptation of TAAABLE to the CCC'2017 Mixology and Salad Challenges, Adaptation of the Cocktail Names." In: *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* (CEUR Workshop Proceedings). Ed. by A. A. Sánchez-Ruiz and A. Kofod-Petersen. Vol. 2028. CEUR-WS.org, 253–268. https://ceur-ws.org/Vol-2028/paper29.pdf.
M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. 2019. "Multi-shot ASP solving with clingo." *Theory and Practice of Logic Programming*, 19, 1, 27–82.
M. Ghallab, D. Nau, and P. Traverso. 2016. *Automated Planning and Acting*. Cambridge University Press.
F. Giunchiglia and T. Walsh. 1992. "A theory of abstraction." *Artificial Intelligence*, 57, 2, 323–389. doi:https://doi.org/10.1016/0004-3702(92)90021-O.
S. Haussmann, O. Seneviratne, Y. Chen, Y. Ne'eman, J. Codella, C.-H. Chen, D. L. McGuinness, and M. J. Zaki. 2019. "FoodKG: A Semantics-Driven Knowledge Graph for Food Recommendation." In: *Proceedings of ISWC'19*. Springer-Verlag, 146–162. ISBN: 978-3-030-30795-0. doi:10.1007/978-3-030-30796-7_10.
P. J. Herrera, P. Iglesias, D. Romero, I. Rubio, and B. Díaz-Agudo. 2008. "JaDaCook: Java Application Developed and Cooked Over Ontological Knowledge." In: *ECCBR 2008, The 9th European Conference on Case-Based Reasoning, Trier, Germany, September 1-4, 2008, Workshop Proceedings*. Ed. by M. Schaaf, 209–218.
C. Hoare. 1985. *Communicating Sequential Processes*. Prentice Hall.
A. Ławrynowicz, A. Wróblewska, W. T. Adrian, B. Kulczyński, and A. Gramza-Michałowska. 2022. "Food Recipe Ingredient Substitution Ontology Design Pattern." *Sensors*, 22, 3, 1095. doi:10.3390/s22031095.
T. Mikolov, K. Chen, G. Corrado, and J. Dean. 2013. "Efficient estimation of word representations in vector space." *CoRR*, 1301.3781.
R. Milner. 1989. *Communication and Concurrency*. Prentice Hall.
R. Milner, J. Parrow, and D. Walker. 1992. "Efficient estimation of word representations in vector space." *Information and Computation*, 100, 1–40.
S. Mori, H. Maeta, Y. Yamakata, and T. Sasada. 2014. "Flow Graph Corpus from Recipe Texts." In: *Proc. Ninth Int. Conf. Lang. Resour. Eval.* 2370–2377.
S. Mota and B. Díaz-Agudo. Aug. 2012. "ACook: Recipe adaptation using ontologies, case-based reasoning systems and knowledge discovery." In: *Cooking with Computers workshop - ECAI 2012*. Ed. by E. N. Amélie Cordier. Montpellier, France, (Aug. 2012). https://projet.liris.cnrs.fr/cwc/cwc2012/cwc2012_submission_4.pdf.
G. Müller and R. Bergmann. 2015. "CookingCAKE: A Framework for the Adaptation of Cooking Recipes Represented as Workflows." In: *Workshop Proceedings of ICCBR'15* (CEUR Workshop Proceedings). Vol. 1520. CEUR-WS.org, 221–232. http://ceur-ws.org/Vol-1520/paper23.pdf.
R. Newo, K. Bach, A. Hanft, and K.-D. Althoff. June 2010. "On-Demand Recipe Processing based on CBR." In: *ICCBR-2010 Workshop Proceedings: Computer Cooking Contest Workshop*. Ed. by C. Marling. Alessandria, (June 2010), 209–218.
A. Olteteanu and Z. Falomir. 2016. "Object replacement and object composition in a creative cognitive system. Towards a computational solver of the Alternative Uses Test." *Cognitive Systems Research*, 39, 15–32.
V. Pallagani, P. Ramamurthy, V. Khandelwal, R. Venkataramanan, K. Lakkaraju, S. N. Aakur, and B. Srivastava. 2022. "A Rich Recipe Representation as Plan to Support Expressive Multi Modal Queries on Recipe Content and Preparation Process." *CoRR*, abs/2203.17109. arXiv: 2203.17109. doi:10.48550/arXiv.2203.17109.
L.-M. Pan, J. Chen, J. Wu, S. Liu, C.-W. Ngo, M.-Y. Kan, Y. Jiang, and T.-S. Chua. 2020. "Multi-Modal Cooking Workflow Construction for Food Recipes." In: *Proceedings of ACM Multimedia'20*, 1132–1141. ISBN: 9781450379885.
C. Pellegrini, E. Özsoy, M. Wintergerst, and G. Groh. 2021. "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, 67–77. doi:10.5220/0010202000670077.
J. Pennington, R. Socher, and C. Manning. 2014. "GloVe: Global vectors for word representation." In: *Proceedings of EMNLP'14*. Association for Computational Linguistics, 1532–1543.

C. A. Petri. 2005. "Introduction to general net theory." In: *Net Theory and Applications: Proceedings of the Advanced Course on General Net Theory of Processes and Systems Hamburg, October 8–19, 1979*. Springer, 1–19.

S. Quintana-Amate, P. Bermell-Garcia, and A. Tiwari. Aug. 2015. "Transforming expertise into Knowledge-Based Engineering tools." *Knowledge-Based Systems*, 84, C, (Aug. 2015), 89–97. doi:10.1016/j.knosys.2015.04.002.

P. Schumacher, M. Minor, K. Walter, and R. Bergmann. 2012. "Extraction of procedural knowledge from the web: a comparison of two workflow extraction approaches." In: *Proceedings of WWW'12*. ACM, 739–747. doi:10.1145/2187980.2188194.

Q. Shao, P. Sun, and Y. Chen. 2009. "WISE: A Workflow Information Search Engine." In: *Proceedings of ICDE'09*. IEEE Computer Society, 1491–1494. doi:10.1109/ICDE.2009.89.

Y. Shidochi, T. Takahashi, I. Ide, and H. Murase. 2009. "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, 9–14. ISBN: 9781605587639. doi:10.1145/1630995.1630998.

S. S. Shirai, O. Seneviratne, M. E. Gordon, C.-H. Chen, and D. L. McGuinness. 2021. "Identifying Ingredient Substitutions Using a Knowledge Graph of Food." *Frontiers in Artificial Intelligence*, 3, 621766. doi:10.3389/frai.2020.621766.

C. Steen and J. Newman. 2010. *The Complete Guide to Vegan Food Substitutions*. Fair Winds Press.

J. Stjepandić, W. J. C. Verhagen, H. Liese, and P. Bermell-Garcia. 2015. "Knowledge-Based Engineering." In: *Concurrent Engineering in the 21st Century: Foundations, Developments and Challenges*. Ed. by J. Stjepandić, N. Wognum, and W. J.C. Verhagen. Springer International Publishing, Cham, 255–286. ISBN: 978-3-319-13776-6. doi:10.1007/978-3-319-13776-6_10.

C.-Y. Teng, Y.-R. Lin, and L. A. Adamic. 2012. "Recipe Recommendation Using Ingredient Networks." In: *Proceedings of ACM Web Science Conference* (WebSci '12). Association for Computing Machinery, Evanston, Illinois, 298–307. ISBN: 9781450312288. doi:10.1145/2380718.2380757.

L. Wang, Q. Li, N. Li, G. Dong, and Y. Yang. 2008. "Substructure Similarity Measurement in Chinese Recipes." In: *Proceedings of WWW'08*. Association for Computing Machinery, 979–988. doi:10.1145/1367497.1367629.

Y. Yamakata, S. Mori, and J. Carroll. 2020. "English Recipe Flow Graph Corpus." In: *Proceedings of LREC'20*. European Language Resources Association, 5187–5194. https://aclanthology.org/2020.lrec-1.638.

Received 13 March 2025; accepted 27 May 2025