Parameterized Logical Theories

Fangzhen Lin
Department of Computer Science & HKUST-Xiaoi Joint Lab
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
flin@cs.ust.hk

Abstract

A theory in first-order logic is a set of sentences. A parameterized theory is a first-order theory with some of its predicates and functions identified as parameters, together with some import statements that call other parameterized theories. A KB is then a collection of these interconnected parameterized theories, similar to how a computer program is constructed as a set of functions in a modern programming language. In this paper, we provide a translational semantics for these parameterized theories in first-order logic using the situation calculus. We also discuss their potential uses in areas such as multi-context reasoning and logical formalization of computer programs.

Introduction

In AI, first-order logic and its variants have been widely used in KR. Central to this use is the construction of a KB, which is normally a logical theory, i.e. a set of sentences, for the domain of interest. When the domain of interest is complex and diverse, it is often desirable to organize its KB in some modular ways. A simple approach is to organize it as a hierarchy of smaller ones (e.g. (Hendrix 1975; Carnegie Group 1985; Lenat 1995; Lifschitz and Ren 2006)). The hierarchy can be in terms of refinement, abstraction, or some input/output relationship. Another approach is to treat the KB as a collection of contexts that are connected by some meta-axioms or rules to connect these contexts (e.g. (McCarthy 1993; Giunchiglia 1993; Giunchiglia and Serafini 1994; McCarthy and Buvač 1997; Brézillon 1999; Ghidini and Giunchiglia 2001)).

In this paper, we propose a notion of *parameterized* theories and consider a KB to be a collection of such theories. Informally, a parameterized theory is like a first-order theory but with some of the predicate and function symbols in it identified as parameters, and these parameters can be replaced when called from another theory. Like contexts, there are no built-in hierarchies in our framework. Unlike contexts, there are no separate meta-axioms or rules. As we will see, to connect some parameterized theories, one can define a new one that "calls" these theories.

In a sense, parameterized theories are like functions in computer programming languages: they can call and be called by others. In fact, our motivation for this work came from a recent work by Lin (2016) that translates a procedural program to a first-order theory. While trying to extend Lin's approach to more complex programs, including those involving functions with side effects, we encountered the need for embedding a logical theory inside another one, for the reason that if a function P in a program is formalized as a logical theory, then when another function Q calls P, there must be a way for the theory of Q to access the theory of P. We will outline how our notion of parameterized theories can be used to address this issue.

The rest of the paper is organized as follows. In the next section, we define the syntax of parameterized theories, and illustrate it with examples. We then propose a formal semantics that translates a set of parameterized theories to a set of first-order theories in the situation calculus, by viewing theory calls as actions. This is the main technical contribution of the paper. We next describe some possible applications, discuss related work, and then conclude the paper.

Parameterized Theories

We consider parameterized theories in first-order logic. Each parameterized theory can have its own language which is a set of function (including constant) and predicate symbols. We use lambda notation to denote anonymous functions and relations. A lambda expression is one of the form $\lambda x_1 ... \lambda x_k.E$, where E is either a term or a formula that has $x_1, .., x_k$ as the free variables. If E is a term, then it defines a k-ary function. If it is a formula then it defines a k-ary predicate. For example, if f is a binary function and g a unary function, then $\lambda x.f(x, g(x))$ is an anonymous unary function defined by the following equation:

$$\forall y.(\lambda x.f(x, g(x)))(y) = f(y, g(y)).$$

Similarly, if P is a binary predicate, then $\lambda x.\exists y P(x, y)$ is an anonymous unary predicate such that $(\lambda x.\exists y P(x, y))(z)$ is true iff $\exists y P(z, y)$ is true.

As we mentioned, a parameterized theory is essentially a first-order theory with some of the functions and predicates in it designated as parameters that can be replaced when it is called or imported into another theory.

Definition 1. *Given a language $\mathcal{L}$, a parameterized theory $C(p_1, ..., p_n)$ consists of a unique name C, a tuple*

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

$(p_1, .., p_n)$ *of* formal parameters*, which are distinct predicate and function symbols in* $\mathcal{L}$*, and a set of* axioms *which are either first-order sentences in* $\mathcal{L}$ *or* import statements *of the form*

$$C'(t_1, ..., t_m), \tag{1}$$

where

- C' *is a parameterized theory with some formal parameters* $q_1, ..., q_m$*;*
- $t_1, ..., t_m$*, called the actual parameters, are expressions (terms, functions, predicates, or lambda expressions) of the same types as* $q_1, ..., q_m$*, respectively.*

Notice that the q_i*'s are in the language of* C' *(the callee), and* t_i*'s in the language of* C*, the caller. One can read the import statement (1) as: import* C' *with each formal parameter* q_i *replaced by the actual one* t_i*,* $1 \leq i \leq m$*.*

In the following, we use the following format

$$\begin{aligned} &\text{Parameterized Theory } C(p_1, ..., p_n): \\ &\qquad\qquad \varphi_1, \\ &\qquad\qquad \vdots \\ &\qquad\qquad \varphi_m \end{aligned}$$

to define a parameterized theory whose name is C, whose parameters are $p_1, ..., p_n$, and whose axioms are $\varphi_1, ..., \varphi_m$. Unless otherwise stated, the language of this theory is taken to be the set of function and predicate symbols that are mentioned in the first-order axioms of the theory.

We also use the convention that free variables in a displayed formula are universally quantified from outside so for example,

$$P(x) \supset Q(x)$$

stands for $\forall x.P(x) \supset Q(x)$.

Example 1 The following is a parameterized theory for a syllogism, a generalization of modus ponens:

Parameterized Theory $Syllogism(a, prem, concl)$:

$$prem(a),$$
$$prem(x) \supset concl(x),$$

where a is a constant symbol and a parameter of the parameterized theory, and x is a variable, thus universally quantified from outside.

This is a base (primitive) parameterized theory that has no import statements. The following parameterized theory uses $Syllogism$. It has an import statement but no first-order axioms of its own:

Parameterized Theory $SocrSyll()$:

$$Syllogism(Socrates, man, mortal),$$

where $Socrates$ is a constant, and man and $mortal$ unary predicates. One can expect that parameterized theory $Syllogism$ will entail $concl(a)$, and as a result, $SocrSyll()$ will entail $mortal(Socrates)$.

A parameterized theory can call another one multiple times. For example, the following import statements can all be in the same parameterized theory:

$$Syllogism(Socrates, man, mortal),$$
$$Syllogism(Xenophon, man, mortal),$$
$$Syllogism(Tweety, bird, fly).$$

There is also no guard against inconsistent calls, like:

$$Syllogism(Tweety, bird, fly),$$
$$Syllogism(Tweety, penguin, \lambda x.\neg fly(x)).$$

Example 2 Parameterized Theories can call each other, like mutually recursive functions. The following is a trivial example on a language with only propositional symbols (0-ary predicates).

Parameterized Theory $Prop1(p)$: $Prop2(p)$.

Parameterized Theory $Prop2(p)$: $Prop1(\neg p)$.

It may appear that these two mutually recursive calls will result in a contradiction. But if one follows the call chain: $Prop2(p) \rightarrow Prop1(\neg p) \rightarrow Prop2(\neg p) \rightarrow$ ..., it is more like a loop that never terminates. We'll see that under our semantics, these two theories are the same as tautology.

Semantics

Given a parameterized theory $C(p_1, ..., p_n)$, a symbol in its language is called a *local* symbol if it is not a parameter.

Local symbols are supposed to be existentially quantified. For example, given

$$C(p) : p \equiv q,$$

the following two calls to it in another parameterized theory:

$$C'(r) : C(r), C(\neg r)$$

should not lead to a contradiction as the local symbol q is supposed to be existentially quantified. For example, $C'(r)$ in this case can be formalized as

$$\exists q(r \equiv q) \wedge \exists q(\neg r \equiv q), \tag{2}$$

which is a tautology.

In general, local symbols can be functions and predicates. Quantifying over them will lead to second-order axioms. Instead of existential quantification, we can introduce a fresh new name for each local symbol every time a parameterized theory is called. For example, $C'(r)$ can also be formalized as

$$(r \equiv q_1) \wedge (\neg r \equiv q_2). \tag{3}$$

Notice that (2) and (3), while not equivalent in general, are equivalent if restricted to the language of C': forgetting about q_1 and q_2 in (3) (Lin and Reiter 1994) also yields tautology.

In this paper we adopt the renaming approach, and propose a systematic way of renaming parameters and local symbols by introducing special *situation* objects to bookmark parameterized theory calls. The idea is similar to how function calls are implemented in programming languages

with so called activation records. Of course, we do not use these data structures. Our semantics is entirely declarative.

We begin by adding a new situation sort $\mathcal{S}$ to the language. Variables of this sort will be named using s, like s, s', s_1 etc. Unary functions from $\mathcal{S}$ to $\mathcal{S}$ will be named using μ, like μ, μ', μ_1 etc. These unary functions are the ones that will be used to bookmark parameterized theory calls, as we will see shortly. For those familiar with the situation calculus (McCarthy 1968; Reiter 2001; Lin 2007), these unary functions are like actions. We assume a special constant S_0, the dummy situation from which new situations are created using unary functions μ_1, μ_2, etc.

Given a symbol $p(\vec{x})$ in the language of a parameterized theory, we extend it with a sort $\mathcal{S}$ argument. Informally $p(\vec{x}, s)$ denotes the value of $p(\vec{x})$ inside the parameterized theory that is called in situation s. This way of extending a predicate or function with a new situation argument is standard in the situation calculus.

Our proposed semantics maps a parameterized theory C to two sets of axioms: one for reasoning within the parameterized theory and the other for reasoning about the parameterized theory when it is called. We call the first set *the primary theory* of C and denote it by $PT(C)$. We call the the latter *the situational theory* of C and denote it by $ST(C)$. We now describe how to construct these two sets.

To simplify our notation, without loss of generality, we assume that there is no overlap between the languages of any two different parameterized theories. This can be achieved by symbol renaming using a uniform scheme like adding a unique name as the prefix to every symbol in a parameterized theory.

Intuitively, the primary theory of C is just the normal first-order axioms in it, plus when there is an import statement in it, the axioms that pass the actual parameters to the called theory by creating a unique calling context:

Definition 2 (Primary Theory of a Parameterized Theory)**.** *Given a parameterized theory C, its primary theory $PT(C)$ consists of the first-order axioms in C and for each import statement $C'(t_1, ..., t_n)$ in it with the prototype $C'(p_1, ..., p_n)$, the following axioms:*

$$\forall \vec{x}.t_i(\vec{x}) \doteq p_i(\vec{x}, \mu(S_0)), 1 \leq i \leq n, \tag{4}$$

where $\mu : \mathcal{S} \rightarrow \mathcal{S}$ is a new unary function introduced for this call, and $t_i \doteq p_i$ is $t_i = p_i$ if p_i is a function, and $t_1 \equiv p_i$ if p_i is a predicate.

The situational theory of C, written $ST(C)$, is obtained from its primary theory by extending all symbols in C with a situation argument s and universally quantifying over it from the outside.

Definition 3 (Situational Theory of a Parameterized Theory)**.** *Given a parameterized theory C, $ST(C)$ is defined as follows:*

$$ST(C) = \{\forall s.\varphi[s] \mid \varphi \in PT(C)\}, \tag{5}$$

where $\varphi[s]$ is obtained from φ by the following substitutions:

- *Replace S_0 by s;*
- *for each symbol p, replace its every occurrence $p(\vec{t})$ by $p(\vec{t}, s)$, where $\vec{t}$ does not contain sort $\mathcal{S}$.*

Intuitively, $\varphi[s]$ means φ holds in s, and is applied only to a formula φ that does not mention any other situation term except S_0. For example, $(f(a, g(b, S_0)) = f(c, x))[s]$ is

$$f(a(s), g(b(s), s), s) = f(c(s), x, s),$$

assuming a, b, and c are constants, g and f functions, and x a variable.

Given a collection of parameterized theories, its semantics is then the union of the translated theories of all the parameterized theories in the collection. However, when reasoning about a specific parameterized theory, there is no need to consider all other theories, only those that may potentially be called by it.

Given a parameterized theory C, we say that C *depends on* C' if C calls C' or C calls a parameterized theory C'' such that C'' depends on C', i.e. the transitive closure of "C calls C'". Given a C, let

$$Dep(C) = \{C' \mid C \text{ depends on } C'\}.$$

Notice that it is possible for $C \in Dep(C)$ for some C.

Definition 4. *Let C be a parameterized theory. The* translated theory *of C, written $\Sigma(C)$, is defined as the following first-order theory:*

$$\Sigma(C) = PT(C) \cup ST(C) \cup \bigcup_{C' \in Dep(C)} (PT(C') \cup ST(C'))$$

In the following we consider the semantics of C to be $\Sigma(C)$. For example, we say that C is a consistent parameterized theory if $\Sigma(C)$ is a consistent first order theory. We say that C entails a sentence φ, written $C \models \varphi$ by overloading the classic entailment symbol "$\models$", if $\Sigma(C) \models \varphi$ in first-order logic.

The following theorem says that for reasoning about sentences in the langauge of C, we only need $PT(C)$ and $ST(C')$ from those C' that C depends on.

Theorem 1. *Let C be a parameterized theory and φ a sentence in C. We have that*

$$\Sigma(C) \models \varphi \textit{ iff } \mathcal{T}(C) \models \varphi,$$

where

$$\mathcal{T}(C) = PT(C) \cup \bigcup_{C' \in Dep(C)} ST(C')$$

Proof. The "if" direction is trivial. We show the "only if" part. Suppose $\mathcal{T}(C) \models \varphi$ is not true, and M is a model that satisfies $\mathcal{T}(C)$ but not φ. From M, we construct a model M' that satisfies $\Sigma(C)$ but not φ. Notice that a symbol p in a language has two versions: its original one and the extended one $p(s)$. M' has the same domains as M. For each p, there are two cases:

1. p is in C. M' interprets p the same as M does. If $C \in Dep(C)$, then M' also interprets $p(s)$ the same as M does. Otherwise, M' interprets $p(s)$ the same as p by ignoring the last argument s.
2. p is in a different parameterized theory C'. M' interprets $p(s)$ the same as M does, and p the same as $p(S_0)$.

Under this construction, one can verify that M' is a model of $\Sigma(C)$. It does not satisfy φ as M does not. Notice how the unique role that S_0 plays in the construction of M'. $\square$

The following result says that when there is no recursive calls, our semantics works like macro or inline expansions. For simplicity, it's given for the basic case of a theory calling another base theory. Again, without loss of generality, we assume the two theories have no common predicates and functions. Our proof makes use of forgetting, and is omitted here for lack of space.

Theorem 2. *Let $C(\vec{p})$ and $C'(\vec{q})$ be two parameterized theories that do not have overlap in their languages. Suppose C has a single import statement $C'(\vec{t})$ and C' has no import statement. For any formula φ in the language of C, $C \models \varphi$ iff $Expand(C) \models \varphi$, where $Expand(C)$ is the conjunction of the first-order axioms in C and C', with each q_i in $\vec{q}$ replaced by its corresponding t_i in $\vec{t}$ in the axioms of C'.*

Examples

We now illustrate this translational semantics with some examples from the previous sections.

Consider first the following two parameterized theories at the beginning of last section:

$$\begin{aligned} C(p): &\quad p \equiv q, \\ C'(r): &\quad C(r), C(\neg r). \end{aligned}$$

The parameterized theory C has no import statement, so its primary theory, $PT(C)$, is just $p \equiv q$. Its situational theory, $ST(C)$, is $\forall s.p(s) \equiv q(s)$.

On the other hand, $PT(C')$, is

$$\begin{aligned} r &\equiv p(\mu_1(S_0)), \\ \neg r &\equiv p(\mu_2(S_0)), \end{aligned}$$

where μ_1 and μ_2 are the two unary functions corresponding to the two calls, respectively. $ST(C')$ is:

$$\begin{aligned} r(s) &\equiv p(\mu_1(s)), \\ \neg r(s) &\equiv p(\mu_2(s)). \end{aligned}$$

Clearly, the union of all these theories is consistent. Furthermore, forgetting $p(s)$, $q(s)$ and $r(s)$ in them will yield $p \equiv q$. This means that the parameterized theory C' does not say anything informative.

Syllogism

Consider the set of parameterized theories $Syllogism$ and $SocrSyll$. For the first one, its primary theory is just the set of axioms in it as there are no import statements:

$$\begin{aligned} &prem(a), \\ &prem(x) \supset concl(x). \end{aligned}$$

Its situational theory is obtained by adding a situation argument to all the axioms:

$$\begin{aligned} &prem(a(s), s), \\ &prem(x, s) \supset concl(x, s). \end{aligned}$$

This parameterized theory entails $concl(a)$. It is easy to see that any parameterized theory that calls $Syllogism(t, p_1, p_2)$ will entail $p_2(t)$. For example, $SocrSyll$ calls $Syllogism(Socrates, man, mortal)$, thus entails $mortal(Socrates)$. This can be independently verified as follows. For $SocrSyll$, its primary theory is:

$$\begin{aligned} &Socrates = a(\mu(S_0)), \\ &man(x) \equiv prem(x, \mu(S_0)), \\ &mortal(x) \equiv concl(x, \mu(S_0)), \end{aligned}$$

where μ is the unary function introduced for the call $Syllogism(Socrates, man, mortal)$. Its situational theory is:

$$\begin{aligned} &Socrates(s) = a(\mu(s)), \\ &man(x, s) \equiv prem(x, \mu(s)), \\ &mortal(x, s) \equiv concl(x, \mu(s)). \end{aligned}$$

The conclusion $mortal(Socrates)$ of $SocrSyll$ can now be verified as it is a logical consequence of $ST(Syllogism) \cup PT(SocrSyll)$, the primary theory of $SocrSyll$ and the situational theory of $Syllogism$ as the former calls the latter.

The following is an example of multiple calls that lead to a contradiction:

$$\begin{aligned} &Syllogism(Tweety, bird, fly), \\ &Syllogism(Tweety, bird, \lambda x.\neg fly(x)). \end{aligned}$$

Let μ_1 and μ_2 be the unary functions for these two calls, respectively. These two calls generate the following axioms:

$$\begin{aligned} &Tweety = a(\mu_1(S_0)), \quad bird(x) \equiv prem(x, \mu_1(S_0)), \\ &fly(x) \equiv concl(x, \mu_1(S_0)), \\ &Tweety = a(\mu_2(S_0)), \quad bird(x) \equiv prem(x, \mu_2(S_0)), \\ &\neg fly(x) \equiv concl(x, \mu_2(S_0)). \end{aligned}$$

They entail both $fly(Tweety)$ and $\neg fly(Tweety)$, a contradiction caused by the two inconsistent calls.

Mutual Recursion

Consider now the mutually recursive theories $Prop1(p)$ and $Prop2(p)$ in the last section. First of all, we rename the parameter p to make it unique to each theory.

$$\begin{aligned} &Prop1(p_1) : Prop2(p_1), \\ &Prop2(p_2) : Prop1(\neg p_2). \end{aligned}$$

The primary theory for $Prop1(p_1)$, $PT(Prop1)$, is

$$p_1 \equiv p_2(\mu_1(S_0)),$$

and $ST(Prop1)$ is

$$p_1(s) \equiv p_2(\mu_1(s)),$$

where μ_1 is the unary function corresponding to the call $Prop2(p_1)$. Similarly $PT(Prop2)$ is $\neg p_2 \equiv p_1(\mu_2(S_0))$, and $ST(Prop2)$ is $\forall s.\neg p_2(s) \equiv p_1(\mu_2(s))$, where μ_2 is the unary function corresponding to the call $Prop1(\neg q_2)$. Thus

$$\begin{aligned} p_1 &\equiv p_2(\mu_1(S_0)) \equiv \neg p_1(\mu_2(\mu_1(S_0))) \\ &\equiv \neg p_2(\mu_1(\mu_2(\mu_1(S_0)))) \\ &\equiv p_1(\mu_2(\mu_1(\mu_2(\mu_1(S_0))))) \equiv \cdots \end{aligned}$$

which is a consistent infinite equivalence chain. In fact, forgetting the two unary predicates $p_1(s)$ and $p_2(s)$ will result in a tautology, meaning the set of the above sentences has no information about the truth values of p_1 and p_2.

Recursively defined functions and relations are most useful on inductively defined domains. For example, we can define the concepts of odd and even numbers recursively using the following two parameterized theories:

Parameterized Theory $Odd(odd)$:

$$Even(tmp_1),$$
$$odd(n) \equiv n \neq 0 \wedge tmp_1(n-1)$$

Parameterized Theory $Even(even)$:

$$Odd(tmp_2),$$
$$even(n) \equiv n = 0 \vee (n > 0 \wedge tmp_2(n-1)),$$

where we assume a pre-defined natural number domain with built-in functions like "-". We will have more to say about sorts and predefined (interpreted) symbols later. As an example, under our semantics, assuming the standard interpretation to these predefined symbols, and letting μ_1 be the unary situation function for the call $Even(tmp_1)$ in Odd, and μ_2 the unary situation function for the call $Odd(tmp_2)$ in $Even$, $odd(3)$ can be deduced as follows:

$$\begin{aligned} odd(3) &\equiv temp_1(2) \equiv even(2, \mu_1(S_0)) \\ &\equiv temp_2(1, \mu_1(S_0)) \equiv odd(1, \mu_2(\mu_1(S_0))) \\ &\equiv temp_1(0, \mu_2(\mu_1(S_0))) \\ &\equiv even(0, \mu_1(\mu_2(\mu_1(S_0)))) \equiv true \end{aligned}$$

Extensions

Conditional Imports

We have defined an import statement as a wholesome call to a parameterized theory. It can be generalized to include conditional ones such as:

$$penguin(Tweety) \supset Syllogism(Tweety, bird, \lambda x.\neg fly(x)),$$
$$parrot(Tweety) \supset Syllogism(Tweety, bird, fly).$$

Conditional imports will be needed for formalizing the semantics of computer programs as discussed in the next section. Our semantics can be extended to these conditional calls straightforwardly. For example, the above two example calls will yield the following axioms in the primary theory of the parameterized theory that contains them:

$$\begin{aligned} penguin(Tweety) \supset Tweety = a(\mu_1(S_0)) \wedge \\ \forall x(bird(x) \equiv prem(x, \mu_1(S_0))) \wedge \\ \forall x(\neg fly(x) \equiv concl(x, \mu_1(S_0))), \\ parrot(Tweety) \supset Tweety = a(\mu_2(S_0)) \wedge \\ \forall x(bird(x) \equiv prem(x, \mu_2(S_0))) \wedge \\ \forall x(fly(x) \equiv concl(x, \mu_2(S_0))). \end{aligned}$$

Their corresponding axioms in the situational theory can then be generated as usual.

Import statements, however, should not be used like ordinary predicates. For example, putting a negation in front of them like $\neg Syllogism(Tweety, bird, fly)$ does not seem to make any sense.

Sorts and Interpreted Symbols

So far we have considered first-order languages without multiple sorts and interpreted or pre-defined symbols. However, these additional features can be accommodated without any problem. A first-order many-sorted language (with interpreted symbols) consists of the following components:

- A finite set of sorts, some of them marked as interpreted, meaning they will be interpreted by some pre-defined sets.
- A finite set of function and predicate symbols. Each of them will come with a type: a function symbol f will have a type of the form $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$, and a predicate of the form $\tau_1 \times \cdots \times \tau_n$, where the τ's are sorts. Some of the symbols can be designated as interpreted, meaning their interpretations will be fixed.

Lambda expressions are the same as before but will have types as well. Parameterized theories and import statements are also defined as before, but with the following restriction:

- Interpreted symbols cannot be formal parameters.
- In an import statement $C(t_1, ..., t_n)$ that calls $C(p_1, ..., p_n)$, each actual parameter t_i must be of the same type as its corresponding formal parameter p_i.

The semantics for these parameterized theories is the same as before.

We have seen earlier that the two mutually recursive theories Odd and $Even$ use interpreted sorts and symbols. We give another example here. It uses one interpreted sort $\mathcal{N}$ for natural numbers, and one (un-interpreted) sort $\mathcal{O}$ for domain objects. We also use $+$ and 0 as interpreted symbols of their usual meaning in $\mathcal{N}$.

The following base parameterized theory defines the transitive closure of a binary relation.

Parameterized Theory $TransClos(base, tr)$:

$$tr(x, y) \equiv \exists n.tr_0(x, y, n),$$
$$tr_0(x, y, 0) \equiv base(x, y),$$
$$tr_0(x, y, n+1) \equiv \exists z.base(x, z) \wedge tr_0(z, y, n),$$

where x and y are variables of sort $\mathcal{O}$, and n a variable of sort $\mathcal{N}$. They are all universally quantified over their respective sorts. Notice that from the axioms, one can see that $base$ is a binary predicate of sort $\mathcal{O} \times \mathcal{O}$. The types of other predicates can be similarly determined.

Other parameterized theories can use $TransClos$ to define transitive closures of some specific binary relations. The following parameterized theory uses $TransClos$ to define when a node is reachable from a starting node.

Parameterized Theory $Reachable(start, edge, reachable)$:

$$TransClos(edge, path),$$
$$reachable(x) \equiv path(start, x),$$

where x is a variable of sort $\mathcal{O}$, and $start$ a constant symbol of sort $\mathcal{O}$.

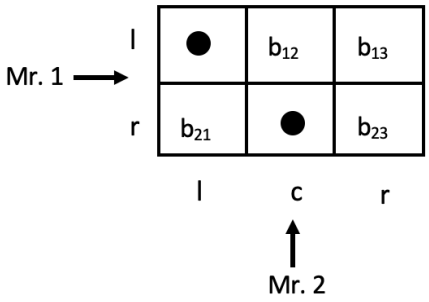

Figure 1: A magic box

Some Applications

Multicontext Reasoning

A collection of parameterized theories is a set of logical theories interconnected by import calls. The connection with multicontext reasoning is apparent, at least informally.

Consider the "magic box" shown in Figure 1, an example adapted from (Ghidini and Giunchiglia 2001). There are two agents, Mr. 1 and Mr. 2, who look at the box from two perpendicular sides. Mr. 1, looking into the box from the side, can see whether the left and the right rows of the box have balls. Mr. 2, looking from the bottom, can tell whether the left, center, and the right column of the box have balls. Their knowledge alone cannot determine exactly which cell is occupied. But combining their knowledge, sometimes one can. For example, if Mr. 1 sees a ball on the left side but no ball on the right side, and Mr. 2 sees a ball on the left side, then we can conclude that the top left corner is occupied by a ball.

This is a classic example in multi-context systems where contexts are used to represent agents' views, and bridge rules for linking contexts to reason about the state of the box.

In our framework, we can specify an agent's point of view by a parameterized theory, and use a "main" parameterized theory to aggregate the views by calling the agents' parameterized theories. For the magic box problem, the main parameterized theory can be written as follows:

$Main()$:

$$
\begin{aligned}
&Mr1(l_1, r_1),\\
&Mr2(l_2, c_2, r_2),\\
&l_1 \equiv b_{11} \vee b_{12} \vee b_{13},\\
&r_1 \equiv b_{21} \vee b_{22} \vee b_{23},\\
&l_2 \equiv b_{11} \vee b_{21},\\
&c_2 \equiv b_{12} \vee b_{22},\\
&r_2 \equiv b_{13} \vee b_{23},
\end{aligned}
$$

where $Mr1()$ and $Mr2()$ are the parameterized theories for the two agents, respectively. For example, if Mr. 1 sees that the left row is occupied but not the right row, while Mr. 2 sees that the left and the right columns are occupied but not the center one, then we have

$Mr1(l, r)$: $l \wedge \neg r$.

$Mr2(l, c, r)$: $l \wedge \neg c \wedge r$.

With these three parameterized theories, it is easy to see that the main parameterized theory entails b_{11} (the top left corner is occupied by a ball).

Program Semantics

Our initial motivation for proposing this framework of logical theories is to provide a logical semantics to programming languages. Lin (2016) proposed a translation from programs in a core programming language to first-order theories. Later they demonstrated the practical effectiveness of this approach in automated program verification (Rajkhowa and Lin 2018, 2019), with SOTA performance in a recent program verification competition.

However, one limitation with their system is that it has difficulty dealing with functions with side effects. For example, consider a function called `AddOne(A)` that takes an array `A` of integers as input and adds 1 to every element in it. If arrays are implemented as reference variables like in C, then this function has no effect on the input array variable itself. Instead, it achieves its goal through its "side" effects of changing the values of a consecutive sequence of locations pointed to by the array variable.

One way to axiomatize the effects of this function is to introduce an array sort and use the following two functions, $value(a, i)$ that denotes of the value of the ith element in a one-dimensional array a, and $len(a)$ that denotes the length of the array a. The effect of `AddOne(A)` can then be formalized by the following axioms:

$$
\begin{aligned}
&A' = A,\\
&len'(a) = len(a),\\
&1 \leq i \leq len(A) \supset value'(A, i) = value(A, i) + 1,\\
&1 \leq i \leq len(a) \wedge a \neq A \supset value'(a, i) = value(a, i),
\end{aligned}
$$

where a is an array variable, and i an integer variable, both are universally quantified. Primed symbols like A' denote the outputs of the function. So A and A' are the values (locations) of the input array `A` before and after `AddOne(A)` is executed, respectively. $A' = A$ means that the function has no effect on the value of the input array as a reference variable. Similarly, $value'$ is $value$ after the function returns, so $value'(a, i)$ denotes the value of the ith element of the array object a after the function returns.

Now, whenever `AddOne(A)` is called in a program, we will need to insert the above set of axioms when translating the program to a first-order theory. One can imagine the challenge of systematically doing this, especially when a function is recursively defined. This is what motivated us to introduce the notion of parameterized theories in this paper, and also the reason why we propose a translational semantics for these parameterized theories. Each function in a program will now be formalized as a parameterized theory, and a function call will be translated to a corresponding import statement.

Notice that in a program, a function call can occurs in an expression such as `X=X+f(X)` or in a conditional statement like `if X==0 then f(X)`. In the first case, one can eliminate calls in an expression by using a new variable, like

`Y=f(X); X=X+Y;` (notice that while $f(X)$ may have side effects, it will not have side effect on a value variable like `X` that occurs in an arithmetic expression). In the latter case, we can extend our import statements to conditional ones of the form $\varphi \supset C(t_1, ..., t_2)$, where φ is a sentence in the language of the caller, as we discussed earlier.

Related Work

In a nutshell, a parameterized theory is a named theory with some of its predicates and functions marked as parameters so that it can be called by other theories. This is of course similar to how functions and procedures are defined and used in high level programming languages, except that there is no real computation done when a parameterized theory is called - it is just a way to import a set of axioms. In this sense, calling a parameterized theory is like importing a package or a module in computer programming, except that packages and modules in software engineering do not normally have parameters.

In logic, variables that range over predicates and functions are second-order. Thus a parameterized theory could be captured by a third-order predicate with second-order arguments. While we do not intend to use third-order logic for reasoning, we are currently exploring a third-order logic semantics and studing its relationship with our first-order semantics here.

There has been much work on various notions of modules and modular systems in KR. In fact, almost every KR formalism, monotonic or nonmonotonic, has seen some work on them. To our understanding, the most closely related work in KR is that on reasoning about contexts (McCarthy 1993; Giunchiglia 1993), especially those on multi-context systems (MCS) (e.g. (Giunchiglia 1993; Giunchiglia and Serafini 1994; Ghidini and Giunchiglia 2001; Brewka, Roelofsen, and Serafini 2007; Brewka et al. 2011, 2018)). MCS has been used for aggregating heterogeneous knowledge sources. The basic idea is that each knowledge source will have its own language and axioms, and they are integrated using so-called bridge rules. A classic example that illustrates the uses of the MCS is the magic box example that we discussed earlier. As we have seen, instead of bridge rules, we use a parameterized theory to import all knowledge sources and aggregate them by standard first-order axioms. It's an interesting open question if the two frameworks can be related in a precise way. Given that MCS has been extended to nonmonotonic logics (Brewka, Roelofsen, and Serafini 2007), a good understanding of the relationships between these two frameworks will help when we consider extending parameterized theories to nonmonotonic logics.

There are two key points that we can make about our approach. One is that different parameterized theories can assumed to have completely different languages. The other is that theory calls can be recursive, especially when there are pre-interpreted sorts for inductively defined objects. The first point simplifies many issues, like how two modules may interact through shared vocabularies, a common issue for work on modules in many formalisms (e.g. separability in DLs (Konev et al. 2009) and input/output interface in ASP (Janhunen et al. 2009)). The second point is what made this work non-trivial, especially given that we want a translational first-order semantics.

While work on module is often based on computational considerations, our main motivation is on *first-order* theory composition, especially in the context of logical formalization of programs: if a function is represented by a first-order theory T, and a program calls this function, then a first-order theory for the program can be "computed" from T in a modular way. If there is no cycle in the call dependency graph, then this can be handled by macro expansion or inline function. Interestingly, this is the approach taken in (Baral, Dzifcak, and Takahashi 2006) for ASP and in a recent work on programming with logical constraints using so-called knowledge units (Liu and Stoller 2020). However, being able to handle recursive calls is crucial for us. Again, the reason is that when we want to have a one to one correspondence between functions in a computer program and parameterized theories here, a set of recursively defined functions will cause cycles in the chains of theory calls. Notice that Lin (2016) also considered recursive functions. But those functions are assumed to return values without any side effects. In a sense, recursive theory calls are our answer to recursive functions with side effects.

Concluding Remarks

Motivated by the need to have callable theories in logical formalization of computer programs, we have proposed a framework of parameterized first-order theories. While reasoning about parameterized theories appears to require higher-order logics, we have been able to provide a first-order semantics by making use of the situation calculus. Our semantics is modular, declarative, and at the same time retains certain procedural information of how parameterized theories are "run". We have shown some possible applications of our framework in formal semantics of computer programs and in reasoning about contexts.

One future work is to implement this framework in program verification, following the work of Rajkhowa and Lin (2018; 2019). Their system was based on a translation (Lin 2016) from programs to first-order theories, and uses the off-the-shelf math system SymPy as the algebraic simplifier and SMT solver Z3 (de Moura and Bjørner 2008) as the theorem prover. Remarkably, despite its simplicity, their system managed to achieve SOTA performance in the tracks that it competed in at SV-COMP'2019[1]. In particular, it came first in the Arrays subcategory and Recursive subcategory of ReachSafety category of the competition. However, it uses inline expansion when there is a function call with array parameters. With this work, we can implement a more general system that is able to handle recursive functions with array parameters as well as other functions with side effects.

We also plan to use the proposed framework in applications other than program verification. We have mentioned multi-context systems. We hope to have more to report on this soon.

[1] See https://verifierintegerassignment.github.io

Acknowledgments

I want to thank Pritom Rajkhowa for many useful discussions that we had during the course of our work on program verification in first-order logic where the need for this work first arose. I also thank Annie Liu, Yisong Wang, and the anonymous reviewers for their many critical, constructive and helpful comments on earlier versions of this paper. Of course, any remaining problems and possible errors are solely the author's responsibility.

References

Baral, C.; Dzifcak, J.; and Takahashi, H. 2006. Macros, Macro Calls and Use of Ensembles in Modular Answer Set Programming. In *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, 376–390.

Brewka, G.; Eiter, T.; Fink, M.; and Weinzierl, A. 2011. Managed Multi-Context Systems. In Walsh, T., ed., *IJCAI 2011*, 786–791. IJCAI/AAAI. doi:10.5591/978-1-57735-516-8/IJCAI11-138.

Brewka, G.; Ellmauthaler, S.; Gonçalves, R.; Knorr, M.; Leite, J.; and Pührer, J. 2018. Reactive multi-context systems: Heterogeneous reasoning in dynamic environments. *Artif. Intell.* 256: 68–104. doi:10.1016/j.artint.2017.11.007.

Brewka, G.; Roelofsen, F.; and Serafini, L. 2007. Contextual Default Reasoning. In Veloso, M. M., ed., *IJCAI 2007*, 268–273.

Brézillon, P. 1999. Context in problem solving: a survey. *The Knowledge Engineering Review* 14(1): 47–80. doi:10.1017/S0269888999141018.

Carnegie Group. 1985. Knowledge Craft 3.0 reference manual. Volume 1 and 2. Technical report, Carnegie Group Inc, Pittsburgh, Pennsylvania, U.S.A.

de Moura, L.; and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In Ramakrishnan, C. R.; and Rehof, J., eds., *Tools and Algorithms for the Construction and Analysis of Systems*, 337–340. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. URL http://research.microsoft.com/en-us/um/redmond/projects/z3/.

Ghidini, C.; and Giunchiglia, F. 2001. Local Models Semantics, or contextual reasoning = locality + compatibility. *Artificial Intelligence* 127(2): 221 – 259. doi:https://doi.org/10.1016/S0004-3702(01)00064-9.

Giunchiglia, F. 1993. Contextual reasoning. *Epistemologia* 16: 341–364.

Giunchiglia, F.; and Serafini, L. 1994. Multilanguage hierarchical logics or: How we can do without modal logics. *Artificial Intelligence* 65(1): 29–70.

Hendrix, G. 1975. Expanding the utility of semantic networks through partitioning. In *Proc. of IJCAI'75*, 115–121.

Janhunen, T.; Oikarinen, E.; Tompits, H.; and Woltran, S. 2009. Modularity Aspects of Disjunctive Stable Models. *J. Artif. Intell. Res.* 35: 813–857.

Konev, B.; Lutz, C.; Walther, D.; and Wolter, F. 2009. *Formal Properties of Modularisation*, 25–66. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-01907-4. doi:10.1007/978-3-642-01907-4_3. URL https://doi.org/10.1007/978-3-642-01907-4_3.

Lenat, D. B. 1995. CYC: A Large-Scale Investment in Knowledge Infrastructure. *Commun. ACM* 38(11): 32–38. doi:10.1145/219717.219745.

Lifschitz, V.; and Ren, W. 2006. A Modular Action Description Language. In *Proceedings, AAAI-2006*, 853–859. AAAI Press.

Lin, F. 2007. Situation Calculus. In van Harmelen, F.; Lifschitz, V.; and Porter, B., eds., *Handbook of Knowledge Representation*. Elsevier.

Lin, F. 2016. A Formalization of Programs in First-Order Logic with a Discrete Linear Order. *Artificial Intelligence* 235: 1–25.

Lin, F.; and Reiter, R. 1994. Forget it! In Greiner, R.; and Subramanian, D., eds., *Working Notes of AAAI Fall Symposium on Relevance*, 154–159. The American Association for Artificial Intelligence, Menlo Park, CA.

Liu, Y. A.; and Stoller, S. D. 2020. Knowledge of Uncertain Worlds: Programming with Logical Constraints. In *Proceedings of the 2020 International Symposium on Logical Foundations of Computer Science*, volume 11972 of *LNCS*, 111–127. Deerfield Beach, Florida: Springer. doi:https://doi.org/10.1007/978-3-030-36755-8_8.

McCarthy, J. 1968. Situations, actions and causal laws. In Minsky, M., ed., *Semantic Information Processing*, 410–417. MIT Press, Cambridge, Mass.

McCarthy, J. 1993. Notes on formalizing context. In *Proc. of IJCAI'93*, 555–560.

McCarthy, J.; and Buvač, S. 1997. Formalizing context (expanded notes). In Aliseda, A.; van Glabbeek, R.; and Westerståhl, D., eds., *Computing Natural Language*, 13–50. CSLI Publications, Stanford.

Rajkhowa, P.; and Lin, F. 2018. Extending VIAP to Handle Array Programs. In *10th Working Conference on Verified Software: Theories, Tools, and Experiments, VSTTE 2018, Oxford, UK, July 18-19*. URL https://github.com/VerifierIntegerAssignment/sv-comp/blob/master/extending-viap-array.pdf.

Rajkhowa, P.; and Lin, F. 2019. VIAP 1.1 - (Competition Contribution). In Beyer, D.; Huisman, M.; Kordon, F.; and Steffen, B., eds., *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, 250–255. Springer. URL https://doi.org/10.1007/978-3-030-17502-3_23.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press.