A Well-typed Lightweight Situation Calculus*

Li Tan

Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA, USA 92507

`ltan003@cs.ucr.edu`

Situation calculus has been widely applied in Artificial Intelligence related fields. This formalism is considered as a dialect of logic programming language and mostly used in dynamic domain modeling. However, type systems are hardly deployed in situation calculus in the literature. To achieve a correct and sound typed program written in situation calculus, adding typing elements into the current situation calculus will be quite helpful. In this paper, we propose to add more typing mechanisms to the current version of situation calculus, especially for three basic elements in situation calculus: situations, actions and objects, and then perform rigid type checking for existing situation calculus programs to find out the well-typed and ill-typed ones. In this way, type correctness and soundness in situation calculus programs can be guaranteed by type checking based on our type system. This modified version of a lightweight situation calculus is proved to be a robust and well-typed system.

1 Introduction

Introduced by John McCarthy in 1963 [10], situation calculus has been widely applied in Artificial Intelligence related research areas and other fields. This formalism is considered as a dialect of logic programming language and mostly used in dynamic domain modeling. Based on First Order Logic (FOL) [1] and Basic Action Theory [9], situation calculus can be used for reasoning efficiently by virtue of dynamic elements, such as actions and fluents. Basic concepts of situation calculus are fundamentals of First Order Logic and Set Theory in Mathematical Logic, which greatly facilitate the process of action-based reasoning in situation calculus.

In order to make programs sound and correct in semantics, people have proposed type systems [12] to ensure such significant properties. A well-typed programming language is determined by two semantic properties: preservation and progress. The first property makes sure that types are invariant under the evaluation and typing rules. And the progress property says a well-typed program never gets stuck. Nevertheless, little attention has been put on equipping formal languages good at dynamic modeling and reasoning, like situation calculus, with strong typing mechanisms. Indeed, situation Calculus is a typed second-order formal language, but from the viewpoint of type checking, it is not enough to finish smoothly. For instance, in situation calculus, only typed quantifiers are introduced for basic variables, while as for other logical expressions consisting of variables and connectives, fluents and predicates, current situation calculus emphasizes little on how to type check whether they are well-typed, how to type them thoroughly. Therefore, equipping other elements in current version of situation calculus with types is greatly needed for a complete and robust programming language with its type system, which is definitely feasible according to our investigation.

In this paper, in addition to the handy available typed variables, we propose to add more typing mechanisms to three basic elements in situation calculus: situations, actions and objects, consider a

---

*This work is also offered to present at *the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP'11)*, Odense, Denmark, July 2011.

classical scenario for a piece of program based on the modified lightweight situation calculus, and then perform rigid type checking for the situation calculus program. If type errors are found, we would provide corresponding recommendation on how to correct the program into a well-typed one. Furthermore, to support our ideas in practice, we implement a type checker to semi-automatically finish the type checking work instead of working manually.

We organize our paper in the following way: section 2 introduces the related work on typing situation calculus and its variants; Section 3 presents the basic ideas on type systems and situation calculus in a straightforward way; Section 4 illustrates the primary ideas on how to type a lightweight core of the original situation calculus; Section 5 evaluates our typing mechanisms by type checking an existing piece of program in situation calculus and section 6 concludes this paper.

2   Related Work

Due to its powerful action-based reasoning ability, situation calculus is often chosen as the formalism to express other models and programming languages which are either too complex to understand and use, like Artificial Intelligence in games [5] and Planning Domain Definition Language (PDDL) [6], or a little powerless to represent an entire complicated systems of different types, like Action Description Language (ADL) [11]. In the literature employing situation calculus as a formal method to express the semantics in PDDL [3] and ADL [4], the authors have tried to introduce some typing mechanisms, which is only limited to add type element in syntax, and only applied to variables. Other significant terms, such as fluents and predicates, are still typeless. Moreover, in semantics and reasoning, typing mechanisms are hardly discussed in these papers, neither is type checking.

Yilan Gu et al. [7] proposed a modified version of the situation calculus built using a two-variable fragment of the first-order logic extended with counting quantifiers. By introducing several additional groups of axiom to capture taxonomic reasoning and using similar regression operator in Raymond Reiter's work [13], the projection and executability problems are proved decidable although an initial knowledge base is incomplete and open. While their system concerns primarily on semantics of the new components proposed but rarely talks about typing on them, our well-typed version of situation calculus mentions typing mechanisms together with a modified situation calculus version in an all around way.

There are also some attempts on modifying situation calculus only based on a lightweight version of the original one. Gerhard Lakemeyer et al. [8] proposed a new logic dialect of situation calculus with the situation terms suppressed, namely, $\mathcal{ES}$. That is, it is merely a similar formalism as a part of the current situation calculus. Moreover, in this paper, the authors consider how to map sentences between $\mathcal{ES}$ and situation calculus and try to prove $\mathcal{ES}$ is powerful enough to handle most cases as the situation calculus does, but mention little about how to type their new logic system as a fragment of situation calculus.

3   Background Knowledge

3.1   Type Systems

In the discipline of computer science, modern type systems are regarded as a formal mechanism originated from Alonzo Church's λ calculus proposed in 1940 [2]. One possible definition of a type system is "a tractable syntactic framework for classifying phrases according to the kinds of values they compute" [12]. By associating types with each computed value, a compiler can detect meaningless or invalid code

written in a given programming language. For instance, the expression "mix = 29 + "Tan"" cannot get through type checking since a string cannot be added to a number.

There are many branches in type systems, such as inferred typing and manifest typing (implicit and explicit), and strong typing and weak typing. As for type checking, people can utilize dynamic type checking and static type checking, or a combination of both.

The primary and most obvious purpose of using type systems is to guarantee the correctness of programs, i.e., detect potential errors, while a well-typed system can further ensure the soundness (safety) of programs. The most important characteristics of a well-typed system are properties of preservation and progress. The former one makes sure a term can keep its type passed into the term that it is evaluated to, and the latter keeps reachability of a term: a typed term can either turn into a value or another related term, which means a well-typed term will not get stuck. In this paper, we plan to equip the current version of situation calculus with appropriate type system besides several original ones for variables. Thus, a program written in situation calculus can be easily type checked correct or not.

3.2   Situation Calculus

Situation Calculus [10] is a formal method based on First Order Logic and Set Theory in Mathematical Logic, with a strong ability of action-based reasoning. This formalism is considered as a dialect of logic programming language and mostly used in dynamic domain modeling. In situation calculus, the world is comprised of situations, actions and objects. The semantics of these three key components in situation calculus is given informally below.

A situation represents a possible world history, simply a sequence of actions, denoted by a first-order term. The constant s_0 is used to denote the initial situation, namely, the empty sequence of actions.

An action represents any possible change to the world, denoted by a function, for example, *drop(A)*, *clean(B)* and *check_in(ID)*.

An object represents an entity defined in the domain of a specific application, denoted by a first-order term, for example, *x*, *robot_A* and *table*.

Moreover, other than aforementioned three elements, there is another significant symbol used frequently in situation calculus, namely, fluents. A fluent represents a relation or a function whose truth values varies from one situation to the next, called relational fluent or functional fluent respectively.

Additionally, introduce two predefined binary symbols of fluents as follows:

Function symbol *do* is defined as *do: Action × Situation → Situation*, which maps an action *a* and a situation *s* to a new situation called successor situation, which results from performing the action *a* in the situation *s*. This successor situation is denoted as *do(a, s)*.

Predicate symbol *Poss* is defined as *Poss: Action × Situation*. Similarly as above, *Poss(a, s)* means it is possible to execute the action *a* in the situation *s*. Note that in the original situation calculus, there is no return value for *Poss*. For consistency, in our well-typed system, we assign a *unit* value for every *Poss* predicate. In other words, *Poss* is defined as *Poss: Action × Situation → Unit* (Capital "U" indicates it is a type but not a value.).

As mentioned before, fluents are used to represent a term whose value varies according to the changing of situations. As a comparison, another symbol is defined to denote a term whose value does varies with situations, namely, predicate. For example, *hunger_status(person, time)* and *weather_condition(location, season)* are relational fluents while *drop(person, object)* is a predicate, since in the first two fluents, the second arguments are actually situations, namely in situation calculus, s, and in the third term there is no specific situation, but only two objects, which means the value of this term will not change when situation changes.

4   A Well-typed Mechanism in Situation Calculus

4.1   A Lightweight Situation Calculus

The situation calculus we study and try to extend here is a lightweight version of its original form. Similarly as Featherweight Java (FJ), we only grab some core features in situation calculus and skip derivable forms to keep our ideas concise and efficient.

According to the language of situation calculus, we keep all the static domain element: situations, actions and objects, and the majority of functional elements like fluents **do** and **poss**, and all the predicates. The components we ignore are those that either can derive from other elements or similarly be expressed by others. For instance, the ordering predicate $\sqsubseteq$, which defines an ordering relation on situations, can be expressed implicitly by the return value of other fluents and predicates. Like the expression $s' \sqsubseteq s$ which denotes that s' is a proper subsequence of s, s' could be replaced with a fluent or predicate which leads s to s', say, **do**(*findajob(person:Object, job:Object), s:Situation)*.

Likewise, we replace countably infinitely many predicate symbols with arity n, $(action \cup object)^n$ with $\bar{t}$, which is a shorthand of a sequence of terms $t_1, t_2, \ldots, t_n$ $(n \geq 1)$.

4.2   Handy Typing Mechanism

In the original situation calculus, several elements such as quantifiers are typed [13]. The handy typed elements are described formally as follows:

A typed notion $\tau(x)$ is used to denote x associated with a finite set of all possible types:

$\tau(x) \stackrel{\text{def}}{=} x : T_1 \vee x : T_2 \vee \ldots \vee x : T_n$, where $T_1, T_2, \ldots, T_n$ are types of terms.

Moreover, typed quantifiers are given by virtue of:

$(\forall x : \tau)\phi(x) \stackrel{\text{def}}{=} (\forall x).\tau(x) \supset \phi(x)$,

$(\exists x : \tau)\phi(x) \stackrel{\text{def}}{=} (\exists x).\tau(x) \wedge \phi(x)$.

Thus, expressions that contain such typed quantifiers could be rewritten as sequences of conjunctions and disjunctions:

$(\forall x : \tau)\phi(x) \equiv \phi(T_1) \vee \phi(T_2) \vee \ldots \vee \phi(T_n)$,

$(\exists x : \tau)\phi(x) \equiv \phi(T_1) \wedge \phi(T_2) \wedge \ldots \wedge \phi(T_n)$.

4.3   A New Type System in the Lightweight Situation Calculus

Although the original version of situation calculus equips some components with corresponding types and semantics, it is not enough to do type checking based on these definitions. We proposed a new well-typed system to enable potential task of type checking in a convenient way.

Syntactic Forms

$t ::= \ldots$		**terms:**
x		*variable*
$\forall x$		*universal quantified variable*
$\exists x$		*existential quantified variable*
$\neg t$		*negative term*
$t_1 \supset t_2$		*subset logical connection*
$t_1 \wedge t_2$		*conjunction logical connection*
$t_1 \vee t_2$		*disjunction logical connection*

$\bar{t}$	*term sequence*
$bt ::= \dots$	**behavioral terms:**
$\neg bt$	*negative behavioral term*
$r(\bar{t},s)$	*relational fluent*
$f(\bar{t})$	*predicate*
$\boldsymbol{do}(bt,s)$	*functional fluent*
$\boldsymbol{poss}(bt,s)$	*predicate fluent*
$v ::= \dots$	**values:**
unit	*poss predicate value*
true	*true boolean value*
false	*false boolean value*
$T ::= \dots$	**types:**
Unit	*type of predicate fluent*
Bool	*type of booleans*
Situation	*type of behavioral terms*
Action	*type of behavioral terms*
Object	*type of terms*

Semantics

Given a world *w* comprised of situations, actions and objects and a set *L(w)* of all instances defined in *w*, if a term *t* holds in *w*, we write $w \models t$. Given a set of situations $S = \{s_0, s_1, \dots, s_n\}$ $(n \geq 0)$, we have:

$$
\begin{aligned}
w &\models x & &\Leftrightarrow & x &\in L(w) \\
w &\models \forall x & &\Leftrightarrow & \forall s_i &\in S, w \models x \\
w &\models \exists x & &\Leftrightarrow & \exists s_i &\in S, w \models x \\
w &\models \neg x & &\Leftrightarrow & w &\not\models x \\
w &\models t_1 \supset t_2 & &\Leftrightarrow & w &\models t_1 \Rightarrow w \models t_2 \\
w &\models t_1 \wedge t_2 & &\Leftrightarrow & w &\models t_1 \text{ and } w \models t_2 \\
w &\models t_1 \vee t_2 & &\Leftrightarrow & w &\models t_1 \text{ or } w \models t_2 \\
w &\models \bar{t} & &\Leftrightarrow & w &\models t_1, w \models t_2, \dots, w \models t_n \\
w &\models \neg bt & &\Leftrightarrow & w &\not\models bt \\
w &\models r(\bar{t},s) & &\Leftrightarrow & w &\models \bar{t} \text{ and } w \models s \text{ in } r \\
w &\models f(\bar{t}) & &\Leftrightarrow & w &\models \bar{t} \text{ in } f \\
w &\models \boldsymbol{do}(bt,s) & &\Leftrightarrow & \exists s_i &\in S, bt \text{ holds in } s_i \\
w &\models \boldsymbol{poss}(bt,s) & &\Leftrightarrow & \exists s_i &\in S, w \models \left(s_i \supset \boldsymbol{do}(bt,s_i)\right)
\end{aligned}
$$

Evaluation Rules $\quad\quad$ t → t'

$$\frac{(x)bt \rightarrow (x')bt}{(\forall x)bt \rightarrow (\forall x')bt} \qquad \text{E-Unv}$$

$$\frac{(x)bt \rightarrow (x')bt}{(\exists x)bt \rightarrow (\exists x')bt} \qquad \text{E-Est}$$

$$\frac{t \rightarrow t'}{\neg t \rightarrow \neg t'}, \frac{bt \rightarrow bt'}{\neg bt \rightarrow \neg bt'} \qquad \text{E-Neg}$$

$$\frac{t_1 \rightarrow t_1'}{t_1 \supset t_2 \rightarrow t_1' \supset t_2} \qquad \text{E-Spt}$$

$$\frac{t_1 \rightarrow t_1'}{t_1 \wedge t_2 \rightarrow t_1' \wedge t_2} \qquad \text{E-Conj}$$

$$\frac{t_1 \rightarrow t_1'}{t_1 \vee t_2 \rightarrow t_1' \vee t_2} \qquad \text{E-Disj}$$

$$\frac{t_1 \rightarrow t_1'}{t_1, t_2, ..., t_n \rightarrow t_1', t_2, ..., t_n} \qquad \text{E-SEQ}$$

$$\boldsymbol{do}(bt,s) \rightarrow [s \mapsto s']bt \qquad \text{E-DO}$$

$$\boldsymbol{poss}(bt,s) \rightarrow s \supset [s \mapsto s']bt \qquad \text{E-POSS}$$

Typing Rules $\qquad\qquad$ **W $\vdash$ t : T**

Here we continue to use *W* (rather than the lower case *w* used in semantics) instead of conventional Γ to denote a typing context. Formally, we have:

$$W \vdash true : Bool \qquad \text{T-TRUE}$$

$$W \vdash false : Bool \qquad \text{T-FALSE}$$

$$\frac{x : T \in W}{W \vdash x : T} \qquad \text{T-VAR}$$

$$\frac{\forall r(x : T, \bar{t}-x, s) \in W}{W \vdash (\forall x : T)\ r(\bar{t}, s)} \qquad \text{T-UNV1}$$

$$\frac{\exists r(x : T, \bar{t}-x, s) \in W}{W \vdash (\exists x : T)\ r(\bar{t}, s)} \qquad \text{T-EST1}$$

$$\frac{\forall f(x : T, \bar{t}-x) \in W}{W \vdash (\forall x : T)\ f(\bar{t})} \qquad \text{T-UNV2}$$

$$\frac{\exists f(x : T, \bar{t}-x) \in W}{W \vdash (\exists x : T)\ f(\bar{t})} \qquad \text{T-EST2}$$

$$\frac{W \vdash t : T, \quad W \vdash bt : T}{W \vdash \neg t : T, \quad W \vdash \neg bt : T} \qquad \text{T-NEG}$$

$$\frac{W \vdash (t_1 : T_1) \supset (t_2 : T_2)}{W \vdash (\forall x \in t_1)\ x : T_1 \supset (\forall y \in t_2)\ y : T_2} \qquad \text{T-SPT}$$

$$\frac{W \vdash (t_1 : T_1) \wedge (t_2 : T_2)}{W \vdash (\forall x \in t_1)\ x : T_1 \wedge (\forall y \in t_2)\ y : T_2} \qquad \text{T-CONJ}$$

$$\frac{W \vdash (t_1 : T_1) \vee (t_2 : T_2)}{W \vdash (\forall x \in t_1)\ x : T_1 \vee (\forall y \in t_2)\ y : T_2} \qquad \text{T-DISJ}$$

$$\frac{W \vdash (t_1 : T_1), (t_2 : T_2), ..., (t_n : T_n)}{W \vdash (\forall x \in t_1)\ x : T_1, ..., (\forall z \in t_n)\ z : T_n} \qquad \text{T-SEQ}$$

$$\frac{W \vdash r : Object \rightarrow Situation \rightarrow Situation,\ \bar{t} : Object,\ s : Situation}{W \vdash r(\bar{t}, s) : Situation} \qquad \text{T-RELFLT}$$

$$\frac{W \vdash f : Object \rightarrow Action \quad W \vdash \bar{t} : Object}{W \vdash f(\bar{t}) : Action} \qquad \text{T-FUNFLT}$$

$$\frac{W, bt : Action \vdash s : Situation}{W \vdash \boldsymbol{do}(bt, s) : Situation} \qquad \text{T-DO}$$

$$\frac{W, bt : Action \vdash s : Situation}{W \vdash \boldsymbol{poss}(bt, s) : Unit} \qquad \text{T-POSS}$$

Item	Description
E-Unv	If one term t' occurred in a given behavioral term bt derives from another term t also in bt, then all such terms t' in bt are also derivable from all such terms t in bt.
E-Est	If one term t' occurred in a given behavioral term bt derives from another term t also in bt, then there exists such a term t' in bt that is derivable from such a term t in bt.
E-Neg	If one term/behavioral term t'/bt' derives from another term t/bt, then `not` t'/bt' also derives from `not` t/bt.
E-Spt	If one term t' derives from another term t, then this also holds in `superset` operation.
E-Conj	If one term t' derives from another term t, then this also holds in `conjunction`.
E-Disj	If one term t' derives from another term t, then this also holds in `disjunction`.
E-Seq	If one term t' derives from another term t, then this relationship holds if t' and t are in a sequence of terms, respectively.
E-Do	In a specific situation s, behavioral term bt gets executed means situation s transits to its successor situation s' while doing bt.
E-Poss	In a specific situation s, behavioral term bt is possible means current situation s is a superset of its successor situation s'.
T-True	As a `Bool` type value, `true` is within the typing map W.
T-False	As a `Bool` type value, `false` is within the typing map W.
T-Var	If a variable x with type T is within the typing map W, then $x : T$ derives from W.
T-Unv1	If all relational fluents r that have an argument x with type T hold, then all occurrence of x in r must have a type T.
T-Est1	If there exists one relational fluent r that has an argument x with type T hold, then there must be one occurrence of x in r with a type T.
T-Unv2	If all functional fluents f that have an argument x with type T hold, then all occurrence of x in r must have a type T.
T-Est2	If there exists one functional fluent f that has an argument x with type T hold, then there must be one occurrence of x in r with a type T.
T-Neg	If one term/behavioral term t'/bt' with a type T derives from the typing map W, then `not` t'/bt' also derives from W with the same type T.
T-Spt	If t_1 with a type T_1 as a superset of t_2 with a type T_2 derives from the typing map W, then in W, all subterms x of t_1, y of t_2 also have types T_1, T_2 respectively, and `superset` relationship still holds.
T-Conj	If the conjunction of t_1 with a type T_1 and t_2 with a type T_2 derives from the typing map W, then all subterms of t_1, t_2 also have types T_1, T_2, satisfying `conjunction`.
T-Disj	If the disjunction of t_1 with a type T_1 and t_2 with a type T_2 derives from the typing map W, then all subterms of t_1, t_2 also have types T_1, T_2, satisfying `disjunction`.
T-Seq	If a sequence of terms with its own types derives from the typing map W, then in W, all subterms of every term have the type their parent has.
T-RelFlt	Straightforward typing relationship of first-order logic.
T-FunFlt	Straightforward typing relationship of first-order logic.
T-Do	Straightforward typing relationship of first-order logic.
T-Poss	Straightforward typing relationship of first-order logic.

Table 1: A Directory of All Evaluation and Typing Rules in the Type System of the Lightweight Situation Calculus

Note:

1. $\bar{t}$ is a shorthand of a sequence of terms $t_1, t_2, \ldots, t_n$ $(n \geq 1)$. Hence $\bar{t}$ cannot possibly be empty.

2. The type *Unit* is defined as the type of value *unit*, where *unit* $\stackrel{\text{def}}{=} \{t | (\forall x \in t)\ x : Bool \vee Situation \vee Action \vee Object\}$, which means all elements in a *unit* should have the same type.

3. $[s \mapsto s']bt$ in the computation rules E-DO and E-POSS means "the successor situation s' that results from executing the behavioral term *bt* in the situation *s*." See the items for E-DO and E-POSS in Table 1.

4. The shadowed typing rules are adapted from the handy typing mechanism for quantifiers in current version of situation calculus, which is discussed in section 4.1.

5. For simplicity, the detailed explanation is not given for typing rules T-RELFLT, T-FUNFLT, T-DO and T-POSS.

5 Evaluation

5.1 Case Description

Let us consider the following scenario: In face of an object x on the floor, say a vase, there is a robot r who wants to pick up this vase and paints it with some color, namely c. In situation calculus, we can describe this scenario using three statements:

In a given situation s that, say, there is a robot r and a vase x ready for situations later on, if the robot r then picked up the vase x and dropped it without holding it firmly, which made the vase became broken, then the vase must be a fragile object:

$$fragile(x,s) \supset broken(x, \boldsymbol{do}(drop(r,x),s)) \tag{1}$$

If the robot successfully picked up the vase x and tried to paint it with one color c, holding it firmly, the vase would turn out to be in the color c:

$$color(x, \boldsymbol{do}(paint(x,c),s)) = c \tag{2}$$

Finally, let us consider the conditions on which it is possible for the robot r to pick up the vase x without any external help. The conditions should be a combination of three: the robot r is not holding any other object z, it is next to x, and x is not heavy:

$$\boldsymbol{poss}(pickup(r,x),s) \supset [(\forall z)\neg holding(r,z,s)] \wedge \neg heavy(x) \wedge nextTo(r,x,s) \tag{3}$$

5.2 Results and Analysis of Type Checking

Now let us do the type checking on the aforementioned three statements that represent a scenario in which a robot r wants to pick up a vase x by itself and paints it with some fancy color c. On the basis of the type system defined in section 4.2, if all the typing goes through and does not get stuck, the program written in situation calculus will be regarded as well-typed.

First, we need to add predefined types for programs written in the original situation calculus by virtue of our new type system. Hence we have:

$$fragile(x : Object, s : Situation) \supset broken(x : Object, \boldsymbol{do}(drop(r : Object, x : Object), s : Situation)) \tag{1'}$$

$$color(x : Object, \boldsymbol{do}(paint(x : Object, c : Object), s : Situation)) = c : Object \tag{2'}$$

$\boldsymbol{poss}(pickup(r : Object, x : Object), s : Situation) \supset [(\forall z : Object)\neg holding(r : Object, z : Object, s : Situation)] \wedge \neg heavy(x : Object) \wedge nextTo(r : Object, x : Object, s : Situation)$ (3')

And then we know the world $w \equiv \{x,s,r,c,z\}$ and $W \equiv \{x : Object, s : Situation, r : Object, c : Object, z : Object\}$.

Now, let us do typing derivation statement by statement. For $(1)'$, We notice that T-SPT cannot be applied since $(1)'$ is of a superset relationship between behavioral terms while T-SPT is for regular terms. Thus, we turn to prove that the type of the left hand side of "⊃" is the same as that of the right hand side.

For typesetting simplicity, we omit "$W \vdash$", return types and the final step of T-VAR, and abbreviate "*Object*", "*Situation*" and "*Action*" to "*Obj*", "*Stn*" and "*Atn*", respectively, in the following typing derivation.

Left hand side of "⊃" in $(1)'$:

$$\frac{fragile : Obj \to Stn \to Stn, \ x : Obj, \ s : Stn}{fragile(x, \ s)}\text{T-RELFLT}$$

Right hand side of "⊃" in $(1)'$:

$$\frac{\dfrac{\dfrac{\dfrac{drop : Obj \to Atn, \ r : Obj, \ x : Obj, \ s : Stn, \ broken : Obj \to Stn \to Stn}{drop(r : Obj, x : Obj), \ s : Stn, \ broken : Obj \to Stn \to Stn}\text{T-FUNFLT}}{\boldsymbol{do}(drop(r : Obj, x : Obj), \ s : Stn), \ broken : Obj \to Stn \to Stn}\text{T-DO}}{broken(x : Obj, \ \boldsymbol{do}(drop(r : Obj, x : Obj), \ s : Stn))}\text{T-RELFLT}}$$

According to this typing derivation, we know that both types of the left hand side and right hand side are the same one: *Situation*. So this situation calculus statement is proved to be well-typed.

For $(2)'$, we have the similar form of typing derivation:

Left hand side of "=" in $(2)'$:

$$\frac{\dfrac{\dfrac{paint : Obj \to Atn, \ x : Obj, c : Obj, \ s : Stn, \ color : Obj \to Stn \to Stn}{paint(x : Obj, c : Obj), \ s : Stn, \ color : Obj \to Stn \to Stn}\text{T-FUNFLT}}{\boldsymbol{do}(paint(x : Obj, c : Obj), \ s : Stn), \ color : Obj \to Stn \to Stn}\text{T-DO}}{color(x : Obj, \ \boldsymbol{do}(paint(x : Obj, c : Obj), \ s : Stn))}\text{T-RELFLT}$$

Right hand side of "=" in $(2)'$:
c: Object

According to this typing derivation, we find that the type of the left hand side is *Situation*, while the right hand side has a type: *Object*, which obviously leads to a mismatch. So this situation calculus statement is proved to be not well-typed. In fact, whatever type of c will bring about stuck terms or mismatches. Anyway, it can still be fixed. A possible correction is to change the right hand side to *inColor(c: Object, s: Situation)*, i.e., to replace c with a corresponding relational fluent to match the type of the left hand side.

Let us check the final sample statement similarly as we did previously:

Left hand side of "⊃" in $(3)'$:

$$\frac{\dfrac{pickup : Obj \to Obj \to Atn, \ r : Obj, \ x : Obj, \ s : Stn}{pickup(r : Obj, \ x : Obj), \ s : Stn}\text{T-FUNFLT}}{\boldsymbol{poss}(pickup(r : Obj, \ x : Obj), \ s : Stn)}\text{T-POSS}$$

When coming to the right hand side of "⊃" in $(3)'$, we notice that T-CONJ cannot be applied since the right hand side of $(3)'$ is of a conjunctive relationship between behavioral terms while T-CONJ is for regular terms. Thus, we turn to check whether types of each part of the conjunction are the same. If so, the final type should be *Unit* according to its definition.

$$\frac{\dfrac{\dfrac{holding : Obj \to Obj \to Stn \to Stn, \ r : Obj, \ z : Obj, \ s : Stn}{holding(r : Obj, \ z : Obj, \ s : Stn)}\text{T-RELFLT}}{\forall \neg holding(r : Obj, \ z : Obj, \ s : Stn)}\text{T-NEG}}{(\forall z : Obj)\neg holding(r : Obj, \ z : Obj, \ s : Stn)}\text{T-UNV1}$$

So we find that $(\forall z)\neg holding(r:Object,z:Object,s:Situation)$ has a type *Situation*. Similarly, we can derive $\neg heavy(x:Object)$ to be of type *Action* by taking two typing derivation steps and $nextTo(r:Object,x:Object,s:Situation)$ to be of type *Situation* by taking one step. By definition, the type of the right hand side of "⊃" in $(3)'$ is not *Unit* since not all the subterms of the conjunction have the same type. Therefore, we can claim that $(3)'$ is not well-typed as well. This time we can fix it intuitively by simply changing the functional fluent $\neg heavy(x:Object)$ into a relational fluent $\neg heavy(x:Object,s:Situation)$.

5.3   Implementation in OCaml

In the last section, types of every term and behavioral term written in a modified lightweight situation calculus with our type system are checked for consistency theoretically. Now we plan to implement a type checker in OCaml which does the same job as we do manually, that is, all the type checking work would be fulfilled by a type checker semi-automatically and efficiently, which can give a great hand for those who are doing tedious type checking alone.

One piece of sample code in OCaml which typechecks situation calculus statement $(1)'$ as described in 5.2 is shown below:

```
(* Types Definition *)
# type unit = Unit of unit;;
# type bool = Bool of bool;;
# type stn  = Situation;;
# type atn  = Action;;
# type obj  = Object;;

(* T-RelFlt *)
# let r t s =
      match t with
          Object -> match s with
                        Situation -> Situation;;

(* T-FunFlt *)
# let f t1 t2 =
      match t1 with
          Object -> match t2 with
                        Object -> Action;;

(* T-Do *)
# let does bt s =
      match bt with
          Action -> match s with
                        Situation -> Situation;;

(* Left Hand Side *)
# let x = Object
```

```
  and s = Situation
  and fragile = r;;
val x : obj = Object
val s : stn = Situation
val fragile : obj -> stn -> stn = <fun>
# fragile (x:obj) (s:stn);;
- : stn = Situation

(* Right Hand Side *)
# let b = Object
  and drop = f
  and broken = r;;
val b : obj = Object
val drop : obj -> obj -> atn = <fun>
val broken : obj -> stn -> stn = <fun>
# broken (x:obj) (does (drop (b:obj) (x:obj)) (s:stn));;
- : stn = Situation

(* This statement is proved to be well-typed *)
```

In the OCaml code above, we firstly defined the types in our type system, and then implemented the
T-RELFLT, T-FUNFLT and T-DO typing rules. Finally some necessary variables, two relational fluents
fragile and *broken*, and a funtional fluent *drop* are declared. As a type checking process, these fluents
fragile, *broken* and *drop* are invoked with inputs of pre-defined variables to show the typing relationship
among them, and the final types calculated for the left hand side and right hand side indicate whether this
statement is well-typed.

In this way, all statements that we typechecked manually just now can be dumped into this type
checker for semi-automatical type checking. Due to some limitation of typing rules in our system, we
do need some additional manual work occasionally. For instance, we need to check by ourselves that
whether the types of the left hand side and right hand side of a symbol "⊃" are the same. Anyway, the
type checker indeed facilitate our process of deciding whether a situation calculus program is well-typed
or not.

6   Conclusions

Type systems have been proposed to guarantee the soundness of program types by rigid typing mech-
anisms. As a popular formal language widely used in Artificial Intelligence related fields, situation
calculus itself has insufficient methods to support a complete and robust type system, with a rudimentary
typing mechanism: only typed quantifiers for variables. It is obviously not enough for type checking
the current situation calculus programs. By virtue of our newly-introduced type system for a lightweight
situation calculus which keep the core of the current one, we can easily do basic type checking for exist-
ing situation calculus programs which are referred a lot in various study of situation calculus. We also
implemented the theoretical type system in OCaml as a type checker to substantiate our ideas. With the
help of this type checker, precedent manual type checking work can be greatly automated for a better
performance. As for the programs checked out to be ill-typed, we provide corresponding ways for cor-

recting them into well-typed forms.

Acknowledgements

References

[1] Jon Barwise (1977): *An introduction to first-order logic*. In: *Handbook of Mathematical Logic, Studies in Logic and the Foundations of Mathematics*.

[2] Alonzo Church (1940): *A Formulation of the Simple Theory of Types*. In: *Journal of Symbolic Logic, Volume 5*.

[3] Jens Claßen, Yuxiao Hu & Gerhard Lakemeyer (2007): *A Situation-Calculus Semantics for an Expressive Fragment of PDDL*. In: *Proceedings of the 22nd National Conference on Artificial Intelligence*, pp. 956–961.

[4] Jens Claßen & Gerhard Lakemeyer (2006): *A Semantics for ADL as Progression in the Situation Calculus*. In: *Proceedings of the 11th Workshop on Non-Monotonic Reasoning*, pp. 334–341.

[5] John Funge (1999): *Representing Knowledge within the Situation Calculus using Interval-valued Epistemic Fluents*. In: *Reliable Computing*, pp. 35–61.

[6] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld & David Wilkins (1998): *PDDL—the planning domain definition language*. In: *Yale Center for Computational Vision and Control Technical Report CVC TR-98-003/DCS TR-1165*.

[7] Yilan Gu & Mikhail Soutchanski (2007): *Decidable Reasoning in a Modified Situation Calculus*. In: *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1891–1897.

[8] Gerhard Lakemeyer & Hector J. Levesque (2005): *Semantics for a useful fragment of the situation calculus*. In: *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 490–496.

[9] Soren B. Lassen (1995): *Basic Action Theory*. In: *BRICS Report Series, Basic Research in Computer Science (BRICS), Department of Computer Science, University of Aarhus*.

[10] John McCarthy (1963): *Situations, actions and causal laws*. In: *Stanford Artificial Intelligence Project, Memo 2*.

[11] Edwin P. D. Pednault (1989): *ADL: Exploring the middle ground between STRIPS and the Situation Calculus*. In: *Proceedings of the International Conference on the Principles of Knowledge Representation and Reasoning*, pp. 324–332.

[12] Benjamin C. Pierce (2002): *Types and Programming Languages*. In: *MIT Press*.

[13] Raymond Reiter (2001): *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. In: *MIT Press*.