A Well-typed Lightweight Situation Calculus

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

Student Presentations of CS 207
Outline

1 Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages
Outline

1. Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2. Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus
Outline

1. Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2. Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus

3. A New Type System in the Lightweight Situation Calculus
   - Syntactic Forms
   - Evaluation Rules
   - Typing Rules
Outline

1. Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2. Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus

3. A New Type System in the Lightweight Situation Calculus
   - Syntactic Forms
   - Evaluation Rules
   - Typing Rules

4. Evaluation
   - Case Description
   - Type Checking
   - Implementation in OCaml
Outline

1 Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2 Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus

3 A New Type System in the Lightweight Situation Calculus
   - Syntactic Forms
   - Evaluation Rules
   - Typing Rules

4 Evaluation
   - Case Description
   - Type Checking
   - Implementation in OCaml

Li Tan | A Well-typed Lightweight Situation Calculus
What is Situation Calculus?

Origin: introduced by John McCarthy (1971 Turing Award Winner) in 1963

Category: a dialect of logic language for dynamic domain modeling

Fundamentals: First Order Logic, Set Theory and Basic Action Theory

Elements: situations, actions and objects

Strength: action-based reasoning

Application: Artificial Intelligence related fields
What is Situation Calculus?

- **Origin**: introduced by John McCarthy (1971 Turing Award Winner) in 1963
What is Situation Calculus?

- **Origin**: introduced by John McCarthy (1971 Turing Award Winner) in 1963
- **Category**: a dialect of logic language for *dynamic domain modeling*
What is Situation Calculus?

- **Origin**: introduced by John McCarthy (1971 Turing Award Winner) in 1963
- **Category**: a dialect of logic language for *dynamic domain modeling*
- **Fundamentals**: First Order Logic, Set Theory and Basic Action Theory
What is Situation Calculus?

- **Origin**: introduced by John McCarthy (1971 Turing Award Winner) in 1963
- **Category**: a dialect of logic language for dynamic domain modeling
- **Fundamentals**: First Order Logic, Set Theory and Basic Action Theory
- **Elements**: situations, actions and objects
What is Situation Calculus?

- **Origin**: introduced by John McCarthy (1971 Turing Award Winner) in 1963
- **Category**: a dialect of logic language for *dynamic domain modeling*
- **Fundamentals**: First Order Logic, Set Theory and Basic Action Theory
- **Elements**: situations, actions and objects
- **Strength**: action-based reasoning
What is Situation Calculus?

- **Origin**: introduced by John McCarthy (1971 Turing Award Winner) in 1963
- **Category**: a dialect of logic language for *dynamic domain modeling*
- **Fundamentals**: First Order Logic, Set Theory and Basic Action Theory
- **Elements**: situations, actions and objects
- **Strength**: action-based reasoning
- **Application**: Artificial Intelligence related fields
Understanding Situation Calculus

In situation calculus, the world is comprised of **situations**, **actions** and **objects**.

- **Situation**: a possible world history, simply a sequence of actions
- **Action**: any possible change to the world. eg.: `drop(robot, vase)`, `clean(people, floor)`
- **Object**: an entity defined in the domain of a specific application. eg.: `x`, `robot_A` and `table`

Other significant symbols to manipulate these key components:

- **Fluents**: relational fluent, functional fluent and predicate fluent
- **Predicate**: usually used to represent `action`
- **Difference**: 
Understanding Situation Calculus

In situation calculus, the world is comprised of situations, actions and objects.

- **Situation**: a possible world history, simply a sequence of actions
- **Action**: any possible change to the world. eg.: \texttt{drop(robot, vase)}, \texttt{clean(people, floor)}
- **Object**: an entity defined in the domain of a specific application. eg.: \texttt{x}, \texttt{robot\_A} and \texttt{table}

Other significant symbols to manipulate these key components:

- **Fluents**: relational fluent, functional fluent and predicate fluent
- **Predicate**: usually used to represent action
- **Difference**:

  \[
  \begin{align*}
  \text{hunger\_status(person, time)} & \quad \text{relational fluent} \\
  \text{weather\_condition(location, season)} & \quad \text{relational fluent} \\
  \text{drop(person, object)} & \quad \text{predicate}
  \end{align*}
  \]
Outline

1. Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2. Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus

3. A New Type System in the Lightweight Situation Calculus
   - Syntactic Forms
   - Evaluation Rules
   - Typing Rules

4. Evaluation
   - Case Description
   - Type Checking
   - Implementation in OCaml
In order to make programs sound and correct in semantics, people have proposed *type systems* in programming languages.
In order to make programs sound and correct in semantics, people have proposed *type systems* in programming languages.

- **Motivation**: "Well-typed programs never go wrong." – Robin Milner
  - Preservation
  - Progress
Types Do Matter in Programming Languages

In order to make programs sound and correct in semantics, people have proposed type systems in programming languages.

- **Motivation**: "Well-typed programs never go wrong." – Robin Milner
  - Preservation
  - Progress

- **Type Systems**: a formal mechanism originated from Alonzo Church’s λ calculus proposed in 1940
  - **Principle**: By associating types with each computed value, a compiler can detect meaningless or invalid code written in a given programming language.
In order to make programs sound and correct in semantics, people have proposed *type systems* in programming languages.

- **Motivation**: "Well-typed programs never go wrong." – Robin Milner
  - Preservation
  - Progress

- **Type Systems**: a formal mechanism originated from Alonzo Church’s $\lambda$ calculus proposed in 1940
  - **Principle**: By associating types with each computed value, a compiler can detect meaningless or invalid code written in a given programming language.

- **Example**: \( \text{mix} = 29 + "\text{Tan}" \)
Outline

1 Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2 Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus

3 A New Type System in the Lightweight Situation Calculus
   - Syntactic Forms
   - Evaluation Rules
   - Typing Rules

4 Evaluation
   - Case Description
   - Type Checking
   - Implementation in OCaml

Li Tan
A Well-typed Lightweight Situation Calculus
Let’s take a look at what we have in original situation calculus:

**Handy Typing Mechanism**

In the original situation calculus, several elements such as quantifiers are typed. 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:

$$\tau(x) \overset{\text{def}}{=} x : T_1 \lor x : T_2 \lor \ldots \lor x : T_n,$$

where $T_1, T_2, \ldots, T_n$ are types of terms.

Moreover, typed quantifiers are given by virtue of:

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

$$(\exists x : \tau) \phi(x) \overset{\text{def}}{=} (\exists x).\tau(x) \land \phi(x).$$

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

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

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

1 Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2 Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus

3 A New Type System in the Lightweight Situation Calculus
   - Syntactic Forms
   - Evaluation Rules
   - Typing Rules

4 Evaluation
   - Case Description
   - Type Checking
   - Implementation in OCaml
We only consider a *lightweight* version of its original form, similarly as *Featherweight Java (FJ)*.

**Core features** are grabbed and **derivable forms** are skimmed to keep a concise idea.

**What can be ignored?**
- those elements that either can derive from other elements or similarly be expressed by others
- $\sqsubseteq \Rightarrow$ *the return value of other fluents and predicates*
- *any symbol $t$ with arity $n \Rightarrow \overline{t}$*
Outline

1. Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2. Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus

3. A New Type System in the Lightweight Situation Calculus
   - Syntactic Forms
   - Evaluation Rules
   - Typing Rules

4. Evaluation
   - Case Description
   - Type Checking
   - Implementation in OCaml

Li Tan
A Well-typed Lightweight Situation Calculus
Syntactic Forms

t ::= \ldots
\begin{align*}
x \\
\forall x \\
\exists x \\
\neg t \\
t_1 \supset t_2 \\
t_1 \land t_2 \\
t_1 \lor t_2 \\
\overline{t}
\end{align*}

bt ::= \ldots
\begin{align*}
\neg \overline{t} \\
r(\overline{t}, s) \\
f(\overline{t}) \\
d(\overline{t}, s) \\
poss(\overline{t}, s)
\end{align*}
v ::= \ldots
\begin{align*}
\text{unit} \\
\text{true} \\
\text{false}
\end{align*}
T ::= \ldots
\begin{align*}
\text{Unit} \\
\text{Bool} \\
\text{Situation} \\
\text{Action} \\
\text{Object}
\end{align*}

terms:
\begin{align*}
\text{variable} \\
\text{universal quantified variable} \\
\text{existential quantified variable} \\
\text{negative term} \\
\text{subset logical connection} \\
\text{conjunction logical connection} \\
\text{disjunction logical connection} \\
\text{term sequence}
\end{align*}

behavioral terms:
\begin{align*}
\text{negative behavioral term} \\
\text{relational fluent} \\
\text{functional fluent} \\
\text{predicate fluent}
\end{align*}

values:
\begin{align*}
\text{poss predicate value} \\
\text{true boolean value} \\
\text{false boolean value}
\end{align*}

types:
\begin{align*}
\text{type of predicate fluent} \\
\text{type of booleans} \\
\text{type of behavioral terms} \\
\text{type of behavioral terms} \\
\text{type of terms}
\end{align*}
Outline

1. Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2. Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus

3. A New Type System in the Lightweight Situation Calculus
   - Syntactic Forms
   - Evaluation Rules
   - Typing Rules

4. Evaluation
   - Case Description
   - Type Checking
   - Implementation in OCaml
Evaluation Rules

\[
\begin{align*}
(t)bt & \rightarrow (t')bt \\
(\forall t)bt & \rightarrow (\forall t')bt \\
(t)bt & \rightarrow (t')bt \\
(\exists t)bt & \rightarrow (\exists t')bt \\
t & \rightarrow t' \\
bt & \rightarrow bt' \\
\neg t & \rightarrow \neg t' \\
t_1 t_2 & \rightarrow t_1' t_2 \\
t_1 & \rightarrow t_1' \\
t_1 t_2 & \rightarrow t_1' t_2 \\
t_1 & \rightarrow t_1' \\
t_1 t_2 & \rightarrow t_1' t_2 \\
t_1 & \rightarrow t_1' \\
t_1, t_2, ..., t_n & \rightarrow t_1' t_2, ..., t_n \\
do(bt, s) & \rightarrow [s \mapsto s']bt \\
poss(bt, s) & \rightarrow s \cup [s \mapsto s']bt \\
t & \rightarrow t' \\
E-UNV \\
E-EST \\
E-NEG \\
E-SPT \\
E-CONJ \\
E-DISJ \\
E-SEQ \\
E-DO \\
E-POSS
\end{align*}
\]
Semantics

Given a world $w$ comprised of situations, actions and objects, if a term $t$ holds in $w$, we write $w \models t$. Given a set of situations $S = s_0, s_1, \ldots, s_n$, we have:

$w \models x \iff x \in L(w)$

$w \models \forall x \iff \forall s_i \in S, w \models x$

$w \models \exists x \iff \exists s_i \in S, w \models x$

$w \models \neg x \iff w \not\models x$

$w \models t_1 \supset t_2 \iff w \models t_1 \Rightarrow w \models t_2$

$w \models t_1 \land t_2 \iff w \models t_1$ and $w \models t_2$

$w \models t_1 \lor t_2 \iff w \models t_1$ or $w \models t_2$

$w \models \bar{t} \iff w \models t_1, w \models t_2, \ldots, w \models t_n$

$w \models \neg bt \iff w \not\models bt$

$w \models r(\bar{t}, s) \iff w \models \bar{t}$ and $w \models s$ in $r$

$w \models f(\bar{t}) \iff w \models \bar{t}$ in $f$

$w \models do(bt, s) \iff \exists s_i \in S, bt$ holds in $s_i$

$w \models poss(bt, s) \iff \exists s_i \in S, w \models (s_i \supset do(bt, s_i))$
A New Type System in the Lightweight Situation Calculus

Outline

1. Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2. Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus

3. A New Type System in the Lightweight Situation Calculus
   - Syntactic Forms
   - Evaluation Rules
   - Typing Rules

4. Evaluation
   - Case Description
   - Type Checking
   - Implementation in OCaml
Typing Rules

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

\[ W \vdash true : \text{Bool} \quad \text{T-TRUE} \]
\[ W \vdash false : \text{Bool} \quad \text{T-FALSE} \]
\[ x : T \in W \quad \text{T-VAR} \]
\[ \forall r(x : T, \bar{t} - x, s) \in W \quad \text{T-UNIV1} \]
\[ \exists r(x : T, \bar{t} - x, s) \in W \quad \text{T-EST1} \]
\[ \forall f(x : T, \bar{t} - x) \in W \quad \text{T-UNIV2} \]
\[ \exists f(x : T, \bar{t} - x) \in W \quad \text{T-EST2} \]
\[ W \vdash t : T \quad \text{T-NEG} \]

\[ W \vdash (t_1 : T_1) \supset (t_2 : T_2) \quad \text{T-SPT} \]
\[ W \vdash (\forall x \in t_1) x : T_1 \supset (\forall y \in t_2) y : T_2 \]
\[ W \vdash (t_1 : T_1) \land (t_2 : T_2) \quad \text{T-CONJ} \]
\[ W \vdash (\forall x \in t_1) x : T_1 \land (\forall y \in t_2) y : T_2 \]
\[ W \vdash (t_1 : T_1) \lor (t_2 : T_2) \quad \text{T-DISJ} \]
\[ W \vdash (\forall x \in t_1) x : T_1 \lor (\forall y \in t_2) y : T_2 \]
\[ W \vdash (t_1 : T_1), (t_2 : T_2), \ldots, (t_n : T_n) \quad \text{T-SEQ} \]
\[ W \vdash (\forall x \in t_1) x : T_1, \ldots, (\forall z \in t_n) z : T_n \]
\[ W \vdash r : \text{Object} \to \text{Situation} \to \text{Situation}, \bar{t} : \text{Object}, s : \text{Situation} \]
\[ W \vdash r(t, s) : \text{Situation} \quad \text{T-RELFLT} \]
\[ W \vdash f : \text{Object} \to \text{Action} \quad W \vdash \bar{t} : \text{Object} \quad \text{T-FUNFLT} \]
\[ W, bt : \text{Action} \vdash s : \text{Situation} \quad \text{T-Do} \]
\[ W \vdash \text{do}(bt, s) : \text{Situation} \]
\[ W, bt : \text{Action} \vdash s : \text{Situation} \quad \text{T-Poss} \]
\[ W \vdash \text{poss}(bt, s) : \text{Unit} \]
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 \( c \).

**Situation Calculus Statements:**

\[
\begin{align*}
\text{fragile}(x, s) & \supset \text{broken}(x, \text{do}(\text{drop}(r, x), s)) \tag{1} \\
\text{color}(x, \text{do}(\text{paint}(x, c), s)) & = c \tag{2} \\
\text{poss}(\text{pickup}(r, x), s) & \supset \left[ (\forall z) \neg \text{holding}(r, z, s) \right] \land \neg \text{heavy}(x) \land \text{nextTo}(r, x, s) \tag{3}
\end{align*}
\]
Statements in Our Type System

Situation Calculus Statements with Types:

\[\text{fragile}(x: \text{Object}, s: \text{Situation}) \supset\]
\[\text{broken}(x: \text{Object}, \text{do}(\text{drop}(r: \text{Object}, x: \text{Object}), s: \text{Situation}))\]  (1)’

\[\text{color}(x: \text{Object}, \text{do}(\text{paint}(x: \text{Object}, c: \text{Object}), s: \text{Situation})) = c: \text{Object}\]  (2)’

\[\text{poss}(\text{pickup}(r: \text{Object}, x: \text{Object}), s: \text{Situation}) \supset\]
\[[(\forall z: \text{Object}) \neg \text{holding}(r: \text{Object}, z: \text{Object}, s: \text{Situation})] \land\]
\[\neg \text{heavy}(x: \text{Object}) \land \text{nextTo}(r: \text{Object}, x: \text{Object}, s: \text{Situation})\]  (3)’
Outline

1. Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2. Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus

3. A New Type System in the Lightweight Situation Calculus
   - Syntactic Forms
   - Evaluation Rules
   - Typing Rules

4. Evaluation
   - Case Description
   - Type Checking
   - Implementation in OCaml

Li Tan
A Well-typed Lightweight Situation Calculus
Let’s take a quick look at how type checking works theoretically:

Left hand side of “⇒” in (1)’:
\[
\frac{\text{fragile}: \text{Obj} \rightarrow \text{Stn} \rightarrow \text{Stn}}{\text{fragile}(x, s)} \quad \frac{x: \text{Obj}, s: \text{Stn}}{\text{T-RelFlt}}
\]

Right hand side of “⇒” in (1)’:
\[
\frac{\text{drop}: \text{Obj} \rightarrow \text{Atn}, r: \text{Obj}, x: \text{Obj}, s: \text{Stn}, \text{broken}: \text{Obj} \rightarrow \text{Stn} \rightarrow \text{Stn}}{\text{drop}(r: \text{Obj}, x: \text{Obj}), s: \text{Stn}, \text{broken}: \text{Obj} \rightarrow \text{Stn} \rightarrow \text{Stn}} \quad \frac{\text{T-FunFlt}}{\text{T-Do}} \quad \frac{\text{T-RelFlt}}{\text{T-Do}}
\]

\[
\frac{\text{do}(\text{drop}(r: \text{Obj}, x: \text{Obj}), s: \text{Stn}), \text{broken}: \text{Obj} \rightarrow \text{Stn} \rightarrow \text{Stn}}{\text{broken}(x: \text{Obj}, \text{do}(\text{drop}(r: \text{Obj}, x: \text{Obj}), s: \text{Stn}))}
\]

\[
\frac{\text{broken}(x: \text{Obj}, \text{do}(\text{drop}(r: \text{Obj}, x: \text{Obj}), s: \text{Stn}))}{\text{T-RelFlt}}
\]
Outline

1. Introduction
   - Situation Calculus
   - Types Do Matter in Programming Languages

2. Motivation
   - Is Situation Calculus Well-typed?
   - A Lightweight Situation Calculus

3. A New Type System in the Lightweight Situation Calculus
   - Syntactic Forms
   - Evaluation Rules
   - Typing Rules

4. Evaluation
   - Case Description
   - Type Checking
   - Implementation in OCaml
One piece of sample code in OCaml is shown below:

```ocaml
# 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;;

(* test *)
# 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
```

Thank you!