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

What is Situation Calculus?

What is Situation Calculus?

- **Origin**: introduced by John McCathy (1971 Turing Award Winner) in 1963

What is Situation Calculus?

- **Origin**: introduced by John McCathy (1971 Turing Award Winner) in 1963
- **Category**: a dialect of logic language for *dynamic domain modeling*

What is Situation Calculus?

- **Origin**: introduced by John McCathy (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 McCathy (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 McCathy (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 McCathy (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.: 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**:

hunger_status(person, time)	*relational fluent*
weather_condition(location, season)	*relational fluent*
drop(person, object)	*predicate*

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

Types Do Matter in Programming Languages

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

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**

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.

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.
- **Example**: mix = 29 + "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

Is Situation Calculus Well-typed?

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:

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

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

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

A Lightweight Situation Calculus

- We only consider a *lightweight* version of its original form, similarly as *Featherweight Java (FJ)*.
- *Core feartures* 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* $n \Rightarrow \bar{t}$

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

Syntactic Forms

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

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

Evaluation Rules $\quad\quad$ $\mathbf{t} \to \mathbf{t'}$

$$\frac{(t)bt \to (t')bt}{(\forall t)bt \to (\forall t')bt} \quad \text{E-UNV}$$

$$\frac{(t)bt \to (t')bt}{(\exists t)bt \to (\exists t')bt} \quad \text{E-EST}$$

$$\frac{t \to t'}{\neg t \to \neg t'}, \quad \frac{bt \to bt'}{\neg bt \to \neg bt'} \quad \text{E-NEG}$$

$$\frac{t_1 \to t_1'}{t_1 \supset t_2 \to t_1' \supset t_2} \quad \text{E-SPT}$$

$$\frac{t_1 \to t_1'}{t_1 \wedge t_2 \to t_1' \wedge t_2} \quad \text{E-CONJ}$$

$$\frac{t_1 \to t_1'}{t_1 \vee t_2 \to t_1' \vee t_2} \quad \text{E-DISJ}$$

$$\frac{t_1 \to t_1'}{t_1,\ t_2,\ \ldots,\ t_n \to t_1',\ t_2,\ \ldots,\ t_n} \quad \text{E-SEQ}$$

$$do(bt, s) \to [s \mapsto s']bt \quad \text{E-DO}$$

$$poss(bt, s) \to s \supset [s \mapsto s']bt \quad \text{E-POSS}$$

Semantics of Evaluation Rules

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:

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

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

Typing Rules $W \vdash t : T$

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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

Situation Calculus Statements:

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

$$color(x, \mathbf{do}(paint(x, c), s)) = c \quad (2)$$

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

Statements in Our Type System

Situation Calculus Statements with Types:

$$fragile(x\colon Object, s\colon Situation) \supset broken(x\colon Object, \mathbf{do}(drop(r\colon Object, x\colon Object), s\colon Situation)) \quad (1)'$$

$$color(x\colon Object, \mathbf{do}(paint(x\colon Object, c\colon Object), s\colon Situation)) = c\colon Object \quad (2)'$$

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

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

Type Checking

Let's take a quick look at how type checking works theoretically:

Left hand side of "$\supset$" in $(1)'$:

$$\frac{fragile{:}Obj \to Stn \to Stn \quad x{:}Obj,\ s{:}Stn}{fragile(x,\ s)}\ \textsc{T-RelFlt}$$

Right hand side of "$\supset$" in $(1)'$:

$$\cfrac{\cfrac{\cfrac{drop{:}Obj \to Atn, r{:}Obj, x{:}Obj,\ s{:}Stn,\ broken{:}Obj \to Stn \to Stn}{drop(r{:}Obj, x{:}Obj),\ s{:}Stn,\ broken{:}Obj \to Stn \to Stn}\ \textsc{T-FunFlt}}{\mathbf{do}(drop(r{:}Obj, x{:}Obj), s{:}Stn),\ broken{:}Obj \to Stn \to Stn}\ \textsc{T-Do}}{broken(x{:}Obj, \mathbf{do}(drop(r{:}Obj, x{:}Obj),\ s{:}Stn))}\ \textsc{T-RelFlt}$$

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

Implementation in OCaml

One piece of sample code in OCaml is shown below:

```
# 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
```

Q & A

Thank you!