Clafer Type System and Attributes

Jimmy Liang and Kacper Bak
Generative Software Development Lab
University of Waterloo

GSD Lab Workshop. February 7, 2012

Recent Progress

Recent Progress

Many people involved

Language

Applications and ecosystem

Documentation and promotion

Recent Progress: Language

Defaults (Leo)

Type system (Jimmy)

Intermediate representation - XML (Jimmy)

Translator (Jimmy, Kacper)

Test suite (Michal, Kacper)

Visualization: CVL and Class-like (Seyi)

Natural language syntax proposal (Michal)

Recent Progress: Applications

Lightweight methodology (Krzysztof, Michal)

Multiobjective optimization (Bo, Derek)

Usability empirical evaluation (Dina)

Experiments with bug tracking (Rafael)

Future: financial domain (Marko), security (with Mahesh)

Recent Progress: Documentation

Tutorial (Michal)

Clafer wiki: github.com/gsdlab/clafer/wiki/ (Michal, Kacper)

Website: clafer.org (Michal, Kacper)

Type System

Motivation

Why does Clafer need a type system?

Type check

Free and automatic sanity check.

```
abstract Y
    x : string
    y : int
    [x + y = 3]
```

Clafer reports that “+” cannot be applied on *x* and *y*.

Semantics

The type of an expression affects the semantics (output Alloy code). Also, needed for code optimization.

```
computer
    dualCpu ?
        speed : integer
    totalSpeed : integer
```

What does the expression *speed* mean?

Two cases

```
computer
    dualCpu ?
        speed : integer
    extraSpeed : integer
    [extraSpeed = (speed ⇒ speed else 0)]
```

The expression *speed* refers 1) presence of clafer, 2) to its integer value.

Semantics OO-analogy

```
class MemberAge {
    int value;
}
class CentenarianClub {
    MemberAge[] memberAge;
}
```

The expression *memberAge* can refer to

- the *MemberAge* object(s)
- or its *value* field

The inferred type of the expression implicitly determines which of the two case applies.

Type system

A Clafer model consists of two parts.

- Clafer definitions
- Constraints

The type system performs two tasks in parallel.

- Type check the expressions in the constraints.
- Infer the types of expressions in the constraints.

Notation & Definition 1

Definition

"::" is shorthand for "is type".

example: "x :: *integer*" is read as "x is type *integer*".

Definition

"$\vdash$" is shorthand for "entails".

example: "$\Gamma \vdash x$ :: *integer*" is read as "Γ entails x :: *integer*".
Sometimes it is clearer to read it as "x :: *integer* given Γ".

Notation & Definition 2

Definition

The letter "x" is a Clafer reference.

In the expression below, "*speed*" is a Clafer reference.

```
[speed > 80]
```

Notation & Definition 3

Definition

The letters "E, F, G" are expressions.

2 leaf expressions: "*speed*" and "80".
1 super expression: "$speed > 80$".

[speed $>$ 80]

Less frequent notations will be explained as they come.

Notation & Definition 4

Definition

A type environment (Γ) is a mapping from Clafer definition to the type of its value.

```
abstract Y : string
    a : integer
    b

X : Y
```

$$\Gamma = \{Y :: string, \quad a :: integer, \quad b :: clafer, \quad X :: string\}$$

Type rule

$$\text{name of rule}\ \frac{\textit{statementA} \qquad \textit{statementB}}{\textit{statementC}}$$

If A and B holds then C follows.

The type system is specified through a series of type rules.

Clafer type rule 1

$$\text{intconst}\ \frac{}{\Gamma \vdash \mathbb{INTEGER} :: \mathit{integer}}$$

Clafer type rule 2

$$\text{eq}\ \frac{\Gamma \vdash E :: \tau \qquad \Gamma \vdash F :: \tau}{\Gamma \vdash E = F :: \mathit{boolean}}$$

Clafer type rule 3

Can we prove that the following model passes type checking?
What is the type of each expression?

```
abstract Y
    [0 = 1]
```

$$\Gamma = \{Y :: \mathit{clafer}\}$$

Clafer type rule 4

$$\Gamma = \{Y :: \mathit{clafer}\}$$

Proof.

$$\text{eq}\ \dfrac{\text{intconst}\ \dfrac{}{\Gamma \vdash\ 0 :: \mathit{integer}} \qquad \text{intconst}\ \dfrac{}{\Gamma \vdash\ 1 :: \mathit{integer}}}{\Gamma \vdash 0 = 1 :: \mathit{boolean}}$$

Clafer type rule 5

$$\text{value}\ \frac{(x :: \tau) \in \Gamma}{\Gamma \vdash x :: \tau}$$

Clafer type rule 6

Prove that the following model is type correct.

```
abstract Y
    a : integer
    [a = 1]
```

$$\Gamma = \{Y :: \mathit{clafer}, \quad a :: \mathit{integer}\}$$

Clafer type rule 7

$$\Gamma = \{Y :: \mathit{clafer}, \quad a :: \mathit{integer}\}$$

Proof.

$$\text{eq}\ \dfrac{\text{value}\ \dfrac{(a :: \mathit{integer}) \in \Gamma}{\Gamma \vdash\ a :: \mathit{integer}} \qquad \text{intconst}\ \dfrac{}{\Gamma \vdash\ 1 :: \mathit{integer}}}{\Gamma \vdash a = 1 :: \mathit{boolean}}$$

Clafer type rule 8

$$\text{clafer}\ \frac{}{\Gamma \vdash x :: \mathit{clafer}}$$

Clafer type rule 9

Prove that the following model is type correct.

```
abstract Y
    a : integer
    b
    [a = b]
```

$$\Gamma = \{Y :: \mathit{clafer}, \quad a :: \mathit{integer}, \quad b :: \mathit{clafer}\}$$

Clafer type rule 10

$$\Gamma = \{Y :: clafer, \quad a :: integer, \quad b :: clafer\}$$

Proof.

$$\text{eq}\ \dfrac{\text{clafer}\ \dfrac{}{\Gamma \vdash\ a :: clafer} \qquad \text{clafer}\ \dfrac{}{\Gamma \vdash\ b :: clafer}}{\Gamma \vdash a = b :: boolean}$$

Clafer type rule precedence

```
abstract Y
    a : integer
    b : integer
    [a = b]
```

Integer equality or clafer equality?

Clafer type rule casting 1

```
abstract Y
    a : real
    b : integer
    [a = b]
```

Clafer type rule casting 2

$$\text{eqcast1}\ \frac{\Gamma \vdash E :: \mathit{real} \qquad \Gamma \vdash F :: \mathit{integer}}{\Gamma \vdash E = F :: \mathit{boolean}}$$

$$\text{eqcast2}\ \frac{\Gamma \vdash E :: \mathit{integer} \qquad \Gamma \vdash F :: \mathit{real}}{\Gamma \vdash E = F :: \mathit{boolean}}$$

Attributes

Primitive Types

```
Version : int *
[Version + 2 > 0]
```

What is the value of Version?

Is the constraint satisfied? When?

Is the meaning of “+” purely arithmetic?

Does “+” handle sets?

Primitive Types

```
Version : int *
[Version + 2 > 0]
```

What is the value of Version?

Is the constraint satisfied? When?

Is the meaning of “+” purely arithmetic?

Does “+” handle sets?

Primitive Types

```
Version : int *
[Version + 2 > 0]
```

What is the value of Version?

Is the constraint satisfied? When?

Is the meaning of "+" purely arithmetic?

Does "+" handle sets?

Primitive Types

```
Version : int *
[Version + 2 > 0]
```

What is the value of Version?

Is the constraint satisfied? When?

Is the meaning of “+” purely arithmetic?

Does “+” handle sets?

Primitive Types

```
Version : int *
[Version + 2 > 0]
```

What is the value of Version?

Is the constraint satisfied? When?

Is the meaning of “+” purely arithmetic?

Does “+” handle sets?

Semantic Variants

Sum

Version : int *
[Version $+ 2 > 0$]

Semantics: [sum(Version) + 2 > 0]

Version 0: returns 0. OK

Version 1: OK

Version +: Set sum. Confusing

abstract comp
  version : int
[comp.version = 1]

Used by Alloy and CDL

Sum

```
Version : int *
[Version + 2 > 0]
```

Semantics: [sum(Version) + 2 > 0]

Version 0: returns 0. OK

Version 1: OK

Version +: Set sum. Confusing

```
abstract comp
  version : int
[comp.version = 1]
```

Used by Alloy and CDL

Sum

```
Version : int *
[Version + 2 > 0]
```

Semantics: [sum(Version) + 2 > 0]

Version 0: returns 0. OK

Version 1: OK

Version +: Set sum. Confusing

```
abstract comp
  version : int
[comp.version = 1]
```

Used by Alloy and CDL

Sum

```
Version : int *
[Version + 2 > 0]
```

Semantics: [sum(Version) + 2 > 0]

Version 0: returns 0. OK

Version 1: OK

Version +: Set sum. Confusing

```
abstract comp
  version : int
[comp.version = 1]
```

Used by Alloy and CDL

Universal Quantification

```
Version : int *
[Version + 2 > 0]
```

Semantics: [all a : Version | a + 2 > 0]

Version 0: ignores constraint

Version 1: OK

Version +: OK

Universal Quantification

```
Version : int *
[Version + 2 > 0]
```

Semantics: [all a : Version | a + 2 > 0]

Version 0: ignores constraint

Version 1: OK

Version +: OK

Enforced Presence

```
Version : int *
[Version + 2 > 0]
```

Semantics: [some Version && Version + 2 > 0]

Version 0: constraint unsatisfied (as in OCL)

Version 1: OK

Version +: unclear. Sum or universal quantification.

Enforced Presence

```
Version : int *
[Version + 2 > 0]
```

Semantics: [some Version && Version + 2 > 0]

Version 0: constraint unsatisfied (as in OCL)

Version 1: OK

Version +: unclear. Sum or universal quantification.

Explicit Guards

```
Version : int *
[Version + 2 > 0]
```

Semantics: [(Version ? Version : 0) + 2 > 0]

Version 0: OK

Version 1: OK

Version +: unclear. Sum or universal quantification.

Explicit Guards

```
Version : int *
[Version + 2 > 0]
```

Semantics: [(Version ? Version : 0) + 2 > 0]

Version 0: OK

Version 1: OK

Version +: unclear. Sum or universal quantification.

Vector Operations

```
Version : int *
[Version + 2 > 0]
```

Semantics: [Version(0) + 2 > 0, Version(1) + 2 > 0, ...]

Version 0: unclear

Version 1: OK

Version +: OK, but complex.

Vector Operations

```
Version : int *
[Version + 2 > 0]
```

Semantics: [Version(0) + 2 > 0, Version(1) + 2 > 0, ...]

Version 0: unclear

Version 1: OK

Version +: OK, but complex.

No good semantics!

Proposed Solution

```
Version : int *
[Version + 2 > 0]
```

Use the sum semantics. Covers 0 and 1 cardinality.

For higher cardinalities, enforce using quantifiers.

Complex but makes intuitive sense.

Discussion:
github.com/gsdlab/clafer/wiki/Experimental:-Attributes

Conclusion

Conclusion

Clafer entered new evolution phase

Type system clarifies semantics

Experiments with design choices

Evaluation needed

Upcoming applications

Thanks for listening!

Questions?

clafer.org