Understanding How You Should (Not) Mix Programming Features

(Project Title: *A Proof Assistant for Contextual Equivalences, Using Hierarchical Graph Rewriting*)

Koko Muroya (RIMS, Kyoto University)

Programming features & behaviour

Programming = the act of combining features

- arithmetic
- conditional branching
- loop, recursion
- mutable state, reference
- system call
- random number generation
- error handling, callback
- …

`1 + 2` ≃ `3`

`if true then P else Q` ≃ `P`

```
int i;
for (i=0; i<5; i++) {
  f(i);
}
```
≃
```
f(0);
f(1);
f(2);
f(3);
f(4);
```

```
int i = 0;
i = 1;
i;
```
≃ `1`

```
int i = 0;
f(5);
```
≃ `f(5);`

Contextual equivalence:

expected behaviour of each feature, in an equational form, meaning "*two sides act the same in any programs*"

Potential danger

Actual behaviour of features depends on how they are mixed:

- **Safe** combination yields expected behaviour
- **Dangerous** combination may yield **undesired** behaviour
  - Example: arithmetic & system call (getting current time)

Possible "bad" programs:

```
t0 = gettime();
n = 1 + 2;
t1 = gettime();
print(t1 - t0);
return n;
```

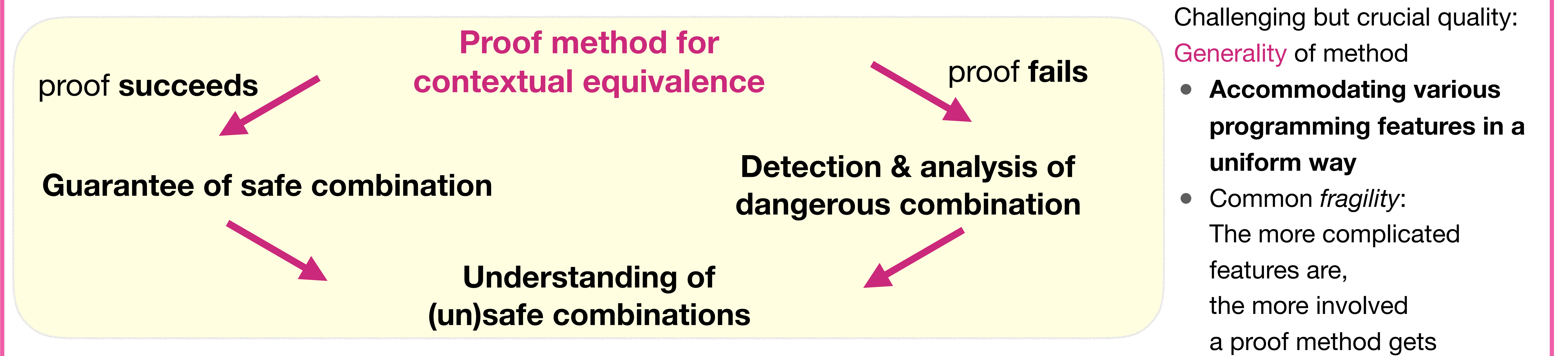
```
t0 = gettime();
n = 3;
t1 = gettime();
print(t1 - t0);
return n;
```

- using system call
- *distinguishing* two sides of `1 + 2` ≄ `3`

Undesired behaviour (i.e. violation of contextual equivalence)

→ No safety of compiler optimisation & refactoring

Goal: Mathematical method of understanding (un)safe combinations

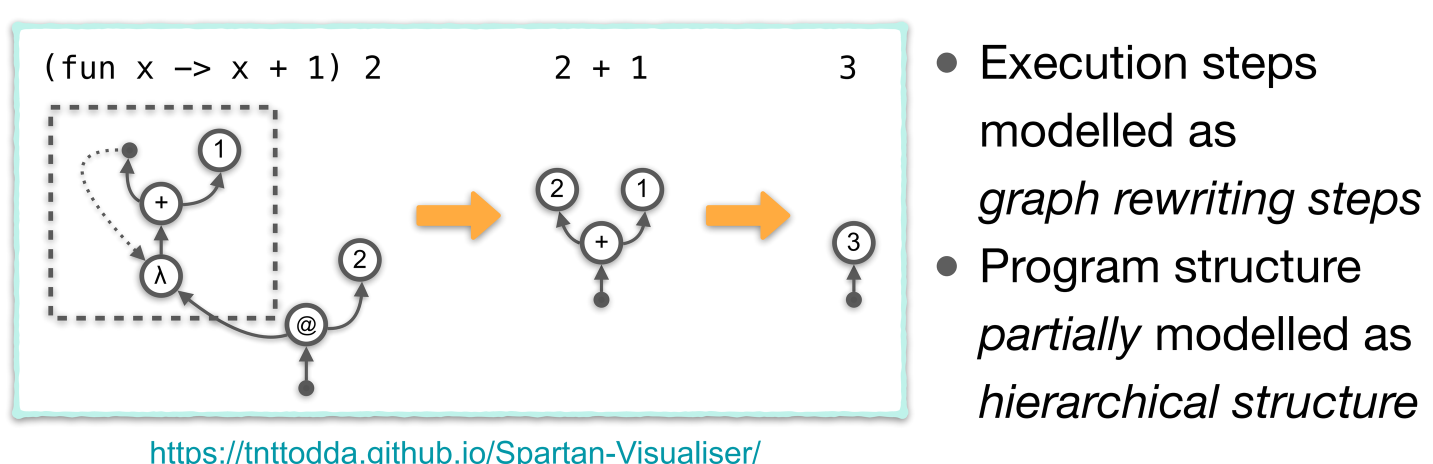


Challenging but crucial quality: Generality of method

- **Accommodating various programming features in a uniform way**
- Common *fragility*: The more complicated features are, the more involved a proof method gets

Proof idea for contextual equivalence

Step 1: Modelling program execution as **hierarchical graph rewriting**

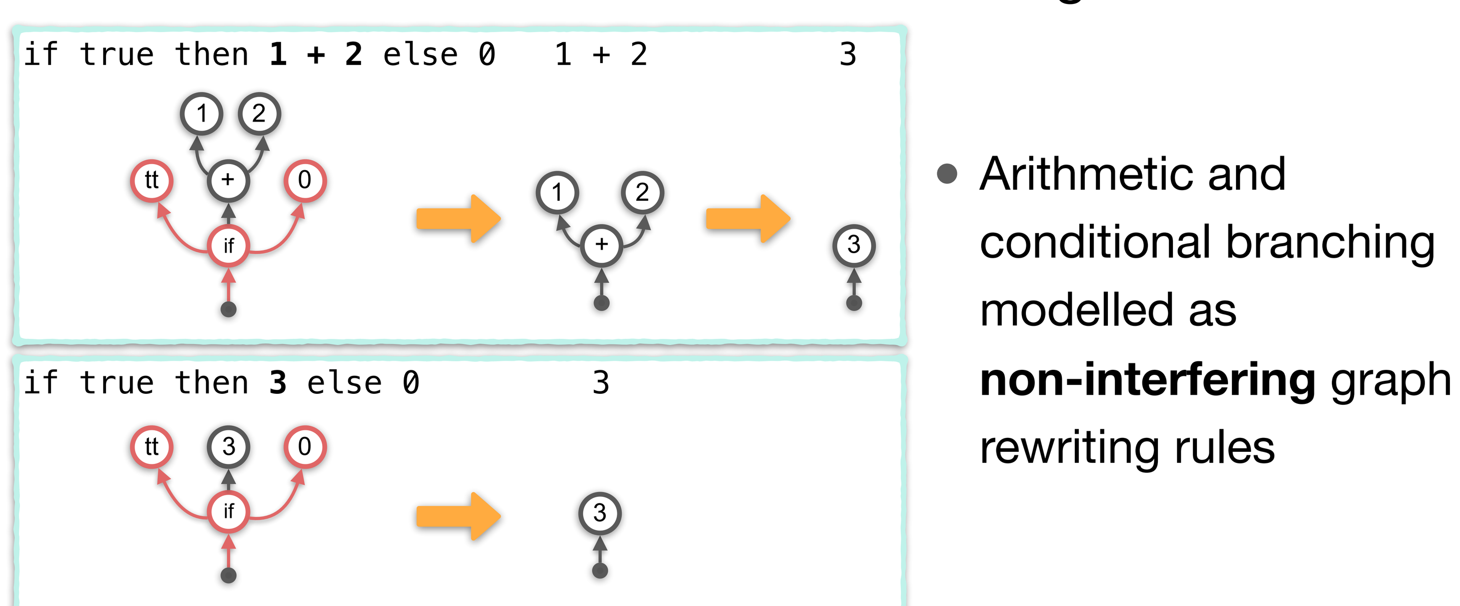


https://tnttodda.github.io/Spartan-Visualiser/

- Execution steps modelled as *graph rewriting steps*
- Program structure *partially* modelled as *hierarchical structure*

Step 2: Checking *robustness* of `M` and `N`

- Example: robustness of `1 + 2` and `3` relative to conditional branching



- Arithmetic and conditional branching modelled as **non-interfering** graph rewriting rules

Step 3: Proving the main theorem

"If `M` and `N` are robust, then `M` ≃ `N` holds"

Progress so far & objectives

Prototypical method

(a part of PhD thesis; with ideas presented at workshops e.g. LOLA 2019)

- Supporting **deterministic features**
  - ✓ arithmetic, conditional branching, recursion, mutable state, (error handling, callback)
  - × random number generation
  - → **Extension to non-deterministic features**
    - Extension of definition of contextual equivalence
    - Modification of the main theorem (Step 3)
- Working fine, but **mathematically a little rough**
  - → **Consulting related theories**
    - Rewriting theory, category theory, theory of state transition systems, graph theory, …
- Involving (intuitive) **case analysis** for robustness check

  Example: identifying & analysing *all* patterns of interferences between graph rewriting rules that implement features

  - → **(Semi-)automation** of case analysis, in particular, case enumeration