Refactoring for Exploratory Specification in VDM-SL

Tomohiro Oda
Keijiro Araki
Shin Sahara
Han-Myung Chang
Peter Gorm Larsen

Overture 19, Oct 22, 2021 @Aarhus
(online from Hamamatsu, Japan)

agenda

- refactoring
- names
- tooling
- discussion, summary and future work

refactoring

background: refactoring

Refactoring is the process of changing a software system in such a way that it does **not alter the external behavior of the code yet improves its internal structure**. --Martin Fowler, "Refactoring: Improving the Design of Existing Code", 1999

related work: Refactoring Support for VDM-SL

Magnusl Pedersen and Peterk Mathiesen, "Refactoring Support for VDM-SL", 2017

- features refactoring for readability and maintainability
- Identified a good collection of refactoring operations
- built a support tool on the Overture tool
- evaluated by refactoring the ALARM model

names

background: Level 0 Formal Specification

Identify what should be named, and give to it a concise definition

- encyclopedia of the system
  - a collection of entry names that together figures the system
    - explicit definitions of names
    - implicit definitions with constraints among names
- formal spec = math + **literature**
  - formula and wording to make sense
  - Names are significant constituent of the literature aspect of the specification

concerns on name

- spelling
  - grounding to the real world
- scope
  - name crash, scope shadowing, modules
- definition
  - sense making
- references
  - networking with other names

refactoring for managing these concerns

revisit: refactoring

not alter the external behavior of the code
yet improves its internal structure

e.g. extract method, rename tmp var, delete class, and so on

refactoring to develop a vocabulary of names

not alter the external behavior of the code

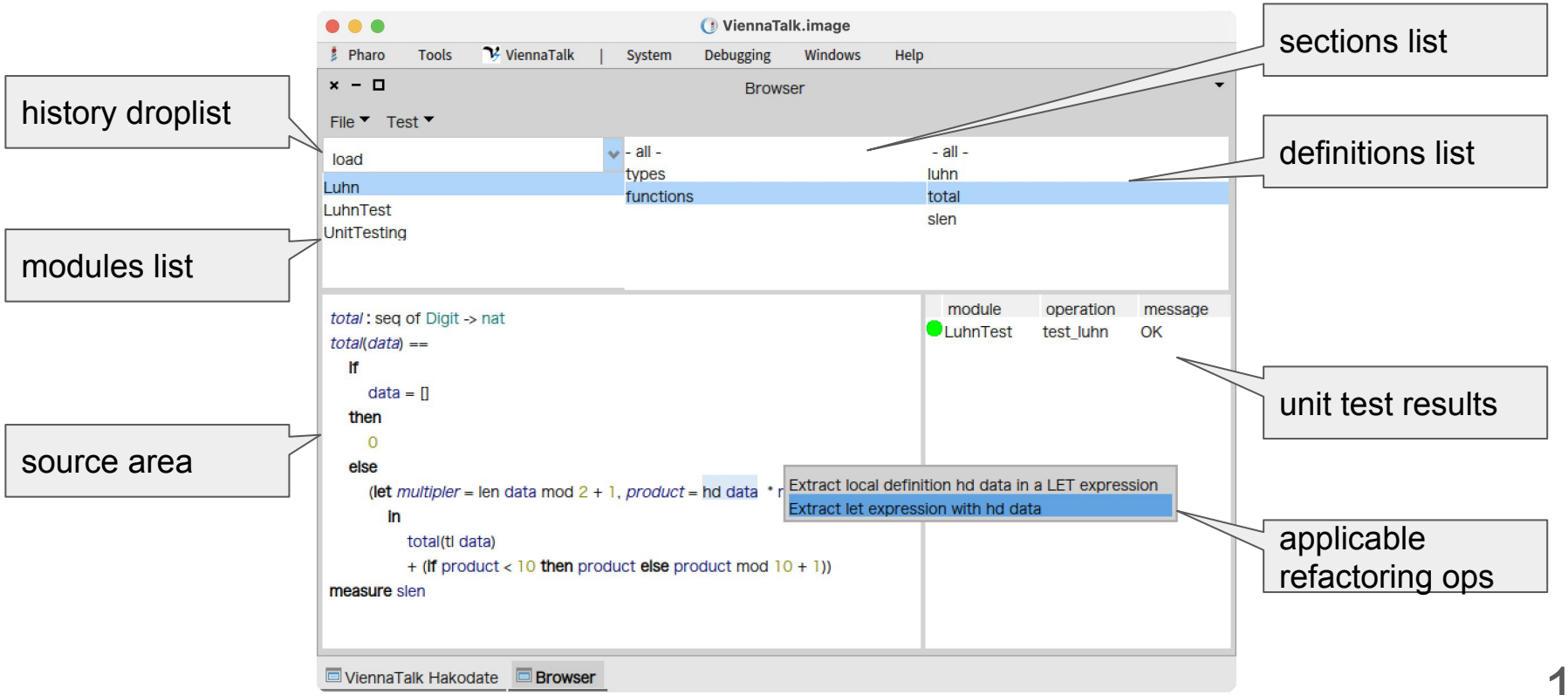
vocabulary

yet improves its ~~internal structure~~

e.g. extract method, rename tmp var, delete class, and so on

tooling

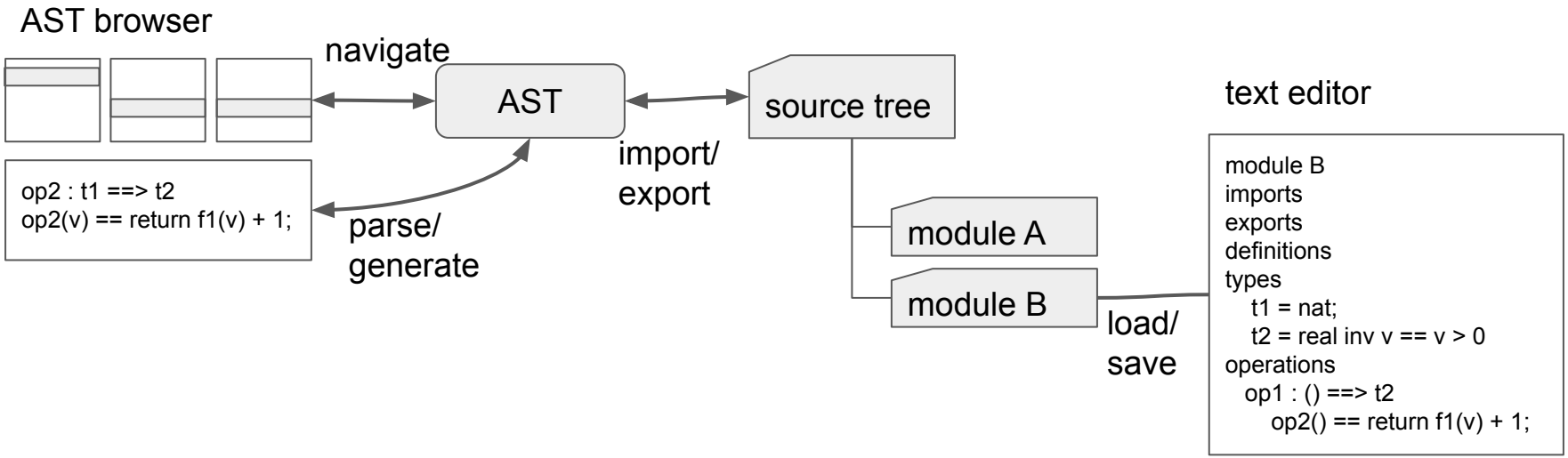
refactoring browser on ViennaTalk



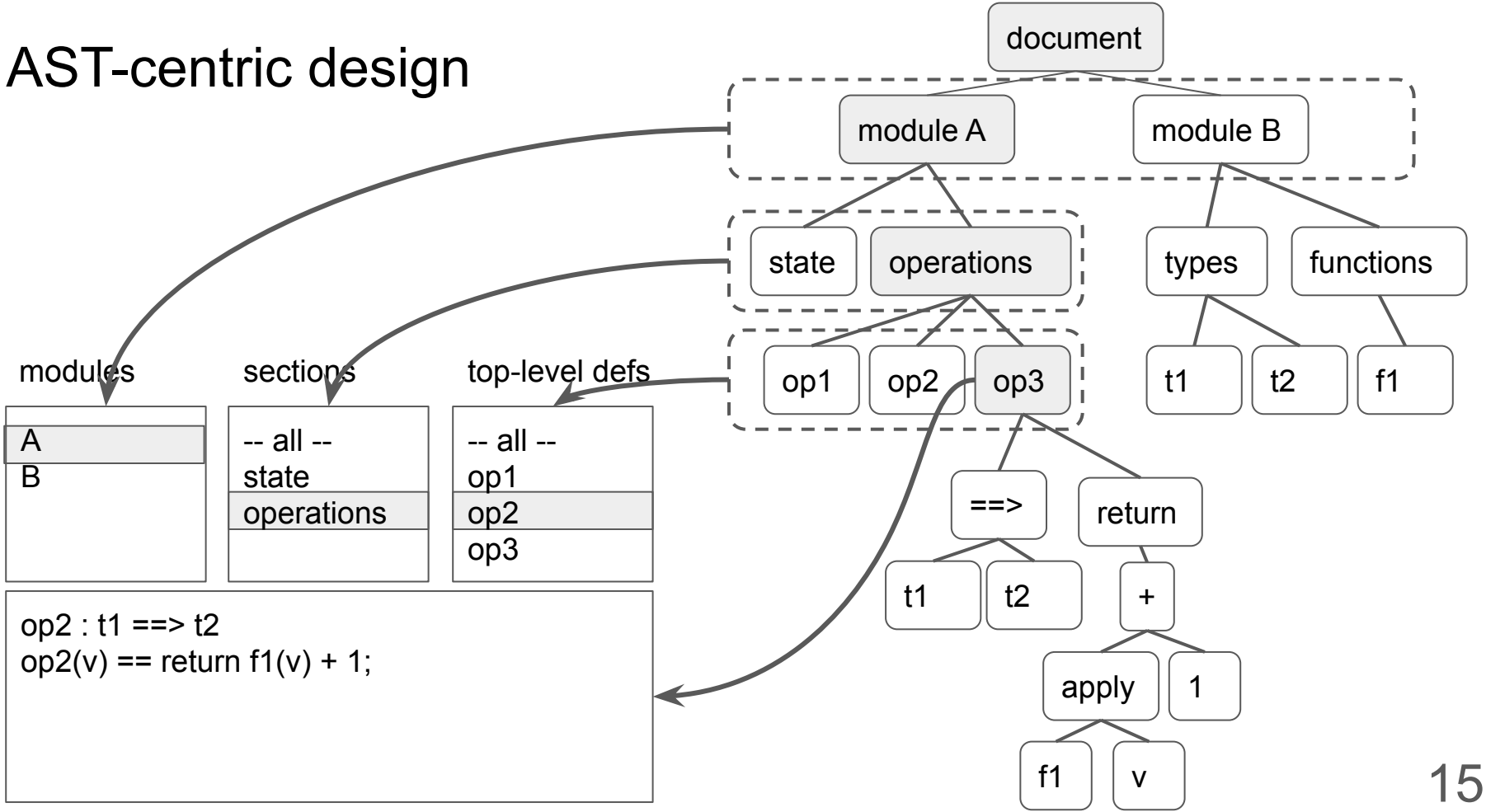
refactoring browser on ViennaTalk

- fully AST centric
  - Textual source is generated only for external IO with humans and disks.
  - offers a list of applicable refactoring operations at the specified AST node.
- pluggable
  - Subclasses of `ViennaRefactoring` are scanned and automatically listed
    - `check` to test applicability
    - `name` to print and `sortingOrder` to list
    - `execute` to apply
- running unit tests
  - runs all tests at every modification to the AST.
- history
  - A full copy of AST is automatically stored at every modification.

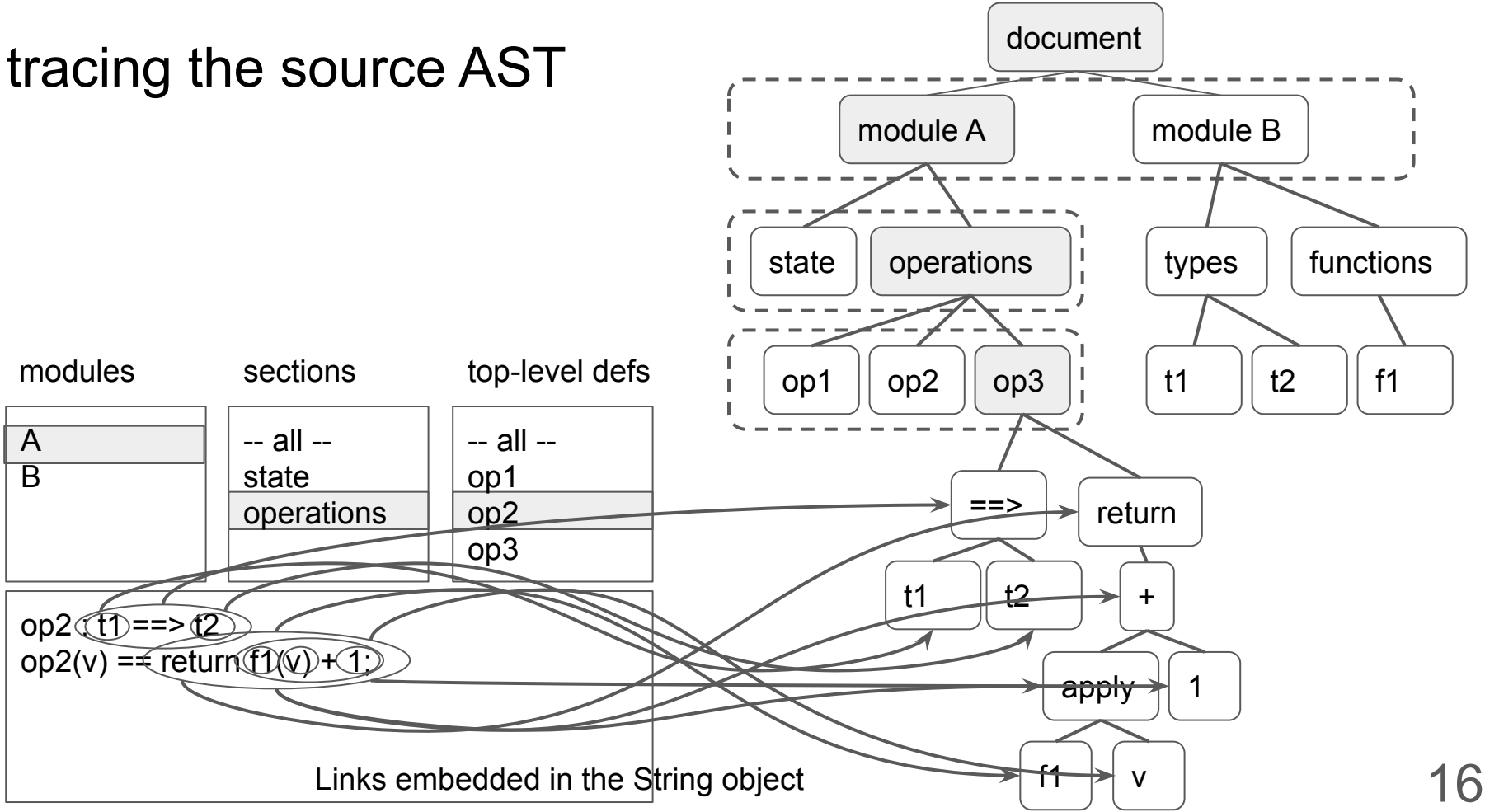
AST-centric browser vs text editor



AST-centric design



tracing the source AST



refactoring operations in ViennaTalk

add name

- extract
  - function from let expression
  - operation from let expression/statement
  - state variable
  - type
  - value
  - let expression/statement
  - parameter
    - local definition in let expression
    - in implicit/explicit function/operation

top-level names

refactoring operations in ViennaTalk

change name

- spelling
  - explicit/implicit function/operation, state constructor/variable, type alias/quote, value
  - import
  - local
- scope
  - widen/narrow let expression/statement
  - split let expression/statement, split block statement
- definition
  - convert from function to pure operation
- reference
  - use function, type, value
  - inline function, type, value
  - inline let, let parameter, block
- move across modules
  - function, operation, type, value

top-level names

refactoring operations in ViennaTalk

remove name

- inline and remove
  - function, type, value
  - let expression/statement, local definition
- remove top-level names
  - operation

discussion

what could be the type of `3.14*radius*radius` ?

```
types
   LENGTH = real inv l == l >= 0;
   AREA = real inv a == a >= 0;

values
   AREA_THRESHOLD:AREA = 100.0;

functions
   isBigCircle : LENGTH -> bool
   isBigCircle(radius) ==
      3.14 * radius * radius
      > AREA_THRESHOLD;
```

extract let

```
types
   LENGTH = real inv l == l >= 0;
   AREA = real inv a == a >= 0;

values
   AREA_THRESHOLD:AREA = 100.0;

functions
   isBigCircle : LENGTH -> bool
   isBigCircle(radius) ==
      let area = 3.14 * radius * radius
      in area > AREA_THRESHOLD;
```

what is the type of `3.14*radius*radius` in the possible well-formedness?

```
types
    LENGTH = real inv l == l >= 0;
    AREA = real inv a == a >= 0;

values
    AREA_THRESHOLD:AREA = 100.0;

functions
    isBigCircle : LENGTH -> bool
    isBigCircle(radius) ==
        let area:? = 3.14 * radius * radius
        in area > AREA_THRESHOLD;
```

`area` could be typed either
unconditionally `real`,
unlikely `int`, `nat`, `nat1`,
fairly possibly `LENGTH` or `AREA`.

what if area is typed `AREA` ?

```
functions
        isBigCircle : LENGTH -> bool
        isBigCircle(radius) ==
                let area:AREA = 3.14 * radius * radius
                in area > AREA_THRESHOLD;
```

`AREA` is more informative than `real` because it means an area and also asserts `area>=0`.

extract function from let expression

```
functions
        isBigCircle : LENGTH -> bool
        isBigCircle(radius) == isBig(3.14*radius*radius);

        isBig : AREA -> bool
        isBig(area) == area > AREA_THRESHOLD;
```

`AREA -> bool` is more informative than `real -> bool`.

summary and future work

- refactoring for specification language
  - categorized by effects on names
  - refactoring to make more informative
  - tool on ViennaTalk
    - AST centric
    - auto test runner
    - history to revert the change
- TODO
  - preview of refactoring operation
    - to graphically illustrate AST manipulation than source diff