OCL parsing / type checking in the context of GF and KeY

Kristofer Johannisson

I. Introduction

Typechecking?

```
context OwnerPIN inv:
    maxPINSize > 0 and maxTries > 0 and
    triesRemaining >= 0 and triesRemaining <= maxTries

context OwnerPIN::reset()
post: not excThrown(java::lang::Exception) and
      not self.isValidated and
      if self.isValidated@pre then
          self.triesRemaining = self.maxTries
      else
          self.triesRemaining = self.triesRemaining@pre
      endif
```

Motivation

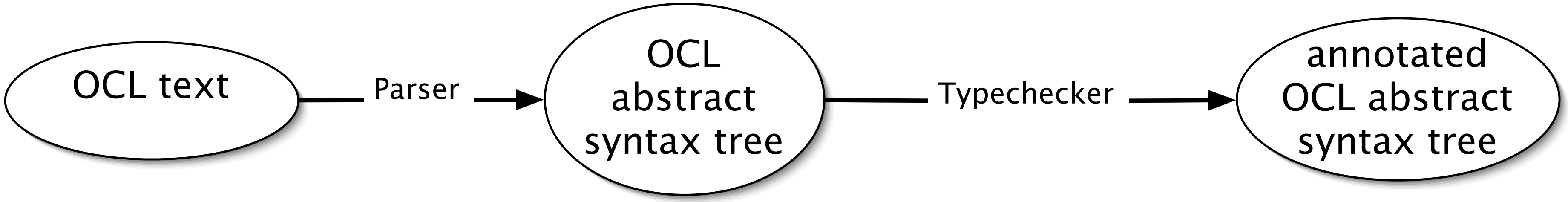
there is need for an OCL parser/typechecker:

- a component of the GF-based rendering of OCL in natural language
- a component in KeY
  - OCL → DL translation
  - partial evaluation of OCL

II. Typechecking OCL

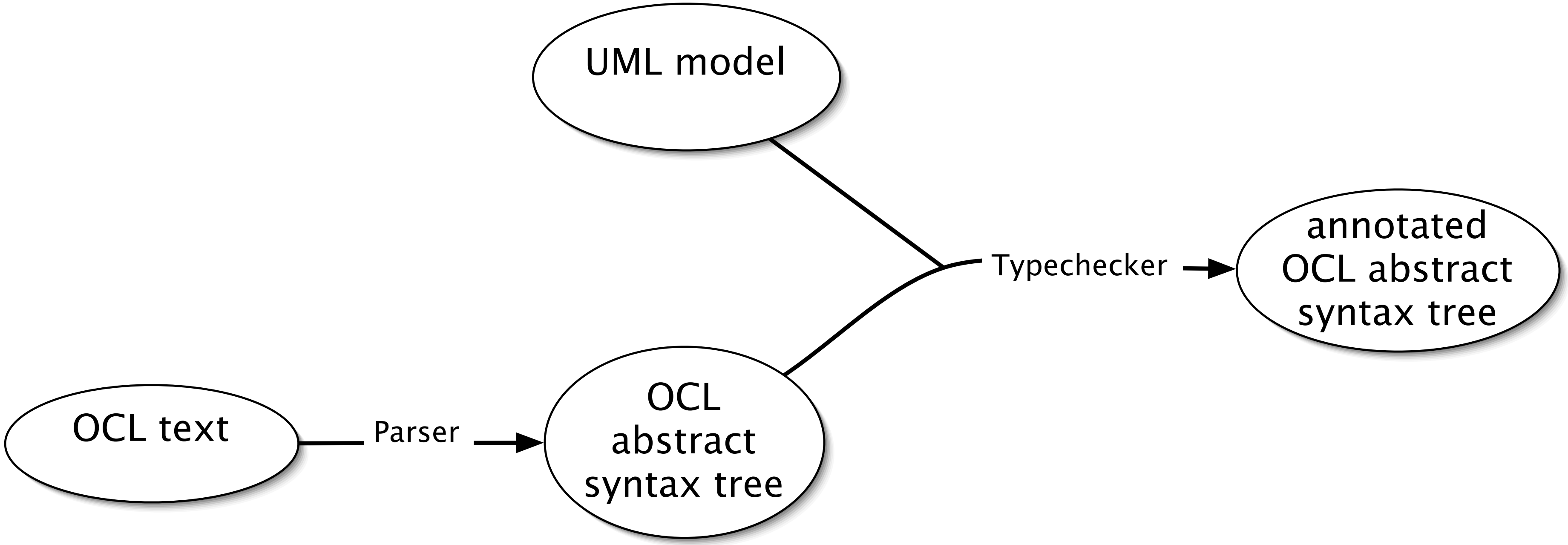
Overview (1)

We go from strings to abstract syntax trees to annotated abstract syntax trees:



Overview (2)

Typechecking is done with respect to an UML model:



General

Side-effect free expressions and let-definitions are used to form class invariants, pre-/postconditions

```
context Person::income(c:Currency) : Integer post:
    let
        hasTitle(t:String) : Boolean = self.jobs->exists(title = t)
    in
    (hasTitle('professor') implies result > c.fromEuro(3000)) and
    (hasTitle('phd student') implies result < c.fromEuro(3000))
```

Implicit goal: Translation into GF

GF abstract syntax trees give a semantic view of OCL specifications. E.g. they contain type annotations, and subtyping is handled with explicit coercion functions. Hence the OCL typechecker should:

- annotate every expression with its type
- insert explicit coercions whenever subtyping is used
- introduce other semantic distinctions

Annotation of OCL terms

$$\Sigma, \Gamma \vdash t \triangleright t'$$

- Σ (theory) contains classes and properties (attributes, operations, queries, associations) provided by OCL library and user UML model

- Γ (context) contains bindings for variables (e.g. `self`) and let-definitions

- t is an OCL term, t' is an annotated OCL term

Foundations

- OMG specification of OCL
- Tony Clark 1999 *Typechecking UML Static Models*
- Cengarle, Knapp 2003 *OCL 1.4/5 vs. 2.0 Expressions — Formal semantics and expressiveness*

Systematic description of our work is forthcoming

Example: if-then-else

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash c \triangleright c' \ : C_1 \\ \Sigma \vdash C_1 \ \ <: \ \texttt{Boolean} \\ \Sigma, \Gamma \vdash t \triangleright t' \ : C_2 \\ \Sigma, \Gamma \vdash e \triangleright e' \ : C_3 \\ \Sigma \vdash C = \mathsf{lub}\ \{C_1, C_2\}\end{array}}{\Sigma, \Gamma \vdash \texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ e\ \triangleright\ \texttt{if}\ [c']_{C_1 <: \texttt{Boolean}}\ \texttt{then}\ [t']_{C_2 <: C}\ \texttt{else}\ [e']_{C_3 <: C}\ : C}$$

where $[t]_{A <: B}$ is an explicit coercion of term t from class A to A:s superclass B.

Example: Implicit self

$$\frac{\Sigma, \Gamma \vdash \texttt{self}.query(t_1, \dots t_n) \triangleright \texttt{self}.query'(t'_1, \dots t'_n) \ : C}{\Sigma, \Gamma \vdash query(t_1, \dots t_n) \triangleright \texttt{self}.query'(t'_1, \dots t'_n) \ : C}$$

“Property calls” (1)

PropCall. Expr ::= Expr ('`.`' | '`->`') Ident ('`(`' Decl? [Expr] '`)`')?

Examples:

`self.query(`x_1`,...`x_n`)`

`coll->size()`

`coll->forAll(x,y | x = y)`

“Property calls” (2)

- Function application
  - `self.attr`, `self.query(`x_1`,...`x_n`)`, `self.assoc`, `coll->size()`
- Variable binding constructions
  - `coll->forAll(x,y | x = y)`, `coll->collect(x | x.attr)`
  - primitive recursion over collections
    - `coll->iterate(x; acc : Integer = 0 | x+acc)`

“Property calls” (3)

- Implicit variable binding
  - `coll->forAll(x | x.age > 18)` can be written as `coll->forAll(age > 18)`
- Implicit collect
  - `coll->collect(x | x.age)` can be written as `coll.age`
- Associations of multiplicity 0..1 can be considered as sets or not
  - `self.husband->notEmpty() implies self.husband <> self`

Other features

- JavaCard support, e.g. null and exceptions
- meta-level operations, e.g. `allInstances` and `oclAsType`
- flattening of collections

OCL 2.0

- records (“tuples”)
- nested collections
- no let-definitions with arguments outside `def:` constraints
- changes to `OclType`
- messages
- ...

Status: current limitations

- flattening
- qualified associations, association classes
- enumerations
- OCL2.0

Status: implemented features

- implicit self, implicit bound variables, implicit collect
- navigation to singleton associations
- let definitions (currently only without arguments)
- meta-operations `allInstances` and `oclAsType` on class literals
- `null`, `excThrown`
- packages

Implementation

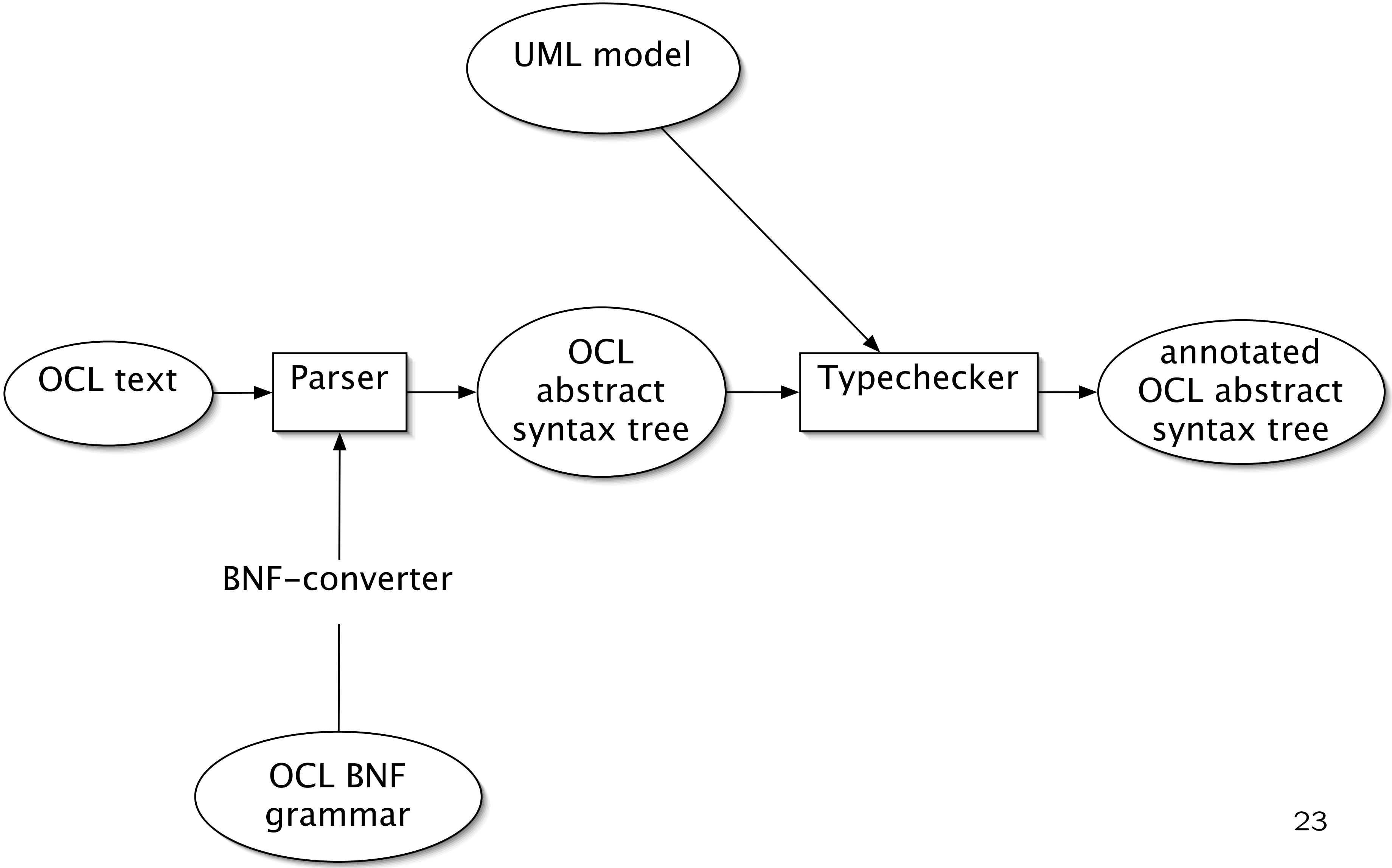
- BNF converter (BNFC) [Ranta et al.]
  - front-end to standard lexer and parser generators

- Haskell
  - GF is implemented in Haskell

The BNF converter

Given a *Labelled BNF grammar* the tool produces:

- an abstract syntax in Haskell / C++ / C / Java
- a case skeleton for the abstract syntax
- an Alex, JLex or Flex lexer generator file
- a Happy, CUP or Bison parser generator file
- a pretty-printer in Haskell / C++ / C / Java
- a readable specification of the language (LaTeX file)

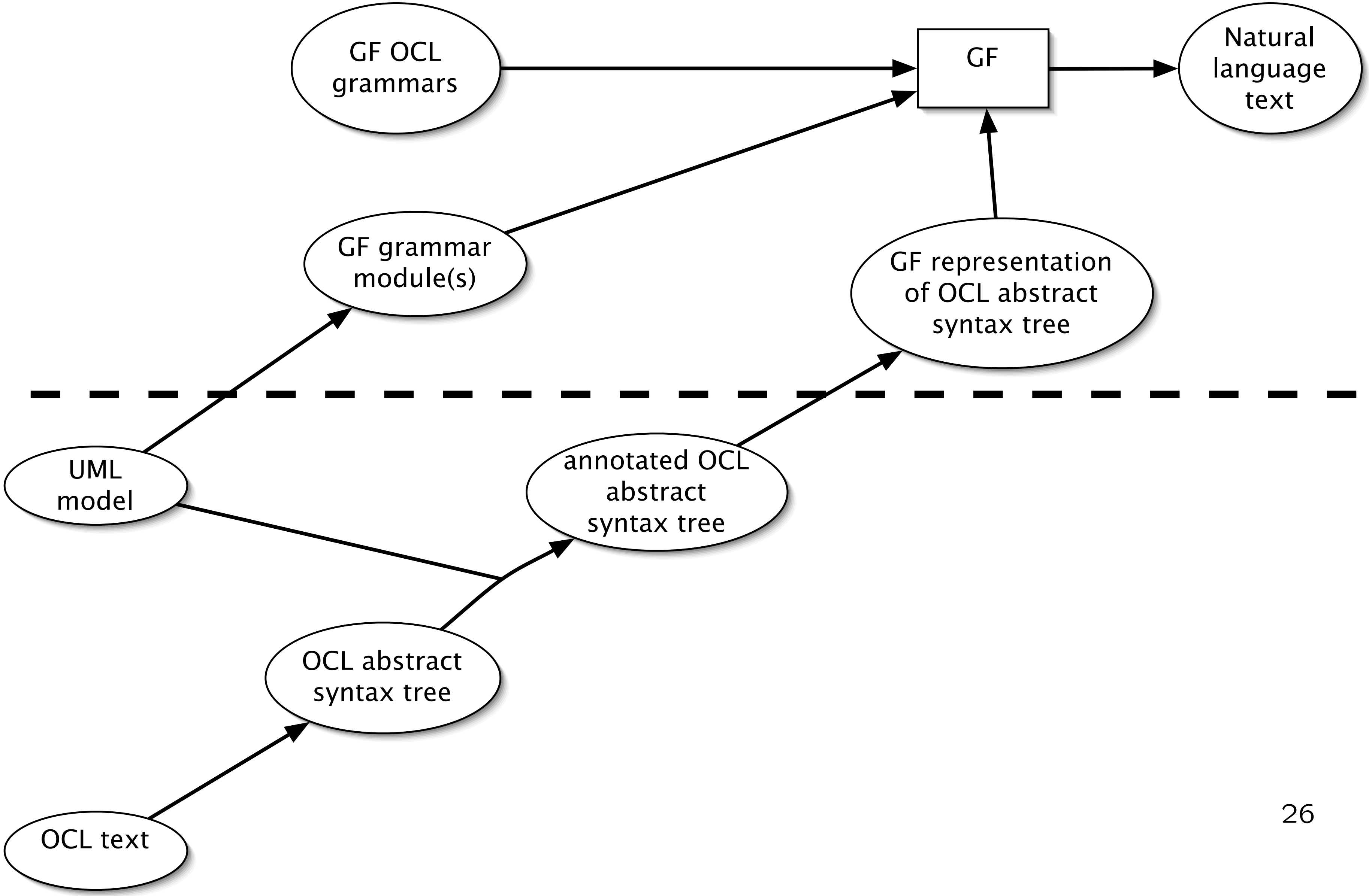

UML model
OCL text
Parser
OCL abstract syntax tree
Typechecker
annotated OCL abstract syntax tree
BNF-converter
OCL BNF grammar

III. Integration with GF

From trees to trees

We have based a parser and a typechecker on a BNF grammar for OCL. In GF we use a different grammar, with another structure (it is not only a difference of formalisms).

We need a translation from the type of trees described by the BNF grammar of OCL to the type of trees described by the GF grammar.



GF OCL grammars

GF

Natural language text

GF grammar module(s)

GF representation of OCL abstract syntax tree

UML model

annotated OCL abstract syntax tree

OCL abstract syntax tree

OCL text

Status

- work in progress
- GF OCL grammars do not have all implicit forms
  - short term: normalize
  - long term: extend grammars
- long term changes to structure of GF grammars

IV. Integration with KeY

Two separate issues

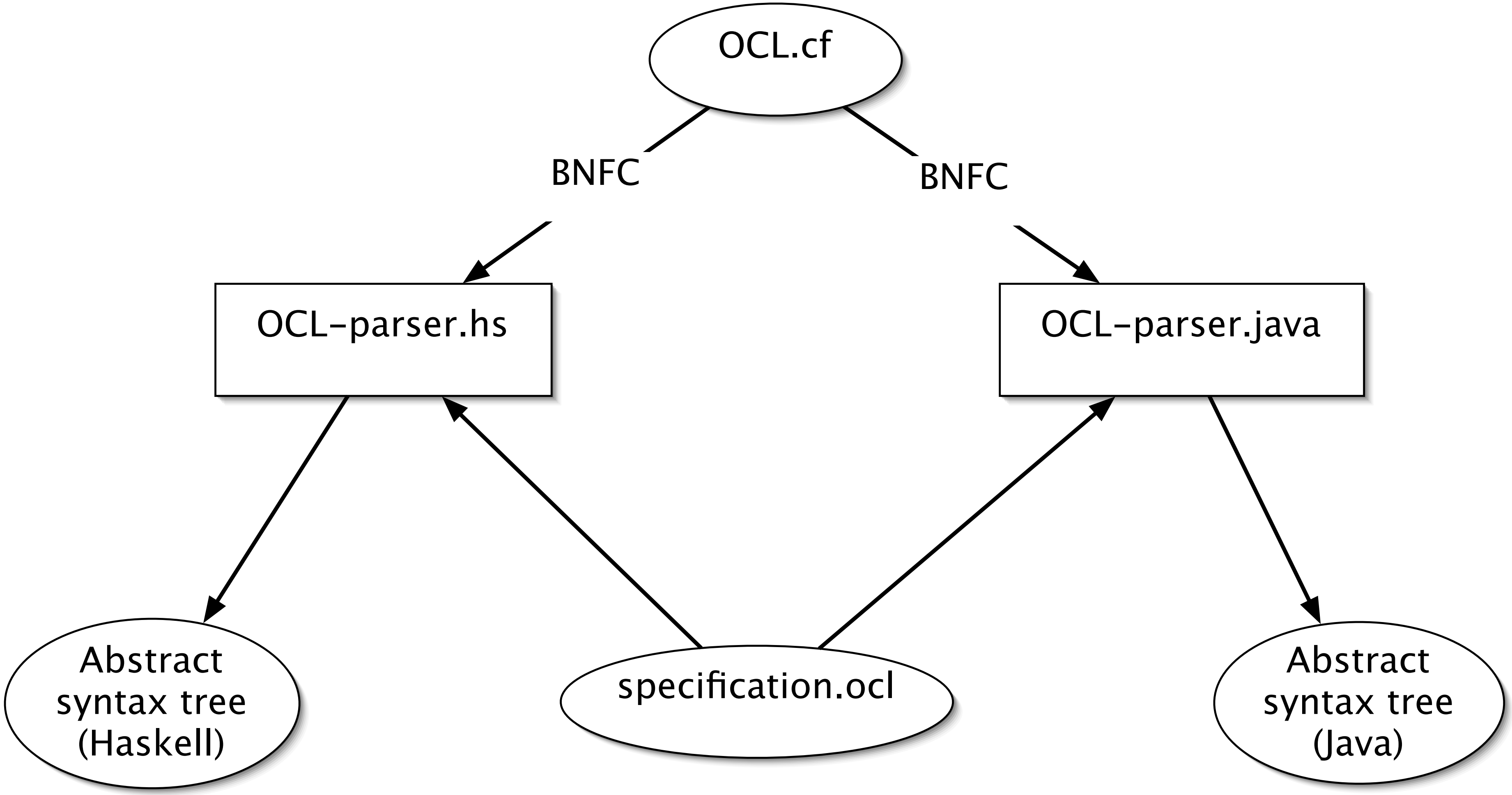
- Java-Haskell integration
  - sending text over a Unix pipe
- There is not one canonical abstract syntax for OCL. Modularity:
  - define what kind(s) of abstract syntax trees are required
  - implement interface to whatever parser is used

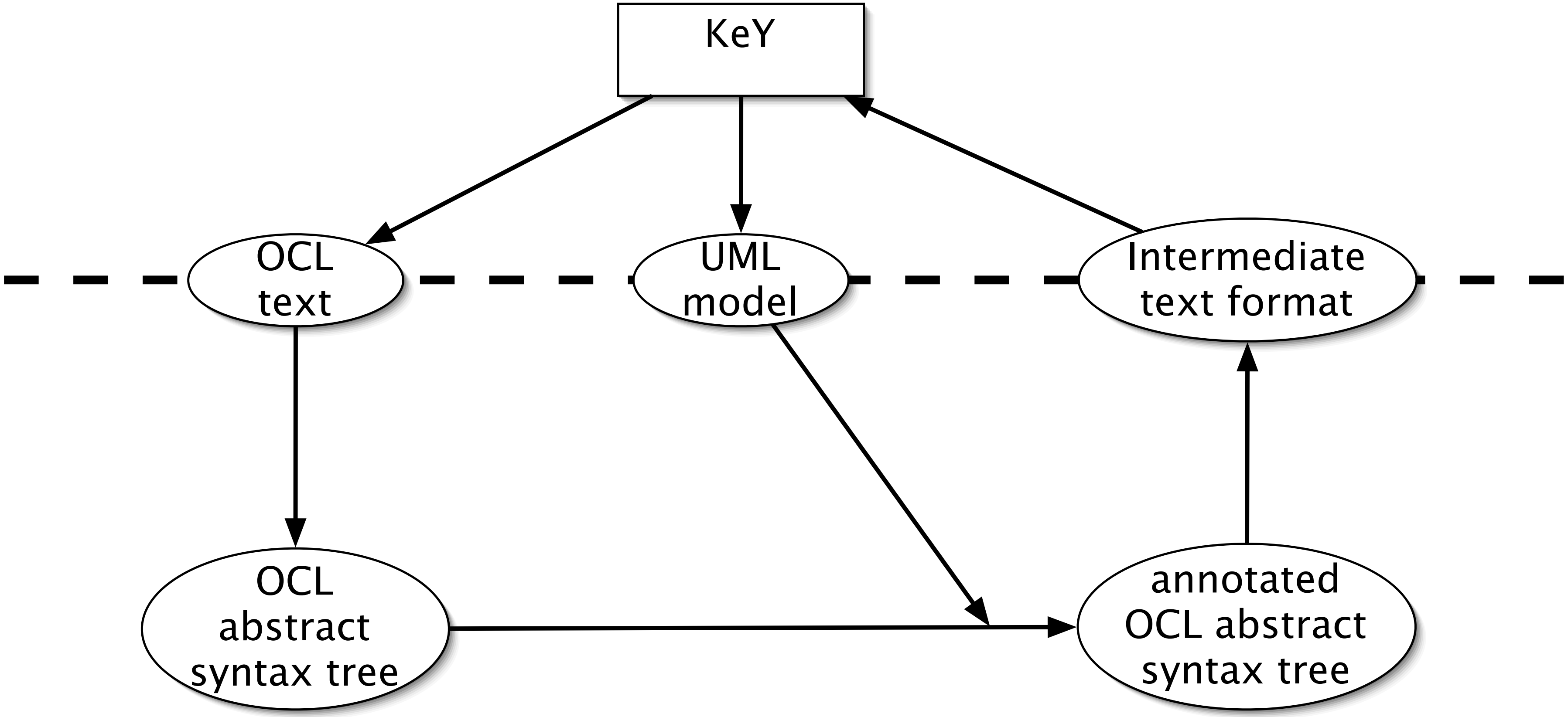
An OCL-parser in Java for free

- one grammar, generate parsers in Haskell/Java using BNFC

- assumption: one grammar (one abstract syntax) fits several purposes

- the typechecker would have to be reimplemented in Java.


OCL.cf
BNFC
BNFC
OCL-parser.hs
OCL-parser.java
Abstract syntax tree (Haskell)
specification.ocl
Abstract syntax tree (Java)



KeY

OCL text

UML model

Intermediate text format

OCL abstract syntax tree

annotated OCL abstract syntax tree

Status

- extraction of model information: works but requires some modifications by hand to the resulting file

- sending abstract syntax trees to KeY:
  - discussions with Martin and Daniel, simple experiments together with Daniel

Use “taclet OCL syntax” as OCL interchange format?

Background:

- partial evaluation of OCL will be based on the taclet mechanism
- then the taclet parser must handle OCL
- avoid problems of combining taclet/OCL-parser by inventing some simple format of “abstract OCL syntax” to be used in taclet descriptions

Avoiding the definition of too many interfaces: the Haskell parser / typechecker can then output to this format

V. Conclusion

- OCL typechecker

- status of integration in GF and KeY

- Future work: case study on rendering OCL specifications of Java-Card API [Larsson, Mostowski] in English.