Basic Laws of Object Modeling

Rohit Gheyi

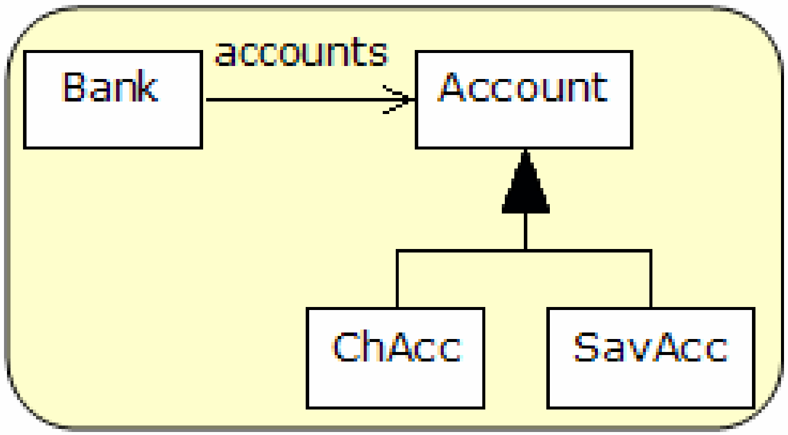
Orientador: Paulo Borba

Introduction

- Identify, formalize and study modeling laws
- Each law defines two transformations
- Focus: modeling laws for Alloy
- Modeling Laws
  - have not been sufficiently studied yet
  - might bring similar benefits of programming laws
    - Reliability
    - Productivity

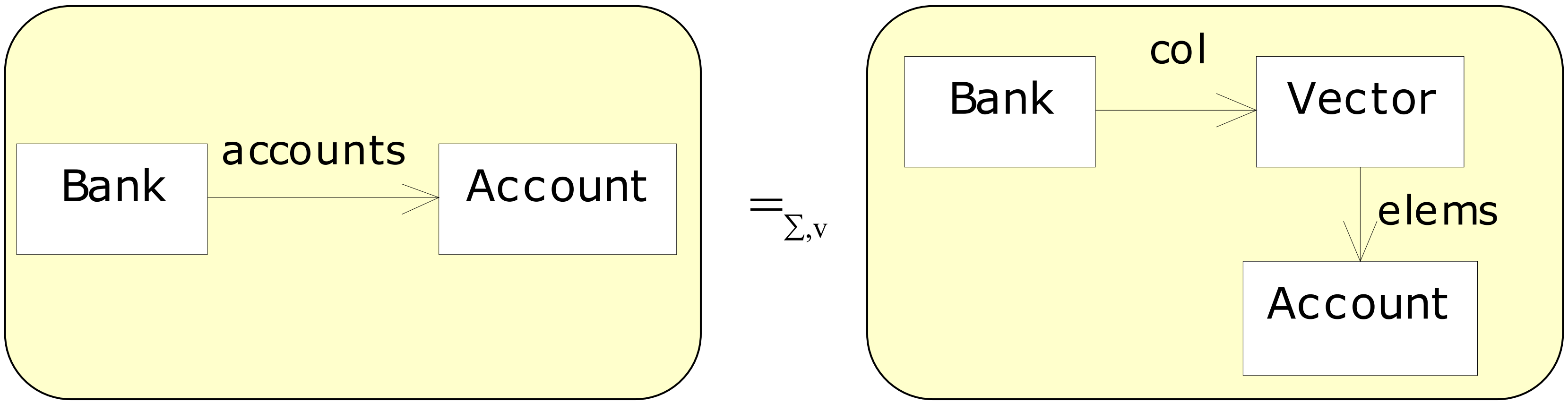
earlier stages of the software development process

Example of an Alloy Specification




```
sig Bank {
    accounts: set Account
}
sig Account {}
sig ChAcc extends Account {}
sig SavAcc extends Account {}
fact AccPartition {
    Account = ChAcc + SavAcc
    no (ChAcc & SavAcc)
}
```

Equivalence Notion



Σ = {Bank, accounts, Account}

v = {(accounts $\rightarrow$ col.elems)}

Basic Laws

- We proposed basic laws defining properties about:
  - signatures
  - facts
  - relations
- Predicate calculus and relational operator properties are also valid

Introduce Empty Signature Law

Set of paragraphs

ps	$=_{\Sigma,v}$	*ps* **sig** *S* {}

provided

(→) *ps* does not declare any paragraph named *S*; *S* is not in Σ;

(←) *S* does not appear in *ps*; if *S* is in Σ then *v* contains an item for *S*.

Introduce Formula Law

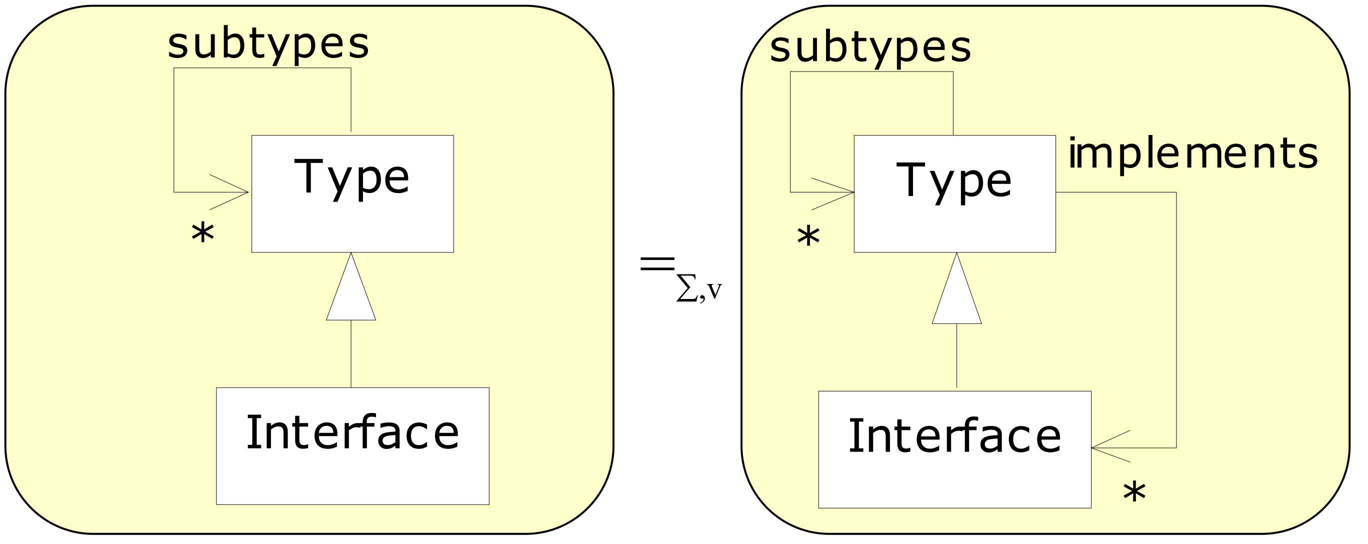
ps
fact *F* {
        forms
}
	$=_{\Sigma,\mathrm{v}}$	*ps*
		fact *F* {
		        forms
		        f
		}

Set of formulas

provided

(↔) The formula *f* can be deduced from the formulas in *ps* and *forms*.

Example: Introduce Relation



Σ = {Type, subtypes, implements, Interface}

v = {(implements $\rightarrow$

(~subtypes & (Type->Interface)))}

Introduce Relation Law

ps
sig *S* {
    rs
}
fact *F* {
    forms
}
	$=_{\Sigma,v}$	*ps*
		sig *S* {
		    rs,
		    r : **set** *T*
		}
		fact *F* {
		    forms
		    r = *exp*
		}

parent signature

(↔) if *r* belongs to ∑ then *v* contains the *r*→*exp* item;

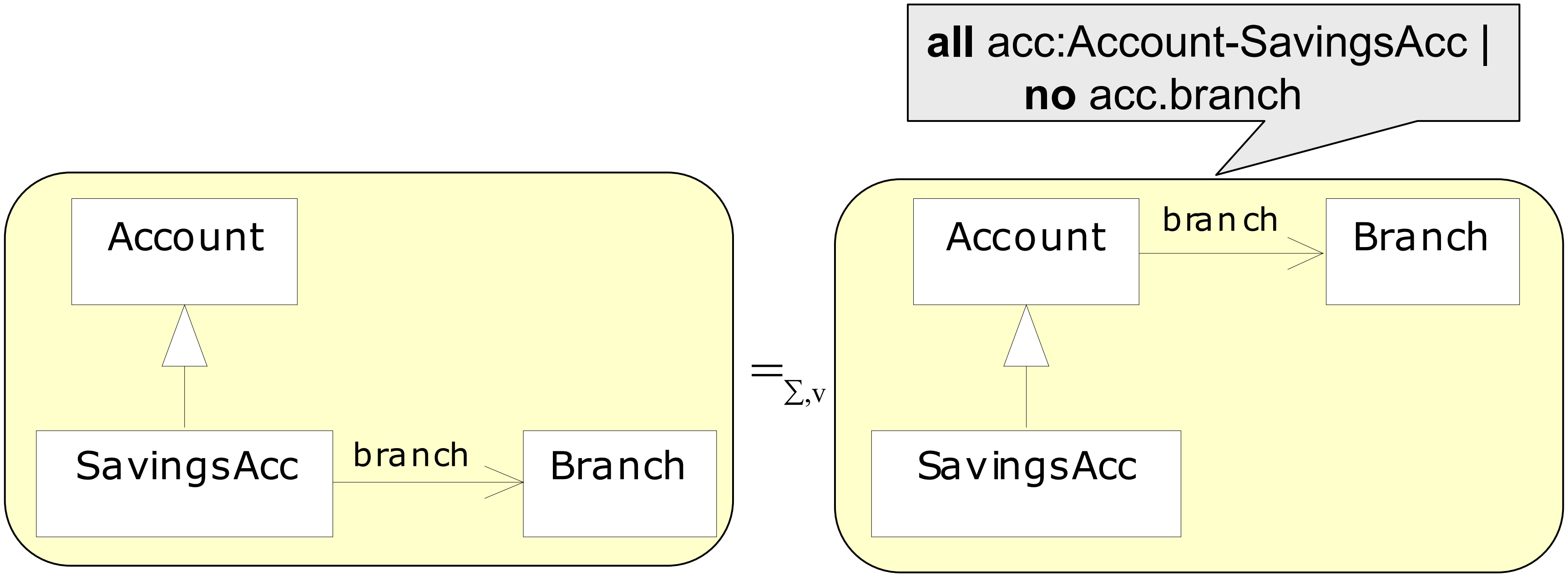
(→) The family of *S* does not declare any relation named *r* in *ps*;

(←) *r* does not appear in *ps* and *forms*.

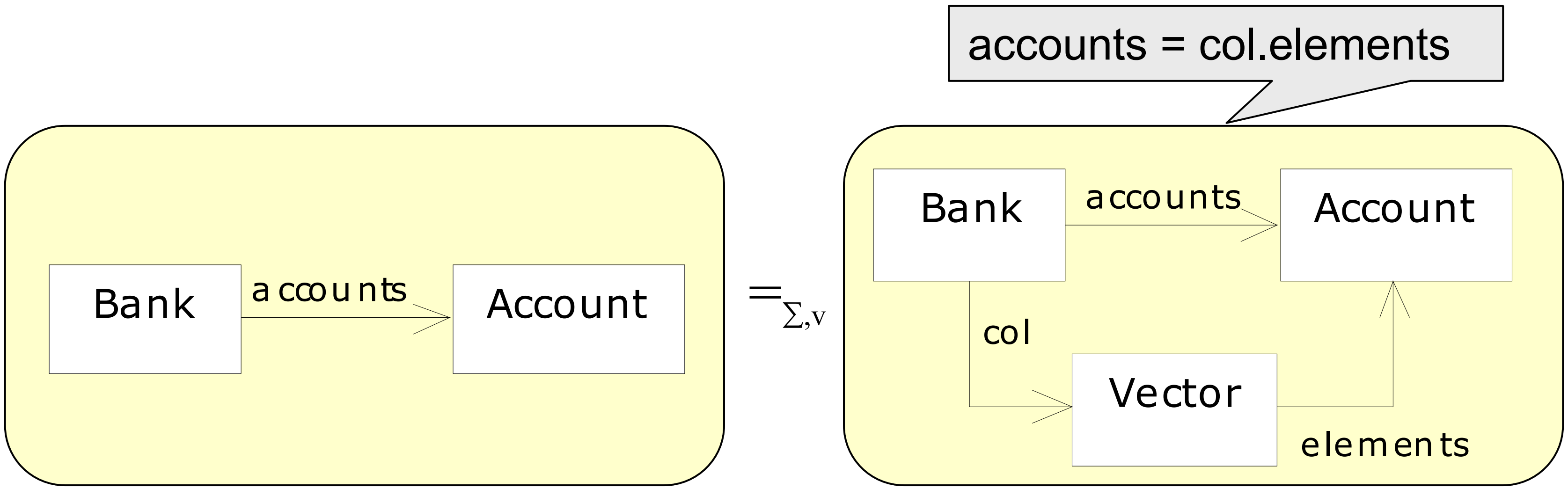
Change Relation Qualifier Type: from set to scalar Law

ps
sig *S* {
    rs,
    r : **set** *T*
}
fact *F* {
    forms
    all *s*: *S* | **one** *s.r*
}
	$=_{\Sigma,v}$	*ps*
		sig *S* {
		    rs,
		    r : **scalar** *T*
		}
		fact *F* {
		    forms
		}

Example: Pull Up Relation



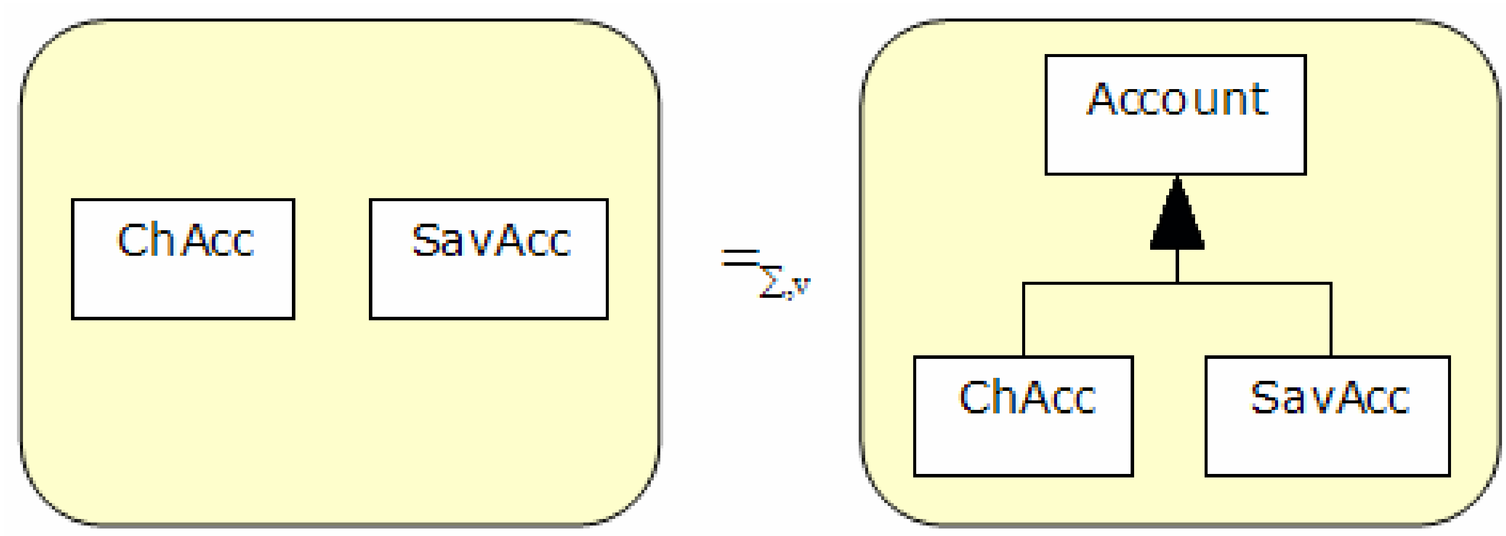
Example: Split Relation



Σ = {Bank, accounts, Account}

v = {(accounts $\rightarrow$ col.elements)}

Example: Introduce a Generalization



Σ = {ChAcc, SavAcc}

v = {(Account $\rightarrow$ ChAcc+SavAcc)}

Alloy Semantics

- The basic laws are not complete yet
- We proved the soundness of the laws using an Alloy’s denotational semantics
- We define part of Alloy’s denotational semantics using Alloy itself
- All the possible assignments of values to the signature names and field names that satisfy the specification’s constraints

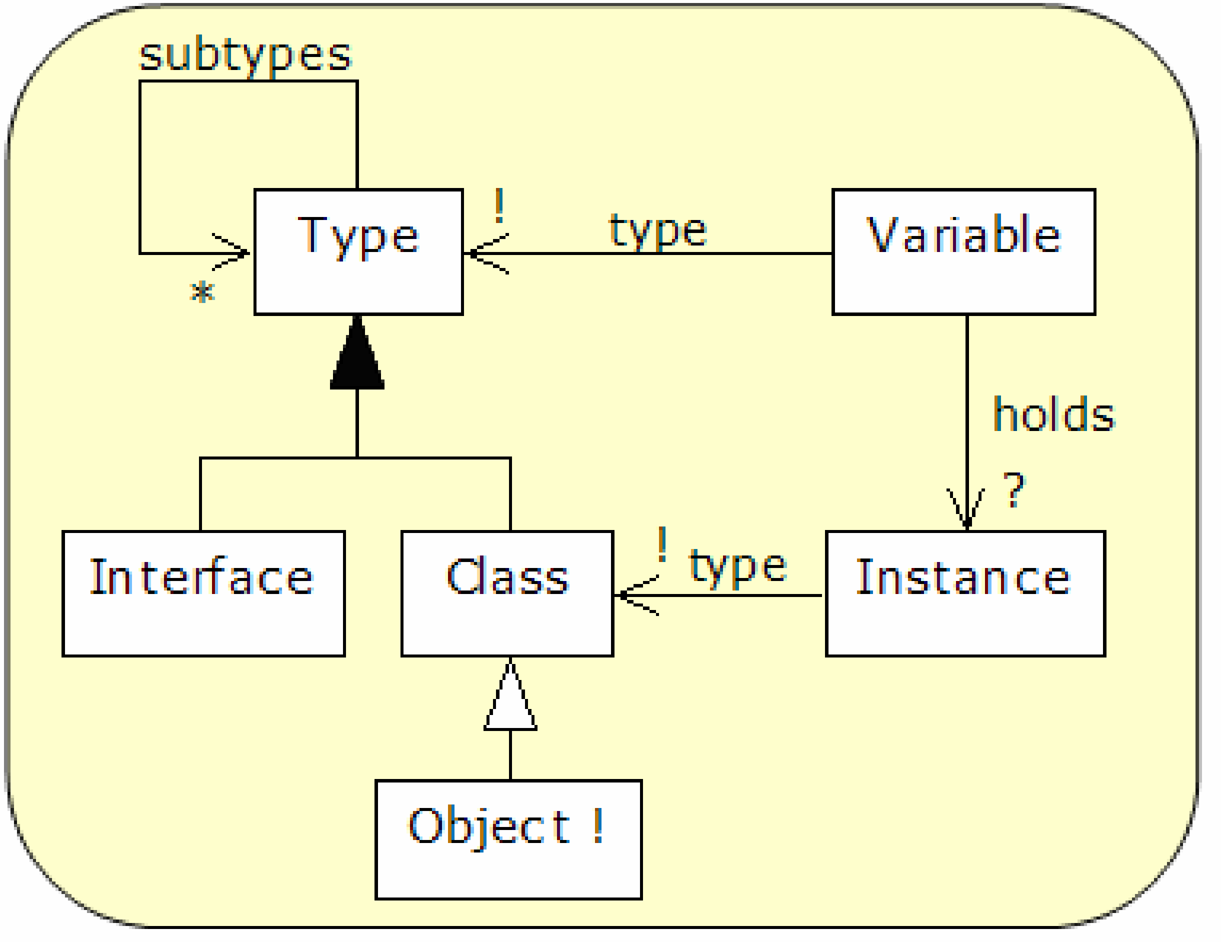
Abstract Syntax

```
sig Signature {
    isStatic: option STATIC,
    name: SignatureName,
    extensions: set Signature,
    relations: set Relation,
    type: Type
} …
fact StaticSemantics {
    all m: Module | all disj s1,s2: m.signatures |
        s1.name != s2.name …
}
```

Semantics Function

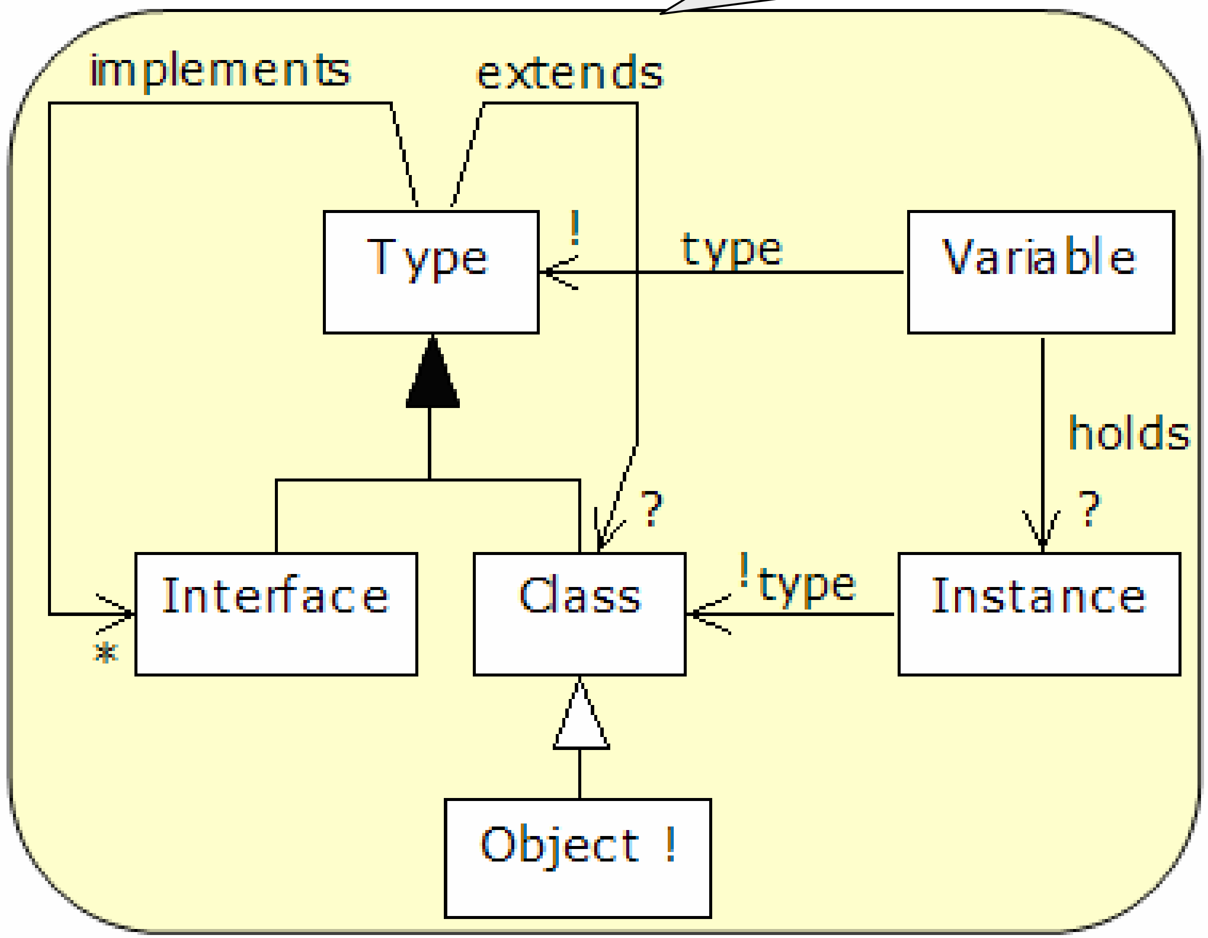
```
det fun Module::semantics(i: Interpretation) {
    this..modNames() in i..interNames()
    this..satisfyImplicitFacts(i.map)
    all f: this..modFormulae() |
        satisfyFormula(f, i.map)
}
```

Java Types Model



Java Types Refactored

subtypes = ~extends + ~implements



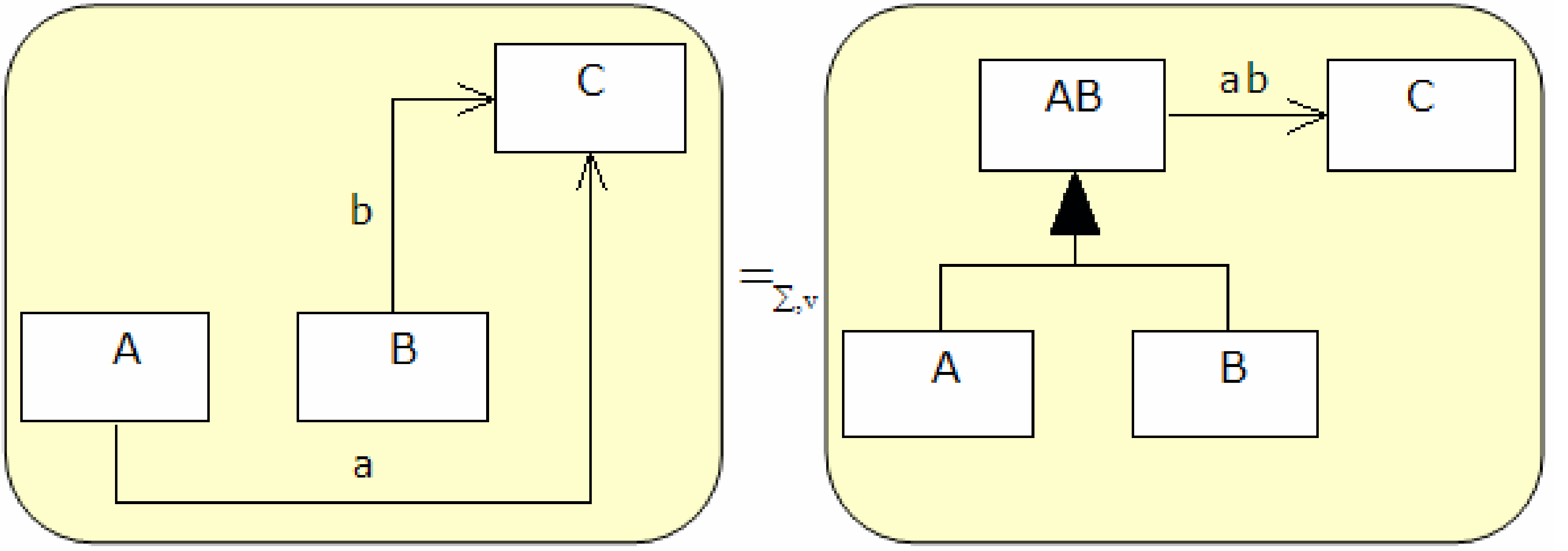
Steps

$\sum$ = names + {subtypes}

v = {(subtypes → ~extends+~implements)}

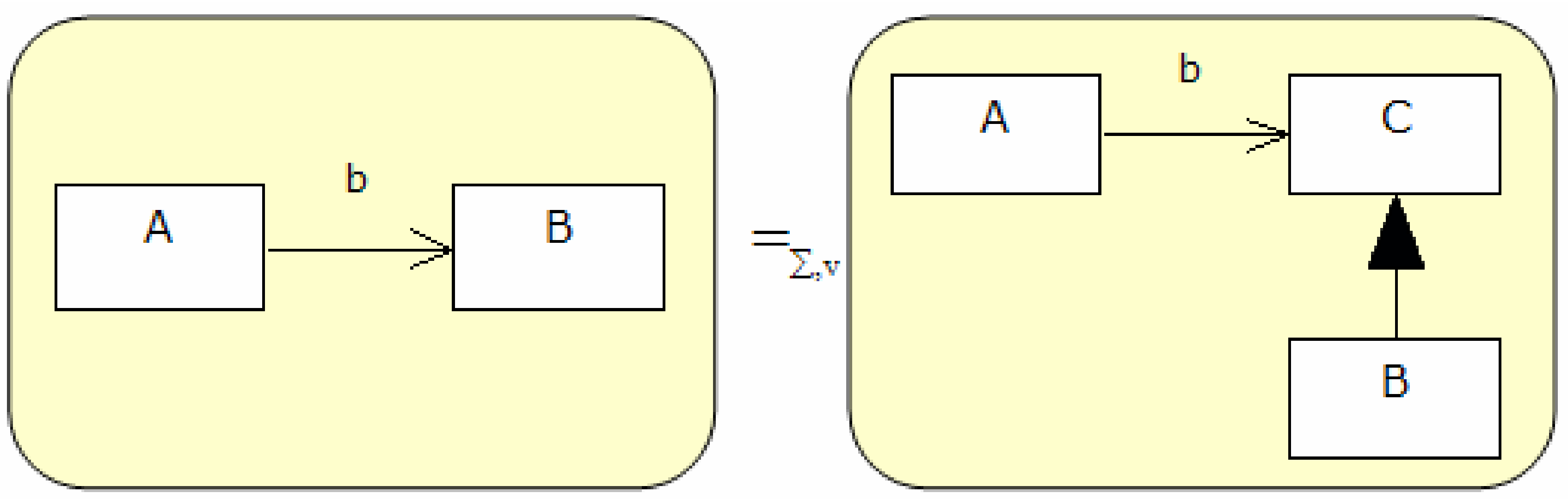
- Introduce the extends and implements relations in Type
  - implements = (~subtypes & (Type->Interface))
  - extends = (~subtypes & (Type->Class))
- From these definitions deduce:
  - subtypes = (~extends + ~implements)
- Replace all formulas containing subtypes by its definition
- Remove subtypes and its definition

Introduce a Generalization Refactoring



- Architects x Designers
- Most of the basic laws are very simple to apply since their pre-conditions are simple syntactic conditions

Interpolating an Interface Refactoring

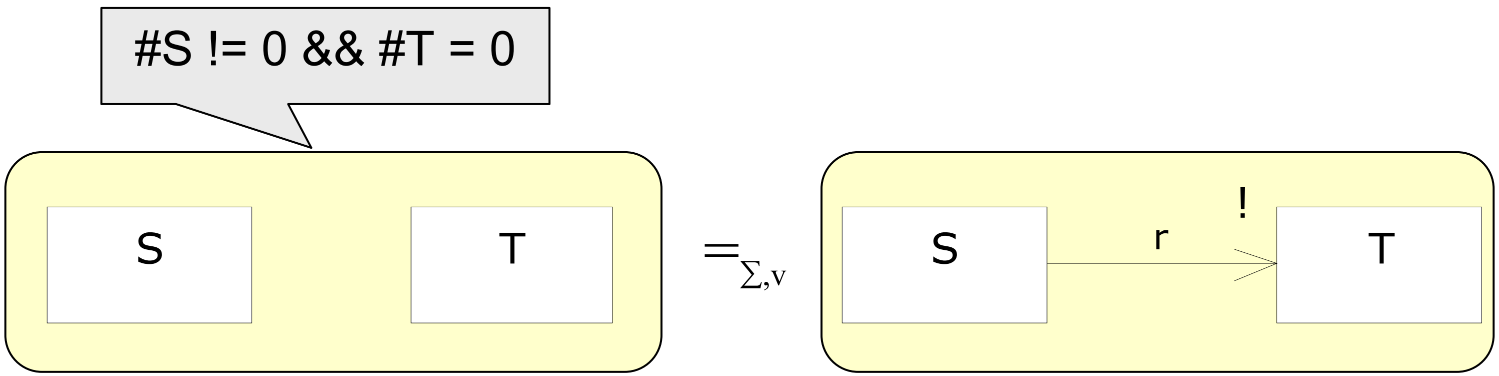


($\leftrightarrow$) if *C* belongs to $\sum$ then *v* contains the *C*$\rightarrow$*B* item;

($\rightarrow$) *ps* does not declare any paragraph named *C;*

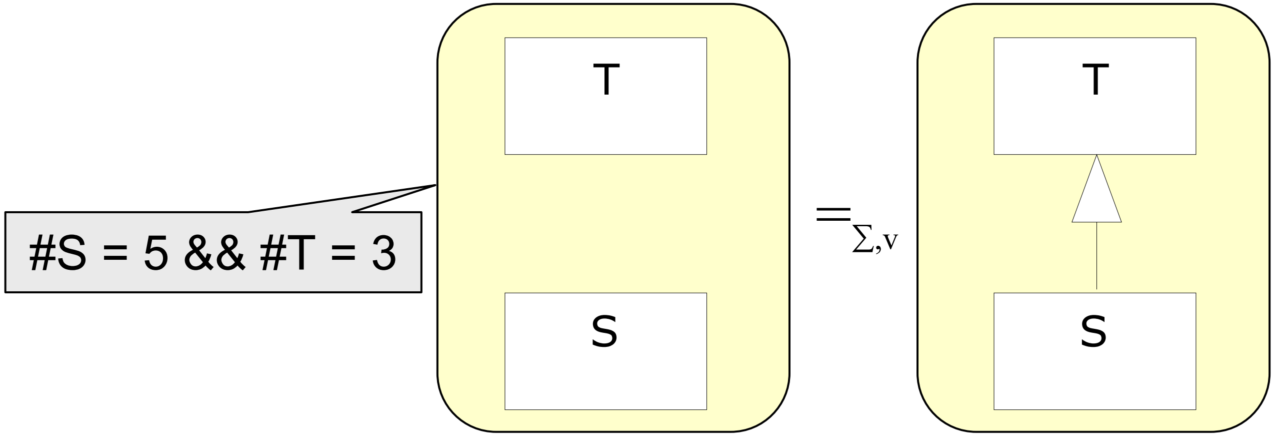
($\leftarrow$) *C* does not appear in *ps*, *rs*, *rs*' and *forms*.

Inserting a Relation



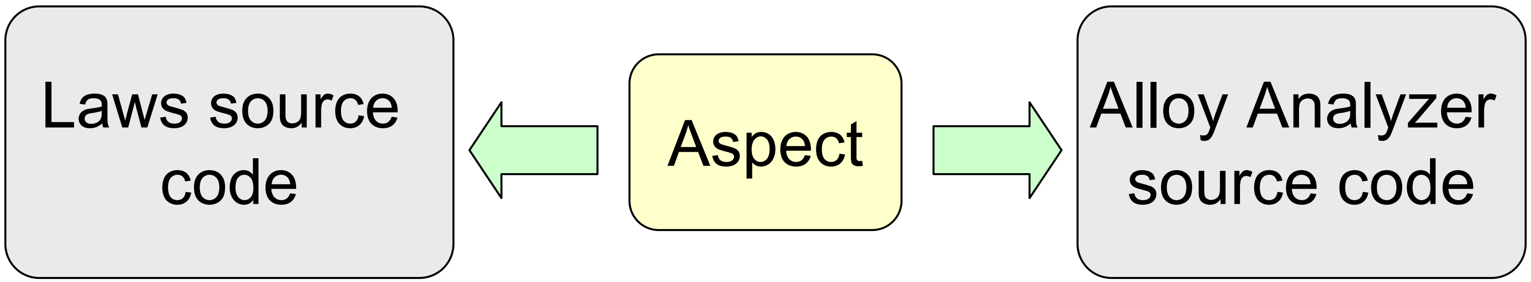
- Constraint
  - **all** s: S | **one** s.r
- Formula deduced
  - **(#**S **!=** 0) **=>** (**#**T **>** 0)

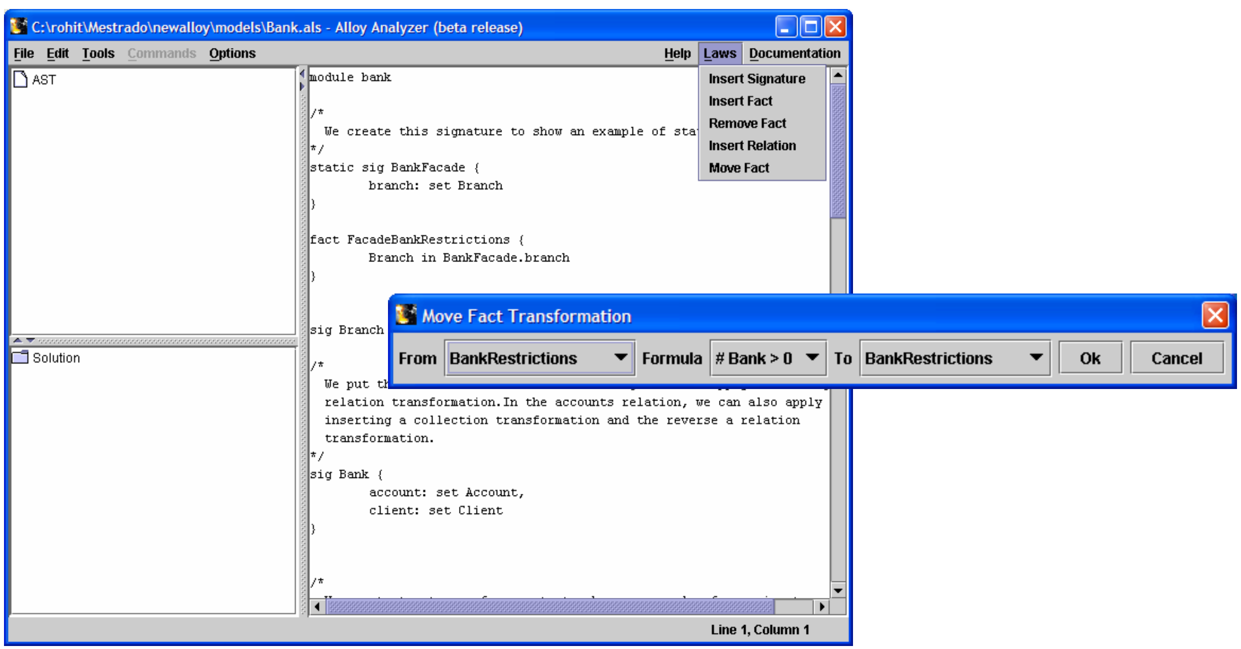
Extending a Signature



- Constraint
  - S **in** T
- Formula deduced
  - (S **in** T) **=>** (**#**S <= **#**T)

Tool Extension





Conclusions

Importance of Modeling Laws

Axiomatic semantics

Refactor specifications

Derive refactorings

Related Work

- Transformations for UML models
- Laws for ROOL
- Refinement laws for UML-RT
- Alphabetised relational calculus (ARC)

Future Work

- Complete Alloy denotational semantics
- Propose a transformational semantics for Alloy
- Define other equivalence notions
- Propose more basic laws and refactorings
- Show that our set of laws is complete (propose a normal form for Alloy)

Future Work

- Use PVS to assure that our laws are sound
- Apply the basic laws to realistic case studies
- Extend the tool with the laws
- Model Refactoring X Code Refactoring

Basic Laws of Object Modeling

Rohit Gheyi

Orientador: Paulo Borba

Backup Slides

Pull Up Relation Law

ps
sig *T* {
    rs
}
sig *S* **extends** *T* {
    rs',
    r : **set** *U*
}
fact *F* {
    forms
}
	$=_{\Sigma,v}$	*ps*
		sig *T* {
		    rs,
		    r : **set** *U*
		}
		sig *S* **extends** *T* {
		    rs'
		}
		fact *F* {
		    forms
		    all *t*: *T-S* | **no** *t.r*
		}

Move Formula Law

ps
fact *F* {
    forms
    f
}
fact *G* {
    forms'
}
	$=_{\Sigma,v}$	*ps*
		fact *F* {
		    forms
		}
		fact *G* {
		    forms'
		    f
		}

Split Relation Law

ps
sig *S* {
    rs,
    r : **set** *T*
}
fact *F* {
    forms
}
	$=_{\Sigma,v}$	*ps*
		sig *S* {
		    rs,
		    r : **set** *T*,
		    u : **set** *U*
		}
		fact *F* {
		    forms && *r* = *u.t*
		}
		sig *U* { *t* : **set** *T* }

($\leftrightarrow$) if *r* belongs to Σ then *v* contains the *r*$\rightarrow$*u.t* item; ...

($\rightarrow$) *ps* does not declare any paragraph named *U;* the family of *S* does not declare any relation named *u* in *ps*;

($\leftarrow$) *U*, *t* and *u* do not appear in *ps*, *rs* and *forms*.

Introduce a Generalization Law

ps
sig *S* { *rs* }
sig *T* { *rs'* }
fact *F* {
    forms
}
	$=_{\Sigma,v}$	*ps*
		sig *U* {}
		sig *S* **extends** *U* { *rs* }
		sig *T* **extends** *U* { *rs'* }
		fact *F* {
		    forms
		    no *S&T* && *U=S+T*
		}

(↔) if *U* belongs to ∑ then *v* contains the *U→(S+T)* item;

(→) *ps* does not declare any paragraph named *U* and the family of *S* and *T* does not declare any relation with the same name in *ps*

(←) *U*, *S* and *T* do not appear in *ps*, *rs*, *rs'* and *forms*.

Module and Fact

```
sig Module {
    signatures: set Signature,
    facts': set Fact
}
sig Fact {
    formulae: set Formula
}
```

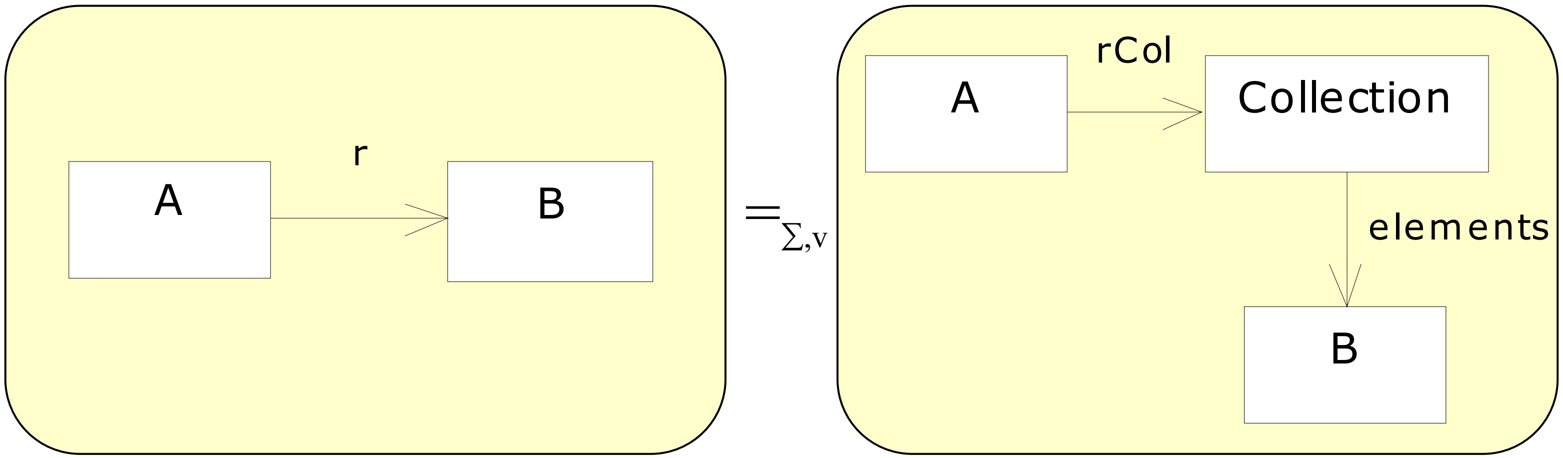
Relations and Names

```
sig Relation {
    name: RelationName,
    type: Seq[Type],
    qualifier: Qualifier
}
sig Name {}
part sig SignatureName, RelationName
                                extends Name {}
```

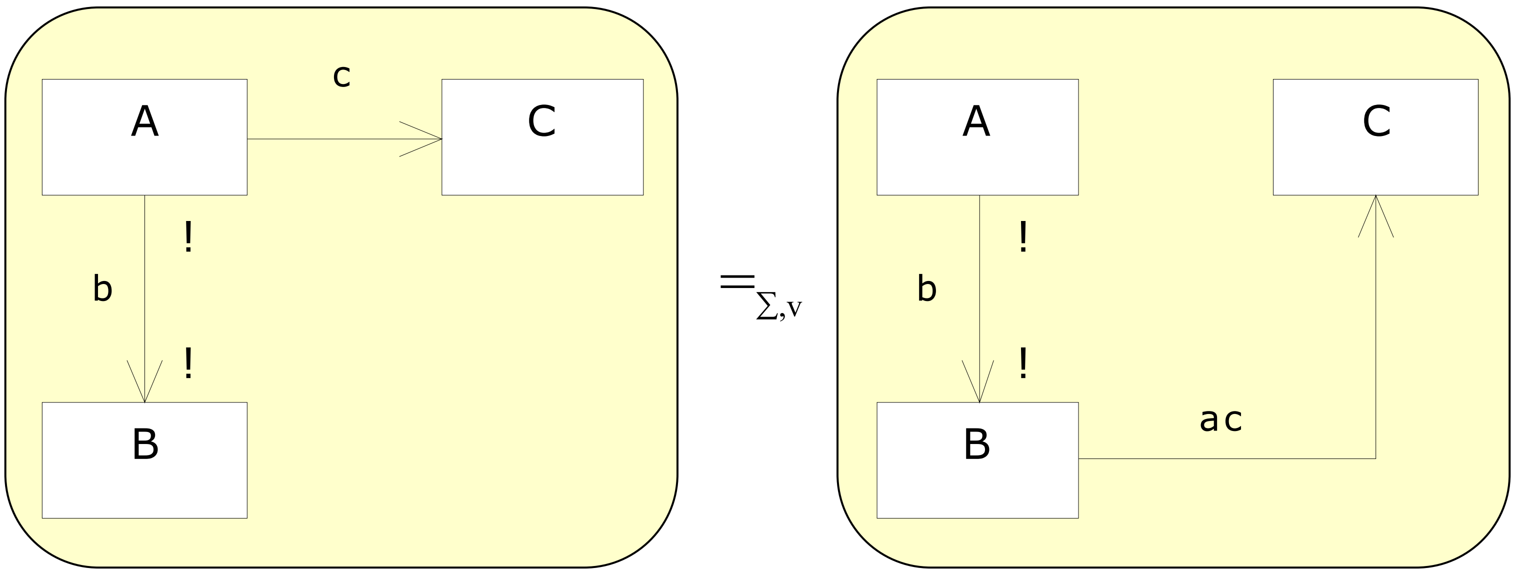
Explicit Constraints

```
det fun Module::satisfyFormula(f: Formula,
                               map: Name->Seq[Atom]) {
    some f && (isComparison(f) =>
            this..satisfyComparison(f, map))
    ...
}
```

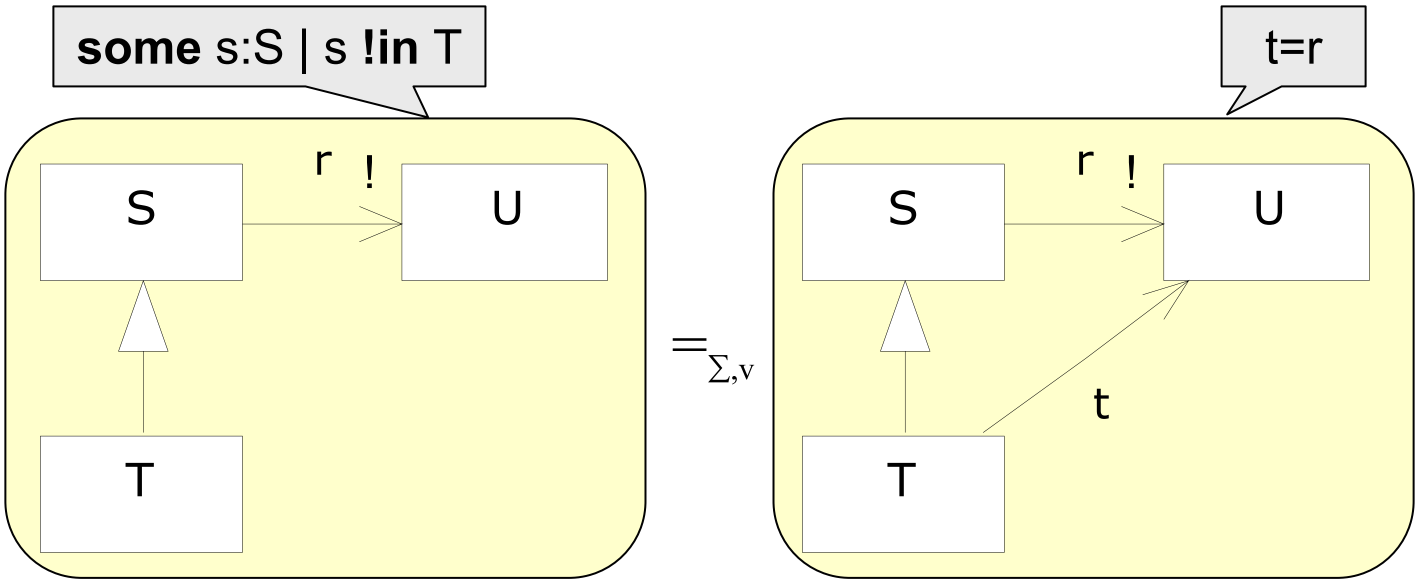
Inserting a Collection Refactoring



Moving a Relation Refactoring



Inserting a Relation in a Subsignature



- Suppose s' be an element that does not belong to T and relates with an element u' of U in r
- After including t=r, we have
  - (((s'->u') in r) && (t = r)) => ((s'->u') in t)

Guidelines for Transformation Systems