Type checking by domain analysis in Ampersand

Stef M. M. Joosten and Sebastiaan J. C. Joosten
RAMiCS 2015, Braga

Why Ampersand?

& as a paradigm

Ampersand helps Businesses control its operations, by formalising the rules of the Business.

A system designed or built with Ampersand helps its users maintain a set of rules.

& as a language

Ampersand-Rules are expressed in RA.

RA presented is almost-heterogeneous.

Ampersand compiler uses heterogeneous RA internally.

& as a database

To prototype systems, database-applications are generated.

The population in the database is always a model to business-rules that are “invariant”.

Specifying business-applications in RA

Model theory	Ampersand	Business
Sentence	rule	Business rule / requirement
Language	concepts + relations	Domain language (NL)
Model	data / population (changes in time!)	Administrative truth
Theory	concepts + relations + rules	Knowledge model
	concepts + relations + rules + interfaces = information system	Business process support system

Why typed relations?

In business, we keep persons and cars separate.

So, from a business point of view things (atoms) must be instance of a concept.

& says: every relation has a signature

RELATION r[A*B]

e.g.

RELATION owns[Person*Car]

Why type checking?

Heterogeneous relation algebra is great, but…

Express things like:

- ‘Every Employee is a Person’
- ‘Every Student is a Person’
- ‘Teaching-Assistant are those Students which are Employees’

Example rules:

- Employees receive their respective salary at the 25th of the month
- Only employees who are not students can give grades

Language presented to the Ampersand user

Heterogeneous algebra

- Every relation r :: A*B has a signature
  (provided by Ampersand user)
- Unary symbols can be typed
- Every term is typable
  (compiler provides a signature)
- For every operation, ; ∩ ∪ −
  type restrictions apply
  (compiler guards these restrictions)

Homogeneous algebra

- Objects (&: Concepts) are sorted
- For the composition, s;t
  the target of s, and source of t,
  need not match
- Every type C∩D arising at
  a composition s;t
  with s :: A*C and t :: D*B,
  has a specific name

How to use domain analysis

- Use the rules to specify the ordering on concepts.

$I[Employee] \cap I[Student] = I[TeachingAssistant]$

$I[Employee] \cap I[Person] = I[Employee]$

A rule has a left hand side, and a right hand side.

$$\overbrace{\mathrm{I}[\mathrm{Employee}] \cap \mathrm{I}[\mathrm{Student}]}^{\text{left hand side}} = \overbrace{\mathrm{I}[\mathrm{TeachingAssistant}]}^{\text{right hand side}}$$

Every term is typable, we get domain knowledge:

$$\underbrace{\mathrm{I}[\mathrm{Employee}] \cap \mathrm{I}[\mathrm{Student}]}_{\substack{\text{two types:} \\ \text{Employee*Employee} \\ \text{Student*Student}}} = \underbrace{\mathrm{I}[\mathrm{TeachingAssistant}]}_{\text{total relation: defines “Teaching Assistant”}}$$

Type checking by domain analysis

Analyse Terms

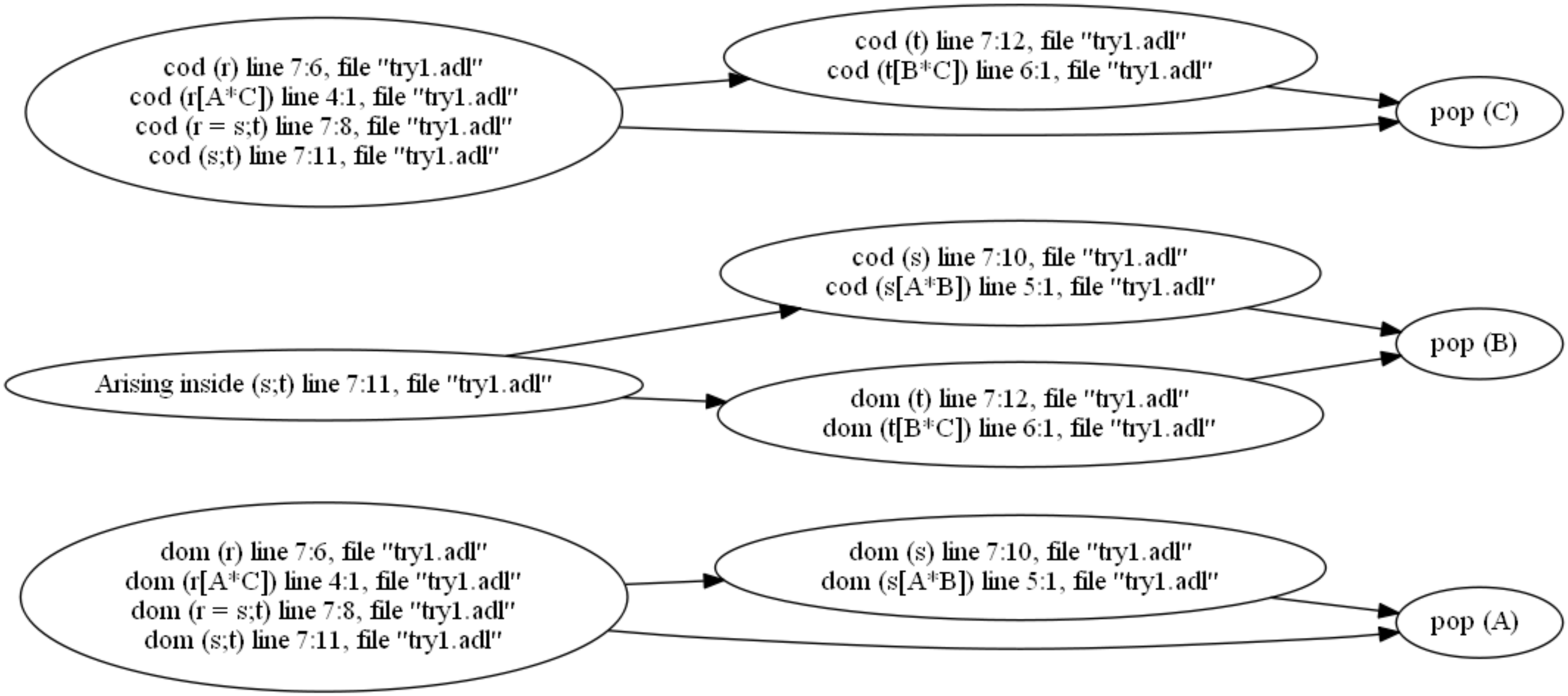
Order TypeTerms

Check TypeTerms

Type checking by domain analysis // Example script

r[A*C], s[A*B], t[B*C]

r = s;t

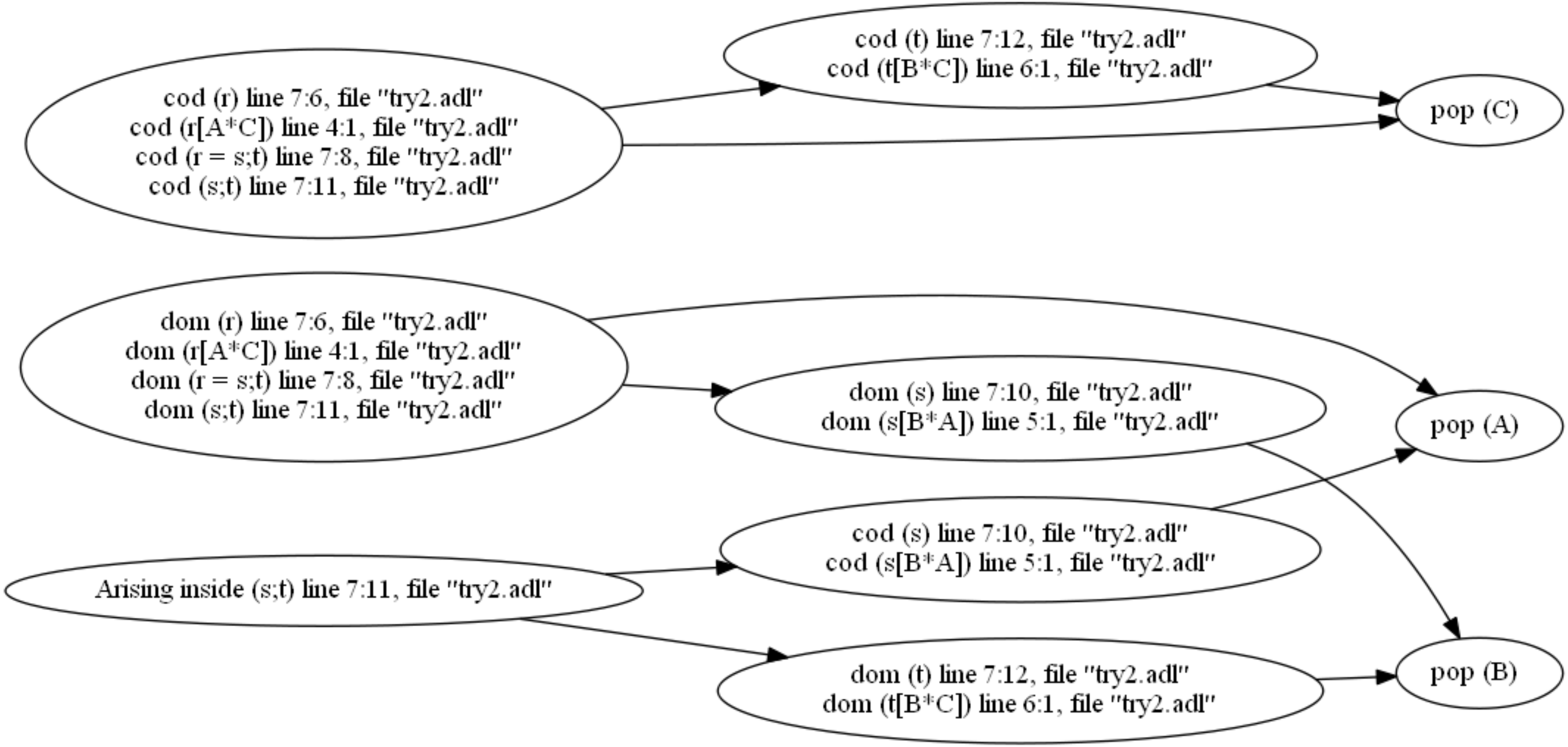


Type checking by domain analysis

Type checking by domain analysis // Example script 2

r[A*C], s[B*A], t[B*C]

r = s;t



Type checking by domain analysis

Algorithm: Type checking by domain analysis

Create a TypeTerm for every Term

Relate all TypeTerms using ‘sub’

Find equivalence classes,
calculate the closure of sub

Find the least concept for each TypeTerm

Every TypeTerm should have a unique least concept

Algorithm: Type checking by domain analysis

Create a TypeTerm for every Term

Relate all TypeTerms using ‘sub’

Find equivalence classes,
calculate the closure of sub

Find the least concepts for each TypeTerm

Every TypeTerm should have a unique least concept

Term	TypeTerm
r;s, r	dom(r;s), dom(r) cod(r;s), cod(r)
Typed Identity element I[A]	pop(A)
Compose: r;s	inter(r,s)

Algorithm: Type checking by domain analysis

Create a TypeTerm for every Term

Relate all TypeTerms using ‘sub’

Find equivalence classes,
calculate the closure of sub

Find the least concepts for each TypeTerm

Every TypeTerm should have a unique least concept

TypeTerm	sub
dom(r;s)	dom(r;s) \`sub\` dom(r)
dom(r), r[A*B]	dom(r) \`sub\` pop(A)
dom(x)	cod(x) \`sub\` dom($x^{\smile}$) dom($x^{\smile}$) \`sub\` cod(x) !
x = y	dom(x) \`sub\` dom(y) cod(x) \`sub\` cod(y) dom(y) \`sub\` dom(y) cod(y) \`sub\` cod(y)

Algorithm: Type checking by domain analysis

Create a TypeTerm for every Term

Relate all TypeTerms using ‘sub’

Find equivalence classes,
calculate the closure of sub

Find the least concepts for each TypeTerm

Every TypeTerm should have a unique least concept

classes: $sub^* \cap I$

pretype(s) of each typeterm:

pretype = $(sub^*)\ ;\ pop^{\smile}$

Algorithm: Type checking by domain analysis

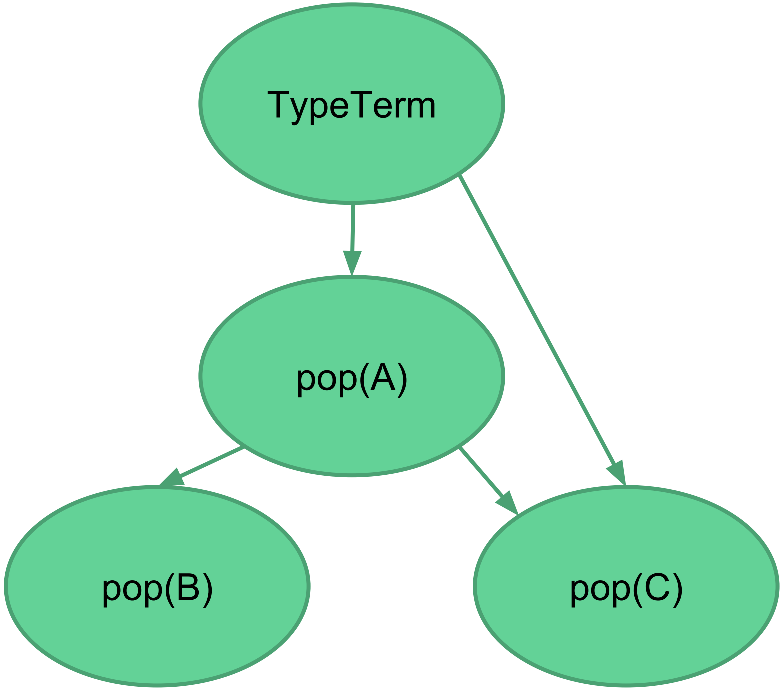
Create a TypeTerm for every Term

Relate all TypeTerms using ‘sub’

Find equivalence classes,
calculate the closure of sub

Find the least concepts for each TypeTerm

Every TypeTerm should have a unique least concept



Algorithm: Type checking by domain analysis

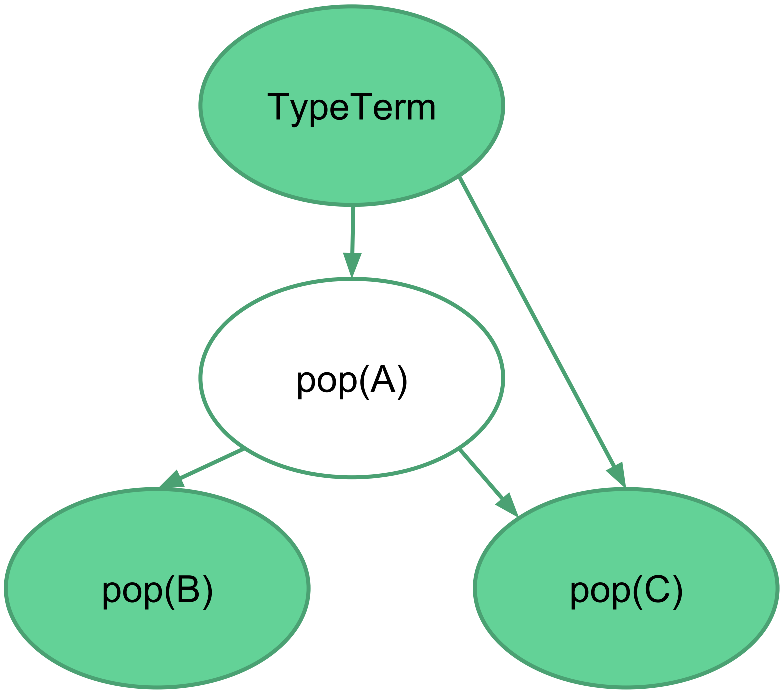
Create a TypeTerm for every Term

Relate all TypeTerms using ‘sub’

Find equivalence classes,
calculate the closure of sub

Find the least conepts for each TypeTerm

Every TypeTerm should have a unique least concept



Experimental results

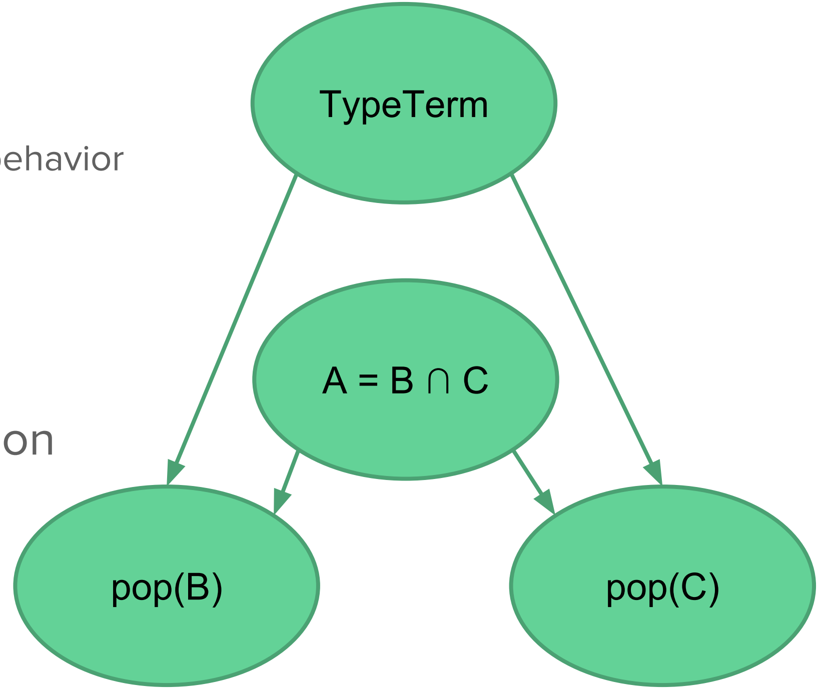
Use graphs as intuitive feedback

Reason with the entire script at once

No need to handle ‘type declaration’ separately: $r \subseteq 1[A*B]$

Type checking by Domain analysis

- Reasoning about the entire script
  - Bad scalability
  - Composing scripts may lead to unpredictable behavior
  - Limitations to graphical feedback
- Graphs as feedback
  - Can not explain why a line is missing
  - Extra maintenance burden
- No separate way to handle type information
  - Type errors become ‘correct’ inferences
  - Equal types – I[A] = I[B] – become type errors



Conclusion

Ampersand needs type checking

Type checking can be done through domain analysis

Currently, a different algorithm is used

The type graphs are visually attractive, so may be useful for some other application.