

TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

Introduction

How to get to
Campus
Thalès
Social event

TYPES 2004 is the 2004 Workshop of the [Types project](#). The current financial support for this project is the Coordination Action "Types" in the IST program of the European Union starting in September 2004. This action is based on the strong collaboration and achievements in four successful European projects ([ESPRIT BRA 6453](#), [ESPRIT working group 21900](#) and [IST working group 29001](#)).

Invited
speakers
Programme
List of
participants

The topic of this workshop is formal reasoning and computer programming based on Type Theory : languages and computerised tools for reasoning, and applications in several domains such as analysis of programming languages, certified software, formalisation of mathematics and mathematics education.

Call for papers

TYPES workshops occur regularly since 1992. In 2003, it was located [in Torino](#). The workshop is not restricted to members of the EU action.

Organising
committee

Talks will not be published as such, but we encourage people who gave a talk to submit their papers after the conference. Post-Workshop Proceedings for TYPES 2004 will be published, after a formal referee process, as a volume of the [Lecture Notes in Computer Science \(LNCS\)](#) series.

Related events
History



TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

Introduction

How to get to campus Thalès

How to get to
Campus
Thalès
Social event

[access map \(PDF\)](#)

**By public transportation from Charles de Gaulle airport,
Orly airport or Paris**

Invited
speakers
Programme
List of
participants

The campus where Université Thalès is located is accessible from the RER C railway stations Jouy-en-Josas (at 700 meters) or Petit Jouy (at 900 meters). On the [interactive map of Jouy-en-Josas](#), it is on the left in the middle.

From Roissy - Charles de Gaulle airport

Call for papers

The simplest way is to take the RER B line (two stations serve the airport) going to Massy-Palaiseau, then the RER C line to Jouy-en-Josas. A ticket from the airport to Jouy-en-Josas via Massy-Palaiseau costs 11.25 euros. The duration of the full trip from the airport RER stations to Université Thalès is about 100 minutes.

Organising
committee

Related events
History

From the terminals to the RER stations

From terminal 2, follow the signs and walk to the combined RER and TGV railway station located between the terminal 2C, 2D, 2E and 2F (from terminal 2A, first walk through terminal 2C and from terminal 2B, first walk through terminal 2D, or take the green shuttle numbered 3). The RER station is called Aéroport Charles de Gaulle 2.

From terminal 1, follow the signs and take the blue shuttle (numbered 2) heading to Roissypole - RER. The RER station is called Aéroport Charles de Gaulle 1.

From the airport RER stations to Massy-Palaiseau RER station

Take any train heading to Massy-Palaiseau, Orsay or Saint-Rémy lès Chevreuse. If you ever take a train to Robinson, stop in Paris center (e.g. Gare du Nord or Châtelet - Les Halles) and wait for the next train on the same platform serving Massy-Palaiseau.

From the airport, trains run at least once every quarter and it takes between 60 and 70 minutes to reach Massy-Palaiseau ([the RER timetable](#)).

From Massy-Palaiseau RER B station to Jouy-en-Josas RER C station

The RER C platforms are located at the opposite side of the RER B in Massy-Palaiseau station (you have to walk a 200m pedestrian bridge starting upstairs on the right at the head of the RER B platforms). There, take a train to Versailles Chantiers (first platform downstairs on the left) and stop at Jouy-en-Josas. In the morning rush hour and between 17:00 and 19:15, it runs every quarter at 3, 18, 33 and 48 minutes after the clock. Then it runs at 19:18, 19:48, 20:18, 20:48, 21:18, 22:18 and finally 23:18. It takes 15 minutes from Massy to Jouy.

From Jouy-en-Josas RER C station to the campus of Université Thalès

Go out of the station *avenue Jean Jaurès* on the left. Walk along the car park until the roundabout. Take the *rue de la Libération* on the right and immediately after take a small lane called *chemin du Pré Bourgeois* on the left. Follow the sign *sentier de la Bièvre*. A few meters further, there are fences, which are the ones of Campus Thalès. Take on the right and reach the river. Follow the fences and the river for a few hundred meters until finding the main entrance of the campus. If too dark on the night, you can take the long way: follow the *rue de la Libération*, then the *rue du Général de Gaulle* where the road to the campus starts at number 67 on the left of the street.

Shuttles will be set up specially for TYPES participants at some hours. More on this later.

Alternative plan via Versailles: faster in the rush hour, but more complex.

At the airport, take the RER B line and go out at Châtelet - les Halles; connect with subway line 4 southbound to Montparnasse; follow the sign *Trains Île-de-France* until the SNCF railway station; take any suburban train that stops at Versailles-Chantiers; in Versailles-Chantiers, go to platform J or H and take the RER C that goes to the station Petit Jouy - les Loges which is 700 meters far from Université Thalès. If you don't lose your way and get the right connections, you can reach Université Thalès in 80-90 minutes.

Alternative plan by TGV

There are a few direct TGV from the combined TGV - RER B station Aéroport Charles de Gaulle 2 (the one at terminal 2) to Massy-Palaiseau combined TGV - RER B - RER C station. They leave at 13:37, 13:45, 15:44, 18:25, 19:12, 19:38 and 21:09. It costs 20 euros and seat reservation is mandatory. These high-speed trains are actually not so fast because they first make a long detour far around the suburb, then take a suburban railtrack. They need 45 minutes to reach Massy-Palaiseau (compare to the 65 minutes required by the RER B). In Massy-Palaiseau, take a taxi to Jouy-en-Josas (15 minutes, about 15 euros), or take the RER C line as above (buy a ticket for about 2 euros).

From Orly airport

The fastest way is to take ORLYVAL which goes to the RER B station Antony. In Antony, take the RER B to go to Massy-Palaiseau (southbound to Saint-Rémy lès Chevreuse).

ORLYVAL is an automatic subway line that connects Orly terminals West and South to RER B station Antony in 5 minutes. From Antony to Massy-Palaiseau it is another less

than 10 minutes run.

From Massy-Palaiseau, follow the directions to Jouy-en-Josas and Université Thalès as above. A ticket from Orly to Jouy-en-Josas is 8.85 euros. The full trip to Thalès is about 50 minutes.

From Paris

Connect either to a station on the RER B line (and follow the directions above to Jouy-en-Josas via Massy-Palaiseau) or to the Montparnasse railways station (take there a train to Versailles Chantiers where you can connect to the RER C line going to the stations Petit Jouy and Jouy-en-Josas). In both cases, a ticket is 3.30 euros

By taxi from the airports

From CDG airport a taxi trip costs around 70 euros. Duration depends on time of the day. Minimum time is 45 minutes; in the rush hour (16:00 - 20:30) it can easily take more than 90 minutes.

From Orly airport it costs around 35 euros and takes about 30 minutes. Cost and duration both depends on traffic.

After 19:00, prices are increased by about 30%.



TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

[Introduction](#)

Social event

[How to get to
Campus
Thalès](#)
[Social event](#)

Thursday December 16 (evening)

Visit and reception at [Château de Breteuil](#)

[Invited
speakers](#)
[Programme](#)
[List of
participants](#)

[Call for papers](#)

[Organising
committee](#)

[Related events](#)
[History](#)



TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

[Introduction](#)

[How to get to
Campus
Thalès](#)
[Social event](#)

[Invited
speakers](#)
[Programme](#)
[List of
participants](#)

[Call for papers](#)

[Organising
committee](#)

[Related events](#)
[History](#)

Invited speakers

- [Zhong Shao](#) (Yale)
- [Thomas C. Hales](#) (CMU Pittsburgh)
- Per Martin-Lof (Stockholm)



TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

Introduction

Registration

How to get to
Campus
Thalès

Social event

Submit a talk

Invited
speakers

Programme

List of
participants

Organising
committee

Related events

History

Tentative Programme

Due to the large number of talk submissions, all submitted talks are planned to be of 15mn plus 5mn for questions.

Wednesday 15 December 2004

09:00 *Coffee*

10:20 *Opening*

10:30 *Session W1*

Mathematics in type theory (*slides*)

Zhaohui Luo (Royal Holloway, University of London)

Codata (*slides*)

Thorsten Altenkirch (University of Nottingham)

Partial co-recursive functions in the calculus of inductive constructions (*slides*)

Yves Bertot (INRIA Sophia-Antipolis)

Truly nested datatypes through dependent datatypes - a challenge for Coq (*slides*)

Ralph Matthes (LMU München)

Standard Equipment for Inductive Datatypes

Conor McBride (Royal Holloway, University of London)

12:10 *Lunch*

14:10 *Session W2*

An E-bicategory of E-categories (*slides*)

Olov Wilander (University of Uppsala)

Pointfree Integration Theory without Choice (*slides*)

Bas Spitters (Nijmegen University)

Relative formal topology

Maria Emilia Maietti, Silvio Valentini (University of Padova)

Mapping the Classical Type Theory of Farmer's STTWU into the Constructive Type Theory of COQ (*slides*)

Freek Wiedijk (Nijmegen University)

15:30 *Break*

16:00 *Session W3*

Type Theory with First-Order Data Types and Size-Change Termination (*slides*)

David Wahlstedt (Chalmers University)

Polymorphism in type theory

Johan Georg Granström (University of Uppsala)

Interactive algebra course with formalised proofs and definitions (*slides*)

Iris Loeb, Lionel Elie Mamane (Nijmegen University), Herman Geuvers (Nijmegen University)

Coherence and Confluence

Zoran Petric (University of Novi Sad), Kosta Došen (SANU, Belgrade)

17:20 *Break*

17:50 *Demo session*

Proof General / Eclipse - A generic interface for interactive theorem provers

Daniel Winterstein (University of Edinburgh) [presentation by Lucas Dixon (University of Edinburgh)]

18:10 *Plain presentation of the posters*

A Modular Lattice Library to Develop Certified

Terminating Static Analyses (*slides*)

David Pichardie (IRISA)

Automatizing geometry using Coq (*slides*)

Julien Narboux (INRIA Futurs)

18:20 *Parallel presentation of the posters*

18:50 *End*

Thursday 16 December 2004

09:00 *Invited talk*

The Essence of Proof-Carrying Code

Zhong Shao (Yale University)

10:00 *Session T1*

A Dependent Type Theory with Names and Binding (*slides*)

Ulrich Schoepp, Ian Stark (University of Edinburgh)

10:20 *Break*

10:50 *Session T1 (continued)*

FOL-nabla in Higher Order Abstract Syntax (*slides*)

Marino Miculan, Ivan Scagnetto (University of Udine)

Higher Order Datatypes Via Containers (*slides*)

Neil Ghani (Leicester University), Thorsten Altenkirch (University of Nottingham), Tarmo Uustalu (IoC Tallinn), Michael Abbott (Leicester University)

A proof search specification of the pi calculus (*slides*)

Alwen Tiu, Dale Miller (INRIA Futurs)

Signals and comonads (*slides*)

Tarmo Uustalu, Varmo Vene (IoC Tallinn)

12:10 *Lunch*

14:10 *Session T2*

Functional Method (*slides*)

Luigi Liquori (INRIA Sophia-Antipolis), Arnaud Spiwack (ENS Cachan)

Interfaces as games, programs as strategies (*slides*)

Markus Michelbrink (University of Swansea)

A machine-checker formalization of the generic model and the random oracle model (*slides*)

Sabrina Tarento, Gilles Barthe (INRIA Sophia-Antipolis)

Verification of BDD-Algorithms in Isabelle / HOL (*slides*)

Veronika Ortner, Norbert Schirmer (TU München)

15:30 *Break*

16:00 *Session T3*

Formalising Bitonic Sort (*slides*)

Ana Bove, Thierry Coquand (Chalmers University)

Proof by contextual menu

Christophe Raffalli (Savoie University)

Overloading in Agda (*slides*)

Catarina Coquand (Chalmers University)

Efficient Type Based Parsing in Strict Languages (*slides*)

Claudio Sacerdoti Coen (INRIA Futurs), Stefano Zacchiroli (Bologna)

17:20 *End*

Visit and reception at Château de Breteuil

Friday 17 December 2004

09:00 *Invited talk*

Toward a formal proof of the Kepler Conjecture ([slides](#))

Tom Hales (Pittsburgh University)

10:00 *Session F1*

Proving Bounds for Real Linear Programs in Isabelle / HOL ([slides](#))

Steven Obua (TU München)

10:20 *Break*

10:50 *Session F1 (continued)*

Verifying inequalities over real numbers in Coq ([slides](#))

Roland Zumkeller (INRIA Futurs)

Formalising Lagrange Theorem in Coq ([slides](#))

Dan Synek (Nijmegen University)

Surreal Numbers in Coq ([slides](#))

Lionel Elie Mamane (Nijmegen University)

Subsets in formalised linear algebra ([slides](#))

Jasper Stein (Nijmegen University)

12:10 *Lunch*

14:10 *Session F2*

The Four Color Theorem in Coq

Georges Gonthier (Microsoft Research Cambridge)

IsaPlanner: A proof planning framework for Isabelle ([slides](#))

Lucas Dixon (University of Edinburgh)

First-order reasoning in the calculus of inductive constructions

Pierre Corbineau (University Paris Sud)

Geometric Logic and Proof Objects ([slides](#))

Marc Bezem (Bergen University)

15:30 *Break*

16:00 *Session F3*

A Formalisation of the Metatheory of Pure Type Systems in Coq (*slides*)

Robin Adams (University of Manchester)

Porting Proofs Between HOL Implementations (*slides*)

Sebastian Skalberg (TU München)

A Quantifier Elimination Procedure over ACFs in Coq using Maple (*slides*)

Micaela Mayero (University Paris 13), David Delahaye (CNAM)

17:00 *Break*

17:30 *Plain demo session*

An overview of the MRG project (*slides*)

Lennart Beringer

PhoX

Christophe Raffalli (Savoie University)

A reflexive tactic for first-order reasoning in Coq

Pierre Corbineau (University Paris Sud)

Epigram

Conor McBride (Royal Holloway, University of London)

18:50 *End*

Business meeting after dinner

Saturday 18 December 2004

09:00 *Invited talk*

100 years of Zermelo's axiom of choice: what was the problem with it?

Per Martin-Löf (Stockholm University)

10:00 *Session S1*

Type Setups for predicate logic (*slides*)

Peter Aczel (University of Manchester)

10:20 *Break*

10:50 *Session S1 (continued)*

Semantical Cut Elimination in Intuitionistic Deduction Modulo (*slides*)

Olivier Hermant (INRIA Futurs)

On constructive existence

Michel Parigot (CNRS, University Paris 7)

A cut elimination theorem for set theory (*slides*)

Gilles Dowek (INRIA Futurs), Alexandre Miquel (University Paris 7)

Continuations, Backtracking and Limits: A Notion of Construction for Classical Logic (*slides*)

Stefano Berardi (University of Torino)

12:10 *Lunch*

13:30 *Session S2*

A logical model for explicit operators (*slides*)

Delia Kesner, Stéphane Lengrand (University Paris 7)
(University Paris 7)

Executing Extracted Programs (*slides*)

Luís Cruz-Filipe (University of Lisbon), Pierre Letouzey (LMU München)

Program extraction in constructive analysis

Helmut Schwichtenberg (LMU München)

Extracting a normalization algorithm from a proof of weak normalization for the simply-typed Lambda-calculus (*slides*)

Stefan Berghofer (TU München)

14:50 *Break*

15:00 *Session S3*

Arguments for a minimal type theory (*slides*)

Giovanni Sambin (University of Padova)

Declarative Language Definitions and Code Generation by Linearization (*slides*)

Aarne Ranta (Chalmers University)

Building Multimodal Dialogue Systems in GF (*slides*)

Björn Bringert (Chalmers University)

Towards a translation of Weak Type Theory into Dependent Type Theory (*slides*)

Georgi Jojgov (Eindhoven University)

16:20 *End*



TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

[Introduction](#)

[How to get to
Campus
Thalès](#)
[Social event](#)

[Invited
speakers](#)
[Programme](#)
[List of
participants](#)

[Call for papers](#)

[Organising
committee](#)

[Related events](#)
[History](#)

List of participants

- Abel Andreas
- Aczel Peter
- Adams Robin
- Altenkirch Thorsten
- Andronick Luis
- Balaa Antonia
- Barendregt Henk
- Barthes Gilles
- Belo Joao Filipe
- Berardi Stefano
- Berghofer Stefan
- Beringer Lennart
- Bertot Yves
- Bezem Marc
- Bove Ana
- Bringert Björn
- Callaghan Paul
- Casteran Pierre
- Chapman James
- Coquand Catarina
- Corbineau Pierre
- Cruz-Filipe Luís
- Curi Giovanni
- David René
- Delahaye David
- Despeyroux Joëlle
- Dixon Lucas
- Dowek Gilles
- Dufay Guillaume
- Filliâtre Jean-Christophe
- Flegontov Alexandre
- Geuvers Herman
- Ghani Neil
- Ghilezan Silvia
- Gonthier Georges

- Granström Johan Georg
- Grégoire Benjamin
- Hales Thomas
- Herbelin Hugo
- Hermant Olivier
- Hernest Mircea Dan
- Jojgov Georgi
- Jouannaud Jean-Pierre
- Kesner Delia
- Kirchner Florent
- Lebresne Sylvain
- Lengrand Stéphane
- Letouzey Pierre
- Lewis-Kelham Edwin
- Loeb Iris
- Luo Zhaohui
- Maenpaa Petri
- Mahboubi Assia
- Maietti Maria Emilia
- Mamane Lionel
- Martin-Löf Per
- Matthes Ralph
- Mayero Micaela
- McBride Conor
- Michelbrink Markus
- Miculan Marino
- Monin Jean-François
- Morris Peter
- Narboux Julien
- Nordström Bengt
- Obua Steven
- Ortner Veronika
- Padovani Luca
- Parigot Michel
- Paulin Christine
- Petric Zoran
- Pichardie David
- Pollack Randy
- Raffalli Christophe
- Ranta Aarne
- Renard Clément
- Ritter Eike
- Rozière Paul
- Ruyer Frédéric
- Rydeheard David
- Sacerdoti Coen Claudio

- Scagnetto Ivan
- Schoepp Ulrich
- Schubert Aleksy
- Schwichtenberg Helmut
- Shao Zhong
- Sinot François-Régis
- Skalberg Sebastian
- Smith Jan
- Spitters Bas
- Soloviev Serguei
- Stein Jasper
- Synek Dan
- Tarento Sabrina
- Tiu Alwen
- Uustalu Tarmo
- Wahlstedt David
- Walukiewicz-Chrzaszcz Daria
- Werner Benjamin
- Wiedijk Freek
- Wilander Olov
- Zacchiroli Stefano
- Zumkeller Roland



TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

Introduction

How to get to
Campus
Thalès
Social event

Invited
speakers
Programme
List of
participants

Call for papers

Organising
committee

Related events
History

Call for papers

OPEN TO ALL INTERESTED RESEARCHERS

The Post-Proceedings of the TYPES 2004 Workshop (types2004.lri.fr) will be published, after a formal referee process, as a volume of the Lecture Notes in Computer Science (LNCS) series. Previous TYPES post-workshop proceedings include LNCS volumes 3085, 2646, 2277, 1657, 1512, 1158, 996 and 806.

We encourage you to submit research papers on the subject of the Types Working Group, i.e. the Computer-Assisted Formal Reasoning, an area which is of deep interest for industry. The aim of our research activities is to develop the technology of formal reasoning based on Type Theory by improving the languages and tools of reasoning and by applying the technology in several domains such as programming languages, certified software, and formalisation of mathematics.

Topics include, but are not limited to:

- foundations of type theory and constructive mathematics
- applications of type theory
- programming with type theory
- industrial uses of type theory technology
- meta-theoretic studies of type systems
- implementation of proof-assistants
- automation in computer-assisted reasoning
- links between type theory and functional programming
- formalizing mathematics using type theory

Also work within the scope of TYPES that was not presented at the workshop or whose authors are not formally involved in the Working Group can be submitted for the proceedings.

SUBMISSION DEADLINE	April 25, 2005
NOTIFICATION OF ACCEPTANCE	June 30, 2005
FINAL VERSION DUE	August 15, 2005

We hope this volume will give a good account of the papers presented at the workshop and of recent research in the field in general. We invite submission of high quality papers, written in English and typeset in LaTeX2e using the LNCS style. (See authors Instructions at <http://www.springeronline.com/lncs>). Submissions should not have been published and should not be under consideration for publication elsewhere. Submissions should be no more than fifteen pages long in LNCS style. Please email your contribution as a self-contained pdf file to:

types2004@inria.fr

In a separate email, give the title, authors and abstract of your submission, as well as email address of the corresponding author. Submissions will be acknowledged (perhaps with some delay).

LNCS is now published in full-text electronic version, as well as printed books. Thus we will need the final LaTeX source files of accepted submissions. The final versions of accepted submissions must be in the LaTeX2e LNCS style, and be as self-contained as possible. With the final version you will also be asked to fill a copyright form for LNCS accepted papers.

The editors (Jean-Christophe Filliatre, Christine Paulin, and Benjamin Werner).

Send your questions and comments to types2004@inria.fr



TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

Introduction

How to get to
Campus
Thalès
Social event

Invited
speakers
Programme
List of
participants

Call for papers

Organising
committee

Related events
History

The organizing committee is composed of:

- [Christine Paulin](#)
- [Hugo Herbelin](#) (programme)
- [Benjamin Werner](#) (programme)
- [Catherine Moreau](#)
- [Catherine Girard](#)
- [Jean-Christophe Filliâtre](#) (web site)



TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

[Introduction](#)

Related events

[How to get to
Campus
Thalès](#)
[Social event](#)

[MoWGLI meeting 2004](#)

[Invited
speakers](#)
[Programme](#)
[List of
participants](#)

[Call for papers](#)

[Organising
committee](#)

[Related events](#)
[History](#)



TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

[Introduction](#)

[How to get to
Campus
Thalès](#)
[Social event](#)

[Invited
speakers](#)
[Programme](#)
[List of
participants](#)

[Call for papers](#)

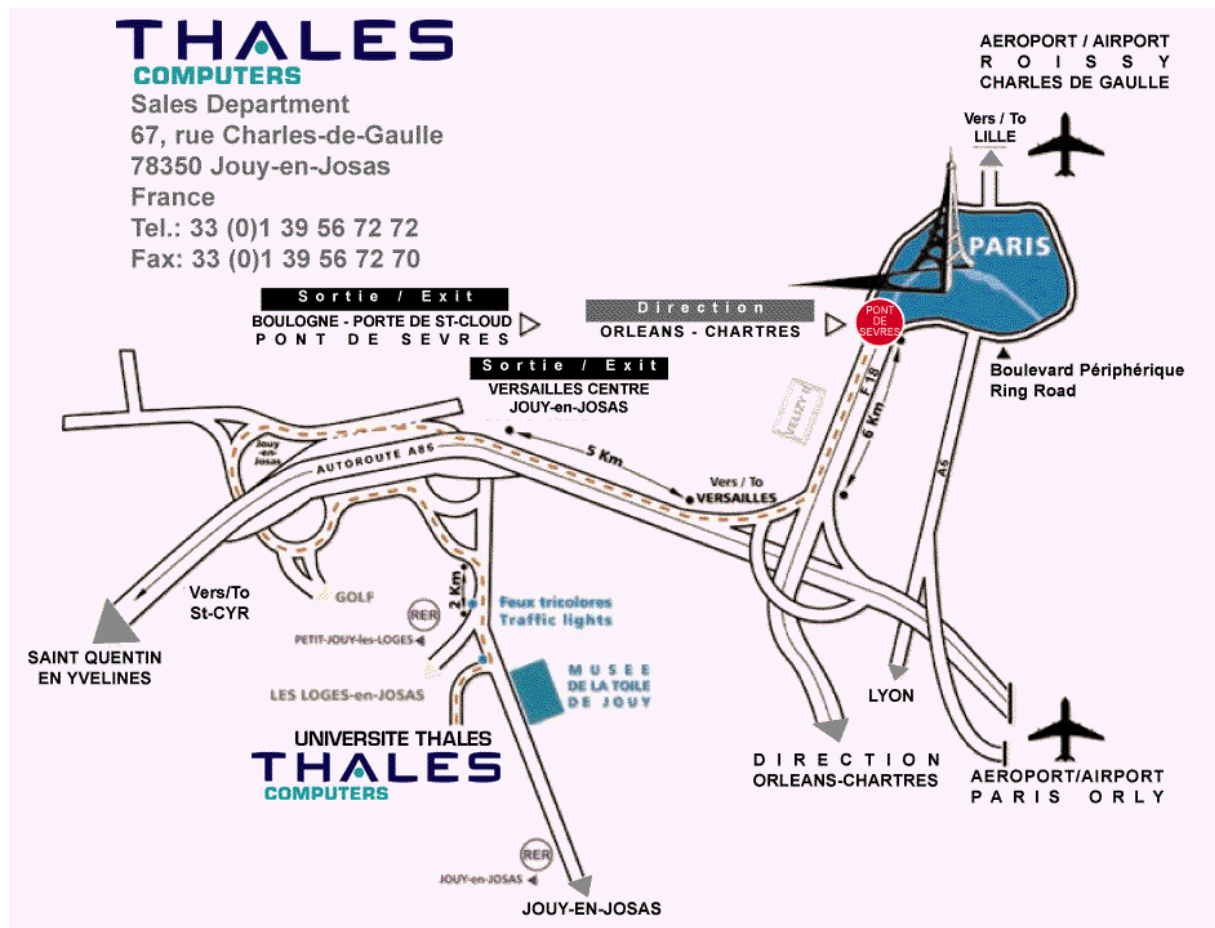
[Organising
committee](#)

[Related events](#)
[History](#)

History

- [2003](#) (Torino, Italy)
- [2002](#) (Berg en Dal, Netherlands)
- [2000](#) (Durham, United Kingdom)
- [1999](#) (Lökeberg, Sweden)
- [1998](#) (Kloster Irsee, Germany)





EN TRAIN • BY TRAIN

La gare de Jouy-en-Josas se trouve à 10 minutes à pied de Thales Computers. Elle est située sur la ligne SNCF reliant Massy-Palaiseau à Versailles Chantiers (train toutes les 1/2 heures). Arrivée à Versailles Chantiers par Paris-Montparnasse (trajet 20 minutes) et à Massy-Palaiseau par les lignes C et B du RER.

Thales Computers is 10 minutes walking distance from Jouy-en-Josas station, which is on the SNCF line between Massy-Palaiseau and Versailles Chantiers (trains runs every half hours). You can reach Versailles Chantiers via Paris-Montparnasse (journey time 20 minutes) and Massy-Palaiseau via the RER B and C lines.

EN VOITURE • BY CAR

De Paris, quitter le périphérique à la sortie Boulogne - Porte de St Cloud. Au Pont de Sèvres prendre la direction Chartres-Orléans puis la direction Versailles (A86). Sortie Versailles-centre, Jouy-en-Josas.

If you are coming from Paris, leave the ringroad (périphérique) at the Boulogne-Porte de St Cloud exit. When you reach Pont de Sèvres, follow the signs for Chartres-Orléans, then the signs for Versailles (A86). Take the Versailles-centre exit and follow the signs for Jouy-en-Josas.

EN AVION • BY AIR

Par / From ORLY: Trajet en taxi environ 30 minutes
About 30 minutes by taxi

Par / From ROISSY: Trajet en taxi environ 1 heure
Charles de Gaulle About 1 hour by taxi

TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

[Introduction](#)

Registration

Registration is no more possible (too many participants).

[How to get to
Campus
Thalès](#)
[Social event](#)

[Invited
speakers](#)
[Programme](#)
[List of
participants](#)

[Call for papers](#)

[Organising
committee](#)

[Related events](#)
[History](#)



TYPES 2004 conference

December 15-18, 2004

Campus Thalès

67, rue Charles de Gaulle - 78350 Jouy-en-Josas - FRANCE

[Introduction](#)

Sorry, deadline has been reached. It is not possible to submit a talk anymore.

[How to get to
Campus
Thalès](#)
[Social event](#)

Please send your abstract by mail to types2004@inria.fr before November 15th.

[Invited
speakers](#)
[Programme](#)
[List of
participants](#)

[Call for papers](#)

[Organising
committee](#)

[Related events](#)
[History](#)



Logics and mathematics in type theory

Zhaohui Luo

Dept of Computer Science

Royal Holloway, Univ of London

Type theory and mathematics

◆ Current type theories

- Martin-Löf's TT (predicative, first-order logic, intuitionistic)
- TTs found in Coq and Lego (impredicative, higher-order logic)

◆ Technology based on type theories

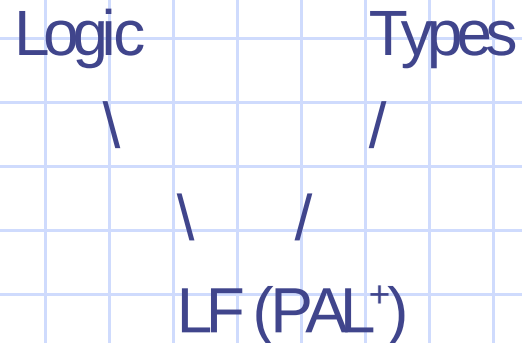
- Mainly applied to constructive math (so far)
- Applications to classical math?

◆ Logic-enriched type theories (Peter Aczel)

- Separation of logics and types
- Flexibility

Logics and type theories in LFs

◆ Structure of logic-enriched TTs in logical framework



* Compare this with existing TTs.

PAL⁺

◆ Logical frameworks

Specifications of TTs and other formal systems

- Martin-Löf's LF
- PAL⁺ – parameterisation and definitions

◆ Some LF notations of “kinds”

- Type – the universe of types
- $\text{El}(A)$ – the kind of elements of type A

Logics in PAL⁺

◆ Propositional universe

- Prop kind
 - ” Note: Prop is not a type (of kind Type or in a type universe).
- Prf[P] kind [P : Prop]
 - ” Note: Propositions are not types.

◆ Classical FOL: an example

- $P \wedge Q : \text{Prop} [P : \text{Prop}, Q : \text{Prop}]$
- $\forall [A, P] : \text{Prop} [A : \text{Type}, P[x:A] : \text{Prop}]$
- $\neg P : \text{Prop} [P : \text{Prop}]$
 - ” $\text{DN}[P, p] : \text{Prf}(P) [P : \text{Prop}, p : \text{Prf}[\neg P]]$

Types

◆ Inductive types/families

- As found in type theory
 - ” Introduced by strictly positive operators (e.g. Nats, Trees, W-types, ...)
- With Induction Rule to eliminate over propositions.

◆ Example: the natural numbers

- $N : \text{Type}$, $0 : N$, $\text{succ}[n] : N \ [n : N]$
- Elimination over types: for $C[n] : \text{Type} \ [n : N]$
 - ” $\text{Elim}_T[C, c, f, n] : C[n]$ (plus computational rules for Elim_T)
- Induction over propositions: for $P[n] : \text{Prop} \ [n : N]$
 - ” $\text{Elim}_P[P, c, f, n] : P[n]$ (computational rules for Elim_P are optional)

◆ Universes

- As before, Peano's fourth axiom can be proved if there are universes (due to Peter Aczel).

(Im)predicativity and other issues

◆ Predicativity

- The system presented so far is predicative.
- Predicative classical mathematics (c.f. Weyl/Feferman's work)

◆ Impredicativity

- E.g. impredicative (typed) sets

$\text{Set}[A] \text{ --- } "A \rightarrow \text{Prop}"$

This introduces impredicativity (and requires further study).

◆ Other issues: e.g., definite description.

Pythagoras

◆ “Pythagoras: machine support for semi-formal proof-oriented mathematics”

- Joint project with Peter Aczel at Manchester, funded by U.K. EPSRC

◆ Work so far (as related to this talk):

- Implementation of PAL^+ , ind types and classical FOL (Yong Luo)
- Formalisation of math (Robin Adams and Yong Luo)
 - ” Proofs and Fundamentals: a first course in mathematics.
Ethan D. Bloch. Birkhauser, 2000. ISBN: 0817641114.
 - ” Predicative development of the number systems
 - ” Impredicative development of sets etc.

◆ Other work in progress

- Semi-formal, user-friendly tool

Flexibility $\Leftarrow$ other applications in reasoning

- E.g., reasoning about dependently-typed programs

Comparisons with existing systems

- Existing TTs
- HOL etc.



Author: Thorsten Altenkirch

Title: Codata

Abstract:

We discuss the rule of codata, i.e. coinductively defined datatypes, in Type Theory. Our basic idea is the symmetry between a producer contract for data which allows us to program with and reason about data and a consumer contract which allows us to deal with codata. Our analysis also reopens the question of extensionality.



Codata

Thorsten Altenkirch
University of Nottingham

Haskell: data = codata ?

Haskell: data = codata ?



```
data List = Nil | Cons Nat List
```

Haskell: data = codata ?

- $\text{data List} = \text{Nil} \mid \text{Cons Nat List}$

- $\text{even} \in \text{List} \rightarrow \text{Bool}$

$\text{even Nil} = \text{True}$

$\text{even (Cons } a \text{ as)} = \neg (\text{even } as)$

Haskell: data = codata ?

- $\text{data List} = \text{Nil} \mid \text{Cons Nat List}$

- $\text{even} \in \text{List} \rightarrow \text{Bool}$

$\text{even Nil} = \text{True}$

$\text{even (Cons } a \text{ as)} = \neg (\text{even as})$

- $\text{from} \in \text{Nat} \rightarrow \text{List}$

$\text{from } n = \text{Cons } n (\text{from } (n + 1))$

Haskell: data = codata ?



data List = Nil | Cons Nat List



even ∈ List → Bool

even Nil = True

even (Cons *a as*) = ¬ (*even as*)



from ∈ Nat → List

from *n* = Cons *n* (*from* (*n* + 1))



even (*from* 0) diverges!

Type Theory: data $\neq$ codata

Type Theory: data $\neq$ codata

data List = Nil | Cons Nat List

codata List[∞] = Nil[∞] | Cons[∞] Nat List[∞]

Type Theory: data $\neq$ codata

data List = Nil | Cons Nat List

codata List $^\infty$ = Nil $^\infty$ | Cons $^\infty$ Nat List $^\infty$

even $\in$ List $\rightarrow$ Bool

even Nil = True

even (Cons *a as*) = $\neg$ (*even as*)

Type Theory: data $\neq$ codata

data List = Nil | Cons Nat List

codata List $^\infty$ = Nil $^\infty$ | Cons $^\infty$ Nat List $^\infty$

even $\in$ List $\rightarrow$ Bool

even Nil = True

even (Cons *a as*) = $\neg$ (*even as*)

from $\in$ Nat $\rightarrow$ List $^\infty$

from *n* = Cons $^\infty$ *n* (*from* (*n* + 1))

Type Theory: data $\neq$ codata

data List = Nil | Cons Nat List

codata List $^\infty$ = Nil $^\infty$ | Cons $^\infty$ Nat List $^\infty$

even $\in$ List $\rightarrow$ Bool

even Nil = True

even (Cons *a as*) = $\neg$ (*even as*)

from $\in$ Nat $\rightarrow$ List $^\infty$

from *n* = Cons $^\infty$ *n* (*from* (*n* + 1))

even (*from* 0) doesn't typecheck.

codata in Type Theory

codata in Type Theory

- Thierry Coquand
Infinite Objects in Type Theory
TYPES 93

codata in Type Theory

- Thierry Coquand
Infinite Objects in Type Theory
TYPES 93
- Eduardo Gimenez
Coinductive Types in COQ
93 – 95
see Coq'Art, pp.347 – 376

Codata ?

Codata ?

- Codata seems more exotic then data.

Codata ?

- Codata seems more exotic than data.
- Categorically codata (terminal coalgebras) is a dual of data (initial algebras)

Codata ?

- Codata seems more exotic than data.
- Categorically codata (terminal coalgebras) is a dual of data (initial algebras)
- Proposal: a conceptual duality based on contracts

Codata ?

- Codata seems more exotic than data.
- Categorically codata (terminal coalgebras) is a dual of data (initial algebras)
- Proposal: a conceptual duality based on contracts
- which justifies *Observational Type Theory* reflecting this symmetry.

Data – the producer contract

Data – the producer contract

The producer of **data** promises that he/she will construct data only using the agreed constructors.

Data – the producer contract

The producer of **data** promises that he/she will construct data only using the agreed constructors.

Consequences:

Data – the producer contract

The producer of **data** promises that he/she will construct data only using the agreed constructors.

Consequences:

- pattern matching

Data – the producer contract

The producer of **data** promises that he/she will construct data only using the agreed constructors.

Consequences:

- pattern matching
- structural recursion

Data – the producer contract

The producer of **data** promises that he/she will construct data only using the agreed constructors.

Consequences:

- pattern matching
- structural recursion
- induction as structural recursion on proofs

Codata – the consumer contract

Codata – the consumer contract

The consumer of **codata** promises that he/she will only analyze codata using the patterns induced by the agreed constructors.

Codata – the consumer contract

The consumer of **codata** promises that he/she will only analyze codata using the patterns induced by the agreed constructors.

Consequences:

Codata – the consumer contract

The consumer of **codata** promises that he/she will only analyze codata using the patterns induced by the agreed constructors.

Consequences:

- constructors

Codata – the consumer contract

The consumer of **codata** promises that he/she will only analyze codata using the patterns induced by the agreed constructors.

Consequences:

- constructors
- guarded corecursion

Codata – the consumer contract

The consumer of **codata** promises that he/she will only analyze codata using the patterns induced by the agreed constructors.

Consequences:

- constructors
- guarded corecursion
- coinduction as guarded recursion on proofs

A simple proposition

A simple proposition

$$\mathit{mapS} \in \mathbf{List}^\infty \rightarrow \mathbf{List}^\infty$$

$$\mathit{mapS} \, \mathbf{Nil}^\infty = \mathbf{Nil}^\infty$$

$$\mathit{mapS} \, \mathbf{Cons}^\infty \, n \, \vec{n} = \mathbf{Cons}^\infty \, (n + 1) \, (\mathit{mapS} \, \vec{n})$$

A simple proposition

$$\mathit{mapS} \in \mathbf{List}^\infty \rightarrow \mathbf{List}^\infty$$

$$\mathit{mapS} \, \mathbf{Nil}^\infty = \mathbf{Nil}^\infty$$

$$\mathit{mapS} \, \mathbf{Cons}^\infty \, n \, \vec{n} = \mathbf{Cons}^\infty \, (n + 1) \, (\mathit{mapS} \, \vec{n})$$

$$\text{let } \frac{n \in \mathbf{Nat}}{\mathit{lem} \, n \in \mathit{mapS} \, (\mathit{from} \, n) = \mathit{from} \, (n + 1)}$$

A simple proposition

$$\mathit{mapS} \in \mathbf{List}^\infty \rightarrow \mathbf{List}^\infty$$

$$\mathit{mapS} \ \mathbf{Nil}^\infty = \mathbf{Nil}^\infty$$

$$\mathit{mapS} \ \mathbf{Cons}^\infty \ n \ \vec{n} = \mathbf{Cons}^\infty \ (n + 1) \ (\mathit{mapS} \ \vec{n})$$

$$\text{let } \frac{n \in \mathbf{Nat}}{\mathit{lem} \ n \in \mathit{mapS} \ (\mathit{from} \ n) = \mathit{from} \ (n + 1)}$$

- Let's have a closer look at =.

Equality for List

Equality for List

data $\frac{\vec{m}, \vec{n} \in \mathbf{List}}{\vec{m} = \vec{n} \in \mathbf{Prop}}$ **where**

Equality for List

data $\frac{\vec{m}, \vec{n} \in \mathbf{List}}{\vec{m} = \vec{n} \in \mathbf{Prop}}$ **where**

$\overline{\text{EqNil} \in \text{Nil} = \text{Nil}}$

Equality for List

data $\frac{\vec{m}, \vec{n} \in \mathbf{List}}{\vec{m} = \vec{n} \in \mathbf{Prop}}$ **where**

$\overline{\text{EqNil} \in \text{Nil} = \text{Nil}}$

$\frac{p \in m = n \quad \vec{p} \in \vec{m} = \vec{n}}{\text{EqCons } p \vec{p} \in \text{Cons } m \vec{m} = \text{Cons } n \vec{n}}$

Properties of $=$

Properties of =

$$\text{let } \frac{\vec{n} \in \mathbf{List}}{\text{refl } \vec{n} \in \vec{n} = \vec{n}}$$

Properties of =

$$\begin{array}{l} \text{let } \frac{\vec{n} \in \mathbf{List}}{\text{refl } \vec{n} \in \vec{n} = \vec{n}} \\ \text{refl Nil} = \text{EqNil} \\ \text{refl (Cons } n \vec{n}) = \text{EqCons (refl } n) (\text{refl } \vec{n}) \end{array}$$

Properties of =

$$\text{let } \frac{\vec{n} \in \mathbf{List}}{\text{refl } \vec{n} \in \vec{n} = \vec{n}}$$
$$\text{refl Nil} = \text{EqNil}$$
$$\text{refl (Cons } n \vec{n}) = \text{EqCons (refl } n) (\text{refl } \vec{n})$$

$$\text{let } \frac{\vec{p} \in \vec{m} = \vec{n} \quad \vec{q} \in \vec{n} = \vec{o}}{\text{trans } \vec{p} \vec{q} \in \vec{m} = \vec{o}}$$

Properties of =

$$\text{let } \frac{\vec{n} \in \mathbf{List}}{\text{refl } \vec{n} \in \vec{n} = \vec{n}}$$
$$\text{refl Nil} = \text{EqNil}$$
$$\text{refl (Cons } n \vec{n}) = \text{EqCons (refl } n) (\text{refl } \vec{n})$$

$$\text{let } \frac{\vec{p} \in \vec{m} = \vec{n} \quad \vec{q} \in \vec{n} = \vec{o}}{\text{trans } \vec{p} \vec{q} \in \vec{m} = \vec{o}}$$
$$\text{trans EqNil} \quad \text{EqNil} = \text{EqNil}$$
$$\text{trans (EqCons } p \vec{p}) (\text{EqCons } q \vec{p})$$
$$= \text{EqCons (trans } p \ q) (\text{trans } \vec{p} \ \vec{q})$$

Equality for List^∞

Equality for List^∞

codata $\frac{\vec{m}, \vec{n} \in \text{List}^\infty}{\vec{m} = \vec{n} \in \text{Prop}}$ **where**

Equality for List^∞

codata $\frac{\vec{m}, \vec{n} \in \text{List}^\infty}{\vec{m} = \vec{n} \in \text{Prop}}$ **where**

$\overline{\text{EqNil}^\infty \in \text{Nil}^\infty = \text{Nil}^\infty}$

Equality for List^∞

codata $\frac{\vec{m}, \vec{n} \in \text{List}^\infty}{\vec{m} = \vec{n} \in \text{Prop}}$ **where**

$$\overline{\text{EqNil}^\infty \in \text{Nil}^\infty = \text{Nil}^\infty}$$

$$\frac{p \in m = n \quad \vec{p} \in \vec{m} = \vec{n}}{\text{EqCons}^\infty p \vec{p} \in \text{Cons}^\infty m \vec{m} = \text{Cons}^\infty n \vec{n}}$$

Properties of $=$

Properties of =

$$\text{let } \frac{\vec{n} \in \mathbf{List}^\infty}{\text{refl } \vec{n} \in \vec{n} = \vec{n}}$$

Properties of =

$$\begin{array}{l} \text{let } \frac{\vec{n} \in \mathbf{List}^\infty}{\text{refl } \vec{n} \in \vec{n} = \vec{n}} \\ \text{refl Nil}^\infty = \text{EqNil}^\infty \\ \text{refl (Cons}^\infty n \vec{n}) = \text{EqCons}^\infty (\text{refl } n) (\text{refl } \vec{n}) \end{array}$$

Properties of =

$$\text{let } \frac{\vec{n} \in \mathbf{List}^\infty}{\text{refl } \vec{n} \in \vec{n} = \vec{n}}$$
$$\text{refl Nil}^\infty = \text{EqNil}^\infty$$
$$\text{refl (Cons}^\infty n \vec{n}) = \text{EqCons}^\infty (\text{refl } n) (\text{refl } \vec{n})$$

$$\text{let } \frac{\vec{p} \in \vec{m} = \vec{n} \quad \vec{q} \in \vec{n} = \vec{o}}{\text{trans } \vec{p} \vec{q} \in \vec{m} = \vec{o}}$$

Properties of =

$$\begin{array}{l} \text{let } \frac{\vec{n} \in \mathbf{List}^\infty}{\text{refl } \vec{n} \in \vec{n} = \vec{n}} \\ \text{refl Nil}^\infty = \text{EqNil}^\infty \\ \text{refl (Cons}^\infty n \vec{n}) = \text{EqCons}^\infty (\text{refl } n) (\text{refl } \vec{n}) \end{array}$$

$$\begin{array}{l} \text{let } \frac{\vec{p} \in \vec{m} = \vec{n} \quad \vec{q} \in \vec{n} = \vec{o}}{\text{trans } \vec{p} \vec{q} \in \vec{m} = \vec{o}} \\ \text{trans EqNil}^\infty \quad \text{EqNil}^\infty = \text{EqNil}^\infty \\ \text{trans (EqCons}^\infty p \vec{p}) (\text{EqCons}^\infty q \vec{q}) \\ \quad = \text{EqCons}^\infty (\text{trans } p q) (\text{trans } \vec{p} \vec{q}) \end{array}$$

A simple proof

let
$$\frac{n \in \mathbf{Nat}}{lem\ n \in mapS\ (from\ n) = from\ (n + 1)}$$

A simple proof

let
$$\frac{n \in \mathbf{Nat}}{lem\ n \in mapS\ (from\ n) = from\ (n + 1)}$$

$$lem\ n = EqCons^\infty\ (n + 1)\ (lem\ (n + 1))$$

A simple proof

let
$$\frac{n \in \mathbf{Nat}}{lem\ n \in mapS\ (from\ n) = from\ (n + 1)}$$

$$lem\ n = EqCons^\infty\ (n + 1)\ (lem\ (n + 1))$$

- Coinductive reasoning can be easy.

A simple proof

let
$$\frac{n \in \mathbf{Nat}}{lem\ n \in mapS\ (from\ n) = from\ (n + 1)}$$

$$lem\ n = EqCons^\infty\ (n + 1)\ (lem\ (n + 1))$$

- Coinductive reasoning can be easy.
- Guarded coinduction is guarded corecursion on proofs.

A simple proof

let
$$\frac{n \in \mathbf{Nat}}{lem\ n \in mapS\ (from\ n) = from\ (n + 1)}$$

$$lem\ n = EqCons^\infty\ (n + 1)\ (lem\ (n + 1))$$

- Coinductive reasoning can be easy.
- Guarded coinduction is guarded corecursion on proofs.
- There is no need to construct bisimulations.



The mirror

The mirror

data	codata

The mirror

data	codata
inductive	

The mirror

data	codata
inductive	coinductive

The mirror

data	codata
inductive finite objects	coinductive

The mirror

data	codata
inductive finite objects	coinductive infinite objects

The mirror

data	codata
inductive finite objects structural recursion	coinductive infinite objects

The mirror

data	codata
inductive	coinductive
finite objects	infinite objects
structural recursion	guarded corecursion

The mirror

data	codata
inductive	coinductive
finite objects	infinite objects
structural recursion	guarded corecursion
structural induction	

The mirror

data	codata
inductive	coinductive
finite objects	infinite objects
structural recursion	guarded corecursion
structural induction	guarded coinduction

The mirror

data	codata
inductive	coinductive
finite objects	infinite objects
structural recursion	guarded corecursion
structural induction	guarded coinduction

- Where do functions live?

The mirror

data	codata
inductive	coinductive
finite objects	infinite objects
structural recursion	guarded corecursion
structural induction	guarded coinduction

- Where do functions live?
- Functions are codata.

The mirror

data	codata
inductive	coinductive
finite objects	infinite objects
structural recursion	guarded corecursion
structural induction	guarded coinduction

- Where do functions live?
- Functions are codata.
- Consumer contract:
You may only apply a function.

Leibniz ?

$$\text{let } \frac{P \in \mathbf{Nat} \rightarrow \mathbf{Type} \quad \vec{q} \in \vec{m} = \vec{n} \quad \vec{m} \in \mathbf{List} \quad p \in P \vec{m}}{\text{leibniz } P \vec{p} \, p \in P \vec{n}}$$

Leibniz ?

let $\frac{P \in \mathbf{Nat} \rightarrow \mathbf{Type} \quad \vec{q} \in \vec{m} = \vec{n} \quad \vec{m} \in \mathbf{List} \quad p \in P \vec{m}}{leibniz\ P\ \vec{p}\ p \in P\ \vec{n}}$

$leibniz\ P\ \text{EqNil} \quad \text{Nil} \quad p = p$

$leibniz\ P\ (\text{EqCons}\ q\ \vec{q})\ (\text{Cons}\ m\ \vec{m})\ p =$
 $leibniz\ (\lambda n \rightarrow P\ (\text{Cons}\ n\ \vec{m}))\ m\ q$
 $(leibniz\ (\lambda \vec{n} \rightarrow P\ (\text{Cons}\ m\ \vec{n}))\ \vec{m}\ \vec{q}\ p)$

Leibniz ?

$$\begin{array}{l} \text{let } \frac{P \in \mathbf{Nat} \rightarrow \mathbf{Type} \quad \vec{q} \in \vec{m} = \vec{n} \quad \vec{m} \in \mathbf{List} \quad p \in P \vec{m}}{leibniz\ P\ \vec{p}\ p \in P\ \vec{n}} \\ leibniz\ P\ \text{EqNil} \quad \text{Nil} \quad p = p \\ leibniz\ P\ (\text{EqCons}\ q\ \vec{q})\ (\text{Cons}\ m\ \vec{m})\ p = \\ \quad leibniz\ (\lambda n \rightarrow P\ (\text{Cons}\ n\ \vec{m}))\ m\ q \\ \quad (leibniz\ (\lambda \vec{n} \rightarrow P\ (\text{Cons}\ m\ \vec{n}))\ \vec{m}\ \vec{q}\ p) \end{array}$$

- *leibniz* doesn't dualize to $\mathbf{List}^\infty$.

Observational Type Theory

Observational Type Theory

- We can implement *leibniz* by internalizing the setoid model – see my LICS 99 paper *Extensional Type Theory, intensionally*.

Observational Type Theory

- We can implement *leibniz* by internalizing the setoid model – see my LICS 99 paper *Extensional Type Theory, intensionally*.
- Using this construction we implement both consumer and producer contracts without giving up decidability.

Observational Type Theory

- We can implement *leibniz* by internalizing the setoid model – see my LICS 99 paper *Extensional Type Theory, intensionally*.
- Using this construction we implement both consumer and producer contracts without giving up decidability.
- This is based on a translation of Observational Type Theory into intensional Type Theory + a proof irrelevant universe of propositions.

Observational Type Theory

- We can implement *leibniz* by internalizing the setoid model – see my LICS 99 paper *Extensional Type Theory, intensionally*.
- Using this construction we implement both consumer and producer contracts without giving up decidability.
- This is based on a translation of Observational Type Theory into intensional Type Theory + a proof irrelevant universe of propositions.
- Alternative: any two hypothetical proofs of False are convertible.

A short history of Type Theory

A short history of Type Theory

Anarchy

A short history of Type Theory

Anarchy

No contracts, not even producer contracts.

A short history of Type Theory

Anarchy

No contracts, not even producer contracts.
Instead of $\prod n \in \mathbf{Nat}: \dots$ we write
 $\prod n \in \mathbf{Nat}. (Ind\ n) \rightarrow \dots$

A short history of Type Theory

Anarchy

No contracts, not even producer contracts.

Instead of $\prod n \in \mathbf{Nat}: \dots$ we write

$\prod n \in \mathbf{Nat}. (Ind\ n) \rightarrow \dots$

Impredicative encodings of data

A short history of Type Theory

Wild West

A short history of Type Theory

Wild West

Producer contracts but no consumer contracts.

A short history of Type Theory

Wild West

Producer contracts but no consumer contracts.

We can quantify over $\mathbf{Nat}$

A short history of Type Theory

Wild West

Producer contracts but no consumer contracts.

We can quantify over Nat

We have to verify again and again that a consumer of codata respects equality.

A short history of Type Theory

Wild West

Producer contracts but no consumer contracts.

We can quantify over $\mathbf{Nat}$

We have to verify again and again that a consumer of codata respects equality.

Intensional Type Theory

A short history of Type Theory

Rule of law

A short history of Type Theory

Rule of law

Producer and consumer contracts.

A short history of Type Theory

Rule of law

Producer and consumer contracts.
We can quantify over $\mathbf{Nat}$

A short history of Type Theory

Rule of law

Producer and consumer contracts.

We can quantify over $\mathbf{Nat}$

We know that any consumer of codata respects equality.

A short history of Type Theory

Rule of law

Producer and consumer contracts.

We can quantify over $\mathbf{Nat}$

We know that any consumer of codata respects equality.

Observational Type Theory

Observational Epigram

Observational Epigram

- The goal of our recently funded EPSRC project *Decidable Type Theory with Observational Equality* is to implement a Type Theory with observational equality (Observational Epigram)

Observational Epigram

- The goal of our recently funded EPSRC project *Decidable Type Theory with Observational Equality* is to implement a Type Theory with observational equality (Observational Epigram)
- We want to improve on my LICS 99 paper by adding the conversion equality
 $leibniz \dots refl\ x \equiv x$

Observational Epigram

- The goal of our recently funded EPSRC project *Decidable Type Theory with Observational Equality* is to implement a Type Theory with observational equality (Observational Epigram)
- We want to improve on my LICS 99 paper by adding the conversion equality
 $leibniz \dots refl\ x \equiv x$
- And hence strictly extend intensional Type Theory.

Observational Epigram

- The goal of our recently funded EPSRC project *Decidable Type Theory with Observational Equality* is to implement a Type Theory with observational equality (Observational Epigram)
- We want to improve on my LICS 99 paper by adding the conversion equality
 $leibniz \dots refl\ x \equiv x$
- And hence strictly extend intensional Type Theory.
- We also want to realize another extension of the mirror:

Observational Epigram

- The goal of our recently funded EPSRC project *Decidable Type Theory with Observational Equality* is to implement a Type Theory with observational equality (Observational Epigram)
- We want to improve on my LICS 99 paper by adding the conversion equality
 $leibniz \dots refl\ x \equiv x$
- And hence strictly extend intensional Type Theory.
- We also want to realize another extension of the mirror:

data	codata
subset types	quotient types

Author: Yves Bertot

Title: Partial co-recursive functions in the calculus of inductive constructions

Abstract:

We describe the problems encountered in formalizing filters, which are typical examples of partial co-recursive on infinite streams. We propose a solution and show how this solution can be applied to provide a new modelization of Eratosthene's sieve.

Partial co-recursive functions

Yves Bertot

INRIA Sophia Antipolis

Inductive to Co-inductive types

- Inductive types
 - Constructors, pattern-matching,
 - Recursive programming parallels proofs by induction,
 - No general recursion, but close (accessibility),
 - Support partial recursion (thanks to dependent types).
- Infinite data-types in functional programming!
- Coinductive types
 - pattern-matching, constructors,
 - A dual form of recursion (co-recursion),
 - Partiality?

Recursion

- Introduction: 0 , $Succ$ (constructors),
- Elimination:
 $rec_{N,A} : A \rightarrow (A \rightarrow A \rightarrow A) \rightarrow A \rightarrow A$,
 - $rec_{N,A} 0 v f = 0$
 - $rec_{N,A} (Succ p) v f = f p (rec_{N,A} p v f)$,
- Dépendance:
 $rec_{N,P} : P(0) \rightarrow$
 $(\forall x, P(x) \rightarrow P(Succ(x))) \rightarrow$
 $\forall x, P(x)$
- Recursion is controlled in the way it *consumes* inductive data,
- References (Girard, Martin-Löf),

Guarded recursion

- A writing style improvement,
- Pattern matching:

$$\begin{aligned}g_1 \ 0 &= a \\g_1 \ (Succ \ x) &= h \ x\end{aligned}$$

- Recursion, g_2 is applied on a subterm:

$$\begin{aligned}g_2 \ 0 &= a \\g_2 \ (Succ \ x) &= f \ x \ (g_2 \ x)\end{aligned}$$

- Partial functions, $(inv \dots)$ must be a subterm:

$$\begin{aligned}g_3 \ 1 \ dom_1 &= a \\g_3 \ x \ dom_x &= f \ x \ (g_3 \ (x/2) \\&\quad (inv \ x \ dom_x \ x \neq 1))\end{aligned}$$

Co-recursion

- observers come first, Constructors come later,
- lists: $hd : L \rightarrow A$ and $tl : L \rightarrow L$ as observers,
- $corec_{L,B} : B \rightarrow (A \times B) \rightarrow B \rightarrow L$,
 - $hd(corec_{L,B} f b) : \pi_1(f b)$,
 - $tl(corec_{L,B} f b) : \pi_2(f b)$.
- Duality: rec applied to a constructor is a redex, an observer applied to $corec$ is a redex.

Guarded corecursion

- Observers as pattern matching constructs on a set of constructors,
- Infinite lists: one constructor $cons : A \rightarrow L \rightarrow L$,
 - $hd(cons\ a\ b) = a, tl(cons\ a\ b) = b,$
- Use in recursive construction:
 $gb = cons\ (f_1\ b)\ (g\ (f_2\ b))$
- Guard: g can only appear as second argument of $cons$,
- A syntactic criterion to ensure *productivity*.

Partial co-recursion

- Problem: several recursive calls may be needed before producing data,
- Favourite example: *filter* on infinite lists (streams),
- Only some input streams ensure that filter produces a stream.
- Beyond the expressive power of guarded co-recursion?

Solution using dependent types

- Add a second argument expressing that the first one is in the domain,
- The domain definition has a co-inductive part and an inductive part,
- Non-productive recursive calls are guarded w.r.t. inductive part of the domain.

Example on filter

- A stream is a good argument for *filter* if:
 - it contains an acceptable element (inductive predicate),
 - its tail also is a good argument (co-inductive predicate).
- Structure of *filter*:
 - A recursive part to compute three items
 - first acceptable element,
 - stream tail after that element,
 - proof that the tail is in the domain,
 - A co-recursive part restarts the search on the tail after producing the element.

Applications

- Eratosthenes' sieve on streams of natural numbers,
- Exact arithmetic on continued fractions (again streams of natural

```
sieve (p:rest) =  
    p:(sieve  
        [y | y<-rest,  
             y 'rem' p /= 0])  
primes = sieve [2..]
```

- Possibility to separate domain computation from “informative computation” numbers).

Author: Ralph Matthes

Title: Truly nested datatypes through dependent datatypes - a challenge for Coq

Abstract:

Following ideas by A. Setzer and P. Aczel, a predicative justification of the truly nested rank-2 datatype $\text{Lam}^\wedge$ representing untyped lambda-terms that also have a notion of explicit flattening (a weak form of explicit substitution that blocks beta-redexes) is given by an implementation within Coq. The true nesting is turned into a strictly-positive inductive family that is indexed over finite lists of booleans. Implementing the complete beta-developments for the heterogeneous rank-2 datatype Lam for pure lambda-terms by help of $\text{Lam}^\wedge$ turned out to be a challenge for pattern-matching definitions and the program extraction mechanism of Coq.

Truly nested datatypes through dependent datatypes – a challenge for Coq

Ralph Matthes

Universität München = LMU = University of Munich

Workshop TYPES 2004
Jouy-en-Josas, France
December 15, 2004

Abstract

Following ideas by A. Setzer and P. Aczel, a predicative justification of the truly nested rank-2 datatype $\widehat{\text{Lam}}$ representing untyped lambda-terms that also have a notion of explicit flattening (a weak form of explicit substitution that blocks beta-redexes) is given by an implementation within Coq. The true nesting is turned into a strictly-positive inductive family that is indexed over finite lists of booleans. Implementing the complete beta-developments for the heterogeneous rank-2 datatype $\widehat{\text{Lam}}$ for pure lambda-terms by help of $\widehat{\text{Lam}}$ turned out to be a challenge for pattern-matching definitions and the program extraction mechanism of Coq.

Outline

1 Nested Datatypes

- Heterogeneous Datatypes
- True Nesting
- Impredicative Encoding

2 Case Study: Lambda Calculus with Explicit Flattening

- Lambda Calculus
- Explicit Flattening
- Developments
- Conclusion

Outline

1 Nested Datatypes

- Heterogeneous Datatypes
- True Nesting
- Impredicative Encoding

2 Case Study: Lambda Calculus with Explicit Flattening

- Lambda Calculus
- Explicit Flattening
- Developments
- Conclusion

Outline

1 Nested Datatypes

- Heterogeneous Datatypes
- True Nesting
- Impredicative Encoding

2 Case Study: Lambda Calculus with Explicit Flattening

- Lambda Calculus
- Explicit Flattening
- Developments
- Conclusion

heterogeneous/nonregular datatype

Definition (heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A .

Example (powerlists)

In Haskell, define

```
data PList a = Zero a | Succ (PList (a,a))
```

Finite elements of type `PList a` are ensured by the type system to have a power of 2 elements of type `a`.

Lists are regular: `data List a = Nil | Cons a (List a)`

heterogeneous/nonregular datatype

Definition (heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A .

Example (powerlists)

In Haskell, define

```
data PList a = Zero a | Succ (PList (a,a))
```

Finite elements of type `PList a` are ensured by the type system to have a power of 2 elements of type `a`.

Lists are regular: `data List a = Nil | Cons a (List a)`

heterogeneous/nonregular datatype

Definition (heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A .

Example (powerlists)

In Haskell, define

```
data PList a = Zero a | Succ (PList (a,a))
```

Finite elements of type `PList a` are ensured by the type system to have a power of 2 elements of type `a`.

Lists are **regular**: `data List a = Nil | Cons a (List a)`

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from a to (a, a) .

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined “stand-alone”.
- No direct support in Coq with predicative `Set`.

Good news: Full support in Coq with impredicative `Set` (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from a to (a, a) .

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined “stand-alone”.
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with impredicative *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from a to (a, a) .

Consequences

- No separate understanding of the **slices**, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined “stand-alone”.
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with impredicative *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from a to (a, a) .

Consequences

- No separate understanding of the **slices**, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined “stand-alone”.
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with impredicative *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from a to (a, a) .

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined “stand-alone”.
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with impredicative *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from a to (a, a) .

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined “stand-alone”.
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with **impredicative** *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Nested datatypes are harder to deal with.

This is no definition by recursion on the type argument: The argument is becoming more complex, e. g., we pass from a to (a, a) .

Consequences

- No separate understanding of the slices, e. g., `PList Int`.
- No recursive definition of a function that operates only on one slice, e. g., summing up the entries of a finite element of `PList Int` cannot be defined “stand-alone”.
- No direct support in Coq with predicative *Set*.

Good news: Full support in Coq with **impredicative** *Set* (needs option `-impredicative-set`). But nested dependent pattern-matching remains a difficult task, see below.

Outline

1 Nested Datatypes

- Heterogeneous Datatypes
- True Nesting
- Impredicative Encoding

2 Case Study: Lambda Calculus with Explicit Flattening

- Lambda Calculus
- Explicit Flattening
- Developments
- Conclusion

Definition (Reminder: heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A .

Definition (truly nested datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A refers to family members for other types that even involve other family members.

Example (self-composition)

```
data Comp a = ... | Constr (Comp (Comp a))  
yields the typing Constr :: Comp (Comp a) -> Comp a
```

Definition (Reminder: heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A .

Definition (**truly** nested datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A refers to family members for other types that **even involve other family members**.

Example (self-composition)

```
data Comp a = ... | Constr (Comp (Comp a))  
yields the typing Constr :: Comp (Comp a) -> Comp a
```

Definition (Reminder: heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A .

Definition (truly nested datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A refers to family members for other types that even involve other family members.

Example (self-composition)

```
data Comp a = ... | Constr (Comp (Comp a))  
yields the typing Constr :: Comp (Comp a) -> Comp a
```

Definition (Reminder: heterogeneous datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A may refer to family members for other types than A .

Definition (truly nested datatype of rank 2)

A simultaneously defined type-indexed family of inductive types where the constructor of the family member for A refers to family members for other types that even involve other family members.

Example (self-composition)

```
data Comp a = ... | Constr (Comp (Comp a))  
yields the typing Constr :: Comp (Comp a) -> Comp a
```


True nesting is much more difficult to handle.

- Coq does not accept these at all:
Non strictly positive occurrence of "Comp" in
"forall A : Set, Comp (Comp A) -> Comp A"
- $\lambda F^{Set \rightarrow Set}. F \circ F : (Set \rightarrow Set) \rightarrow (Set \rightarrow Set)$ is not even monotone in a naive sense because F would have to be restricted to be monotone as well.
- The recursive calls in recursive function definitions are in most cases not with structurally smaller arguments but refer to the whole function about to be defined.
- Termination of the naturally arising algorithms needs thorough justification.

True nesting is much more difficult to handle.

- Coq does not accept these at all:
Non strictly positive occurrence of "Comp" in
"forall A : Set, Comp (Comp A) -> Comp A"
- $\lambda F^{Set \rightarrow Set}. F \circ F : (Set \rightarrow Set) \rightarrow (Set \rightarrow Set)$ is not even monotone in a naive sense because F would have to be restricted to be monotone as well.
- The recursive calls in recursive function definitions are in most cases not with structurally smaller arguments but refer to the whole function about to be defined.
- Termination of the naturally arising algorithms needs thorough justification.

True nesting is much more difficult to handle.

- Coq does not accept these at all:
Non strictly positive occurrence of "Comp" in
"forall A : Set, Comp (Comp A) -> Comp A"
- $\lambda F^{Set \rightarrow Set}. F \circ F : (Set \rightarrow Set) \rightarrow (Set \rightarrow Set)$ is not even monotone in a naive sense because F would have to be restricted to be monotone as well.
- The recursive calls in recursive function definitions are in most cases **not** with **structurally smaller** arguments but refer to the whole function about to be defined.
- Termination of the naturally arising algorithms needs thorough justification.

True nesting is much more difficult to handle.

- Coq does not accept these at all:
Non strictly positive occurrence of "Comp" in
"forall A : Set, Comp (Comp A) -> Comp A"
- $\lambda F^{Set \rightarrow Set}. F \circ F : (Set \rightarrow Set) \rightarrow (Set \rightarrow Set)$ is not even monotone in a naive sense because F would have to be restricted to be monotone as well.
- The recursive calls in recursive function definitions are in most cases **not** with **structurally smaller** arguments but refer to the whole function about to be defined.
- **Termination** of the naturally arising algorithms needs thorough justification.

Outline

1 Nested Datatypes

- Heterogeneous Datatypes
- True Nesting
- Impredicative Encoding

2 Case Study: Lambda Calculus with Explicit Flattening

- Lambda Calculus
- Explicit Flattening
- Developments
- Conclusion

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports iteration on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19???)
- It does not even support a constant-time predecessor operation on the natural numbers (Urzyczyn/Splawski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article “Iteration and coiteration schemes for higher-order and nested datatypes”

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports **iteration** on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19??)
- It does not even support a constant-time predecessor operation on the natural numbers (Urzyczyn/Splawski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article “Iteration and coiteration schemes for higher-order and nested datatypes”

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports iteration on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19??)
- It does **not** even **support** a constant-time predecessor operation on the natural numbers (Urzyczyn/Spławski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article “Iteration and coiteration schemes for higher-order and nested datatypes”

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports iteration on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19??).
- It does not even support a constant-time predecessor operation on the natural numbers (Urzyczyn/Splawski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article “Iteration and coiteration schemes for higher-order and nested datatypes”

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports iteration on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19??).
- It does not even support a constant-time predecessor operation on the natural numbers (Urzyczyn/Splawski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article “Iteration and coiteration schemes for higher-order and nested datatypes”

Reconstruction within F^ω

- System F^ω (Girard 1972) is pure higher-order parametric polymorphism.
- It supports iteration on inductive datatypes (Böhm/Berarducci 1985) and coiteration on coinductive datatypes (Dual 19??).
- It does not even support a constant-time predecessor operation on the natural numbers (Urzyczyn/Splawski 1999).
- The simulation of (co-)iteration can be generalized to arbitrary monotone constructors of all finite kinds.
- This includes truly nested datatypes of rank 2 like `Comp`.
- See my article with Andreas Abel and Tarmo Uustalu, already online as TCS article “Iteration and coiteration schemes for higher-order and nested datatypes”

Outline

- 1 Nested Datatypes
 - Heterogeneous Datatypes
 - True Nesting
 - Impredicative Encoding
- 2 Case Study: Lambda Calculus with Explicit Flattening
 - Lambda Calculus
 - Explicit Flattening
 - Developments
 - Conclusion

Aims of this case study

- Justify a truly nested datatype within predicative type theory.
- Implement it in Coq with predicative *Set*.
- Do an extended example of pattern matching with a heterogeneous datatype (as represented within Coq with impredicative *Set*).
- Try program extraction from the proof development.

Aims of this case study

- Justify a truly nested datatype within predicative type theory.
- Implement it in Coq with predicative *Set*.
- Do an extended example of pattern matching with a heterogeneous datatype (as represented within Coq with impredicative *Set*).
- Try program extraction from the proof development.

Aims of this case study

- Justify a truly nested datatype within predicative type theory.
- Implement it in Coq with predicative *Set*.
- Do an extended example of pattern matching with a heterogeneous datatype (as represented within Coq with impredicative *Set*).
- Try program extraction from the proof development.

Aims of this case study

- Justify a truly nested datatype within predicative type theory.
- Implement it in Coq with predicative *Set*.
- Do an extended example of pattern matching with a heterogeneous datatype (as represented within Coq with impredicative *Set*).
- Try program extraction from the proof development.

Outline

- 1 Nested Datatypes
 - Heterogeneous Datatypes
 - True Nesting
 - Impredicative Encoding
- 2 Case Study: Lambda Calculus with Explicit Flattening
 - Lambda Calculus
 - Explicit Flattening
 - Developments
 - Conclusion

Higher-order de Bruijn representation of untyped λ -calculus

We follow Altenkirch/Reus, Bird/Paterson and Fiore/Plotkin/Turi (all 1999).

Lam shall be the least pre-fixed point of the rank 2 “functor”

$$LamF := \lambda X \lambda A. A + XA \times XA + X(1 + A).$$

In Coq with impredicative *Set*, this looks like

```
Inductive Lam : Set -> Set :=
  var : forall A : Set, A -> Lam A
| app : forall A : Set, Lam A -> Lam A -> Lam A
| abs : forall A : Set, Lam (One + A) -> Lam A
```

Note that the functor *LamF* has to be taken apart in order to get reasonable induction and recursion principles from Coq.

Higher-order de Bruijn representation of untyped λ -calculus

We follow Altenkirch/Reus, Bird/Paterson and Fiore/Plotkin/Turi (all 1999).

Lam shall be the least pre-fixed point of the rank 2 “functor”

$$LamF := \lambda X \lambda A. A + XA \times XA + X(1 + A).$$

In Coq with impredicative *Set*, this looks like

```
Inductive Lam : Set -> Set :=
  var : forall A : Set, A -> Lam A
| app : forall A : Set, Lam A -> Lam A -> Lam A
| abs : forall A : Set, Lam (One + A) -> Lam A
```

Note that the functor *LamF* has to be taken apart in order to get reasonable induction and recursion principles from Coq.

Higher-order de Bruijn representation of untyped λ -calculus

We follow Altenkirch/Reus, Bird/Paterson and Fiore/Plotkin/Turi (all 1999).

Lam shall be the least pre-fixed point of the rank 2 “functor”

$$LamF := \lambda X \lambda A. A + XA \times XA + X(1 + A).$$

In Coq with impredicative *Set*, this looks like

```
Inductive Lam : Set -> Set :=
  var : forall A : Set, A -> Lam A
| app : forall A : Set, Lam A -> Lam A -> Lam A
| abs : forall A : Set, Lam (One + A) -> Lam A
```

Note that the functor $LamF$ has to be taken apart in order to get reasonable induction and recursion principles from Coq.

Substitution is not so easy to represent.

Parallel substitution naturally has the type

$$\forall A \forall B. \underbrace{\text{Lam } A}_{\text{term argument}} \rightarrow \underbrace{(A \rightarrow \text{Lam } B)}_{\text{substitution rule}} \rightarrow \text{Lam } B.$$

During recursion, the substitution rule also has to change its type and hence is not just a parameter of the definition. So, the definition has to be given simultaneously for all types B . More precisely, a syntactic Kan extension is needed in the pure iterative style—an essentially impredicative construction.

Substitution is not so easy to represent.

Parallel substitution naturally has the type

$$\forall A \forall B. \underbrace{\text{Lam } A}_{\text{term argument}} \rightarrow \underbrace{(A \rightarrow \text{Lam } B)}_{\text{substitution rule}} \rightarrow \text{Lam } B.$$

During recursion, the substitution rule also has to change its type and hence is not just a parameter of the definition. So, the definition has to be given simultaneously for all types B . More precisely, a syntactic Kan extension is needed in the pure iterative style—an essentially impredicative construction.

Substitution is not so easy to represent.

Parallel substitution naturally has the type

$$\forall A \forall B. \quad \underbrace{\text{Lam } A}_{\text{term argument}} \rightarrow \underbrace{(A \rightarrow \text{Lam } B)}_{\text{substitution rule}} \rightarrow \text{Lam } B.$$

During recursion, the substitution rule also has to change its type and hence is not just a parameter of the definition. So, the definition has to be given simultaneously for all types B . More precisely, a syntactic Kan extension is needed in the pure iterative style—an essentially impredicative construction.

Substitution is not so easy to represent.

Parallel substitution naturally has the type

$$\forall A \forall B. \quad \underbrace{\text{Lam } A}_{\text{term argument}} \rightarrow \underbrace{(A \rightarrow \text{Lam } B)}_{\text{substitution rule}} \rightarrow \text{Lam } B.$$

During recursion, the substitution rule also has to change its type and hence is not just a parameter of the definition. So, the definition has to be given simultaneously for all types B . More precisely, a **syntactic Kan extension** is needed in the pure iterative style—an essentially impredicative construction.

Outline

1 Nested Datatypes

- Heterogeneous Datatypes
- True Nesting
- Impredicative Encoding

2 Case Study: Lambda Calculus with Explicit Flattening

- Lambda Calculus
- **Explicit Flattening**
- Developments
- Conclusion

Formal definition

Formally, just add the rule of *Comp* to *Lam*:

$$\widehat{Lam}F := \lambda X \lambda A. LamF + X(XA)$$

and call its least pre-fixed point $\widehat{Lam}$.

As data constructors, we need

$$\begin{aligned}\widehat{var} &: \forall A. A \rightarrow \widehat{Lam} A \\ \widehat{app} &: \forall A. \widehat{Lam} A \rightarrow \widehat{Lam} A \rightarrow \widehat{Lam} A \\ \widehat{abs} &: \forall A. \widehat{Lam} (1 + A) \rightarrow \widehat{Lam} A \\ \widehat{flat} &: \forall A. \widehat{Lam} (\widehat{Lam} A) \rightarrow \widehat{Lam} A\end{aligned}$$

Formal definition

Formally, just add the rule of *Comp* to *Lam*:

$$\widehat{Lam}F := \lambda X \lambda A. LamF + X(XA)$$

and call its least pre-fixed point $\widehat{Lam}$.

As data constructors, we need

$$\begin{aligned}\widehat{var} &: \forall A. A \rightarrow \widehat{Lam} A \\ \widehat{app} &: \forall A. \widehat{Lam} A \rightarrow \widehat{Lam} A \rightarrow \widehat{Lam} A \\ \widehat{abs} &: \forall A. \widehat{Lam} (1 + A) \rightarrow \widehat{Lam} A \\ \widehat{flat} &: \forall A. \widehat{Lam} (\widehat{Lam} A) \rightarrow \widehat{Lam} A\end{aligned}$$

Informal understanding

$\widehat{flat}$ takes a term of type $\widehat{Lam}(\widehat{Lam} A)$, hence a term whose variables are themselves terms but which are **blocked**. If such a variable is x , one could write $x\{x := r\}$ for the explicit substitution of r for x in x . Since this calculus only allows this restricted form of explicit substitution, we write $\{r\}$ instead.

Example

$\lambda y. y \{ \lambda z. z x_1 \} \{ x_2 \}$ can be represented by a term of type $\widehat{Lam}(1 + 1)$ through an intermediary term of type $\widehat{Lam}(1 + \widehat{Lam}(1 + 1))$.

Informal understanding

$\widehat{flat}$ takes a term of type $\widehat{Lam}(\widehat{Lam} A)$, hence a term whose variables are themselves terms but which are blocked. If such a variable is x , one could write $x\{x := r\}$ for the explicit substitution of r for x in x . Since this calculus only allows this restricted form of explicit substitution, we write $\{r\}$ instead.

Example

$\lambda y. y \{ \lambda z. z x_1 \} \{ x_2 \}$ can be represented by a term of type $\widehat{Lam}(1 + 1)$ through an intermediary term of type $\widehat{Lam}(1 + \widehat{Lam}(1 + 1))$.

A predicative implementation in Coq

- The construction follows a suggestion by Anton Setzer and a manuscript by Peter Aczel but now adds the algorithmic aspects.
- The idea is to trace how the argument to $\widehat{Lam}$ is transformed during the recursive calls: It can either be augmented by 1, or $\widehat{Lam}$ can be applied (in the case of explicit flattening).
- So we start with a family of inductive types indexed by the initial type A (in a predicative manner since it is just a fixed parameter) and also indexed by a finite list of booleans that encode the type actually given to $\widehat{Lam}$.

A predicative implementation in Coq

- The construction follows a suggestion by Anton Setzer and a manuscript by Peter Aczel but now adds the algorithmic aspects.
- The idea is to trace how the argument to $\widehat{Lam}$ is transformed during the recursive calls: It can either be augmented by 1, or $\widehat{Lam}$ can be applied (in the case of explicit flattening).
- So we start with a family of inductive types indexed by the initial type A (in a predicative manner since it is just a fixed parameter) and also indexed by a finite list of booleans that encode the type actually given to $\widehat{Lam}$.

A predicative implementation in Coq

- The construction follows a suggestion by Anton Setzer and a manuscript by Peter Aczel but now adds the algorithmic aspects.
- The idea is to trace how the argument to $\widehat{Lam}$ is transformed during the recursive calls: It can either be augmented by 1, or $\widehat{Lam}$ can be applied (in the case of explicit flattening).
- So we start with a family of inductive types indexed by the initial type A (in a predicative manner since it is just a fixed parameter) and also indexed by a finite list of booleans that encode the type actually given to $\widehat{Lam}$.

A predicative implementation in Coq, contd.

The actual inductive definition is as follows:

```
Definition index := list bool.
```

```
Inductive L (A:Set) : index -> Set :=
```

```
  | initL: A -> L A nil
```

```
  ...
```

```
  | absL: forall i:index, L A (true::false::i) -> L A (true::i)
```

```
  | exsL: forall i:index, L A (true::true::i) -> L A (true::i).
```

Then the slice for index $\text{true}::\text{nil}$ qualifies as $\widehat{Lam}$.

For this, one has to show $L (L A i2) i1 \rightarrow L A (i1++i2)$ and monotonicity of the whole construction in the argument A . By help of these, the data constructors can be **defined**.

Outline

- 1 Nested Datatypes
 - Heterogeneous Datatypes
 - True Nesting
 - Impredicative Encoding
- 2 Case Study: Lambda Calculus with Explicit Flattening
 - Lambda Calculus
 - Explicit Flattening
 - **Developments**
 - Conclusion

Recalling developments

One maximal β -development in untyped λ -calculus:

$$\begin{aligned} dev\ x &:= x \\ dev\ (\lambda x. r) &:= \lambda x. dev\ r \\ dev\ ((\lambda x. r)\ s) &:= (dev\ r)[x := dev\ s] \\ dev\ (r\ s) &:= (dev\ r)(dev\ s) \quad \text{otherwise} \end{aligned}$$

We do not want to execute the substitution immediately but use an explicit substitution. So we define a function $devel$ of type $\forall A. \widehat{Lam} A \rightarrow \widehat{Lam} A$ and then use a function $eval$ of type $\forall A. \widehat{Lam} A \rightarrow Lam\ A$ that resolves the explicit substitutions.

Recalling developments

One maximal β -development in untyped λ -calculus:

$$\begin{aligned} dev\ x &:= x \\ dev\ (\lambda x. r) &:= \lambda x. dev\ r \\ dev\ ((\lambda x. r)\ s) &:= (dev\ r)[x := dev\ s] \\ dev\ (r\ s) &:= (dev\ r)(dev\ s) \quad \text{otherwise} \end{aligned}$$

We do not want to execute the substitution immediately but use an explicit substitution. So we define a function *devel* of type $\forall A. \widehat{Lam} A \rightarrow \widehat{Lam} A$ and then use a function *eval* of type $\forall A. \widehat{Lam} A \rightarrow Lam\ A$ that resolves the explicit substitutions.

A glance at the definition in Coq

The definition in Coq is as follows (thanks to Yves Bertot):

```
Fixpoint devel (A:Set)(t:Lam A){struct t} : Lam' A :=
  match t in Lam A' return Lam' A' with
  | var _ a => var' a
  | abs _ r => abs' (devel r)
  | app _ r s =>
    let r' := devel r in let s' := devel s in
    match r in Lam A'' return Lam' A'' -> Lam' A'' -> Lam' A''
    | abs _ r0 => fun _ y => let r0' := devel r0 in
      esubst r0' (fun z: _ => match z with
        | inl _ => y
        | inr a => var' a
        end)
    | _ => fun x y => (app' x y)
    end r' s'
  end.
```

Experiments with program extraction

- Astonishingly, the construction computes the normal forms for exponentiation of Church numerals.
- Space and time considerations as well as the demand for interoperability with algorithms from other sources dictate the extraction into functional programming languages that are optimized for those purposes.
- The program extraction facility of Coq still needs the use of recursive type equations in Ocaml and sometimes (e. g., for the function `devel`) it needs an explicit type annotation—despite the fact that already plenty of unsafe coerces take place.

Experiments with program extraction

- Astonishingly, the construction computes the normal forms for exponentiation of Church numerals.
- Space and time considerations as well as the demand for interoperability with algorithms from other sources dictate the extraction into functional programming languages that are optimized for those purposes.
- The program extraction facility of Coq still needs the use of recursive type equations in Ocaml and sometimes (e. g., for the function `devel`) it needs an explicit type annotation—despite the fact that already plenty of unsafe coerces take place.

Experiments with program extraction

- Astonishingly, the construction computes the normal forms for exponentiation of Church numerals.
- Space and time considerations as well as the demand for interoperability with algorithms from other sources dictate the extraction into functional programming languages that are optimized for those purposes.
- The program extraction facility of Coq still needs the use of recursive type equations in Ocaml and sometimes (e. g., for the function `devel`) it needs an explicit type annotation—despite the fact that already plenty of unsafe coerces take place.

Outline

- 1 Nested Datatypes
 - Heterogeneous Datatypes
 - True Nesting
 - Impredicative Encoding
- 2 Case Study: Lambda Calculus with Explicit Flattening
 - Lambda Calculus
 - Explicit Flattening
 - Developments
 - Conclusion

Insights gained from the case study

- Strictly positive **dependent** families of inductive types can implement truly nested datatypes.
- The obtained implementation in Coq does not use rewrite rules but just definitional equality. (Is there a support of definitional equality in the module system of Coq?)
- Nested dependent pattern matching requires a lot of thought from the Coq user.
- Program extraction into a typed programming language requires plenty of forgiveness from the type-checker. But it should trust in the reasoning system of the proof assistant from where the code originates.

Insights gained from the case study

- Strictly positive dependent families of inductive types can implement truly nested datatypes.
- The obtained implementation in Coq does not use rewrite rules but just **definitional equality**. (Is there a support of definitional equality in the module system of Coq?)
- Nested dependent pattern matching requires a lot of thought from the Coq user.
- Program extraction into a typed programming language requires plenty of forgiveness from the type-checker. But it should trust in the reasoning system of the proof assistant from where the code originates.

Insights gained from the case study

- Strictly positive dependent families of inductive types can implement truly nested datatypes.
- The obtained implementation in Coq does not use rewrite rules but just definitional equality. (Is there a support of definitional equality in the module system of Coq?)
- **Nested** dependent pattern matching requires a lot of thought from the Coq user.
- Program extraction into a typed programming language requires plenty of forgiveness from the type-checker. But it should trust in the reasoning system of the proof assistant from where the code originates.

Insights gained from the case study

- Strictly positive dependent families of inductive types can implement truly nested datatypes.
- The obtained implementation in Coq does not use rewrite rules but just definitional equality. (Is there a support of definitional equality in the module system of Coq?)
- Nested dependent pattern matching requires a lot of thought from the Coq user.
- Program extraction into a typed programming language requires plenty of forgiveness from the type-checker. But it should trust in the reasoning system of the proof assistant from where the code originates.

Things still to do

- a general explanation of the method for all nested datatypes
- proof of soundness of the obtained iteration principles

Thank you. Merci.

Typeset with $\text{\LaTeX}$ using Till Tantau's `beamer.cls`

Author: Conor McBride

Title: Standard Equipment for Inductive Datatypes

Abstract:

Every inductive datatype can be equipped with useful (if unremarkable) theorems by following some standard 'recipes': constructors are injective and disjoint; constructor-guarded recursion is admissible; cyclic equations are false. Given a heterogeneous equality, these recipes extend straightforwardly to indexed families of datatypes (in a way that 'equality-respects-predecessor' proofs of injectivity do not, for example). This technology is five years old, so it is time that I explained it properly.

Author: Olov Wilander

Title: An E-bicategory of E-categories

Abstract:

A type-theoretic formalisation of bicategories is introduced, and it is shown that small E-categories, together with their functor categories, form such an E-bicategory. This is carried out using only basic recursive definitions, in the version of predicative type theory with a hierarchy of universes implemented by Agda. This relates to earlier work by Huet and Saïbi, who constructed a large category of small categories in Coq, but with the use of inductive families. The construction presented here may be considered more natural, particularly from the point of view of higher-dimensional category theory.



UPPSALA
UNIVERSITET

An E-bicategory of E-categories

K.O. Wilander

Department of Mathematics

Uppsala University



Slogan: Objects are not equal, but isomorphic

- E-categories support this idea
- Leads to higher-dimensional category theory
- Can't have the category of categories



Using Agda:

- Formalized bicategories
- Formalized the bicategory of categories
- No inductive families



Recall: A bicategory B consists of

- A collection of objects
- For every pair a, b of objects, a hom-category $B(a, b)$
- Composition functors $B(b, c) \times B(a, b) \longrightarrow B(a, c)$
 - associative up to given natural isomorphism
- Identity functors $1 \longrightarrow B(a, a)$
 - with given “unit morphisms”

and satisfies some axioms.



Problems when formalizing:

- With a little help, things easily get out of hand.
 - A failed attempt.
 - Close attention needed to all details
- Very long formalization
 - Hidden arguments may help



An oddity:

```
RevElimLeftIdfunctorNatTra (C::E_category)(D::E_category)(F::E_functor C D)
  :: E_natural_transformation C D
  F (E_functorcomposition C D D (E_idfunctor D) F)
= struct
  arrows      = (ElimLeftIdfunctorNatTra C D F).arrows
  ax_naturality = (ElimLeftIdfunctorNatTra C D F).ax_naturality
```

works, but

```
RevElimLeftIdfunctorNatTra (C::E_category)(D::E_category)(F::E_functor C D)
  :: E_natural_transformation C D
  F (E_functorcomposition C D D (E_idfunctor D) F)
= ElimLeftIdfunctorNatTra C D F
```

gives an error.

Author: Bas Spitters

Title: Pointfree Integration Theory without Choice

Abstract:

In this talk I will present a choice-free and point-free development of parts of Bishop's integration theory, culminating in an easy proof of the spectral theorem. This proof is based on Coquand's constructive version of the Stone representation theorem combined with some elementary theory of f-rings.

Point-free Integration Theory without Choice

Bas Spitters

TYPES 2004

Choice

Several proposals to avoid (countable) choice in constructive mathematics.

Richman:

- Black box interpretation of $\exists$
 - Abstract data types,
can be modeled (impredicatively) by existential types
(Mitchell, Plotkin)
- Local continuous choice
 - sheaf models

Choice

Several proposals to avoid (countable) choice in constructive mathematics.

Richman:

- Black box interpretation of $\exists$
 - Abstract data types,
can be modeled (impredicatively) by existential types
(Mitchell, Plotkin)
- Local continuous choice
 - sheaf models

More importantly, the mathematics gets better.

Choice

Several proposals to avoid (countable) choice in constructive mathematics.

Richman:

- Black box interpretation of $\exists$
 - Abstract data types,
can be modeled (impredicatively) by existential types
(Mitchell, Plotkin)
- Local continuous choice
 - sheaf models

More importantly, the mathematics gets more uniform.

Choice

Several proposals to avoid (countable) choice in constructive mathematics.

Richman:

- Black box interpretation of $\exists$
 - Abstract data types,
can be modeled (impredicatively) by existential types
(Mitchell, Plotkin)
- Local continuous choice
 - sheaf models

More importantly, the mathematics gets more uniform.

E.g. Richman's proof of the fundamental theorem of algebra.

Richman's challenge

Richman

Measure theory and the spectral theorem are major challenges for a choiceless development of constructive mathematics and I expect a choiceless development of this theory to be accompanied by some surprising insights and a gain of clarity.

Richman's challenge

Richman

Measure theory and the spectral theorem are major challenges for a choiceless development of constructive mathematics and I expect a choiceless development of this theory to be accompanied by some surprising insights and a gain of clarity.

To address this I will use a pointfree approach.

Point Free Topology

Choice is used to construct *ideal* points (real numbers, max. ideals).

Avoiding points one can avoid choice and non-constructive reasoning.

Point Free Topology

Choice is used to construct *ideal* points (real numbers, max. ideals).

Avoiding points one can avoid choice and non-constructive reasoning.

Point free approaches to topology:

- Formal topology/ locale theory (formal opens)
- C^* -algebras (formal continuous functions)

These formal objects model basic observations

Point Free Topology

Choice is used to construct *ideal* points (real numbers, max. ideals).

Avoiding points one can avoid choice and non-constructive reasoning.

Point free approaches to topology:

- Formal topology/ locale theory (formal opens)
- C^* -algebras (formal continuous functions)

These formal objects model basic observations

Similar approaches for integration theory

Algebraic Integration Theory

Kolmogorov:

Set-theoretical measure theory not satisfactory:

- σ -additivity
- Probability zero does not imply absolute impossibility

Algebraic Integration Theory

Kolmogorov:

Set-theoretical measure theory not satisfactory:

- σ -additivity
- Probability zero does not imply absolute impossibility

Start with Boolean algebra of sets which is σ -complete

Algebraic Integration Theory

Kolmogorov:

Set-theoretical measure theory not satisfactory:

- σ -additivity
- Probability zero does not imply absolute impossibility

Start with Boolean algebra

Algebraic Integration Theory

Kolmogorov:

Set-theoretical measure theory not satisfactory:

- σ -additivity
- Probability zero does not imply absolute impossibility

Start with Boolean algebra

Approaches are equivalent classically

Algebraic Integration Theory

Kolmogorov:

Set-theoretical measure theory not satisfactory:

- σ -additivity
- Probability zero does not imply absolute impossibility

Start with Boolean algebra

Approaches are equivalent classically

cf. formal topology, locale theory

Start with Heyting algebra of open sets

Algebraic Integration Theory

Kolmogorov:

Set-theoretical measure theory not satisfactory:

- σ -additivity
- Probability zero does not imply absolute impossibility

Start with Boolean algebra

Approaches are equivalent classically

cf. formal topology, locale theory

Start with Heyting algebra of opens

Boolean Algebra

Coquand/Palmgren:

Boolean algebra $\mathcal{A}$ of basic observables with a measure μ .

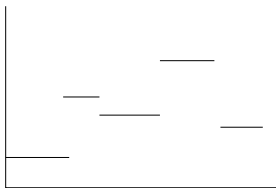
The measure defines a metric d_μ on $\mathcal{A}$.

Complete $(\mathcal{A}, d_\mu)$ to obtain a complete measure space.

[metric completion instead of σ -completion]

Boolean Algebra

Define the integral on the *simple* functions



$$\int (\sum c_i \chi_{A_i}) := \sum c_i \mu(A_i)$$

Complete wrt the norm $\int |f|$.

Riemann/Lebesgue integral is obtained from the Boolean algebra of rational intervals and $\mu((a, b)) := |b - a|$.

Bishop's integration theory

Daniell's functional analytic approach to integration theory:
Start with pos linear functional $\int$ on $C(X)$. X locally compact

Bishop's integration theory

Daniell's functional analytic approach to integration theory:

Start with pos linear functional $\int$ on $C(X)$. X locally compact

Constructive version by Bishop:

Complete wrt the norm $\int |f|$

Lebesgue-integral is the completion of the Riemann integral.

Bishop's integration theory

Daniell's functional analytic approach to integration theory:

Start with pos linear functional $\int$ on $C(X)$. X locally compact

Constructive version by Bishop:

Complete wrt the norm $\int |f|$

Lebesgue-integral is the completion of the Riemann integral.

One obtains L_1 as the completion of $C(X)$.

$$\begin{array}{ccc} C(X) & \rightarrow & \mathcal{L}_1 \\ & & \downarrow \\ & & L_1 \end{array}$$

Bishop's integration theory

Daniell's functional analytic approach to integration theory:

Start with pos linear functional $\int$ on $C(X)$. X locally compact

Constructive version by Bishop:

Complete wrt the norm $\int |f|$

Lebesgue-integral is the completion of the Riemann integral.

One obtains L_1 as the completion of $C(X)$.

$$\begin{array}{ccc} C(X) & \rightarrow & \mathcal{L}_1 \\ & \searrow & \downarrow \\ & & L_1 \end{array}$$

Integration f-algebras

We generalize Bishop/Cheng and the Boolean algebra approach: towards integration f-algebras (formal functions)

Definition

An *f-algebra* is a vector lattice (Riesz space) which has an algebra structure compatible with the lattice space structure. That is, $fg \geq 0$, whenever $f, g \geq 0$, and if $f \wedge g = 0$, then $hf \wedge g = 0$, for all h .

One can prove that $a^2 \geq 0$, $|ab| = |a||b|$ and that if $a \wedge b = 0$, then $ab = 0$.

Example: algebras of (continuous) functions

Integration algebras

Segal's integration algebras

Definition

An *integration algebra* is a real Abelian algebra A with a linear functional I

- $I(a^2) \geq 0$ and $I(a^2) = 0$ iff $a = 0$.
- All elements are bounded: For all $b \in A$ there exists $\alpha \in \mathbf{R}$ such that $I(ba^2) \leq \alpha I(a^2)$ for all $a \in A$.

Algebra of simultaneously observable quantities.

$I(a)$ is the expectation an event a .

Integration algebras

Segal's integration algebras

Definition

An *integration algebra* is a real Abelian algebra A with a linear functional I

- $I(a^2) \geq 0$ and $I(a^2) = 0$ iff $a = 0$.
- All elements are bounded: For all $b \in A$ there exists $\alpha \in \mathbf{R}$ such that $I(ba^2) \leq \alpha I(a^2)$ for all $a \in A$.

Algebra of simultaneously observable quantities.

$I(a)$ is the expectation an event a .

These can be used to give a new and constructive proof of the Peter-Weyl theorem for the representation of compact groups. (Coquand/Spitters).

Integration f-algebras

Generalizing Bishop's definition.

Definition

An *integral* I on an f-algebra is a positive linear functional I on L such that $I(a \wedge n) \rightarrow I(a)$ and $I(a \wedge 1/n) \rightarrow 0$ when n tends to ∞ . An f-algebra with an integral we call an integration f-algebra.

Every integration f-algebra with a strong unit is an integration algebra.

Examples

Simple functions on a Boolean algebra, $(C(X), f)$, $(L_1, f), \dots$

Profile theorem

Most of Bishop's results generalize to integration f-algebras!

Profile theorem

Most of Bishop's results generalize to integration f-algebras!

But we need a substitute of the profile theorem.

A positive variant of: 'every increasing function has at most countably many discontinuities.'

It implies that integrable functions can be approximated by simple functions. This is used for instance to prove that $\phi \circ f$ is integrable, when ϕ is a test function.

Profile theorem

Most of Bishop's results generalize to integration f-algebras!

But we need a substitute of the profile theorem.

A positive variant of: 'every increasing function has at most countably many discontinuities.'

It implies that integrable functions can be approximated by simple functions. This is used for instance to prove that $\phi \circ f$ is integrable, when ϕ is a test function.

However,

Profile theorem

Most of Bishop's results generalize to integration f-algebras!

But we need a substitute of the profile theorem.

A positive variant of: 'every increasing function has atmost countably many discontinuities.'

It implies that integrable functions can be approximated by simple functions. This is used for instance to prove that $\phi \circ f$ is integrable, when ϕ is a test function.

However,
 it also implies that the reals are uncountable.

Profile theorem

Most of Bishop's results generalize to integration f-algebras!

But we need a substitute of the profile theorem.

A positive variant of: 'every increasing function has at most countably many discontinuities.'

It implies that integrable functions can be approximated by simple functions. This is used for instance to prove that $\phi \circ f$ is integrable, when ϕ is a test function.

However,
 it also implies that the reals are uncountable.
 We use Stone representation as a substitute.

Stone representation

Stone representation theorem:

'Every f-algebra can be embedded in a **space** of continuous functions.'

Used in two ways:

- Substitute for the profile theorem
- Towards spectral theorem

Stone representation

Three interpretations of this space.

- ① as in ordinary space, for which we need classical logic and the axiom of choice (Stone);
- ② as a point-free space, which can be done constructively without the axiom of countable choice (Coquand);
- ③ as a compact metric space, which can be done in Bishop's constructive mathematics when R is a separable ring of normable operators on a separable Hilbert space (Bishop).

We will later see how 3 follows from 2.

Applying Stone

Let (A, I) be an integration f-algebra

By Coquand's Stone representation theorem:

A may be embedded into a formal space of continuous functions $C(X)$ and I can be extended to $C(X)$.

Bishop's spectral theorem

Bishop's spectral theorem defines an embedding of a space of **bounded integrable functions** into a space of operators.

The Stone representation theorem can be used to define such an embedding for **continuous functions**.

Stone's theorem can be used to prove Bishop's spectral theorem.

Bishop's spectral theorem

The proof is based on a simple observation:

Theorem

Let (A, I) be an f -algebra and an integration algebra. Suppose that $I(ab^2) \geq 0$, for all $b \in A$. Then a is positive in the f -algebra A .

Consequently, $(\mathcal{A}, I)$ is an integration f -algebra.

Bishop's spectral theorem now follows from Stone's theorem, where the previous theorem is used to show that the bound is preserved.

Summary

- Observational mathematics
 - Topology
 - Measure theory
- Integration f-algebras (meeting Richman's challenge).
 - 'functions' instead of 'opens'
 - Most of Bishop's results can be generalized to this setting!
- New (easier) proof of Bishop's spectral theorem using Coquand's Stone representation theorem (formal topology)

Authors: Maria Emilia Maietti, Silvio Valentini

Title: Relative formal topology

Abstract:

The concept of formal topology introduced by P. Martin-Lof and G. Sambin to deal with topology in a constructive way, that is in intuitionistic type theory, is based on a binary cover relation and a positivity predicate.

To deal predicatively with closed subsets of a topological space, G. Sambin extended formal topology with a binary positivity predicate.

In this talk we consider formal topology with a binary positivity predicate furtherly extended with an indexed cover, called "relative formal topology", in order to represent the topology induced on a closed subset and we connect the corresponding category with that of topological spaces and that of formal topologies via categorical adjunctions.

Author: Freek Wiedijk

Title: Mapping the Classical Type Theory of Farmer's STTWU into the Constructive Type Theory of COQ

Abstract:

When one considers the foundations of a classical proof assistant like HOL, it has various "features" that one doesn't have in a constructive framework: (1) the law of the excluded middle, (2) extensional equality, and (3) a choice operator (which for any predicate returns an object that satisfies the predicate, if there is one.) To model such a system in a constructive world it seems that one cannot avoid having to add "axioms" to obtain these features.

However, for the theory - introduced by William Farmer - called "STTwU" (simple type theory with undefinedness), this turns out not to be the case. This theory is classical, extensional, and it has a choice operator, but it can be mapped into a constructive world without having to add any axioms.

In this talk we will both present a LF context corresponding to STTwU, as well inhabit that context in Coq. We use a stack of tricks to do this: a double negation translation for the classical logic, setoids for the extensionality, `_partial_` setoids to deal with undefinedness, and finally dual setoids for the choice operator.

Our work shows that we can have the classical world of STTwU and still consider our work meaningful to constructive mathematicians.

STTwU in Coq

Freek Wiedijk

Radboud Universiteit, Nijmegen

Free University, Amsterdam

TYPES 2004

2004 12 15, 15:10

overview

the three points of this talk

- advertisement for Farmer's STTwU system
- report on a small Coq formalization
- how to map classical mathematics into constructive mathematics

isn't that **just** the double negation translation?

no:

extensionality of equality?

choice?

classical higher order logic in Coq

HOL

simply typed lambda calculus + ML-style polymorphism

type constants: `bool`, `ind`, $\rightarrow$

term constants: `=`, ε

$$\sqrt[n]{x} := \varepsilon y. (y^n = x \wedge (0 < x \Rightarrow 0 < y))$$

importing into Coq: two axioms **needed**

classical logic `forall A:Prop, not (not A) -> A`

choice operator `forall A:Set, not (notT A) -> A`

extensionality `forall (A B:Set) (f g:A -> B),
 (forall x:A, f x = g x) -> f = g`

STTwU

Bill Farmer IMPS system $\rightarrow$ LUTINS logic

Logic of Undefined Terms for Inference in a Natural Style

Simple Type Theory with Undefinedness

simply typed lambda calculus + undefined terms

type constants: `bool`, `ind`, $\rightarrow$

term constants: `=`, ι

choice operator becomes partial

$$(\iota x. Px) \downarrow \Leftrightarrow \exists! x. Px$$

`bool` is special: all terms of type `bool` are defined

the STTwU system as an LF context

```
Parameter _type : Type.  
Parameter _expr : _type -> Type.  
  
Parameter _bool : _type.  
Parameter _ind : _type.  
Parameter _fun : _type -> _type -> _type.  
  
Parameter _infer : _expr _bool -> Type.  
  
Parameter _app : forall alpha beta:_type,  
  _expr (_fun alpha beta) -> (_expr alpha -> _expr beta).  
Parameter _lambda : forall alpha beta:_type,  
  (_expr alpha -> _expr beta) -> _expr (_fun alpha beta).  
  
Parameter _eq : forall alpha:_type,  
  _expr alpha -> (_expr alpha -> _expr _bool).  
Parameter _iota : forall alpha:_type,  
  (_expr alpha -> _expr _bool) -> _expr alpha.
```

etcetera

Coq formalization

a constructive model for STTwU in Coq

classical logic	double negation translation
extensionality	setoids
partial logic	partial setoids
choice operator	dual of partial setoid

no axioms needed!

a way to do the double negation translation

classical propositions = copy of the **constructive propositions**

Definition __bool := Prop.

... but interpreted **negatively**

Definition __true : __bool := False.

Definition __false : __bool := True.

Definition __implies (A B : __bool) : __bool := ~(B -> A).

Definition _infer (A : __bool) : Type := ~A.

Definition __not (A : __bool) : __bool := __implies A __false.

Lemma __notnot_elim:

forall A : __bool, _infer (__not (__not A)) -> _infer A.

the dual of a partial setoid

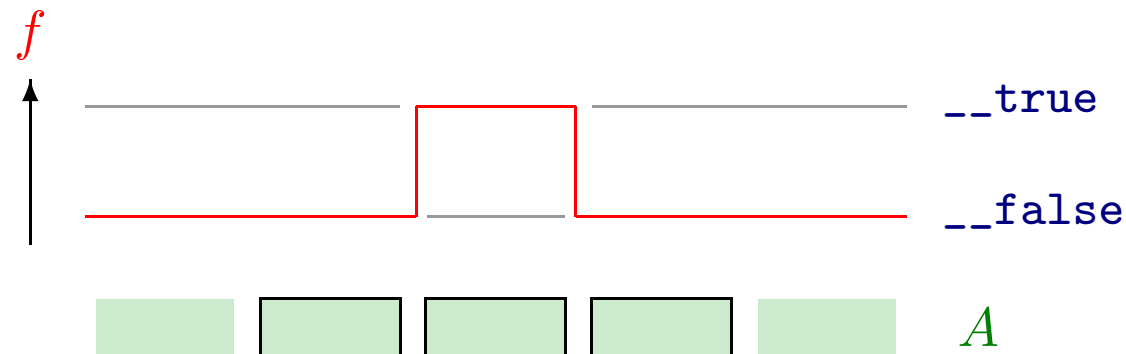
- **partial setoid**

$(A, \sim)$ where $\sim$ is a **partial** equivalence relation on A

a is 'proper element' when $a \sim a$

- **dual**

partial setoid of functions $f : A \rightarrow \text{\_\_bool}$



$$f \sim g \Leftrightarrow \exists a. f(a) \wedge (\forall b. f(b) \Leftrightarrow a \sim b) \wedge (\forall b. g(b) \Leftrightarrow a \sim b)$$

$$\iota f. P(f) = \lambda a. (\exists f. P(f)) \wedge (\forall f. P(f) \Rightarrow (\forall b. f(b) \Leftrightarrow a \sim b))$$

... and can you really not do better than this?

Coq formalization heavily uses impredicativity of Prop

questions:

- is it possible to do the same in a predicative system?

Agda

NuPRL

- is it possible to do this for another Farmer system: STMM

Set Theory for Mechanized Mathematics

axiom NBG31 = the axiom of choice ?

maybe work in: Gödel's constructible universe L

Author: David Wahlstedt

Title: Type Theory with First-Order Data Types and Size-Change Termination

Keywords: Type Theory, Dependent types, Lambda-calculus, Normalization, Size-Change Termination, Type system, Logical Framework, Reducibility, Pattern-matching, Term rewriting.

Abstract:

We prove normalization for a dependently typed lambda-calculus extended with first-order data types and computation schemata for first-order size-change terminating recursive functions.

Size-change termination, introduced by C.S. Lee, N.D. Jones and A.M. Ben-Amram, can be seen as a generalized form of structural induction, which allows inductive computations and proofs to be defined in a straight-forward manner. The language can be used as a proof system---an extension of Martin-L"_o's Logical Framework.

Type Theory with First-Order Data Types and Size-Change Termination

TYPES 2004

David Wahlstedt

davidw@cs.chalmers.se

The **halting** problem is undecidable ...

Known characterizations of termination:

Primitive recursion

$$\begin{aligned}f(\vec{x}, \mathbf{S}(n)) &= g(f(\vec{x}, n), \vec{x}, n) \\ f(\vec{x}, \mathbf{0}) &= h(\vec{x})\end{aligned}$$

More general schema needed

$$\min(\mathbf{S}(m), \mathbf{S}(n)) = \mathbf{S}(\min(m, n))$$

$$\min(\mathbf{0}, \mathbf{S}(n)) = \mathbf{0}$$

$$\min(m, \mathbf{0}) = \mathbf{0}$$

A general approach to “Structural Termination”:

Size-Change Termination (SCT)

(C.S. Lee, N.D. Jones and A.M. Ben-Amram '01)

- More than primitive recursion
- Subsumes lexicographical termination
- Simple algorithm with non-trivial correctness proof
(Ramsey's Theorem involved)

Size-change termination criterion:

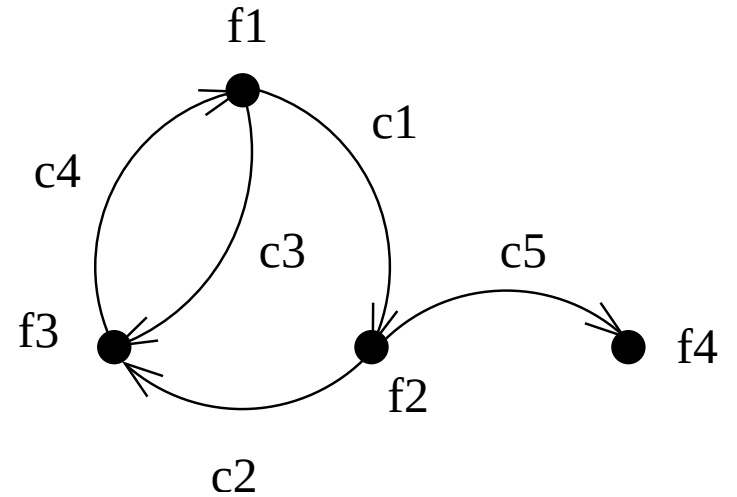
*All the infinite call sequences
contain an infinitely decreasing thread.*

Program

$$\begin{array}{rcl} f_1(p_{11}, \dots, p_{1n}) & = & a_1 \\ \vdots & \vdots & \vdots \\ f_n(p_{m1}, \dots, p_{mn}) & = & a_m \end{array}$$

$\Rightarrow$

Call Graph



For each **call**, c , to f_j in a_i there is an arc from f_i to f_j .

For each recursive call c ,
relate formal parameters and actual parameters:

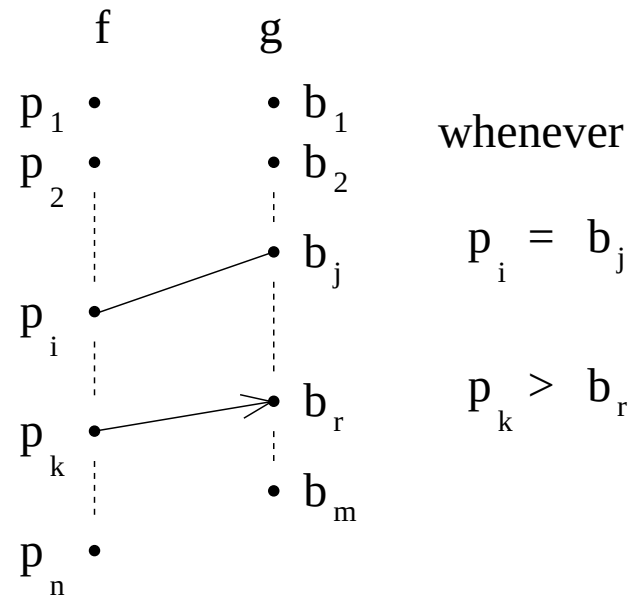
Recursive call c $\Rightarrow$ **Size-change graph G_c**

$f(p_1, \dots, p_n) =$

$\dots$

$g(b_1, \dots, b_m)$

$\dots$

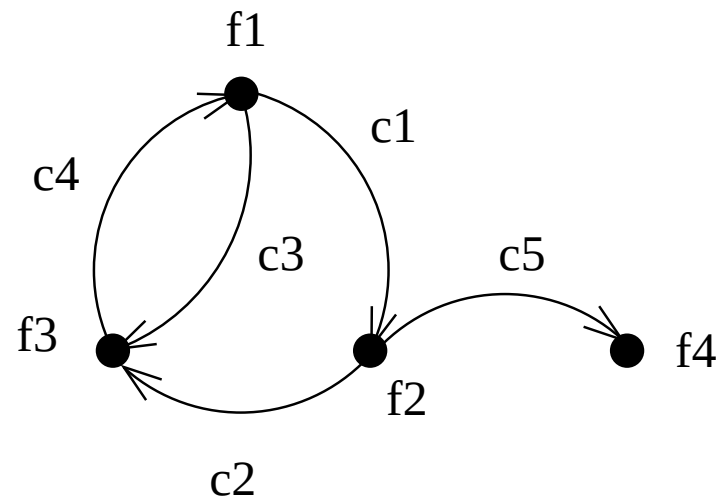


Take $>$ to be the **structural** size-change relation

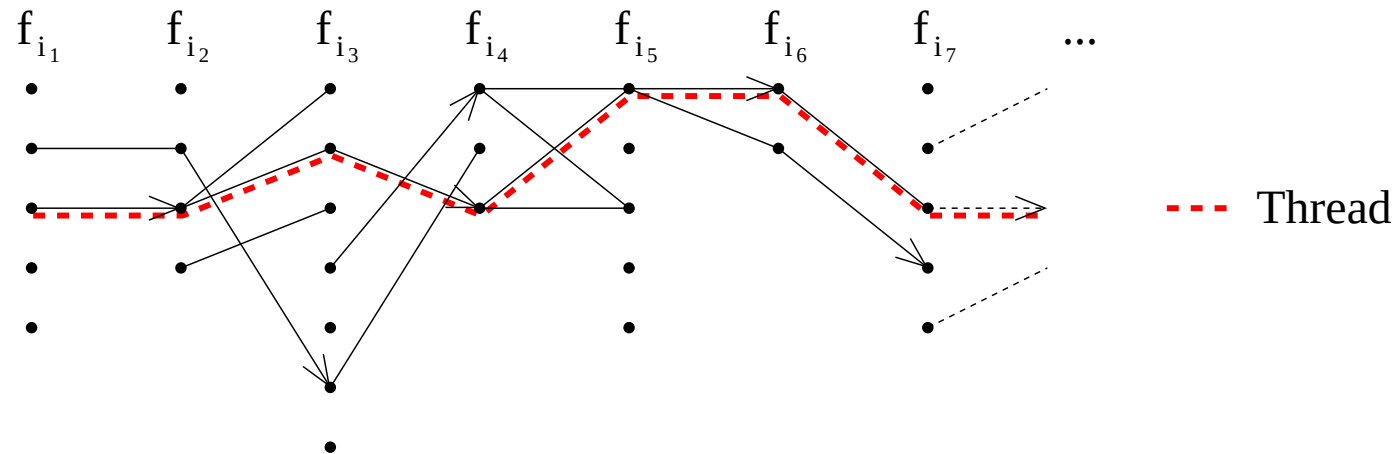
$$\frac{}{\mathbf{c}(t_1, \dots, t_n) > t_k} \qquad \frac{t_k > t}{\mathbf{c}(t_1, \dots, t_n) > t}$$

where $\mathbf{c}$ is a constructor.

Call graph

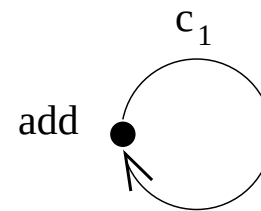


Path

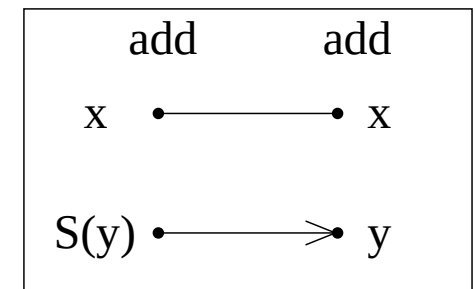


Addition

$$\begin{aligned}\text{add}(x, \mathbf{0}) &= \mathbf{0} \\ \text{add}(x, \mathbf{S}(y)) &= \mathbf{S}(\text{add}(x, y))\end{aligned}$$



Size-change graph:



because

$x = x$ and $S(y) > y$

Ackermann function (Rózsa Péter's version)

$$\text{ack}(\mathbf{0}, y) = \mathbf{S}(y)$$

$$\text{ack}(\mathbf{S}(x), \mathbf{0}) = {}^1\text{ack}(x, \mathbf{S}(\mathbf{0}))$$

$$\text{ack}(\mathbf{S}(x), \mathbf{S}(y)) = {}^2\text{ack}(x, {}^3\text{ack}(\mathbf{S}(x), y))$$

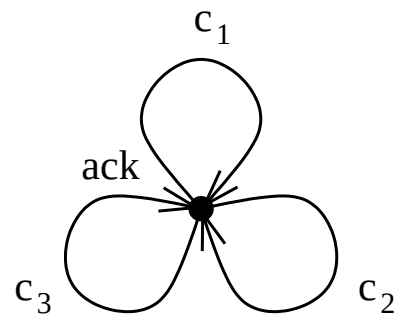
We compare parameters for each call:

$$c_1 : \mathbf{S}(x) > x$$

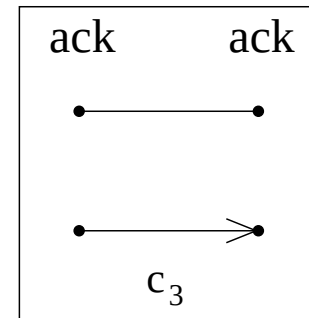
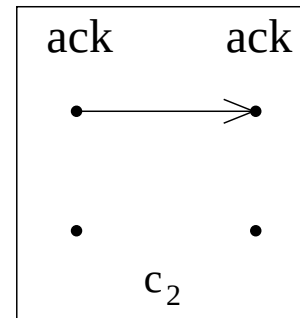
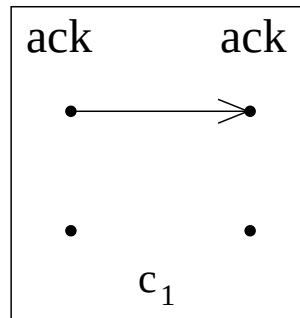
$$c_2 : \mathbf{S}(x) > x$$

$$c_3 : \mathbf{S}(x) = \mathbf{S}(x), \mathbf{S}(y) > y.$$

Call graph:



Size-change graphs:



Quicksort — non-structural

$\text{qsort}[] = []$

$\text{qsort}(x : xs) =$

$\quad {}^1\text{append}({}^2\text{qsort}({}^3\text{filter\_small}(x, xs)),$

$\quad x : {}^4\text{qsort}({}^5\text{filter\_big}(x, xs)))$

No structural decrease in calls c_2 and c_4 .

Another non-structurally terminating program:

Tree flattening — rearrange instead of decreasing.

data Tree a = **Leaf**

| **Node** a (Tree a) (Tree a)

flatten(**Leaf**) = []

flatten(**Node** a **Leaf** t) = a : ¹flatten t

flatten(**Node** a (**Node** b t_1 t_2) t_3) =
²flatten(**Node** b t_1 (**Node** a t_2 t_3))

The call c_2 has no decreasing parameter.

Let us use size-change termination in a proof system !

Martin-Löf's logical framework '86

(in Nordström, Petersson, Smith '90)

$$\textit{Terms} \quad t, u, v ::= x \mid t \ u \mid \lambda x. t$$
$$\textit{Types} \quad T, U, V ::= \text{Set} \mid \text{El } t \mid \text{Fun } T \ (\lambda x. U)$$

The notation $(x : T) \rightarrow U$ is shorthand for $\text{Fun } T \ (\lambda x. U)$.

An open system—we extend the language:

$$\begin{aligned} \textit{Terms } \quad t, u, v \quad ::= \quad & x \mid t \ u \mid \lambda x. t \\ & \mid \mathbf{c}(t_1, \dots, t_n) \mid f(t_1, \dots, t_n) \mid D \end{aligned}$$

Function constants f , constructors $\mathbf{c}$, and set codes D .

Restricted sublanguage

$$\text{FO-Terms} \quad a \quad ::= \quad x \mid \mathbf{c}(t_1, \dots, t_n) \mid f(t_1, \dots, t_n) \mid D$$

$$\text{Patterns} \quad p \quad ::= \quad x \mid \mathbf{c}(p_1, \dots, p_n)$$

$$\textit{Rewrite rules} \quad f(p_1, \dots, p_n) = a$$

A formalism for reasoning about inductively defined objects.

Judgement forms

$$\vdash U \text{ Type} \quad \vdash t : U$$

Propositions-as-types

$\vdash t : U$ interpreted as *t is a proof of the proposition U*.

Type-valued functions

$\lambda x. \text{El } x$ returns different type dependent on the argument.

We can write propositional functions by pattern-matching.

(Smith '89)

Contribution

We prove normalization for a combination of

- Martin-Löf's logical framework.
- First-order recursive pattern-matching rules, justified by size-change termination.
- First-order mutually recursive data-types.

- The new definitions are stored in a *Signature* $(\mathcal{D}, \mathcal{F}, \mathcal{R})$
 - $\mathcal{D}$ Mutually recursive first-order data type definition.
All data types are given simultaneously.
 - $\mathcal{F}$ Function constants with typing

$$f : (x_1 : \text{El } a_1, \dots, x_n : \text{El } a_n) \rightarrow A$$
with A on the form $\text{El } a \mid \text{Set}$.
 - $\mathcal{R}$ Collection of mutually recursive pattern-matching
rules $f(p_1, \dots, p_n) = a$ for the constants in $\mathcal{F}$.
- Valid signature—correct types in $\mathcal{F}$
and well-typed computation rules in $\mathcal{R}$.

Theorem

If

1. The pattern-matching rules in $\mathcal{R}$ satisfies SCT,
2. the signature is valid,
3. and $() \vdash t : T$,

then t has a normal form.

A proof about sub-typing:

We define the propositional function

$$\text{subType} : (x_1 : \text{El Num}, x_2 : \text{El Num}) \rightarrow \text{El Bool}$$

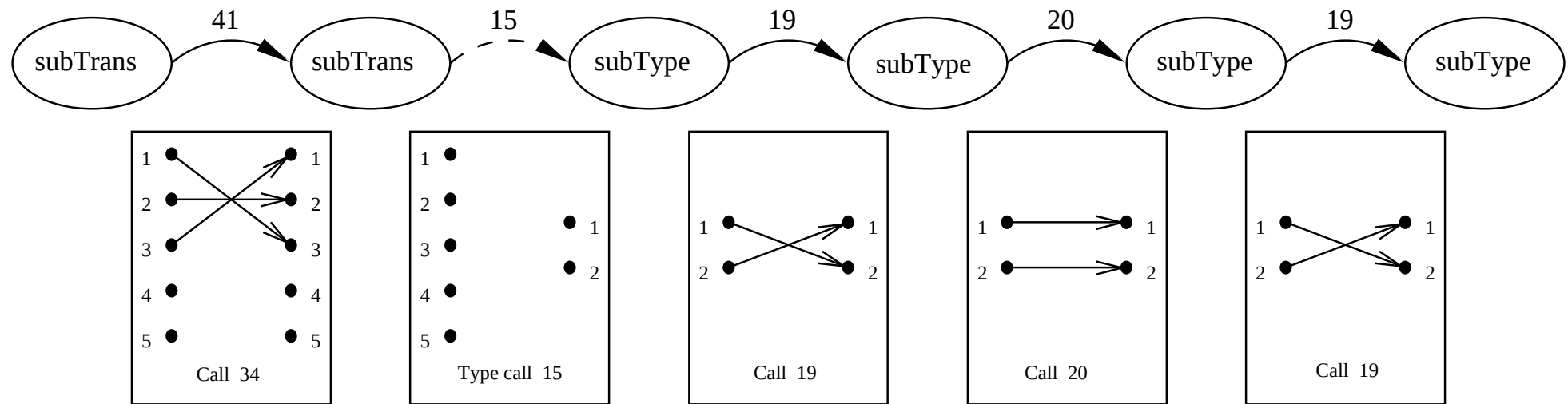
the recursive case is

$$\begin{aligned} \text{subType}(\mathbf{Arrow}(x_1, x_2), \mathbf{Arrow}(y_1, y_2)) = \\ \text{and}(\text{subType}(y_1, x_1), \text{subType}(x_2, y_2)) \end{aligned}$$

We can prove that the relation is transitive using the constant,

$$\begin{aligned} \text{subTrans} : & (x : \text{El Num}, y : \text{El Num}, z : \text{El Num}, \\ & h_1 : \text{El (T(subType}(x, y)))}, \\ & h_2 : \text{El (T(subType}(y, z)))} \\ &) \rightarrow \text{El (T(subType}(x, z)))} \end{aligned}$$

Path:



Thread: $2 \xrightarrow{>} 1 \xrightarrow{>} 1 \xrightarrow{>} 2$

Author: Johan Georg Granström

Title: Polymorphism in type theory

Abstract:

A notion of implicit arguments is introduced. The resulting theory is a conservative extension of Martin-Löf's monomorphic type theory, with decidable type checking; and it allows for more implicit arguments than a corresponding notion used in the calculus of constructions. The conservativity of this extension is related to Salvesens result that the 1984 version of polymorphic type theory is not conservative over the monomorphic version.

Authors: Iris Loeb, Lionel Elie Mamane, Herman Geuvers

Title: Interactive algebra course with formalised proofs and definitions

Abstract:

To enable interoperability and intelligent searching, web-documents require a semantical mark-up. For readability, however, a nice presentation and hence a presentational mark-up is needed. With this in mind, we have replaced the proofs and definitions of a chapter of existing interactive algebra course notes (IDA by Cohen, Cuypers and Sterk) by their formal counterparts, that we had first formalised in the proof assistant Coq. To this end we have used and extended technology for the creation and maintenance of a virtual, distributed, hypertextual library of formal mathematical knowledge (HELM). This research is part of the MoWGLI project.

Interactive Algebra Course with Formalised Proofs and Definitions

Iris Loeb

Radboud University Nijmegen

Joint work with

Andrea Asperti

University of Bologna

Herman Geuvers

Radboud University Nijmegen

Lionel Elie Mamane

Radboud University Nijmegen

Claudio Sacerdoti Coen

Inria Futurs

Partially supported by the European Project IST-2001-33562 MoWGLI

Wanted Features of a Mathematical Document

- * trustworthy mathematics
- * interaction with a proof assistant
- * intelligent searching
- * readable proofs

Needed for a Mathematical Document

- * trustworthy mathematics $\rightarrow$ formal mathematics
- * interaction with a proof assistant $\rightarrow$ formal mathematics
- * intelligent searching $\rightarrow$ semantical mark-up
- * readable proofs $\rightarrow$ presentational mark-up

IDA: Interactive Document on Algebra

Cohen, Cuypers, Sterk

- * theory enriched with links
- * multiple choice exercises
- * various ways to experiment with the theory

But

- * No formal mathematics
- * Not yet a semantical mark-up

IDA: Interactive Document on Algebra

Cohen, Cuypers, Sterk

Chapter 6 Monoids and Groups

E.g.

Lemma 6. A semi-group has at most one unit.

Proof

Suppose that e and f are units of the semi-group $(S, *)$. Then we have

$$e = e * f = f.$$

C-CoRN

Constructive Coq Repository at Nijmegen

- * library of formal mathematics

- algebra

- analysis

But

- * no mark-up

- * proofs are not readable

C-CoRN

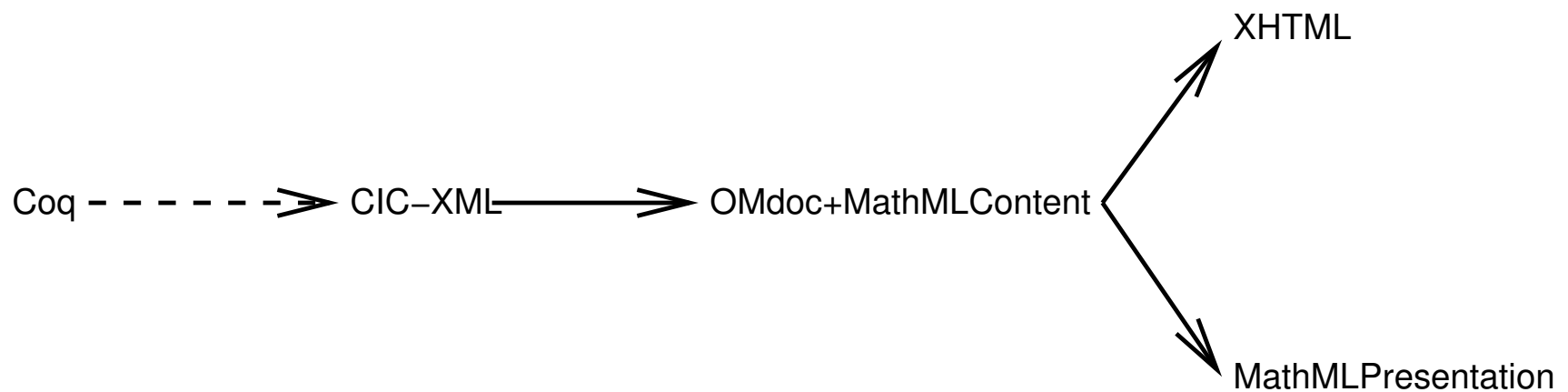
Constructive Coq Repository at Nijmegen

Lemma cs_unique_unit :
forall (S:CSemiGroup) (e f:S),
 (is_unit e) /\ (is_unit f) -> e[=]f.

Proof.
 intros S e f. intros H2 H3.
 unfold is_unit. elim (H1 e).
 intros H. clear H1.
 elim H. intros H4 H5.
 clear H. astepr (e[+]f).
 intros H0 H1. astepl (e[+]f).
 elim (H0 f). apply eq_reflexive.
 clear H0. Qed.

HELM

Hypertextual Electronic Library of Mathematics



Demo

`<http://helm.cs.kun.nl>`

Encountered Problems

- * context dependent rendering of mathematics
- * links
- * inline rendering
- * moving mathematical objects

Conclusions

- * Proof display is not satisfactory: maybe proof-trees?
- * XSLT is too weak to program in a straightforward way.

Authors: Zoran Petric, Kosta Dosen

Title: Coherence and Confluence

Abstract:

It seems that what categorists call coherence logicians would, roughly speaking, call completeness. This is a question whether we assumed for a particular brand of categories all the equations between arrows we should have assumed. The simplest coherence questions are those where it is intended that all arrows of the same type should be equal, i.e. where the category envisaged is a preorder. In such cases a technique similar to the one used in proving confluence of reductions in lambda-calculus can be applied.

Author: Daniel Winterstein

Title: Proof General / Eclipse - A generic interface for interactive theorem provers

Abstract:

Good interfaces and development tools are crucial if interactive proof systems are to be adopted by mathematicians and computer programmers. Unfortunately, the development tools available for theorem provers are often quite rudimentary. The Proof General project is an ongoing attempt to redress this. PG/Eclipse marks an exciting new phase in the project. The new interface is not only much prettier than the old Emacs based system, but also offers more sophisticated functionality. It is built using a protocol for interactive proof sessions called PGIP, and will work with any prover that implements this protocol. Currently, that means the latest version of Isabelle - but hopefully it will be relatively easy to adapt any interactive prover to use PGIP. Lucas Dixon will be demonstrating the software, showing off some new features and a wide range of bugs.

Author: David Pichardie

Title: A Modular Lattice Library to Develop Certified Terminating Static Analyses

Abstract:

We present a framework to develop and extract certified static analysis in the Coq proof assistant. Using the abstract interpretation methodology, static analyses are specified as least solution of system of equations (inequations) on lattice structures. A library of Coq modules is proposed to construct complex and efficient lattices by combination of functors. The lattice signature possesses a parameter which ensure termination of a generic fixed-point solver. The delicate problem of termination is hence solved by combination of the different lattice functors. This talk will focus on the extension of this termination property into a more general one which allow to certify more sophisticated static analyses. We will present the different proposed functors and sketch their different (constructive) proofs.

A Modular Lattice Library to Develop Certified Terminating Static Analyses

David Pichardie

IRISA / ENS Cachan, Rennes, France

Motivations

Goal : develop certified static analyses

- ▶ using the Coq proof assistant
- ▶ extracting a certified Ocaml program
- ▶ for real analyses on real programming languages

Motivations

Goal : develop certified static analyses

- ▶ using the Coq proof assistant
- ▶ extracting a certified Ocaml program
- ▶ for real analyses on real programming languages

Bottleneck : workforce cost

- ▶ we need a generic framework
- ▶ reusable for many analyses

Design of a static analysis

Two parts

Design of a static analysis

Two parts

1. specification of an abstract semantics
 - ▶ as a least fixed point of a monotone operator F , on a poset structure
 - ▶ should be proved correct with respect to the concrete semantics

Design of a static analysis

Two parts

1. specification of an abstract semantics

- ▶ as a least fixed point of a monotone operator F , on a poset structure
- ▶ should be proved correct with respect to the concrete semantics

2. solving tool

- ▶ computation of the least fixed point
- ▶ by a generic fixed point iteration

$$\perp \rightarrow F(\perp) \rightarrow F^2(\perp) \rightarrow \dots \rightarrow F^n(\perp) \xrightarrow{\text{should terminate !}} \dots \rightarrow \text{lfp}(F)$$

Design of a static analysis

Two parts

1. specification of an abstract semantics

- ▶ as a least fixed point of a monotone operator F , on a poset structure
- ▶ should be proved correct with respect to the concrete semantics

2. solving tool

This Talk

- ▶ computation of the least fixed point
- ▶ by a generic fixed point iteration

$$\perp \rightarrow F(\perp) \rightarrow F^2(\perp) \rightarrow \dots \rightarrow F^n(\perp) \rightarrow \dots \rightarrow \text{lfp}(F)$$

should terminate !

The Lattice contract

Module Type Lattice.

End Lattice.

The Lattice contract

```
Module Type Lattice.  
  Parameter t : Set.
```

```
End Lattice.
```

The Lattice contract

```
Module Type Lattice.  
  Parameter t : Set.  
  Parameter eq : t → t → Prop.  
  Parameter eq_prop : ...  
    (* eq is a computable equivalence relation *)  
  
End Lattice.
```


The Lattice contract

```
Module Type Lattice.  
  Parameter t : Set.  
  Parameter eq : t → t → Prop.  
  Parameter eq_prop : ...  
    (* eq is a computable equivalence relation *)  
  Parameter order : t → t → Prop.  
  Parameter order_prop : ...  
    (* order is a computable order relation *)  
  
End Lattice.
```

The Lattice contract

```
Module Type Lattice.  
  Parameter t : Set.  
  Parameter eq : t → t → Prop.  
  Parameter eq_prop : ...  
    (* eq is a computable equivalence relation *)  
  Parameter order : t → t → Prop.  
  Parameter order_prop : ...  
    (* order is a computable order relation *)  
  Parameter join : t → t → t.  
  Parameter join_prop : ...  
    (* join is a binary least upper bound *)  
    (* (useful to specify the analysis) *)  
  
End Lattice.
```

The Lattice contract

```
Module Type Lattice.  
  Parameter t : Set.  
  Parameter eq : t → t → Prop.  
  Parameter eq_prop : ...  
    (* eq is a computable equivalence relation *)  
  Parameter order : t → t → Prop.  
  Parameter order_prop : ...  
    (* order is a computable order relation *)  
  Parameter join : t → t → t.  
  Parameter join_prop : ...  
    (* join is a binary least upper bound *)  
    (* (useful to specify the analysis) *)  
  Parameter bottom : t.  
    (* bottom element to start iteration *)  
  Parameter bottom_is_bottom :  $\forall x : t, \text{order bottom } x$ .  
  
End Lattice.
```

The Lattice contract

```
Module Type Lattice.  
  Parameter t : Set.  
  Parameter eq : t → t → Prop.  
  Parameter eq_prop : ...  
    (* eq is a computable equivalence relation *)  
  Parameter order : t → t → Prop.  
  Parameter order_prop : ...  
    (* order is a computable order relation *)  
  Parameter join : t → t → t.  
  Parameter join_prop : ...  
    (* join is a binary least upper bound *)  
    (* (useful to specify the analysis) *)  
  Parameter bottom : t.  
    (* bottom element to start iteration *)  
  Parameter bottom_is_bottom : ∀ x : t, order bottom x.  
  Parameter termination_property : ...  
    (* to ensure iteration termination *)  
End Lattice.
```

Two termination criteria

Two termination criteria

Ascending Chain condition (LatticeWf contract)

- ▶ classical definition : “there is no infinite strictly increasing chain”

$$x_0 \sqsubset x_1 \sqsubset x_2 \sqsubset \dots \sqsubset x_n \sqsubset \dots$$

- ▶ constructive definition : $\sqsubset$ is well-founded

$$\forall x : t, \text{Acc} \sqsubset x$$

Two termination criteria

Ascending Chain condition (LatticeWf contract)

- ▶ classical definition : “there is no infinite strictly increasing chain”

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots \sqsubseteq x_n \sqsubseteq \dots$$

- ▶ constructive definition : $\sqsubseteq$ is well-founded

$$\forall x : t, \text{Acc } \sqsubseteq x$$

Widening operator (LatticeWidening contract)

- ▶ modification of the solving algorithm

$$x_0 = \perp, \dots, x_{n+1} = x_n \nabla F(x_n), \dots$$

- ▶ classical definition : for all increasing chains $x_0 \sqsubseteq \dots \sqsubseteq x_n \sqsubseteq \dots$, the chain $y_0 = x_0, \dots, y_{n+1} = y_n \nabla x_n, \dots$ is not strictly increasing
- ▶ constructive definition : $\forall x : t, \text{Acc } \prec_{\nabla} (x, x)$ with

$$(x_1, y_1) \prec_{\nabla} (x_2, y_2) \text{ iff } x_2 \sqsubseteq x_1 \wedge y_1 = y_2 \nabla x_1 \wedge y_1 \neq y_2$$

WANTED

$$L = R \times S \times L \times H$$

with

$$R = \text{MethodName} \rightarrow \text{Context} \rightarrow \text{Val}^\sharp$$

$$S = (\text{MethodName} \times \text{ProgPoint}) \rightarrow \text{Context} \rightarrow \text{List}(\text{Val}^\sharp)$$

$$L = (\text{MethodName} \times \text{ProgPoint}) \rightarrow \text{Context} \rightarrow (\text{Var} \rightarrow \text{Val}^\sharp)$$

$$H = (\text{ClassName} \times \text{FieldName}) \rightarrow \text{Val}^\sharp$$

$$\text{Val}^\sharp = \wp(\text{ClassName}) + \text{Num}^\sharp \quad \text{Context} = (\text{MethodName} \times \text{ProgPoint})^{*\leq k}$$

and $\text{Num}^\sharp$ which

- ▶ either respects `LatticeWf` (constant lattice)
 - ▶ we must prove that L respects `LatticeWf` !
- ▶ or respects `LatticeWiden` (interval lattice)
 - ▶ we must prove that L respects `LatticeWiden` !

Proposed solution : module functors

Module functors : functions that take modules as arguments and produce modules as results

Various functors

- ▶ product, sum, list

Example of property proved:

$$\left(\begin{array}{l} \forall a : A, \text{Acc} \prec_{\nabla_A} (a, a) \\ \forall b : B, \text{Acc} \prec_{\nabla_B} (b, b) \end{array} \right) \implies \forall c : A \times B, \text{Acc} \prec_{\nabla_{A \times B}} (c, c)$$

- ▶ function on finite sets
 - ▶ implemented with functional map or dictionaries on binary keys
 - ▶ finite sets are constructed by functor compositions too

See the poster for more details !

Conclusions

- ▶ For many analyses, with an appropriate combination of the proposed functors, no termination proofs are needed
- ▶ A nice illustration of module functors, more involved (wrt. modularity) than classical examples (set, dictionaries)
- ▶ A nice example mixing mathematical structures and efficient extracted data structures

Interested ?

Let's discuss together !

Offering a postdoc position for October 2005 ?

Let's discuss together !

Author: Julien Narboux

Title: Automatizing geometry using Coq

Abstract:

We present the development of a decision procedure for affine plane geometry in the Coq proof assistant. Among the existing decision methods, we have chosen to implement one based on the area method developed by Chou, Gao and Zhang, which provides short and ``readable" proofs for geometry theorems. The idea of the method is to express the goal to be proved using three geometric quantities and eliminate points in the reverse order of their construction thanks to some elimination lemmas.

The Chou, Gao and Zhang method

- Find a point which is not used to build any other point.
 - The theorem must be stated *constructively*.
- Eliminate every occurrence of this point from the goal.
 - We need some theorem to *eliminate* the point.
- Repeat until the goal contains only *free* points.
- Deal with the *free* points.
- Check if the remaining goal (an equation on a field) is true.

Elimination lemmas

Construction	Description	Elimination formulas	
	(Nondegeneracy condition)	$\mathcal{S}_{ABY} =$	If $AY \parallel CD$ then $\frac{\overline{AY}}{\overline{CD}} =$
	Take Y on line PQ such that $\frac{\overline{PY}}{\overline{PQ}} = \lambda$. ($P \neq Q$)	$\lambda \mathcal{S}_{ABQ} + (1 - \lambda) \mathcal{S}_{ABP}$	$\begin{cases} \frac{\frac{\overline{AP}}{\overline{PQ}} + \lambda}{\frac{\overline{CD}}{\overline{PQ}}} & \text{if } A \in PQ \\ \frac{\mathcal{S}_{APQ}}{\mathcal{S}_{CPDQ}} & \text{otherwise}^!. \end{cases}$
	Take Y at the intersection of PQ and UV . ($PQ \nparallel UV$)	$\frac{\mathcal{S}_{PUV} \mathcal{S}_{ABQ} + \mathcal{S}_{QVU} \mathcal{S}_{ABP}}{\mathcal{S}_{PUQV}}$	$\begin{cases} \frac{\mathcal{S}_{AUUV}}{\mathcal{S}_{CUDV}} & \text{if } A \notin UV \\ \frac{\mathcal{S}_{APQ}}{\mathcal{S}_{CPDQ}} & \text{otherwise.} \end{cases}$
	Take Y on the parallel to PQ passing through R such that $\frac{\overline{RY}}{\overline{PQ}} = \lambda$. ($P \neq Q$)	$\mathcal{S}_{ABR} + \lambda \mathcal{S}_{APBQ}$	$\begin{cases} \frac{\frac{\overline{AR}}{\overline{PQ}} + \lambda}{\frac{\overline{CD}}{\overline{PQ}}} & \text{if } A \in RY \\ \frac{\mathcal{S}_{APRQ}}{\mathcal{S}_{CPDQ}} & \text{otherwise.} \end{cases}$
	Take Y at the intersection of UV and the parallel to PQ passing through R . ($PQ \parallel UV$)	$\frac{\mathcal{S}_{PUQR} \mathcal{S}_{ABV} - \mathcal{S}_{PVQR} \mathcal{S}_{ABU}}{\mathcal{S}_{PUQV}}$	$\begin{cases} \frac{\mathcal{S}_{AUUV}}{\mathcal{S}_{CUDV}} & \text{if } A \notin UV \\ \frac{\mathcal{S}_{APRQ}}{\mathcal{S}_{CPDQ}} & \text{otherwise.} \end{cases}$
	Take Y at the intersection of the parallel to PQ passing through R and the parallel to UV passing through W . ($PQ \parallel UV$)	$\frac{\mathcal{S}_{PWQR}}{\mathcal{S}_{PUQV}} \cdot \mathcal{S}_{AUBV} + \mathcal{S}_{ABW}$	$\begin{cases} \frac{\mathcal{S}_{APRQ}}{\mathcal{S}_{CPDQ}} & \text{if } AY \nparallel PQ \\ \frac{\mathcal{S}_{AUUV}}{\mathcal{S}_{CUDV}} & \text{otherwise.} \end{cases}$

The axiomatic

Points Point : Set

Field F is a field
 $2 \neq 0$

Signed distance $\overline{} : \text{Point} \rightarrow \text{Point} \rightarrow F$
 $\overline{AB} = 0 \iff A = B$

Signed area $\mathcal{S} : \text{Point} \rightarrow \text{Point} \rightarrow \text{Point} \rightarrow F$
 $\mathcal{S}_{ABC} = \mathcal{S}_{CAB}$
 $\mathcal{S}_{ABC} = -\mathcal{S}_{BAC}$

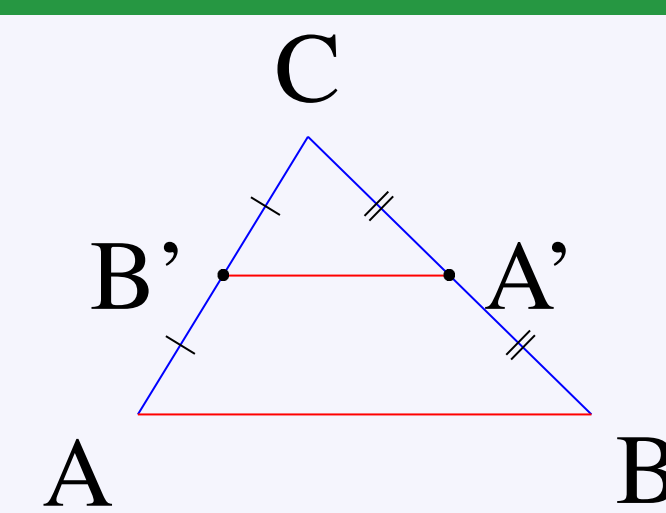
Chasles' axiom $\mathcal{S}_{ABC} = 0 \rightarrow \overline{AB} + \overline{BC} = \overline{AC}$

Dimension $\exists A, B, C : \text{Point}, \mathcal{S}_{ABC} \neq 0$
 $\mathcal{S}_{ABC} = \mathcal{S}_{DBC} + \mathcal{S}_{ADC} + \mathcal{S}_{ABD}$

Construction $\forall r : F \exists P : \text{Point}, \mathcal{S}_{ABP} = 0 \wedge \overline{AP} = r \overline{AB}$
 $A \neq B \wedge \mathcal{S}_{ABP} = 0 \wedge \overline{AP} = r \overline{AB} \rightarrow P = P'$
 $\wedge \mathcal{S}_{ABP'} = 0 \wedge \overline{AP'} = r \overline{AB}$

Proportions $A \neq C \rightarrow \mathcal{S}_{PAC} \neq 0 \rightarrow \mathcal{S}_{ABC} = 0 \rightarrow \frac{\overline{AB}}{\overline{AC}} = \frac{\mathcal{S}_{PAB}}{\mathcal{S}_{PAC}}$

The midpoint theorem using Coq



Let ABC be a triangle, and let A' and B' be the midpoints of BC and AC respectively. Then the line $A'B'$ is parallel to the base AB .

forall A B C A' B' : Point, midpoint A' B C -> midpoint B' A C ->
parallel A' B' A B.

At this step it would be enough to type `autogeom` to solve the goal using our decision procedure, but for this presentation we mimic the behavior of the decision procedure using some sub-tactics. For this presentation the fact that A, B, C, D and E are of type `point` has been removed from the context.

`geoInit.`

```
H : on_line_d A' B C (1 / 2)
H0 : on_line_d B' A C (1 / 2)
=====
S A' A B' + S A' B' B = 0
```

`eliminate B'.`

```
H : on_line_d A' B C (1 / 2)
=====
1 / 2 * S A' A C + (1 - 1 / 2) * S A' A A +
(1 / 2 * S B A' C + (1 - 1 / 2) * S B A' A) = 0
```

`basic_simpl.`

```
H : on_line_d A' B C (1 / 2)
=====
1 / 2 * S A' A C + (1 / 2 * S B A' C + 1 / 2 * S B A' A) = 0
```

`eliminate A'.`

```
=====
1 / 2 * (1 / 2 * S A C C + (1 - 1 / 2) * S A C B) +
(1 / 2 * (1 / 2 * S C B C + (1 - 1 / 2) * S C B B) +
1 / 2 * (1 / 2 * S A B C + (1 - 1 / 2) * S A B B)) = 0
```

`basic_simpl.`

```
=====
1 / 2 * (1 / 2 * S A C B) + 1 / 2 * (1 / 2 * S A B C) = 0
```

`unify_signed_areas.`

```
=====
1 / 2 * (1 / 2 * S A C B) + 1 / 2 * (1 / 2 * - S A C B) = 0
```

`field_and_conclude.`

`Proof completed.`

The formalization of some geometric notions

Geometric notions	Formalization
A, B and C are collinear	$\mathcal{S}_{ABC} = 0$
$AB \parallel CD$	$\mathcal{S}_{ABC} = \mathcal{S}_{ABD}$
I is the midpoint of AB	$\frac{\overline{AB}}{\overline{AI}} = 2 \wedge \mathcal{S}_{ABI} = 0$
$AB \perp BC$	$\mathcal{P}_{ABC} = 0$
$AB \perp CD$	$\mathcal{P}_{ACD} = \mathcal{P}_{BCD}$
$A = B$	$\mathcal{P}_{ABA} = 0$

References

- [1] S.C. Chou, X.S. Gao, and J.Z. Zhang. *Machine Proofs in Geometry*. World Scientific, Singapore, 1994.
- [2] Frédérique Guilhot. Formalisation en coq d'un cours de géométrie pour le lycée. In *Journées Francophones des Langages Applicatifs*, Janvier 2004.
- [3] Laura I. Meikle and Jacques D. Fleuriot. Formalizing Hilbert's Grundlagen in Isabelle/Isar. In *Theorem Proving in Higher Order Logics*, pages 319–334, 2003.
- [4] Julien Narboux. A Decision Procedure for Geometry in Coq. In *Theorem Proving in Higher Order Logics*, pages 225–240, 2004.

Author: Zhong Shao

Title: The Essence of Proof-Carrying Code

Abstract:

Proof-Carrying Code (PCC) is a very general framework that can, in principle, be used to verify arbitrary properties of machine language programs. Existing PCC systems, including Foundational Proof-Carrying Code (FPCC), however, have not fulfilled such promises. In this talk, we contemplate the important principles that embody the essence of PCC and FPCC, describe a framework that satisfies these principles, and show that the new framework is simpler, more general and flexible, and more likely to help us realize the full potential of PCC.

Authors: Ulrich Schoepp, Ian Stark

Title: A Dependent Type Theory with Names and Binding

Abstract:

This talk is about how names and binding can be integrated in dependent type theory. Names and name-binding are standard tools for working informally with abstract syntax and variables binding. Gabbay and Pitts have shown that Fraenkel Mostowski (FM) set theory supports a formal notion of names using which informal work with names and binding can be made precise. Based on the notion of name-swapping, Gabbay and Pitts give a number of useful constructions for working with names, such as a syntax-independent notion of freshness that formalises when two objects do not share names and a freshness quantifier that simplifies work with names.

To make FM-style name constructs suitable for integration in dependent type theory, we give an alternative presentation of them based on freshness rather than swapping. This presentation has as its central primitive a monoidal structure capturing freshness. In analogy to the definition of simple dependent products (Π) and sums (Σ) for a cartesian structure, we define monoidal dependent products (Π^*) and sums (Σ^*) for the monoidal structure. We express FM constructs for names and binding, such as alpha-equivalence classes, unique choice of fresh names and the freshness quantifier in terms of these dependent monoidal products and sums.

We use bunches to combine dependent types with the monoidal structure, Π^* and Σ^* . The resulting bunched type theory is related to the simply typed λ -calculus of O'Hearn and Pym. In addition to Π^* and Σ^* , the type theory also features hidden-name types which are in propositions-as-types correspondence to the freshness quantifier.

A Dependent Type Theory with Names and Binding

Ulrich Schöpp Ian Stark

Laboratory for Foundations of Computer Science
University of Edinburgh

December 16, 2004

Names and Binding for Abstract Syntax

Example: Reasoning about the λ -calculus.

Barendregt Variable Convention [Barendregt, 1984]:

2.1.12. CONVENTION *Terms that are α -congruent are identified. [...]*

2.1.13. VARIABLE CONVENTION *If $M_1, \dots, M_n$ occur in a certain mathematical context (e.g. definition, proof) then in these terms all bound variables are chosen to be different from the free variables. [...]*

2.1.14. MORAL *Using conventions 2.1.12 and 2.1.13 one can work with λ -terms in the naive way.*

Aim: Support this convention in formal reasoning, in particular in dependent type theory.

We build on *lots* of previous work, e.g. [McKinna, Pollack], [Honsell, Miculan, Scagnetto], [Gabbay, Pitts], ...

A Dependent Type Theory with Names and Binding

We give a dependent type theory with

- Type of names $\mathbf{N}$ with decidable equality,
- Additive Types $(\Sigma, \Pi, \dots)$,
- Multiplicative Types $(\Sigma^*, \Pi^*, \dots)$.

Main use of multiplicatives is to represent α -equivalence classes.

(Π^*) Weak Higher Order Abstract Syntax style

[Despeyroux, Felty, Hirschowitz], [Honsell, Miculan, Scagnetto], ...

(Σ^*) FM abstraction set style

[Gabbay, Pitts]

A Bunched Dependent Type Theory

We use *bunches* [O'Hearn, Pym] to integrate the additive structure with the multiplicative structure.

$$\frac{}{\diamond \text{ Bunch}} \qquad \frac{\Gamma \text{ Bunch} \quad \Gamma \vdash A \text{ Type}}{\Gamma, x : A \text{ Bunch}} \\[1em] \frac{\Gamma \text{ Bunch} \quad \Delta \text{ Bunch}}{\Gamma * \Delta \text{ Bunch}}$$

Examples:

$$(x : \text{Lam}, y : \text{Lam}) * z : \text{Lam}$$

$$((x : A * y : B), z : C) * u : D$$

Semantic intuition:

$$A * B = \{ \langle x, y \rangle : A \times B \mid x \text{ and } y \text{ can be built using disjoint name sets} \}$$

Monoidal Products (Π^*)

The type $\Pi^*x:A. B$ consists of partial functions that are defined on arguments containing just fresh names.

$$\frac{\Gamma * x:A \vdash M : B}{\Gamma \vdash \lambda^*x:A. M : \Pi^*x:A. B} \qquad \frac{\Gamma \vdash M : \Pi^*x:A. B \quad \Delta \vdash N : A}{\Gamma * \Delta \vdash M@N : B [N/x]}$$

Example (Syntax in WHOAS-style)

$\text{Lam} ::= \text{var of } \mathbf{N} \mid \text{app of } \text{Lam} \times \text{Lam} \mid \text{lam of } \Pi^*n:\mathbf{N}. \text{Lam}$

Substitution of $R:\text{Lam}$ for $m:\mathbf{N}$.

$$\begin{aligned} \text{subst} &: \text{Lam} \rightarrow \text{Lam} \\ \text{subst}(\text{var}(n)) &= \text{if } (n = m) \text{ then } R \text{ else } \text{var}(n) \\ \text{subst}(\text{app}(M, N)) &= \text{app}(\text{subst}(M), \text{subst}(N)) \\ \text{subst}(\text{lam}(M)) &= \text{lam}(\lambda^*n. \text{subst}(M@n)) \end{aligned}$$

Freefrom Types

The type $B^{*M:A}$ consists of all the elements of B that are *free from* the names in the term $M : A$.

- Example:

$$\text{remove} : \prod n : \mathbf{N}. (\mathbf{L} \rightarrow \mathbf{L}^{*n:\mathbf{N}}).$$

- Current syntax is incomplete (for a categorical semantics):
In $B^{*M:A}$ both A and B have to be closed.

Monoidal Sums (Σ^*)

The type $\Sigma^*x:A. B$ consists of pairs $M.N$ in which the names of M are hidden.

$$\frac{x:A \vdash B \text{ Type} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : B [M/x]}{\Gamma \vdash \text{bind}(M, N) : (\Sigma^*x:A. B)^{*M:A}}$$

$$M.N \stackrel{\text{def}}{=} (\text{let bind}(M, N) \text{ be } y^{*x} \text{ in } y) : \Sigma^*x:A. B$$

$$\frac{\Gamma \vdash M : \Sigma^*x:A. B \quad (\Gamma * x:A), y:B \vdash N : C^{*x:A}}{\Gamma \vdash \text{let } M \text{ be } x.y \text{ in } N : C}$$

Example (Syntax in FM-style)

$\text{Lam} ::= \text{var of } \mathbf{N} \mid \text{app of } \text{Lam} \times \text{Lam} \mid \text{lam of } \Sigma^*n:\mathbf{N}. \text{Lam}$

Substitution.

$$\begin{aligned} \text{subst}(\text{lam}(M)) = \\ \text{let } M \text{ be } n.x \text{ in } (\text{let bind}(n, \text{subst}(x)) \text{ be } y^{*n} \text{ in } \text{lam}(y)^{*n}) \end{aligned}$$

Hidden-Name Types

Both $(\Pi^*n:\mathbf{N}.\text{Lam})$ and $(\Sigma^*n:\mathbf{N}.\text{Lam})$ represent α -equivalence classes. $\Rightarrow$ They are isomorphic.

Introduce *hidden-name types* as a syntax for both types.

$$\Sigma^*n:\mathbf{N}.B \quad \cong \quad \mathbf{H}n.B \quad \cong \quad \Pi^*n:\mathbf{N}.B$$

Rules and equations for H are those from both Σ^* and Π^* plus interaction equations, such as

$$(n.M)@n = M$$

$$n.(N@n) = N$$

...

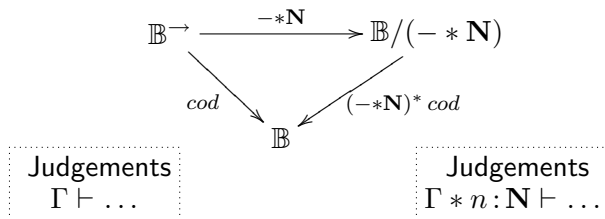
Mixing Σ^* and Π^* is useful:

$$\text{new } n. M \stackrel{\text{def}}{=} \text{let } (\lambda^*n. M) \text{ be } n.x \text{ in } x$$

Formalising the Barendregt Variable Convention

- Category $\mathbb{B}$ with pullbacks.
- **Names:** Object $\mathbf{N}$.
- **Freshness:** Monoidal structure $*$.
- **Working with α -equivalence classes of terms is the same as working with freshly named instances:**

Equivalence of fibrations



Models of this structure: FM-Sets, Realisability version of FM-Sets, Species of Structures.

The equivalence of fibrations

If $- * \mathbf{N}: \mathbb{B}^{\rightarrow} \rightarrow \mathbb{B}/(- * \mathbf{N})$ is an equivalence, then we get two adjunctions

$$\mathbf{H} \dashv (- * \mathbf{N}) \dashv \mathbf{H}$$

related by $\varepsilon' = \eta^{-1}$ and $\eta' = \varepsilon^{-1}$.

$\mathbf{H}$ generalises both the abstraction set $[\mathbf{N}]B$ and the freshness quantifier $\mathbb{H}$ of FM set theory. Good properties of $\mathbf{H}$, such as

$$\begin{aligned}\mathbf{H}(A + B) &\cong \mathbf{H}A + \mathbf{H}B \\ \mathbf{H}(A \times B) &\cong \mathbf{H}A \times \mathbf{H}B \\ \mathbf{H}(A \rightarrow B) &\cong \mathbf{H}A \rightarrow \mathbf{H}B,\end{aligned}$$

are immediate, since $\mathbf{H}$ is an equivalence.

The equivalence of fibrations

The adjunctions

$$H \dashv (- * \mathbf{N}) \dashv H$$

are a special case of

$$\Sigma_A^* \dashv (- * A) \dashv \Pi_A^*,$$

on which the rules for Σ^* and Π^* are modelled.

Freeform types are the syntax for $- * A$.

We get

$$\Sigma_{\mathbf{N}}^* \cong H \cong \Pi_{\mathbf{N}}^*.$$

The equations $\varepsilon' = \eta^{-1}$ and $\eta' = \varepsilon^{-1}$ are the interaction equations for hidden-name types: $(n.M)@n = M, \dots$

Conclusion and Further Work

Conclusion

Σ^* , H and Π^* are good for names!

Further Work

- Better free from types.
- Decidability questions.
- Practicable?
- Semantics
 - Other models?
 - Free construction of H ?
 - Model ∇ ?
- ...

Authors: Marino Miculan, Ivan Scagnetto

Title: FOL-nabla in Higher Order Abstract Syntax

Abstract:

Miller-Tiu's FOL-nabla is a logic recently proposed for reasoning about generic judgments. In this work, we present a formalization of this logic in Coq, using Higher-Order abstract syntax and the Theory of Contexts.

$FO\lambda^\nabla$ in Higher Order Abstract Syntax

Marino Miculan Ivan Scagnetto

Department of Mathematics and Computing Science
University of Udine

TYPES 2004, Jouy-en-Josas, December 16, 2004

1 / 16

Reasoning with abstractions

In recent years, much work on systems for reasoning about object systems featuring *abstractions*.

Two approaches (among several others):

- HOAS(+ToC): weak higher-order abstract syntax in usual TT (e.g., CIC) possibly “patched” with specific properties for reasoning about second-order terms (the Theory of Contexts)
- $FO\lambda^\nabla$ [Miller&Tiu]: FO logic specifically designed for dealing with “schematic” properties

A given object system with abstractions (e.g., λ -calculus, π -calculus) can be easily encoded in any of them.

Question

Which is the relation between $FO\lambda^\nabla$ and HOAS-based encodings?

2 / 16

Weak HOAS encodings: the untyped λ -calculus

Canonical example: take $M, N ::= x \mid MN \mid \lambda x.M$.

Its “weak” HOAS encoding in Coq is

```
Parameter Var : Set.
Inductive Tm : Set :=
  var : Var -> Tm
| app : Tm -> Tm -> Tm
| lam : (Var -> Tm) -> Tm.
```

Example:

$\lambda x.xy \mapsto (\text{lam } (\text{fun } x:\text{Var} \Rightarrow (\text{app } (\text{var } x) (\text{var } y))))$

Advantage

α -conversion *and* induction/recursion over terms are automatically provided by the metalanguage.

3 / 16

Weak HOAS encodings: capture-avoiding substitution

- Capture avoiding substitution $M[N/x]$ is not given for free
- It can be represented by means of a ternary predicate subst:


```
Inductive subst (M:Tm) : (Var->Tm) -> Tm -> Prop :=...
```

 Intended meaning: (subst M N O) holds iff O is the result of “filling” the hole of N with M.
- Some crucial properties of substitution are needed, such as:


```
forall x:Var, forall M:Var->Tm, forall N:Tm,
  (subst (var x) M N) -> N=(M x).
```

```
forall M:Tm,forall N:Var->Tm, exists O:Tm,(subst M N O)
```

- These results cannot be proved in pure CIC, but they can be proved assuming the Theory of Contexts

4 / 16

The Theory of Contexts (ToC)

The Theory of Contexts is a set of axioms formalizing some simple properties about variables and contexts (i.e., terms with *holes*):

Decidability of equality over variables

$$\forall x, y. x = y \vee x \neq y$$

Existence of fresh variables

$$\forall M. \exists x. x \notin FV(M)$$

 β -expansion

$$\forall M. \forall x. \exists M'[\cdot]. x \notin FV(M'[\cdot]) \wedge M = M'[x]$$

Restricted extensionality

$$\forall M[\cdot], N[\cdot]. \forall x. x \notin FV(M[\cdot], N[\cdot]) \Rightarrow M[x] = N[x] \Rightarrow M[\cdot] = N[\cdot]$$

5 / 16

Syntax of FO λ^{∇} [Miller&Tiu, LICS 2003]

- Types: usual simple types: $\tau ::= o \mid \gamma \mid \tau_1 \rightarrow \tau_2$
- Terms: usual simply typed λ -calculus: $\Sigma \vdash t : \tau$
- Object-level datatypes can be represented by adding types and constructors (even higher-order)
- (Basic) Formulas: terms of type o :

$$A, B ::= \top \mid \perp \mid A \supset B \mid \dots \mid \forall_{\gamma} x. A \mid \exists_{\gamma} x. A \mid \nabla_{\gamma} x. A$$

- Generic Judgments:

$$A, B ::= \overbrace{(x_1 : \tau_1, \dots, x_n : \tau_n)}^{\sigma} \triangleright B$$

Think of $x_1, \dots, x_n$ as *locally scoped constants*, i.e., *eigenvariables*.

6 / 16

Proof system

Original proof system is in Gentzen-style, for deriving sequents

$$\Sigma : B_1, \dots, B_n \vdash B$$

Here we consider an equivalent variant:

Proof system in Natural Deduction—some simple rules

$$\begin{array}{c} \frac{\sigma \triangleright A \quad \sigma \triangleright B}{\sigma \triangleright A \wedge B} \wedge I \quad \frac{\sigma \triangleright A \wedge B}{\sigma \triangleright A} \wedge E I \quad \frac{\sigma \triangleright A \wedge B}{\sigma \triangleright B} \wedge E r \\ \frac{\sigma \triangleright A}{(\sigma \triangleright A)} \\ \vdots \\ \frac{\sigma \triangleright B}{\sigma \triangleright A \supset B} \supset I \quad \frac{\sigma \triangleright A \supset B \quad \sigma \triangleright A}{\sigma \triangleright B} \supset E \\ \frac{\sigma, x : \gamma \triangleright B}{\sigma \triangleright \nabla_{\gamma} x. B} \nabla I \quad \frac{\sigma \triangleright \nabla_{\gamma} x. B}{\sigma, x : \gamma \triangleright B} \nabla E \quad \frac{}{\sigma \triangleright \top} \top I \quad \frac{\sigma \triangleright \perp}{B} \perp E \end{array}$$

7 / 16

Proof system

Rules using *raising*

$$\frac{(h : |\sigma| \rightarrow \gamma) \quad \frac{\sigma \triangleright B[(h \sigma)/x]}{\sigma \triangleright \forall_{\gamma} x. B} \forall I}{\sigma \triangleright \forall_{\gamma} x. B} \forall I \quad \frac{(\sigma) \quad \frac{t : \gamma \quad \sigma \triangleright \forall_{\gamma} x. B}{\sigma \triangleright B[t/x]} \forall E}{\sigma \triangleright B[t/x]} \forall E$$

Notice: when $\sigma = ()$, we get back the usual intuitionistic FOL.

8 / 16

$FO\lambda^\nabla$ vis-a-vis weak HOAS

	$FO\lambda^\nabla$	weak HOAS
schematic terms	$\lambda x : \tau. t$	$\lambda x : Var_\tau. t$
bound variables in terms	metavariables of type τ	metavariables of type Var_τ
generic quantifier	$\nabla x. B$	—
generic judgments	$(x_1, \dots, x_n) \triangleright B$	$\lambda x_1 : Var \dots \lambda x_n : Var. B$
eigenvariables in judgments	locally scoped constants	locally scoped <i>distinct metavariables</i>

Translation idea

We can map $FO\lambda^\nabla$'s generic judgments into CIC propositions, by formalizing this correspondence, and by internalizing the notion of “locally scoped distinct metavariable”.

9 / 16

Encoding $FO\lambda^\nabla$ syntax in weak HOAS

(Harmless simplification: just 1 type of terms)

Terms and Formulas

Parameter Var:Set.

Inductive Term:Set :=...

Inductive Form : Set :=

tt : Form

| And : Form -> Form -> Form

| Imp : Form -> Form -> Form

| Forall : (Var->Form) -> Form

| Nabla : (Var->Form) -> Form

| ...

10 / 16

Encoding $FO\lambda^\nabla$ syntax in weak HOAS

Generic Judgments

```
Fixpoint GenJ (n:nat) {struct n} : Set :=
  match n with
  | 0 => Form
  | (S m) => Var -> (GenJ m)
  end.
```

Thus, a generic judgment is a “formula with holes”:

$$(x_1, \dots, x_n) \triangleright B \mapsto \text{fun } x_1 \dots x_n : \text{Var} \Rightarrow B$$

$$: (\text{GenJ } n)$$

$$= \text{Var} \rightarrow \dots \rightarrow \text{Var} \rightarrow \text{Form}$$

11 / 16

Translating $FO\lambda^\nabla$ formulae to CIC using HOAS

By means of an inductive predicate, capturing the “intuitive” meaning of formulas:

Inductive T_F : Form -> Prop := ...

| T_F_And: forall A B:Form, (T_F A) -> (T_F B) -> (T_F (And A B))

| T_F_Forall : forall A:Var->Form,
 (forall t:Term, forall B:Form, (substf t A B) -> (T_F B))
 -> (T_F (Forall A))

| T_F_Nabla : forall A:Var->Form,
 (forall x:Var, (notin_Form x (Nabla A)) -> (T_F (A x)))
 -> (T_F (Nabla A)).

where notin_Form is as usual in weak HOAS encodings:

Inductive notin_Form (x:Var): Form -> Prop := ...

Formal meaning: (notin_Form x A) holds iff $x \notin FV(A)$.

12 / 16

Translating $FO\lambda^\nabla$ judgments to CIC using HOAS

Generic judgments are translated by recursion on the length of the local signature, similarly to ∇ :

```
Fixpoint T_GJ (n:nat) {struct n} : (GenJ n) -> Prop :=
  match n return (GenJ n) -> Prop with
  | 0 => T_Form
  | (S m) => (fun t:Var->(GenJ m) =>
    forall x:Var, (notin_GenJ (S m) x t)
      -> (T_GJ m (t x))
    )
  end.
```

Notice

The proof system of $FO\lambda^\nabla$ is *not* encoded: only the “intended” meaning of formulas and judgment is captured.

13 / 16

The translation is complete...

Theorem

If $\Sigma : \sigma_1 \triangleright B_1, \dots, \sigma_n \triangleright B_n \vdash \sigma \triangleright B$ in $FO\lambda^\nabla$, then

Goal (T_GJ s1 B1) -> ... (T_GJ sn Bn) -> (T_GJ s B)

is provable in CIC+ToC.

Proof.

Every rule of $FO\lambda^\nabla$ translates into a Theorem in CIC, e.g.:

Lemma AND_I: forall n:nat, forall A B:GJ n,
(T_GJ n A) -> (T_GJ n B) -> (T_GJ n (AND n A B)).

which can be proved using standard techniques and possibly the axioms from the Theory of Contexts. $\square$

14 / 16

...but CIC+ToC is strictly stronger

Some non-theorems of $FO\lambda^\nabla$ are provable in CiC+ToC:

- $\forall x.A \supset \nabla x.A$ translates to
(forall t:Term,forall B:Form,(substf t A B)->(T_F B))
->
(forall x:Var,(notin_Form x (Nabla A)) -> (T_F (A x)))
directly provable *without* using ToC axioms (apart in properties of subst)
- $\nabla x.A \supset \exists x.A$ translates to
(forall x:Var,(notin_Forall x (Nabla A)) -> (T_F (A x))
->
(exists t:Term,forall B:Form,(substf t A B)->(T_F B))
provable using “freshness” axiom (and properties subst)

15 / 16

Conclusions

 $FO\lambda^\nabla$ is easily translated into CIC with HOAS

The translation is complete, but CIC+ToC is strictly stronger (which is not necessarily good).

(Some) Future work

- How to weaken ToC in order to match $FO\lambda^\nabla$'s strength?
- Or shall $FO\lambda^\nabla$ be strengthened to match CIC+ToC (and FM-logics), still keeping the good proof theoretical properties?
- Generalize to multi-sorted variable sets
- Move to $FO\lambda^{\Delta\nabla}$, by accommodating also *definitions*.

16 / 16

Authors: Neil Ghani, Thorsten Altenkirch, Tarmo Uustalu, Michael Abbott

Title: Higher Order Datatypes Via Containers

Abstract:

Type systems of functional languages such as Haskell permit the definition of sophisticated datatypes incorporating advanced features such as variable binding and structural constraints. Classic examples of these phenomena are lambda calculus syntax, explicit substitutions and balanced trees. However, the traditional presentation of such datatypes hides the fact that they are concrete data structures in that they store data in certain locations. This is unfortunate as a number of algorithms specifically exploit this structure.

We rectify the situation by showing that a wide class of nested datatypes (inductive constructors of rank 2) are examples of our recently introduced idea of containers. The direct pay off for this is that we can then show how algorithms, eg Huet's zipper, which rely on the storage of data at locations can be generalised to such datatypes.

Abstract Syntax Via Containers

Neil Ghani

Dept of Maths and Computer Science
University of Leicester

(Also Michael Abbott, Thorsten Altenkirch, Tarmo Uustalu)

An Overview of the Talk

- **Abstract Syntax:** Variable binding via initial algebra
 - A semantics in functor categories
 - Examples are nests, lambda terms and explicit substitutions
- **Containers:** A semantic approach to concrete data structures
 - Fundamental idea of data stored at locations in memory
 - Strictly positive types (lists, trees etc) are containers
- **Ambition:** Do algorithms for lists, trees etc generalise smoothly to nests, lambda terms and explicit substitutions etc?
 - Yes if these examples are containers

Abstract Syntax

What is Abstract Syntax

- **Examples:** Nests, Lambda terms and Explicit Substitutions are canonical examples of constrained datatypes, everything, and abstract machines

$$N(X) = X + N(X \times X)$$

$$L(X) = X + L(X) \times L(X) + L(X + 1)$$

$$\begin{aligned} E(X) &= X + E(EX) \\ &= X + \int \mathcal{C}(n, EX) \times E(n) \end{aligned}$$

- **Solutions:** Rewrite and solve in functor categories

$$N = \text{Id} + N \circ \Delta$$

$$L = \text{Id} + \Delta \circ L + L \circ \delta$$

$$E = \text{Id} + E \circ E$$

where $\Delta(X) = X \times X$ and $\delta(X) = X + 1$

A Few Pesky Details

- **Key Idea:** Solve $F = \Phi(F)$ where $\Phi : [\mathcal{C}, \mathcal{C}] \rightarrow [\mathcal{C}, \mathcal{C}]$
 - If $\mathcal{C}$ has ω -colimits, so has $[\mathcal{C}, \mathcal{C}]$
 - If Φ is built from $K_F, +, \times, \circ$, then Φ preserves ω -colimits
 - Size matters - use lfp categories. Others use variants of Sets
- **Theorem:** Such Φ have an initial algebra. Tarmo showed that such structures are often monadic.
- **Reflection:** This is great because we love initial algebra semantics. But
 - Are these examples concrete data structures
 - Is lfp-ness the right approach

Containers

What are Containers

- **Lists:** Given by a length and a mapping of cells to data

$$\text{List}(X) = \Sigma n:\mathbb{N}. \{0, \dots, n-1\} \longrightarrow X$$

- **Pairs:** One shape containing two positions for data

$$X \times X = \Sigma s:1. 2 \longrightarrow X$$

- **Definition:** A container is a dependent type $s : S \vdash P_s$ type

- A presentation of $T_{S,P}(X) = \Sigma s:S. P_s \longrightarrow X$
- Formalises the idea of data stored at memory locations
- Containers are more general than polynomial functors.

Some Results about Containers

- **Lemma 1:** Strictly positive types are containers.

$$\tau := X \mid \tau + \tau \mid \tau \times \tau \mid K_A \mid \mu X. \tau \mid \nu X. \tau \mid \tau \circ \tau$$

- **Lemma 2:** Semantics require LCCC + W-types. That is, W-types calculates shapes and positions.
- **Lemma 3:** W-types = least fixed points of containers
 - A semantic definition of W-types
- **Lemma 4:** Containers can be differentiated etc (McBride, Hancock)

Abstract Syntax Via Containers

From the work on SP types

- If $NX = X + N(X \times X)$ satisfies $N = T_{S,P}$ then

$$\begin{aligned} S &= 1 + S \\ P(0) &= 1 \\ P(n+1) &= P(n) + P(n) \end{aligned}$$

- If $LX = X + LX \times LX + L(X + 1)$ satisfies $L = T_{S,P}$ then

$$\begin{aligned} S &= 1 + S \times S + \Sigma s : S.P(s) \rightarrow \text{Bool} \\ P(*) &= 1 \\ P(s_1, s_2) &= P(s_1) + P(s_2) \\ P(s, f) &= \Sigma t : P(s).\text{if } f(t) \text{ then } 1 \text{ else } 0 \end{aligned}$$

- If $EX = X + E(EX)$ satisfies $E = T_{S,P}$ then

$$\begin{aligned} S &= 1 + \Sigma s : S.P(s) \rightarrow S \\ P(*) &= 1 \\ P(s, f) &= \Sigma p : P(s).P(fp) \end{aligned}$$

A grammar of higher order datatypes

- **Motivation:** If so algorithms such as zippers, differentiation, resource analysis may generalise
- **Question:** A grammar of higher order datatypes?
 - Let n denote $[\mathcal{C}^n, \mathcal{C}]$ and $\bar{m} \rightarrow n = m_1 \times \cdots m_k \rightarrow n$

$$\frac{}{\overline{X_i : \bar{m} \rightarrow m_i}} \qquad \frac{F : n}{K_F : \bar{m} \rightarrow n}$$
$$\frac{\phi : \bar{m}, n \rightarrow n}{\mu\phi, \nu\phi : \bar{m} \rightarrow n} \qquad \frac{\phi : \bar{m} \rightarrow n \quad \psi_i : \bar{k} \rightarrow m_i}{\phi(\psi_1, \dots, \psi_r) : \bar{k} \rightarrow n}$$

Can represent E, L, N and define $\llbracket \phi \rrbracket$ as a functor.

- **Problem:** But we wanted containers, not functors!

Container Transformers

- Let G_n be the category of n -ary containers
 - Any $\phi : n \rightarrow n$ is a container transformer $[[\phi]] : G_n \rightarrow G_n$
 - ϕ -preserves filtered colimits (of a certain size)
- **But:** G_n doesn't have ω -colimits
 - But G_n has enough ω -colimits - cartesian chains!
 - So $\mu\phi : n$ is a container as required
- **Future Work:** Replace size conditions, lfp-conditions with W -types. EPSRC grant for RA and PhD student to
 - Dependent containers, resource containers, generic navigation algorithms on containers ...

Authors: Alwen Tiu, Dale Miller

Title: A proof search specification of the pi calculus

Abstract:

We present a meta-logic that contains a new quantifier nabla (for encoding ``generic judgments") and inference rules for reasoning within fixed points of a given specification. We then specify the operational semantics and bisimulation relations for the finite pi-calculus within this meta-logic. Since we restrict to the finite case, the ability of the meta-logic to reason within fixed points becomes a powerful and complete tool since simple proof search can compute this one fixed point. The nabla quantifier helps with the delicate issues surrounding the scope of variables within pi-calculus expressions and their executions (proofs). We shall illustrate several merits of the logical specifications we write: they are natural and declarative; they contain no side conditions concerning names of variables while maintaining a completely formal treatment of such variables; differences between late and open bisimulation relations are easy to see declaratively; and proof search involving the application of inference rules, unification, and backtracking can provide complete proof systems for both one-step transitions and for bisimulation. We shall also discuss our recent work on specification of modal logics for pi-calculus and present an "open" modal logic which characterizes open bisimulation for pi-calculus.

A Proof Search Specification of the π -calculus

Alwen Tiu
INRIA Lorraine

Joint work with Dale Miller (*INRIA Futurs/École polytechnique*)

TYPES 2004, 16 December 2004

Motivations

- To use logic, more precisely, *proof theory*, as a *framework* for specifying and reasoning about operational semantics.
- We focus on the presentation of operational semantics as *deductive systems*, e.g., structural operational semantics, and on computation as proof search.
- Reasoning about *mobility* of *names*. The kind of mobility we are considering: link mobility (as in π -calculus), which are basically the changing in scoping of names. Its operational aspects can (hopefully) be understood in the quantification theory of logic.

The λ -tree syntax encoding of operational semantics

- The usual presentations of SOS for name-passing calculi are not “syntax-directed” enough. Various side conditions on names and scoping of names need to be carefully stated along with the inference rules.
- We adopt an abstract syntax, called λ -tree syntax. λ -tree syntax is a logic programming approach to *higher-order abstract syntax*.
- It is based on simply typed λ -calculus: abstractions in process calculi are represented using λ -abstraction, and process constructors are represented as constants with appropriate types. Capture-avoiding substitution is modelled by β -reduction.

An example of λ -tree syntax encoding

An inference rule for (late) transition system of π -calculus:

$$\frac{P \xrightarrow{\alpha} P'}{(x)P \xrightarrow{\alpha} (x)P'} \quad x \notin n(\alpha) \qquad \frac{\forall x (Px \xrightarrow{A} P'x)}{\nu(\lambda x.Px) \xrightarrow{A} \nu(\lambda x.P'x)}$$

P, P' and A are *scheme variables*, subject to *capture avoiding* substitutions. The constants ν is of type $(n \rightarrow p) \rightarrow p$ where the types n and p denote names and processes.

Since substitution is capture-avoiding, the scheme variable A cannot be instantiated with term containing free occurrences of x .

Sequents and generic judgments

We use Gentzen's *sequent calculus* for presenting our logic. A sequent is an expression of the form

$$\Sigma; B_1, \dots, B_n \vdash B_0$$

where $B_0, \dots, B_n$ are formulas and Σ is the *signature* of the sequent, containing declarations of the free variables in each B_i (also called *eigenvariables*).

To capture properly the notion of name restriction in process calculi, we enrich sequents with locally bound variables within the formulas:

$$\Sigma; \sigma_1 \triangleright B_1, \dots, \sigma_n \triangleright B_n \vdash \sigma_0 \triangleright B_0$$

where σ_i is a list of variables locally scoped over B_i .

Variables in σ_i are binding occurrences, so the usual α -equivalence applies.

Overview of $FO\lambda^{\Delta\nabla}$: propositional fragment

$$\frac{}{\Sigma; \sigma \triangleright A, \Gamma \vdash \sigma \triangleright A} \text{init}$$

$$\frac{\Sigma; \Delta \vdash \mathcal{B} \quad \Sigma; \mathcal{B}, \Gamma \vdash \mathcal{C}}{\Sigma; \Delta, \Gamma \vdash \mathcal{C}} \text{cut}$$

$$\frac{}{\Sigma; \sigma \triangleright \perp, \Gamma \vdash \mathcal{B}} \perp \mathcal{L}$$

$$\frac{}{\Sigma; \Gamma \vdash \sigma \triangleright \top} \top \mathcal{R}$$

$$\frac{\Sigma; \mathcal{B}, \mathcal{B}, \Gamma \vdash \mathcal{C}}{\Sigma; \mathcal{B}, \Gamma \vdash \mathcal{C}} c\mathcal{L}$$

$$\frac{\Sigma; \Gamma \vdash \mathcal{C}}{\Sigma; \mathcal{B}, \Gamma \vdash \mathcal{C}} w\mathcal{L}$$

$$\frac{\Sigma; \sigma \triangleright B_i, \Gamma \vdash \mathcal{D}}{\Sigma; \sigma \triangleright B_1 \wedge B_2, \Gamma \vdash \mathcal{D}} \wedge \mathcal{L}$$

$$\frac{\Sigma; \Gamma \vdash \sigma \triangleright B_1 \quad \Sigma; \Gamma \vdash \sigma \triangleright B_2}{\Sigma; \Gamma \vdash \sigma \triangleright B_1 \wedge B_2} \wedge \mathcal{R}$$

$$\frac{\Sigma; \sigma \triangleright B_1, \Gamma \vdash \mathcal{D} \quad \Sigma; \sigma \triangleright B_2, \Gamma \vdash \mathcal{D}}{\Sigma; \sigma \triangleright B_1 \vee B_2, \Gamma \vdash \mathcal{D}} \vee \mathcal{L}$$

$$\frac{\Sigma; \Gamma \vdash \sigma \triangleright B_i}{\Sigma; \Gamma \vdash \sigma \triangleright B_1 \vee B_2} \vee \mathcal{R}$$

$$\frac{\Sigma; \Gamma \vdash \sigma \triangleright B \quad \Sigma; \sigma \triangleright C, \Gamma \vdash \mathcal{D}}{\Sigma; \sigma \triangleright B \supset C, \Gamma \vdash \mathcal{D}} \supset \mathcal{L}$$

$$\frac{\Sigma; \sigma \triangleright B, \Gamma \vdash \sigma \triangleright C}{\Sigma; \Gamma \vdash \sigma \triangleright B \supset C} \supset \mathcal{R}$$

Overview of $FO\lambda^{\Delta\nabla}$: quantifiers

$$\frac{\Sigma; (\sigma, y : \tau) \triangleright B[y/x], \Gamma \vdash \mathcal{C}}{\Sigma; \sigma \triangleright \nabla_{\tau} x. B, \Gamma \vdash \mathcal{C}} \nabla \mathcal{L}$$

$$\frac{\Sigma; \Gamma \vdash (\sigma, y : \tau) \triangleright C[y/x]}{\Sigma; \Gamma \vdash \sigma \triangleright \nabla_{\tau} x. C} \nabla \mathcal{R}$$

$$\frac{\Sigma, \sigma \vdash t : \gamma \quad \Sigma; \sigma \triangleright B[t/x], \Gamma \vdash \mathcal{C}}{\Sigma; \sigma \triangleright \forall_{\gamma} x. B, \Gamma \vdash \mathcal{C}} \forall \mathcal{L}$$

$$\frac{\Sigma, h; \Gamma \vdash \sigma \triangleright B[(h \sigma)/x]}{\Sigma; \Gamma \vdash \sigma \triangleright \forall x. B} \forall \mathcal{R}$$

$$\frac{\Sigma, h; \sigma \triangleright B[(h \sigma)/x], \Gamma \vdash \mathcal{C}}{\Sigma; \sigma \triangleright \exists x. B, \Gamma \vdash \mathcal{C}} \exists \mathcal{L}$$

$$\frac{\Sigma, \sigma \vdash t : \gamma \quad \Sigma; \Gamma \vdash \sigma \triangleright B[t/x]}{\Sigma; \Gamma \vdash \sigma \triangleright \exists_{\gamma} x. B} \exists \mathcal{R}$$

Overview of $FO\lambda^{\Delta\nabla}$: raising of types

Dependency between eigenvariables and local variables is encoded using the technique of $\forall$ -lifting [Paulson] or *raising* [Miller92] of the types of the eigenvariables. Example:

$$\frac{\frac{\{x_\alpha, h_{\tau \rightarrow \gamma \rightarrow \beta}\}; \Gamma \vdash (a_\tau, b_\gamma) \triangleright B \ (h \ a \ b) \ b}{\{x_\alpha\}; \Gamma \vdash (a_\tau, b_\gamma) \triangleright \forall_\beta y. B \ y \ b} \forall \mathcal{L}}{\{x_\alpha\}; \Gamma \vdash (a_\tau) \triangleright \nabla_\gamma z. \forall_\beta y. B \ y \ z} \nabla \mathcal{R}$$

Overview of $FO\lambda^{\Delta\nabla}$: a notion of definitions

We extend the logic further by allowing a non-logical constants (predicate) to be introduced. To each predicate, we associate some *definition clauses*. We write

$$\forall \bar{x}. p \bar{x} \triangleq B \bar{x}$$

to denote a definition clause for predicate p . This notion of definition has been previously studied by Schroeder-Heister, Girard, Miller and McDowell.

Definitions and Equality

The usual form of definitions allows matching on the head of definition, e.g.,

$$\begin{aligned} nat\ z &\triangleq \top. \\ nat\ (sX) &\triangleq nat\ X. \end{aligned}$$

In our setting, pattern matching is encoded explicitly as equations, e.g.,

$$nat\ X \triangleq (X = z) \vee \exists Y. (X = (sY) \wedge nat\ Y).$$

These two presentations are equivalent. We use the former in presenting examples and the latter for meta-theoretic discussions.

Introduction rules for definitions and equality

$$\frac{\Gamma \vdash \sigma \triangleright B \bar{t}}{\Gamma \vdash \sigma \triangleright p \bar{t}} \text{def}\mathcal{R}$$

$$\frac{\Gamma, \sigma \triangleright B \bar{t} \vdash \mathcal{C}}{\Gamma, \sigma \triangleright p \bar{t} \vdash \mathcal{C}} \text{def}\mathcal{R}$$

where $\forall \bar{x}. [p \bar{x} \stackrel{\Delta}{=} B \bar{x}]$

$$\frac{\{\Gamma \theta \vdash C \theta \mid \theta \in CSU(\lambda \bar{x}.s, \lambda \bar{x}.t)\}}{\bar{x} \triangleright s = t, \Gamma \vdash C} \text{eq}\mathcal{L}$$

$$\overline{\Gamma \vdash \sigma \triangleright t = t}$$

Specification of π -calculus

We consider the finite fragment of π -calculus (no replication or recursion):

$$P := 0 \mid \tau.P \mid x(y).P \mid \bar{x}y.P \mid (P \mid P) \mid (P + P) \mid (x)P \mid [x = y]P$$

To encode the process expressions in λ -tree syntax, we introduce the types n (names), p (processes), and the following constants

$$\begin{aligned} 0 &: p, & \tau &: p \rightarrow p, & out &: n \rightarrow n \rightarrow p \rightarrow p, & in &: n \rightarrow (n \rightarrow p) \rightarrow p, \\ + &: p \rightarrow p \rightarrow p, & | &: p \rightarrow p \rightarrow p, & match &: n \rightarrow n \rightarrow p \rightarrow p, & \nu &: (n \rightarrow p) \rightarrow p. \end{aligned}$$

Translation to λ -tree syntax: process expressions

$$\begin{aligned}\llbracket 0 \rrbracket &= 0, \\ \llbracket [x = y]P \rrbracket &= \textit{match } x \ y \ \llbracket P \rrbracket, \\ \llbracket \bar{x}y.P \rrbracket &= \textit{out } x \ y \ \llbracket P \rrbracket, \\ \llbracket x(y).P \rrbracket &= \textit{in } x \ \lambda y. \llbracket P \rrbracket, \\ \llbracket P + Q \rrbracket &= \llbracket P \rrbracket + \llbracket Q \rrbracket, \\ \llbracket P|Q \rrbracket &= \llbracket P \rrbracket \mid \llbracket Q \rrbracket, \\ \llbracket \tau.P \rrbracket &= \tau \ \llbracket P \rrbracket, \\ \llbracket (x)P \rrbracket &= \nu \lambda x. \llbracket P \rrbracket.\end{aligned}$$

Translation to λ -tree syntax: one-step transitions

We introduce the type a to denote *actions*. One-step transition judgments are encoded as follows.

$$\begin{array}{ll} \llbracket P \xrightarrow{\bar{x}y} Q \rrbracket = \llbracket P \rrbracket \xrightarrow{\uparrow xy} \llbracket Q \rrbracket & \llbracket P \xrightarrow{\tau} Q \rrbracket = \llbracket P \rrbracket \xrightarrow{\tau} \llbracket Q \rrbracket \\ \llbracket P \xrightarrow{x(y)} Q \rrbracket = \llbracket P \rrbracket \xrightarrow{\downarrow x} \lambda y. \llbracket Q \rrbracket & \llbracket P \xrightarrow{\bar{x}(y)} Q \rrbracket = \llbracket P \rrbracket \xrightarrow{\uparrow x} \lambda y. \llbracket Q \rrbracket \end{array}$$

where $\downarrow: n \rightarrow n \rightarrow a$ encodes *input action* and $\uparrow: n \rightarrow n \rightarrow a$ encodes *output action*. Transitions involving *bound actions* are encoded as

$$\cdot \xrightarrow{\cdot} \cdot : p \rightarrow a \rightarrow (n \rightarrow p)$$

Specification of (late) transition system of π -calculus

Some sample clauses:

$$\frac{P \xrightarrow{A} Q}{(x)P \xrightarrow{\alpha} (x)Q} \quad x \notin \text{fn}(\alpha)$$

$$\begin{aligned} \nu x.Mx &\xrightarrow{A} \nu x.Nx \triangleq \nabla x(Mx \xrightarrow{A} Nx). \\ \nu x.Mx &\xrightarrow{A} \lambda y \nu x.Nxy \triangleq \nabla x(Mx \xrightarrow{A} Nx). \end{aligned}$$

$$\frac{P \xrightarrow{\alpha} Q}{[x = x]P \xrightarrow{\alpha} Q}$$

$$\begin{aligned} \text{match } x \ x \ P &\xrightarrow{A} Q \triangleq P \xrightarrow{A} Q. \\ \text{match } x \ x \ P &\xrightarrow{A} M \triangleq P \xrightarrow{A} M. \end{aligned}$$

$$\frac{P \xrightarrow{\bar{x}w} P'}{(y)P \xrightarrow{\bar{x}(w)} P'[w/y]} \quad \begin{array}{l} y \neq x, \\ w \notin \text{fn}((y)P') \end{array}$$

$$\nu y.My \xrightarrow{\uparrow^X} M' \triangleq \nabla y(My \xrightarrow{\uparrow^X y} M'y).$$

Example: one-step transition

Consider the process $(y)[x = y]\bar{x}z.0$. It cannot make any transition, since y has to be “fresh”.

$$\begin{array}{c}
 \frac{}{\{x, z, Q, \alpha\} ; y \triangleright ([x = y](\bar{x}z.0) \xrightarrow{\alpha} Q) \vdash \perp} \text{def}\mathcal{L} \\
 \frac{}{\{x, z, Q, \alpha\} ; . \triangleright \nabla y.([x = y](\bar{x}z.0) \xrightarrow{\alpha} Q) \vdash \perp} \nabla\mathcal{L} \\
 \frac{}{\{x, z, Q, \alpha\} ; . \triangleright ((y)[x = y](\bar{x}z.0) \xrightarrow{\alpha} Q) \vdash \perp} \text{def}\mathcal{L} \\
 \frac{}{\{x, z, Q, \alpha\} ; \vdash . \triangleright ((y)[x = y](\bar{x}z.0) \xrightarrow{\alpha} Q) \supset \perp} \supset \mathcal{R}
 \end{array}$$

The success of $\text{def}\mathcal{L}$ depends on the failure of unification problem

$$\lambda y.x = \lambda y.y.$$

Specification of late bisimulation

$$\begin{aligned}
l\text{bisim } P \ Q \triangleq & \ \forall A \forall P' \ [(P \xrightarrow{A} P') \supset \exists Q'. (Q \xrightarrow{A} Q') \wedge l\text{bisim } P' \ Q'] \wedge \\
& \forall A \forall Q' [(Q \xrightarrow{A} Q') \supset \exists P'. (P \xrightarrow{A} P') \wedge l\text{bisim } Q' \ P'] \wedge \\
& \forall X \forall P' [(P \xrightarrow{\downarrow X} P') \supset \exists Q'. (Q \xrightarrow{\downarrow X} Q') \wedge \forall w. l\text{bisim } (P'w) \ (Q'w)] \wedge \\
& \forall X \forall Q' [(Q \xrightarrow{\downarrow X} Q') \supset \exists P'. (P \xrightarrow{\downarrow X} P') \wedge \forall w. l\text{bisim } (Q'w) \ (P'w)] \wedge \\
& \forall X \forall P' [(P \xrightarrow{\uparrow X} P') \supset \exists Q'. (Q \xrightarrow{\uparrow X} Q') \wedge \nabla w. l\text{bisim } (P'w) \ (Q'w)] \wedge \\
& \forall X \forall Q' [(Q \xrightarrow{\uparrow X} Q') \supset \exists P'. (P \xrightarrow{\uparrow X} P') \wedge \nabla w. l\text{bisim } (Q'w) \ (P'w)]
\end{aligned}$$

The need for case split on names

The previous specification is *not* complete. Consider $P = x(u).(\tau.\tau + \tau)$ and $Q = x(u).(\tau.\tau + \tau + \tau.[u = y]\tau)$ and checking $P \sim Q$.

$$Q \xrightarrow{x(u)} \tau.\tau + \tau + \tau.[u = y]\tau \xrightarrow{\tau} [u = y]\tau$$

$$P \xrightarrow{x(u)} \tau.\tau + \tau \longrightarrow ?$$

Here the names u and y are encoded as eigenvariables in logic. The possible matching moves of P :

- (1) $\tau.\tau + \tau \xrightarrow{\tau} \tau$ if u is equal to y
- (2) $\tau.\tau + \tau \xrightarrow{\tau} 0.$, if u is not equal to y

Since name is not inductively defined, this case split cannot be captured intuitionistically.

Adequacy of Ibisim

To get a complete encoding of late bisimulation we need to add explicitly excluded middle on names, i.e., the set

$$\mathcal{E} = \left\{ \begin{array}{l} \forall x \forall y (x = y \vee x \neq y), \\ \nabla n_1 \forall x \forall y (x = y \vee x \neq y), \\ \nabla n_1 \nabla n_2 \forall x \forall y (x = y \vee x \neq y), \\ \vdots \end{array} \right\}$$

Theorem 1. *Let P and Q be two processes and let $\bar{n}$ be the free names in P and Q . Then $P \sim_l Q$ if and only if the sequent*

$$.; \mathcal{X} \vdash \nabla \bar{n}. \text{Ibisim } P \ Q$$

is provable for some $\mathcal{X} \subseteq_f \mathcal{E}$.

Open bisimulation

What does $\text{Ibisim } P \ Q$ correspond to (without the excluded middle assumption)?

It turns out that if we drop the excluded middle axiom and $\forall$ -quantify all the free names in P and Q we get a sound and complete encoding of *open bisimulation* [Sangiorgi].

Names in open bisimulation are instantiated lazily, and this bisimulation is closed under arbitrary substitution of free names.

Automation of proof search

- Proof search strategy for open bisimulation: apply left-introduction rules if applicable, otherwise apply right introduction rules. This proof strategy is *complete*.
- Proof search strategy for late bisimulation: need to know when to use the excluded middle on names. A complete proof strategy would be to do case split early, e.g., doing $n + 1$ case split for processes with n distinct free names.

Implementation

- Implementation is rather straightforward: it is based on known old technology, e.g., HOAS based on Church's simply typed λ -calculus, higher-order pattern unification, and other related implementation techniques for logic programming.
- We currently have a working prototype implementation for (automated) checking of open bisimulation. Available from the web: <http://www.loria.fr/~tiu>

Related work

- Formalization of π -calculus has been done in different frameworks, e.g., in Coq [Hirschkoﬀ TPHOL'97, Despeyroux IFIP-TCS'00, Honsell et. al. TCS'01], in nominal logic [Gabbay'03].
- In those works, formalizing freshness of names, scoping, substitution, etc., is considerable work. In our framework, the same work has to be done, but it is done as a part of the description of the logic. Formalization of open bisimulation, as far as we know, has not been done prior to our work.
- Formalization of π -calculus with replication has been done in [Tiu'04].

Future work

- Relation with *symbolic bisimulation*. Find correspondence between decision procedure in, e.g., [Hennessy and Lin, TCS'95] and [Boreale and Nicola, IC. 96], to proof search.
- Formalizing other systems. Current work in progress: spi-calculus and framed bisimulation, modal logics for π -calculus.
- Proof procedures for bisimulation checking for non-terminationg processes, e.g., via *circular proofs*, tabled deduction, domain specialized proof procedures, etc.

Authors: Tarmo Uustalu, Varmo Vene

Title: Signals and comonads

Abstract:

We give a practical demonstration of the claim that, just as monads can be used as an abstraction of notions of computation, comonads abstract notions of values-in-context. We discuss two comonads whose coKleisli arrows are general resp. causal stream functions, i.e., transformers of signals in discrete time. In particular, we discuss the application of such comonads as modularity tools in programming and in semantic descriptions of intensional and dataflow languages a la Lucid and Lustre. We also compare our approach to that of "arrows" by J. Hughes.

(This abstract and title are tentative, I might wish to detail them.)

Signals and Comonads

Tarmo UUSTALU

Varmo VENE

TYPES'04, Jouy en Josas, 15–18 Dec. 2004

Motivation

- Following Moggi and Wadler, it is standard in programming and semantics to analyze various notions of computation with side-effects as monads.
- But there is a need for both finer and more permissive mathematical abstractions to uniformly describe the numerous function-like concepts encountered in programming.
- Hughes has promoted arrows as an abstraction especially handy in programming with signals/flows.
- This has been taken up in functional reactive programming; there is by now both a library and specialized syntax for arrows in Haskell.
- However, there is also the standard concept of comonad which has not found extensive use (some examples by Kieburtz, but mostly artificial).

This talk

- Claim: Used properly, comonads are exactly the right tool for programming signal/flow functions, accounting both for general signal functions and for causal ones (where the output at a given time can only depend on the input until that time).
- This extends Moggi's modular approach to language semantics to languages for implicit context based paradigms such as intensional programming in Lucid or synchronous dataflow programming in Lustre: context relying functions are interpreted as pure functions via a comonad translation.

For such languages, Moggi-style accounts have not been available thus far.

Outline

- Monads, monads in programming and semantics
- Arrows and programming with signals
- Comonads for programming with signals, semantics

Monads

- A monad (in the Kleisli format) on a category $(C, \text{id}, \circ)$ is given by a mapping $T : |C| \rightarrow |C|$ together with a $|C|$ -indexed family η of maps $\eta_A : A \rightarrow TA$ (**unit**), and an operation $-^\star$ taking every map $k : A \rightarrow TB$ in C to a map $k^\star : TA \rightarrow TB$ (**extension operation**) such that
 - for any $f : A \rightarrow TB$, $k^\star \circ \eta_A = k$,
 - $\eta_A^\star = \text{id}_{TA}$,
 - for any $k : A \rightarrow TB$, $\ell : B \rightarrow TC$, $(\ell^\star \circ k)^\star = \ell^\star \circ k^\star$.
- Any monad $(T, \eta, -^\star)$ defines a category $(C_T, \text{jd}, \bullet)$ where $|C_T| = |C|$ and $C_T(A, B) = C(A, TB)$, $\text{jd}_A = \eta_A$, $\ell \bullet k = \ell^\star \circ k$ (Kleisli category).

- In programming and semantics, monads are used to model notions of computation with a side-effect; TA is the type of computations of values of A .

An function with effect from A to B is a map $A \rightarrow B$ in the Kleisli category, which is a map $A \rightarrow TB$ in the base category.

- Some examples applied in semantics:
 - $TA = A + E$, error/exceptions,
 - $TA = E \Rightarrow A$, environment,
 - $TA = \text{List}A = \mu X.1 + A \times X$, non-determinism,
 - $TA = S \Rightarrow A \times S$, state,
 - $TA = (A \Rightarrow R) \Rightarrow R$, continuations,
 - $TA = \mu X.A + (U \Rightarrow X)$, interactive input,
 - $TA = \mu X.A + Y \times X \cong A \times \text{List}Y$, interactive output,
 - $TA = \mu X.A + FX$, the free monad over F ,
 - $TA = \nu X.A + FX$, the free completely iterative monad over F .

Monads in Haskell

- The monad class is defined in the Prelude:

```
class Monad t where
    return :: a -> t a
    (>>=)  :: t a -> (a -> t b) -> t b
```

- The error monad:

```
instance Monad Maybe where
    return      = Just
    Just a >>= k = k a
    Nothing >>= k = Nothing
```

```
error :: Maybe b
error = Nothing
```

- The non-determinism monad:

```
instance Monad [] where
    return a    = [a]
    []          >>= f = []
    (a : as) >>= f = f a ++ (as >>= f)
```

```
choose :: (a, a) -> [a]
choose a0 a1 = [a0, a1]
```

```
deadlock :: [a]
deadlock = []
```


Arrows

- Arrows are a generalization of Kleisli arrows of a monad. They were invented in functional programming by Hughes to model function-like entities such as stream functions.
- Arrows in Haskell (as in `Control.Arrow`):

```
class Arrow r where
```

```
  arr :: (a -> b) -> r a b
```

```
  (>>>) :: r a b -> r b c -> r a c
```

```
  first :: r a b -> r (a, c) (b, c)
```

```
  second :: r a b -> r (d, a) (d, b)
```

```
  second f = arr swap >>> first f >>> arr swap
```

```
  (***) :: r a b -> r a' b' -> r (a, a') (b, b')
```

```
  f *** g = first f >>> second g
```

```
  (&&&) :: r a b -> r a b' -> r a (b, b')
```

```
  f &&& g = arr (\ a -> (a, a)) >>> f *** g
```

- The stream functions arrow (to model signal transformers in discrete time):

```
data Stream a = a :< Stream a                                -- coinductive

newtype SF a b = SF (Stream a -> Stream b)

instance Arrow SF where
    arr f = SF (\ a :< as -> f a :< arr f as)
    SF k >>> SF l = SF (l . k)
    first SF k = SF (uncurry zip . (\ (as, ds) -> k as, ds) . unzip)

delay :: a -> SF a a
delay a0 = SF (\ as -> a0 :< as)
```

- Categorically, arrows are not so simple as monads (they are premonoidal or indexed categories with extra structure), nor are they as fundamental ...

Comonads

- Comonads are the formal dual of monads.
- A comonad on a category $(C, \text{id}, \circ)$ is given by a mapping $D : |C| \rightarrow |C|$ together with a $|C|$ -indexed family η of maps $\varepsilon_A : DA \rightarrow A$ (counit), and an operation $-^\star$ taking every map $k : DA \rightarrow B$ in C to a map $k^\dagger : DA \rightarrow DB$ (coextension operation) such that
 - for any $k : DA \rightarrow B$, $\varepsilon_B \circ k^\dagger = k$,
 - $\varepsilon_A^\dagger = \text{id}_{TA}$,
 - for any $k : DA \rightarrow B$, $\ell : DB \rightarrow C$, $(\ell \circ k^\dagger)^\dagger = \ell^\dagger \circ k^\dagger$.
- Any comonad $(D, \varepsilon, -^\dagger)$ defines a category $(C_D, \text{jd}, \bullet)$ where $|C_D| = |C|$ and $C_D(A, B) = C(DA, B)$, $\text{jd}_A = \varepsilon_A$, $\ell \bullet k = \ell \circ k^\dagger$ (coKleisli category).

- Comonads should be usable to model notions of value-in-context; DA would be the type of contextually situated values of A .
A context-relying functions from A to B would be a map $A \rightarrow B$ in the coKleisli category, i.e., a map $DA \rightarrow B$ in the base category.
- Some examples:
 - $DA = A \times E$, the product comonad,
 - $DA = \text{Str}A = \nu X. A \times X$, the streams comonad,
 - $DA = \nu X. A \times FX$, the cofree comonad over F ,
 - $DA = \mu X. A \times FX$, the cofree recursive comonad over F .

Comonads in Haskell

- The basic implementation:

```
class Comonad d where
  counit  :: d a -> a
  cobind  :: (d a -> b) -> d a -> d b
```

- The product comonad:

```
data With e a = a :- e

instance Comonad (With e) where
  counit (a :- _) = a
  cobind k p@(_ :- e) = k p :- e
```

- The streams comonad:

```
data Stream a = a :< Stream a                -- coinductive
```

```
instance Comonad Stream where
```

```
  counit (a :< _) = a
```

```
  cobind k s@(_ :< as) = k s :< cobind k as
```

Comonads for signal transformers (discrete time)

- Streams (discrete-time signals) are isomorphic to functions from natural numbers: $\text{Str}A \cong \text{Nat} \Rightarrow A$.
- General stream functions $\text{Str}A \rightarrow \text{Str}B$ are thus in natural bijection with maps $(\text{Nat} \Rightarrow A) \times \text{Nat} \rightarrow B$.
- Hence the values of A in context for general stream functions are $D^g A = (\text{Nat} \Rightarrow A) \times \text{Nat} \cong \text{Str}A \times \text{Nat} \cong \text{List}A \times A \times \text{Str}A$.
- The values of A in context for causal stream functions are $D^c A = \text{List}A \times A \cong \mu X. A \times (1 + X)$.

This is the cofree recursive comonad over the functor $X \mapsto 1 + X$ (i.e., the Maybe functor).

- The idea is that at any time, the past of a value is a (snoc) list and the future is a stream.

- The isomorphism of streams to functions from naturals:

```
data Stream a = a :< Stream a                -- coinductive
```

```
str2fun :: Stream a -> Int -> a
```

```
str2fun (a :< as) 0 = a
```

```
str2fun (a :< as) (i + 1) = str2fun as i
```

```
fun2str :: (Int -> a) -> Stream a
```

```
fun2str f = fun2str' f 0
```

```
fun2str' f i = f i :< fun2str' f (i + 1)
```


- Streams with a marked position: values-in-context for general stream functions:

```
data StrPos a = SP (Stream a) Int
```

```
instance Comonad StrPos where
```

```
  counit (SP as i) = str2fun as i
```

```
  cobind k (SP as i) = SP (fun2str (\ i' -> k (SP as i')) i
```

```
runSP :: (StrPos a -> b) -> Stream a -> Stream b
```

```
runSP k as = runSP' k as 0
```

```
runSP' k as i = k (SP as i) :< runSP' k as (i + 1)
```

- Delay:

```
delaySP :: a -> StrPos a -> a
```

```
delaySP a (SP as 0) = a
```

```
delaySP _ (SP as (i + 1)) = str2fun as i
```

- Summation:

```
sumSP :: Num a => StrPos a -> a
```

```
sumSP (SP as 0) = str2fun as 0
```

```
sumSP (SP as (i + 1)) = str2fun as (i + 1) + sumSP (SP as i)
```

- Compression (non-causal!):

```
compress :: StrPos a -> (a, a)
```

```
compress (SP as i) = (str2fun as (2 * i), str2fun as (2 * i + 1))
```

- List-element pairs, values in context for causal stream functions:

```
data List a = Nil | List a :> a -- inductive
```

```
data LE a = List a := a
```

```
instance Comonad LE where
```

```
  counit (_ := a) = a
```

```
  cobind k le@(al := _) = cobindp al := k le where
```

```
    cobindp Nil = Nil
```

```
    cobindp (al' :> a') = cobindp al' :> k (al' := a')
```

```
runLE :: (LE a -> b) -> Stream a -> Stream b
```

```
runLE k (a :< as) = runLE' k Nil a as
```

```
runLE' k al a (a' :< as')
```

```
    = k (al := a) :< runLE' k (al :> a) a' as'
```

- A feedback resolution combinator:

```
feedback :: (List (a, b) -> a -> b) -> (LE a -> b)
feedback k al = k abl' a
               where (abl' := (a, _))
                   = cobind (pair counit (feedback k)) al
```

- Feedbacks can be run directly:

```
runbase :: (List (a, b) -> a -> b) -> Stream a -> Stream b
runbase k (a :< as) = runbase' k Nil a as

runbase' k abl a (a' :< as')
    = b :< runbase' k (abl :> (a, b)) a' as'
    where b = k abl a
```

- Feedbacks can also be composed directly:

```
combase :: (List (a, b) -> a -> b)
        -> (List ((a, b), c) -> (a, b) -> c)
        -> List (a, (b, c)) -> a -> (b, c)

combase k l e a
  = let
      e'  = fmap (\ (a, (b, c)) -> (a, b)) e
      e'' = fmap (\ (a, (b, c)) -> ((a, b), c)) e
      b   = k e' a
      c   = l e'' (a, b)
  in (b, c)
```

- Delay:

```
delayLE :: a -> LE a -> a
```

```
delayLE a0 (Nil      := _) = a0
```

```
delayLE _  ((_ :=> a') := _) = a'
```

- Summation directly and with feedback:

```
sumLE :: Num a => LE a -> a
```

```
sumLE (Nil      := a) = a
```

```
sumLE ((a1' :=> a') := a) = sumLE (a1' := a') + a
```

```
sumbase : Num a => List (a, a) -> a -> a
```

```
sumbase Nil a = a
```

```
sumbase (_ :=> (_, b)) a = b + a
```

Conclusions and Future Work

- A general framework for signal/flow based programming and for semantics.
- Based on a well-understood mathematical construction—comonad—, allowing generalizations from signal/flow processing to more sophisticated implicit context based paradigms of programming.
- Allows for modular simultaneous use of multiple notions of context via combinations of comonads (e.g., multiple dimensions in multidimensional Lucid).
- In progress: From discrete time to continuous time, from clock-tick-based to event-based programming with signals.
- In progress: Semantics of languages like Lucid (“intensional”, general stream functions), Lustre/Lucid Synchrone (synchronous dataflow, causal stream functions).

Authors: Luigi Liquori, Arnaud Spiwack

Title: Functional Method

Abstract:

This paper introduces a functional calculus that features objects and message sending via functional application. A sound (message-not-found preventing) first-order type system with recursive-types and width subtyping is presented.

OhML

(The “Zen Essence” of Methods-as-Functions)

$$(e.m) \rightsquigarrow (m\ e)$$

Arnaud Spiwack

ENS Cachan

joint work with

Luigi Liquori

INRIA Sophia

OhML

Motivations

Modeling Method Call via Function Application

- A modular integer would feature:
 - ★ An implicit type: `int`
 - ★ An addition function `plus : int → int → int`
 - ★ Addition will be performed like: `plus 3 5`

Modeling Method Call via Function Application

- A modular integer would feature:
 - ★ An implicit type: `int`
 - ★ An addition function `plus : int → int → int`
 - ★ Addition will be performed like: `plus 3 5`
- An objective integer would be:
 - ★ A class: `int`
 - ★ Featuring an addition method `plus : int → int`
 - ★ Addition will be performed like: `3.plus 5`
 - ★ Moreover `plus` is not “first-class”, *i.e.* it cannot be passed as an argument or be a result of a call

Modeling Method Call via Function Application

- A modular integer would feature:
 - ★ An implicit type: `int`
 - ★ An addition function `plus : int → int → int`
 - ★ Addition will be performed like: `plus 3 5`
- An objective integer would be:
 - ★ A class: `int`
 - ★ Featuring an addition method `plus : int → int`
 - ★ Addition will be performed like: `3.plus 5`
 - ★ Moreover `plus` is not “first-class”, *i.e.* it cannot be passed as an argument or be a result of a call
- Though they have the same semantics, they have a radically different syntax. And functions designed to work with one do not work straightforwardly with the other

Simple, Uniform, and Powerful

- Simple functional calculus *à la* ML with fixpoint, simple objects with `this`, no inheritance and message sending via functional application ... this means that for

-

$\text{point} \triangleq \text{obj } \text{this} \rightarrow \text{data } x = 1 \text{ data } \text{getx} = (x \text{ this})$

... we have that:

$$\begin{aligned}
 (\text{fun } z \rightarrow (z \text{ point})) \underbrace{((\text{fun } z \rightarrow z) \text{ getx})} &\rightarrow \frac{(\text{fun } z \rightarrow (z \text{ point})) \text{ getx}}{(\text{getx point})} \\
 &\rightarrow \frac{(\text{getx point})}{(x \text{ point})} \\
 &\rightarrow 1
 \end{aligned}$$

Simple, Uniform, and Powerful

- Simple functional calculus *à la* ML with fixpoint, simple objects with `this`, no inheritance and message sending via functional application ... this means that for

-

$\text{point} \triangleq \text{obj } \text{this} \rightarrow \text{data } x = 1 \text{ data } \text{getx} = (x \text{ this})$

... we have that:

$$\begin{aligned}
 (\text{fun } z \rightarrow (z \text{ point})) \underbrace{((\text{fun } z \rightarrow z) \text{ getx})} &\rightarrow \frac{(\text{fun } z \rightarrow (z \text{ point})) \text{ getx}}{(\text{getx point})} \\
 &\rightarrow \frac{(\text{getx point})}{(x \text{ point})} \\
 &\rightarrow 1
 \end{aligned}$$

- A sound (message-not-found preventing) first-order type system with width subtyping

Simple, Uniform, and Powerful

- Simple functional calculus *à la* ML with fixpoint, simple objects with `this`, no inheritance and message sending via functional application ... this means that for

-

$$\text{point} \triangleq \text{obj } \text{this} \rightarrow \text{data } x = 1 \text{ data } \text{getx} = (x \text{ this})$$

... we have that:

$$\begin{aligned}
 (\text{fun } z \rightarrow (z \text{ point})) \underbrace{((\text{fun } z \rightarrow z) \text{ getx})} &\rightarrow \frac{(\text{fun } z \rightarrow (z \text{ point})) \text{ getx}}{(\text{getx point})} \\
 &\rightarrow \frac{(\text{getx point})}{(x \text{ point})} \\
 &\rightarrow 1
 \end{aligned}$$

- A sound (message-not-found preventing) first-order type system with width subtyping
- Decidable (?) type inference (plug'n play with Damas-Milner W)

Summarizing

- An alternative to **row-polymorphism** *à la* Wand, just record-types and simple subtyping, no polymorphism (yet)
- Method names are first-class citizens, *i.e.* they can be passed as function arguments (in a single dispatch setting)

$$(\text{fun } x \rightarrow x) \text{ get } x \mapsto \text{get } x$$

Summarizing

- An alternative to **row-polymorphism** *à la* Wand, just record-types and simple subtyping, no polymorphism (yet)
- Method names are first-class citizens, *i.e.* they can be passed as function arguments (in a single dispatch setting)

$$(\text{fun } x \rightarrow x) \text{ get } x \mapsto \text{get } x$$

- Method call reduced to function application (plus a method lookup of course!)

$$m \text{ (obj } x \rightarrow o) \mapsto mbody(o; m)[\text{obj } x \rightarrow o/x]$$

- Inheritance is out of the scope of this paper; however, it would not be difficult to add an object-based inheritance
- A decidable type inference algorithm is our current challenge

Summarizing

- An alternative to **row-polymorphism** *à la* Wand, just record-types and simple subtyping, no polymorphism (yet)
- Method names are first-class citizens, *i.e.* they can be passed as function arguments (in a single dispatch setting)

$$(\text{fun } x \rightarrow x) \text{ get } x \mapsto \text{get } x$$

- Method call reduced to function application (plus a method lookup of course!)

$$m \text{ (obj } x \rightarrow o) \mapsto mbody(o; m)[\text{obj } x \rightarrow o/x]$$

- Inheritance is out of the scope of this paper; however, it would not be difficult to add an object-based inheritance
- A decidable type inference algorithm is our current challenge
- **OhML** use the most economic type theory for a rich list of features:
Great value for money guaranteed! :-)

OhML

The Syntax

OhML's Syntax

$$e ::= c \mid m \mid x \mid \text{fun } x \rightarrow e \mid \text{obj } x \rightarrow o \mid e \ e \mid$$
$$\text{let } x = e \text{ in } e \mid \text{fix}$$
$$o ::= \epsilon \mid o \text{ data } m = e$$

OhML

One Step Semantics

Values and One Step

$$v ::= c \mid x \mid m \mid \text{fun } x \rightarrow e \mid \text{obj } x \rightarrow o \mid$$
$$\text{fix } (\text{fun } x \rightarrow e) \mid \text{fix} \mid \text{error}$$

Values and One Step

$$v ::= c \mid x \mid m \mid \text{fun } x \rightarrow e \mid \text{obj } x \rightarrow o \mid \\ \text{fix } (\text{fun } x \rightarrow e) \mid \text{fix} \mid \text{error}$$

$$(\text{fun } x \rightarrow e) \ v \mapsto e[v/x] \quad (\beta_v)$$

$$m \ (\text{obj } x \rightarrow o) \mapsto \text{mbody}(o; m)[\text{obj } \dots / x](\text{call}_v)$$

$$\text{let } x = v \text{ in } e \mapsto e[v/x] \quad (\text{let}_v)$$

$$\text{fix } v_1 \ v_2 \mapsto v_1 \ (\text{fix } v_1) \ v_2 \quad (\delta_{\text{fix}})$$

Lookup Algorithm (No Inheritance Yet)

$$\frac{}{mbody(\epsilon ; m) = \text{error}} \quad (\text{Empty})$$

$$\frac{}{mbody(o \text{ data } m = e ; m) = e} \quad (\text{Top})$$

$$\frac{m \neq n}{mbody(o \text{ data } m = e ; n) = mbody(o ; m)} \quad (\text{Next})$$

Just try before I tell you the full story ...

```
let meth  = gtx in
let point = fun    x ->
               fun    y ->
                   obj this -> data gtx = x data gty = y in
let p = (point 3 4) in (meth p)
```

Just try before I tell you the full story ...

```
let meth  = gtx in
let point = fun    x ->
               fun    y ->
                   obj this -> data gtx = x data gty = y in
let p = (point 3 4) in (meth p)
→
let point = fun    x ->
               fun    y ->
                   obj this -> data gtx = x data gty = y in
let p = (point 3 4) in (gtx p)
```

Just try before I tell you the full story ...

```

let meth  = gtx in
let point = fun    x ->
               fun    y ->
                   obj this -> data gtx = x data gty = y in
let p = (point 3 4) in (meth p)
→
let point = fun    x ->
               fun    y ->
                   obj this -> data gtx = x data gty = y in
let p = (point 3 4) in (gtx p)
→
let p = ((fun    x -> fun    y ->
           obj this -> data gtx = x data gty = y) 3 4) in (gtx p)

```

Just try before I tell you the full story ...

```
let meth  = gtx in
let point = fun    x ->
              fun    y ->
                obj this -> data gtx = x data gty = y in
let p = (point 3 4) in (meth p)
→
let point = fun    x ->
              fun    y ->
                obj this -> data gtx = x data gty = y in
let p = (point 3 4) in (gtx p)
→
let p = ((fun    x -> fun    y ->
          obj this -> data gtx = x data gty = y) 3 4) in (gtx p)
→
let p = obj this -> data gtx = 3 data gty = 4 in (gtx p)
```

Just try before I tell you the full story ...

```

let meth  = gtx in
let point = fun    x ->
               fun    y ->
                   obj this -> data gtx = x data gty = y in
let p = (point 3 4) in (meth p)
→
let point = fun    x ->
               fun    y ->
                   obj this -> data gtx = x data gty = y in
let p = (point 3 4) in (gtx p)
→
let p = ((fun    x -> fun    y ->
           obj this -> data gtx = x data gty = y) 3 4) in (gtx p)
→
let p = obj this -> data gtx = 3 data gty = 4 in (gtx p)
→
gtx (obj this -> data gtx = 3 data gty = 4)

```

Just try before I tell you the full story ...

```

let meth  = gtx in
let point = fun    x ->
               fun    y ->
                   obj this -> data gtx = x data gty = y in
let p = (point 3 4) in (meth p)
→
let point = fun    x ->
               fun    y ->
                   obj this -> data gtx = x data gty = y in
let p = (point 3 4) in (gtx p)
→
let p = ((fun    x -> fun    y ->
           obj this -> data gtx = x data gty = y) 3 4) in (gtx p)
→
let p = obj this -> data gtx = 3 data gty = 4 in (gtx p)
→
gtx (obj this -> data gtx = 3 data gty = 4)
→ 3

```

OhML

Types

Type Syntax Contexts and Judgments

$$\tau ::= \mathbf{b} \mid \alpha \mid \langle \overline{\mathbf{m}} : \overline{\tau} \rangle \mid \tau \rightarrow \tau$$

Types

$$\Gamma ::= \epsilon \mid \Gamma, \mathbf{x} : \tau \mid \Gamma, \mathbf{c} : \tau$$

Contexts

$$\Gamma \vdash \mathbf{e} : \tau$$

Typing judg.

$$\tau_1 <: \tau_2$$

Subtyping judg.

Type Rules (I)

$$\frac{c:b \in \Gamma}{\Gamma \vdash c : b} \text{ (Const)} \qquad \frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ (Var)}$$

$$\frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2} \text{ (Fun)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \text{ (Appl)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x:\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{ (Let)}$$

Type Rules (II)

$$\frac{}{\Gamma \vdash \mathbf{m} : \langle \mathbf{m} : \tau \rangle \rightarrow \tau} \text{ (Meth)}$$

$$\frac{\tau \equiv \tau_1 \rightarrow \tau_2}{\Gamma \vdash \mathbf{fix} : (\tau \rightarrow \tau) \rightarrow \tau} \text{ (Fix)}$$

$$\frac{\Gamma \vdash \mathbf{e} : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash \mathbf{e} : \tau_2} \text{ (Sub)}$$

Type Rules (III)

$$o \triangleq \text{data } m_1 = e_1 \ \dots \ \text{data } m_n = e_n$$

$$\forall i, j = 1 \dots n. m_i \neq m_j$$

$$\tau \triangleq \langle m_1 : \tau_1 \ \dots \ m_n : \tau_n \rangle$$

$$\frac{\Gamma, x : \tau \vdash e_i : \tau_i \quad \forall i = 1 \dots n}{\Gamma \vdash \text{obj } x \rightarrow o : \tau} \text{ (Obj)}$$

SubType Rules (IV)

$$\frac{}{\tau <: \tau} \text{ (Refl)}$$

$$\frac{(\mathbf{b}_1, \mathbf{b}_2) \in \mathcal{R}}{\mathbf{b}_1 <: \mathbf{b}_2} \text{ (Taut)}$$

$$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \text{ (Trans)}$$

$$\frac{\tau_3 <: \tau_1 \quad \tau_2 <: \tau_4}{\tau_1 \rightarrow \tau_2 <: \tau_3 \rightarrow \tau_4} \text{ (Arr)}$$

$$\frac{h \geq k}{\langle \mathbf{m}_1 : \tau_1 \dots \mathbf{m}_h : \tau_h \rangle <: \langle \mathbf{m}_1 : \tau_1 \dots \mathbf{m}_k : \tau_k \rangle} \text{ (Width)}$$

OhML:

Examples

A simple functional point (with its typing)

Consider the point

```
let meth  = gtx in
let point = fun  x ->
              fun  y ->
                obj  this ->
                  data gtx = x
                  data gty = y
in let p = (point 3 4)
in (gtx p)
```

A simple functional point (with its typing)

Consider the point

```
let meth  = gtx in
let point = fun  x ->
              fun  y ->
                obj  this ->
                  data gtx = x
                  data gty = y
in let p = (point 3 4)
in (gtx p)
```

with typing

```
|- meth          : ⟨gtx:int gty:int⟩ -> int
|- point         : int -> int -> ⟨gtx:int gty:int⟩
|- (point 3 4)   : ⟨gtx:int gty:int⟩
|- gtx           : ⟨gtx:int gty:int⟩ -> int
|- (gtx p)       : int
```


OhML:

Metatheory

Properties

Spec If $\Gamma, x:\tau_1 \vdash e : \tau$ and $\tau_2 <: \tau_1$ then $\Gamma, x:\tau_2 \vdash e : \tau$.

Subst If $\Gamma, x:\tau_1 \vdash e_1 : \tau$ and $\Gamma \vdash e_2:\tau_1$ then $\Gamma \vdash e_1[e_2/x] : \tau$.

SR If $\Gamma \vdash e_1 : \tau$, and $e_1 \rightarrow e_2$, then $\Gamma \vdash e_2 : \tau$.

Progress If $\Gamma \vdash e : \tau$, then the evaluation of e cannot produce error, *i.e.* $e \not\rightarrow^* C[\text{error}]$ for every context $C[\cdot]$.

Progress If $\Gamma \vdash e_1 : \tau$ then, either e_1 is a value, or there exists e_2 such that $e_1 \rightarrow^* e_2$.

Conclusions and Agenda

- Already done:
 - ★ simple and uniform object-based language
 - ★ with subtyping (especially useful for traits)
 - ★ method as first-class citizens
 - ★ economic type system
- To be done:
 - ★ an inference algorithm
 - ★ featuring inheritance (and related issues)
 - ★ featuring polymorphism
 - ★ traits (featuring multiple inheritance)
 - ★ featuring references

Author: Markus Michelbrink

Title: Interfaces as games, programs as strategies

Abstract:

Peter Hancock and Anton Setzer developed the notion of interface to introduce interactive programming into dependent type theory. We generalise their notion and get an even simpler definition for interfaces. With this definition the relationship of interfaces to games becomes apparent. In fact from a game theoretical point of view interfaces are nothing than special games. Programs are strategies for these games. There is an obvious notion of refinement which coincides exactly with the intuition of what a refinement should do. Interfaces together with the refinement relation build a complete lattice (without least and greatest element) in the sense that every family of interfaces has a least common refinement and there is an interface such that all members of this family refine this interface and this interface is a refinement for every other interface with this property. A program/strategy on a refinement of an interface gives a strategy on the interface. We can define several operators on interfaces: tensor, par, choice, bang et cetera. Every notion has a dual notion by interchanging the viewpoint of player and opponent. Lollipop is defined as usual in terms of tensor and negation. We define the notion of fair strategy on $A \multimap B$. A fair strategy is essentially a pair of a strategy and a proof that this strategy plays in both games eventually. By a slight restriction of the notion of interface we are able to define a composition for fair strategies on $A \multimap B$ and $B \multimap C$. The obtained strategy is again fair. We can define fair strategies on $A \multimap A$ and $A \multimap (A \multimap B) \multimap B$. These strategies are versions of copy-cat strategies. Identifying strategies by some kind of behavioural equivalence we expect to receive a linear category. However we have not checked all details until now. All results so far can be achieved in intensional Martin-Loef Type Theory and are verified in the theorem prover Agda. There are numerous links to other areas including refinement calculus, linear logic, game semantics, formal topology, process algebra, model checking and so on. However we have not explored all these relationships until now but intend to do so in the future.

Interfaces as games, programs as strategies

Markus Michelbrink

Department of Computer Science
University of Wales, Swansea

TYPES 2004
Paris, December 2004

Interfaces

Due to Peter Hancock and Anton Setzer

An (state dependent) **Interface** is given by

- $S : \text{Set}$
a set of states S ,
- $C : S \rightarrow \text{Set}$
a set of commands $C\ s$ for every state $s : S$,
- $R : (s : S) \rightarrow C\ s \rightarrow \text{Set}$
a set of responses $R\ s\ c$ for every command $c : C\ s$ and state $s : S$,
- $\text{next} : (s : S) \rightarrow (c : C\ s) \rightarrow R\ s\ c \rightarrow S$
a function **next** giving a state **next** $s\ c\ r$ for every response $r : R\ s\ c$, every command $c : C\ s$ and every state $s : S$.

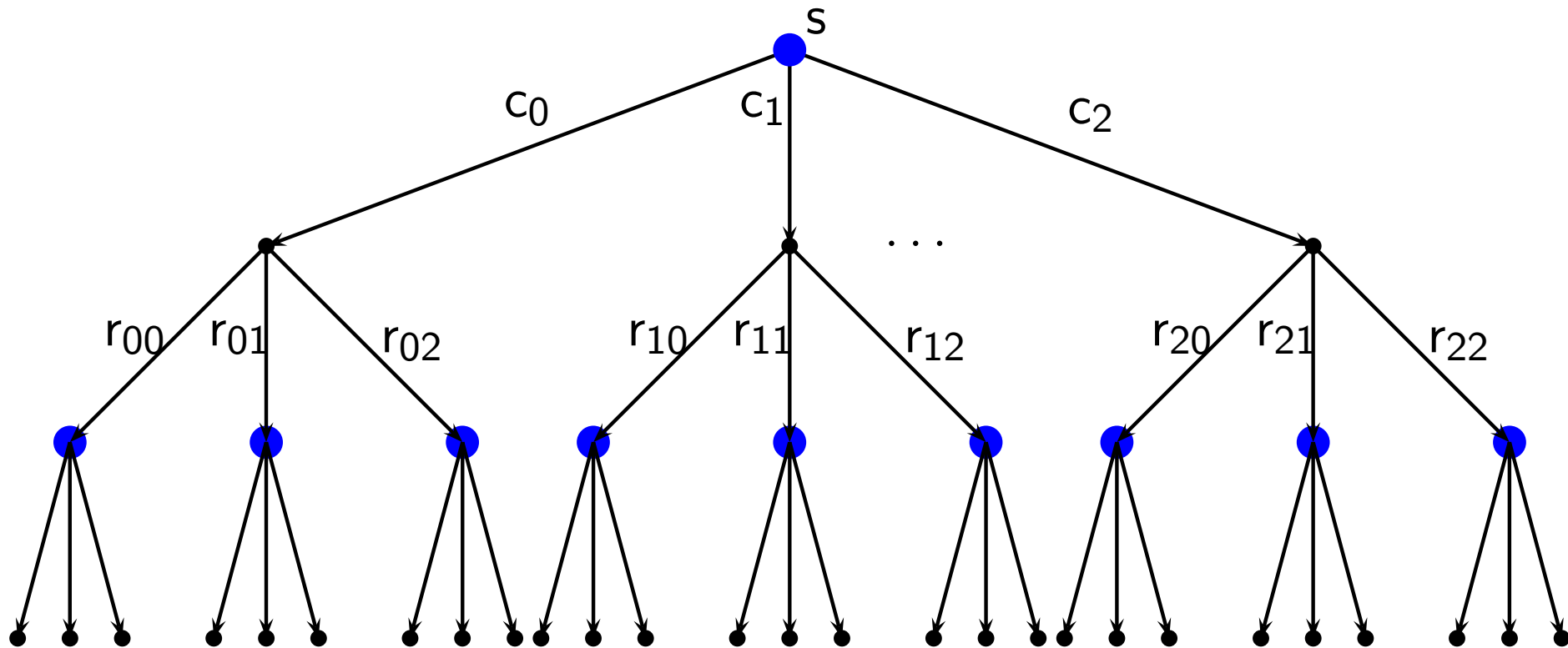
Programs

Due to Peter Hancock and Anton Setzer

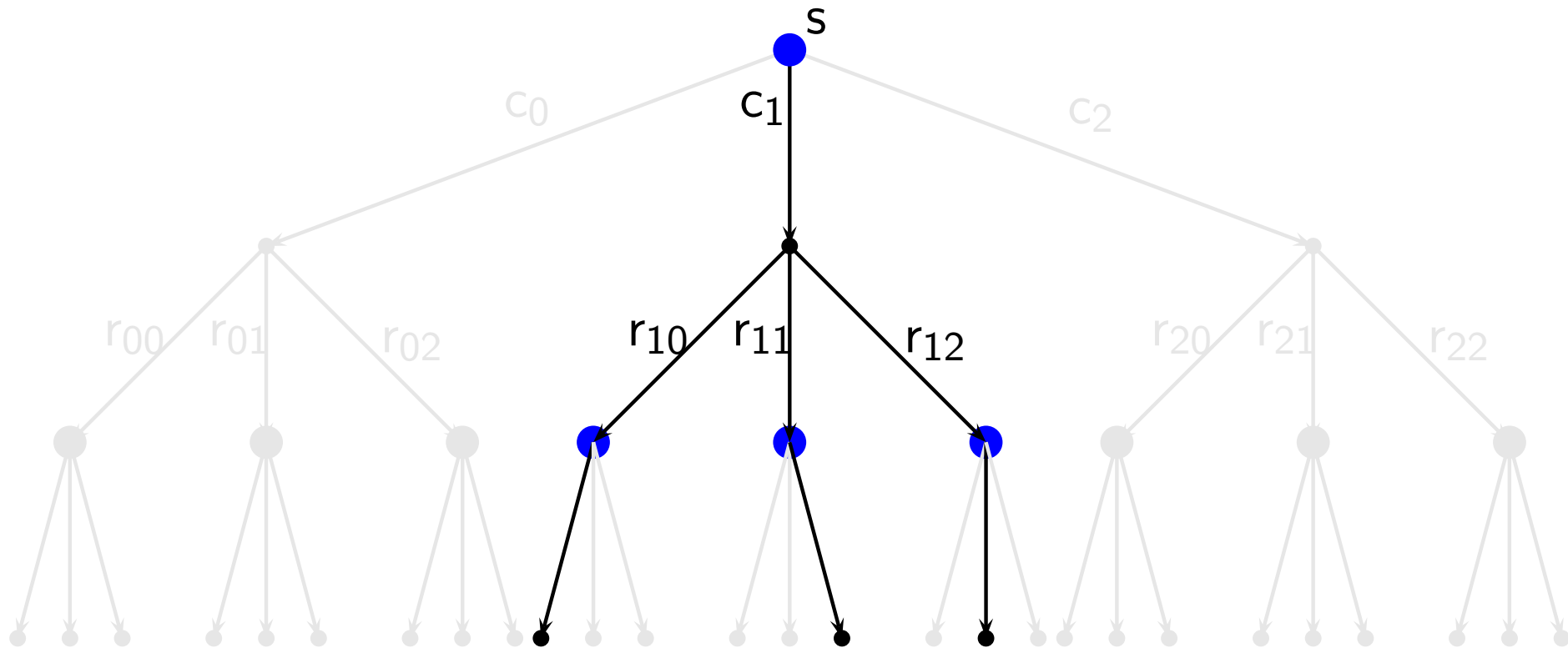
An interactive **Program** for such an Interface is given by

- $\text{IO} : S \rightarrow \text{Set}$
a set $\text{IO } s$ (of programs) for every state $s : S$,
- a function **onestep**
giving for every program $p : \text{IO } s$ and every state $s : S$ a command $c : C s$ and a function $f : (r : R s c) \rightarrow \text{IO } (\text{nxt } s c r)$ which gives for every response r to c a new program.
- an element $p : \text{IO } s$ for some state $s : S$.

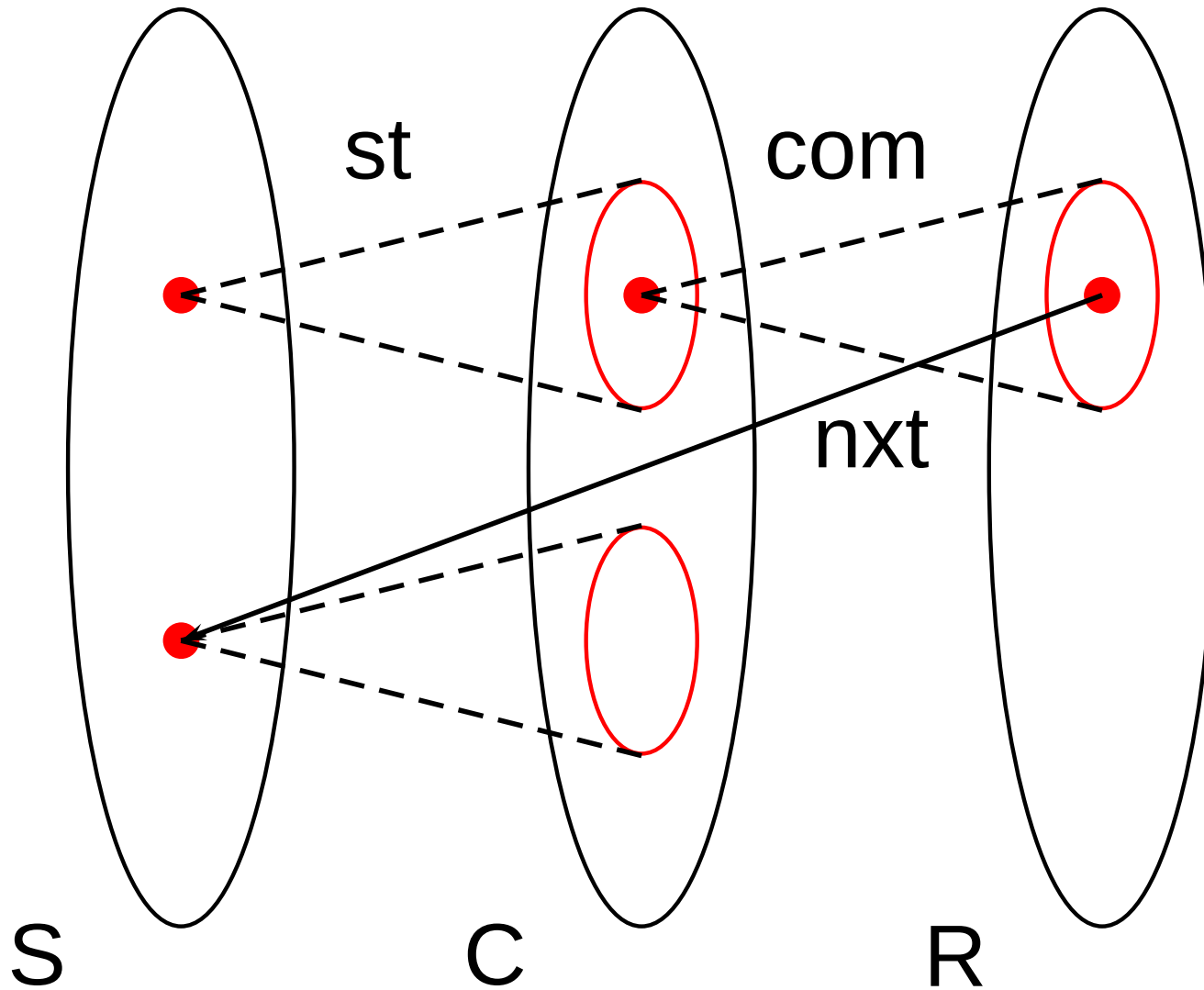
Interfaces



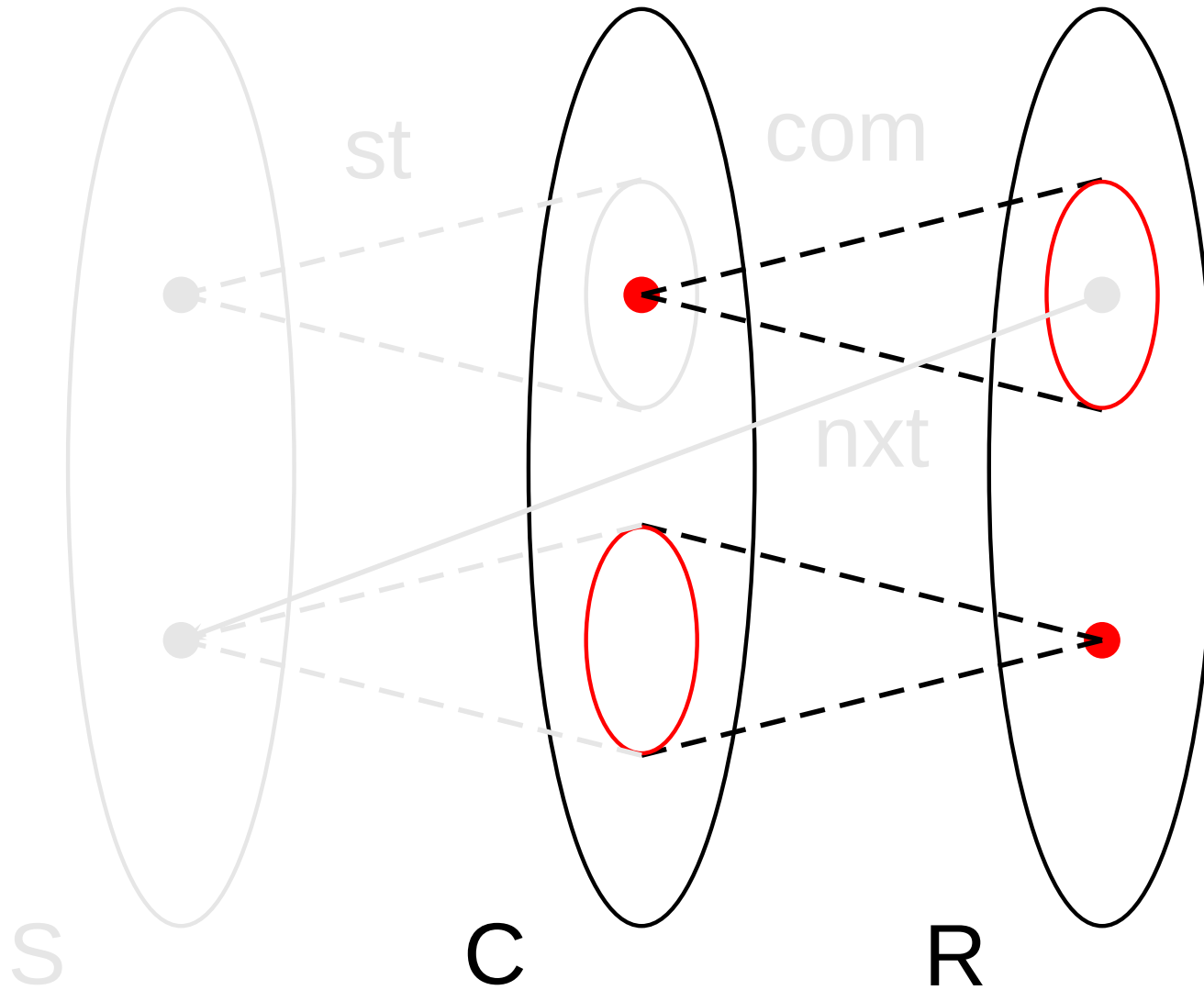
Programs



Motivation



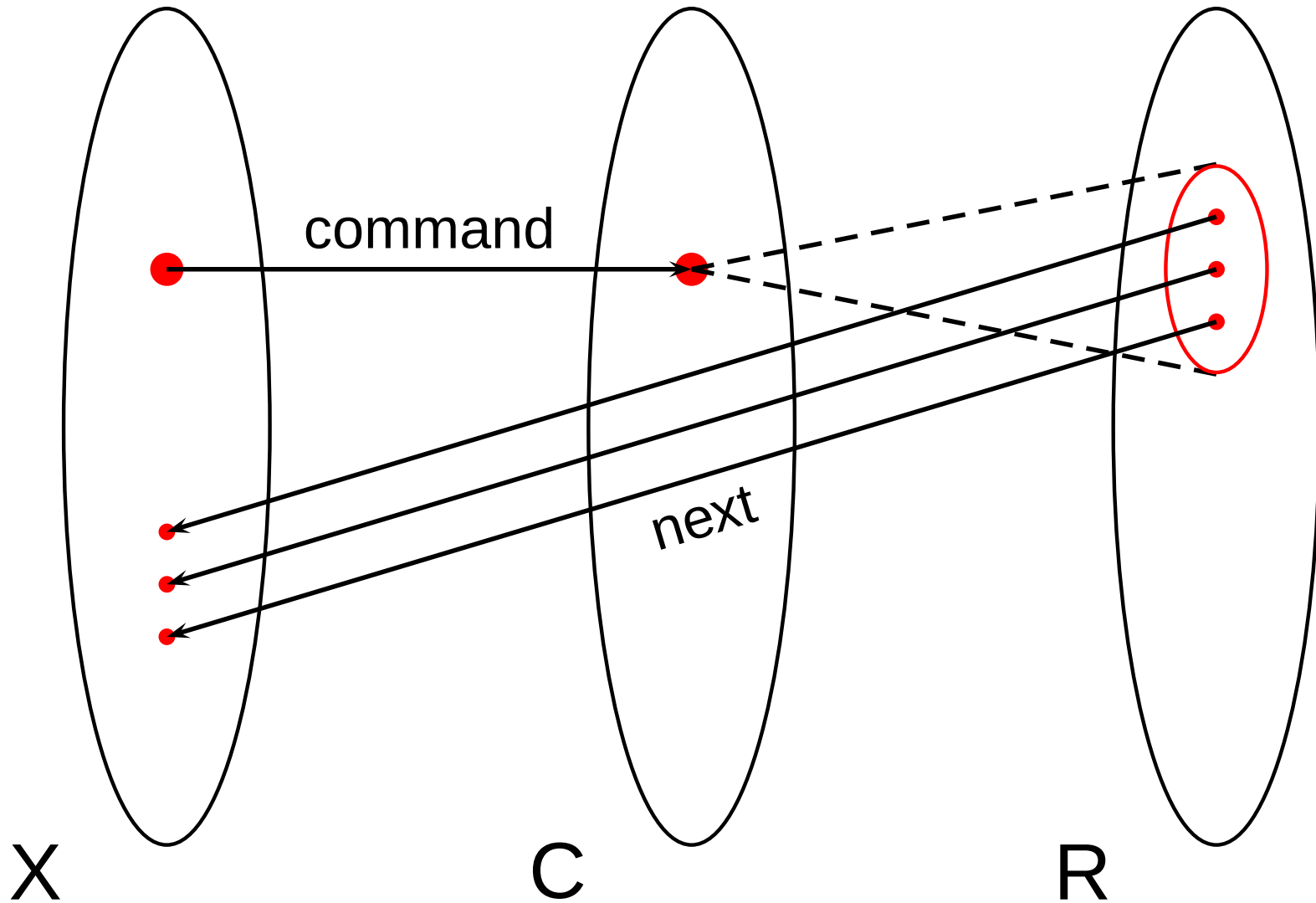
Motivation



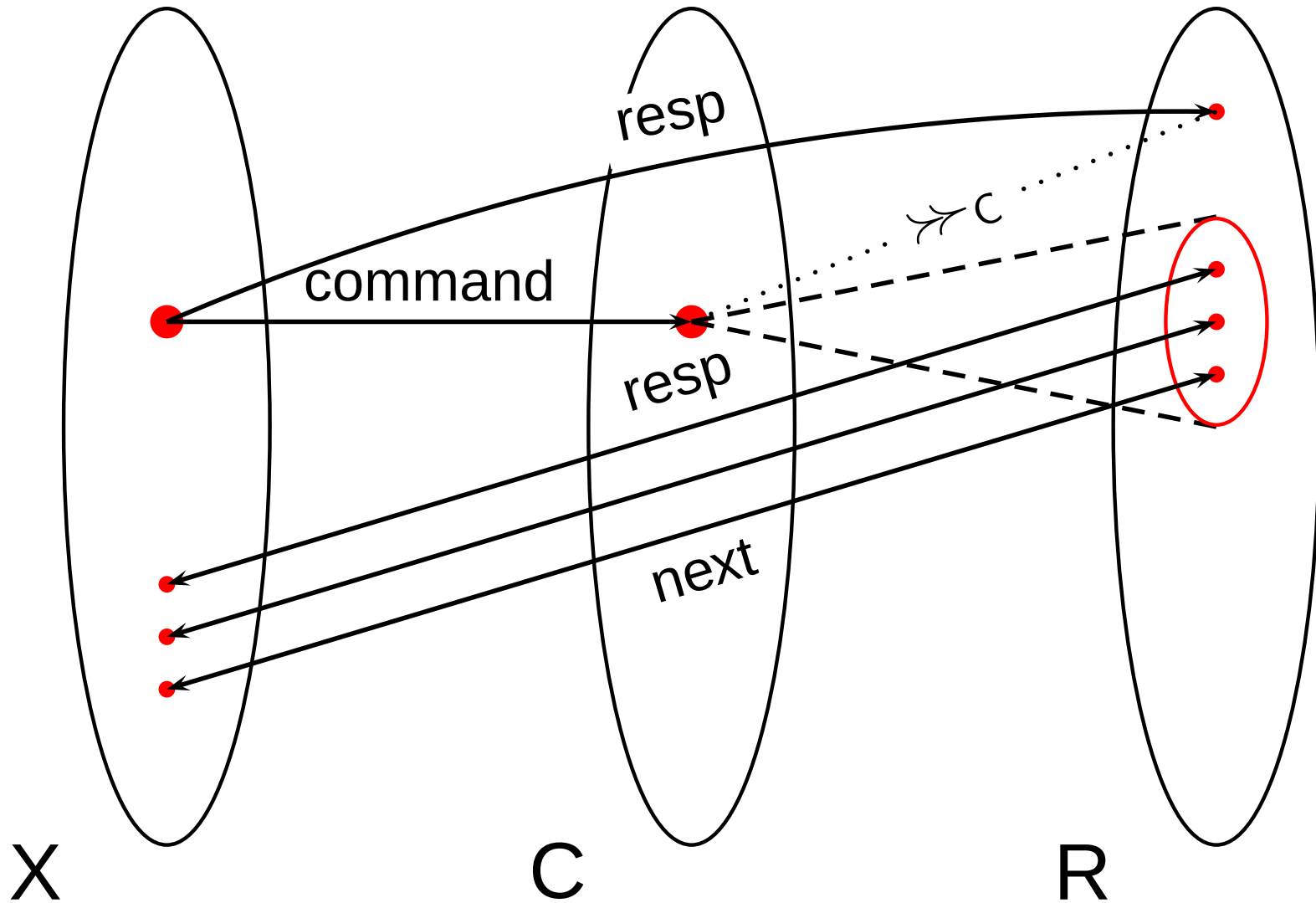
Interface/Game

- $C, R : \text{Set}$
a set of commands (players moves) and a set of responses (opponents moves),
- $\Leftarrow_C: R \rightarrow C \rightarrow \text{Set}$
a relation between R and C ,
- $\Leftarrow_R: C \rightarrow R \rightarrow \text{Set}$
a relation between C and R .

Programs/Strategies



Soundness



Programs/Strategies

A program/strategy is given by

- X : Set
a set X
- a function **onestep**
giving for every $x : X$ a command $c : C$ and a function
 $f : (r : R) \rightarrow c \ll_R r \rightarrow X$.

We write **command** x for c and **next** x for f .

A **sound** strategy is a strategy $(X, \text{onestep})$ and a function $\text{resp} : X \rightarrow R$ such that for $x : X$:

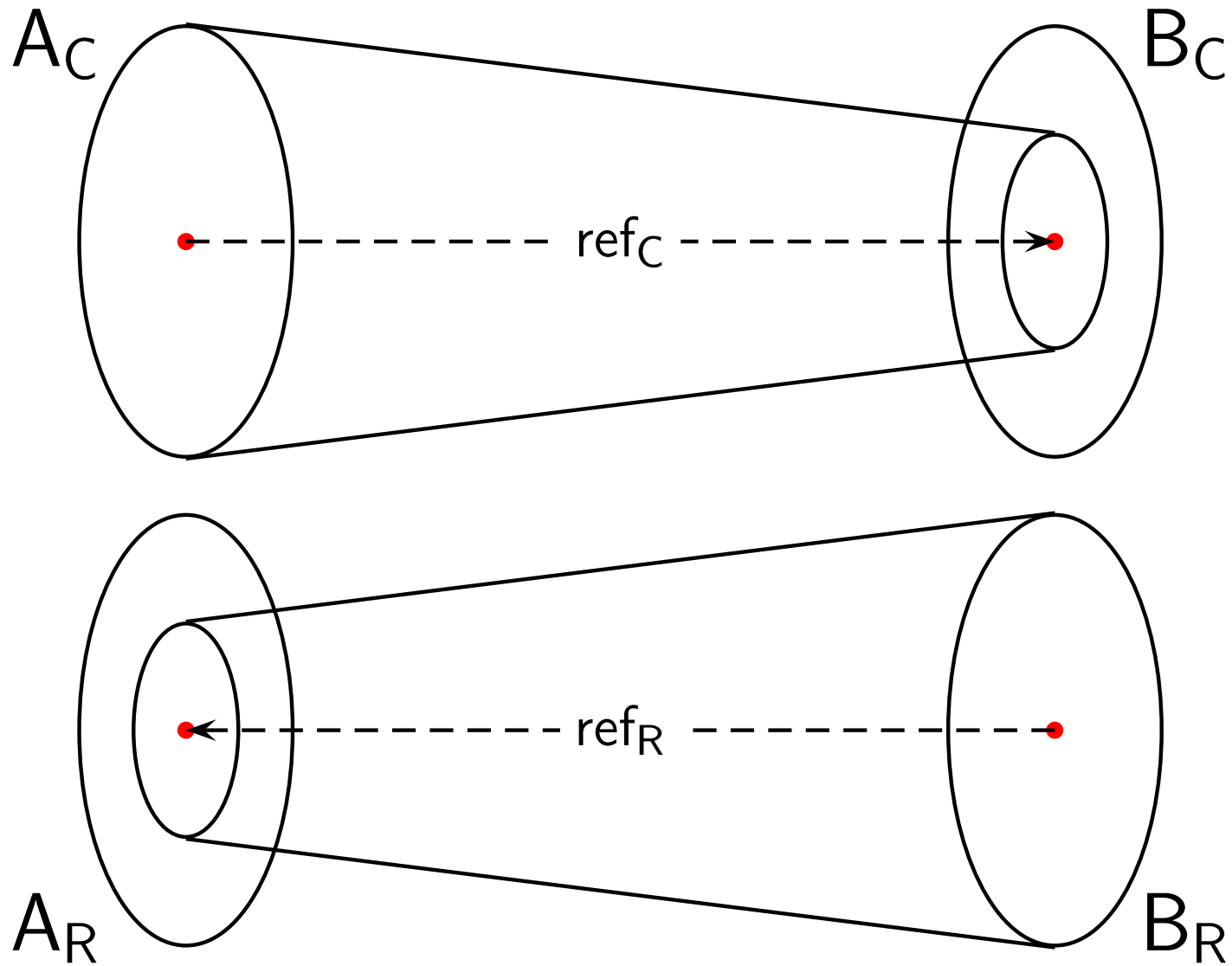
- $\text{resp } x \ll_C \text{command } x$
- $\text{next } x \ r \ \text{fitr} = r \quad \text{for} \quad \text{fitr} : \text{command } x \ll_R r$.

Refinement

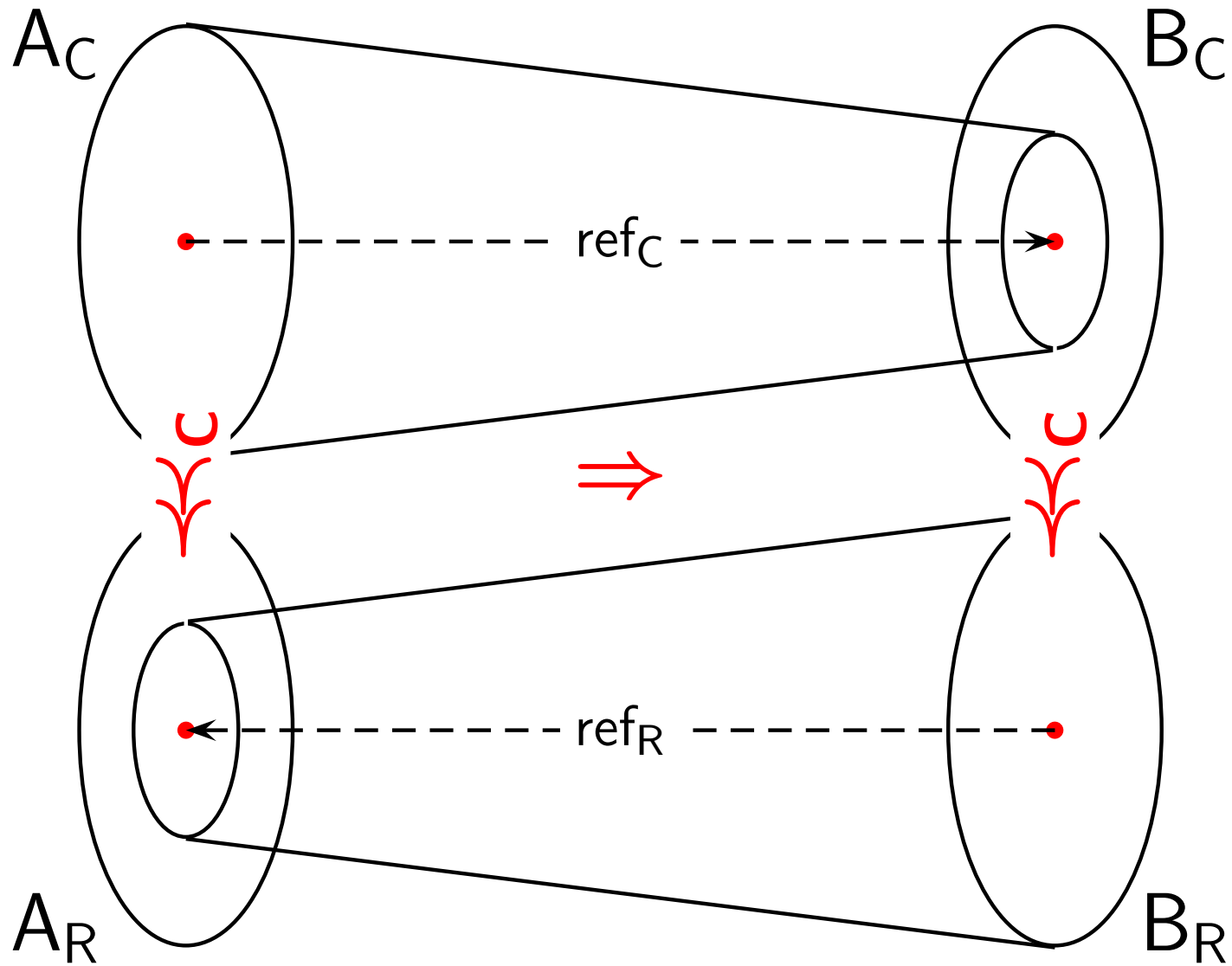
Idea: One side can do more,
the other side can do less

Let opponent have more freedom ...

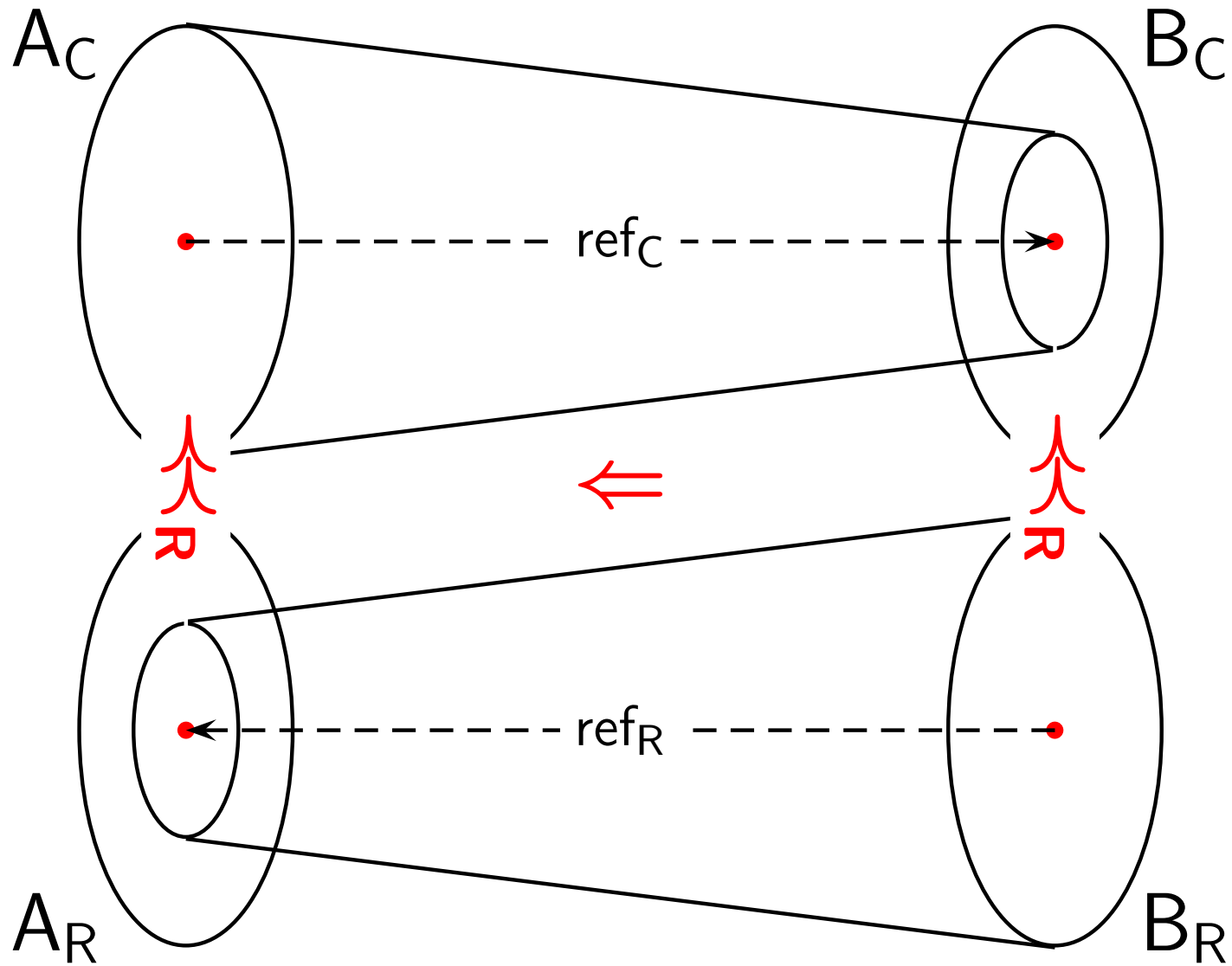
Refinement



Refinement



Refinement



Refinement

Given two interfaces $A = (C_A, R_A, \preccurlyeq_C^A, \preccurlyeq_R^A)$ and $B = (C_B, R_B, \preccurlyeq_C^B, \preccurlyeq_R^B)$.
A is a **refinement** of B is the type which elements consist of:

• a function

$$\text{ref}_C : C_A \rightarrow C_B,$$

• a function

$$\text{ref}_R : R_B \rightarrow R_A,$$

• and proofs for

$$\text{ref}_R r_b \preccurlyeq_C^A c_a \Rightarrow r_b \preccurlyeq_C^B \text{ref}_C c_a$$

and

$$\text{ref}_C c_a \preccurlyeq_R^B r_b \Rightarrow c_a \preccurlyeq_R^A \text{ref}_R r_b$$

for $r_b : R_B, c_a : C_A$.

Refinement

If A is a **Refinement** of B and $(X_A, \text{command}_A, \text{next}_A, \text{resp}_A)$ a sound strategy on A then is $(X_B, \text{command}_B, \text{next}_B, \text{resp}_B)$ a sound strategy on B where

- elements of X_B are triples

$$x : X, r_b : R_B, \text{id} : \text{resp}_A x = \text{ref}_R r_b$$



$$\text{resp}_B x r_b \text{id} = r_b$$



$$\text{command}_B x r_b \text{id} = \text{ref}_C(\text{command}_A x)$$



$$\text{next}_B x r_b \text{id } r'_b \text{ rfit}_B = \text{next}_B x (\text{ref}_R r'_b) \text{ rfit}_A$$

and rfit_A obtained from rfit_B by the refinement.

Refinement

- Refinement relation on interfaces/games is reflexiv and transitiv.
- Infimum A of $A_i, i : I$ is given by

$$A_R = (i : I) \rightarrow A_{iR}$$

$$A_C = \sum (I, \lambda i : I. A_{iC})$$

$$f \ll_C^A (i, c) :\Leftrightarrow f i \ll_C^{A_i} c$$

$$(i, c) \ll_R^A f :\Leftrightarrow c \ll_R^{A_i} f i$$

- Supremum A of $A_i, i : I$ is given by

$$A_R = \sum (I, \lambda i : I. A_{iR})$$

$$A_C = (i : I) \rightarrow A_{iC}$$

$$(i, r) \ll_C^A f :\Leftrightarrow r \ll_C^{A_i} f i$$

$$f \ll_R^A (i, r) :\Leftrightarrow f i \ll_R^{A_i} r$$

Cogame, Tensor

Cogame: Player becomes opponent,
opponent becomes player

Tensor: As in usual game semantics:
opponent chooses where to play

Cogame, Tensor, Lolipop

- The cogame $A^\perp$ of a game A is given by

$$C_A^\perp = R_A, R_A^\perp = C_A, \ll_C^{A^\perp} = \ll_R^A, \ll_R^{A^\perp} = \ll_C^A.$$

- The tensor $A \otimes B$ is given by

$$(A \otimes B)_C = A_C \times B_C$$

$$(A \otimes B)_R = (A_R \times B_C) + (A_C \times B_R)$$

$$\text{inl}(r_a, c_b) \ll_C^{A \otimes B} (c_a, c'_b) :\Leftrightarrow r_a \ll_C^A c_a \quad \& \quad c_b = c'_b$$

$$\text{inr}(c_a, r_b) \ll_C^{A \otimes B} (c'_a, c_b) :\Leftrightarrow r_b \ll_C^B c_b \quad \& \quad c_a = c'_a$$

$$(c_a, c'_b) \ll_R^{A \otimes B} \text{inl}(r_a, c_b) :\Leftrightarrow c_a \ll_R^A r_a \quad \& \quad c_b = c'_b$$

$$(c'_a, c_b) \ll_R^{A \otimes B} \text{inr}(c_a, r_b) :\Leftrightarrow c_b \ll_R^B r_b \quad \& \quad c_a = c'_a$$

- Lolipop $A \multimap B := (A \otimes B^\perp)^\perp$.

A Copy-Cat Strategy

Strategy on

$$A \multimap A$$

Idea: Take command c , response r
and proof for $c \multimap r$ or $r \multimap c$.

Play the diagonal (r, r) or (c, c) .

A Copy-Cat Strategy

(CopyCat_A, command_A, next_A, resp_A) given by

$$\begin{aligned} \text{CopyCat}_A = & ((r_a : A_R) \times (c_a : A_C) \times (\text{fitc} : r_a \Leftarrow_C^A c_a)) + \\ & ((c_a : A_C) \times (r_a : A_R) \times (\text{fitr} : c_a \Leftarrow_R^A r_a)) \end{aligned}$$

$$\text{resp}_A \text{ inl}(r_a \ c_a \ \text{fitc}) := (r_a, c_a)$$

$$\text{resp}_A \text{ inr}(r_a \ c_a \ \text{fitr}) := (r_a, c_a)$$

$$\text{command}_A \text{ inl}(r_a \ c_a \ \text{fitc}) := \text{inr}(c_a, c_a)$$

$$\text{command}_A \text{ inr}(r_a \ c_a \ \text{fitr}) := \text{inl}(r_a, r_a)$$

$$\text{next}_A \text{ inl}(r_a \ c_a \ \text{fitc}) (r'_a, c_a) \ \text{fitr} := \text{inl}(r'_a, c_a, \text{fitr})$$

$$\text{next}_A \text{ inr}(r_a \ c_a \ \text{fitr}) (r_a, c'_a) \ \text{fitc} := \text{inr}(r_a, c'_a, \text{fitc})$$

is a sound strategy on $A \multimap A$.

Composition

$$\sigma : A \multimap B$$

$$\tau : B \multimap C$$

$$\sigma; \tau : A \multimap C$$

Idea: Play strategy σ against strategy τ in B.

Problem: Infinite play in B.

Usual solution: Winning strategies.

Solution here: Fairness!

Some restrictions on games

For $A : \text{Set}$ let

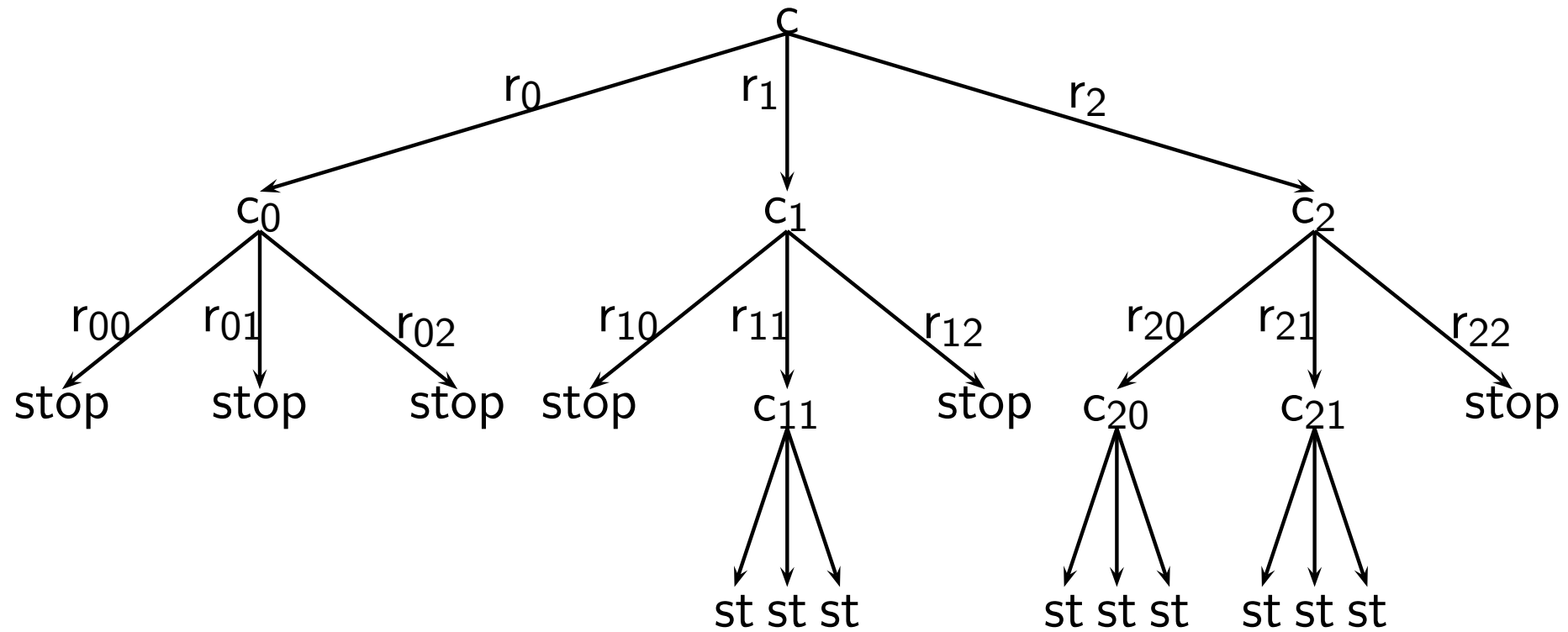
$$\text{Collapse } A := (a, b : A) \rightarrow \text{Id } A \ a \ b$$

$$\text{UIP } A := (a, b : A) \rightarrow \text{Collapse } (\text{Id } A \ a \ b)$$

From now on for all games A

- $\text{Collapse } r \prec_C^A c$ and $\text{Collapse } c \prec_R^A r$
- $\text{UIP } A_C$ and $\text{UIP } A_R$.
- Everything must have a predecessor (reasonable game).

Beginning



[illegible]

Fair Strategy

For games A, B a **fair strategy** on $A \multimap B$ is given by

- a strategy $\sigma = (X_\sigma, \text{command}_\sigma, \text{next}_\sigma)$



$$\text{maxABeg}_\sigma, \text{maxBBeg}_\sigma : X \rightarrow (\text{Beginning } A \multimap B)$$



with

$$\text{maxvalidABeginning } A \ B \ \sigma \ (\text{maxABeg}_\sigma \ x) \ x$$

and

$$\text{maxvalidBBeginning } A \ B \ \sigma \ (\text{maxBBeg}_\sigma \ x) \ x$$

Link

For games A, B, C , $c_{ab} : (A \multimap B)_C$, $c_{bc} : (B \multimap C)_C$

Link $A \ B \ C \ c_{ab} \ c_{bc}$

is given by

- Link $A \ B \ C \text{ inl}(r_a, r_b) \text{ inl}(r'_b, r_c) = \text{Id } B_R \ r_b \ r'_b$
- Link $A \ B \ C \text{ inl}(r_a, r_b) \text{ inr}(c_b, c_c) = \text{False}$
- Link $A \ B \ C \text{ inr}(c_a, c_b) \text{ inl}(r_b, r_c) = (c_b \multimap_R^B r_b) + (r_b \multimap_C^B c_b)$
- Link $A \ B \ C \text{ inr}(c_a, c_b) \text{ inr}(c'_b, c_c) = \text{Id } B_C \ c_b \ c'_b$

Composition

For fair strategies $\sigma = (X_\sigma, \text{command}_\sigma, \text{next}_\sigma) : A \multimap B$ and $\tau = (X_\tau, \text{command}_\tau, \text{next}_\tau) : B \multimap C$ let $X_{\sigma;\tau}$ be the set which elements are triples

• $x : X_\sigma$

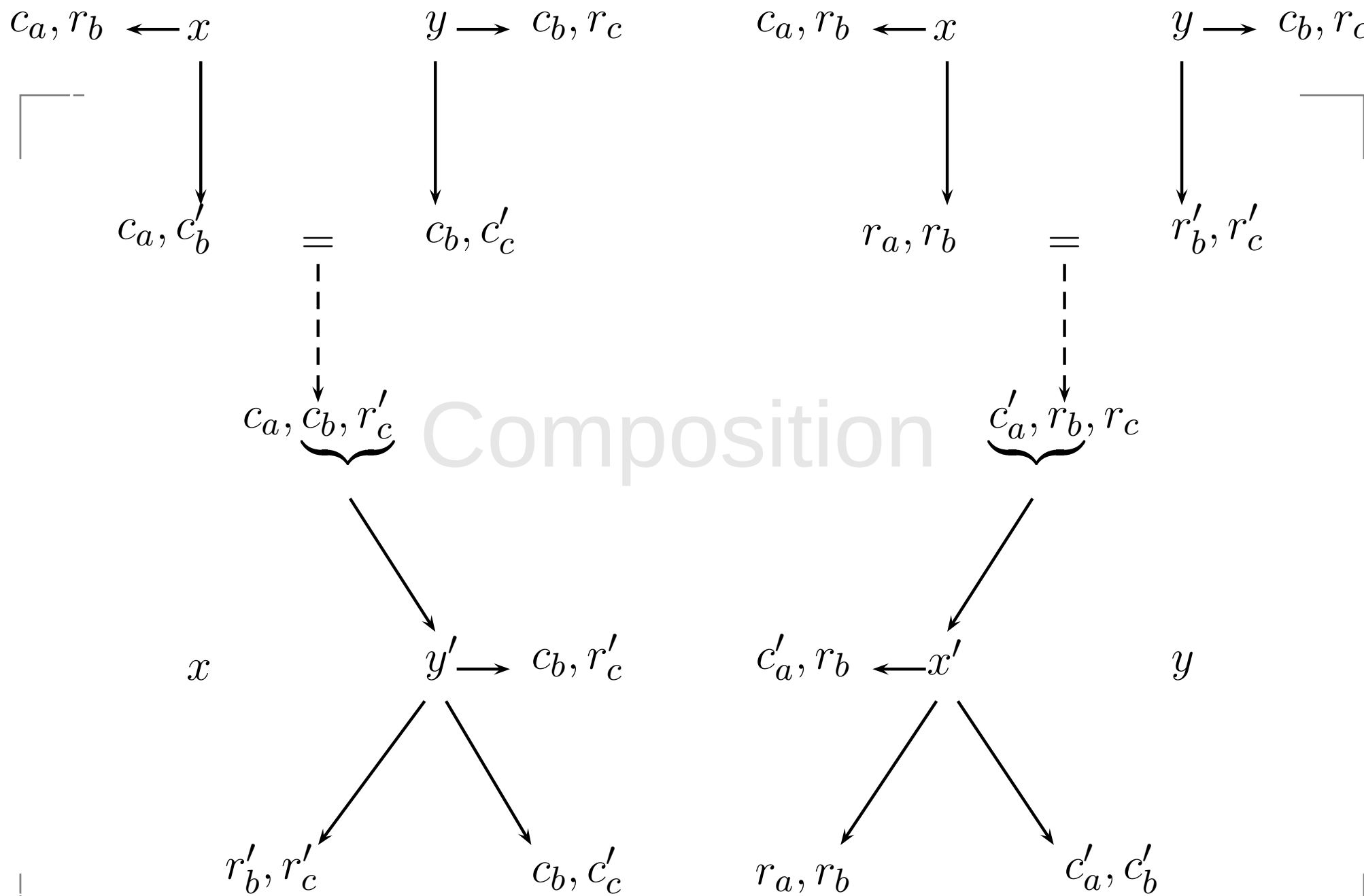
• $y : X_\tau$

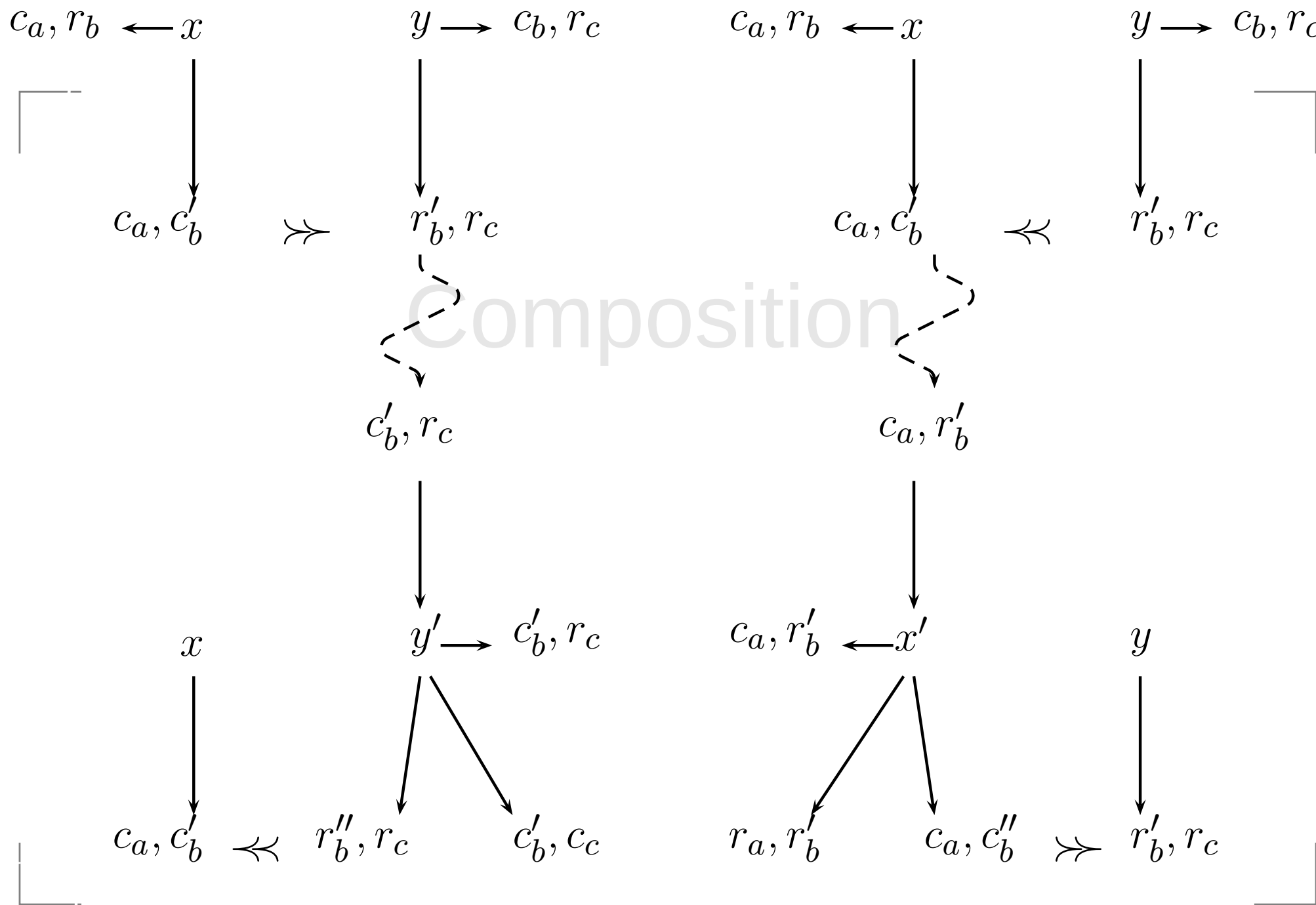
• $\text{link} : \text{Link } A \ B \ C \ (\text{command}_\sigma \ x) \ (\text{command}_\tau \ y)$

and

$$\text{resp}_{\sigma;\tau} (x, y, \text{link}) = (c_a, r_c)$$

where $\text{resp}_\sigma x = (c_a, r_b)$ and $\text{resp}_\tau y = (c_b, r_c)$.





Bisimulation

Let $\sigma = (X_\sigma, \text{command}_\sigma, \text{next}_\sigma)$ and $\tau = (X_\tau, \text{command}_\tau, \text{next}_\tau)$ strategies on A . $x : X_\sigma$, $y : X_\tau$, $n : \text{Nat}$ are n -bisimilar
($x \sim_n y$) iff $n = 0$ or

$$\text{command}_\sigma x = \text{command}_\tau y$$

and for $r :: A_R$ and $\text{rfit} : \text{command}_\sigma x \ll_R r$

$$\text{next}_\sigma x r \text{ rfit} \sim_{n-1} \text{next}_\tau y r \text{ rfit}'$$

where rfit' is obtained by transferring rfit .

$x : X_\sigma$, $y : X_\tau$ are bisimilar iff they are n -bisimilar f.a.
 $n : \text{Nat}$.

Behavioural equivalence

Strategies $\sigma = (X_\sigma, \text{command}_\sigma, \text{next}_\sigma)$ and $\tau = (X_\tau, \text{command}_\tau, \text{next}_\tau)$ on A are behavioural equivalent ($x \approx y$) iff there are $f : X_\sigma \rightarrow X_\tau$ and $g : X_\tau \rightarrow X_\sigma$ such that

$$f \ x \sim x$$

and

$$g \ y \sim y$$

f.a. $x : X_\sigma, y : X_\tau$.

The Category of reasonable games with fair strategies

For a fair and sound strategy $\sigma : A \multimap B$ is

$$\text{id}_A; \sigma \approx \sigma \approx \sigma; \text{id}_B$$

where $\text{id}_A = \text{copyCat}_A$, $\text{id}_B = \text{copyCat}_B$ and A, B reasonable.

For fair and sound strategies

$\sigma : A \multimap B, \tau : B \multimap C, \mu : C \multimap D$:

$$(\sigma; \tau); \mu \approx \sigma; (\tau; \mu)$$

where A, B, C, D reasonable.

Conclusion and future work

- Linear category.
- Investigate relationship to work of Hancock/Hyvernat.
- Improve notion of refinement. Investigate relation to refinement calculus.
- Formal topology.
- ...

Maxvalid Beginnings

For A game let

Beginning A $\quad :=$ data $\quad$ stop |

fork($c : A_C$)($f : (r : A_R) \rightarrow c \prec_R r \rightarrow$ Beginning A)

For games A, B, $\sigma = (X, \text{command}, \text{next})$ strategy on $A \multimap B$, $x : X$ and
 $beg : \text{Beginning A} \multimap B$ let

maxvalidABeginning A B σ stop x $\quad =$ “sigma plays in B”

maxvalidABeginning A B σ (fork c f) x $\quad =$ “sigma plays in A and
 command $x = c$ and

maxvalidABeginning A B σ (f r rfit) (next x r rfit)
 for $r : (A \multimap B)_R$ and rfit : $c \prec_R^{A \multimap B} r$.”

maxvalidBBeginning A B σ beg x analog.

Reasonable game

A is **reasonable** is the type which elements consists of

- a function $\text{reas}_C : A_C \rightarrow A_R$
- a function $\text{reas}_R : A_R \rightarrow A_C$

such that

$$\text{reas}_C c \preccurlyeq_C c \text{ for } c : A_C$$

$$\text{reas}_R r \preccurlyeq_R r \text{ for } r : A_R$$

"Everything is needed"

Authors: Sabrina Tarento, Gilles Barthe

Title: A machine-checker formalization of the generic model and the random oracle model

Abstract:

Most approaches to the formal analyses of cryptographic protocols make the perfect cryptographic assumption, i.e the hypothesis that there is no way to obtain knowledge about the plaintext pertaining to a ciphertext without knowing the key. Ideally, one would prefer to rely on a weaker hypothesis on the computational cost of gaining information about the plaintext pertaining to a ciphertext without knowing the key. Such a view is permitted by the generic model and the random oracle model which provide non-standard computational models in which one may reason about the computational cost of breaking a cryptographic scheme. Using the proof assistant Coq, we provide a machine-checker account of the Generic Model and the Random Oracle Model.

A Machine-Checked Formalization of the Generic Model and the Random Oracle Model

Sabrina Tarento

joint work with Gilles Barthe and Jan Cederquist

INRIA Sophia Antipolis

Overview

- Perfect Cryptography assumption
- ElGamal
- Generic Model
- Formalization of generic algorithm
- Random Oracle Model
- Formalization of interactive generic algorithm
- Parallel attacks

Perfect cryptography assumption

there is no way to obtain knowledge about the plaintext pertaining to a ciphertext without knowing the key.

Assumption taken e.g in:

- belief logics (M.Burrows, M.Abadi, and R.Needham)
- model checkers (G.Lowe)
- proof assistants (L.C. Paulson)

Generic Model

- introduced by Shoup in 1997 and extended by Schnorr and Jakobsson
- focus on attacks that work for all groups
- attackers make group operations and tests of collisions in order to find information about secrets
- ideal model with some difficulties (the same as ROM), but useful to prove security and well adapted to Coq
- used for ECDSA ...

Running Example: ElGamal

G cyclic group of prime order q with generator g

A chooses randomly $x \in \{0, \dots, q-1\}$

G , g et g^x are public data

$$A \longrightarrow B : g^x$$

B wants to send the message m to A , so B chooses randomly

$r \in \{0, \dots, q-1\}$

$$B \longrightarrow A : (g^r, m \cdot (g^x)^r)$$

decryption: $(g^r)^{-x} \cdot m \cdot (g^x)^r = m$

What is a random value?

- secrets are random in $\mathbb{Z}_q$
- rather than formalizing random elements, we introduce a type Sec of secrets, use an interpretation function $f : Sec \rightarrow \mathbb{Z}_q$ and treat input and output are polynomials in $\mathbb{Z}_q[Sec]$
- the probability space is: $Sec \rightarrow \mathbb{Z}_q$

Generic algorithm

a *generic algorithm* performs t generic steps

- $f_1, \dots, f_{t'} \in G$ (inputs) $1 \leq t' < t$,
- $f_i = \prod_{j=1}^{i-1} f_j^{a_j}$ for $i = t' + 1, \dots, t$ where $(a_1, \dots, a_{i-1}) \in \mathbb{Z}_q^{i-1}$

modeled as a list:

$$\overline{\text{empty\_run} \in \text{Run}}$$

$$\frac{R \in \text{Run} \quad e \in (\text{list } \mathbb{Z}_q)}{\text{step}(R, e) \in \text{Run}}$$

Collisions

$$CO_t := \{(j, k) | f_j = f_k, 1 \leq j < k \leq t\}$$

- only non trivial collision reveals information about the secrets

example of ElGamal: $g, g^x, g^r, m_b \cdot (g^x)^r$

$$\log_g f_i = a_{i,1} + a_{i,2}x + a_{i,3}r + a_{i,4}(\log_g m_b + rx)$$

- finding informations about the secrets amounts solving

$$\log_g f_i - \log_g f_j = c_1 + c_2x + c_3r + c_4(\log_g m_b + rx) = 0$$

Generic algorithm

$GA = \{Sec : Set; run : Run; inp : (listT \mathbb{Z}_q[Sec]); condition\}$

- Output: the list of polynomials in $\mathbb{Z}_q[Sec]$ resulting from multivariate exponentiations
- Concrete output: a list of elements in $\mathbb{Z}_q$ using the interpretation function $f : Sec \rightarrow \mathbb{Z}_q$ to the output
- Collisions: tests of non trivial equalities between the concrete outputs
- Condition: eliminate trivial equalities

Results

Lemma 1 (Schwartz Lemma) *Let $p(x_1, \dots, x_k)$ be a polynomial in k variables, not identical to 0, with degree at most d , and the values chosen uniformly and independently in $[0, q - 1]$. Then $\Pr(p(x_1, \dots, x_k) = 0) \leq d/q$.*

Lemma 2 $\Pr(CO_t \neq \emptyset) \leq \theta(\frac{d}{q} \cdot t^2)$.

Proof: use of Schwartz lemma.

Lemma 3 $\Pr(\text{SecFound } x) \leq \Pr(CO_t \neq \emptyset) + \Pr(\text{guess } x)$

Application to ElGamal

- Generic DL-complexity lower bound:

inputs: g, g^x

$$Pr(CO_t \neq \emptyset) \leq \theta\left(\frac{t^2}{q}\right)$$

$$Pr(\text{SecFound } x) \leq Pr(CO_t \neq \emptyset) + \theta\left(\frac{1}{q}\right)$$

- Indistinguishability:

inputs: $g, g^x, g^r, m_b \cdot (g^x)^r, m_0, m_1$

$$Pr(CO_t \neq \emptyset) \leq \theta\left(\frac{t^2}{q}\right)$$

$$Pr(\text{SecFound } b) \leq Pr(CO_t \neq \emptyset) + \frac{1}{2}$$

Random Oracle Model

- group operations
- queries to the hash oracle H
- interactions with a decryption oracle

we find informations about the secrets by finding collisions or a valid signed ciphertexts using interactions

- There exist signature and encryption schemes which are secure in the Random Oracle Model, but for which **any implementation** of the random oracle results in insecure schemes.(Canetti)

How do we formalize ideal hash function ?

- for communication with oracles: type Val

$$Val \sqsubseteq Sec$$

- interpretation function $rom : Val \rightarrow \mathbb{Z}_q$
- to formalize an hash function $H : \mathbb{Z}_q[Val]^3 \rightarrow Val$, we define a type $HashQuery := \mathbb{Z}_q[Val]^3 \times Val$

$$h : HashQuery := (a, b, d, c) \Rightarrow c = H(a, b, d)$$

Interactive generic algorithm

$$\overline{eRun : Run}$$

$$\frac{R : Run \quad e : list \quad Val}{step(R, e) : Run}$$

$$\frac{R : Run \quad c : HashQuery}{hashstep(R, c) : Run}$$

$$\frac{R : Run \quad cip : \mathbb{Z}_q^4}{decstep(R, cip) : Run}$$

Results

We assume that we have t interactive steps including l interactions with the decryptor.

Lemma 4 $Pr(CO_t \neq \emptyset) \leq \theta(\frac{d}{q} \cdot (t - l)^2).$

Lemma 5 $Pr(Extr\ r) \leq \frac{(d+1) \cdot l}{q}.$

Lemma 6 $Pr(SecFound\ x) \leq Pr(CO_t \neq \emptyset) + Pr(Extr\ r) + Pr(guess\ x)$

Parallel attacks

- group operations
- queries to the hash oracle H
- interactions with a signature oracle (signer):

the signer picks a random $r \in \mathbb{Z}_q$, computes g^r and sends it to the user who selects $c \in \mathbb{Z}_q$ and sends it to the signer who computes $z := r + cx$ and responses z .

Future work

- Reason about parallel attacks
- Reason about anonymous broadcast
- ...

Authors: Veronika Ortner, Norbert Schirmer

Title: Verification of BDD-Algorithms in Isabelle/HOL

Abstract:

We present the verification of the normalisation of a binary decision diagram (BDD).

The normalisation follows the original algorithm presented by Bryant in 1986 and transforms an ordered BDD in a reduced, ordered and shared BDD. The verification is based on Hoare logics and is carried out in the theorem prover Isabelle/HOL. The work is both a case study for verification of procedures on a complex pointer structure, as well as interesting on its own, since it is the first proof of functional correctness of the normalisation process we are aware of.

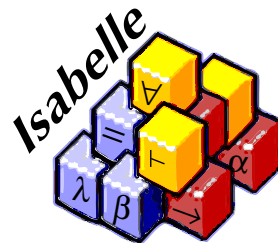
TYPES '04

Verification of BDD Algorithms

Veronika Ortner
Norbert Schirmer

Institut für Informatik, TU München

16th December 2004



Content

Content

1. Subject and Goals

Content

1. Subject and Goals
2. The Verification Environment

Content

1. Subject and Goals
2. The Verification Environment
3. Verified BDD Algorithms

Content

1. Subject and Goals
2. The Verification Environment
3. Verified BDD Algorithms
4. Results

Subject and Goals

- Goals:
 - Verification of pointer-based data structures
 - Formal proof of the normalization algorithm on BDDs

Subject and Goals

- Goals:
 - Verification of pointer-based data structures
 - Formal proof of the normalization algorithm on BDDs
- Approach:
 - Formalization of BDDs and their properties in Isabelle/Isar
 - Implementation of BDD algorithms in a C-like language
 - Specification and verification using our verification environment for Hoare Logic

Basics of the Verification Environment

- Definition of a **record** for program variables and the heap

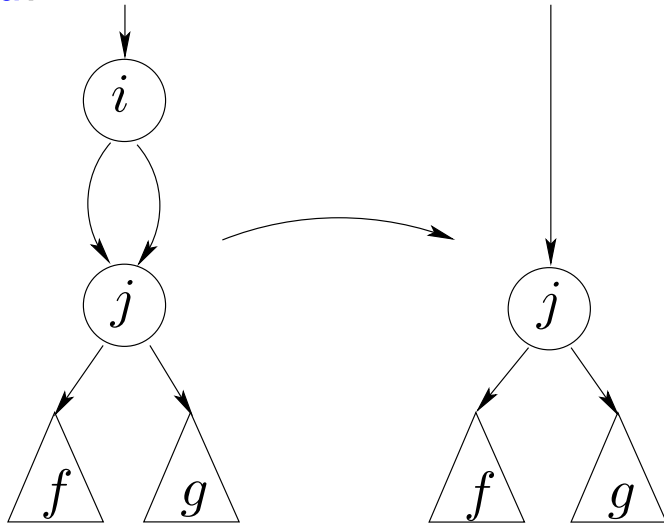
Basics of the Verification Environment

- Definition of a **record** for program variables and the heap
- Hoare-annotated program $\xrightarrow{vcg}$ proof obligation

This proof obligation is the basis for our verification work.

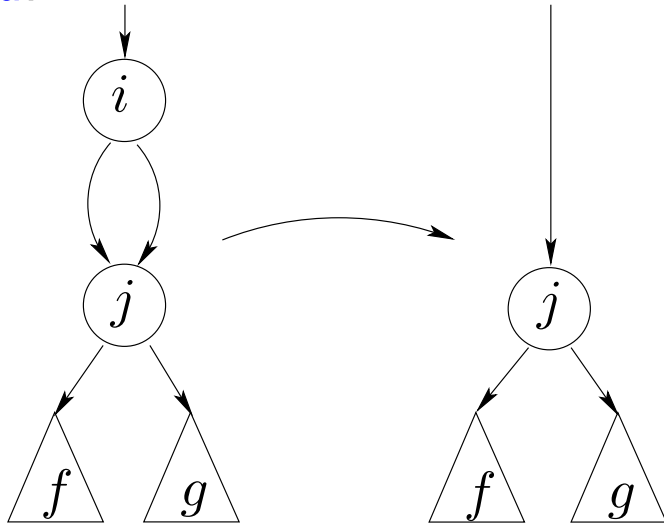
Properties of BDDs

reduced:

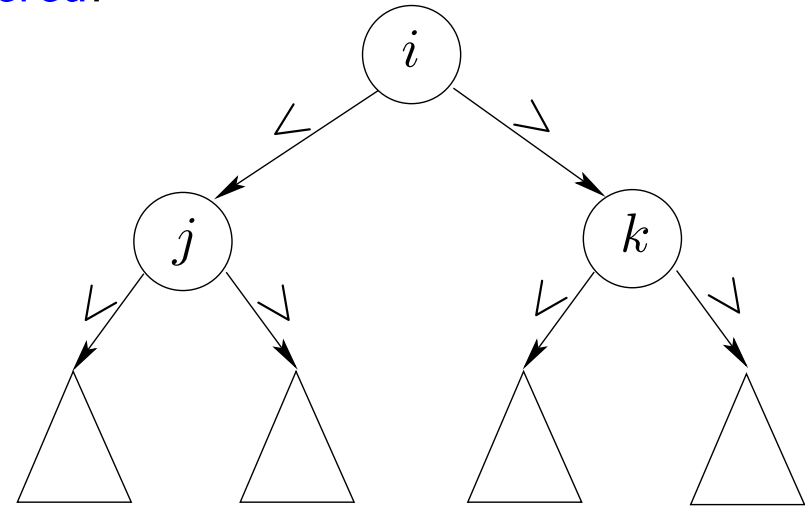


Properties of BDDs

reduced:

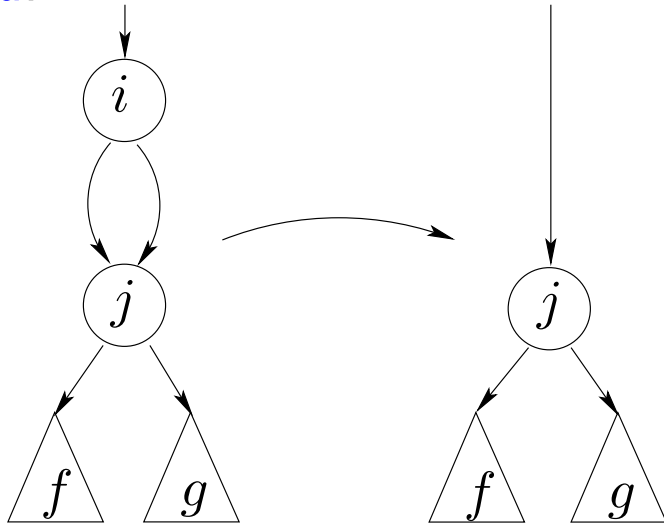


ordered:

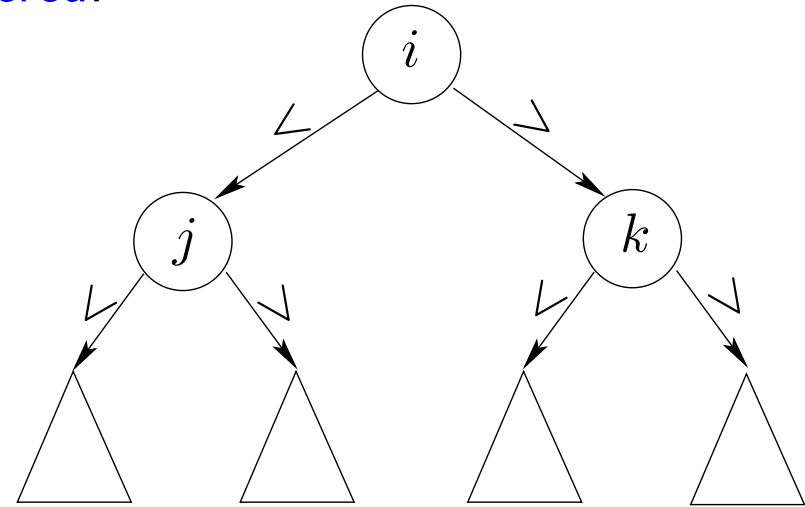


Properties of BDDs

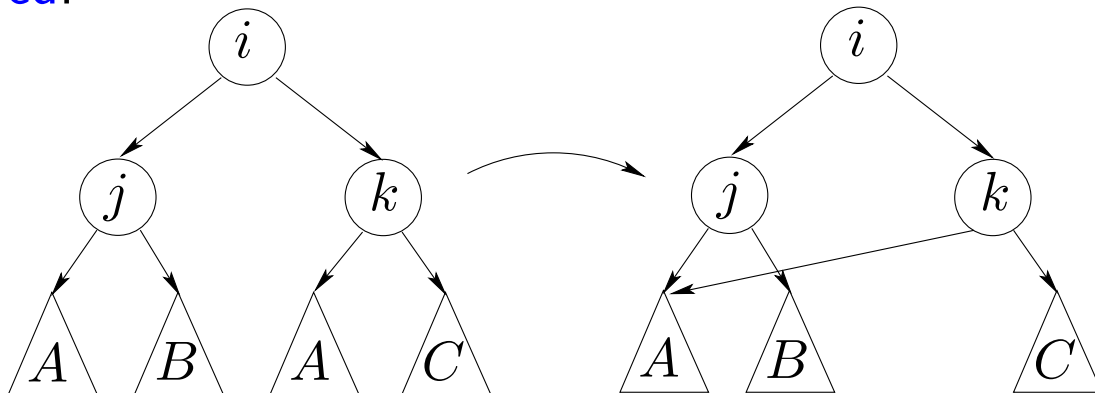
reduced:



ordered:

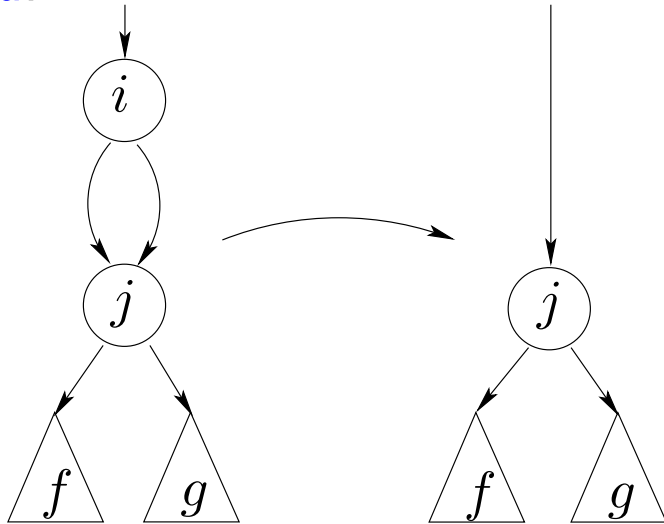


shared:

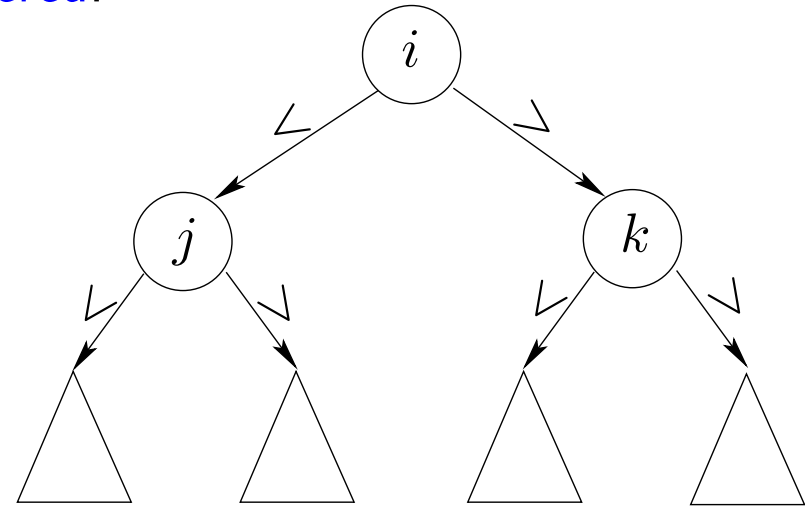


Properties of BDDs

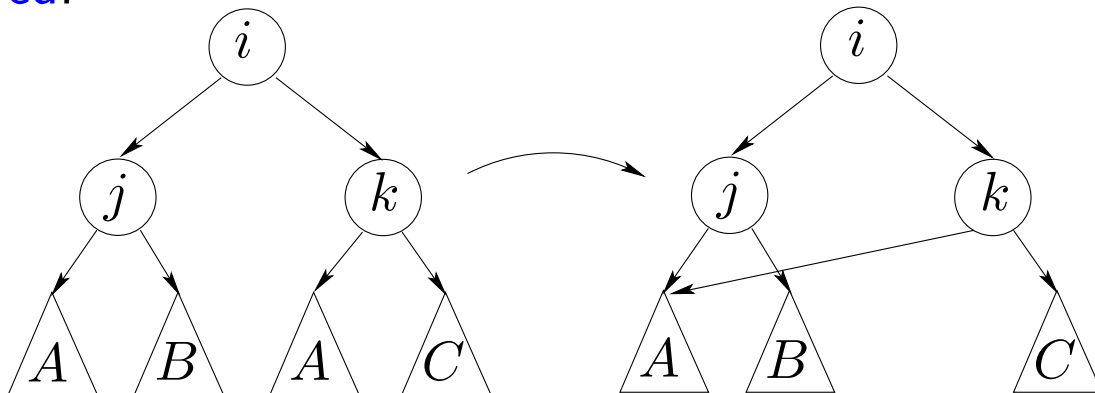
reduced:



ordered:



shared:

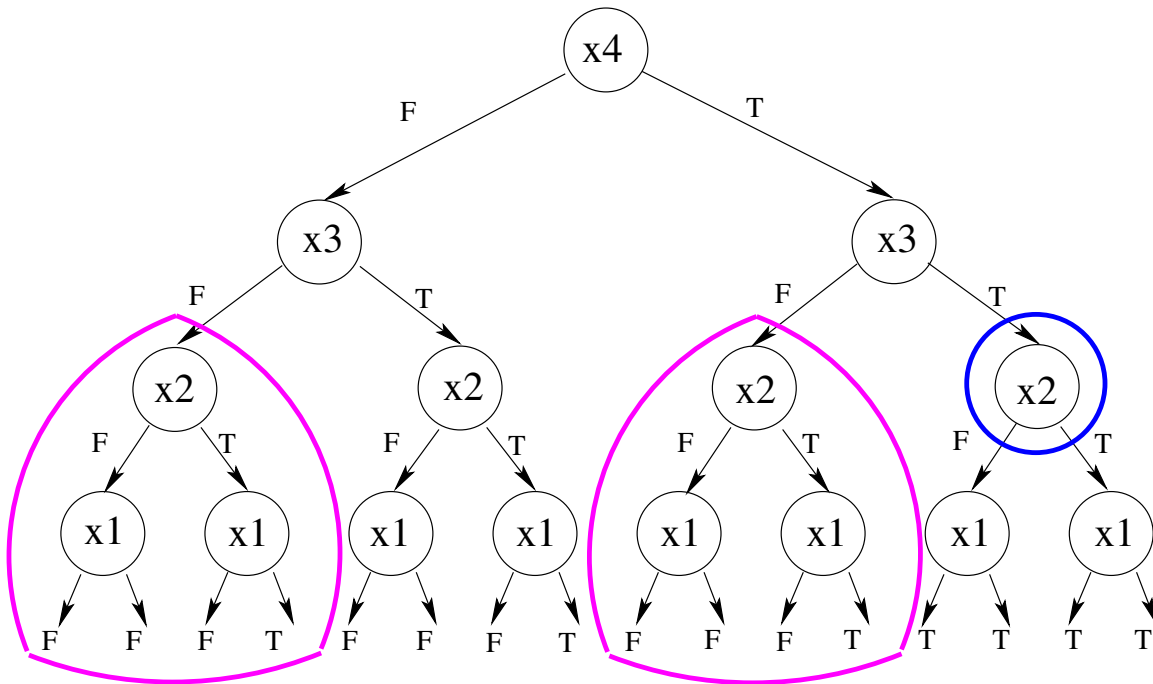


Result:

- Unique representation for each function
- No redundant parts

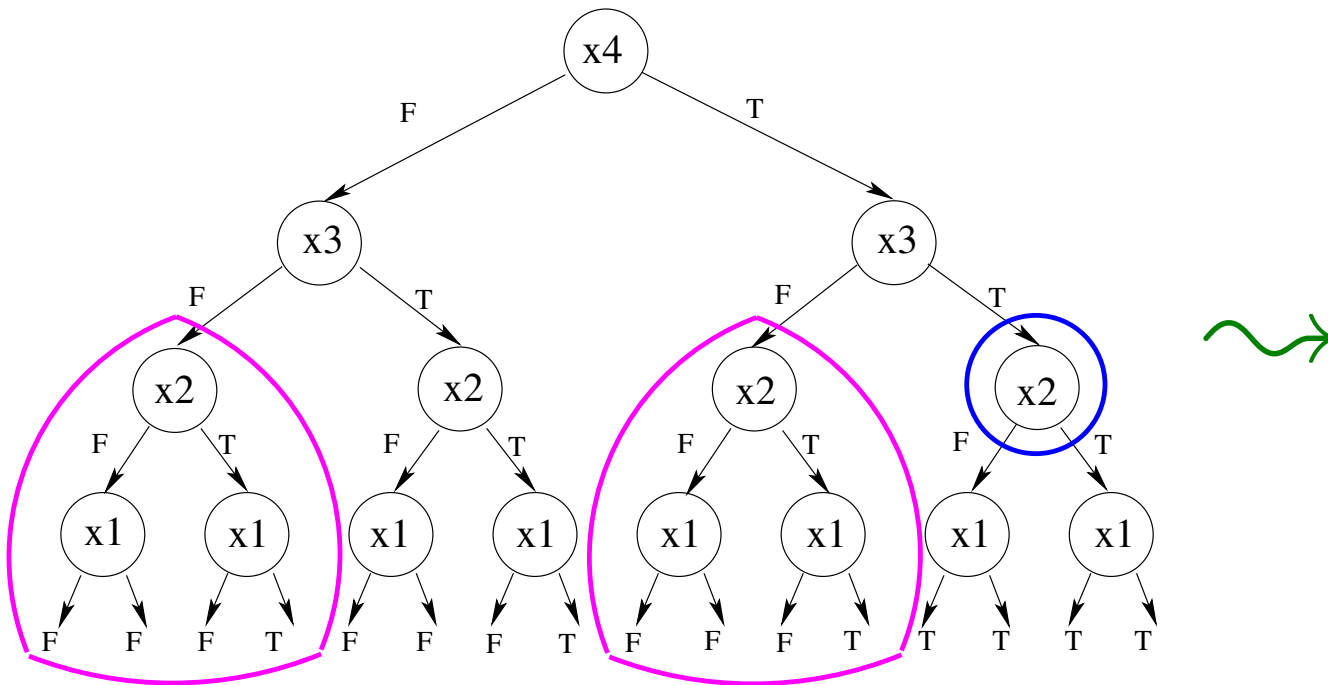
Example

$$f(x1, x2, x3, x4) := x1 \wedge x2 \vee x3 \wedge x4$$



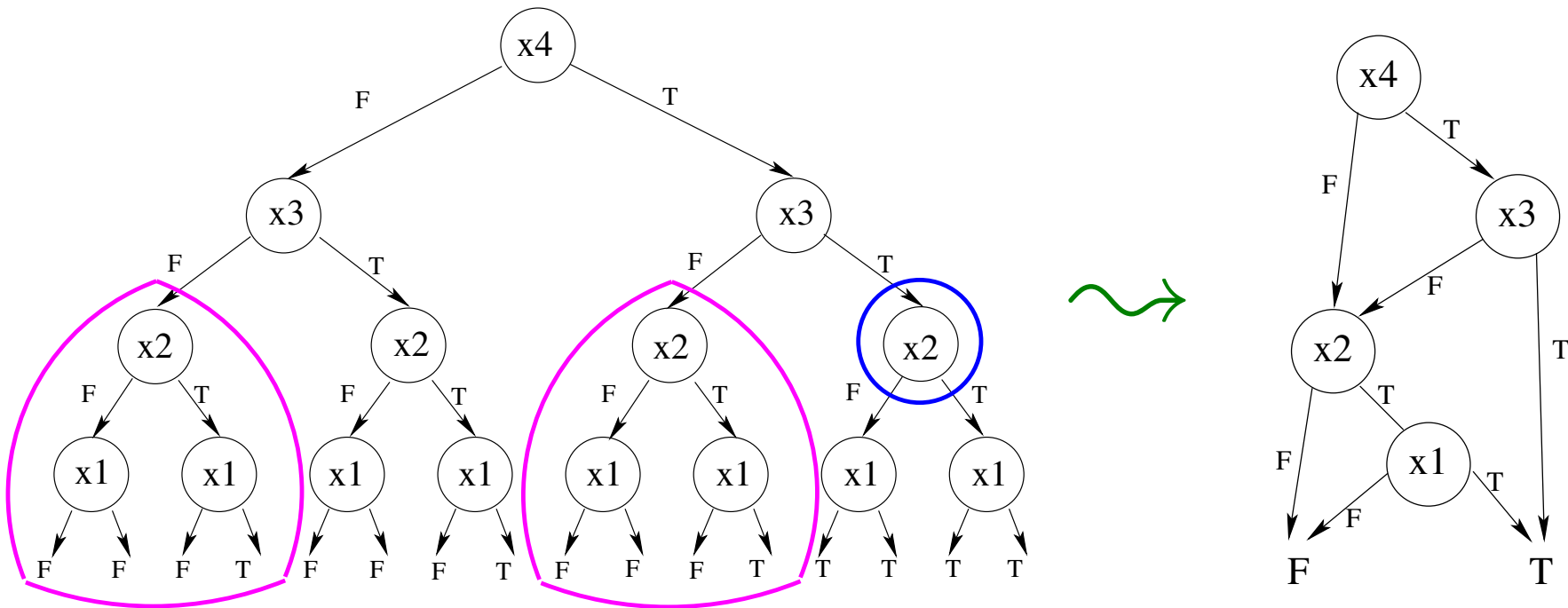
Example

$$f(x1, x2, x3, x4) := x1 \wedge x2 \vee x3 \wedge x4$$



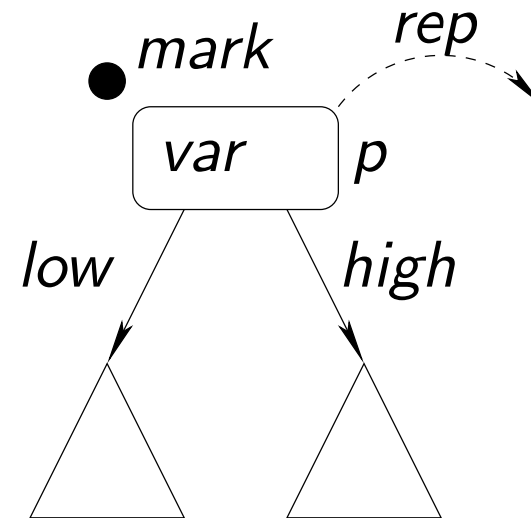
Example

$$f(x1, x2, x3, x4) := x1 \wedge x2 \vee x3 \wedge x4$$



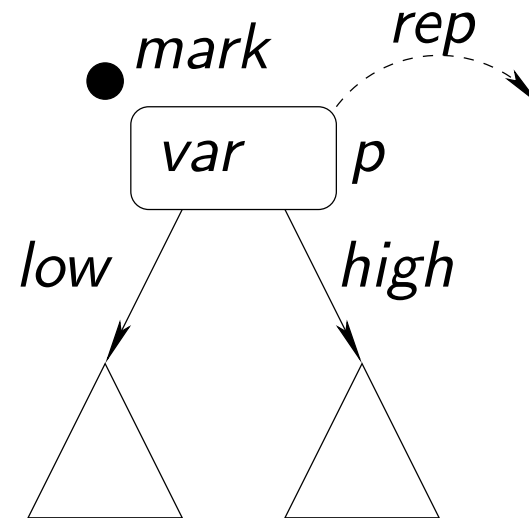
Representation of BDDs in the heap

```
struct {  
    nat var;  
    node* low;  
    node* high;  
    node* rep;  
    bool mark;  
} node;
```



Representation of BDDs in the heap

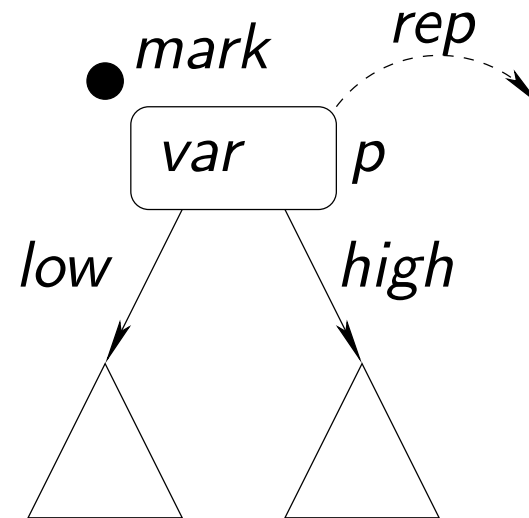
```
struct {  
    nat var;  
    node* low;  
    node* high;  
    node* rep;  
    bool mark;  
} node;
```



```
record bdd_data =  
  var :: ref  $\Rightarrow$  nat  
  low :: ref  $\Rightarrow$  ref  
  high :: ref  $\Rightarrow$  ref  
  ref :: ref  $\Rightarrow$  ref  
  mark :: ref  $\Rightarrow$  bool  
  :
```

Representation of BDDs in the heap

```
struct {  
    nat var;  
    node* low;  
    node* high;  
    node* rep;  
    bool mark;  
} node;
```



```
record bdd_data =  
  var :: ref  $\Rightarrow$  nat  
  low :: ref  $\Rightarrow$  ref  
  high :: ref  $\Rightarrow$  ref  
  ref :: ref  $\Rightarrow$  ref  
  mark :: ref  $\Rightarrow$  bool  
  :
```

bdd_data shows split heap approach

Advantage:

Specification of properties on relevant heap components instead of on whole heap

BDD Formalization - Illustration

2 levels of specification:

- Graph structure in order to represent *sharing*:

datatype *dag* = *Tip* | *Node dag ref dag*

- Decision tree (e.g. for evaluation purposes):

datatype *bdt* = *Zero* | *One* | *Bdt_Node bdt nat bdt*

BDD Formalization - Illustration

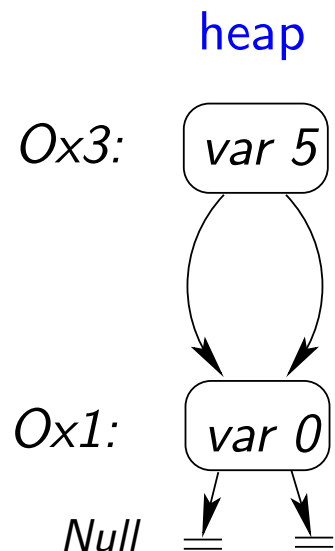
2 levels of specification:

- Graph structure in order to represent *sharing*:

datatype *dag* = *Tip* | *Node dag ref dag*

- Decision tree (e.g. for evaluation purposes):

datatype *bdt* = *Zero* | *One* | *Bdt_Node bdt nat bdt*



BDD Formalization - Illustration

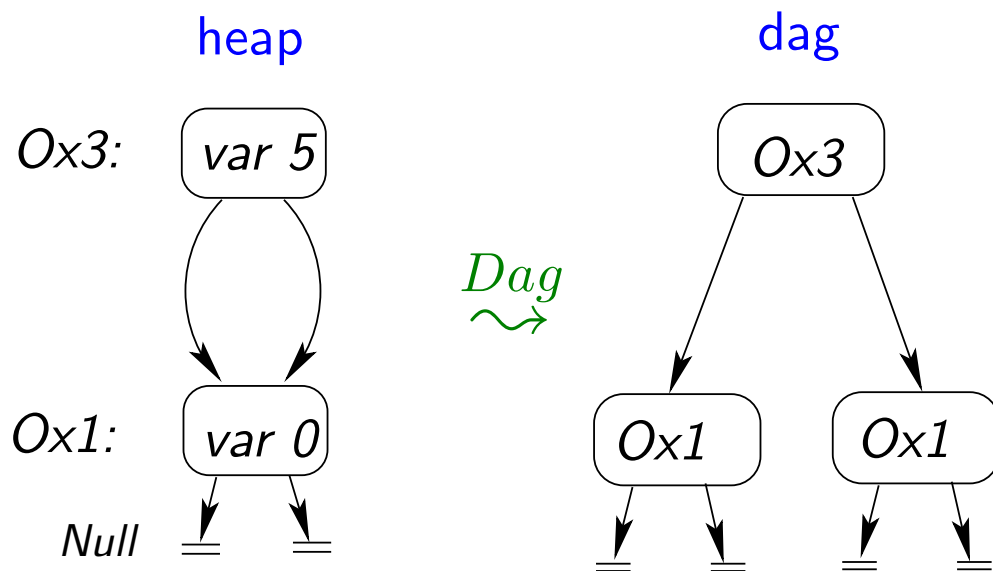
2 levels of specification:

- Graph structure in order to represent *sharing*:

datatype *dag* = *Tip* | *Node dag ref dag*

- Decision tree (e.g. for evaluation purposes):

datatype *bdt* = *Zero* | *One* | *Bdt_Node bdt nat bdt*



BDD Formalization - Illustration

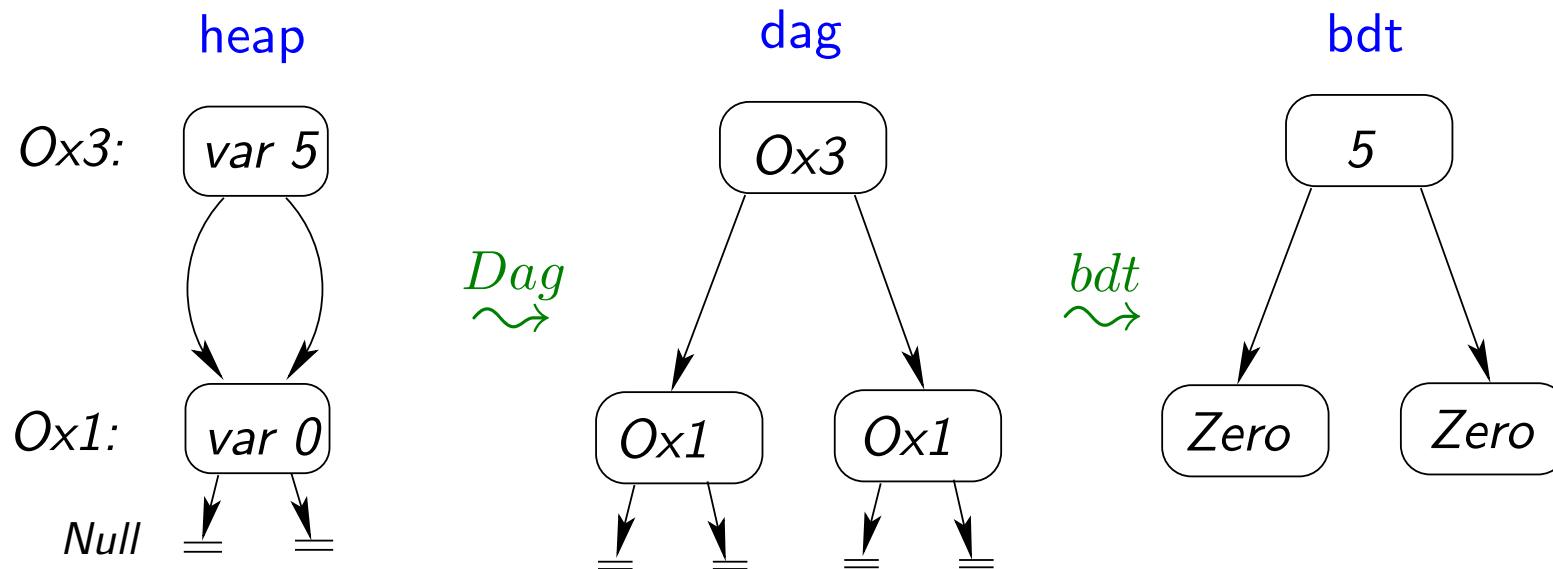
2 levels of specification:

- Graph structure in order to represent *sharing*:

datatype *dag* = *Tip* | *Node dag ref dag*

- Decision tree (e.g. for evaluation purposes):

datatype *bdt* = *Zero* | *One* | *Bdt_Node bdt nat bdt*



BDD Formalization - Illustration

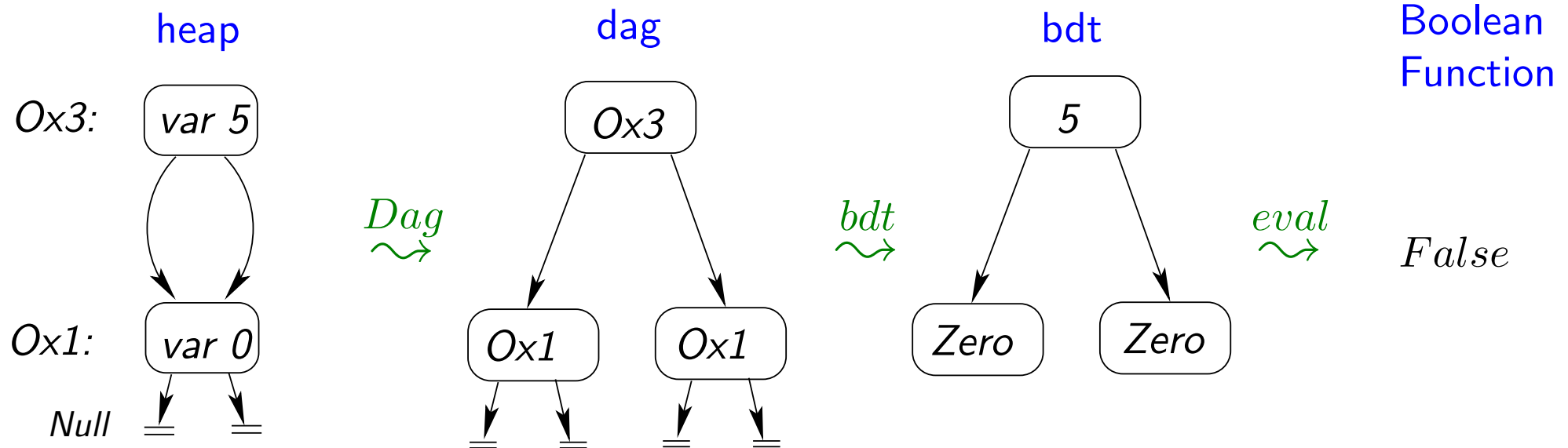
2 levels of specification:

- Graph structure in order to represent *sharing*:

datatype *dag* = *Tip* | *Node dag ref dag*

- Decision tree (e.g. for evaluation purposes):

datatype *bdt* = *Zero* | *One* | *Bdt_Node bdt nat bdt*



Transition Functions

- Construction of a *dag* in the heap:

consts *Dag* :: *ref* $\Rightarrow$ (*ref* $\Rightarrow$ *ref*) $\Rightarrow$ (*ref* $\Rightarrow$ *ref*) $\Rightarrow$ *dag* $\Rightarrow$ *bool*

primrec

Dag *p* *low* *high* *Tip* = (*p* = *Null*)

Dag *p* *low* *high* (*Node* *ldag* *a* *rdag*) = (*p* = *a* $\wedge$ *p* $\neq$ *Null* $\wedge$
Dag (*low* *p*) *low* *high* *ldag* $\wedge$ *Dag* (*high* *p*) *low* *high* *rdag*)

Transition Functions

- Construction of a *dag* in the heap:

consts *Dag* :: *ref* $\Rightarrow$ (*ref* $\Rightarrow$ *ref*) $\Rightarrow$ (*ref* $\Rightarrow$ *ref*) $\Rightarrow$ *dag* $\Rightarrow$ *bool*

primrec

Dag *p* *low* *high* *Tip* = (*p* = *Null*)

Dag *p* *low* *high* (*Node* *ldag* *a* *rdag*) = (*p* = *a* $\wedge$ *p* $\neq$ *Null* $\wedge$
Dag (*low* *p*) *low* *high* *ldag* $\wedge$ *Dag* (*high* *p*) *low* *high* *rdag*)

- Construction of a *bdt* out of a *dag*:

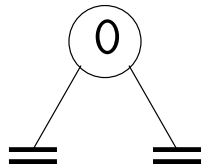
consts *bdt* :: *dag* $\Rightarrow$ (*ref* $\Rightarrow$ *nat*) $\Rightarrow$ *bdt* *option*

Problems with the bdt Conversion

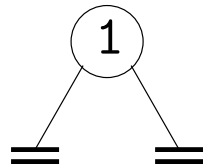
Reasons, why the conversion could go wrong:

- *bdt Tip var* = ?
- *bdt*-endnodes (*Zero/One*) must be represented by an inner node in the *dag*-representation:

Zero node:



One node:



- No inner *bdt*-node may contain variable values 0 and 1
- Balanced structure of *bdt*-nodes

Normalize - Auxiliary Functions for the Specification

- eval:

consts *eval* :: *bdt* $\Rightarrow$ (*nat* $\Rightarrow$ *bool*) $\Rightarrow$ *bool*

Normalize - Auxiliary Functions for the Specification

- eval:

consts $eval :: bdt \Rightarrow (nat \Rightarrow bool) \Rightarrow bool$

- cong_eval ($\sim$):

$bdt_1 \sim bdt_2 \equiv (eval\ bdt_1 = eval\ bdt_2)$

Normalize - Auxiliary Functions for the Specification

- eval:

consts *eval* :: *bdt* $\Rightarrow$ (*nat* $\Rightarrow$ *bool*) $\Rightarrow$ *bool*

- cong_eval ($\sim$):

*bdt*₁ $\sim$ *bdt*₂ $\equiv$ (*eval* *bdt*₁ = *eval* *bdt*₂)

- isomorphic_dags_eq:

isomorphic_dags_eq *dag*₁ *dag*₂ *var* $\equiv$

$\forall bdt_1\ bdt_2. (bdt\ dag_1\ var = Some\ bdt_1 \wedge bdt\ dag_2\ var = Some\ bdt_2 \wedge$
 $(bdt_1 = bdt_2)) \longrightarrow dag_1 = dag_2$

Normalize - Auxiliary Functions for the Specification

- eval:

consts *eval* :: *bdt* $\Rightarrow$ (*nat* $\Rightarrow$ *bool*) $\Rightarrow$ *bool*

- cong_eval ($\sim$):

*bdt*₁ $\sim$ *bdt*₂ $\equiv$ (*eval* *bdt*₁ = *eval* *bdt*₂)

- isomorphic_dags_eq:

isomorphic_dags_eq *dag*₁ *dag*₂ *var* $\equiv$

$\forall bdt_1 bdt_2. (bdt\ dag_1\ var = Some\ bdt_1 \wedge bdt\ dag_2\ var = Some\ bdt_2 \wedge$
 $(bdt_1 = bdt_2)) \longrightarrow dag_1 = dag_2$

- shared:

shared *dag* *var* $\equiv$

$\forall dag_1 dag_2. (dag_1 \leq dag \wedge dag_2 \leq dag) \longrightarrow isomorphic\_dags\_eq\ dag_1\ dag_2\ var$

Normalize - Specification

```
'levellist := replicate ('p→'var + 1) [];  
'levellist := CALL Levellist('p, ¬'p→'mark, 'levellist);  
'n := 0;  
WHILE 'n < length 'levellist  
DO CALL ShareReduceRepList('levellist ! 'n); 'n := 'n + 1 OD;  
'p := CALL Repoint('p)
```

Normalize - Specification

$\forall \sigma \text{ predag prebdt}. \Gamma \vdash$

$\{ \sigma. \text{Dag } 'p \text{ 'low 'high predag} \wedge \text{ordered predag 'var} \wedge$
 $\text{bdt predag 'var} = \text{Some prebdt} \}$

`'levellist := replicate ('p → 'var + 1) [];`

`'levellist := CALL Levellist('p, ¬'p → 'mark, 'levellist);`

`'n := 0;`

`WHILE 'n < length 'levellist`

`DO CALL ShareReduceRepList('levellist ! 'n); 'n := 'n + 1 OD;`

`'p := CALL Repoint('p)`

Normalize - Specification

$\forall \sigma \text{ predag prebdt}. \Gamma \vdash$

$\{ \sigma. \text{Dag } 'p \text{ 'low 'high predag} \wedge \text{ordered predag 'var} \wedge$
 $\text{bdt predag 'var} = \text{Some prebdt} \}$

`'levellist := replicate ('p  $\rightarrow$  'var + 1) [];`

`'levellist := CALL Levellist('p,  $\neg$ 'p  $\rightarrow$  'mark, 'levellist);`

`'n := 0;`

`WHILE 'n < length 'levellist`

`DO CALL ShareReduceRepList('levellist ! 'n); 'n := 'n + 1 OD;`

`'p := CALL Repoint('p)`

$\{ (\exists \text{ postdag}. \text{Dag } 'p \text{ 'low 'high postdag} \wedge \text{reduced postdag} \wedge \text{shared postdag } \mathcal{V}\text{var} \wedge$
 $\text{ordered postdag } \mathcal{V}\text{var} \wedge \text{set\_of postdag} \subseteq \text{set\_of predag} \wedge$
 $(\exists \text{ postbdt}. \text{bdt postdag } \mathcal{V}\text{var} = \text{Some postbdt} \wedge \text{prebdt} \sim \text{postbdt})) \}$

Results

Results

- Complete formalization of the BDD concept
- First machine-checked proof of the normalization algorithms
- Verification of C-like pointer programs possible using the verification environment for Hoare Logic in Isabelle

Thank you for your attention!

Authors: Ana Bove, Thierry Coquand

Title: Formalising Bitonic Sort

Abstract:

Bitonic sort (Batcher, 1968) is one of the fastest sorting networks. A sorting network is a sorting algorithm performing only comparison-and-swap operation on its data. This makes sorting networks, and in particular bitonic sort, very suitable for implementation in hardware or in parallel processor arrays.

Although the algorithm is short and computationally simple, its correctness proof is not very intuitive. In addition, previous formal proofs of the correctness of this algorithm needed to consider around 50 different cases.

In this talk we show how basic notions of distributive lattices help in specifying and proving the correctness of bitonic sort.

Formalising Bitonic Sort

Ana Bove

Partly joint work with Thierry Coquand

Chalmers University of Technology

December 16th, 2004 – TYPES 2004

Formalising Bitonic Sort Twice!

Ana Bove

Partly joint work with Thierry Coquand

Chalmers University of Technology

December 16th, 2004 – TYPES 2004

Content

- Why 2 formalisations of Bitonic Sort?
- Bitonic Sort:
 - * A functional algorithm
 - * How it works
- Brief description of the 2 formalisations of Bitonic Sort in type theory
- Conclusions: Comparison of both formalisations

Why 2 formalisations of Bitonic Sort?

It started as an example within the *CoVer* group at Chalmers.

- First formalisation proved the sorted property for sequences of *Booleans* and used the *0-1 principle* to extend the result to other ordered sets.
- Second formalisation used *Lattice theory* and *NO* use of the 0-1 principle.

Based on Thierry's ideas.

What is Bitonic Sort?

It was introduced by Batcher in 1968.

It is one of the fastest sorting algorithm where the sequence of comparisons is not data-dependent

(such algorithms are called *sorting networks*).

It consists of $O(n * \log(n)^2)$ comparisons in $O(\log(n)^2)$ stages.

This makes sorting networks suitable for implementation in hardware or in parallel processor arrays.

Bitonic Sort Algorithm

Bitonic sort works on sequences of 2^n elements.

Bitonic Sort Algorithm

Bitonic sort works on sequences of 2^n elements.

Haskell implementation that works on complete and balanced trees of height n (von Sydow):

```
data Tree a = Lf a | Bin (Tree a) (Tree a)
```

```
reverseT (Lf x) = Lf x
```

```
reverseT (Bin l r) = Bin (reverseT r) (reverseT l)
```

Bitonic Sort on Trees

```
sortTree :: Tree a -> Tree a
sortTree (Lf x) = Lf x
sortTree (Bin l r) = merge (Bin (sortTree l)
                                (reverseT (sortTree r)))
```

```
merge (Lf x) = Lf x
merge (Bin l r) = Bin (merge l1) (merge r1)
  where (l1,r1) = min-max_Swap l r
```

```
min-max_Swap (Lf x) (Lf y) = (Lf (x /\ y), Lf (x \/ y))
min-max_Swap (Bin l1 r1) (Bin l2 r2) = (Bin a c, Bin b d)
  where (a,b) = min-max_Swap l1 l2
        (c,d) = min-max_Swap r1 r2
```

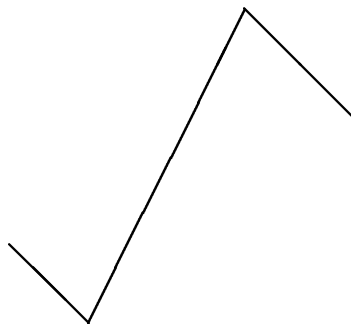
Bitonic Sequence

Definition: A sequence $a_1, a_2, \dots, a_n$ is *bitonic* if there is a k such that either

$$a_1 \leq a_2 \leq \dots \leq a_k \geq \dots \geq a_n$$

or there is a cyclic shift of the sequence such that this is true.

Bitonic sequences have the general shape of a “*bolt*”:



How Does It Work?

How Does It Work?

Main property:

Let $ls * rs$ be a *bitonic* sequence and let $(as, bs) = \text{min-max\_Swap } ls \ rs$. Then

- as and bs are both bitonic sequences
- $\forall a \in as. \forall b \in bs. a \leq b$

Towards a Formalisation

We work on sequences.

A sequence is a function from a *decidable linear order* to an *ordered set*.

Towards a Formalisation

We work on sequences.

A sequence is a function from a *decidable linear order* to an *ordered set*.

We need to compare the elements in the sequences.

We formalise the ordered sets as *linear orders* and we define the *minimum* ($\wedge$) and *maximum* ($\vee$) operations in the normal way.

Towards a Formalisation

We work on sequences.

A sequence is a function from a *decidable linear order* to an *ordered set*.

We need to compare the elements in the sequences.

We formalise the ordered sets as *linear orders* and we define the *minimum* ($\wedge$) and *maximum* ($\vee$) operations in the normal way.

Main Problem: Not easy to reason about bitonic sequences!

A formalisation of bitonic sort in PVS (Couturier, 1998)

- needed to consider 56 cases!
- lemmas rather incomprehensible

First Formalisation

- We formalise sequences as *dependent binary trees*
- We reduce the linear order (with the elem. to sort) to the set of *Booleans*
- We define *labels* on boolean trees: O, I, OI, IO, OIO, IOI, W
- We use the *0-1 principle* (Knuth, 1973; Bouricius, ca 1954) to extend the result to other ordered sets
- We use the *parametricity theorem* (Reynolds, 1983) to prove the 0-1 principle for our bitonic sort (Day, Launchbury and Lewis, 1999)
- Max. number of cases in the main proofs: 24 (+ 23 empty cases) which can be grouped in 6 main cases.

Second Formalisation

- We formalise sequences as functions from linear orders to linear orders
(Can be generalised to distributive lattices)
- The sequence $f: I \rightarrow D$ is a *bitonic sequence* if

$$\begin{aligned} \forall i, j, k, l: I. \quad i < j \ \& \ j < k \ \& \ k < l \Rightarrow \\ & f(i) \wedge f(k) \leq f(j) \vee f(l) \ \& \\ & f(j) \wedge f(l) \leq f(i) \vee f(k) \end{aligned}$$

- Max. number of cases in the main proofs: 3 (+ 10 empty cases)!
- Elegant proof!

Conclusions

Comparing the second formalisation to the first one:

- Consider fewer cases than previous formalisations: 3 (+ 10)
 - * 24 (+ 23) grouped in 6 subcases using the 0-1 principle in Agda
 - * 56 in PVS using general definition of bitonic sequences
- Less intuitive: mainly proving “mathematical” properties
- Very few proofs by induction
- Few “annoying” inequality-reasoning proofs
- Good example for testing Agda/FOL and Agda/AgSy
- Good test case for Agda!

Author: Christophe Raffalli

Title: Proof by contextual menu

Abstract:

We will describe the concept of proof by contextual menu whose goal is to allow a newcomer to use a theorem prover knowing only a few concept. To achieve this goal, the text of the menus should be very clear (like assume A and prove B) and therefore they are build by the prover (from the click position and an eventuel selection) and sent to the interface.

Author: Catarina Coquand

Title: Overloading in Agda

Abstract:

We will present a suggestion for overloading in Agda. The motivation for this work was that we wanted to add built-in types for Agda such as Integer, Rational, Character and Strings. It is natural for these types to have overloaded symbols, for example +.

The solution taken is type-classes similar to Haskell. By translating the classes and their instances to signatures and records that already exists in Agda, we don't have to change the theory for the language.

Beside the overloaded functions for built/in types we will also show programs with states represented in a monadic style.

Overloading in Agda

Catarina Coquand

December 15, 2004

What is Agda?

Not important for this talk, we will talk about a dependent type theory with

1. signature and records
2. meta-variables
3. hidden arguments

Example : Expressions

```
data Exp = EVar (s :: String)
         | ENum (n :: Nat)
         | EPlus (e1, e2 :: Exp)

eval :: Env -> Exp -> Maybe Nat
eval = \env -> \e ->
  case e of
    (EVar s) -> lookup env s
    (ENum n) -> return n
    (EPlus e1 e2) -> do i1 <- eval env e1
                        i2 <- eval env e2
                        return (nat_plus i1 i2)
```

Example : Exp with Integer

```
data Exp = EVar (s :: String)
         | ENum (n :: Integer)
         | EPlus (e1,e2 :: Exp)

eval :: Env -> Exp -> Maybe Integer
eval = \env -> \e ->
  case e of
    (EVar s) -> lookup env s
    (ENum n) -> return n
    (EPlus e1 e2) -> do i1 <- eval env e1
                        i2 <- eval env e2
                        return (integer_plus i1 i2)
```

How would we like to write eval

```
eval = \env -> \e ->
  case e of
    (EVar s) -> lookup env s
    (ENum n) -> return n
    (EPlus e1 e2) -> do i1 <- eval env e1
                        i2 <- eval env e2
                        return (i1 + i2)
```

Why Overloading?

- Readability
- Reusability (of both code and proofs)

What Kind of Overloading?

- Type directed overloading
- User-defined
- Similar to the Classes in Haskell
- Similar to Axiomatic Type Classes in Isabelle

Example: Plus

```
class Plus (a::Set) :: Set exports
  (+) :: a -> a -> a

instance plusNat :: Plus Nat where
  (+) = \n n' -> case n of
    (Zero) -> n'
    (Succ m) -> Succ (m + n')

instance plusInteger :: Plus Integer where
  (+) = .....
```

Example: Use of Plus

```
twiceNat :: Nat -> Nat
twiceNat = \x -> x + x

-- twiceNat = \x -> (+) Nat plusNat x x

twiceInt :: Integer -> Integer
twiceInt = \x -> x + x

-- twiceInt = \x -> (+) Integer plusInteger x x
```

Example ctd: Use of Plus

```
twice (|a::Set)(|pa::Plus a) :: a -> a
twice = \x -> x + x

-- twice = \x -> (+) a pa x x

twiceInt :: Integer -> Integer
twiceInt = \x -> twice x
-- twiceInt = \x -> twice Integer plusInteger x
```


Example: Properties

```
class PlusProp(a::Set) :: Set extends (plusa::Plus a) exports
  assoc_plus :: (x,y,z::a) -> (x+y)+z == x+(y+z)
  comm_plus  :: (x,y::a) -> x+y == y+x
  ....
instance natPlusProp :: PlusProp Nat where
  assoc_plus = ?
  comm_plus  = ?
  ...

lemma (|a::Set)(ppa::PlusProp a)
  :: (x,y,z::a) -> (x+y)+z == z+(y+x)
lemma = ?
```

Example: Monads

```
class Monad (m :: Set -> Set) :: Type exports
  (>>=)  :: (a,b::Set) |-> m a -> (a -> m b) -> m b
  (>>)   :: (a,b::Set) |-> m a -> m b -> m b
  return :: (a::Set) |-> a -> m a
```

Typechecking Classes and Overloaded Functions

- Classes are translated to signatures
- Instances are translated to records
- Overloaded functions are implemented as projections
- Overloaded functions calls are saturated with meta-variables, i.e. $5 + 7$ is replaced by $(+) ?_1 ?_2 5 7$

Typechecking then yields constraints, such as for example: $?_2 :: \textit{Plus Integer}$ these are solved by a special constraint-solver for classes.

Possible Extensions

- Multi-parameter classes seems easy
- Default definitions
- Since we can name instances, we could have overlapping instances that is not used in constraint solving

Conclusions

- A “light-way” to add a rather powerful technique for overloading
- Works nicely for the intended application: to be able to overload standard functions.

Authors: Claudio Sacerdoti Coen, Stefano Zacchiroli

Title: Efficient Type Based Parsing in Strict Languages

Abstract:

We present an efficient technique for parsing ambiguous mathematical formulae. Our technique can be applied when the output of parsing and semantic analysis is a calculus that can be extended with metavariables and exploits the refinement function. We compare the efficiency of an implementation based on the Calculus of (co)Inductive Constructions with the state of the art.

Efficient Ambiguous Parsing of Mathematical Formulae

Claudio Sacerdoti Coen

`<sacerdot@cs.unibo.it>`

Stefano Zacchioli

`<zacchiro@cs.unibo.it>`

University of Bologna

December 16, 2004 - Jouy-en-Josas (France)

Project PCRI, CNRS, École Polytechnique,
INRIA, Université Paris-Sud.

MoWGLI: Math on the Web, Get it by Logic
and Interfaces, EU IST-2001-33562.

Summary

1. Overloading in formal mathematics
2. Type based parsing: the inefficient way
3. Exploiting metavariables
4. Benchmarks and comparisons

Forms of Ambiguity

$$(1 + 2) * 3 = 9$$

- natural, integer, rational, real, complex, ...?
- which representation?
Peano numbers, Church numerals, binary integers, ...
- which equality?
Leibniz's polymorphic equality, decidable monomorphic equality for natural numbers, undecidable equality for real numbers, ...?
- coercions? implicit arguments?

Only a few combinations of choices are well-typed

A naive solution



Desired properties

1. **compositionality**: the semantic analysis phase should be as separate as possible from the parsing phase
2. **efficiency**: semantic errors must be detected as early as possible, avoiding duplicate efforts. $(1? +? 2?) *_{\mathbb{R}} 3? =_{\mathbb{N}} 9?$

Laziness is not enough

$$\frac{
 \begin{array}{c}
 \bigvee \\
 g : B \rightarrow T'
 \end{array}
 \quad
 \frac{
 \begin{array}{c}
 \bigvee \\
 \Gamma \vdash f : \Pi x : A. T(x)
 \end{array}
 \quad
 \begin{array}{c}
 \bigvee \\
 \Gamma \vdash M : A'
 \end{array}
 \quad
 \begin{array}{c}
 \bigvee \\
 \Gamma \vdash A \equiv A'
 \end{array}
 }{
 \Gamma \vdash (f \ M) : (T \ M)
 }
 \quad
 \begin{array}{c}
 \bigvee \\
 \Gamma \vdash (T \ M) \equiv B
 \end{array}
 }{
 \Gamma \vdash (g \ (f \ M)) : T'
 }$$

- a mismatch between the output type of f and the input type of g is detected by $\Gamma \vdash (T \ M) \equiv B \dots$
- ...but only after the recursion $\Gamma \vdash M : A'$
- $\Gamma \vdash M : A'$ cannot be postponed (otherwise $\Gamma \vdash (T \ M) \equiv B$ can diverge)

Exploiting metavariables

$$\frac{
 \begin{array}{c}
 \bigvee \\
 g : B \rightarrow T'
 \end{array}
 \quad
 \frac{
 \begin{array}{c}
 \bigvee \\
 \Gamma \vdash f : \Pi x : A. T(x)
 \end{array}
 \quad
 \Gamma \vdash ?_1 : ?_2
 \quad
 \frac{
 \begin{array}{c}
 \bigvee \\
 \Gamma \vdash A \equiv ?_2
 \end{array}
 }{
 \Gamma \vdash (f \ ?_1) : (T \ ?_1)
 }
 \quad
 \frac{
 \begin{array}{c}
 \bigvee \\
 \Gamma \vdash (T \ ?_1) \equiv B
 \end{array}
 }{
 }
 }{
 \Gamma \vdash (g \ (f \ ?_1)) : T'
 }$$

- introduce **metavariable** in the calculus
- replace conversion with unification and type-checking with refinement
- the unification and refinement judgements are three valued:
 σ (Keep) ϵ (Prune) $\perp$ (Keep)

An example

[1/2]

E.g.: parsing $(5/2)!$ where $/ \in \{/_\mathbb{N}, /_\mathbb{R}\}$ and $2, 5 \in \mathbb{N} \cup \mathbb{R}$

$$D = \begin{cases} / & \mapsto [/_\mathbb{N} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} ; /_\mathbb{R} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}] \\ 2 & \mapsto [2 : \mathbb{N} ; 2 : \mathbb{R}] \\ 5 & \mapsto [5 : \mathbb{N} ; 5 : \mathbb{R}] \end{cases}$$

$$\phi_0 = \phi_{???} \begin{cases} / & \mapsto Placeholder \\ 2 & \mapsto Placeholder \\ 5 & \mapsto Placeholder \end{cases}$$

$$\Phi_0 = \{\phi_0\}$$

$$\begin{aligned} refine(t_\uparrow(\phi_0)) = \\ refine(?_1!) = [?_1 : \mathbb{N}] \end{aligned}$$

An example

[2/2]

$$\phi_{1??} = \begin{cases} / & \mapsto /_{\mathbb{N}} \\ 2 & \mapsto Placeholder \\ 5 & \mapsto Placeholder \end{cases}$$

$$\begin{aligned} refine(t_{\uparrow}(\phi_1)) &= \\ refine((?_1 /_{\mathbb{N}} ?_2)!) &= \\ [?_1 : \mathbb{N} ; ?_2 : \mathbb{N}] \end{aligned}$$

$$\phi_{2??} = \begin{cases} / & \mapsto /_{\mathbb{R}} \\ 2 & \mapsto Placeholder \\ 5 & \mapsto Placeholder \end{cases}$$

$$\begin{aligned} refine(t_{\uparrow}(\phi_1)) &= \\ refine((?_1 /_{\mathbb{R}} ?_2)!) &= \epsilon \end{aligned}$$

$\Phi_1 = \{\phi_{1??}\}$ 4 ASTs pruned at once

...

$\Phi_2 = \{\phi_{11?}\}$ 2 ASTs pruned at once; $t_{\uparrow}(\phi_{11?}) = (2_{\mathbb{N}} /_{\mathbb{N}} ?_1)!$

...

$\Phi_3 = \{\phi_{111}\}$ 1 AST pruned; $t_{\uparrow}(\phi_{111}) = (2_{\mathbb{N}} /_{\mathbb{N}} 5_{\mathbb{N}})!$ is well typed and closed

Benchmarks

NA	PA	I	NA	PA	I	NA	PA	I	NA	PA	I
1	2	-1	32	33	-1	128	38	90	1280	43	1237
1	4	-3	32	33	-1	128	38	90	1536	54	1482
1	6	-5	32	33	-1	160	38	122	1536	48	1488
3	5	-2	32	33	-1	192	108	84	1680	35	1645
4	6	-2	32	33	-1	192	111	81	1792	51	1741
4	7	-3	32	39	-7	224	46	178	2688	54	2634
8	7	1	32	33	-1	224	45	179	3584	55	3529
14	24	-10	42	13	29	224	47	177	3584	54	3530
21	13	8	63	83	-20	256	40	216	7168	63	7105
32	32	0	63	83	-20	320	20	300	8192	60	8132
32	33	-1	63	19	44	480	41	439	14336	62	14274
32	38	-6	96	37	59	512	45	467	21504	65	21439
32	33	-1	128	40	88	512	41	471	21504	60	21444
32	33	-1	128	43	85	896	51	845	36864	79	36785
32	33	-1	128	42	86	896	51	845	53760	65	53695
32	33	-1	128	42	86	896	51	845	53760	67	53693
32	40	-8	128	39	89	896	49	847			
32	39	-7	128	38	90	896	47	849			
32	33	-1	128	38	90	1024	42	982			
32	33	-1	128	38	90	1280	44	1236			
32	33	-1	128	38	90	1280	43	1237			

Comparison to Coq notational scopes [1/3]

Coq notational scopes:

- one active scope at a time (e.g. *nat*, *R*, *types*)
- no overloading inside one scope
- a concrete syntax to change the scope (e.g. $(\pi + (5/2)\%nat)\%R$)
- notational scopes associated to operator arguments
(e.g. $length : \forall A\%types, (list A)\%list \rightarrow nat$)

Cfr. Weak Type Theory

Comparison to Coq notational scopes [2/3]

Pros:

- Since the weak type system is unrelated to the Coq type system it is more general.
- Maximal modularity.

Comparison to Coq notational scopes [3/3]

Cons:

- Serious problem of consistency between the two type systems (the weak types are assigned by the user).
- Puts burden on the user.
- Does not really exploits much of the environment even if available.
- Weaker than our approach. E.g. $id : \forall T : \mathbf{Set}. T \rightarrow T$ No scope is associated to the second argument T . Thus $(id \mathbb{N} 0)$ must be written as $(id \mathbb{N} 0\%nat)$ (even if $(id \mathbb{N})$ is known to have type $\mathbb{N} \rightarrow \mathbb{N}$).

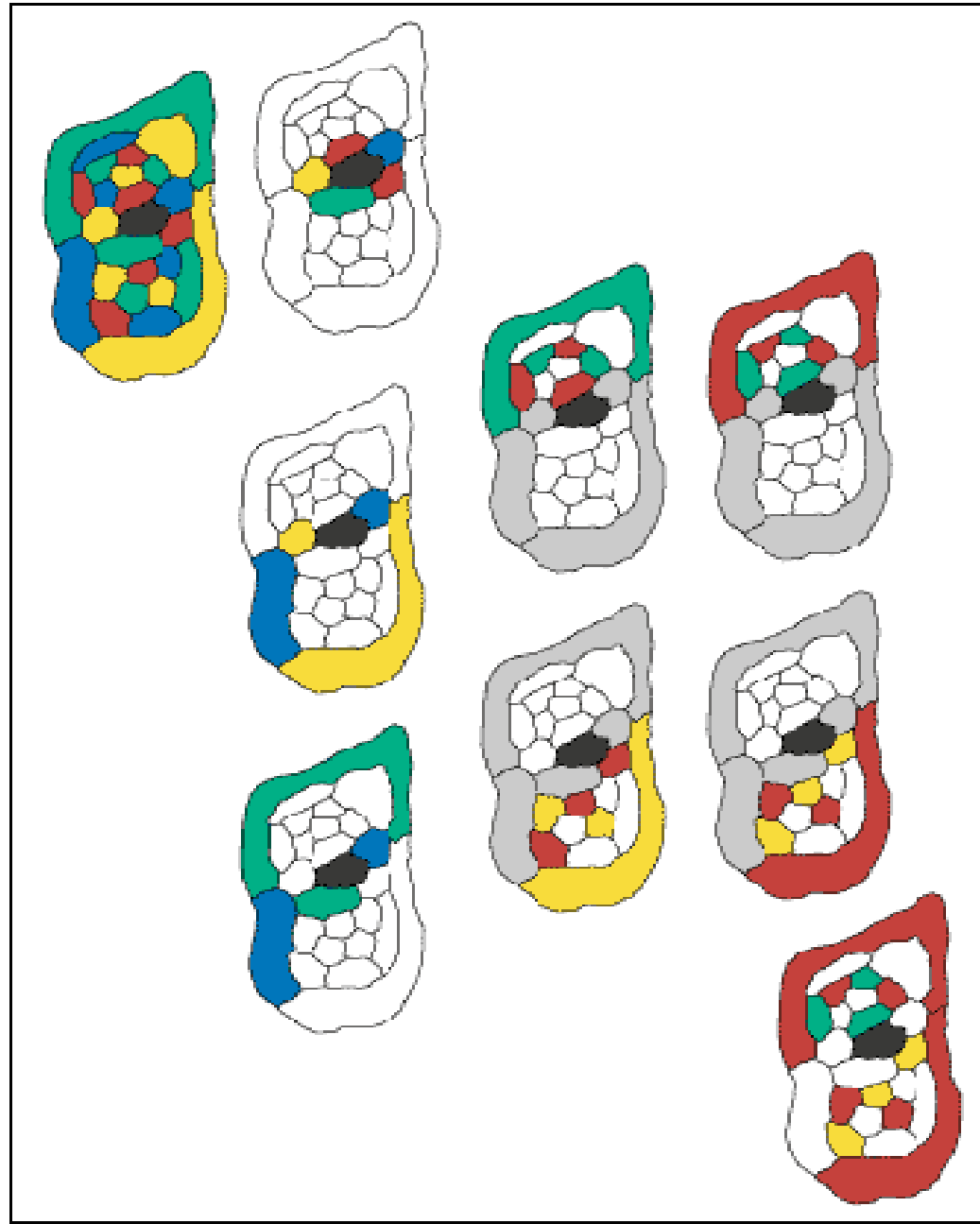
FLYSPECK

Toward a Formal Proof of the Kepler Conjecture

December 17, 2004

T. HALES

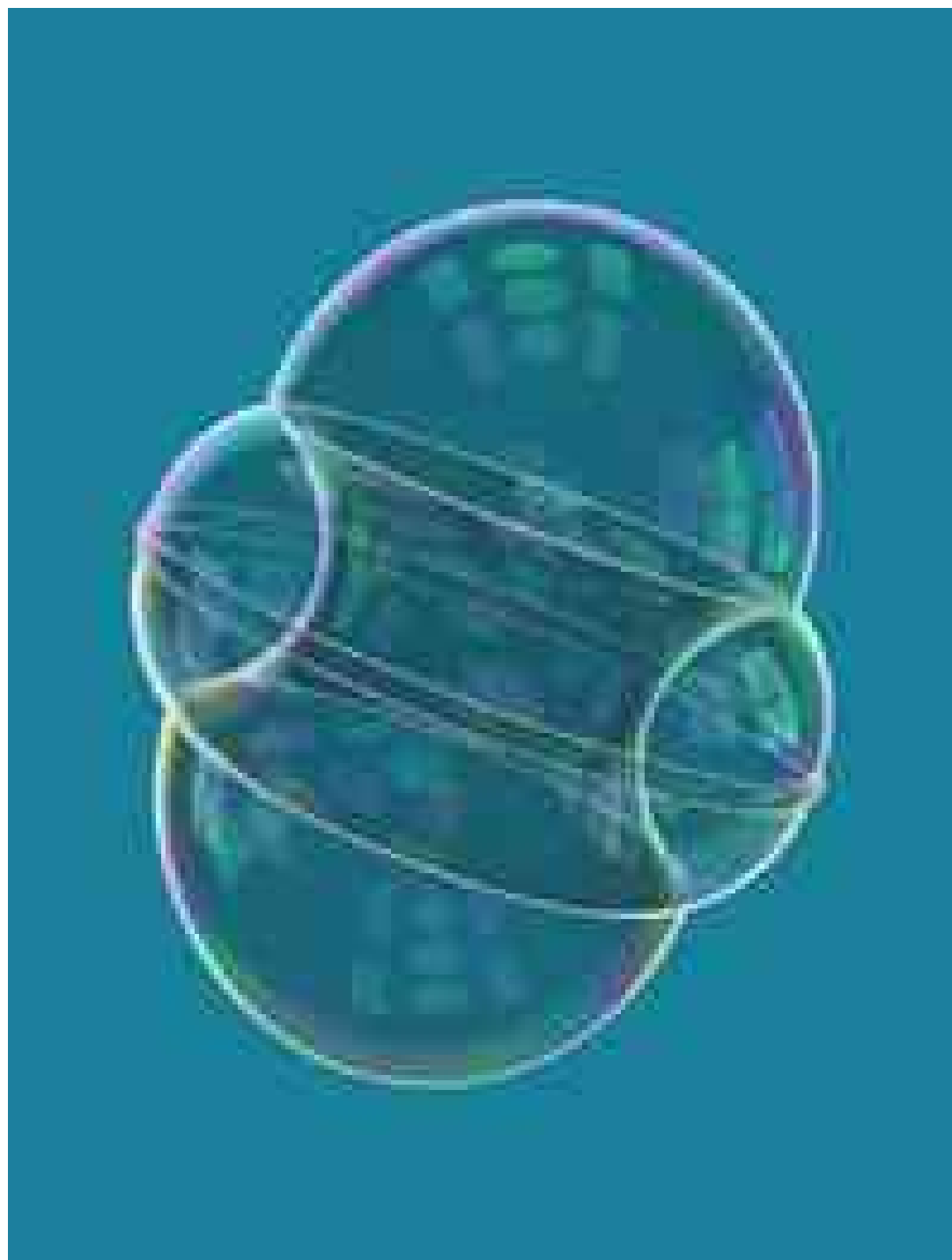
THALES







— 2019 —

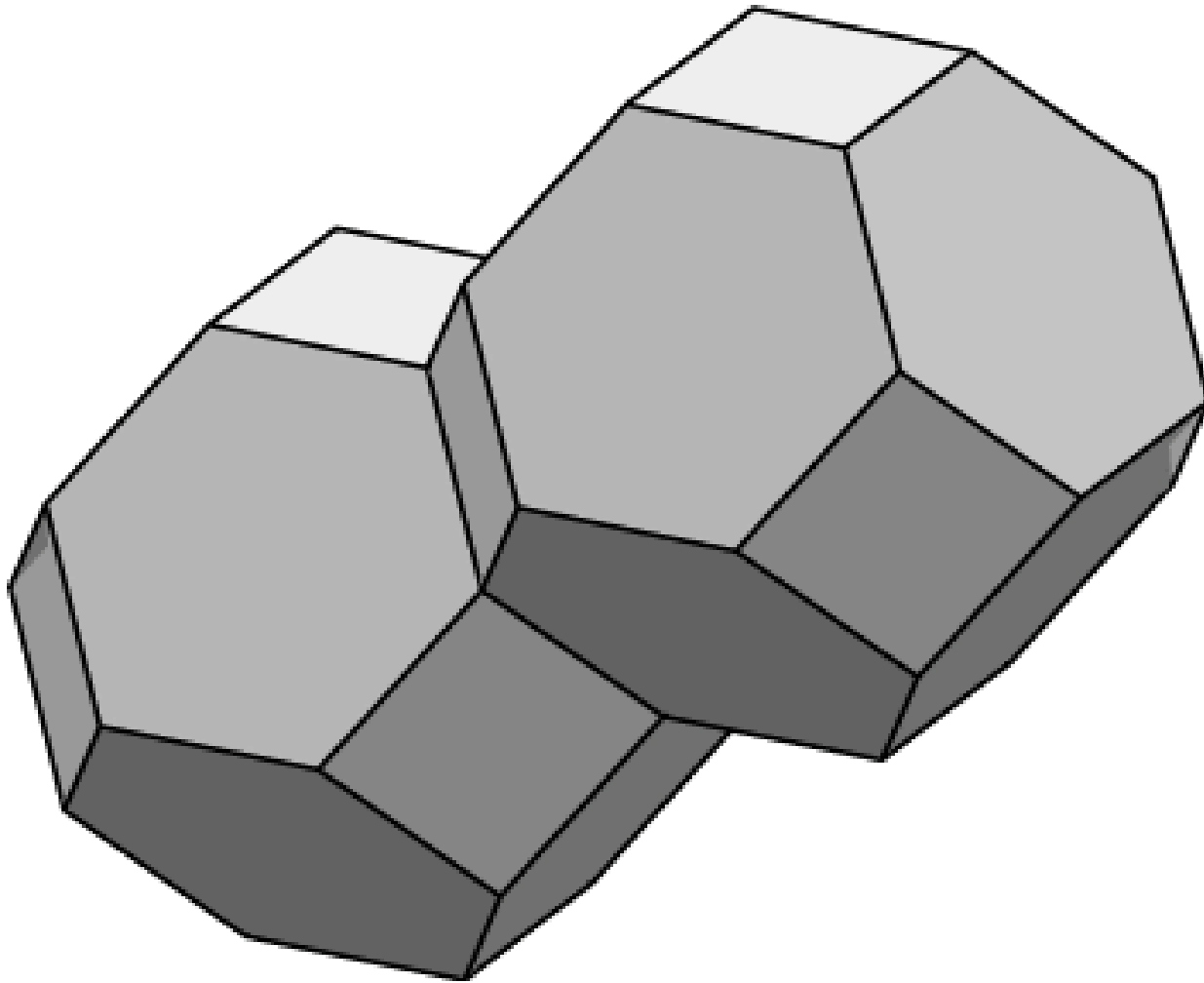


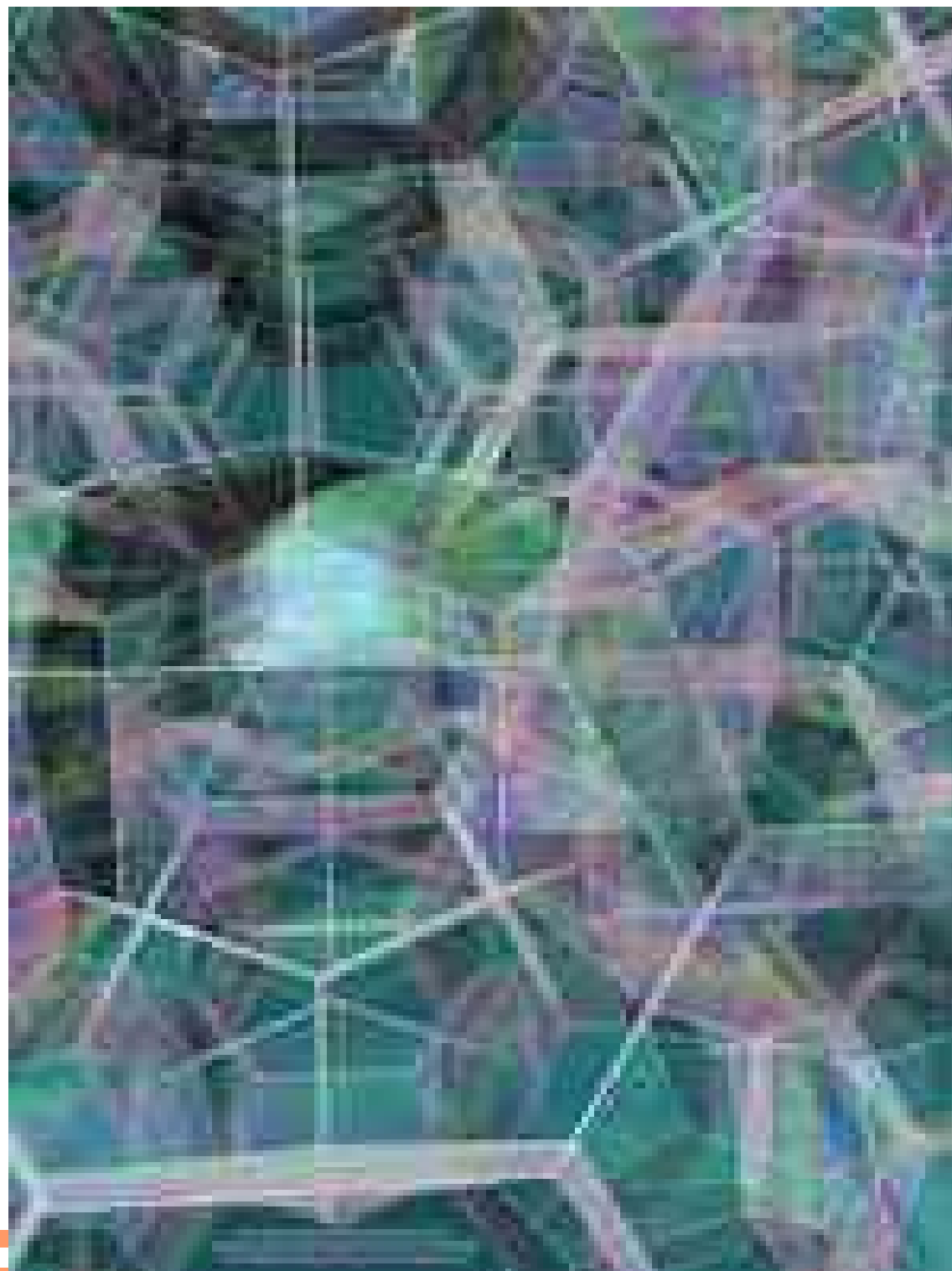
Double Bubble Conjecture

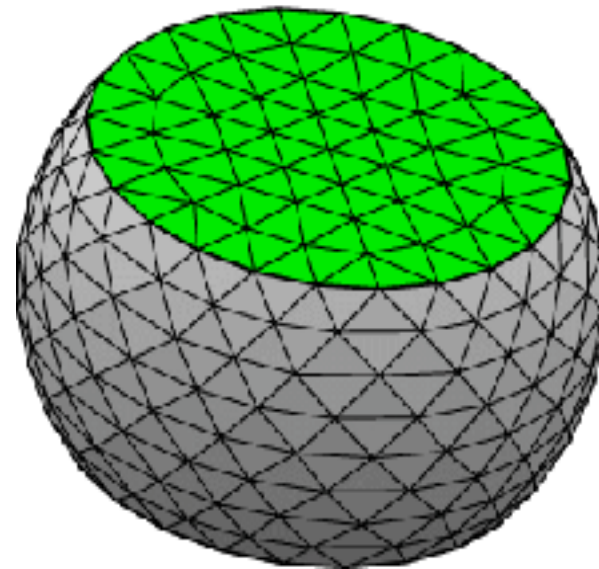
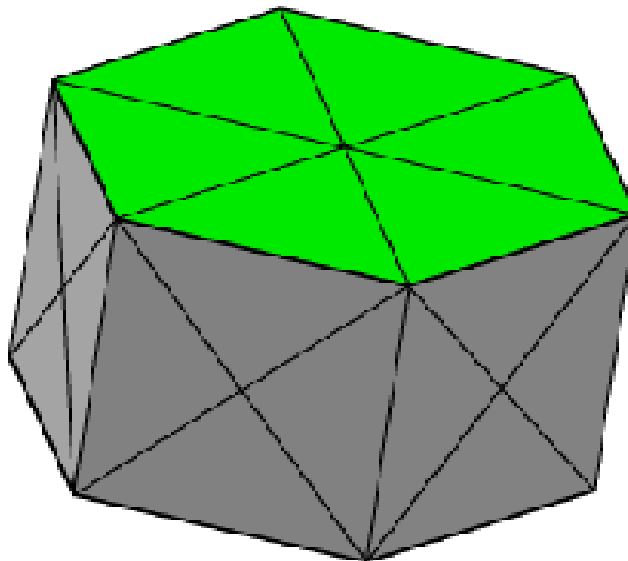
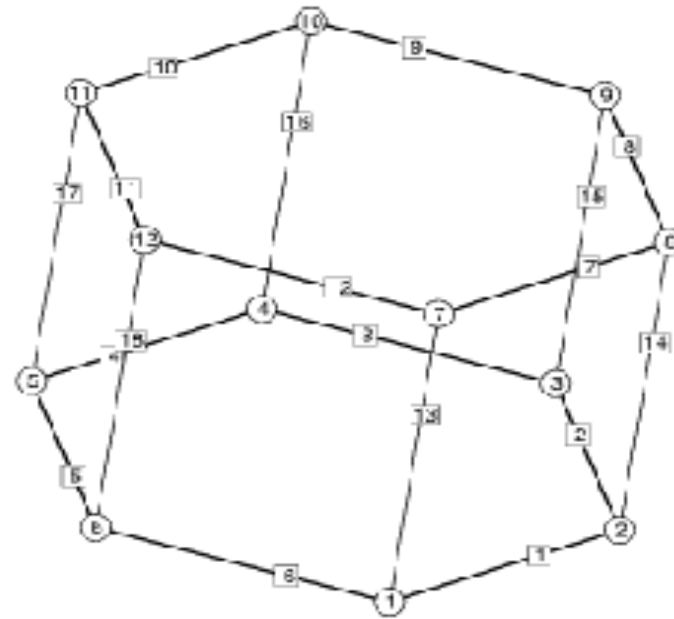
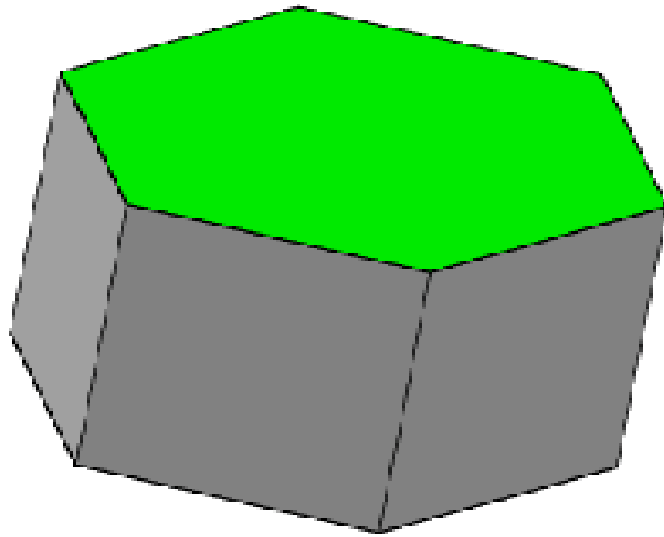
- ◆ What is the surface minimizing way of enclosing two equal volumes?
- ◆ In 1911, David Boys in his book on Soap Bubbles described the appearance of two bubbles.
- ◆ Proved by Hass, Hutchings, and Schlafly in 1995.
- ◆ The proof is a mixture of computer calculations and traditional paper-and-pencil mathematics.
- ◆ The computer calculations use interval arithmetic to rule out
- ◆ other possible configurations.

Proof of the Double Bubble Conjecture

- ◆ The proof relies on computer calculations that show that torus bubbles are
 - ◆ Not minimal surfaces, or
 - ◆ Have area greater than the standard double bubble.
- ◆ Calculations used IEEE 754 standard floating point.
- ◆ The code takes about 10 seconds to run on a fast (1999) PC and examines about 15K rectangles.

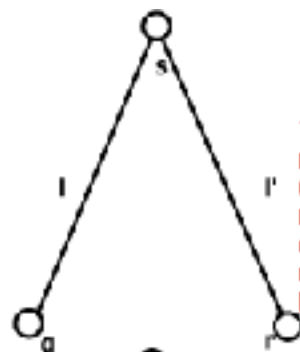






Kelvin Problem

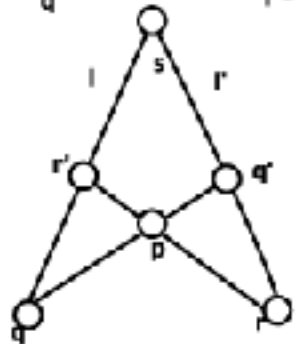
- ◆ Find the surface area minimizing partition of space into equal volume cells.
- ◆ A counterexample was constructed by Phelan and Weaire using an educated guess and Brakke's "surface evolver software."
- ◆ The output from the computer software was later rigorously checked by John Sullivan.



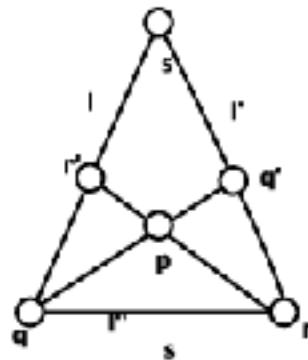
1. By the axioms for a projective plane, there are four points (p, q, r, s) , no three of which are collinear. So there must be two distinct intersecting lines l and l' as pictured.



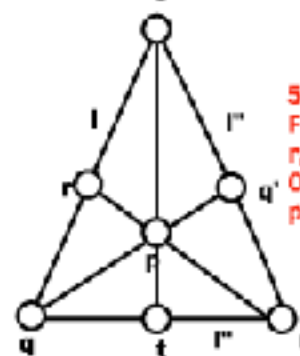
2. Let q and r be the points that lie on l and l' , respectively, that are distinct from the point of intersection s , and let p be the fourth point that lies on neither l nor l' .



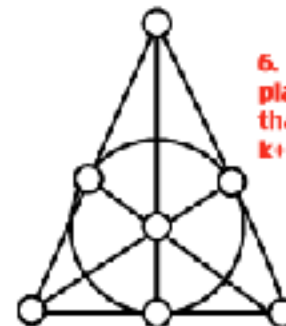
3. Now any two points determine a unique line, and any two lines intersect at a unique point. So create a line from l to l' from q through p . In addition, create another line from l' to l from r through p . We will then have the intersection points q' and r' as illustrated.



4. Now, by the axioms for a projective plane, all points must be connected, thus, we connect q and r , creating line l'' . Note that for any two lines (not through p), we must be able to go from a given point on any line, through p , to some point on another given line. Thus, we must create a line from s through p , to l'' .



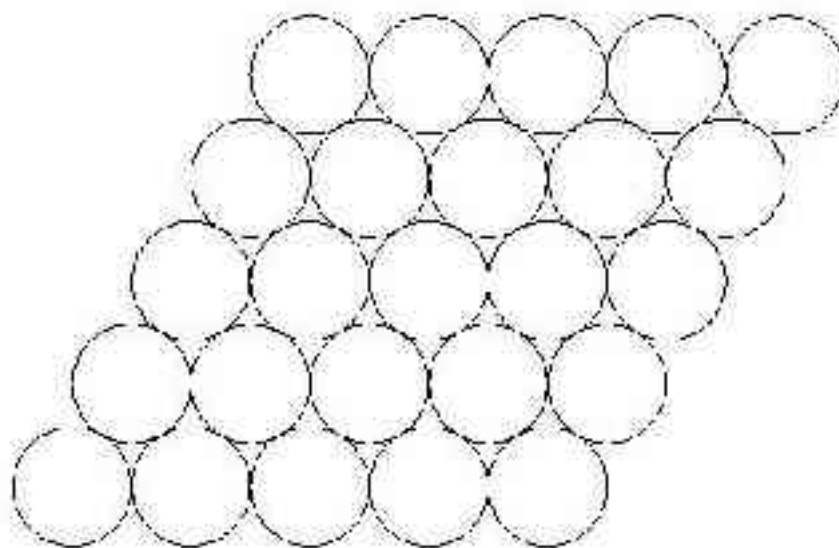
5. We will call our new point t . Finally, we must have our points r, q' and t connected by a line. Our result is the Fano plane!



6. The Fano plane, and note that $k+1 = 3$.

Nonexistence of the projective plane of order 10

- ◆ In 1989, Lam, Thiel, and Swiercz proved the non-existence.
- ◆ The calculation relied on an estimated 2000 hours of CRAY computations.
- ◆ “Because of the use of a computer, one should not consider these results as a `proof` in the traditional sense,... They are experimental results and there is always a possibility of mistakes.”



Lagrange's Theorem

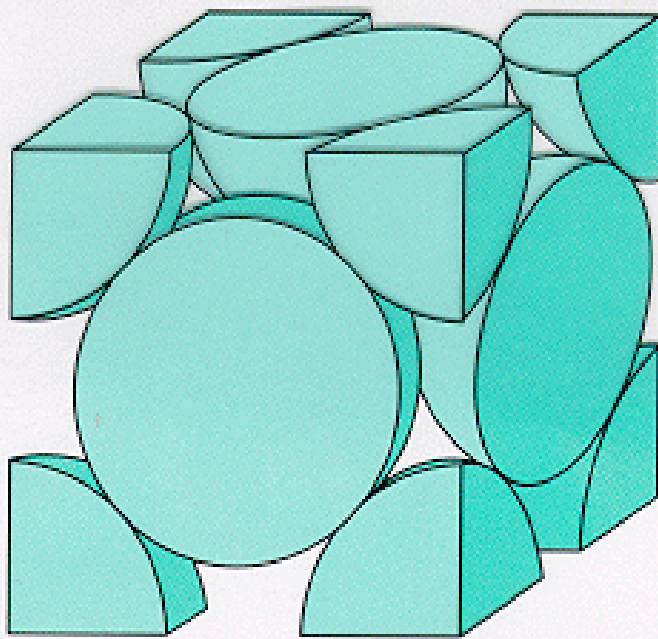
- ◆ In 1773, Lagrange proved that the best lattice packing of equal disks is the obvious solution.
- ◆ The general 2D packing problem was not settled until around 1900.
- ◆ The best lattices were found in dimensions
 - ◆ 3 by Gauss in 1836
 - ◆ 4, 5, 6, 7, 8 Korkine and Zolotareff, Blichfeldt (1935)
- ◆ The best lattice was found in dimension
 - ◆ 8, 24 by H. Cohn and A. Kumar in 2004.

Cohn and Kumar's proof of the densest lattice packing in 24 dimensions

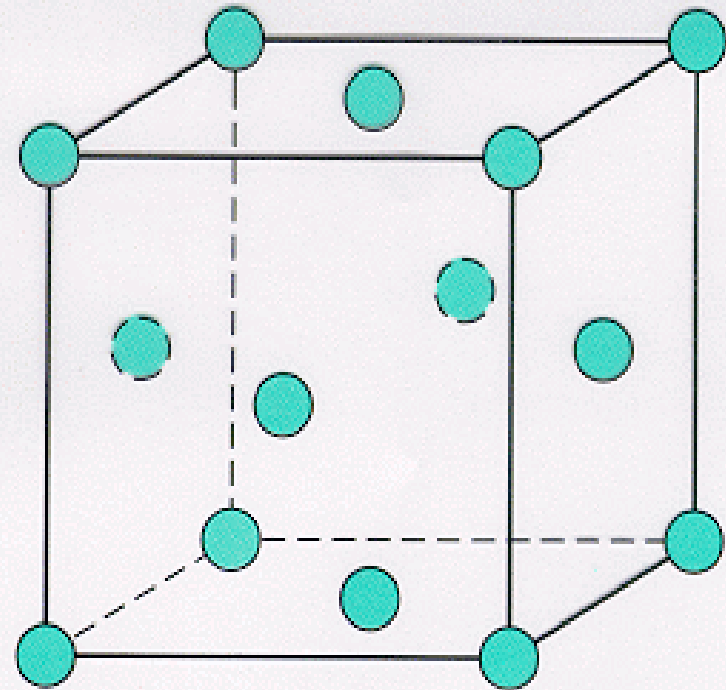
- ◆ The proof relies on computer calculations performed in PARI.
- ◆ The calculations use high-precision floating point arithmetic (3000 digits).
- ◆ The calculations take about 6 hours on a PC.

Hyperbolic manifolds

- ◆ D. Gabai, Meyerhoff, and N. Thurston (2003) proved that homotopy hyperbolic 3-manifolds are hyperbolic.
- ◆ The proof relies on long computer calculations (about 60 CPU days).
- ◆ The calculations use IEEE-754. They do not use interval arithmetic, rather they make error-estimates in the paper that become part of the computer code.
- ◆ “We pose the simple question: why should one have confidence in our proof?”



(a)



(b)

The Kepler Conjecture

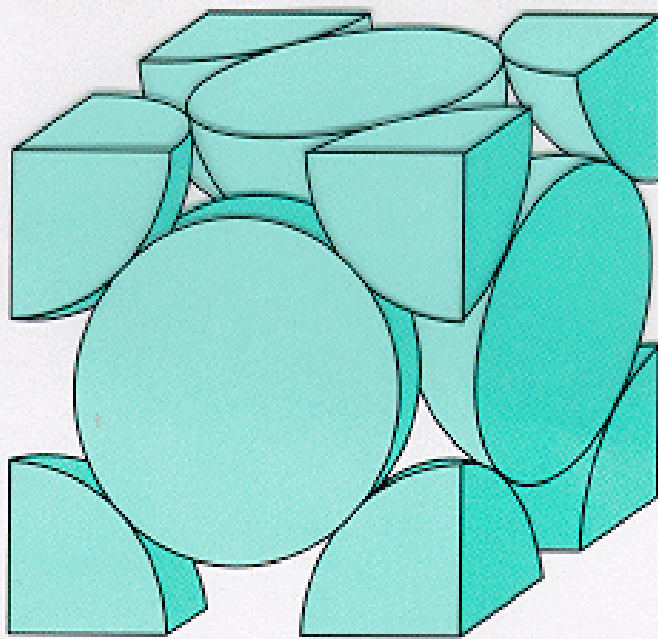
- ◆ (Kepler 1610) The cannonball packing is (one of) the densest packings of sphere in 3D.
- ◆ The problem was solved in 1998 by computers.
 - ◆ Interval arithmetic ~3 months on a PC.
 - ◆ Linear programming ~1 week on a PC.
 - ◆ Planar graph combinatorics, a few hours.
- ◆ The FlysPecK project = Formal Proof of the Kepler conjecture

Referees:

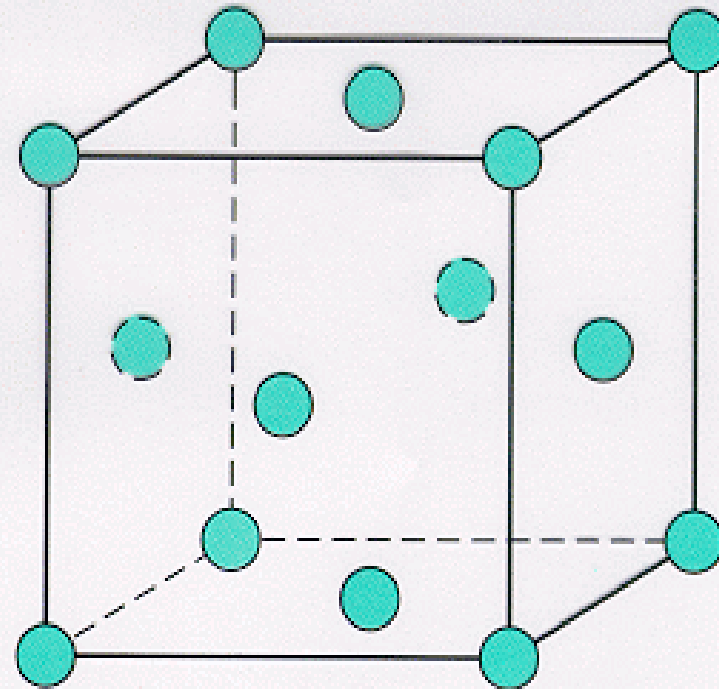
- ◆ 12 referees were assigned to the paper.
- ◆ After 6 months the referees were 99% certain of the proof.
- ◆ After 5 more years of checking, the referees were 99% certain of the proof.

The official Annals of Math policy on computers (2004)

- ◆ The human part of the proof ... will be refereed for correctness in the traditional manner.
- ◆ The computer part will be examined for the methods by which the authors have eliminated or minimized possible sources of error: (e.g. round-off error eliminated by interval arithmetic,...).
- ◆ [A] second and independent implementation of a computer assisted proof has its place, and should be properly recognized just as a duplication of an experiment is recognized in physics.



(a)



(b)

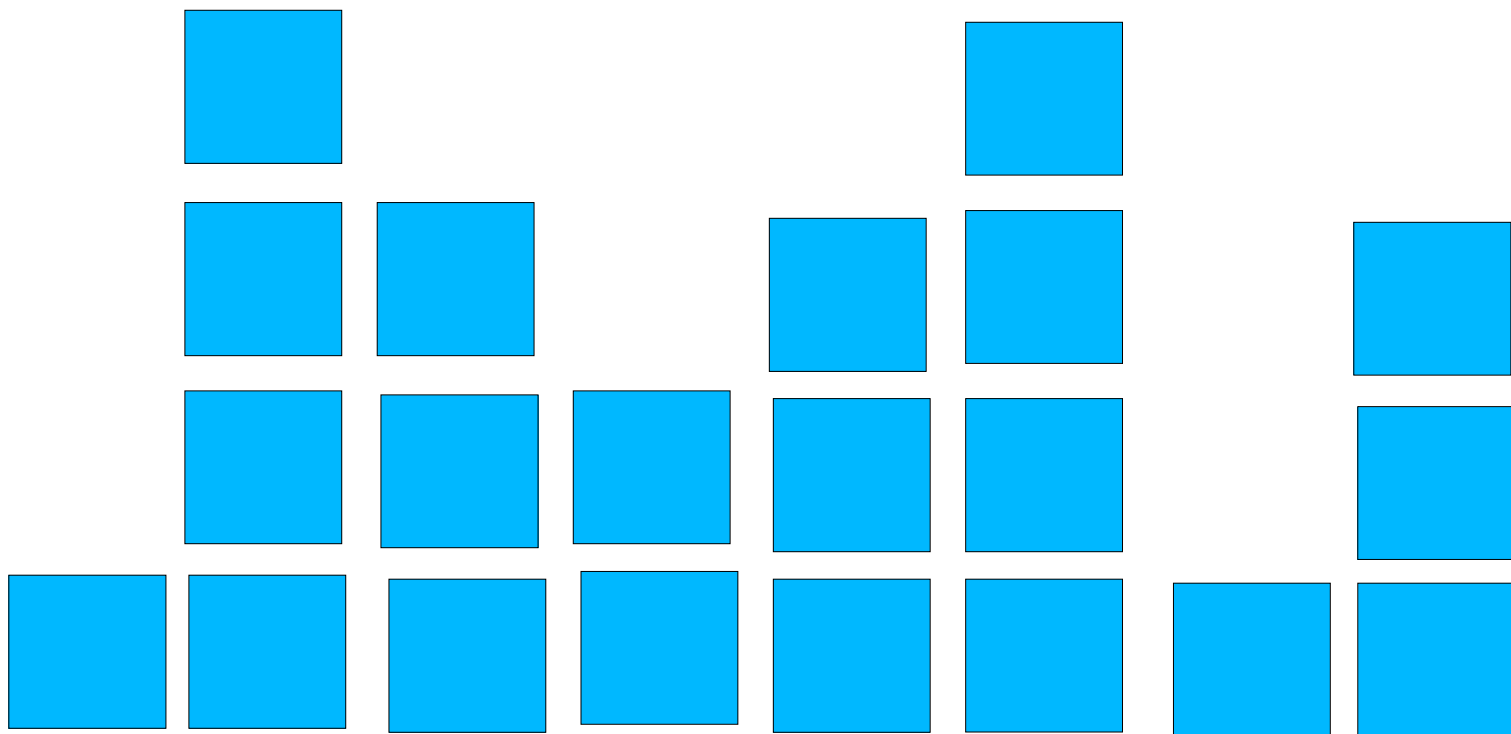
Sketch of Proof of the Kepler Conjecture

- ◆ How to reduce from an infinite number of variables to a finite (~ 150) number of variables.
- ◆ Nonlinear optimization problems are in general NP-hard. What special structures permit a solution to this non-linear optimization problem?

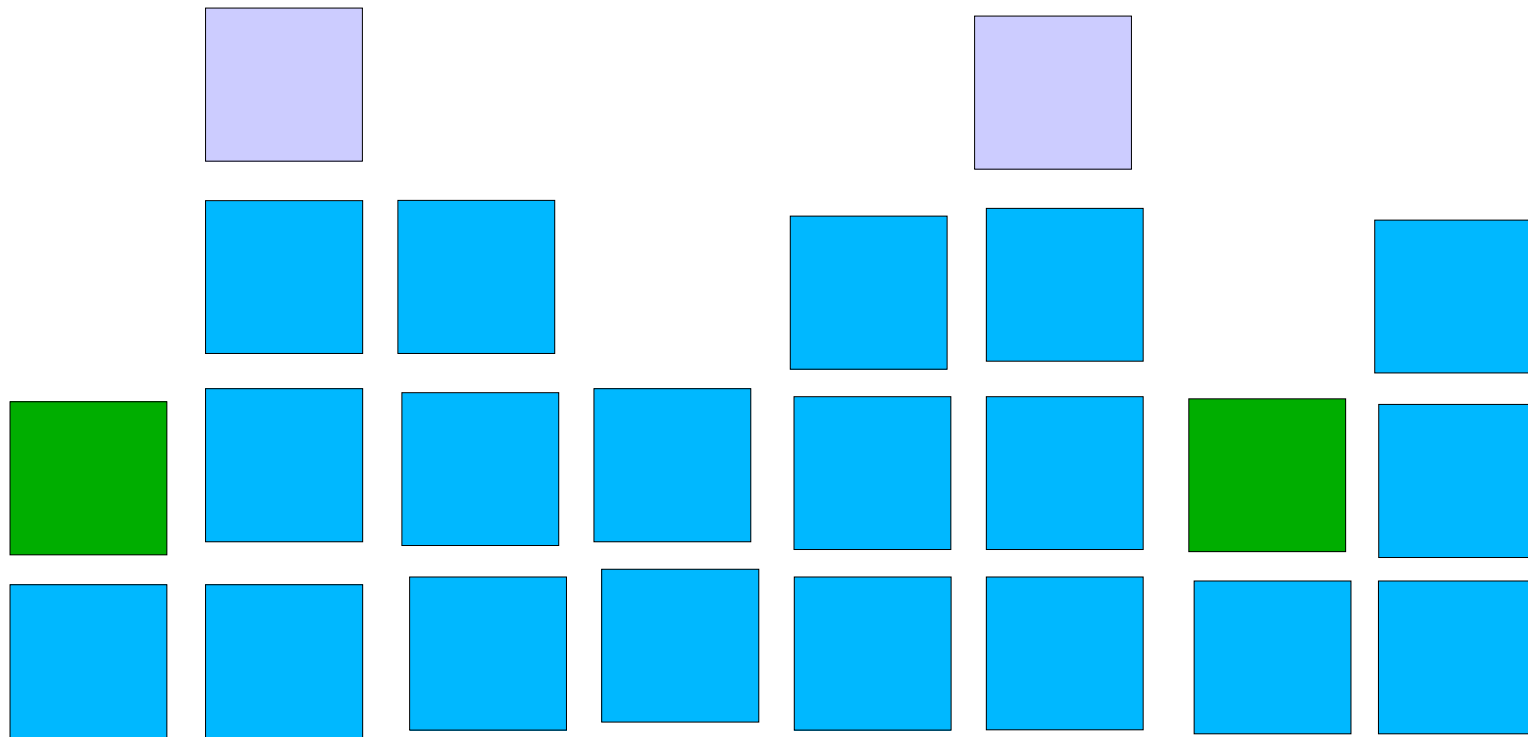
How to reduce to a finite number of variables

- ◆ A density is a limit of an average over larger and larger regions of space.
- ◆ How can infinite averages be reduced to finite averages?

An infinite brick wall ----



Reassigning the bricks in the wall



Analogy

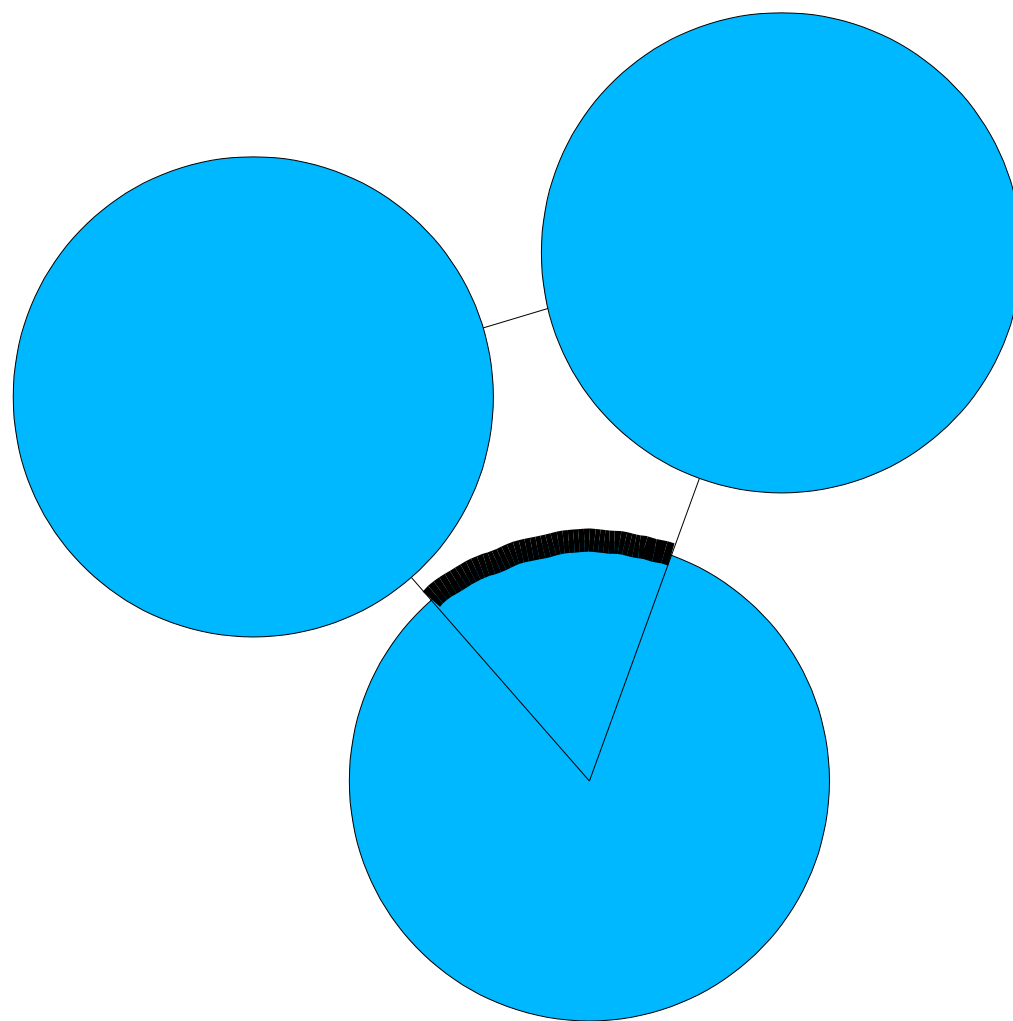
- ◆ Wall = Sphere Packing
- ◆ High parts of the wall = regions of high density
- ◆ Average ht of wall = asymptotic density
- ◆ Moving bricks about = changing geometric decomposition of space
- ◆ Highest point on the wall = local nonlinear optimization problem

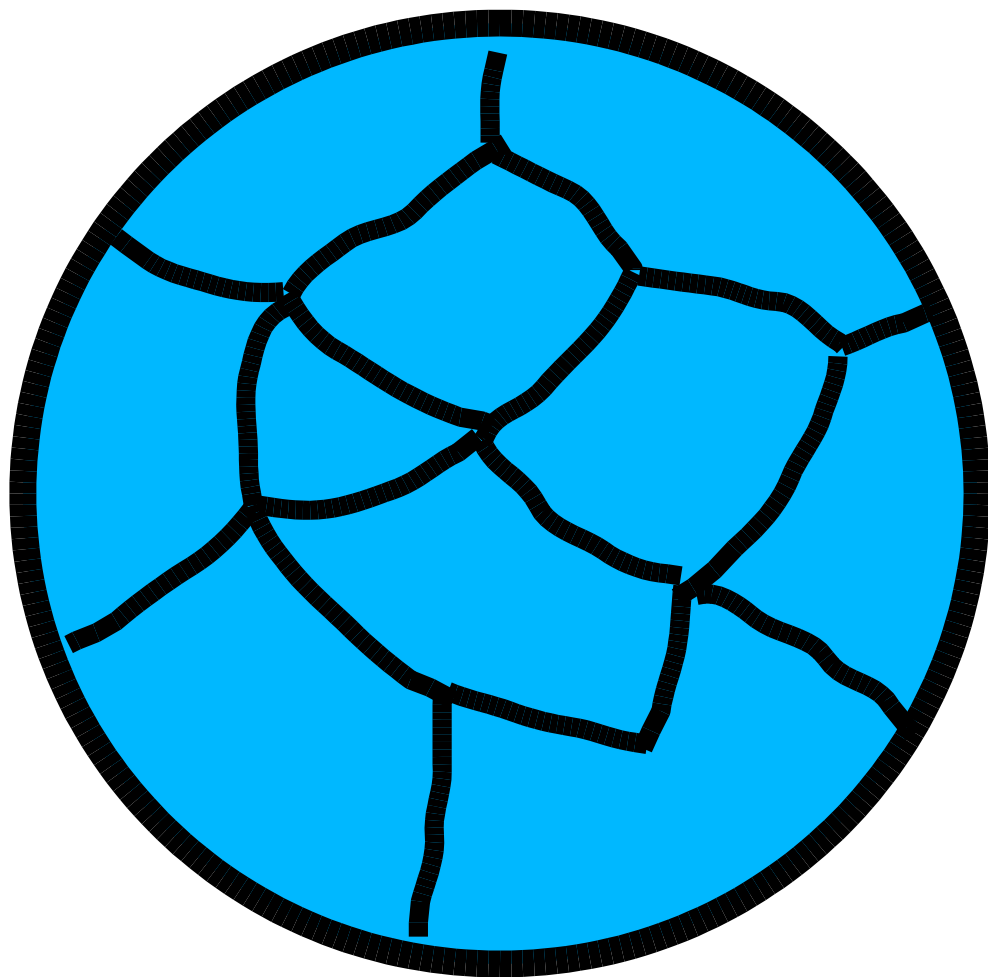
Nonlinear optimization

- ◆ The local nonlinear optimization problem is simplified considerably in two ways
 - ◆ A graph that encodes the local geometry.
 - ◆ Linear programming.

The graph associated to a local cluster of spheres.

- ◆ Let T be a cut-off parameter.
- ◆ Connect the center of every pair of spheres in the packing if the centers have distance at most T from one another.
- ◆ Fix one sphere center A , draw an arc on the sphere for every triangle (A,B,C) of centers that are connected by an edge.





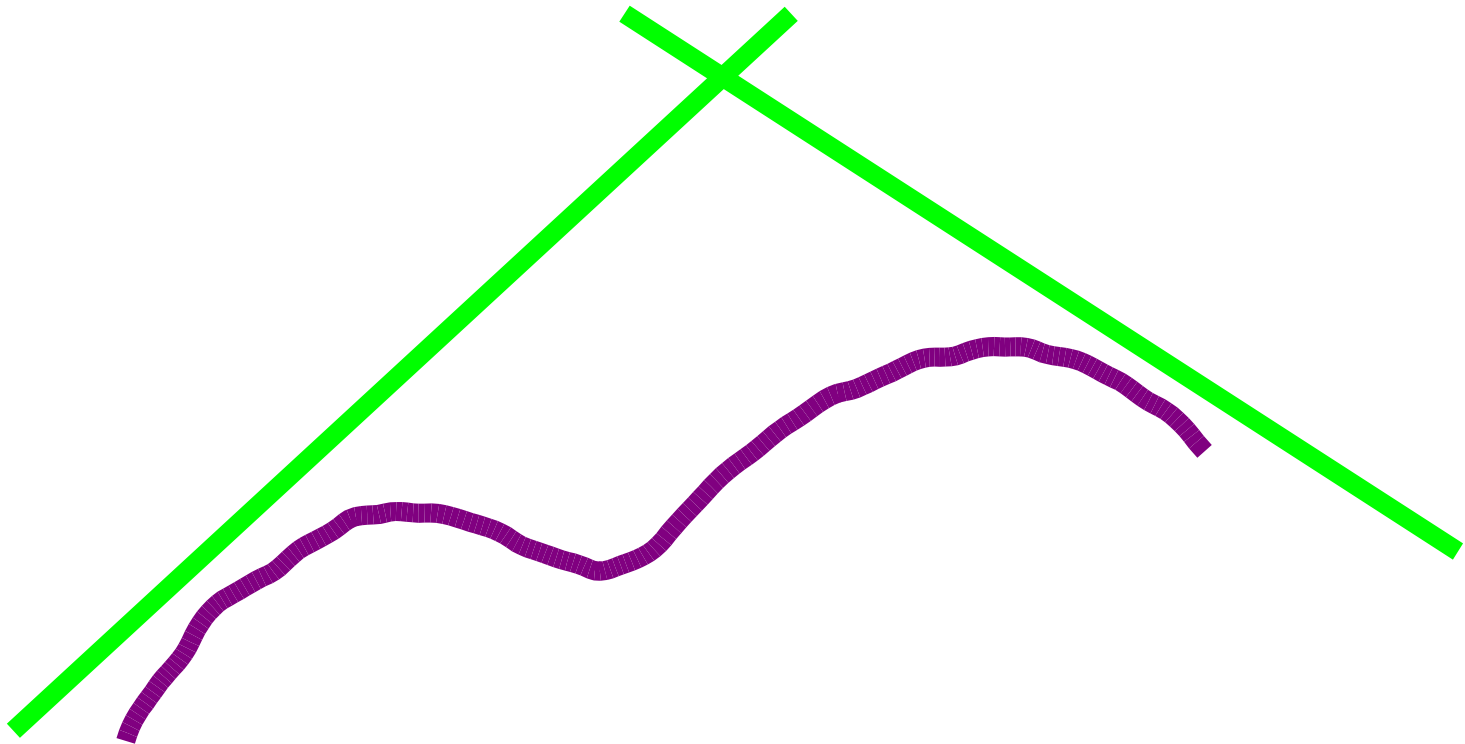
Graph Properties

- ◆ Each “face” of the planar graph gives a contribution to the nonlinear optimization problem.
- ◆ Translate properties of the nonlinear optimization problem into combinatorial properties of this planar graph.
- ◆ Classify (by computer) all planar graphs that are associated with potential counterexamples of the Kepler conjecture
- ◆ The computer yields about 5000 graphs.

Linear Programming

- ◆ Fix a planar graph and look at the nonlinear optimization problem over all local configurations attached to that planar graph.
- ◆ This tends to be a nonlinear optimization in about 30-50 variables.
- ◆ Solve by linear programming

Linear relaxation



Linear relaxation

- ◆ Each hyperplane appears in the proof in two guises:
 - ◆ Uninterpreted variables in linear programs
 - ◆ Interpreted variables in nonlinear inequalities
- ◆ Example:
 - ◆ $dih + vol + y3 < 7$
- ◆ To justify the use of an uninterpreted inequality as a hyperplane in a linear program, it is necessary to prove the interpreted nonlinear inequality.
- ◆ Use inequalities that involve a small number of variables that can be proved by computer.

FLYSPECK

- ◆ There are few prerequisites:
 - ◆ Jordan curve theorem
 - ◆ Multivariable calculus.
- ◆ The rest is in HOL-light (real analysis)
- ◆ The project should take about 20 work years.
- ◆ Fortunately, several others have become involved in the project.
 - ◆ Steven Obua – Linear Programming
 - ◆ Roland Zumkeller – Nonlinear inequalities

FLYSPECK

- ◆ The original proof is highly repetitive:
 - ◆ Similar estimates are applied to different cases.
 - ◆ Computers were avoided in many places even when they would have been helpful.
- ◆ My immediate goal is to produce a “higher order version” of the proof – a version that is both easier to understand and easier to implement in a formal system.

References

- ◆ Google: Kepler Hales
- ◆ Google: Flyspeck Hales
- ◆ <http://www.math.pitt.edu/~thales/>

Author: Steven Obua

Title: Proving Bounds for Real Linear Programs in Isabelle/HOL

Abstract:

Linear programming is a basic mathematical technique for optimizing a linear function on a domain that is constrained by linear inequalities. We restrict ourselves to linear programs on bounded domains that involve only real variables. In the context of theorem proving, this restriction makes it possible for any given linear program to obtain certificates from external linear programming tools that help to prove arbitrarily precise bounds for the given linear program. To this end, an explicit formalization of matrices in Isabelle/HOL is presented, and how the concept of lattice-ordered rings allows for a smooth integration of matrices with the axiomatic type classes of Isabelle.

As our work is a contribution to the Flyspeck project, we demonstrate that via reflection and with the above techniques it is now possible to prove bounds for the linear programs arising in the proof of the Kepler conjecture sufficiently fast.

Proving Bounds for Real Linear Programs in Isabelle/HOL

TYPES 2004, Jouy-en-Josas

Steven Obua

Technische Universität München

obua@in.tum.de



The Flyspeck Project

Initiated by Tom Hales in 2003.

Goal: *Complete* formalization and proof of the Kepler conjecture within a *mechanical theorem prover*.



The Flyspeck Project

Initiated by Tom Hales in 2003.

Goal: *Complete* formalization and proof of the Kepler conjecture within a *mechanical theorem prover*.

A subproblem: Prove upper bounds for about **100,000** real linear programs, each involving about **200** variables and **2000** constraints.



The General Linear Program (LP)

Given:



The General Linear Program (LP)

Given:

• a matrix $A \in \mathbb{R}^{m \times n}$



The General Linear Program (LP)

Given:

- a matrix $A \in \mathbb{R}^{m \times n}$
- a column vector $b \in \mathbb{R}^{m \times 1}$



The General Linear Program (LP)

Given:

- a matrix $A \in \mathbb{R}^{m \times n}$
- a column vector $b \in \mathbb{R}^{m \times 1}$
- a row vector $c \in \mathbb{R}^{1 \times n}$



The General Linear Program (LP)

Given:

- a matrix $A \in \mathbb{R}^{m \times n}$
- a column vector $b \in \mathbb{R}^{m \times 1}$
- a row vector $c \in \mathbb{R}^{1 \times n}$

Goal:

maximize $c * x$,

where the column vector $x \in \mathbb{R}^{n \times 1}$ is constrained by

$$A * x \leq b.$$



Our Approach

Usual method for solving LP's: *Simplex method*.



Our Approach

Usual method for solving LP's: *Simplex method*.

But our goal is:

- we don't want only an approximate maximum:
we also want it to be an ***upper bound***
- we need a ***proof within Isabelle/HOL*** that we actually
computed an upper bound



Our Approach

Usual method for solving LP's: *Simplex method*.

But our goal is:

- we don't want only an approximate maximum:
we also want it to be an ***upper bound***
- we need a ***proof within Isabelle/HOL*** that we actually
computed an upper bound

We want a theorem like

$$A * x \leq b \implies c * x \leq \text{bound}$$



Our Approach

Usual method for solving LP's: *Simplex method*.

But our goal is:

- we don't want only an approximate maximum:
we also want it to be an ***upper bound***
- we need a ***proof within Isabelle/HOL*** that we actually
computed an upper bound

We want a theorem like

$$A * x \leq b \implies c * x \leq \text{bound}$$

Proposal by Tom Hales:

Use the solution y of the dual linear program as a *certificate*.



The Dual Linear Program

Given:

- a matrix $A \in \mathbb{R}^{m \times n}$
- a column vector $b \in \mathbb{R}^{m \times 1}$
- a row vector $c \in \mathbb{R}^{1 \times n}$



The Dual Linear Program

Given:

- a matrix $A \in \mathbb{R}^{m \times n}$
- a column vector $b \in \mathbb{R}^{m \times 1}$
- a row vector $c \in \mathbb{R}^{1 \times n}$

Goal:

minimize $y * b$,

where the row vector $y \in \mathbb{R}^{1 \times m}$ is constrained by

$$y * A = c \wedge y \geq 0.$$



Duality Theorem

Given an LP:

- a *feasible solution* is a vector that fulfills the constraints of the LP
- the *feasible region* is the set of all feasible solutions of the LP



Duality Theorem

Given an LP:

- a *feasible solution* is a vector that fulfills the constraints of the LP
- the *feasible region* is the set of all feasible solutions of the LP

Observation: $y * b \geq y * (A * x) = (y * A) * x = c * x$
for all feasible solutions y of the dual LP and all feasible solutions x of the general LP



Duality Theorem

Given an LP:

- a *feasible solution* is a vector that fulfills the constraints of the LP
- the *feasible region* is the set of all feasible solutions of the LP

Observation: $y * b \geq y * (A * x) = (y * A) * x = c * x$
for all feasible solutions y of the dual LP and all feasible solutions x of the general LP

Duality Theorem: If both general and dual LP have feasible solutions, then

$$\min\{y * b \mid \dots\} = \max\{c * x \mid \dots\}$$



Steps in Proving a Bound for an LP

Prove a **generic theorem** based on duality that gives an upper bound for any LP that fulfills certain assumptions.



Steps in Proving a Bound for an LP

Prove a **generic theorem** based on duality that gives an upper bound for any LP that fulfills certain assumptions.
Then for a given LP:

1. get the LP data

- right now, the data is loaded from a file
- in the complete proof of the Kepler Conjecture, the data will arise from previous steps



Steps in Proving a Bound for an LP

Prove a **generic theorem** based on duality that gives an upper bound for any LP that fulfills certain assumptions.
Then for a given LP:

1. get the LP data
 - right now, the data is loaded from a file
 - in the complete proof of the Kepler Conjecture, the data will arise from previous steps
2. form the dual LP and calculate the certificate y
 - calculation can be done by any external solver
 - interface to GLPK, CPLEX implemented



Steps in Proving a Bound for an LP

Prove a **generic theorem** based on duality that gives an upper bound for any LP that fulfills certain assumptions.
Then for a given LP:

1. get the LP data
 - right now, the data is loaded from a file
 - in the complete proof of the Kepler Conjecture, the data will arise from previous steps
2. form the dual LP and calculate the certificate y
 - calculation can be done by any external solver
 - interface to GLPK, CPLEX implemented
3. instantiate the generic theorem with the LP data and the calculated certificate, then simplify



Generic Theorem for Upper Bound

Data A and c of general LP is known only ***approximately***:

$$A_1 \leq A \leq A_2, \quad c_1 \leq c \leq c_2$$



Generic Theorem for Upper Bound

Data A and c of general LP is known only ***approximately***:

$$A_1 \leq A \leq A_2, \quad c_1 \leq c \leq c_2$$

We can deal only with ***a priori bounded feasible regions***:

$$r_1 \leq x \leq r_2$$



Generic Theorem for Upper Bound

Data A and c of general LP is known only ***approximately***:

$$A_1 \leq A \leq A_2, \quad c_1 \leq c \leq c_2$$

We can deal only with ***a priori bounded feasible regions***:

$$r_1 \leq x \leq r_2$$

Variables that will be instantiated are shown in **red**:

$$\llbracket A * x \leq \textcolor{red}{b}; \textcolor{red}{A}_1 \leq A; A \leq \textcolor{red}{A}_2; \textcolor{red}{c}_1 \leq c; c \leq \textcolor{red}{c}_2; \textcolor{red}{r}_1 \leq x; x \leq \textcolor{red}{r}_2; 0 \leq \textcolor{red}{y} \rrbracket$$

$$\implies c * x$$

$$\leq \textcolor{red}{y} * \textcolor{red}{b} +$$

$$(\text{let } s_1 = \textcolor{red}{c}_1 - \textcolor{red}{y} * \textcolor{red}{A}_2; s_2 = \textcolor{red}{c}_2 - \textcolor{red}{y} * \textcolor{red}{A}_1$$

$$\text{in } s_2^+ * \textcolor{red}{r}_2^+ + s_1^+ * \textcolor{red}{r}_2^- + s_2^- * \textcolor{red}{r}_1^+ + s_1^- * \textcolor{red}{r}_1^-)$$



Before / After Simplification

Before: Variables that will be instantiated are shown in red:

$$\llbracket A * x \leq b; A_1 \leq A; A \leq A_2; c_1 \leq c; c \leq c_2; r_1 \leq x; x \leq r_2; 0 \leq y \rrbracket$$

$$\implies c * x$$

$$\leq y * b +$$

$$(let\ s_1 = c_1 - y * A_2; s_2 = c_2 - y * A_1$$

$$in\ s_2^+ * r_2^+ + s_1^+ * r_2^- + s_2^- * r_1^+ + s_1^- * r_1^-)$$

After: Simplified terms without variables are shown in blue:

$$\llbracket A * x \leq b; A_1 \leq A; A \leq A_2; c_1 \leq c; c \leq c_2; r_1 \leq x; x \leq r_2 \rrbracket$$

$$\implies c * x \leq bound$$



Further Details

- Finite Matrices
- Lattice-ordered Rings
- Axiomatic Type Classes
- Sparse Matrix Representation
- Representation of Real Numbers



Finite Matrices

`type 'a infmatrix = nat  $\Rightarrow$  nat  $\Rightarrow$  'a`



Finite Matrices

```
type 'a infmatrix = nat  $\Rightarrow$  nat  $\Rightarrow$  'a
```

```
typedef 'a matrix =
```

```
{A::'a infmatrix | finite {(r, c) | A r c  $\neq$  0}}
```



Finite Matrices

```
type 'a infmatrix = nat  $\Rightarrow$  nat  $\Rightarrow$  'a
```

```
typedef 'a matrix =
```

```
{A::'a infmatrix | finite {(r, c) | A r c  $\neq$  0}}
```



Operations on Finite Matrices

We can now define the usual operations on matrices:

- Matrix multiplication $A * B$
- Addition $A + B$
- Negation $-A$
- Subtraction $A - B$



Operations on Finite Matrices

We can now define the usual operations on matrices:

- Matrix multiplication $A * B$
- Addition $A + B$
- Negation $-A$
- Subtraction $A - B$

We also need:

- Componentwise comparison $A \leq B$
- Positive part A^+ , negative part A^-



Example: Addition of Finite Matrices



Example: Addition of Finite Matrices

$$\begin{pmatrix} 1 & 7 & 3 \\ 2 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 7 & 3 \\ 2 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$



Example: Addition of Finite Matrices

$$\begin{pmatrix} 1 & 7 & 3 \\ 2 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 7 & 3 \\ 2 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 2 \\ 10 \\ 5 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 10 & 0 & 0 \\ 5 & 0 & 0 \end{pmatrix}$$



Example: Addition of Finite Matrices

$$\begin{pmatrix} 1 & 7 & 3 \\ 2 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 7 & 3 \\ 2 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 2 \\ 10 \\ 5 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 10 & 0 & 0 \\ 5 & 0 & 0 \end{pmatrix}$$

$\Rightarrow$

$$\begin{pmatrix} 1 & 7 & 3 \\ 2 & 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 2 \\ 10 \\ 5 \end{pmatrix} = \begin{pmatrix} 1 & 7 & 3 \\ 4 & 1 & 0 \\ 10 & 0 & 0 \\ 5 & 0 & 0 \end{pmatrix}$$



Lattice-ordered Rings

In *Lattice theory* (G. Birkhoff, 1967):
the ring of $n \times n$ -matrices is a lattice-ordered ring
 $\Rightarrow$ this can be adapted to our finite matrices



Lattice-ordered Rings

In *Lattice theory* (G. Birkhoff, 1967):
the ring of $n \times n$ -matrices is a lattice-ordered ring
 $\Rightarrow$ this can be adapted to our finite matrices

Definition: A *lattice-ordered* ring (l-ring) is a structure, s.t.



Lattice-ordered Rings

In *Lattice theory* (G. Birkhoff, 1967):
the ring of $n \times n$ -matrices is a lattice-ordered ring
 $\Rightarrow$ this can be adapted to our finite matrices

Definition: A *lattice-ordered* ring (l-ring) is a structure, s.t.

- the structure is a ring



Lattice-ordered Rings

In *Lattice theory* (G. Birkhoff, 1967):
the ring of $n \times n$ -matrices is a lattice-ordered ring
 $\Rightarrow$ this can be adapted to our finite matrices

Definition: A *lattice-ordered* ring (l-ring) is a structure, s.t.

- the structure is a ring
- the structure is a lattice



Lattice-ordered Rings

In *Lattice theory* (G. Birkhoff, 1967):
the ring of $n \times n$ -matrices is a lattice-ordered ring
 $\Rightarrow$ this can be adapted to our finite matrices

Definition: A *lattice-ordered* ring (l-ring) is a structure, s.t.

- the structure is a ring
- the structure is a lattice
- the partial order is compatible with addition and multiplication:

- $a \leq b \rightarrow c + a \leq c + b$

- $a \leq b \wedge 0 \leq c \rightarrow c * a \leq c * b$

- $a \leq b \wedge 0 \leq c \rightarrow a * c \leq b * c$



Utilizing Axiomatic Type Classes

Axiomatic type classes are Isabelle's built-in mechanism for abstracting the properties of a type



Utilizing Axiomatic Type Classes

Axiomatic type classes are Isabelle's built-in mechanism for abstracting the properties of a type

- a special axiomatic type class models the property of a type to form a lattice-ordered ring



Utilizing Axiomatic Type Classes

Axiomatic type classes are Isabelle's built-in mechanism for abstracting the properties of a type

- a special axiomatic type class models the property of a type to form a lattice-ordered ring
- lattice-ordered rings have been integrated with Isabelle's numeric type classes:
types `int` and `real` are lattice-ordered rings



Utilizing Axiomatic Type Classes

Axiomatic type classes are Isabelle's built-in mechanism for abstracting the properties of a type

- a special axiomatic type class models the property of a type to form a lattice-ordered ring
- lattice-ordered rings have been integrated with Isabelle's numeric type classes:
types `int` and `real` are lattice-ordered rings
- if α is a lattice-ordered ring, then so is α `matrix`



Utilizing Axiomatic Type Classes

Axiomatic type classes are Isabelle's built-in mechanism for abstracting the properties of a type

- a special axiomatic type class models the property of a type to form a lattice-ordered ring
- lattice-ordered rings have been integrated with Isabelle's numeric type classes:
types `int` and `real` are lattice-ordered rings
- if α is a lattice-ordered ring, then so is α `matrix`
 $\Rightarrow$ many already existing theorems and tactics are reused



Bounds for Products in L-Rings

Given elements a and b of an l-ring:

$$\begin{array}{ccccc} a_1 & \leq & a & \leq & a_2 \\ b_1 & \leq & b & \leq & b_2 \end{array}$$

Bounds for $a * b$?



Bounds for Products in L-Rings

Given elements a and b of an l-ring:

$$\begin{array}{ccccc} a_1 & \leq & a & \leq & a_2 \\ b_1 & \leq & b & \leq & b_2 \end{array}$$

Bounds for $a * b$?

Use positive and negative part:

$$\begin{array}{rclcl} x^+ & = & \text{join } x \ 0 & \geq & 0 \\ x^- & = & \text{meet } x \ 0 & \leq & 0 \\ \Rightarrow \quad x & = & x^+ + x^- & & \end{array}$$



Bounds for Products in L-Rings

Given elements a and b of an l-ring:

$$\begin{array}{ccccc} a_1 & \leq & a & \leq & a_2 \\ b_1 & \leq & b & \leq & b_2 \end{array}$$

Bounds for $a * b$?

Use positive and negative part:

$$\begin{array}{rclcl} x^+ & = & \text{join } x \ 0 & \geq & 0 \\ x^- & = & \text{meet } x \ 0 & \leq & 0 \\ \Rightarrow \quad x & = & x^+ + x^- & & \end{array}$$

Upper Bound:

$$a * b$$



Bounds for Products in L-Rings

Given elements a and b of an l-ring:

$$\begin{array}{ccccc} a_1 & \leq & a & \leq & a_2 \\ b_1 & \leq & b & \leq & b_2 \end{array}$$

Bounds for $a * b$?

Use positive and negative part:

$$\begin{array}{rclcl} x^+ & = & \text{join } x \ 0 & \geq & 0 \\ x^- & = & \text{meet } x \ 0 & \leq & 0 \\ \Rightarrow \quad x & = & x^+ + x^- & & \end{array}$$

Upper Bound:

$$a * b = (a^+ + a^-) * (b^+ + b^-)$$



Bounds for Products in L-Rings

Given elements a and b of an l-ring:

$$\begin{array}{ccccc} a_1 & \leq & a & \leq & a_2 \\ b_1 & \leq & b & \leq & b_2 \end{array}$$

Bounds for $a * b$?

Use positive and negative part:

$$\begin{array}{rcll} x^+ & = & \text{join } x \ 0 & \geq \ 0 \\ x^- & = & \text{meet } x \ 0 & \leq \ 0 \\ \Rightarrow \ x & = & x^+ + x^- & \end{array}$$

Upper Bound:

$$\begin{aligned} a * b &= (a^+ + a^-) * (b^+ + b^-) \\ &= a^+ * b^+ + a^+ * b^- + a^- * b^+ + a^- * b^- \end{aligned}$$



Bounds for Products in L-Rings

Given elements a and b of an l-ring:

$$\begin{array}{ccccc} a_1 & \leq & a & \leq & a_2 \\ b_1 & \leq & b & \leq & b_2 \end{array}$$

Bounds for $a * b$?

Use positive and negative part:

$$\begin{array}{rcll} x^+ & = & \text{join } x \ 0 & \geq \ 0 \\ x^- & = & \text{meet } x \ 0 & \leq \ 0 \\ \Rightarrow \ x & = & x^+ + x^- & \end{array}$$

Upper Bound:

$$\begin{aligned} a * b &= (a^+ + a^-) * (b^+ + b^-) \\ &= a^+ * b^+ + a^+ * b^- + a^- * b^+ + a^- * b^- \\ &\leq a_2^+ * b_2^+ + a_1^+ * b_2^- + a_2^- * b_1^+ + a_1^- * b_1^- \end{aligned}$$



Sparse Matrix Representation

We need to actually calculate with matrices
 $\Rightarrow$ *sparse* representation of matrices as list of lists:

`type 'a spvec = (nat * 'a) list`

`type 'a spmat = (nat * 'a spvec) list`



Sparse Matrix Representation

We need to actually calculate with matrices
 $\Rightarrow$ *sparse* representation of matrices as list of lists:

type 'a spvec = (nat * 'a) list

type 'a spmat = (nat * 'a spvec) list

Example:

$[(1, [(1, a), (3, b)]), (2, [(0, c), (1, d)])]$



Sparse Matrix Representation

We need to actually calculate with matrices
 $\Rightarrow$ *sparse* representation of matrices as list of lists:

type 'a spvec = (nat * 'a) list

type 'a spmat = (nat * 'a spvec) list

Example:

$[(1, [(1, a), (3, b)]), (2, [(0, c), (1, d)])]$

sparse-row-matrix $\rightarrow$



Sparse Matrix Representation

We need to actually calculate with matrices
 $\Rightarrow$ *sparse* representation of matrices as list of lists:

type 'a spvec = (nat * 'a) list

type 'a spmat = (nat * 'a spvec) list

Example:

$[(1, [(1, a), (3, b)]), (2, [(0, c), (1, d)])]$

sparse-row-matrix $\rightarrow$

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & a & 0 & b \\ c & d & 0 & 0 \end{pmatrix}$$



Representation of Real Numbers

constdefs

$\text{pow2} :: \text{int} \Rightarrow \text{real}$

$\text{pow2 } z \equiv \text{if } z \geq 0 \text{ then } 2^{\text{nat } z} \text{ else inverse } 2^{\text{nat } (-z)}$

$\text{float} :: \text{int} * \text{int} \Rightarrow \text{real}$

$\text{float } (m, e) \equiv (\text{real } m) * (\text{pow2 } e)$



Representation of Real Numbers

constdefs

$\text{pow2} :: \text{int} \Rightarrow \text{real}$

$\text{pow2 } z \equiv \text{if } z \geq 0 \text{ then } 2^{\text{nat } z} \text{ else inverse } 2^{\text{nat } (-z)}$

$\text{float} :: \text{int} * \text{int} \Rightarrow \text{real}$

$\text{float } (m, e) \equiv (\text{real } m) * (\text{pow2 } e)$

Advantages of this representation:

- simple rules for calculation with floats



Representation of Real Numbers

constdefs

$\text{pow2} :: \text{int} \Rightarrow \text{real}$

$\text{pow2 } z \equiv \text{if } z \geq 0 \text{ then } 2^{\text{nat } z} \text{ else inverse } 2^{\text{nat } (-z)}$

$\text{float} :: \text{int} * \text{int} \Rightarrow \text{real}$

$\text{float } (m, e) \equiv (\text{real } m) * (\text{pow2 } e)$

Advantages of this representation:

- simple rules for calculation with floats
- given $r \in \mathbb{R}$, and $\delta > 0$, there are m_1, m_2, e_1, e_2 such that

$$r - \delta \leq \text{float}(m_1, e_1) \leq r \leq \text{float}(m_2, e_2) \leq r + \delta$$



Performance Issues

Three options for simplifying:

- Isabelle's simplifier
- Theorem generating code generator (speedup of factor 3.5)
- Reflection via rewriting oracle (further speedup of factor ≈ 100)



Performance Issues

Three options for simplifying:

- Isabelle's simplifier
- Theorem generating code generator (speedup of factor 3.5)
- Reflection via rewriting oracle (further speedup of factor ≈ 100)

Using reflection, a bound for a typical Flyspeck-LP is obtained on a 3Ghz Pentium IV in about

8 seconds

$\Rightarrow$ the bounds for all LP's should be provable in about

10 days



Future Work / Next Step

Bring together LP's with the work (in progress) of

- Gertrud Bauer: Graph Generation
- Roland Zumkeller: Nonlinear Inequalities



Thanks for listening!!



Author: Roland Zumkeller

Title: Verifying inequalities over real numbers in Coq

Abstract:

In 1611 Johannes Kepler stated that the maximal density of a packing of spheres having the same radius is $\pi/\sqrt{18}$. This value is in fact achieved by arranging the spheres in a rather intuitive way, as usually done by fruit sellers. Proving this conjecture remained a challenge for several centuries - some "proofs" were given, but they turned out to be erroneous. In 1998 Thomas C. Hales published a proof, which is not only very large, but makes also extensive use of computer programs (similarly to the proof of the four colour theorem). The complexity of these calculations leaves some uncertainty about their correctness, which has lead Hales himself to call for a formalization of his proof. My talk will focus on one type of problem which reoccurs throughout the proof: the formal verification of inequalities in real numbers. A global optimization method used in conjunction with a branch-and-bound algorithm is presented together with its correctness proofs in Coq.

Verifying inequalities over real numbers in Coq

TYPES 2004, Paris

17 December 2004

Roland Zumkeller

supervised by Benjamin Werner

LogiCal, INRIA Futurs

Motivation

- Inequalities over real numbers occur throughout Prof. Hales' proof of the Kepler conjecture.
- Example: linear relaxation (Steven) requires us to proof bounds on non-linear functions.

$x_1, x_2, x_5, x_6 \in [4 ; 6.3001]$ $x_3 \in [6.3001 ; 8]$
 $x_4 \in [6.3001 ; 6.3001]$ (sic)

$$\begin{aligned} & \pi/2 + \arctan(\\ & \quad -(-x_2*x_3 - x_1*x_4 + x_2* x_5 + \\ & \quad \quad x_3*x_6 - x_5*x_6 + x_1* (- x_1 + x_2 + x_3 - x_4 + x_5 + x_6)) \\ & / \sqrt{4*x_1*(x_1*x_4*(- x_1 + x_2 + x_3 - x_4 + x_5 + x_6) + \\ & \quad x_2*x_5*(x_1 - x_2 + x_3 + x_4 - x_5 + x_6) + \\ & \quad x_3*x_6*(x_1 + x_2 - x_3 + x_4 + x_5 - x_6) \\ & \quad - x_2*x_3*x_4 - x_1*x_3*x_5 - x_1*x_2*x_6 - x_4*x_5*x_6))) > 1.033 \end{aligned}$$

- The primitive status of programs in Coq allows to integrate reasoning and computation efficiently.

Interval Arithmetic (1/2)

- The set of intervals is defined as $\mathbb{I} = (\mathbb{R} \times \mathbb{R}) \cup \{\perp\}$
- $\hat{f} : \mathbb{I}^n \rightarrow \mathbb{I}$ is *correct* for $f : \mathbb{R}^n \rightarrow \mathbb{R}$ iff:

$$\forall \vec{r} \in \mathbb{R}^n, \vec{i} \in \mathbb{I}^n. (\forall j < n. \vec{r}_j \in \vec{i}_j) \rightarrow f \vec{r} \in \hat{f} \vec{i}$$
- Expressions are modelled by an inductive type (signature).
- Two interpretations for each function symbol – a Coq module type with two implementations, and a functor connecting them:

	$\mathcal{R}$	$\mathcal{I}$	$\sim$ (compatibility)
num	$\mathbb{R}$	$(\mathbb{R} \times \mathbb{R}) \cup \{\perp\}$	$\in$
prop	Prop	$\{\text{T}, \text{F}, ?, \perp\}$	$\forall P. P \rightarrow P \sim \text{T}$ $\forall P. \neg P \rightarrow P \sim \text{F}$ $\forall P. P \sim ?$
$+$: num $\rightarrow$ num $\rightarrow$ num	$+$	$\hat{+}$	
$<$: num $\rightarrow$ num $\rightarrow$ prop	$<$	$\hat{<}$	

Interval Arithmetic (2/2)

- Examples:

- $$- [a, b] \hat{+} [c, d] = [a + c, b + d] \text{ and } \perp \hat{+} \dots = \dots \hat{+} \perp = \perp$$

- $$- \frac{1}{[a, b]} = \begin{cases} (\frac{1}{b}, \frac{1}{a}) & \text{if } a > 0 \vee b < 0 \\ \perp & \text{if } a \leq 0 \leq b \end{cases} \text{ and } \frac{1}{\perp} = \perp$$

- $$- [a, b] \hat{<} [c, d] = \begin{cases} T & \text{if } b < c \\ F & \text{if } d < a \\ ? & \text{otherwise} \end{cases} \text{ and } \perp \hat{<} \dots = \dots \hat{<} \perp = \perp$$

- Applying correctness: $x \in [1, 3] \wedge y \in [4, 5] \wedge z \in [2, 4]$ implies

- $$- x + y \in [1, 3] \hat{+} [4, 5] = [5, 8]$$

- $$- x < y \text{ and } \neg(y < x), \text{ since } 3 < 4$$

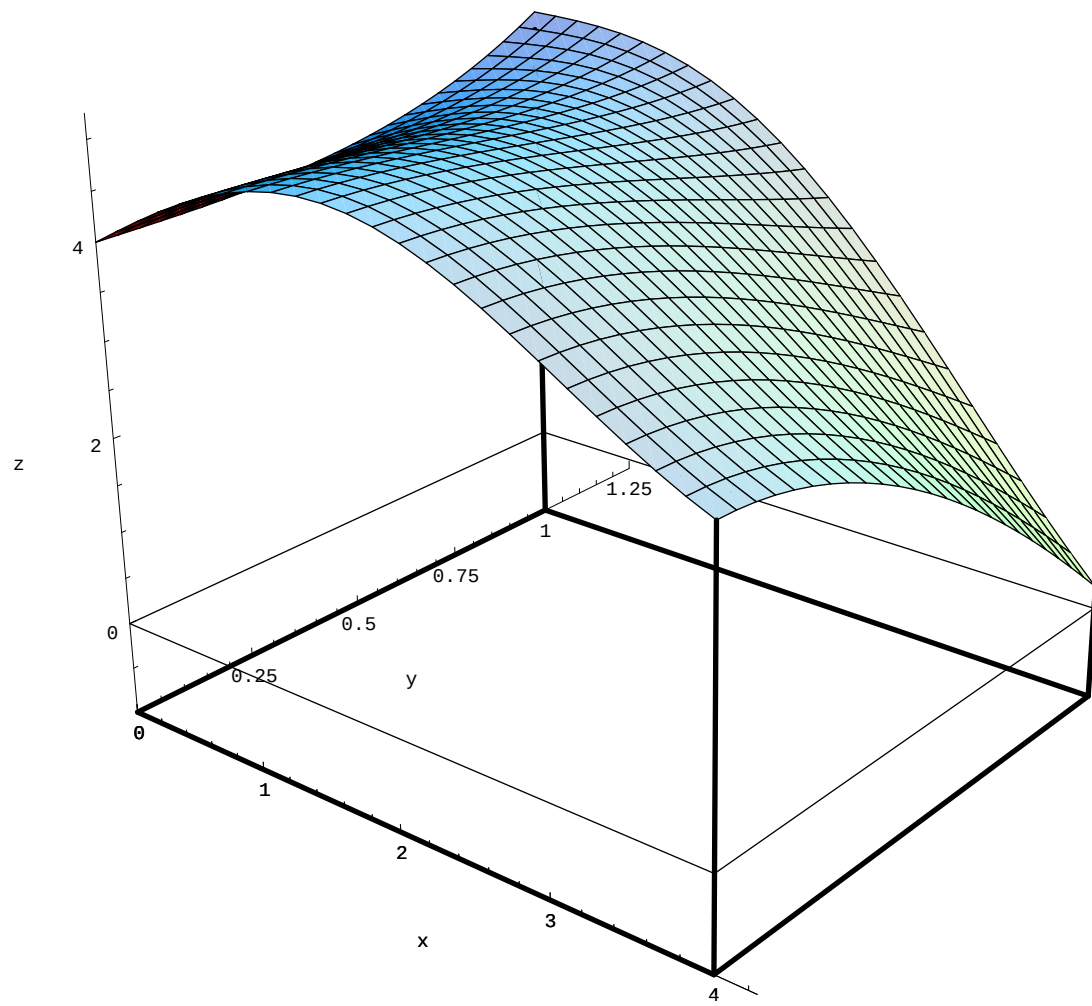
- $$- \dots \text{but it does neither imply } x < z, \text{ nor } z < x$$

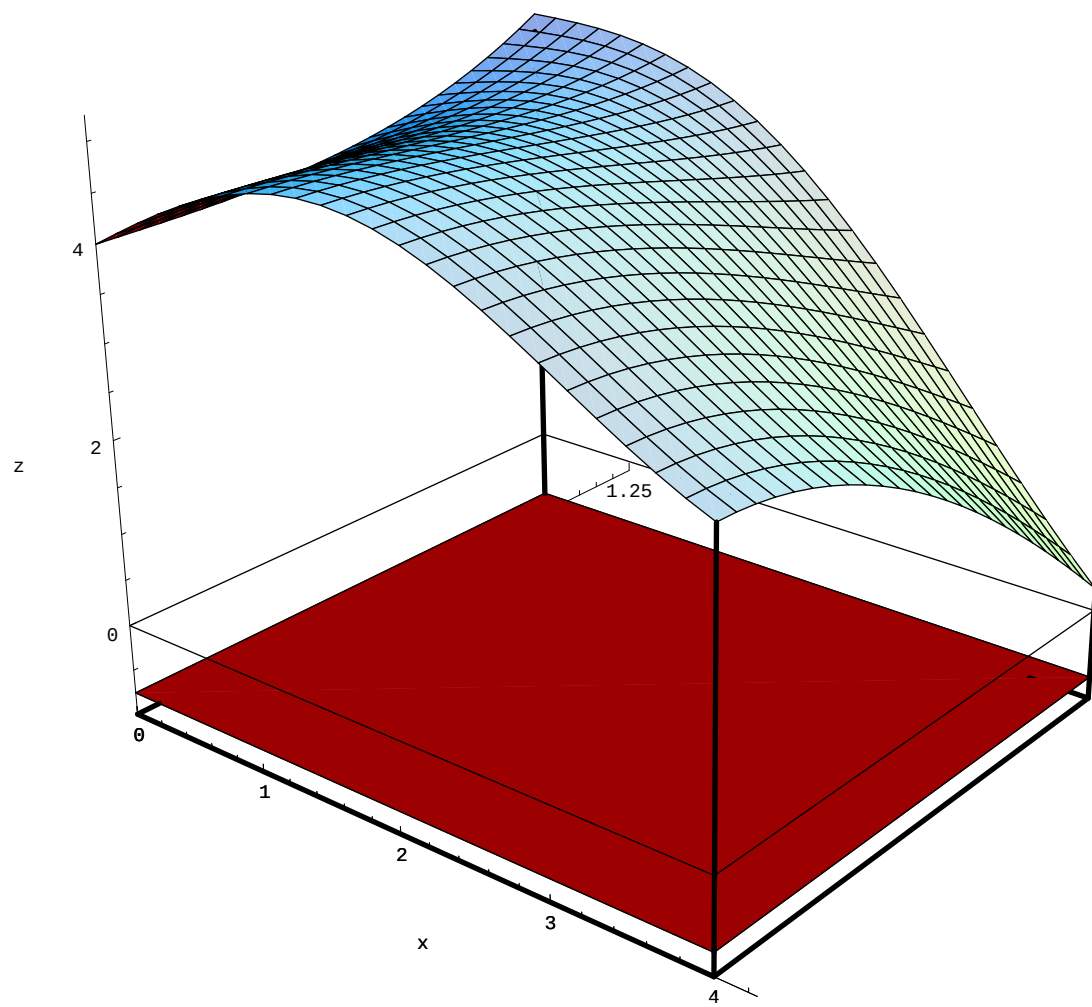
The Dependence Problem

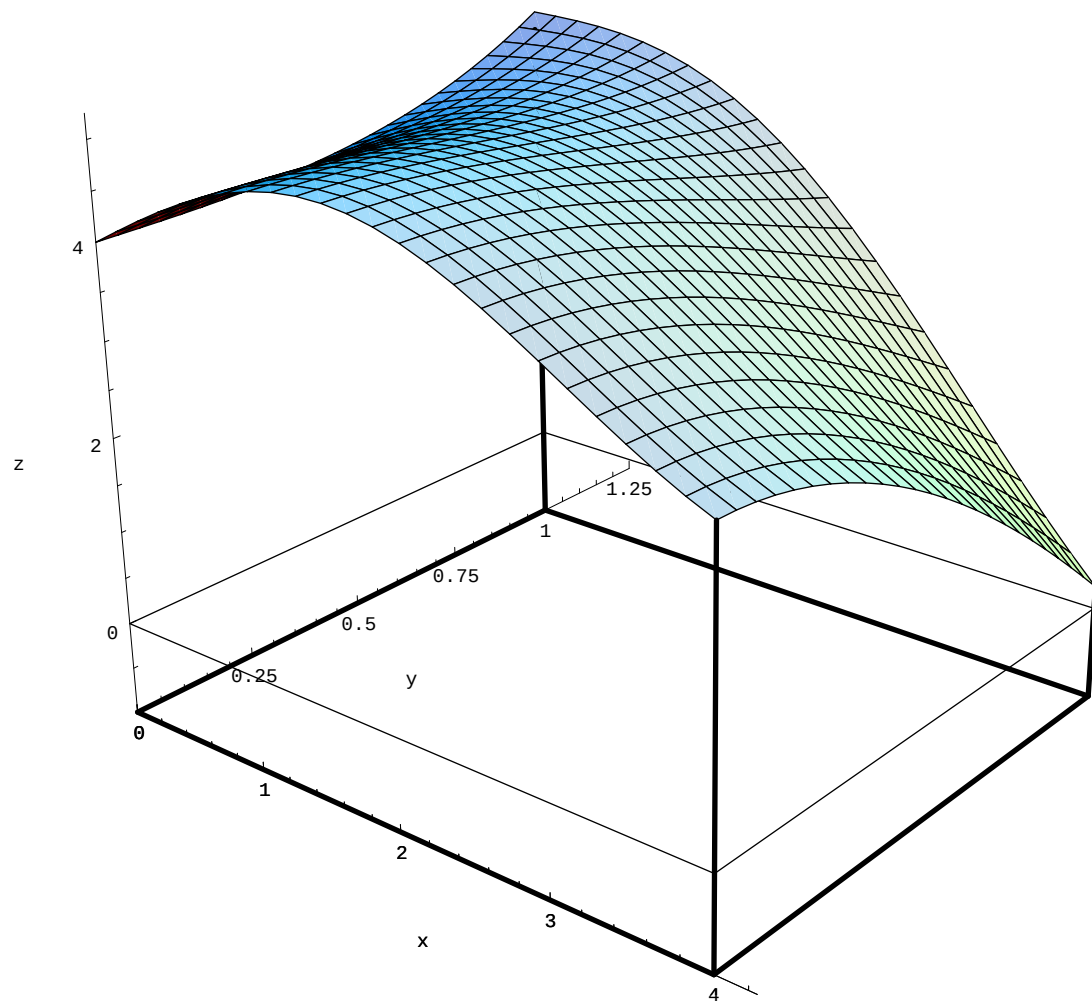
- $[a, b] \hat{-} [c, d] = [a - d, b - c]$
- Interval arithmetic fails to take into account variable dependencies:
for $x, y \in [2; 5]$ it yields the same result on $x - y$ and $x - x$, namely $[-3; 3]$
- ... but: $x - x = 0$
- Term rewriting can improve the result, but doesn't solve the problem in general (e.g. take $x - \sin x$).
- “Solution”: break up the interval into pieces, evaluate each term, take the union of the results.
$$([2; 3] \hat{-} [2; 3]) \cup ([3; 5] \hat{-} [3; 5]) = [-1; 1] \cup [-2; 2] = [-2; 2]$$

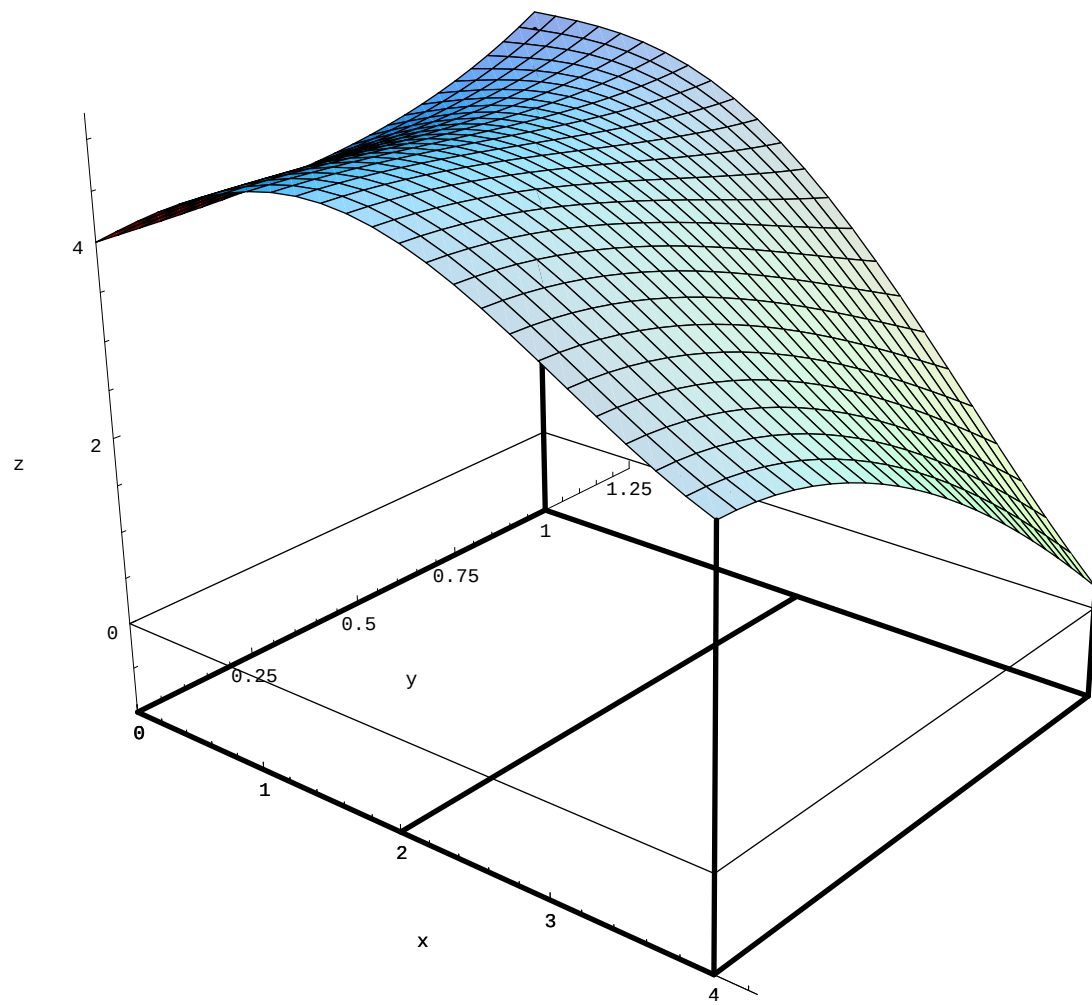
Branch & Bound

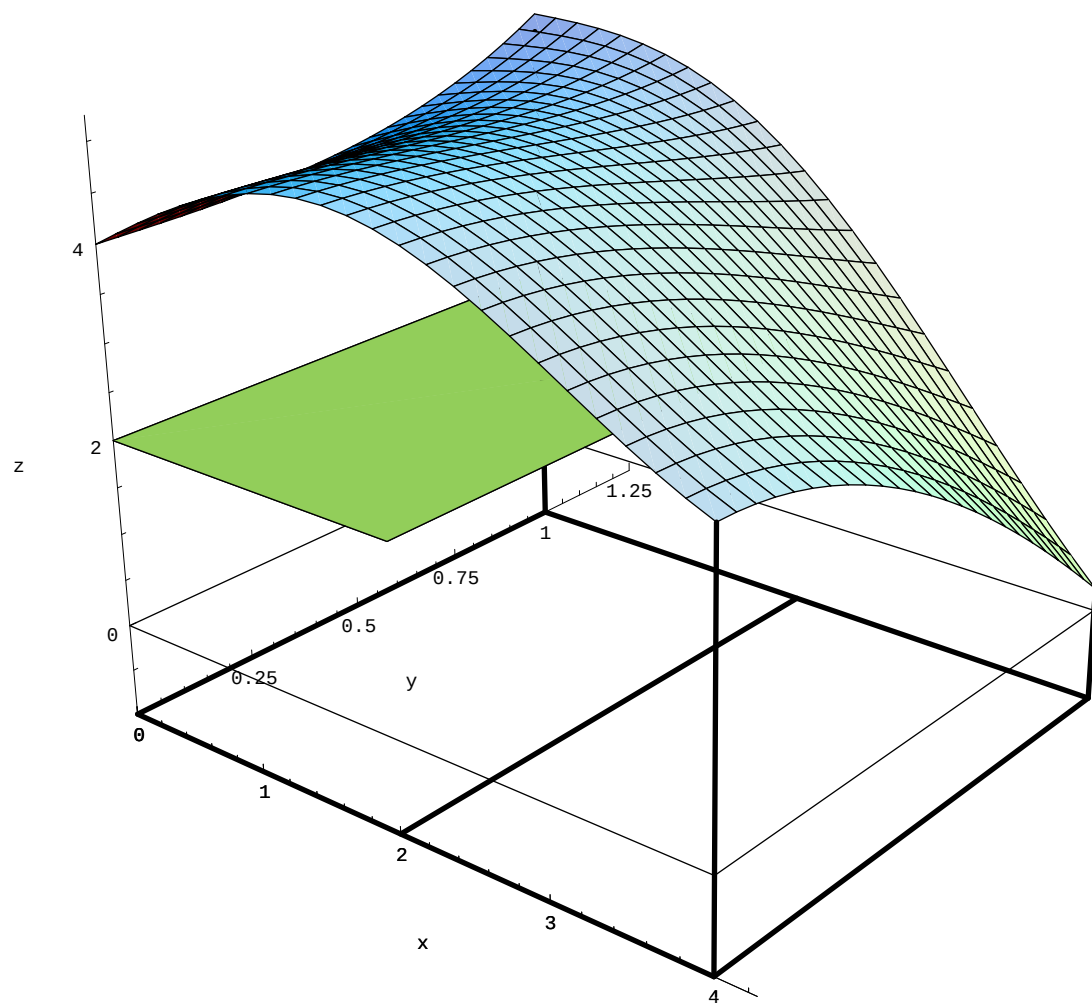
- Most of the inequalities contain 6 variables.
- Which variable's interval is to be cut? (choice: the largest)
- Evaluating formal partial derivatives can establish monotonicity in one direction. In this case we can reduce the domain to its border.
- Demo ...

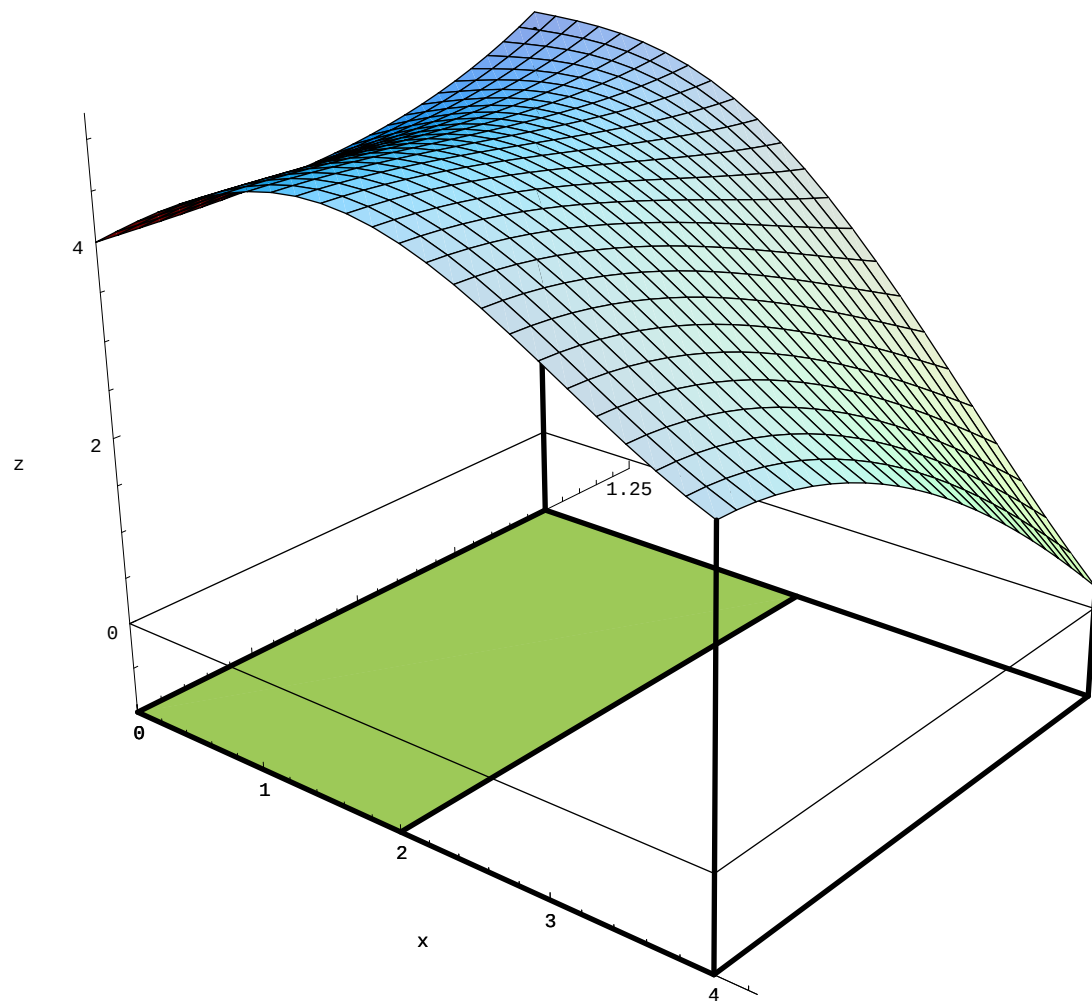


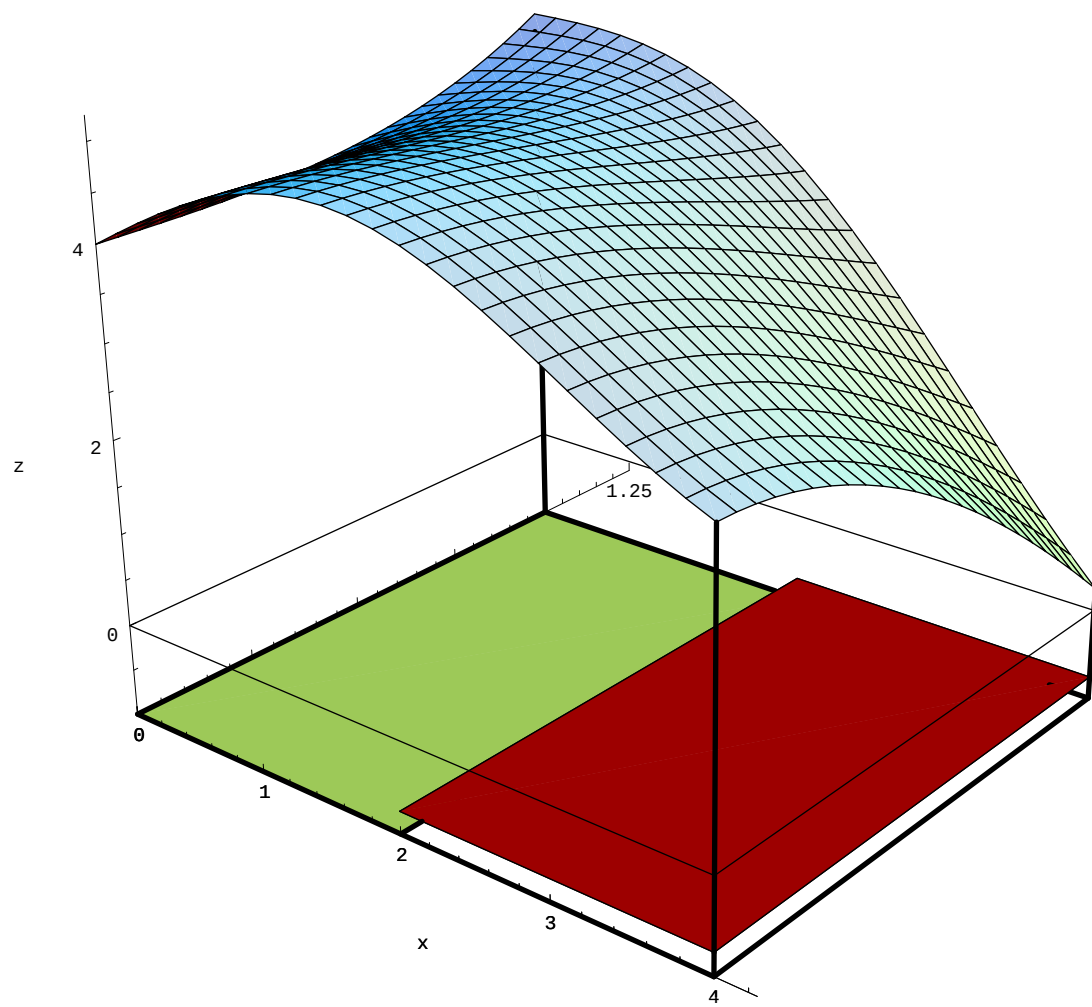


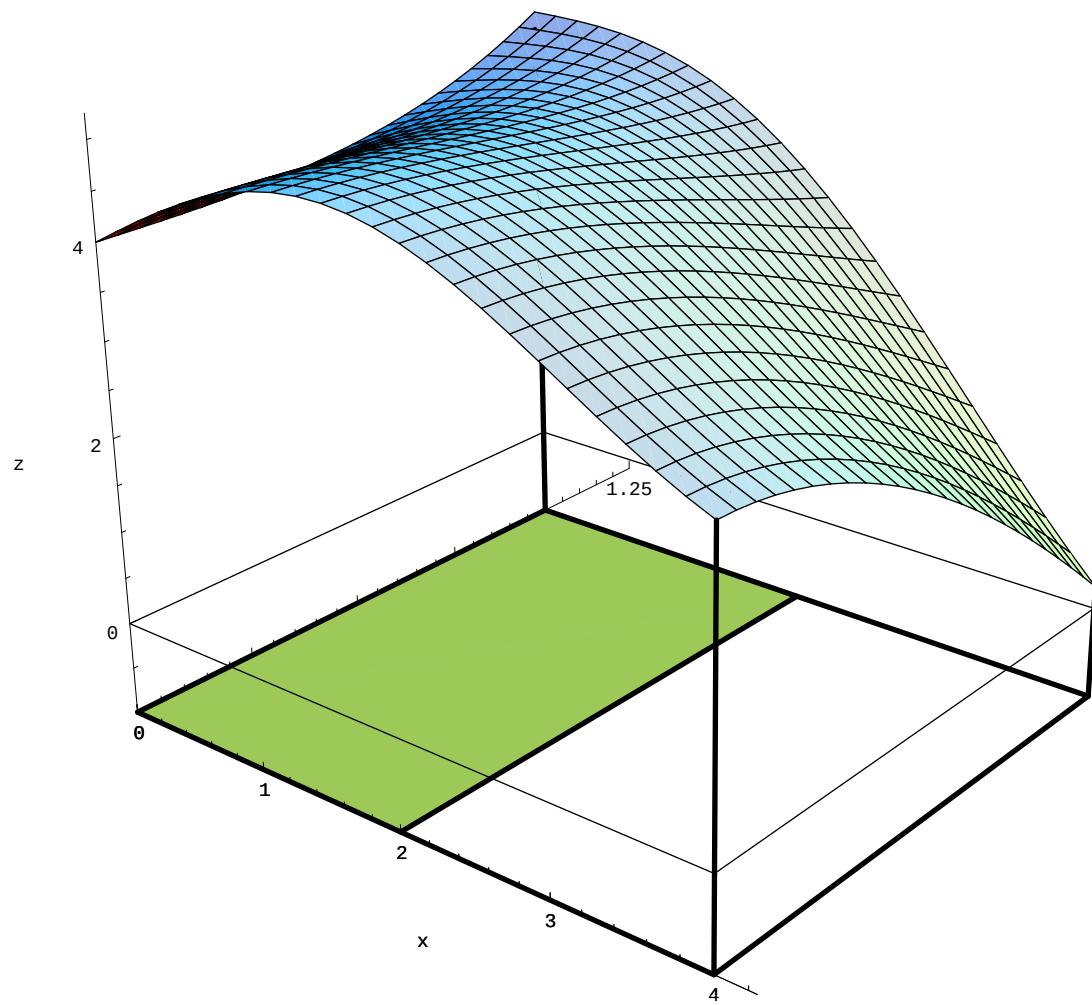


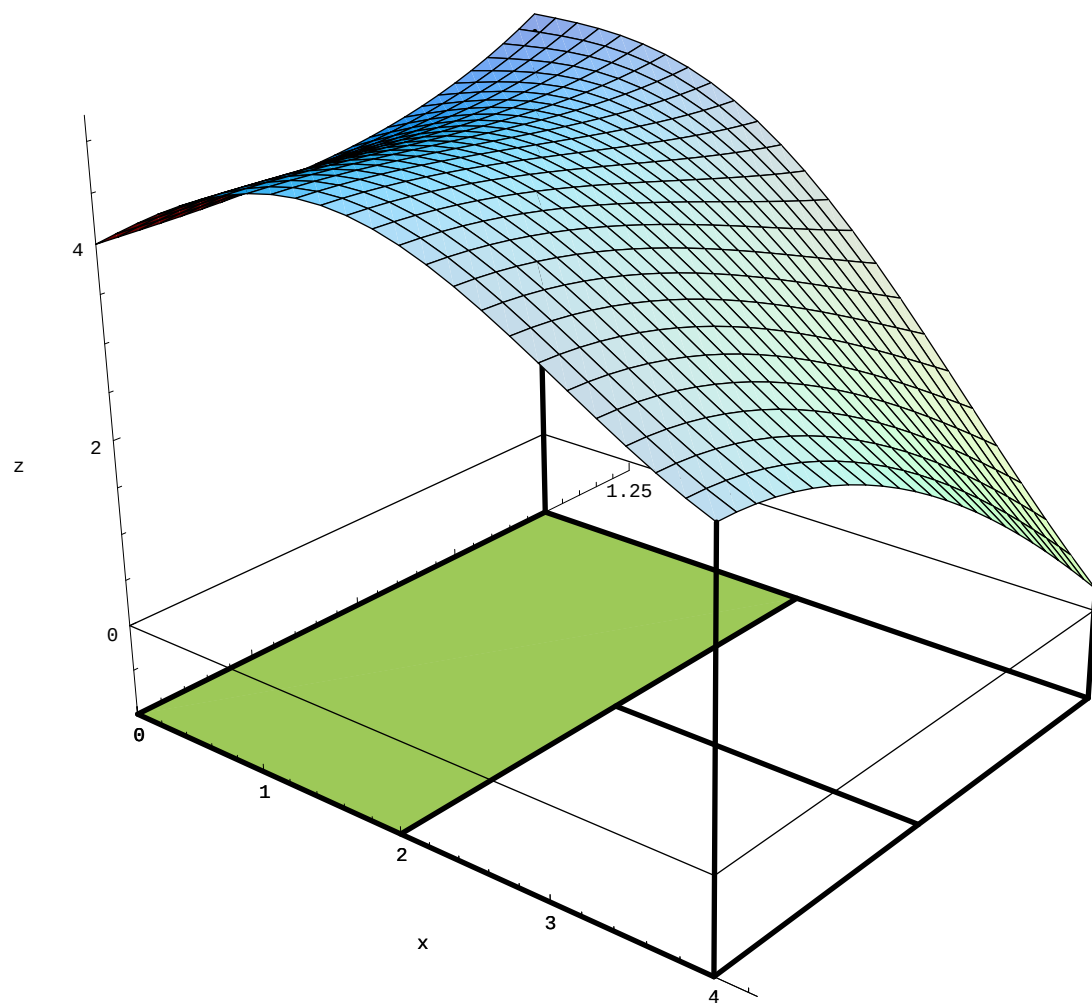


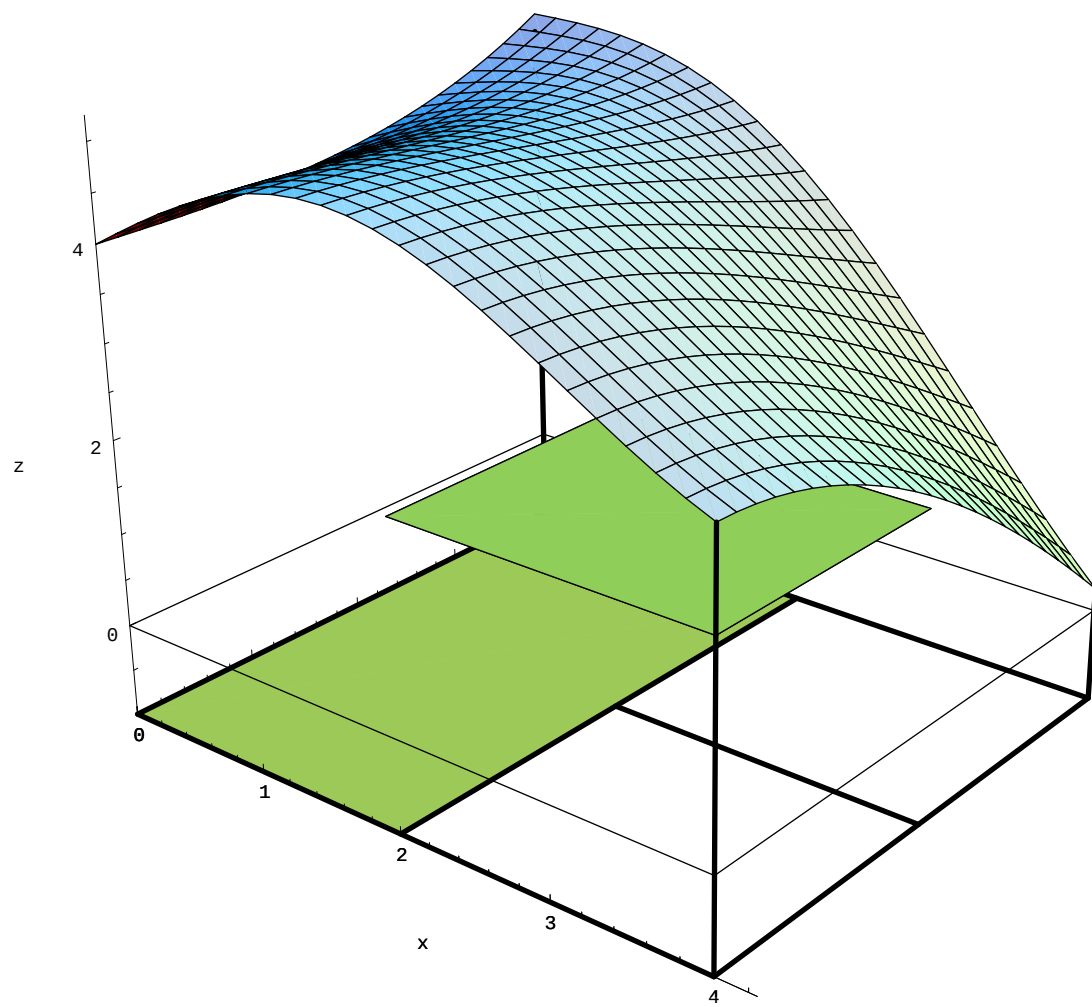


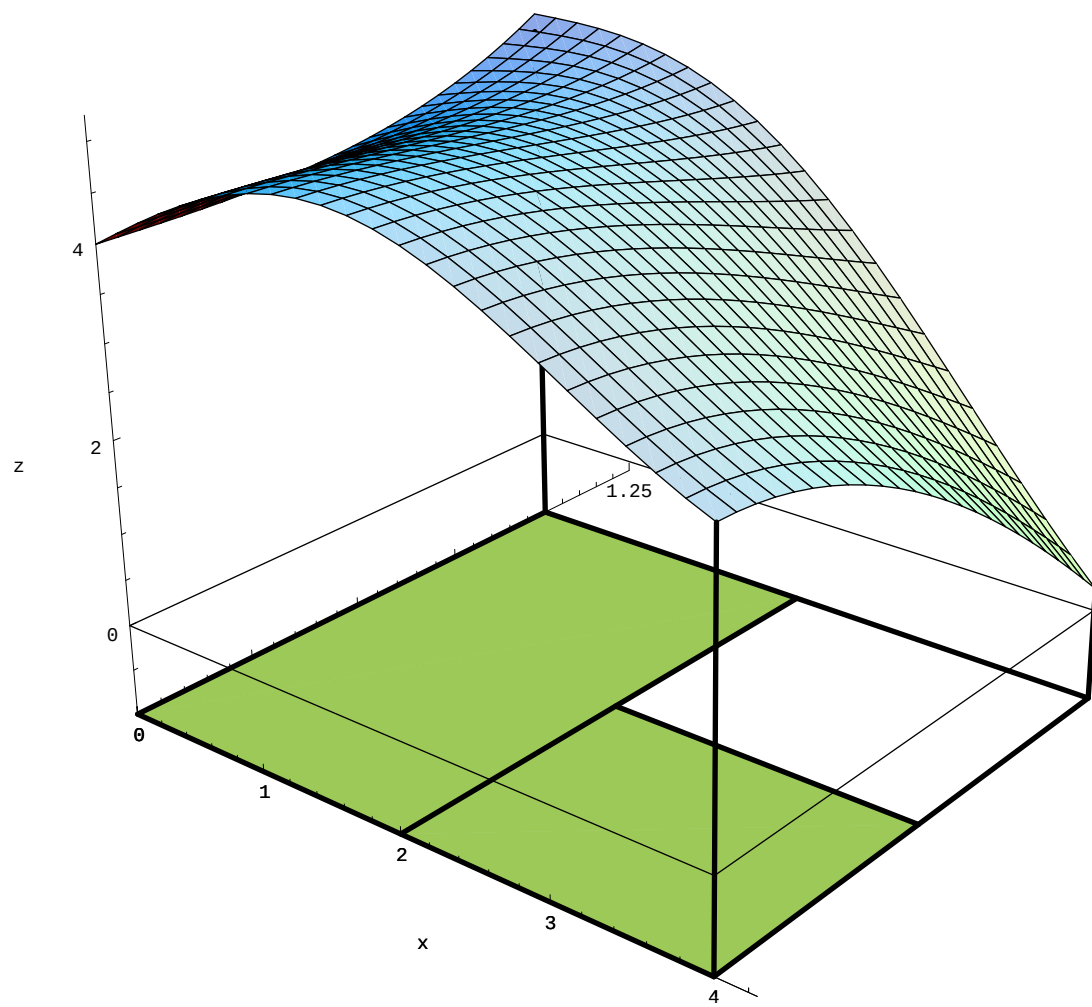


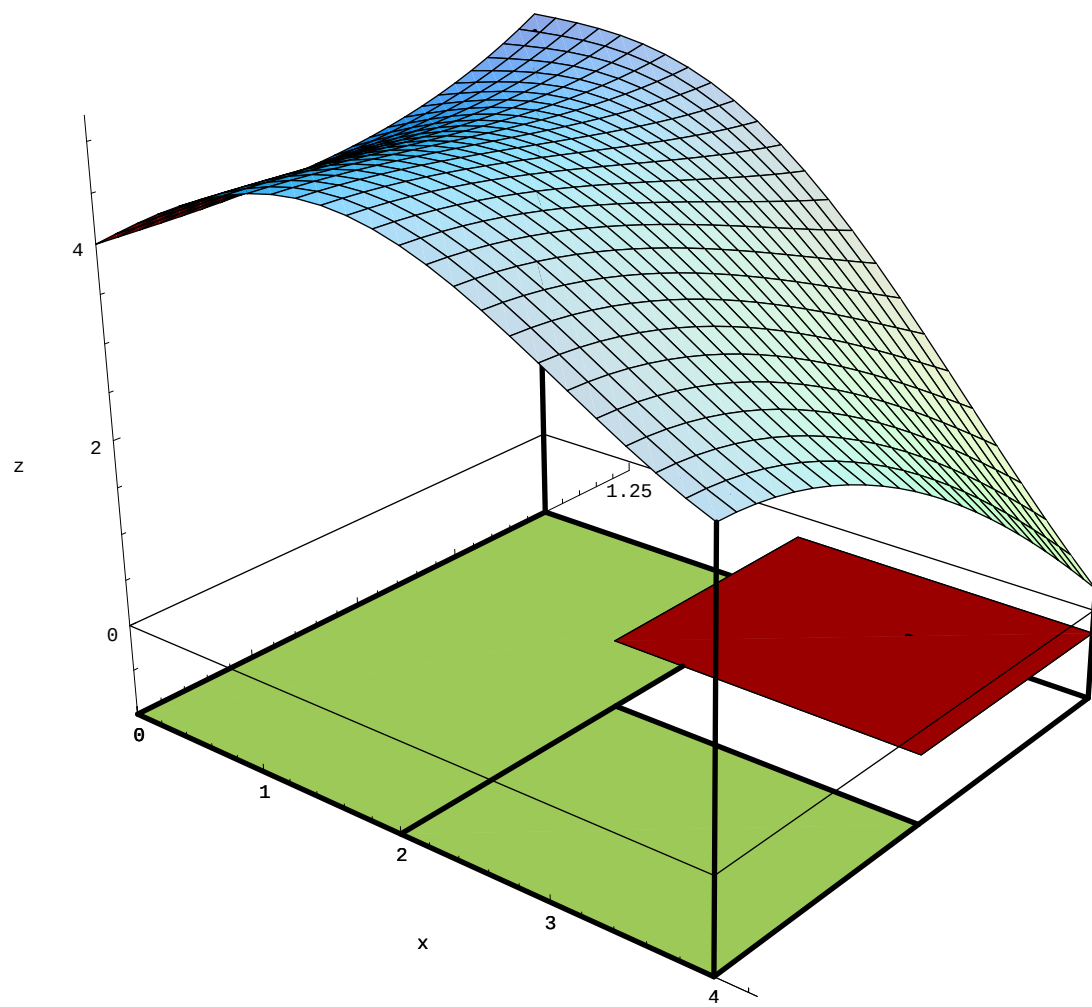


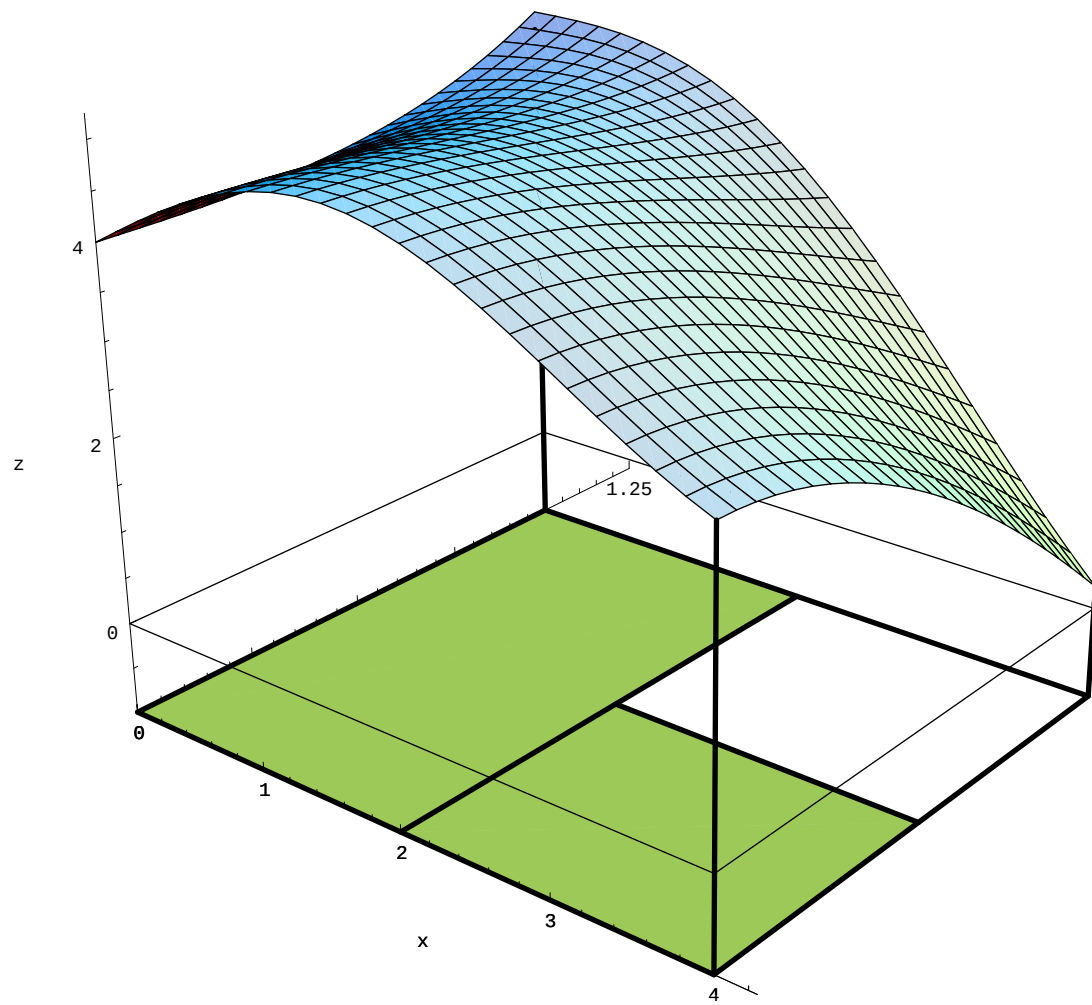


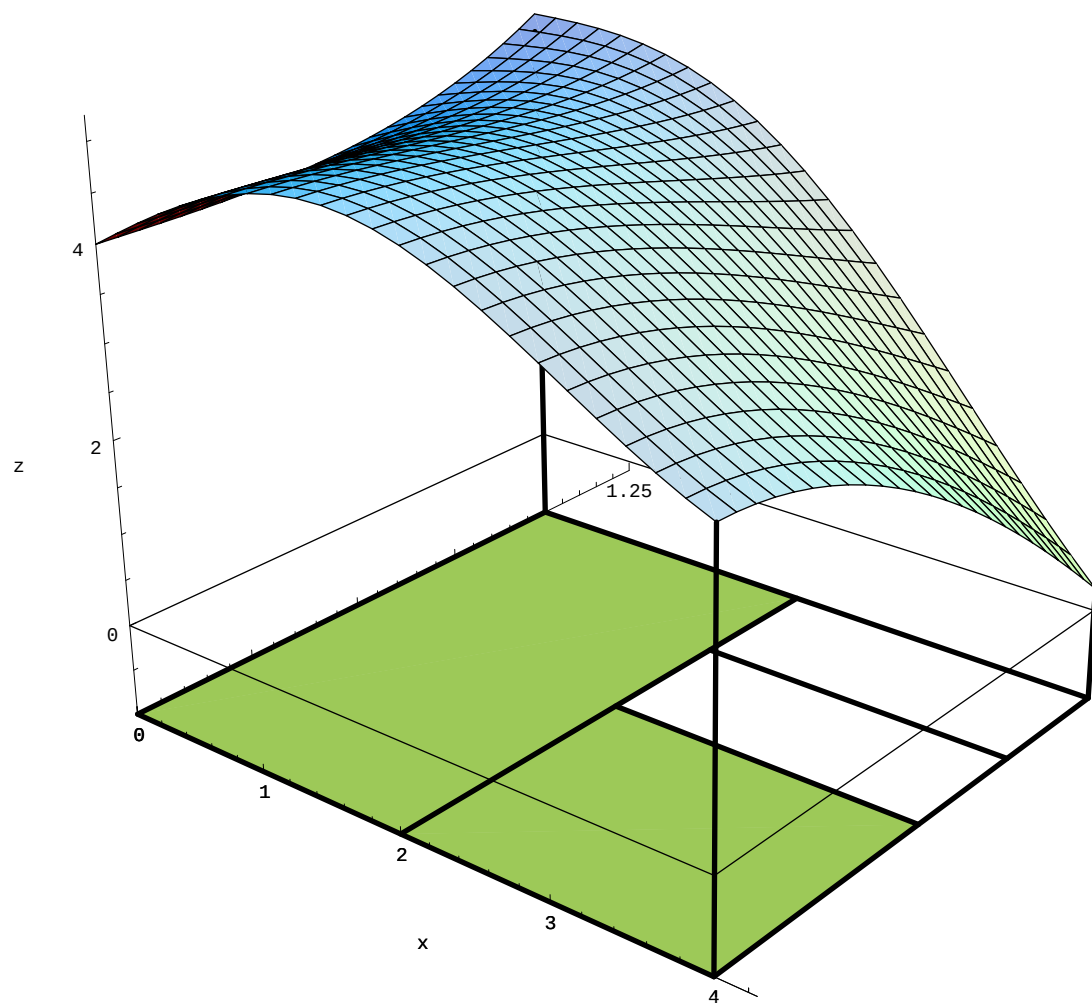


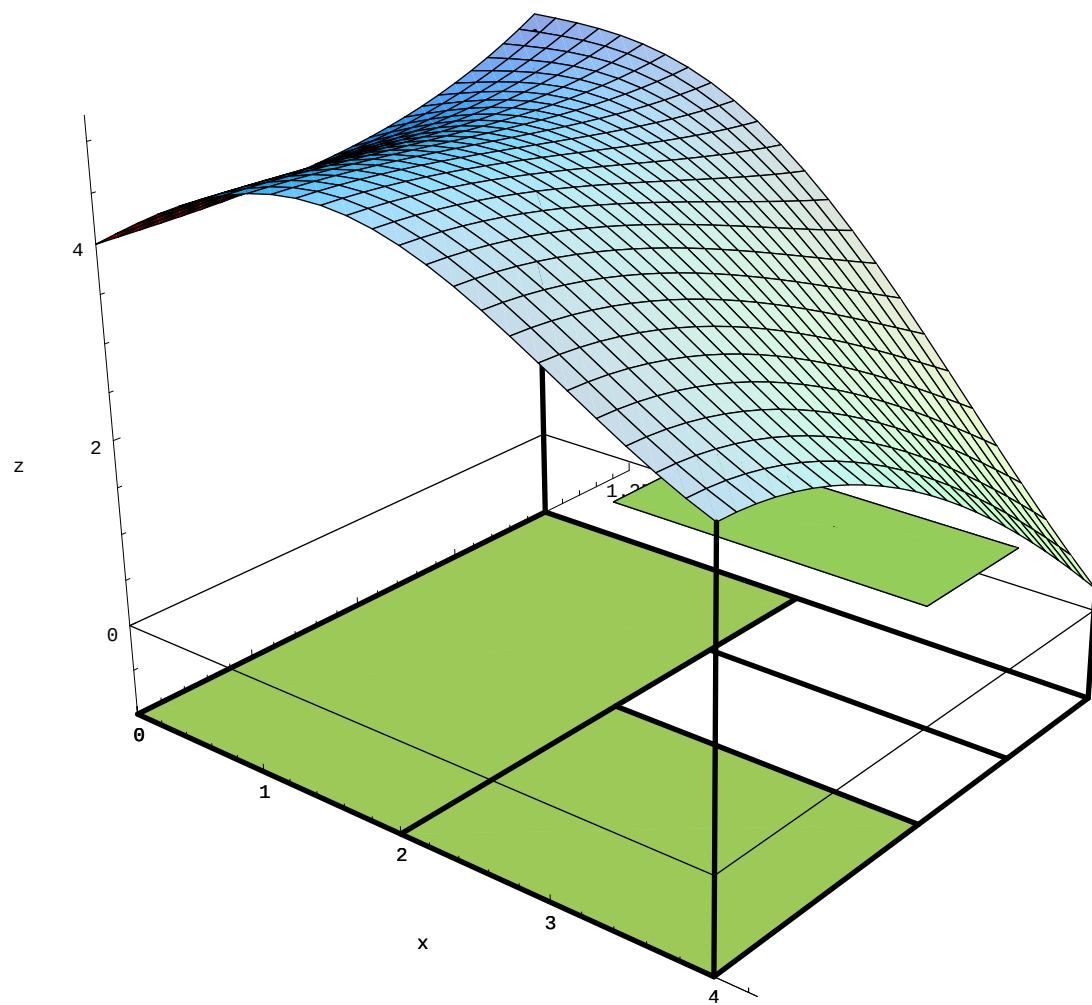


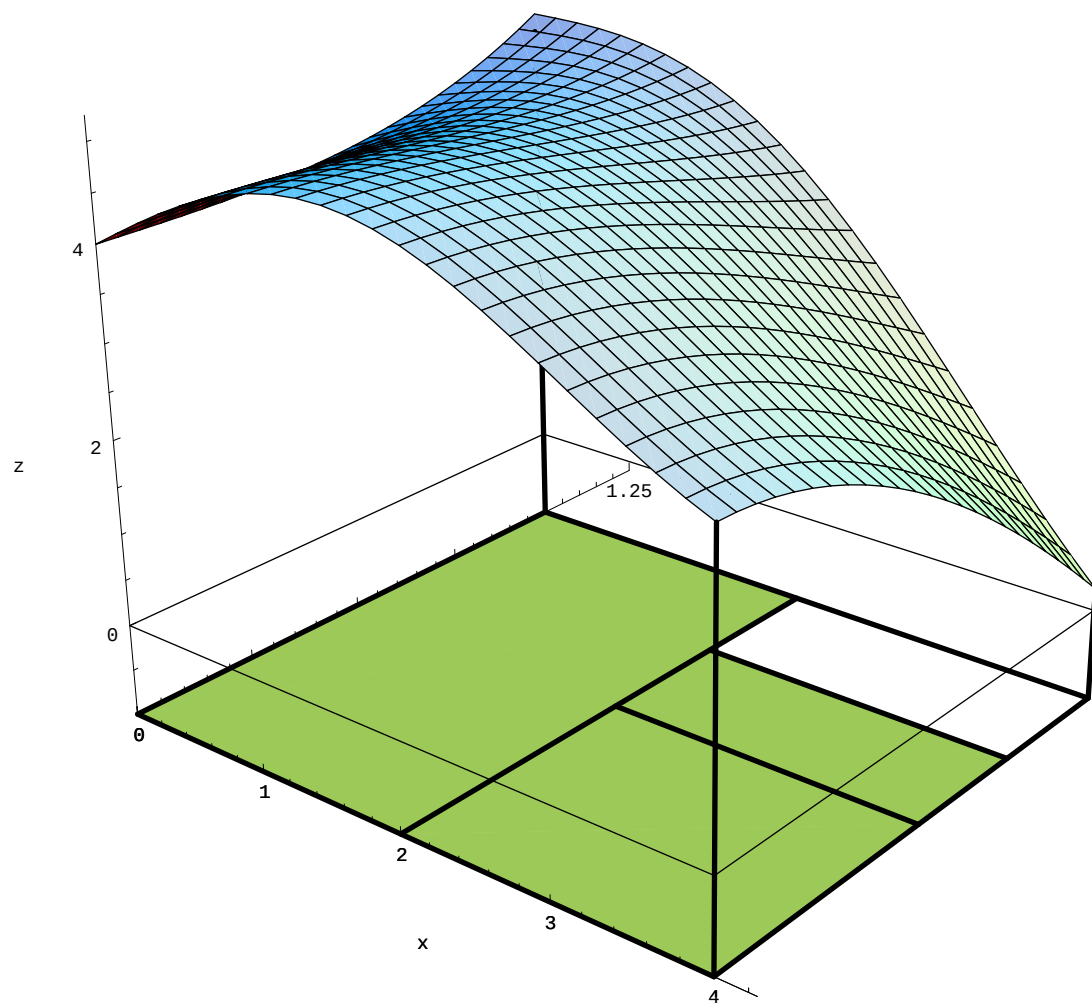


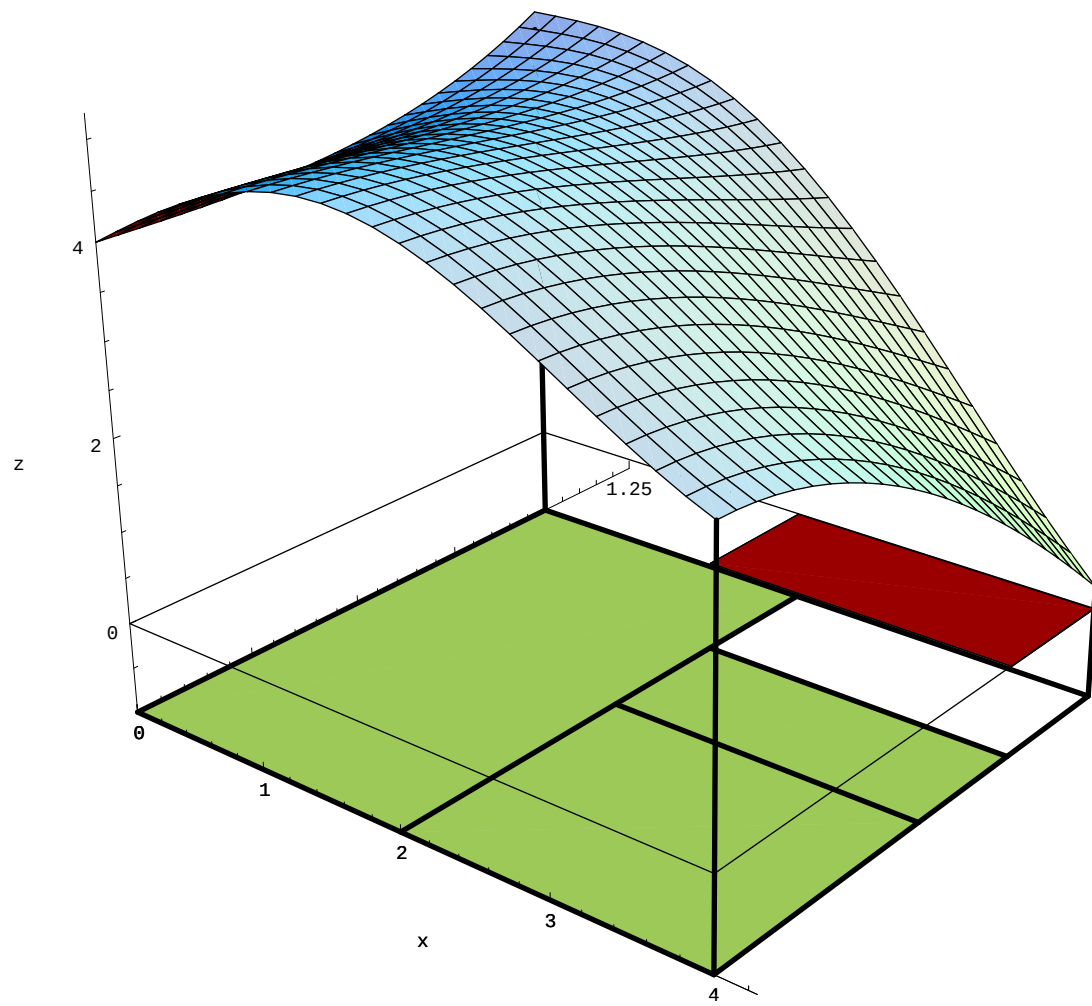


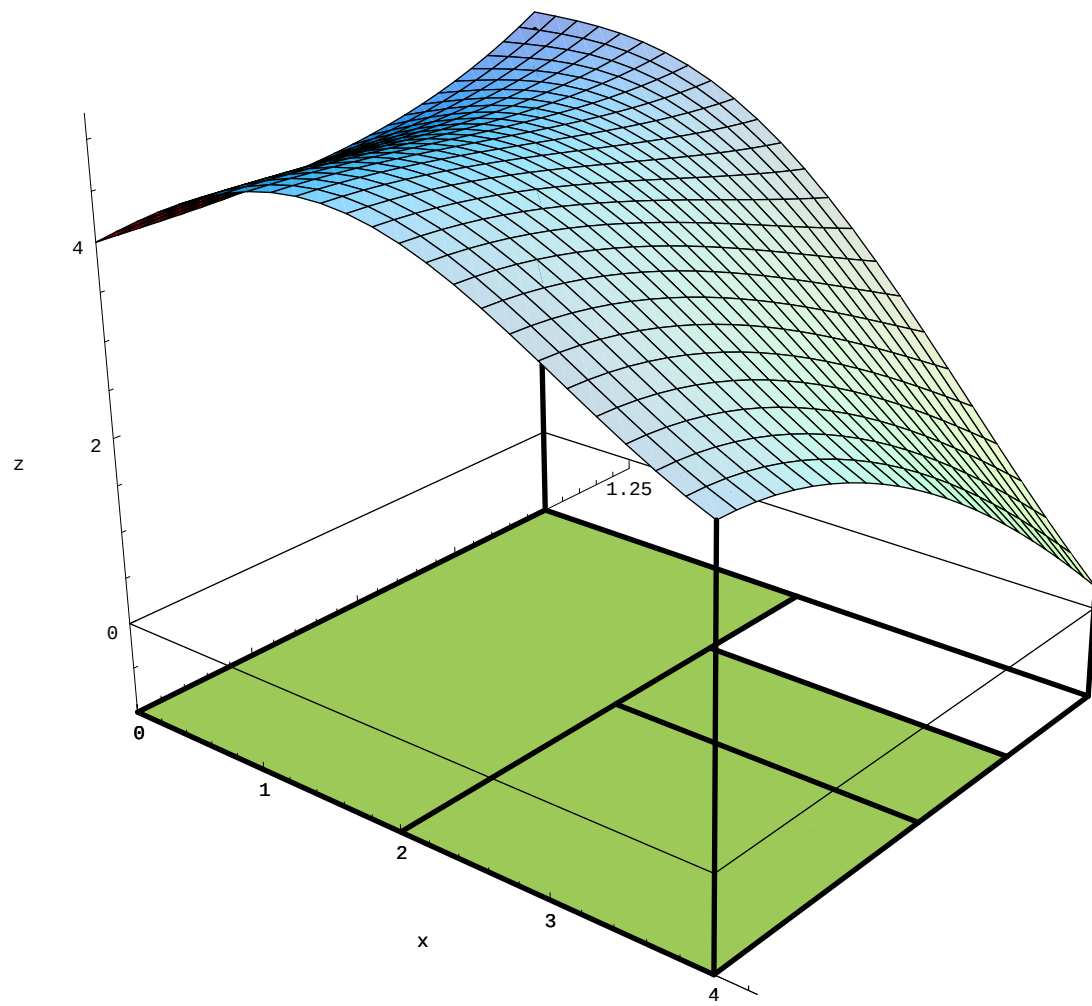


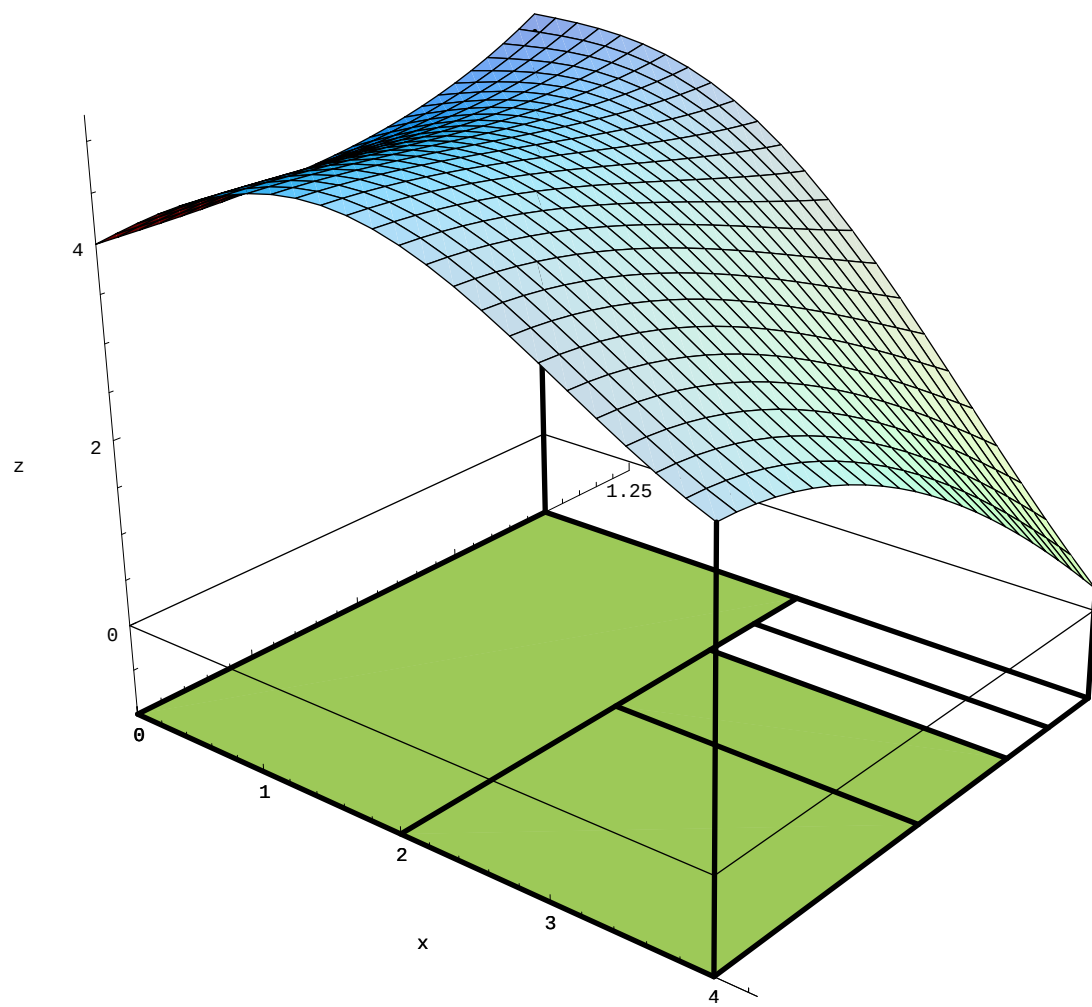


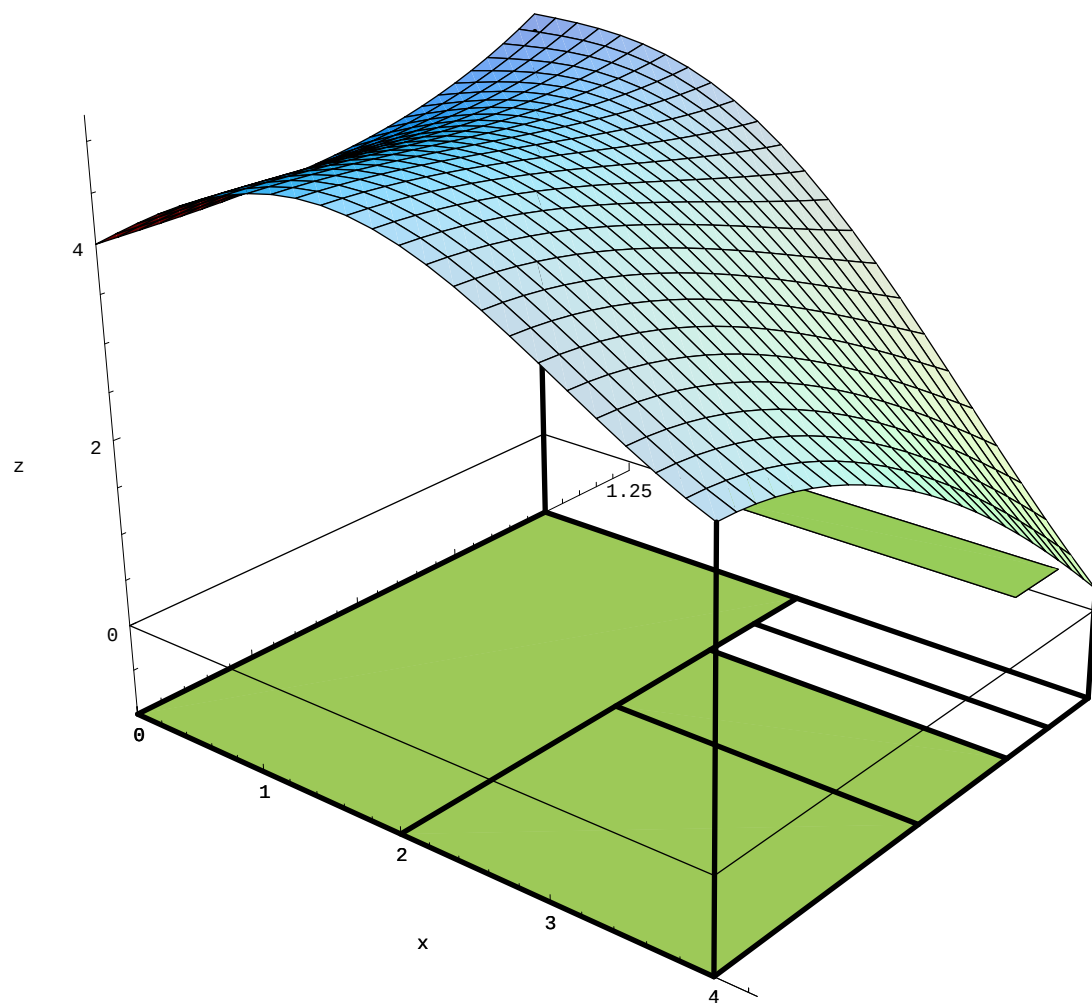


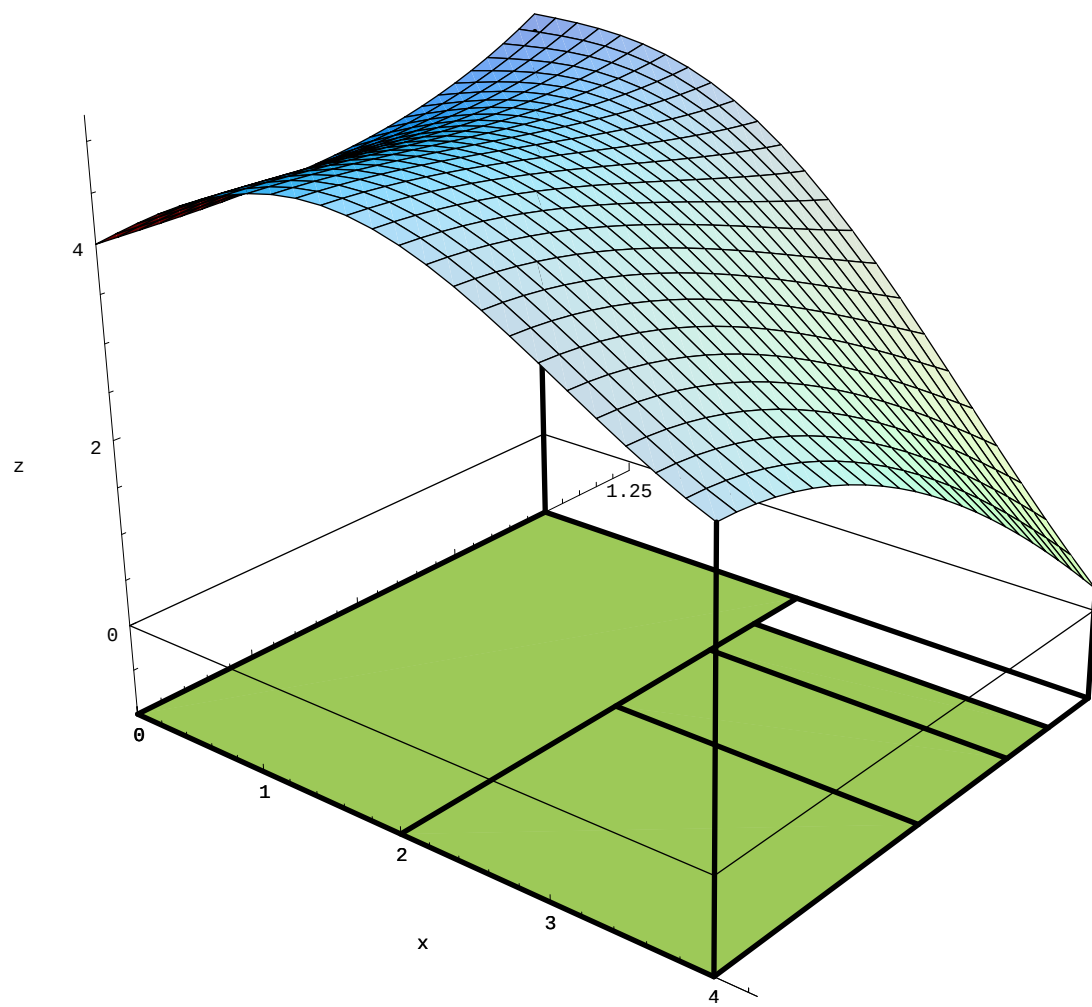


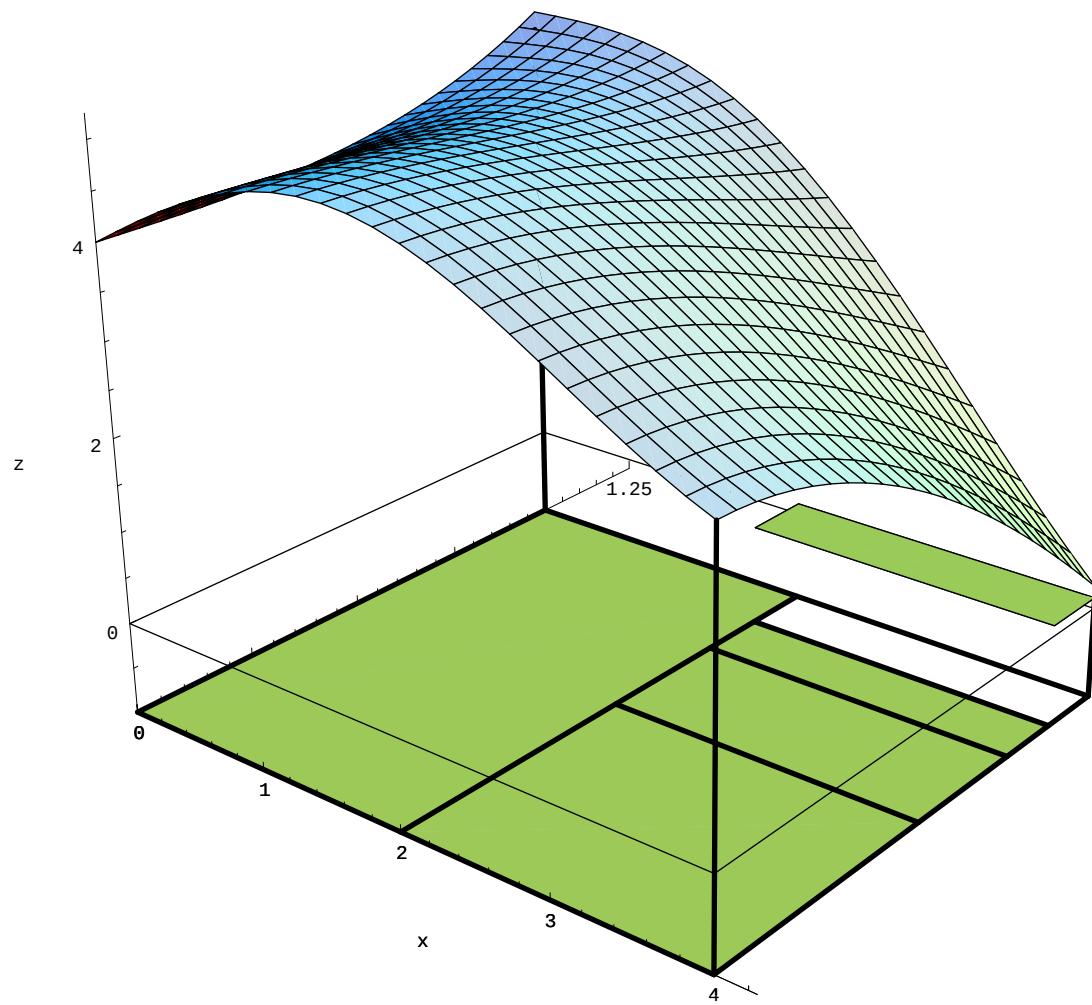


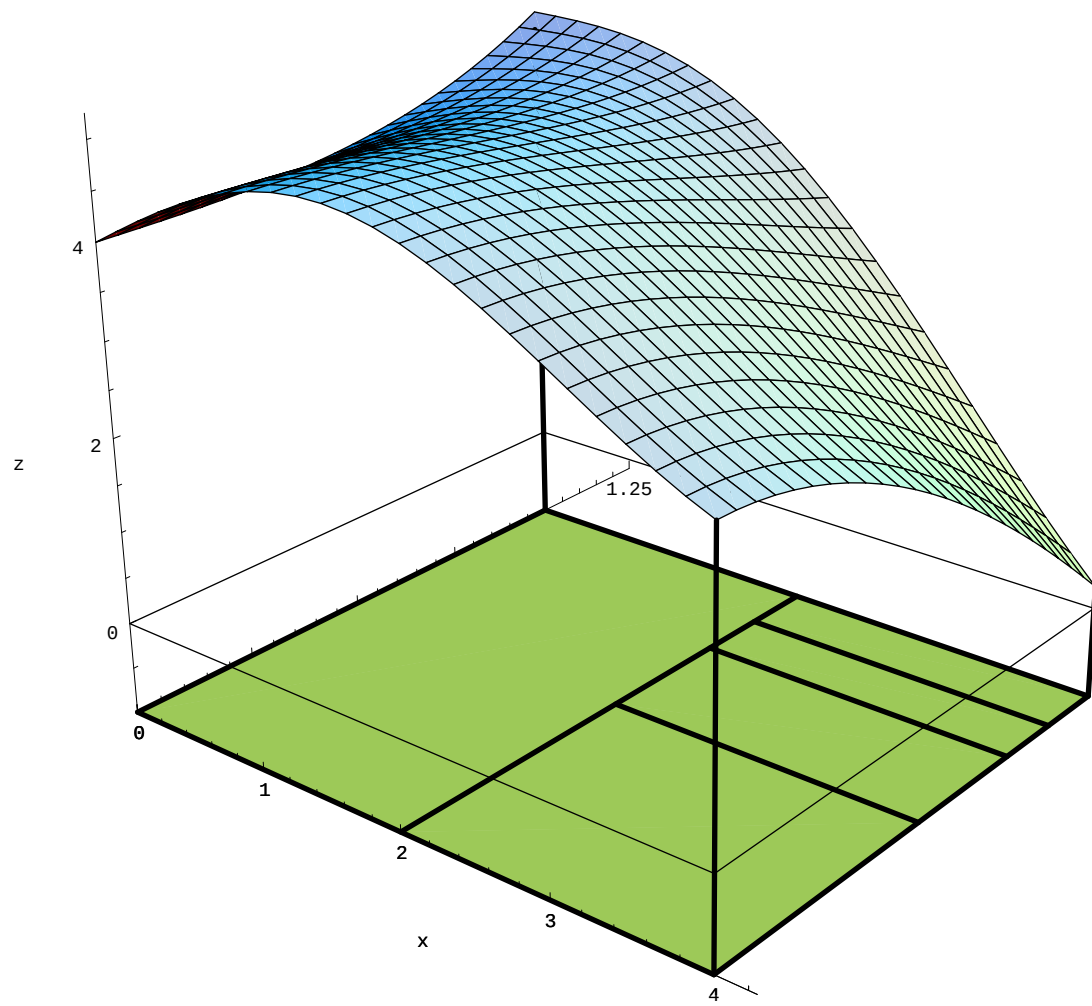












Interval Bounds

- Most existing interval arithmetic software uses rationals, or floating-point numbers with directed rounding.
- This means that transcendent functions have to be approximated to a certain precision: $\sqrt{(0, 2)} = (\sqrt{0}, \sqrt{2}) = (0, 1.42)$
- We can no longer deduce $\sqrt{x} \leq 1.416$ from $x \in (0, 2)$! The tactic would blindly continue to break up intervals.
- Constructive real numbers^a $\mathbb{R} = \mathbb{Z} \rightarrow \mathbb{Z}$ and for any real r :
$$\forall z \in \mathbb{Z}. |4^z - r(z)| < 1$$
- They allow to dynamically increase precision:
$$\sqrt{(0, 2)} = (\sqrt{0}, \sqrt{2}) = (0, 1.4 \dots) = (0, 1.41 \dots) = (0, 1.414 \dots)$$
- Decidability of $=$ and $\leq$ is lost. No proofs for sharp irrational bounds.

^amodel: Valérie Ménéssier-Morain, implementation OCaml: Jean-Christophe Filliâtre

Semi-Decision Tactics in Coq

- The tactic is entirely written in Coq (reflection, no ML)
- Its type is $t = \text{forall } p : I, \text{ nat} \rightarrow \text{option } p$
 - I is the type of inequalities
 - p is a particular inequality
 - the natural number is the search depth
 - the returned value is ...
 - * $t \ p \ 5 = \text{None}$... nothing (tactic failure)
 - * $t \ p \ 50 = \text{Some } (\text{fun } x1 \ x2 \dots \Rightarrow \dots)$... or a proof of p
- Variant: simultaneous search for counter-examples

Thank You!

Author: Dan Synek

Title: Formalising Lagrange Theorem in Coq

Abstract:

Lagrange Theorem states that if G is a finite group and H is a subgroup of G , then the order of H divides the order of G . It is a nice exercise to formalise it, since it contains many important notions from algebra, such as the formation of quotients and the notion of finiteness. The proof, when given in normal textbooks, is only a few lines long. The formalised proof is many pages long. I will try to analyse why, and in particular point out problems with the use of subsetoids.

Formalising Lagrange's Theorem in Coq

Dan Synek

Radboud University Nijmegen

WHY?

Informal Proof

Let G be a finite group. The order of a subgroup H of G divides the order of G .

Informal Proof: Form the cosets Hg of G where $Hg := \{hg | h \in H\}$

1. The cosets form a partition, e.g. $Hg_1 = Hg_2$ or $Hg_1 \cap Hg_2 = \emptyset$:
Assume $Hg_1 \cap Hg_2 \neq \emptyset$ so $h_1g_1 = h_2g_2$ and thus $g_1 = h_1^{-1}h_2g_2$,
then $g_1 \in Hg_2$ and thus $Hg_1 \subseteq Hg_2$. By symmetry $Hg_2 = Hg_1$.
2. Each coset has exactly $|H|$ elements: If $h_1g = h_2g$ then $h_1 = h_2$.

Informal Version of Formal Proof

$$1. G \simeq G/H \times H$$

$$2. |A \times B| = |A||B|$$

where the equality on G/H is defined by $\hat{x} = \hat{y}$ iff $xy^{-1} \in H$. We do not always have:

$$\forall a : A \exists b : B, (Rab) \rightarrow \exists f : (A \Rightarrow B), \forall a : A, (Ra(fa))$$

but we can prove it for finite A .

The isomorphism is given by:

$$f(g) = \langle \hat{g}, g \cdot (c(g))^{-1} \rangle$$

$$f^{-1} \langle \hat{g}, h \rangle = h \cdot (c(g))$$

(2) is proved by induction over the size of B .

What I had and what needed to be done

Corn has:

- Setoids, including functions, union and products between setoids.
- Groups, including simplification of algebraic expressions.

I had to do:

- Define finiteness.
- Define isomorphisms and prove lemmas about it.
 $\simeq$ forms equivalence, $A \simeq (S \mid P) \cup (A \mid \neg P)$ and
 $A \times (B \cup C) \simeq A \times B \cup A \times C$ about 15 more
- Relate decidability to isomorphisms, finiteness and subsets.
- Define extensional choice and prove it for finite sets.

So what is the problem with Setoids

A Setoid is a type and an equivalence relation.

- No obvious choice how to represent set operations. Should one work with predicates representing sets, with union as predicate operation, or work with disjoint union on the carriers? Is a subgroup and group based on different notion of set?
- Intensionality of type theory together with setoids often forces you to use isomorphisms, e.g. subset and quotient don't commute.
- A common construction is to make a setoid based on a sigma type with the carrier and a predicate, with the equality "inherited" from the original setoid. In practice using these setoids leads to extensive packing and unpacking (as observed by Luis Cruz-Filipe).
- $A \mid P \mid Q \simeq A \mid Q \mid P$ is not a meaningful statement.

Some observations

- Around 6000 lines of proof.
- Most lines are spent on proving lemmas about isomorphisms.
- We could hide the size of the proof behind fancy tactics. But is that what we want?
- Things like better syntax, setoid rewriting and a big library is not the solution.

Future Work

- Formalising can be a nice way to illustrate the power of abstraction. Motivation for proof assistants as a tool in teaching.
- Is PER/subsetoids a better framework to represent sets?
- Or more radically: Work in ZF coded in type theory a la Miguel
- Or even more radically: Find an application that justifies the choice of framework!

Author: Lionel Elie Mamane

Title: Surreal Numbers in Coq

Abstract:

Surreal (or Conway) Numbers form a totally ordered commutative Field, containing a copy of the reals and of the ordinals. The natural presentation of surreal numbers uses an induction-recursion scheme, which is not available in Coq. However, the original presentation by Conway explicitly works around needing induction-recursion, too. I encoded the Ring structure of surreal numbers in Coq. This encoding relies on Peter Aczel's encoding of set theory in type theory. The definition of the order, originally a direct inductive definition, needed to be separated into a mutually inductive definition of "at most" and "not at least" to satisfy Coq's positivity criterion. The recursion / induction schemes used in definitions / proofs turned out to be too subtle for Coq to accept straight away. This development making use of two different defined equalities, the new "Setoid Rewriting" code in Coq CVS is of great help.

Surreal Numbers in Coq

Types 2004 presentation

Lionel Elie Mamane^{*}
Radboud University Nijmegen[†]

17 December 2004

^{*} Position partially paid by NWO, the Netherlands Organisation for Scientific Research

[†] work started at Technical University of Eindhoven / École Normale Supérieure de Lyon

Overview

- Introduction
- Surreal Numbers
 - interpretation
- Games
 - As Bisets
 - Sets in TT
 - Games in TT
- Order
- Summary
- Conclusions

Introduction

Surreal Numbers

- John Horton Conway, *“On Numbers and Games”*
- Class of numbers containing
 - real numbers
 - all ordinals
- Fills the “holes” between the ordinals, like $\mathbb{R}$ fills the holes between the natural numbers.
- Fills the “holes” between the reals, too.
- Equipped with **totally ordered field** structure

Introduction

Surreal Numbers

And all this with only **one** set of inductive definitions!

Introduction

Surreal Numbers

And all this with only **one** set of inductive definitions!

Contrast with usual

Set Theory $\rightarrow \mathbb{N} \rightarrow \mathbb{Z} \rightarrow \mathbb{Q} \rightarrow \mathbb{R}$

construction.

Surreal Numbers

Definition

A surreal number x is a pair of arbitrary sets of surreal numbers L_x and R_x (left (resp. right) of x), with the condition:

$$\forall l \in L_x, \neg \exists r \in R_x \text{ s.t. } l \geq r$$

This x is denoted:

$$\{L_x \mid R_x\}$$

The class of all surreal numbers is denoted No .

Surreal Numbers

Interpretation

$x := \{L_x \mid R_x\}$ is the **simplest** number such that:

$$\forall x_l \in L_x, x > x_l$$

$$\forall x_r \in R_x, x < x_r$$



Surreal Numbers

Examples

$$\{\emptyset | \emptyset\} =: 0$$

$$\{\{0\} | \emptyset\} =: 1$$

$$\{\emptyset | \{0\}\} =: -1$$

$$\{\{0\} | \{0\}\} \text{ Forbidden!}$$

$$\{\{1\} | \{0\}\} \text{ Forbidden!}$$

$$\{\{0\} | \{1\}\} =: \frac{1}{2}$$

$$\{\{-1\} | \{0\}\} =: -\frac{1}{2}$$

$$\{\emptyset | \{1\}\} = 0 = \{\{-1\} | \{1\}\}$$

...in Coq

Games - The Need

$$x := \{L_x \mid R_x\}, \forall x_l \in L_x, \neg \exists x_r \in R_x, x_l \geq x_r$$

Mutually inductive definition of a number and of order on numbers.

Impossible in:

- Coq
- Formal mathematics as seen by Conway.

Possible in

- ALF & Co

...in Coq

Games

1. Games

A game x is a pair of arbitrary sets of games.

2. Order

3. Predicate “Is a number”

Games

Bisets

Games:

$$x := \{L_x \mid R_x\}$$

Sets:

$$s := \{L_s\} \simeq \{L_s \mid \emptyset\}$$

Natural injection: collection of all sets $\rightarrow$ collection of all games.

...in Coq

Sets

Two schemes:

- Subset of an existing type (universe): $s : G \rightarrow *_{\mathcal{U}}$

Like in $\{x \in G : s(x)\}$

- Indexed by some “index set/type”: (I, f) , where $f : I \rightarrow G$

Like in $\{f(x) : x \in I\}$

...in Coq

Games in Coq

```
Inductive Game : Type :=  
  Game_cons : (LI, RI : Type)  
              (LI -> Game) ->  
              (RI -> Game) ->  
              Game .
```

Credit: Benjamin Werner, *“Sets in Types, Types in Sets”*:

```
Inductive Ensemble : Type :=  
  sup : (A : Type) (A -> Ensemble) -> Ensemble .
```

Surreal Numbers...

Order

$$x \leq y \stackrel{\text{def}}{\iff} (\forall x_l, \neg(x_l \geq y)) \wedge (\forall y_r, \neg(x \geq y_r))$$

_____ x _____

_____ y _____

Surreal Numbers...

Order

$$x \leq y \stackrel{\text{def}}{\iff} (\forall x_l, \neg(x_l \geq y)) \wedge (\forall y_r, \neg(x \geq y_r))$$

_____ x _____

_____ y _____

$$x_l < x$$

If $y \leq x_l$, then $y < x$, and $\neg(x \leq y)$.

...in Coq

Order

$$x \leq y \stackrel{\text{def}}{\longleftrightarrow} (\forall x_l, \neg(y \leq x_l)) \wedge (\forall y_r, \neg(y_r \leq x))$$

Cannot define $\leq$ inductively from its negation! Idea from Frank Rosemeier: $\triangleleft \leftrightarrow \neg \geq$
and define both mutually inductive:

$$\begin{aligned} x \leq y & \stackrel{\text{def}}{\longleftrightarrow} (\forall x_l, x_l \triangleleft y) \wedge (\forall y_r, x \triangleleft y_r) \\ x \triangleleft y & \longleftrightarrow \neg((\forall y_l, \neg(x \leq y_l)) \wedge (\forall x_r, \neg(x_r \leq y))) \\ & \stackrel{\text{def}}{\longleftrightarrow} (\exists x_r, x_r \leq y) \vee (\exists y_l, x \leq y_l) \end{aligned}$$

Summary

Done

- Definition: games, order, “is a number”, addition, opposite, multiplication
- Properties: Commutative ordered group, distributivity, morphisms, ...
- Nearly: Commutative ordered ring; associativity of Mult is missing.

Conclusions, Future Plans

- Many “similar” (symmetric) cases; Coq could handle that better.
- Permuting induction schemes: Easier than writing lambda-term? Automatic? (cf size-change termination criterion)

Linked to accepting a larger class of recursive definitions.

- Defined equality: Getting much better (Claudio Sacerdoti Coen)
- More fine-grained control of reduction.
- `Destruct x`: Would want to “remember” $x = \{L_x \mid R_x\}$.

Closing

That's all, folks!
Questions?

Coq

Order

```
Inductive Glte : Game -> Game -> Prop :=
Glte_cons : (xLI, xRI : Type) (xLf : xLI -> Game)
              (xRf : xRI -> Game) (yLI, yRI : Type)
              (yLf : yLI -> Game) (yRf : yRI -> Game)
              [x := (Game_cons xLI xRI xLf xRf);
               y := (Game_cons yLI yRI yLf yRf)]
              ((l : xLI) [x1 := (xLf l)] (NGgte x1 y)) ->
              ((r : yRI) (NGgte x (yRf r))) ->
              (Glte x y)
```

```

with NGgte : Game -> Game -> Prop :=
  NGgte_xr : (xLI, xRI : Type) (xLf : xLI -> Game)
    (xRf : xRI -> Game) (y : Game)
    [x := (Game_cons xLI xRI xLf xRf)]
    (exT xRI [r : xRI] (Glte (xRf r) y)) ->
    (NGgte x y)
| NGgte_yl : (x : Game) (yLI, yRI : Type) (yLf : yLI -> Game)
    (yRf : yRI -> Game)
    [y := (Game_cons yLI yRI yLf yRf)]
    (exT yLI [l : yLI] (Glte x (yLf l))) ->
    (NGgte x y)

```

•

Surreal Numbers...

Examples - $\mathbb{N}$

$$0 := \{\emptyset \mid \emptyset\}$$

$$1 := \{\{0\} \mid \emptyset\}$$

$$2 := \{\{1\} \mid \emptyset\} = \{\{0, 1\} \mid \emptyset\}$$

...

$$n + 1 := \{\{n\} \mid \emptyset\}$$

We now have a copy of $\mathbb{N}$ in No .

Surreal Numbers...

Examples - On

φ the canonical injection $On \rightarrow No$, n an ordinal, N a set of ordinals.

$$\varphi(\text{Successor of } n) := \{\{\varphi(n)\} \mid \emptyset\} = \{\{\varphi(p), p \leq n\} \mid \emptyset\}$$

$$\varphi(\text{Limit ordinal of } N) := \{\{\varphi(n) : n \in N\} \mid \emptyset\}$$

We now have a copy of On in No .

Surreal Numbers...

Addition

$$x + y \stackrel{\text{def}}{=} \{x_l + y, x + y_l \mid x_r + y, x + y_r\}$$

Surreal Numbers...

Addition

$$x + y \stackrel{\text{def}}{=} \{x_l + y, x + y_l \mid x_r + y, x + y_r\}$$

$$x_l < x \rightarrow x_l + y < x + y$$

Surreal Numbers...

Examples

$$\frac{1}{2} := \{\{0\} \mid \{1\}\}$$

$$\frac{1}{4} := \{\{0\} \mid \{\frac{1}{2}\}\}$$

$$\frac{3}{8} := \{\{\frac{1}{4}\} \mid \{\frac{1}{2}\}\}$$

$$\frac{1}{3} := \left\{ \left\{ \frac{1}{4}, \frac{1}{4} + \frac{1}{16}, \frac{1}{4} + \frac{1}{16}, \frac{1}{4} + \frac{1}{16} + \frac{1}{32}, \dots \right\} \mid \left\{ \frac{1}{2}, \frac{1}{2} - \frac{1}{8}, \dots \right\} \right\}$$

We now have a copy of $\mathbb{R}$ in No .

Surreal Numbers...

Examples

Beyond $\mathbb{R}$...

What about $\varepsilon := \left\{ \{0\} \mid \left\{ \frac{1}{n} : n \in \mathbb{N} \right\} \right\}$?

Surreal Numbers...

Examples

Beyond $\mathbb{R}$...

What about $\varepsilon := \left\{ \{0\} \mid \left\{ \frac{1}{n} : n \in \mathbb{N} \right\} \right\}$?

- $\varepsilon > 0$
- $\forall n \in \mathbb{N}, \varepsilon < \frac{1}{n}$

Surreal Numbers...

Examples

Beyond $\mathbb{R}$...

What about $\varepsilon := \left\{ \{0\} \mid \left\{ \frac{1}{n} : n \in \mathbb{N} \right\} \right\}$?

- $\varepsilon > 0$
- $\forall n \in \mathbb{N}, \varepsilon < \frac{1}{n}$

It is an **infinitesimal**.

Surreal Numbers...

Examples

Beyond $\mathbb{R}$...

What about $\varepsilon := \left\{ \{0\} \mid \left\{ \frac{1}{n} : n \in \mathbb{N} \right\} \right\}$?

- $\varepsilon > 0$
- $\forall n \in \mathbb{N}, \varepsilon < \frac{1}{n}$

It is an **infinitesimal**. In fact, it is $\frac{1}{\omega}$.

Surreal Numbers...

Examples

Beyond $\mathbb{R}$...

What about $\varepsilon := \left\{ \{0\} \mid \left\{ \frac{1}{n} : n \in \mathbb{N} \right\} \right\}$?

- $\varepsilon > 0$
- $\forall n \in \mathbb{N}, \varepsilon < \frac{1}{n}$

It is an **infinitesimal**. In fact, it is $\frac{1}{\omega}$.

$\omega - \varepsilon, 1 - \varepsilon^2, \omega + \varepsilon, \frac{3}{7} \times \varepsilon$ all have a meaning.

Surreal Numbers...

Examples

Beyond $\mathbb{R}$... Not finished! Second-level infinitesimal:

$$\{\{0\} \mid \{\varepsilon^n : n \in \mathbb{N}\}\}$$

And you can go on... ω -th level infinitesimal, $\aleph_{13}$ -th level infinitesimal, ...

...in Coq

Addition

$$x + y \stackrel{\text{def}}{=} \{x_l + y, x + y_l \mid x_r + y, x + y_r\}$$

...in Coq

Addition

$$\begin{aligned}x + y &= \{x_l + y, x + y_l \mid x_r + y, x + y_r\} \\&= \{x_l + y, (+_x y_l) \mid x_r + y, (+_x y_r)\}\end{aligned}$$

where $\forall z, (+_x z) \equiv (x + z)$

$$\stackrel{\text{def}}{=} \{x_l + y, (+_x y_l) \mid x_r + y, (+_x y_r)\}$$

where

$$+_x \stackrel{\text{def}}{=} z. \{x_l + z, (+_x z_l) \mid x_r + z, (+_x z_r)\}$$

... in Coq

Addition

$$x + y \stackrel{\text{def}}{=} \{x_l + y, (+_x y_l) \mid x_r + y, (+_x y_r)\}$$
$$\text{where } +_x \stackrel{\text{def}}{=} z. \{x_l + z, (+_x z_l) \mid x_r + z, (+_x z_r)\}$$

- This construction “obviously” terminates.
- But breaks the symmetry of addition: Why $+_x$ and not $+_y$?

Credits: Classical Ackerman Function example (and encouragement from Benjamin Werner)

Conclusion

Done:

- $\leq$ is a (pre-)order
- Commutative additive group
- Multiplication
 - 0 absorbent
 - 1 neutral
 - Commutativity, associativity
 - $(-x)y = -(xy) = x(-y)$
 - Distributivity

Conclusion

To do:

- Multiplication
 - Compatibility with order
 - Division
- Transitivity of order: Constructive?
- Normal form
- Irreducible Numbers
- Algebra and Analysis
- Integers, continued fractions, ...

Conclusion

Coq flaws and limitations

- Lack of inductive-recursive definitions
- Forces the mathematician to think constructively in his structures definitions
- Throws away information
- Uncontrolled calculus (only normalisation)
- No good way to treat similar cases
- No good way to invoke intricate induction schemes
- The tools don't deal gracefully with a defined equality

Conclusion

- Formalising in Coq takes too much time.

- Some proofs in ONAG misleadingly simple.

Formalising in Coq shows this, it is thus relevant.

- Proof assistants are tools. Document in user-oriented way. Specialised documentation for various users.
- Per-result dependency tracking

THE END

$\leq$ is an order

Summary

$\leq$ is an order on Games (pre-order on representants):

- reflexive
- transitive

Reflexivity

$$x \leq x$$

$$\leftrightarrow \quad \{\text{Def}\}$$

$$\forall x_l, x_l \triangleleft x \wedge \forall x_r, x \triangleleft x_r$$

$$\leftarrow \quad \{\text{Def}\}$$

$$\forall x_l \exists x'_l, x_l \leq x'_l \wedge \forall x_r \exists x'_r, x_r \leq x_r$$

$$\leftarrow \quad \{\exists\text{-intro}\}$$

$$\forall x_l, x_l \leq x_l \wedge \forall x_r, x_r \leq x_r$$

$$\leftarrow \quad \{\text{IH}\}$$

⊤

Transitivity

$$x \leq y \rightarrow y \leq z \rightarrow x \leq z$$

Here's how ONAG does it:

Since $x \leq y$, we cannot have $x_l \geq y$, and so by induction we cannot have $x_l \geq z$.

Similarly we cannot have $x \geq z_r$, and so we must have $x \leq z$.

Indeed, $x \leq z \leftrightarrow \neg(x_l \geq z) \wedge \neg(x \geq z_r)$

Transitivity

$$x \leq y \rightarrow y \leq z \rightarrow x \leq z$$

Here's how ONAG does it:

Since $x \leq y$, we cannot have $x_l \geq y$, and so by induction we cannot have $x_l \geq z$.

Similarly we cannot have $x \geq z_r$, and so we must have $x \leq z$.

Indeed, $x \leq z \leftrightarrow \neg(x_l \geq z) \wedge \neg(x \geq z_r)$

Who is convinced?

Transitivity

$$x \leq y \rightarrow y \leq z \rightarrow x \leq z$$

Here's how ONAG does it:

Since $x \leq y$, we cannot have $x_l \geq y$, and so by induction we cannot have $x_l \geq z$.

Similarly we cannot have $x \geq z_r$, and so we must have $x \leq z$.

Indeed, $x \leq z \leftrightarrow \neg(x_l \geq z) \wedge \neg(x \geq z_r)$

Who is convinced? You should not be!

Transitivity

$$x \leq y \rightarrow y \leq z \rightarrow x \leq z$$

Here's how ONAG does it:

Since $x \leq y$, we cannot have $x_l \geq y$, and so by induction we cannot have $x_l \geq z$. Similarly we cannot have $x \geq z_r$, and so we must have $x \leq z$.

Transitivity

$$x \leq y \rightarrow y \leq z \rightarrow x \leq z$$

Here's how ONAG does it:

Since $x \leq y$, we cannot have $x_l \geq y$, and so by induction we cannot have $x_l \geq z$. Similarly we cannot have $x \geq z_r$, and so we must have $x \leq z$.

$$x \leq z \leftrightarrow \neg(x_l \geq z) \wedge \neg(x \geq z_r)$$

thus it suffices to prove:

$$x \leq y \rightarrow y \leq z \rightarrow \neg(x_l \geq z)$$

$$x \leq y \rightarrow y \leq z \rightarrow \neg(x \geq z_r)$$

Transitivity

$$x \leq y \rightarrow y \leq z \rightarrow x \leq z$$

$$x \leq y \rightarrow y \leq z \rightarrow \neg(x_l \geq z)$$

$$x \leq y \rightarrow y \leq z \rightarrow \neg(x \geq z_r)$$

Transitivity

$$x \leq y \rightarrow y \leq z \rightarrow x \leq z$$

$$x \leq y \rightarrow y \leq z \rightarrow \neg(x_l \geq z)$$

$$x_l \geq z$$

$$\rightarrow \{IH : x_l \geq z \rightarrow z \geq y \rightarrow x_l \geq y, z \geq y\}$$

$$\{IH : y \leq z \rightarrow z \leq x_l \rightarrow y \leq x_l, y \leq z\}$$

$$x_l \geq y$$

$$\rightarrow \{x \leq y\}$$

$\perp$

Transitivity

$$x \leq y \rightarrow y \leq z \rightarrow x \leq z$$

$$x \leq y \rightarrow y \leq z \rightarrow \neg(x_l \geq z)$$

$$x_l \geq z$$

$$\rightarrow \{IH_x : x_l \geq z \rightarrow z \geq y \rightarrow x_l \geq y, z \geq y\}$$

$$x_l \geq y$$

$$\rightarrow \{x \leq y\}$$

$\perp$

Proof of $\neg(x_l \geq z)$ uses $\text{IH}_x(x, z, y)$, but proof of $\neg(x \geq z_r)$ uses $\text{IH}_z(y, x, z)$

$$x_l \geq z$$

$$\rightarrow \{ \text{IH}_x : x_l \geq z \rightarrow z \geq y \rightarrow x_l \geq y, z \geq y \}$$

$$x_l \geq y$$

$$\rightarrow \{x \leq y\}$$

$$\perp$$

$$x \geq z_r$$

$$\rightarrow \{ \text{IH}_z : y \geq x \rightarrow x \geq z_r \rightarrow y \geq z_r, y \geq x \}$$

$$y \geq z_r$$

$$\rightarrow \{y \leq z\}$$

$$\perp$$

Proof of $\neg(x_l \geq z)$ uses $\text{IH}_x(x, z, y)$, but proof of $\neg(x \geq z_r)$ uses $\text{IH}_z(y, x, z)$

But

$$\begin{aligned} & (\forall x, y, z : (\forall z, y : P(x_l, z, y)) \rightarrow (\forall x, y : P(x, y, z_r)) \rightarrow P(x, y, z)) \\ & \rightarrow (\forall x, y, z : P(x, y, z)) \end{aligned}$$

is not a valid induction scheme!

Simplified version:

$$\begin{aligned} & (\forall x, y : (\forall y : P(x_l, y)) \rightarrow (\forall x : P(x, y_r)) \rightarrow P(x, y)) \\ & \rightarrow (\forall x, y : P(x, y)) \end{aligned}$$

$$\begin{aligned}
 &(\forall x, y, z : (\forall z, y : P(x_l, z, y)) \rightarrow (\forall x, y : P(x, y, z_r)) \rightarrow P(x, y, z)) \\
 &\rightarrow (\forall x, y, z : P(x, y, z))
 \end{aligned}$$

is not a valid induction scheme!

Correct induction scheme:

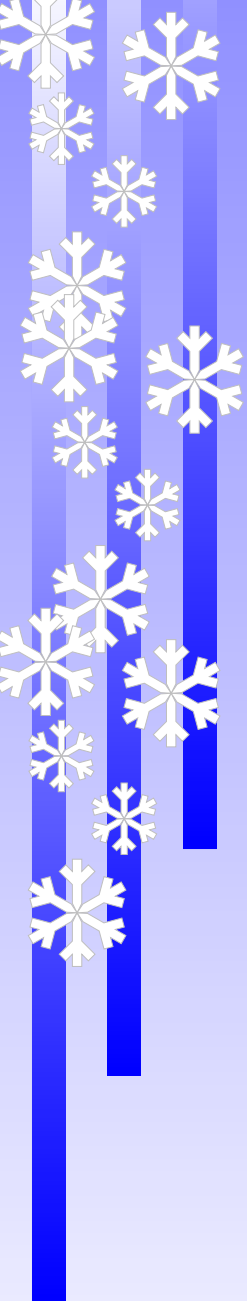
$$\begin{aligned}
 &(\forall x, y, z : P(y, z, x_l) \rightarrow P(z_r, x, y) \rightarrow P(x, y, z)) \\
 &\rightarrow (\forall x, y, z : P(x, y, z))
 \end{aligned}$$

Author: Jasper Stein

Title: Subsets in formalised linear algebra

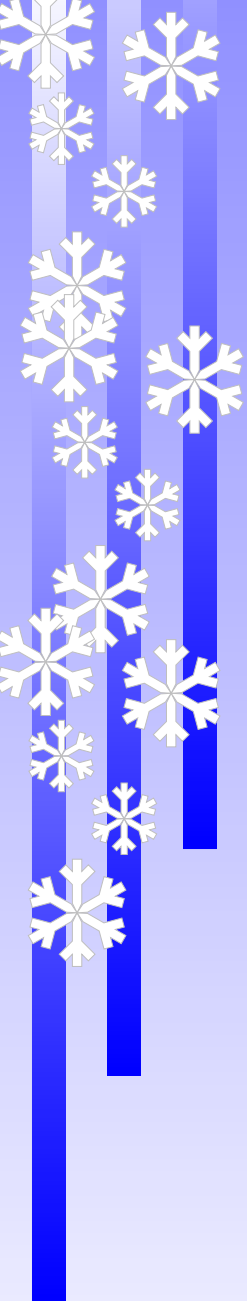
Abstract:

This talk is based on my LinAlg contribution, where I formalised preliminary linear algebra based on the setoid formalism. In mathematics, if $B \subseteq A$, then $a \in B$ and $a \in A$ simultaneously. In Coq, however, the setoid B needs a different carrier type, essentially $\Sigma a:A. P(a)$ for some predicate P . So typing both $\{a:A\}$ and $\{a:B\}$ is not possible, since terms have unique types up to conversion. One certainly wants π_1 as a coercion $B \rightarrow A$, but this is not enough when one formalises linear algebra. Additional type casting functions are needed when one works with subsets, and with subsets of subsets. In my talk I will introduce a few of them, and explain their necessity.



Subsets in formalised linear algebra

Jasper Stein



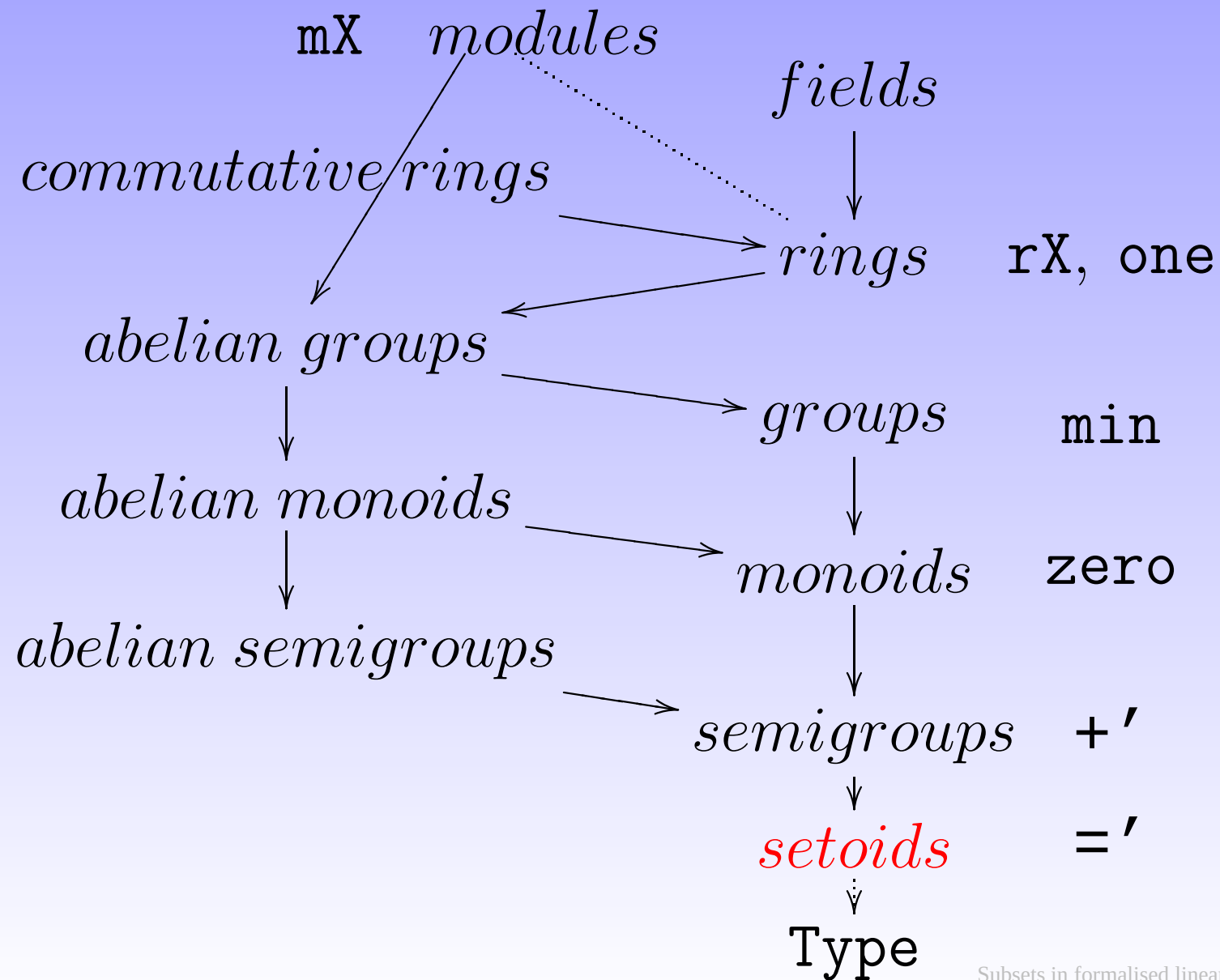
Contents

- ✓ Introduction
- ✓ The setoid formalism
- ✓ Up and down the subset ladder
- ✓ Across the subset ladder
- ✓ Conclusions

Introduction

- ✓ **LinAlg**: linear algebra in Coq
- ✓ vectorspaces, subspaces, linear combinations, linear (in)dependence, bases and dimension
- ✓ Book: *Linear Algebra*, 2nd ed. (chapter 1) by Friedberg, Insel and Spence
- ✓ Extends Loïc Pottier's *Algebra*

The setoid formalism from Algebra



The setoid formalism (contd.)

- ✓ A *setoid* is a triple $A := \langle X, =', p_A \rangle$ where p_A proves that $='$ is an equivalence relation on X .
- ✓ *(setoid-)compatibility*:
 $a = ' a' \text{ in } A \rightarrow$
 - ★ $(f \ a) = ' (f \ a') \text{ in } B$
 - ★ $(P \ a) \iff (P \ a')$
- ✓ Use this type for functions:

```
Record Map (A B:Setoid) :Type :=  
{ Ap :> A -> B ;  
  prf : a = ' a' -> Ap a = ' Ap a' }
```
- ✓ Similarly *Predicate A*

Subsets(1)

The setoid `part_set A` has:

- ✓ carrier Predicate `A`
- ✓ equality $P ='_Q \stackrel{\text{def}}{\iff} [\forall a : A. (P\ a) \iff (Q\ a)]$

Every such Predicate $P : (\text{part\_set } A)$ induces:

```
Record subtype (P : part_set A) :=  
  { subtype_elt : A  
    ; subtype_prf : P subtype_elt }
```

so essentially $P \rightsquigarrow \Sigma a : A. P(a)$

and `subtype_elt` = π_1

Subsets (2)

Any predicate $B : \text{part\_set } A$ is coerced to a (sub)setoid $B : \text{Setoid}$, with:

- ✓ carrier subtype $B \quad (= \Sigma a : A. (B \ a))$
- ✓ equality $x ='_B x' \text{ in } B \xLeftrightarrow{\text{def}} (\text{subtype\_elt } x) ='_A (\text{subtype\_elt } x') \text{ in } A$
(i.e. $\langle a, H_a \rangle ='_B \langle a', H_{a'} \rangle \xLeftrightarrow{\text{def}} a ='_A a')$

Coercions make the following well-typed:

$A : \text{Setoid} ; \quad a : A$

$B : \text{part\_set } A ; \quad x : B$

whence $x \equiv \langle a, H_a \rangle$ with $a : A; H_a : P \ a$

Dealing with subsets

✓ type theory:

$$B : (\text{part\_set } A) \iff B : \text{Setoid} = \langle \Sigma a : A. P(a), \dots, \dots \rangle$$

$$x : B \iff x = \langle a, H_a \rangle$$

$$x : B \quad f : A \rightarrow Z \rightsquigarrow (f \ x) \quad \textcolor{red}{\text{⚡⚡⚡}}$$

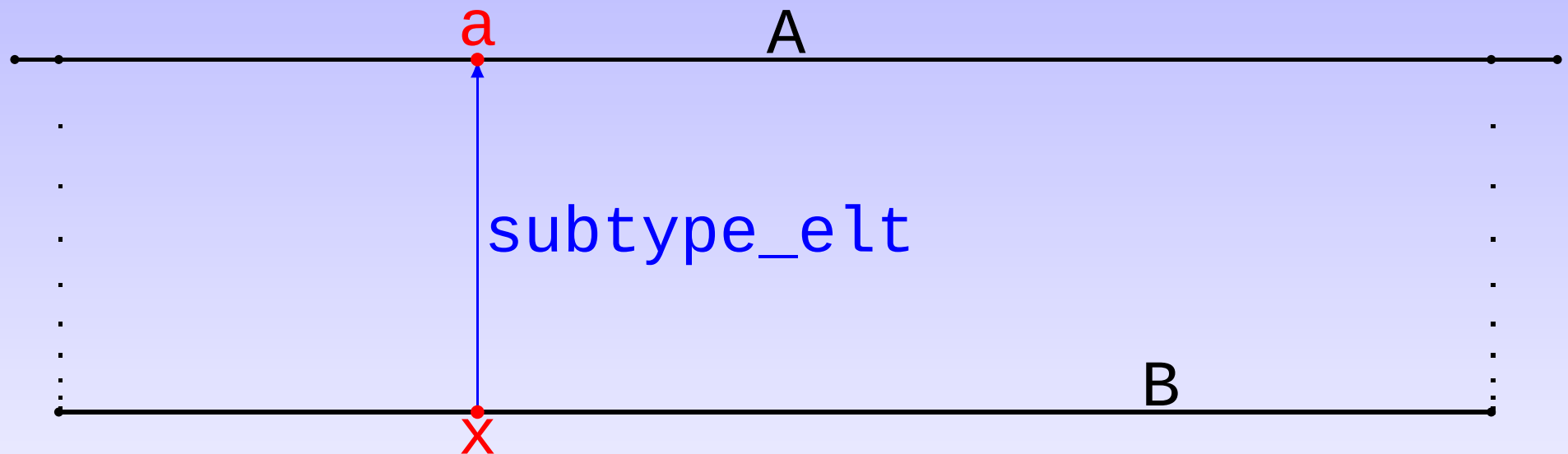
✓ Compare with set theory:

$$\begin{array}{lcl} B \subset A & \iff & \forall x \in B. x \in A \\ a \in B \quad f : A \rightarrow Z & \rightsquigarrow & f(a) \in Z \quad \textcolor{green}{OK} \end{array}$$

We need `subtype_elt` = π_1 as a coercion

Dealing with subsets

- ✓ $x \equiv \langle a, H_a \rangle$
- ✓ x and `suptype_elt x` both represent a
- ✓ `subtype_elt x` $\equiv a$ lives “one level higher”:



Map_embed

To say “ X is linearly dependent” we need:

$$\left. \begin{array}{l} X \subset V \\ \langle a_0, \dots, a_{n-1} \rangle \in \mathbb{F}^n \\ \langle v_0, \dots, v_{n-1} \rangle \in X^n \end{array} \right| \begin{array}{l} X : \text{part\_set } V \\ a : (\text{seq } n \text{ } \mathbb{F}) \\ v : (\text{seq } n \text{ } X) \end{array}$$

for which

$$\sum_{i < n} a_i \cdot v_i = 0_V \quad \text{OK}$$

however:

$$(\text{sum } (\text{mult\_by\_scalars } a \text{ } v)) \quad \text{⚡⚡⚡}$$

Map_embed

$X : (\text{part\_set } V)$ is an *ordinary setoid*:

- ✓ $a : (\text{seq } n \ F); v : (\text{seq } n \ X)$ is OK, but
- ✓ no multiplication ($a_i \ mX \ v_i$ is undefined)
- ✓ no addition (even “just” $\text{sum } v$ is undefined)
- ✓ we must apply `subtype_elt` to each v_i

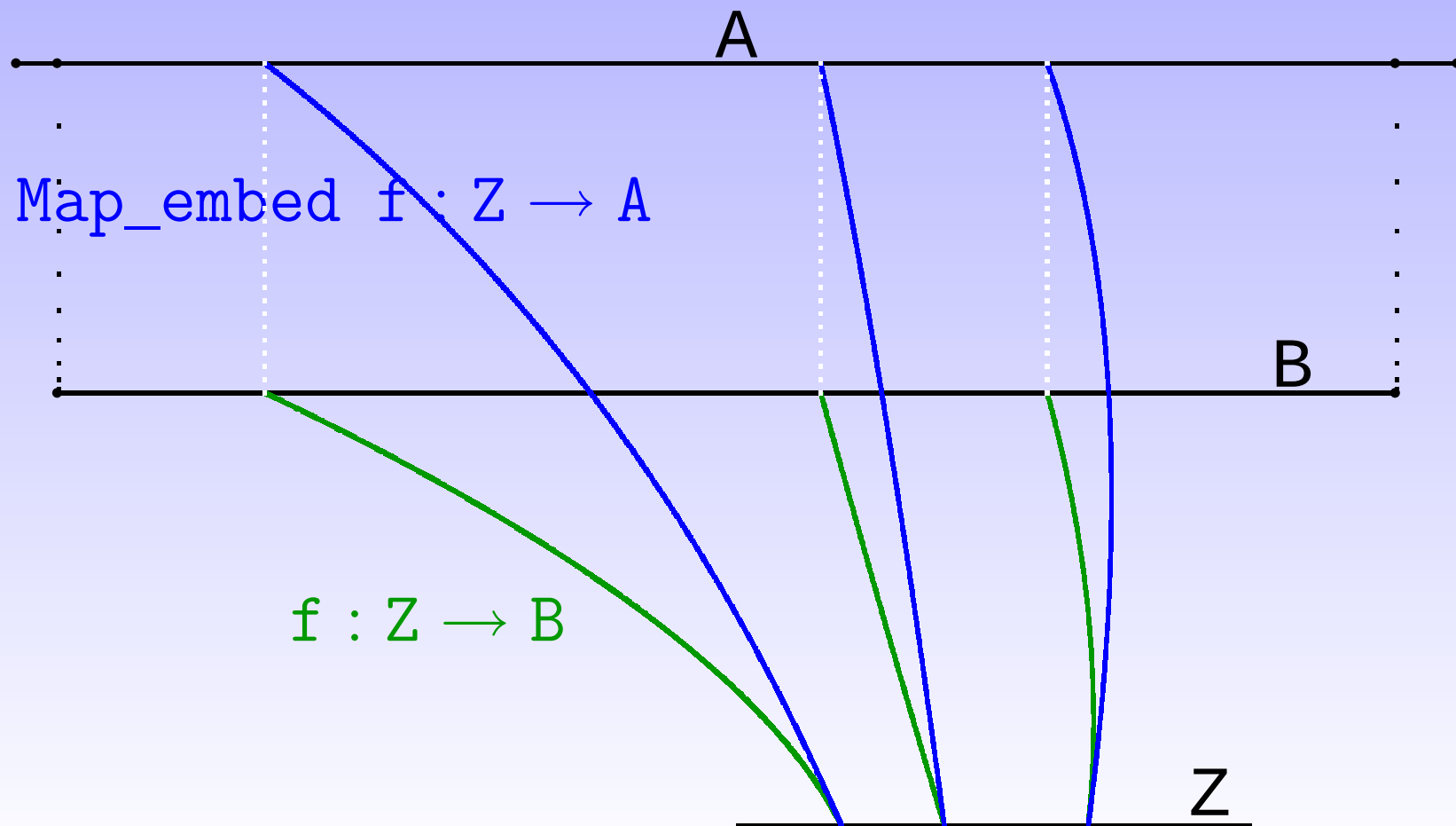
More generally, whenever $B : (\text{part\_set } A)$,

$$\text{Map\_embed} : (\text{Map } Z \ B) \rightarrow (\text{Map } Z \ A)$$

$$(\text{sum } (\text{mult\_by\_scalars } a \ (\text{Map\_embed } v))) \quad \text{OK}$$

Map_embed

- ✓ Both f and $\text{Map\_embed } f$ represent f
- ✓ $\text{Map\_embed } f$ lives “one level higher”



inject_subsets

Theorem 1.9: let $W \subset V$. If W generates V , then $\exists W_0 \subset W$ that is a basis for V

Definition $\text{is\_basis } (X : \text{part\_set } V) :=$
 $\text{generates } X \text{ (full } V) \wedge \text{lin\_indep } X$

$W : (\text{part\_set } V)$

$W_0 : (\text{part\_set } W)$

$(\text{is\_basis } W_0)$ ⚡⚡⚡

W_0 lives 2 levels below V , not 1.

inject_subsets

$$\begin{aligned} w : W_0 &\rightsquigarrow (\text{subtype\_elt } w) : W \\ &\rightsquigarrow (\text{subtype\_elt } (\text{subtype\_elt } w)) : V \end{aligned}$$

These form a subset of V corresponding to W_0 .

Generally, for $B : (\text{part\_set } A)$,

`inject_subsets :`

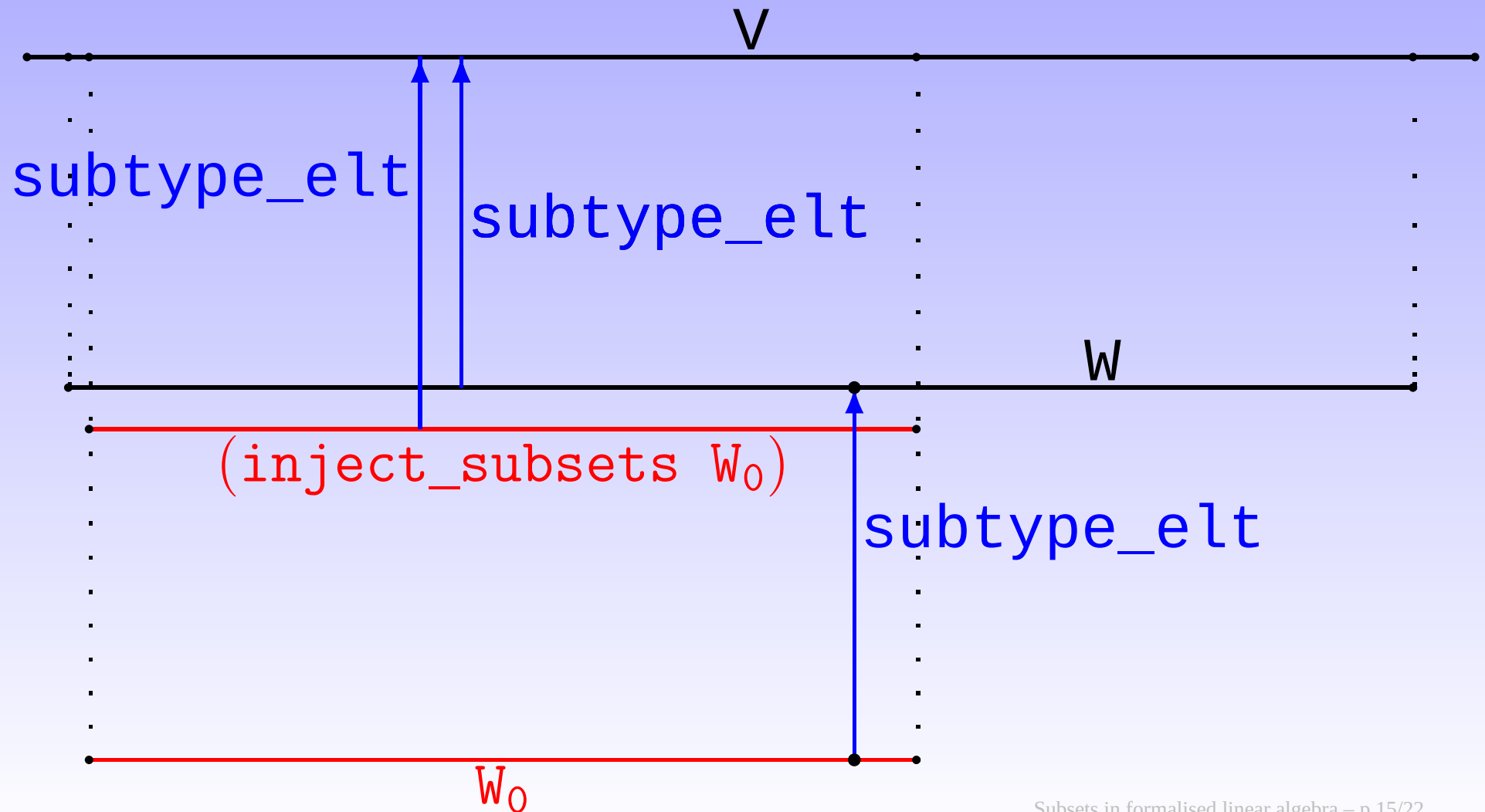
$$(\text{part\_set } B) \rightarrow (\text{part\_set } A)$$

`(inject_subsets C)` is the Predicate based on

$$\lambda a : A. [\exists c : C. a = ' (\text{subtype\_elt } (\text{subtype\_elt } c))]$$

inject_subsets

W_0 and $\text{inject\_subsets } W_0$ both represent W_0
 $\text{inject\_subsets } W_0$ lives “one level higher”:



(un) inject_subsetsify

Let W be a subspace of V . Suppose $X \subset W$.
Then X is linearly dependent in W iff it is in V .

Lemma inject_subsets_lin_dep :

forall (W : subspace V) (X : part_set W),
lin_dep X $\iff$ lin_dep (inject_subsets X).

lin_dep X $\rightsquigarrow$ v : (seq n X)

we need : v : (seq n (inject_subsets X))

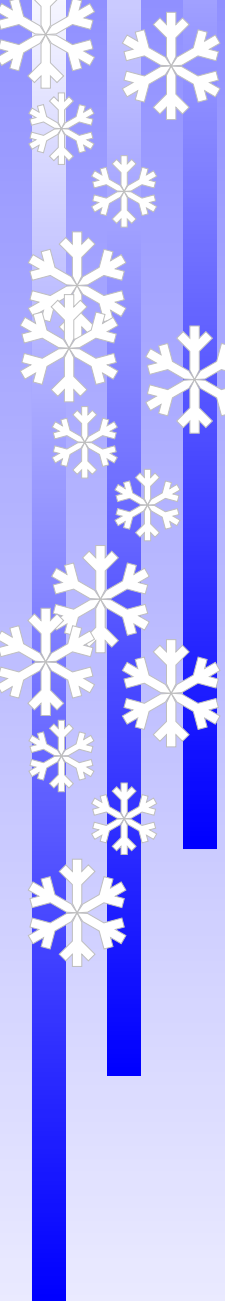
(inject_subsets X) represents the same X as X , one level higher.

(un) inject_subsetsify

✓ we have $v : (\text{seq } n \rightarrow X)$

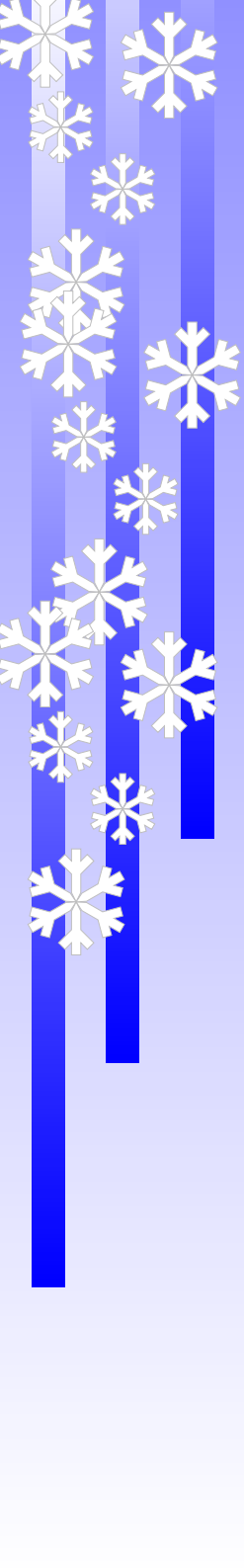
(un) inject_subsetsify

- ✓ we have $v : (\text{seq } n \ X)$
- ✓ $v : (\text{seq } n \ (\text{inject\_subsets } X))$ needed.



(un) inject_subsetsify

- ✓ we have $v : (\text{seq } n \ X)$
- ✓ $v : (\text{seq } n \ (\text{inject\_subsets } X))$ needed.
- ✓ $(\text{inject\_subsets } X)$ is one level above X .

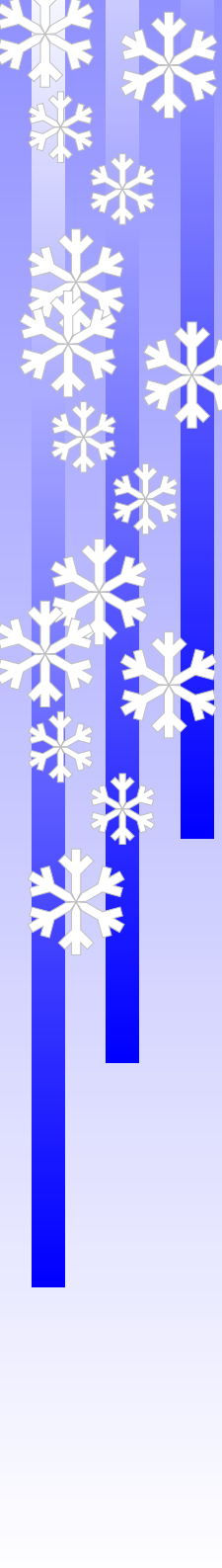


(un) inject_subsetsify

- ✓ we have $v : (\text{seq } n \ X)$
- ✓ $v : (\text{seq } n \ (\text{inject\_subsets } X))$ needed.
- ✓ $(\text{inject\_subsets } X)$ is one level above X .
- ✓ `Map_embed` lifts sequences one level.

(un) inject_subsetsify

- ✓ we have $v : (\text{seq } n \ X)$
- ✓ $v : (\text{seq } n \ (\text{inject\_subsets } X))$ needed.
- ✓ $(\text{inject\_subsets } X)$ is one level above X .
- ✓ Map_embed lifts sequences one level.
- ✓ $\text{Map\_embed } v : (\text{seq } n \ W)$ *wrong!*



(un) inject_subsetsify

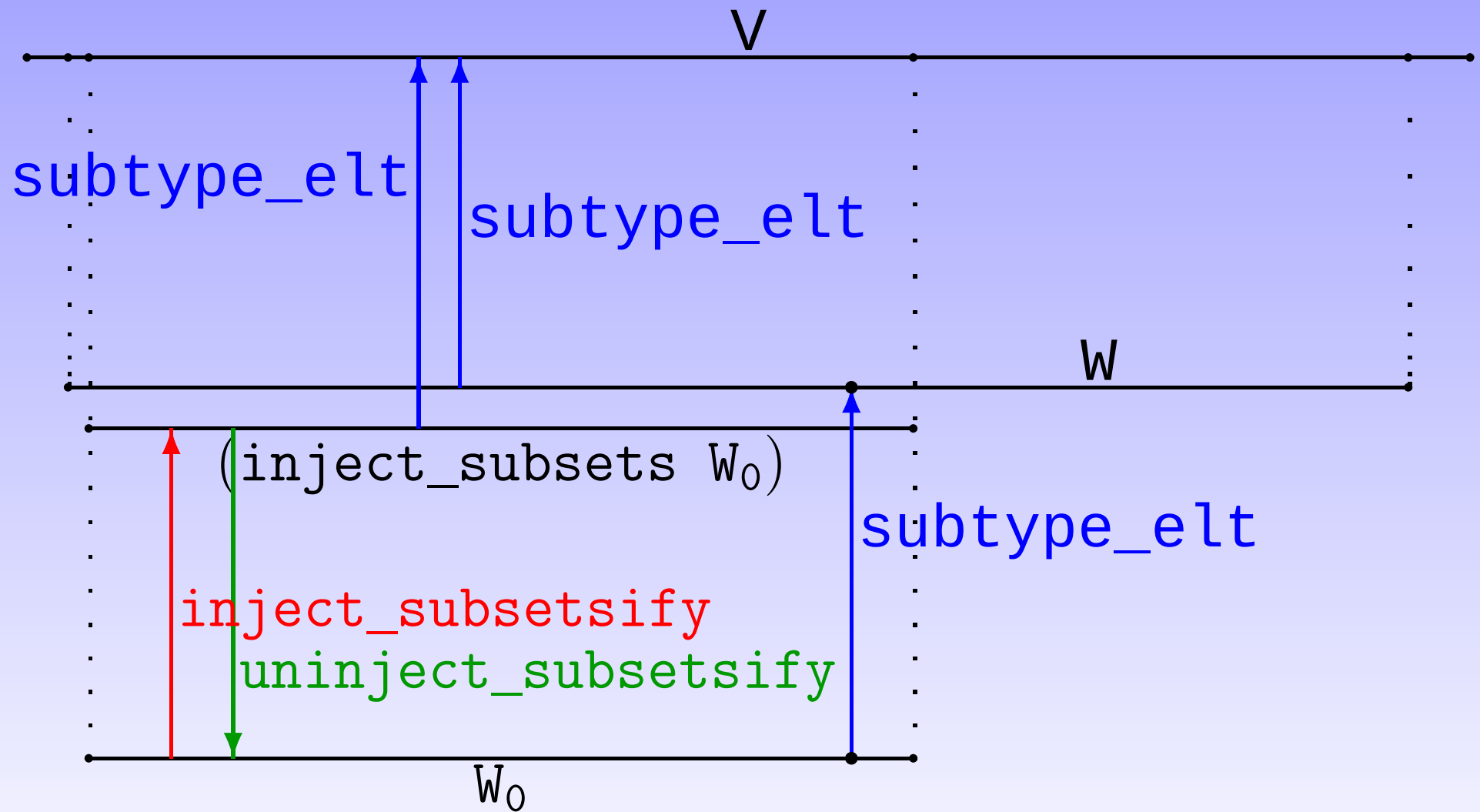
- ✓ we have $v : (\text{seq } n \ X)$
- ✓ $v : (\text{seq } n \ (\text{inject\_subsets } X))$ needed.
- ✓ $(\text{inject\_subsets } X)$ is one level above X .
- ✓ Map_embed lifts sequences one level.
- ✓ $\text{Map\_embed } v : (\text{seq } n \ W)$ *wrong!*

new function:

$\text{inject\_subsetsify} : W_0 \rightarrow (\text{inject\_subsets } W_0)$

plus its inverse $\text{uninject\_subsetsify}$

$(un)inject\_subsetsify$



map_{between|to}_equal_subsets

Subsets may be given in different guises:

✓ $X_1 := (\text{span } W) \quad \vee. \quad X_2 := (\text{span } (\text{span } W))$

✓ $X_1 := \{a, b, c\} \quad \vee. \quad X_2 := \text{the set of } \langle a, c, c, b, a, b \rangle$

Corresponding Predicates are different too!

$$x : X_1 \equiv \langle a, H_a \rangle \quad x : X_2 \equiv \langle a, H'_a \rangle$$

where H_a and H'_a have different types.

$$\text{map\_between\_equal\_subsets} : X_1 =^{\prime} X_2 \rightarrow X_1 \rightarrow X_2$$

$$\text{map\_to\_equal\_subsets} : X_1 =^{\prime} X_2 \rightarrow$$

$$(\text{Map } Z \ X_1) \rightarrow (\text{Map } Z \ X_2)$$

Map_include

*Lemma: suppose $X_1 \subset X_2$ both are subsets of V .
Then $\text{span}(X_1) \subset \text{span}(X_2)$*

From $w : (\text{span } (\text{inject\_subsets } X_1))$ we get

$a : (\text{seq } n \text{ } F); \quad v : (\text{seq } n \text{ } (\text{inject\_subsets } X_1))$

that we must turn into

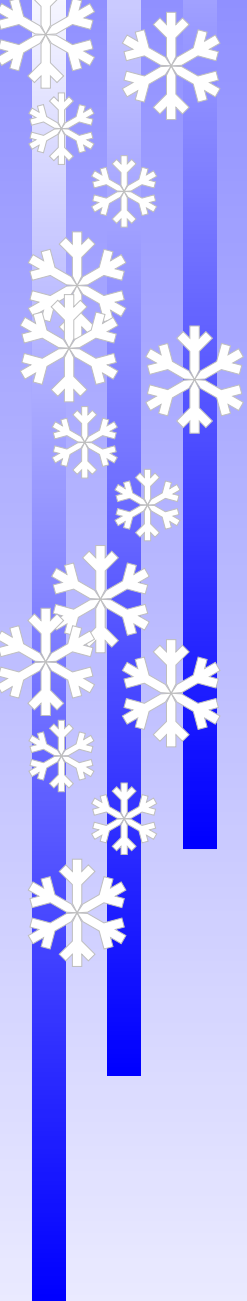
$a : (\text{seq } n \text{ } F); \quad v : (\text{seq } n \text{ } X_2)$

$\text{Map\_include} : (\text{included } X_1 \text{ } X_2) \rightarrow$

$(\text{Map } Z \text{ } X_1) \rightarrow (\text{Map } Z \text{ } X_2)$

Conclusions

- ✓ In math, vectors can be members of several subsets besides the space itself
- ✓ Due to Coq's unique typing, the setoid formalism can accomodate subsets only so well
- ✓ Mathematical objects (vectors, sequences, ...) need several representations
- ✓ In **LinAlg** I used 8 different “pure type-cast” operations
- ✓ Even the subset relation $\subset$ is formalised in 2 different ways



The End

Thank you!

Questions?

Author: Georges Gonthier

E-mail: gonthier@microsoft.com

Title: The Four Color Theorem in Coq

Abstract:

This fall we completed a fully formal proof of the famous Four Color Theorem, using the Coq proof assistant, thus bringing a 5-year old side project to a close. In this talk we will review the main obstacles of this formal proof, and present the main techniques that were used to overcome them, in particular a general and pervasive use of computational reflection, and a simple tactic "shell" that very effectively simplifies proof context management.

Author: Lucas Dixon

Title: IsaPlanner: A proof planning framework for Isabelle

Abstract:

We introduce our work on the development of a proof planner for Isabelle and highlight some recent developments. The central aim of this work is to improve automation in Isabelle by employing techniques, such as rippling, from the paradigm of proof planning. A second goal of our work is to use the correspondence between proof plans and Isar proof scripts to provide tools for the automatic generation and manipulation of proof scripts.

IsaPlanner: A Proof Planner for Isabelle

Lucas Dixon

December 17, 2004



Motivation

- Improve automation in Isabelle by using tools from proof planning
- A generic approach?
- Take advantage of the proof tools available in Isabelle
- Expressing the reasoning process and the resulting proof plans in a *readable* form (Surveyable proofs)
- Proof planning to aid the interactive user

IsaPlanner Approach to Proof Planning

- Encode Techniques: capture common patterns of reasoning
- Applying a technique generates high level descriptions of the proofs called *proof plans* (a representation of proof scripts)
- Our approach derives Isar style proof scripts.
- Rippling: a difference removal technique, principally used to guide rewriting of the step case during inductive proofs

IsaPlanner

- A framework for encoding and applying reasoning techniques that derives readable proof scripts.
- Mechanisms for sharing non-logical information between proof techniques.
- Applying proof techniques produces Isar proof scripts
 - Allow human inspection of the proof attempt
 - Support debugging of proof
 - Support the exploration of proof
- Provide well-behaved techniques

Hierarchy of Behaviour

1. Bug
2. Non termination
3. Uninformative result (big goals, exceptions, "no")
4. Informative result (meaningful progress)
5. Proof/Refutation

Example Generated Proof Script

```
theorem sum_of_odds:  $(\sum_{i < n} 2 * i + 1) = n^2$   
  proof (induct n)  
    show  $(\sum_{i < 0} 2 * i + 1) = 0^2$  by simp  
  next  
    fix n  
    assume IH:  $(\sum_{i < n} 2 * i + 1) = n^2$   
    have  $(\sum_{i < n} 2 * i + 1) + ((2 * n) + 1) = (n + 1)^2$   
      gap (conjecture_lemma  $n^2 + ((2 * n) + 1) = (n + 1)^2$ )  
    thus  $(\sum_{i < (n + 1)} 2 * i + 1) = (n + 1)^2$  by rippling  
  qed
```

Example Generated Proof Script

```

theorem sum_of_odds:  $(\sum_{i < n} 2 * i + 1) = n^2$ 
proof (induct n)
  show  $(\sum_{i < 0} 2 * i + 1) = 0^2$  by simp
next
  fix n
  assume IH:  $(\sum_{i < n} 2 * i + 1) = n^2$ 
  have  $(\sum_{i < n} 2 * i + 1) + ((2 * n) + 1) = (n + 1)^2$ 
    gap (conjecture_lemma  $n^2 + ((2 * n) + 1) = (n + 1)^2$ )
  thus  $(\sum_{i < (n + 1)} 2 * i + 1) = (n + 1)^2$  by rippling
qed

```

Proof Planning?

- Declarative description (datatype) of Isar proof scripts
- Compared with tactics:
 - More information - history of the proof
 - Surveyable proofs
- Mix human/computer manipulation of 'readable' formal proof

Encoded Techniques

- Induction, Case splitting, Equational Reasoning
- Higher order rippling (provides rewriting)
- Lemma speculation and generalisation critics
- This combination forms a powerful inductive theorem prover

Some Observations

- Supports *exploration* of Isar proofs
- Can prove some apparently complicated problems, for example: $(x^y)^z = x^{(y \cdot z)}$ in ordinal arithmetic.
- Conjecturing lemmas is a directed form of theory formation
- Unsuccessful branches/failed proof attempts can still result in proving useful lemmas
- Caching conjectures (proved and unprovable) prunes the search space significantly

Some Issues

- Where to draw the line between work done by a tactic and work done by proof planning?
- What information to store in source files for proof re-execution?
- Interface issues: how to use support interactive proof planning? (Proof by contextual menu!)

Author: Pierre Corbineau

Title: First-order reasoning in the calculus of inductive constructions

Abstract:

We present a contraction-free sequent calculus including inductive definitions for the first-order intuitionistic logic. We show that it is a natural extension to Dyckhoff's LJT calculus and we prove the contraction- and cut-elimination properties, thus extending Dyckhoff's result, in order to validate its use as a basis for proof-search procedures. Finally we describe the proof-search strategy used in our implementation as a tactic in the Coq proof assistant.

Author: Marc Bezem

Title: Geometric Logic and Proof Objects

Abstract:

Geometric logic (GL) is the logic preserved by geometric morphisms between topoi. The first-order fragment of GL extends resolution logic in that geometric clauses can have existentially quantified conclusions. A substantial number of reasoning problems (e.g., in confluence theory, lattice theory and projective geometry) can be formulated directly in GL without any skolemization. Thus no skolem axioms are necessary and the domain of discourse can be kept finite. These advantages can outweigh the disadvantage that a more expressive logic is generally more difficult to implement than a simpler one. We explore the balance between the advantages and disadvantages with an implementation in Prolog of (incomplete) proof search in GL including the generation of proof objects. If a proof has been obtained it can be verified directly in Coq.

A depth-first implementation of Geometric Logic in Prolog

(with proof objects)

Marc Bezem
University of Bergen

GL as a fragment of FOL

- Geometric formula: $C \Rightarrow D$
- $C = A_1 \wedge \dots \wedge A_n$ ($n \geq 0$, A_i atoms)
- $D = E_1 \vee \dots \vee E_m$ ($m \geq 0$)
- $E_j = (E \ x_1 \dots x_k) C_j$ ($k \geq 0$, C_j like C)
- Implicit universal closure
- No function symbols (yet), only constants

Examples

- Lattices (meet is associative, Horn clause):
$$x \cap y = u \wedge u \cap z = v \wedge y \cap z = w \Rightarrow x \cap w = u$$
- Projective unicity (resolution clause):
$$p|l \wedge p|m \wedge q|l \wedge q|m \Rightarrow p=q \vee l=m$$
- Diamond property (geometric clause):
$$a \rightarrow b \wedge a \rightarrow c \Rightarrow (\exists d) (b \rightarrow d \wedge c \rightarrow d)$$
- In general: $A_1 \wedge \dots \wedge A_n \Rightarrow$
$$((\exists \mathbf{x}) A_{11} \wedge \dots \wedge A_{1i}) \vee \dots \vee ((\exists \mathbf{y}) A_{k1} \wedge \dots \wedge A_{kj})$$

Rationale

- Less skolemization
- Direct proofs
- Constructive logic
- Natural proof theory/objects

Inductive definition of $X \vdash_{(T)} D$

- (base) $X \vdash D$ if $X \downarrow D$
- (step)
$$\frac{X, \underline{C_1} \vdash D, \dots, X, \underline{C_n} \vdash D}{X \vdash D}$$

X a finite set of facts (= closed atoms)

D closed geometric disjunction (parameters in D must occur in X)

$X \downarrow D$ iff $D = \dots \vee (E \mathbf{x}) C \vee \dots$ and X contains all facts in $C[\mathbf{x}:=\mathbf{a}]$
for suitable

parameters $\mathbf{a}$

- () there exists a *closed* instance $C_0 \Rightarrow D_0$ of an axiom in T with
 C_0 *included in* X (X contains all facts in C_0) and
 $D_0 = \dots \vee (E \mathbf{x}) C_i \vee \dots$ and each $\underline{C_i}$ a *fresh* instance of C_i
 $(1 \leq i \leq n)$

Metaproperties

- Soundness
- Completeness
- Constructivity
- Conservativity
- Semidecidability
- Automation (SATCHMO!)

Examples

- exist.in
- or.in
- nijm.in

Case studies

- Confluence theory: induction steps in Newman's Lemma, Hindley-Rosen, Self-lengthening Thm, ..
- Lattice theory: $x \cap (y \cup z) \leq (x \cap y) \cup (x \cap z)$ for all x, y, z implies $(x \cup y) \cap (x \cup z) \leq x \cup (y \cap z)$ for all x, y, z
- Projective geometry: equivalence of two versions of Pappus' Axiom (1 minute, 1MB proof)

Author: Robin Adams

Title: A Formalisation of the Metatheory of Pure Type Systems in Coq

Abstract:

The idea of representing the syntax of a language with variable binding as a nested data type --- defining the type "term V " of terms which use the objects of type V as free variables --- is not new, but seems never to have been applied to the formalisation of results in the metatheory of a type theory. I shall present a formalisation in Coq of several results in the theory of Pure Type Systems, including the quite difficult proof of the strengthening lemma, based on this approach to syntax. I hope to show that it compares favourably in many ways with previous approaches to the formalised metatheory of a type theory.

A Formalisation of the Metatheory of Pure Type Systems in Coq

Robin Adams
Department of Computer Science
University of Manchester

Pure Type Systems

- A general theory covering a large number of type systems.
- A generalization of the lambda cube.
- Invented independently by Berardi and Terlouw in 1989.

H. Barendregt, 1992. "Lambda Calculi with Types". In Abramsky, Gabbat and Maibaum (eds.): *Handbook of Logic in Computer Science*, Vol II. Oxford University Press.

Pure Type Systems

A *specification* of a Pure Type System (PTS) consists of:

- a set $\mathcal{S}$ of *sorts*;
- a set $\mathcal{A} \subseteq \mathcal{S}^2$ of *axioms*;
- a set $\mathcal{R} \subseteq \mathcal{S}^3$ of *rules*.

The set of *terms* is given by the grammar

$$\text{Term } M ::= x \mid s \mid MM \mid \Pi x : M.M \mid \lambda x : M.M$$

A *judgement* has the form

$$x_1 : A_1, \dots, x_n : A_n \vdash M : B$$

The derivable judgements are given by seven rules of deduction.

Theorem

A PTS has the *Uniqueness of Types* property:

$$\Gamma \vdash M : A, \Gamma \vdash M : B \Rightarrow A \simeq_\beta B$$

iff the specification is *functional*:

$$\begin{aligned} (s, t) \in \mathcal{A}, (s, t') \in \mathcal{A} &\Rightarrow t = t' \\ (s_1, s_2, s_3) \in \mathcal{R}, (s_1, s_2, s'_3) \in \mathcal{R} &\Rightarrow s_3 = s'_3 \end{aligned}$$

Other Approaches to Formalization

De Bruijn Indices

Bruno Barras. *Coq in Coq*

Bruno Barras, 1999. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7

- Uses `nat` for free and bound variables.
- α -convertible terms become equal.
- Many technical lemmas about lifting and substitution operations needed.

Parameters and Bound Variables

James McKinna and Robert Pollack, 1999. *Some Lambda Calculus and Type Theory Formalized*. *Journal of Automated Reasoning* 23 (3–4), pp. 373–409.

- Uses separate types P and V for parameters (free variables) and bound variables.
- Very close to implementation.
- Many technicalities — such as variant induction principles for some relations — needed.

Terms as a Nested Datatype

Francoise Bellegarde and James Hook, 1994. *Substitution: A formal methods case study using monads and transformations*. Science of Computer Programming 23 (2–3), pp. 287–311.

Define the inductive family $\mathcal{T}$

$(V : \text{Type}) \mathcal{T}_V : \text{Type}$

with constructors

$$\begin{array}{c} \frac{x : V}{\text{var } x : \mathcal{T}_V} \qquad \frac{s : \mathcal{S}}{\text{srt } s : \mathcal{T}_V} \qquad \frac{M : \mathcal{T}_V \quad N : \mathcal{T}_V}{MN : \mathcal{T}_V} \\[2ex] \frac{A : \mathcal{T}_V \quad B : \mathcal{T}_{V_\perp}}{\Pi AB : \mathcal{T}_V} \qquad \frac{A : \mathcal{T}_V \quad M : \mathcal{T}_{V_\perp}}{\lambda AM : \mathcal{T}_V} \end{array}$$

$\mathcal{T}_V$ is the type of terms formed using the objects of V as free variables.

```
Inductive term : Set -> Type :=
var : (V : Set) V -> term V |
srt : (V : Set) sort -> term V |
app : (V : Set) term V -> term V -> term V |
pi  : (V : Set) term V -> term (maybe V) -> term V |
lda : (V : Set) term V -> term (maybe V) -> term V.
```

Nat-indexed Family of Terms

Define an inductive family $\mathcal{T}$

$$(n : \mathbb{N}) \ \mathcal{T}_n : \text{Set}$$

with constructors

$$\begin{array}{c} \frac{x : \mathcal{F}_n}{\text{var } x : \mathcal{T}_n} \qquad \frac{s : \mathcal{S}}{\text{srt } s : \mathcal{T}_n} \qquad \frac{M : \mathcal{T}_n \quad N : \mathcal{T}_n}{MN : \mathcal{T}_n} \\[10pt] \frac{A : \mathcal{T}_n \quad B : \mathcal{T}_{n+1}}{\Pi AB : \mathcal{T}_n} \qquad \frac{A : \mathcal{T}_n \quad M : \mathcal{T}_{n+1}}{\lambda AM : \mathcal{T}_n} \end{array}$$

where $\mathcal{F}_n$ is a type with n objects.

$\mathcal{T}_n$ is the type of terms formed using the objects of $\mathcal{F}_n$ as free variables; i.e. the type of terms with at most n free variables.

```
Inductive term : nat -> Set :=
var : (n : nat) fin n -> term n |
srt : (n : nat) sort -> term n |
app : (n : nat) term n -> term n -> term n |
pi  : (n : nat) term n -> term (S n) -> term n |
lda : (n : nat) term n -> term (S n) -> term n.
```


Substitution on the Nested Datatype

We would like to define

$$(M : \mathcal{T}_U, \rho : U \rightarrow \mathcal{T}_V) \quad M[\rho] : \mathcal{T}_V$$

as follows:

$$\begin{aligned} (\text{var } x)[\rho] &= \rho x \\ (\text{srt } s)[\rho] &= \text{srt } s \\ (MN)[\rho] &= (M[\rho])(N[\rho]) \\ (\Pi AB)[\rho] &= \Pi(A[\rho]) \left(B \left[\begin{array}{cc} \perp & \mapsto \perp \\ \uparrow x & \mapsto (\rho x)[\uparrow] \end{array} \right] \right) \\ (\lambda AM)[\rho] &= \lambda(A[\rho]) \left(M \left[\begin{array}{cc} \perp & \mapsto \perp \\ \uparrow x & \mapsto (\rho x)[\uparrow] \end{array} \right] \right) \end{aligned}$$

The recursion can be shown to terminate.

Observe:

- $\uparrow$ maps variables to variables.
- If ρ maps variables to variables, so does

$$\begin{array}{ccc} \perp & \mapsto & \perp \\ \uparrow x & \mapsto & (\rho x)[\uparrow] \end{array}$$

Replacement — Substituting Variables for Variables

Define

$$(M : \mathcal{T}_U, \rho : U \rightarrow V) \quad M\{\rho\} : \mathcal{T}_V$$

the result of replacing each variable x with ρx throughout M .

The definition is by *structural recursion* on M as follows:

$$(\text{var } x)\{\rho\} = \text{var}(\rho x)$$

$$(\text{srt } s)\{\rho\} = \text{srt } s$$

$$(MN)\{\rho\} = (M\{\rho\})(N\{\rho\})$$

$$(\Pi AB)\{\rho\} = \Pi(A\{\rho\}) \left(B \left\{ \begin{array}{cc} \perp & \mapsto \perp \\ \uparrow x & \mapsto \uparrow(\rho x) \end{array} \right\} \right)$$

$$(\lambda AM)\{\rho\} = \lambda(A\{\rho\}) \left(M \left\{ \begin{array}{cc} \perp & \mapsto \perp \\ \uparrow x & \mapsto \uparrow(\rho x) \end{array} \right\} \right)$$

Substitution Defined Using Structural Recursion

We can now define

$$(M : \mathcal{T}_U, \rho : U \rightarrow \mathcal{T}_V) \ M[\rho] : \mathcal{T}_V$$

by structural recursion as follows:

$$\begin{aligned} (\text{var } x)[\rho] &= \rho x \\ (\text{srt } s)[\rho] &= \text{srt } s \\ (MN)[\rho] &= (M[\rho])(N[\rho]) \\ (\Pi AB)[\rho] &= \Pi(A[\rho]) \left(B \left[\begin{array}{ccc} \perp & \mapsto & \perp \\ \uparrow x & \mapsto & (\rho x)\{\uparrow\} \end{array} \right] \right) \\ (\lambda AM)[\rho] &= \lambda(A[\rho]) \left(M \left[\begin{array}{ccc} \perp & \mapsto & \perp \\ \uparrow x & \mapsto & (\rho x)\{\uparrow\} \end{array} \right] \right) \end{aligned}$$

We can proceed to define β -reduction and β -convertibility, prove the Church-Rosser Theorem, etc.

Grammar Using the Nested Datatype

We define

$$(M : \mathcal{T}_U, \rho : U \rightarrow V) \ M\{\rho\} : \mathcal{T}_V$$

the result of replacing x with the variable ρx throughout M ;

$$(M : \mathcal{T}_U, \rho : U \rightarrow \mathcal{T}_V) \ M[\rho] : \mathcal{T}_V$$

the result of substituting the term ρx for x throughout M .

We prove the various forms of the Substitution Lemma:

$$\begin{aligned} M\{\rho\}\{\sigma\} &= M\{\sigma \circ \rho\} \\ M\{\rho\}[\sigma] &= M[\sigma \circ \rho] \\ M[\rho]\{\sigma\} &= M[x \mapsto (\rho x)\{\sigma\}] \\ M[\rho][\sigma] &= M[x \mapsto (\rho x)[\sigma]] \end{aligned}$$

Richard S. Bird and Ross Paterson, 1999. *De Bruijn Notation as a Nested Datatype*. J. Functional Programming 9 (1): 77–91.

Thorsten Altenkirch and Bernhard Reus, 1999. *Monadic Presentations of Lambda Terms using Generalized Inductive Types*. CSL '99, LNCS 1683, pp. 453–468.

Rules of Deduction of a PTS

Define the type $\mathcal{C}_n$ of *contexts of length n* :

$$\begin{aligned}\mathcal{C}_0 &= \{*\} \\ \mathcal{C}_{n+1} &= \mathcal{C}_n \times \mathcal{T}_n\end{aligned}$$

Now we define the relation `Inductive PTS : (n : nat) ctxt n -> term n -> term n -> Prop` := by simply writing down the rules of deduction of a PTS:

$$\text{(axioms)} \frac{}{\vdash s : t} (s : t \in \mathcal{A})$$

```
axioms : (s t : sort) ax s t ->
PTS 0 tt (srt s) (srt t)
```

$$\text{(weak)} \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} (x \notin \text{dom } \Gamma)$$

```
weak : (n : nat) (Gamma : ctxt n) (A : term n)
(s : sort)
PTS n Gamma A (srt s) ->
PTS (S n) (Gamma, A) (var bot) (lift A)
```

etc.

“Big-Step” Weakening and Substitution Lemmas

For $\Gamma : \mathcal{C}_m$, $\Delta : \mathcal{C}_n$, define $\Gamma \subseteq_\rho \Delta$ to be

$$(\forall x : \mathcal{F}_m) \Delta_{\rho x} \equiv \Gamma_x \{\rho\}$$

where Γ_x is the type assigned to x by the context Γ .

Weakening Lemma If $\Gamma \subseteq_\rho \Delta$, $\Gamma \vdash M : A$, and Δ is a valid context, then $\Delta \vdash M\{\rho\} : A\{\rho\}$.

Substitution Lemma Let $\Gamma : \mathcal{C}_m$, $\rho : \mathcal{F}_m \rightarrow \mathcal{T}_n$, $\Delta : \mathcal{C}_n$. If $\Gamma \vdash M : A$ and

$$(\forall x : \mathcal{F}_m) \Delta \vdash \rho x : \Gamma_x[\rho]$$

then

$$\Delta \vdash M[\rho] : A[\rho].$$

We proceed to prove Generation Lemmas, Subject Reduction, and Uniqueness of Types for functional PTSs.

Conclusion

Defining terms as a nested datatype is useful when formalizing the metatheory of a type theory:

- α -convertible terms become equal.
- “Big-step” definitions can be made, and “big-step” theorems proved.
- Technicalities are few, and have a type-theoretic flavour.

Coq scripts available at:

<http://www.cs.man.ac.uk/~radams/coqPTS>

Author: Sebastian Skalberg

Title: Porting Proofs Between HOL Implementations

Abstract:

Many of the current HOL implementations (HOL, Isabelle/HOL, HOL Light) have formulations of HOL that are similar enough to facilitate a straightforward port of proof objects from one system to another. In the talk, I will focus on the addition of proof objects to HOL98 and the subsequent automatic porting of over 2000 theorems from that system to Isabelle/HOL.

Porting Proofs Between HOL Implementations

Sebastian Skalberg

skalberg@in.tum.de

Technische Universität München

Overview

- Motivation
- Recording and saving proofs
- Replaying proofs
- Conclusion

Motivation: HOL4 vs. Isabelle/HOL

HOL4:

- Industrial strength.
- Massive formalization of real analysis.
- Primitive user interface.

Isabelle/HOL:

- Massive formalization of Java.
- Very accessible user interface.
- Not many results in real analysis.

Porting Theorems

- Oracles
 - Little workload
 - Not safe

Porting Theorems

- Oracles
 - Little workload
 - Not safe
- Port proof scripts
 - New system has all functionality from original system
 - Completely safe
 - Large workload
 - Must be done for each new proof tactic

Porting Theorems

- Oracles
 - Little workload
 - Not safe
- Port proof scripts
 - New system has all functionality from original system
 - Completely safe
 - Large workload
 - Must be done for each new proof tactic
- Port proofs
 - Medium workload
 - Completely safe
 - Tools not available in new system

Design Goals

Export from HOL:

- The extension should be as unobtrusive as possible, should “clearly” not have an impact on the soundness of the system.
- Proof recording should be reasonably fast.

Import into Isabelle/HOL:

- Only import new types, terms, and theorems.
- Produce valid Isar theories.

Also, proof file format should not be specific to any given ML system.

Concrete Representation

```
datatype proof_info
  = Info of {disk_info: (string * string) option ref,
             references: int ref}

datatype proof = Proof of proof_info * proof_content
               and proof_content
               = PRefl of term
               | PInstT of proof * (hol_type, hol_type) Lib.subst
               | PSubst of proof list * term * proof
               | PAbs of proof * term
               | PDisch of proof * term
               | PMp of proof * proof
               | PHyp of term
               | PAXm of string * term
               | ...

datatype thm = THM of tag * term HOLset.set * term * proof
```


Efficiency

Due to their size, several problems present themselves once we want to write proofs to disk:

- Proofs, terms, and types shared in memory need to be shared on disk too.
- Even with sharing, efficient I/O is required.

On the other hand, we do not need to

- remove types (variables may differ in only their type).
- compress terms.
- normalize proofs (they are first-order).

The Poincaré Principle

It has been argued that proof objects for proofs in classical logics (as opposed to constructive type theories) would be infeasible due to the explicit proofs of β -convertibility. As with other practical issues, this is a question of trading the complexity of proof checkers for the size of proof objects.

	logical kernel	HOL4 kernel	proof objects
matching modulo inference rules	α 8	α 42	$\beta\eta$ 33

Importing in Isabelle/HOL

The overall strategy is to implement an HOL4 kernel in Isabelle/HOL. Some of the technical problems encountered were:

- All variables occurring in an Isabelle term must have either different names or the same type, not so in HOL4.
- Isabelle has a logical level more than HOL4.
- Printing of terms in Isabelle, so that Isabelle could parse it back.

Informing Isabelle/HOL

Many type and term constants of HOL4 are already present in Isabelle/HOL, as well as many theorems. To minimize the import, we make sure that the Isabelle variants are used whenever possible.

- For type and term constants, this means proving the relevant abstract properties of the constant, and telling Isabelle about the renaming.
- For theorems, we make an exhaustive search of the Isabelle/HOL theorem data base for “equivalent” theorems.

The Actual Import

The import has two stages:

- First, Isabelle takes configuration information and proof objects, and spits out a number of Isar theories.
- Second, the new Isar theories are processed like any other in Isabelle/HOL.

Example Configuration Section

```
import_theory sum
```

```
type_maps  
  sum > "+"
```

```
const_maps  
  INL      > Inl  
  INR      > Inr  
  ISL      > HOL4Compat.ISL  
  ISR      > HOL4Compat.ISR  
  OUTL     > HOL4Compat.OUTL  
  OUTR     > HOL4Compat.OUTR  
  sum_case > Datatype.sum.sum_case
```

```
ignore_thms sum_TY_DEF sum_ISO_DEF IS_SUM_REP INL_DEF  
  INR_DEF sum_axiom sum_Axiom
```

```
end_import
```

Example Output

```
setup_theory sum
```

```
lemma ISL_OR_ISR: "ALL x. ISL x | ISR x"  
  by (import sum ISL_OR_ISR)
```

```
lemma INL: "ALL x. ISL x --> Inl (OUTL x) = x"  
  by (import sum INL)
```

```
lemma INR: "ALL x. ISR x --> Inr (OUTR x) = x"  
  by (import sum INR)
```

```
lemma sum_case_cong: "ALL (M::'b + 'c) (M'::'b + 'c) (f::'b => 'a)  
  g::'c => 'a. M = M' &  
  (ALL x::'b. M' = Inl x --> f x = (f'::'b => 'a) x) &  
  (ALL y::'c. M' = Inr y --> g y = (g'::'c => 'a) y) -->  
  sum_case f g M = sum_case f' g' M'"  
  by (import sum sum_case_cong)
```

```
end_setup
```

Empirical Data

Time taken to build the base HOL4 system:

original system	with proof recording	with proof saving
42m	44m	1h32m

The base HOL4 system comprises 5,000 named theorems. These are saved in 80,000 files, taking up 350Mb (13Mb when gzipped).

The generating of Isar theories in Isabelle/HOL takes about 42m, the replaying 1h22m.

Conclusion

- Proof recording has been added to HOL4 and HOL Light (the latter by Steven Obua).
- Replaying of proof objects in Isabelle/HOL is implemented and functional.
- Recording, saving, and replaying of proofs seems acceptably efficient.
- Data sharing is essential, probably more so than compression per se.

Authors: Micaela Mayero, David Delahaye

Title: A Quantifier Elimination Procedure over ACFs in Coq using Maple

Abstract:

We present an interface between the Coq proof assistant and the Maple computer algebra system. The aim of the interface is to import, into Coq, computations over algebraic expressions carried out by Maple. With this interface, it is either possible to perform pure computations, which do not require any validation, or to perform computations inside a proof, which have to be proved correct in Coq (skeptical approach). In the particular case of algebraic expressions over a field, the correctness proofs are fully automatic and handled by the Field tactic.

As an example, we describe a quantifier elimination procedure over algebraically closed fields, where the interface has been extended to deal with polynomial gcds. Thus, this shows that this interface may be used to increase not only the power of computation of Coq but also its automation.

A Quantifier Elimination Procedure over ACFs in Coq using Maple

Micaela Mayero¹ & David Delahaye²

¹LCR-LIPN

Micaela.Mayero@lipn.univ-paris13.fr

<http://www-lipn.univ-paris13.fr/~mayero/>

²CPR-CEDRIC

CNAM (Paris)

David.Delahaye@cnam.fr

<http://cedric.cnam.fr/~delahaye/>

Introduction

We focus on :

- Theorem Proving
- Computer Algebra

Two distinct domains with :

- Their own communities
- Rather few interactions

However, they are quite complementary, especially w.r.t. their *weak points*.

In a Deduction System (DS),

→ hard to perform **efficient computations**.

In a Computer Algebra System (CAS),

→ no notion of **consistency**.

Approaches

CASs $\rightarrow$ dedicated to computations
DSs $\rightarrow$ dedicated to validation
 $\Rightarrow$ make them interact.

Several approaches :

1. To import validation into CASs
2. To import computations into DSs
 - *believing approach*
 - *skeptical approach*
 - *autarkic approach*
3. To build a system with both

Our approach

We focus on the method (2), *skeptical* approach :

→ Interface between Coq and Maple.

Based on the design of the tactic `Field` :

→ To prove *automatically* equalities over fields.

With this interface, we can :

- Use, in Coq, Maple functions over fields
- Prove these computations with `Field`

The computation/proof paradigm ^(1/2)

Pure computations :

- No verification needed
- No knowledge of the imported functions required
- New computational behavior
- Very general (not only for Computer Algebra)

The computation/proof paradigm (2/2)

Validating computations :

- When used inside a proof
- Verified w.r.t. the computation/deduction rules

$$\Gamma \vdash P(t), t' = f_{\text{ext}}(t)$$

$$\frac{\Gamma \vdash t = t' \quad \Gamma \vdash P(t')}{\Gamma \vdash P(t)} \text{ (=Cont)}$$

$t = t'$ must be proved :

- Either using a function g_{prf} s.t. $g_{\text{prf}}(t) = t'$
- Or using the deduction rules
- Either manually (*oracle*)
- Or automatically (*tactic*)

Interfacing Coq and Maple

Pure computations :

`Eval` $\langle$ Maple *function* $\rangle$ `in` $\langle$ Coq *term* $\rangle$

New Coq *tactics* :

$\langle$ Maple *function* $\rangle$ $\langle$ Coq *term* $\rangle$.

Imported Maple functions :

Maple function	Action
<code>simplify</code>	Simplify an expression
<code>factor</code>	Factor a multivariate polynomial
<code>expand</code>	Expand an expression
<code>normal</code>	Normalize a rational expression

Implementation

Term transfer between Coq and Maple
→ Reflexive coding of `Field`.

Computations in proofs :

- Cut of $t = t'$ + rewriting of t into t'
- Proof of $t = t'$ with `Field`

Regarding the interface :

- Realized in a very light way
- No specific architecture, no state
- A simple pipe created for each call
- Asymmetric role : Coq → Maple

Available as a Coq (V7.3↑) contribution :

- Chalmers/MapleMode, about 300 lines of ML (quite short)

Examples (1/2)

$\forall x, y \in \mathbb{R}$, if $x \neq 0$ and $y \neq 0$ then :

$$\left(\frac{x}{y} + \frac{y}{x}\right) x.y - (x.x + y.y) + 1 > 0$$

`simp1 < intros.`

```
x, y : R
H : x <> 0
H0 : y <> 0
```

```
=====
```

$$(x / y + y / x) * x * y - (x * x + y * y) + 1 > 0$$

`simp1 < simplify (x/y+y/x)*x*y-(x*x+y*y)+1.`
 2 subgoals

```
H : x <> 0
H0 : y <> 0
```

```
=====
```

$$1 > 0$$

subgoal 2 is:
 $y * x <> 0$

Examples (2/2)

Computation of a characteristic polynomial :

$$\begin{vmatrix} -1-x & 4 & 2 \\ -5 & 7-x & 5 \\ 7 & -6 & -6-x \end{vmatrix}$$

Expansion w.r.t. the first column :

```
Coq < Definition car1 (x:R) := Eval Factor in
Coq < (-1-x)*((7-x)*(-6-x)+30)+5*(4*(-6-x)+12)+
Coq < 7*(20-2*(7-x)).
car1 is defined
```

```
Coq < Print car1.
car1 = fun x : R => -( x + -1 ) * ( x + -2 ) * ( x + 3 )
      : R->R
```

Extensions

This interface can be (easily) extended :

- To other functions with an arity of 1 (direct)
- To other function with higher arities
- To other computations (limits, derivatives, **gcd's**, etc)

Quantifier elimination (1/3)

Algebraically closed field (ACF) :

$$\forall P \in K[X]. \deg(P) > 0 \Rightarrow \exists x \in K. P(x) = 0$$

Formal statement :

Given the polynomials $P_i, i=1..n$ and $Q_j, j=1..m$.

The problem (Φ)

$\exists x \in K. P_1(x) = 0 \wedge \dots \wedge P_n(x) = 0 \wedge Q_1(x) \neq 0 \wedge \dots \wedge Q_m(x) \neq 0$
is decidable.

which is equivalent to show $\Phi \vee \neg\Phi$ (**constructively**)

$\hookrightarrow \Phi$: **ACF's definition**

$\hookrightarrow \neg\Phi$: **it fails**

Quantifier elimination (2/3)

Auxiliary lemmas

Proposition 1 :

$G = \gcd(P_1, P_2)$. We have :

$\exists x. P_1(x) = 0 \wedge P_2(x) = 0$ iff $\exists x. G(x) = 0$.

Proposition 2 :

If $Q \neq 0$ then $\exists x. Q(x) \neq 0$.

Proposition 3 :

P and Q are relatively prime. We have :

$\exists x. P(x) = 0 \wedge Q(x) \neq 0$ iff $\exists x. P(x) = 0$.

Proposition 4 :

$G = \gcd(P, Q)$ and P_1 s.t. $P(x) = G(x)P_1(x)$. We have :

$\exists x. P(x) = 0 \wedge Q(x) \neq 0$ iff $\exists x. P_1(x) = 0 \wedge G(x) \neq 0$.

Proposition 5 :

$G = \gcd(P, Q)$ and P_1 s.t. $P(x) = G(x)P_1(x)$. If P and Q are not relatively prime then $\deg(P_1) < \deg(P)$.

Quantifier elimination (3/3)

Algorithm

- I. if $n \neq 0$ then compute $P(x) = \gcd(P_{i,i=1..n}(x))$ else go to III.
- II. if $m = 0$: apply [proposition 1](#). This is equivalent to show $\exists x.P(x) = 0$:
 1. if $\deg(P) = 0$ or $P = 0$: if $P = 0$ then conclude trivially else it fails
 2. else conclude applying [ACF](#)'s definition.
- III. if $m \neq 0$:
 1. compute $Q(x) = \prod_{i=1}^m Q_i(x)$
 2. if $n = 0$ then apply [proposition 2](#)
 3. if $n \neq 0$ then this is equivalent to show $\exists x.P(x) = 0 \wedge Q(x) \neq 0$:
 - a. if $\deg(Q) = 0$ or $Q = 0$: if $Q = 0$ then it fails else go to II
 - b. if $\deg(P) = 0$ or $P = 0$: if $P = 0$ then go to III2 else it fails
 - c. else compute $G(x) = \gcd(P(x), Q(x))$:
 - i. if $\deg(G) = 0$, $G(x) = a$ and $a \neq 0$ then apply [proposition 3](#)
 - ii. else apply [proposition 4](#) and [re-apply](#) the algorithm.

Use of Maple

We need :

- gcd (G)
- quotients (P_1, Q_1)
- cofactors (A, B)

Maple : computation of G, P_1, Q_1, A and B .

Coq : Bézout's theorem (converse) to show (automatically) :

$$\begin{cases} P = P_1 G \\ Q = Q_1 G \\ G = AP + BQ \end{cases}$$

Bézout's theorem (converse) :

If G divides P and Q , and if there exists two polynomials A and B s.t. $AP + BQ = G$ then G is the gcd of P and Q .

Example

$$\begin{aligned} \text{parabola}(x) &= x^2 + 1 \\ \text{cubic}(x) &= x^3 + x^2 + x + 1 \\ \text{line}(x) &= x + 1 \end{aligned}$$

$$\exists x. x^2 + 1 = 0 \wedge x^3 + x^2 + x + 1 = 0 \wedge x + 1 \neq 0$$

3 solutions for $x^2 + 1 = 0$ and $x^3 + x^2 + x + 1 = 0$: i , $-i$ and -1 .
The solution -1 does not satisfy $x + 1 \neq 0$.

In Coq :

```
Coq < Goal exists x:C, (UPeval x parabola)=C0/\
Coq <                               (UPeval x cubic)=C0/\~(UPeval x line)=C0.
Unnamed_thm < unfold parabola cubic line; qelim.
Proof completed.
```

Conclusion

Several goals have been achieved :

- Computation/proof paradigm $\rightarrow$ CAS/DS
- Interface between Coq and Maple
- Examples inside and outside proofs
- Extension to the gcd $\Rightarrow$ QElim (ACF, univariate polynomials)

This interface increases the **power of computation** of Coq but also its **power of automation** !

Related work :

- CAS/DS : HOL/Maple (1998), PVS/Maple (2001)
- Quantifier elimination : Tarski tactic (2002)

Future work :

- Other computations (limits, derivatives, etc)
- Quantifier elimination :
multivariate case, real closed fields, other algebras ($<$), etc

Author: Lennart Beringer

Title: An overview of the MRG project

Abstract:

We give an overview and demo of the Mobile Resource Guarantees framework: a system for ensuring that downloaded programs are free from run-time violations of resource bounds. Certificates are attached to code in the form of efficiently checkable proofs of resource bounds; in contrast to cryptographic certificates of code origin, these are independent of trust networks. A novel programming language with resource constraints encoded in function types is used to streamline the generation of proofs of resource usage.

An overview of the Mobile Resource Guarantees Project

Lennart Beringer
Laboratory for Foundations of Computer Science
University of Edinburgh

Funded by the EC, under initiative “Global Computing”, IST-2001-33149

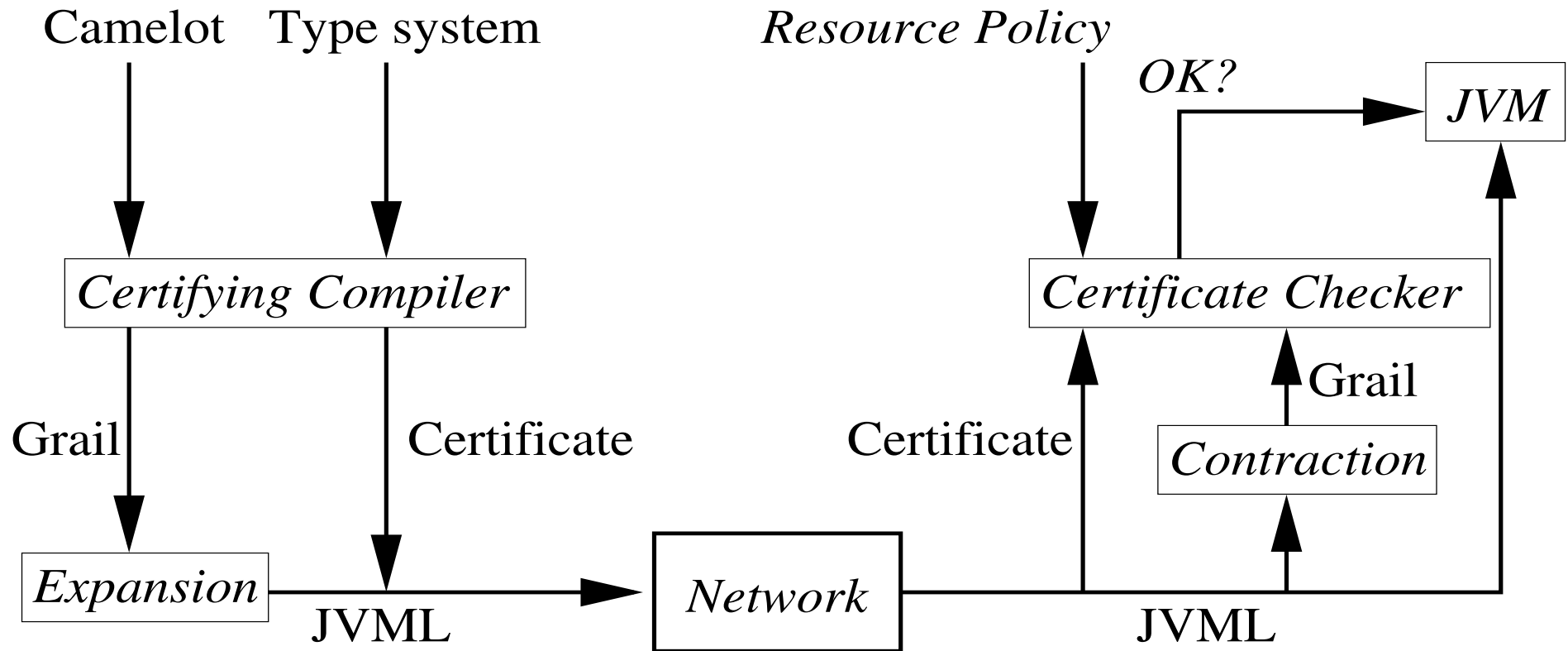
TYPES 2004

MRG: PCC infrastructure for resource-related properties

- Applications with resource considerations: portable devices (phones, PDA's,...), Smartcards, embedded processors (car electronics,...), satellites, GRID services,...
- Example resources: memory (heap & stack), time, energy, network bandwidth, parameter values of system calls
- PCC: code consumer requires transmitted (low-level) program to come with automatically verifiable proof that his resource policy is fulfilled
- Certifying compilation: translation from high-level language into machine language (semi-automatically) derives independently verifiable certificates
- MRG: certificate generation by type systems for resource usage and consumption
- Complements applications of PCC that target memory (type) safety

This talk: brief overview, short demo, conclusion

MRG architecture



Works because of reversible expansion of Grail into JVML subset

Camelot

Camelot: ML-like first-order functional language (polymorphism, no references)

- Example program: (monomorphic) insertion sort:

```
type iList = !Nil | Cons of int * iList
let ins a l =
  match l with Nil -> Cons(a, Nil)
              | Cons(x, t)@_ -> if a < x then Cons(a, Cons(x, t))
                                else Cons(x, ins a t)
let sort l = match l with Nil -> Nil | Cons(a, t)@_ -> ins a (sort t)
```

- Notation @_ indicates destructive pattern match
- Whole program compilation where each Camelot function yields one JVM method
- Compilation includes an explicit memory manager (freelist)

Wish to certify memory consumption of compiled output (initial length of freelist).

Heap analysis & certification

```
let ins a l =  
  match l with Nil -> Cons(a,Nil)  
             | Cons(x,t)@_ -> if a < x then Cons(a,Cons(x,t))  
                               else Cons(x, ins a t)  
let sort l = match l with Nil -> Nil | Cons(a,t)@_ -> ins a (sort t)
```

- Memory consumption inferred from program using Hofmann-Jost's type system
- Result: `ins` consumes one memory cell, independent from actual input, `sort` does not consume any memory (in-place)
- In general: memory consumption expressed relative to size of input
- Certificate: encoding of the result of the type inference as method specifications

Insertion sort: compiler output

```
method static public List ins(int a, D l) = ...D.make(a, null)...
```

```
method static public List sort(D l) =
```

```
  if l = null then null
```

```
    else let h = l.HD in let t = l.TL in let _ = D.free(l) in
```

```
      let l = List.sort(t) in List.ins(h, l)
```

... plus code for memory management and runtime environment methods

- **D.make**(...): takes object from freelist, or calls **new**
- **D.free**(x): inserts object into freelist
- **D.main**(l): constructs initial freelist, calls **List.sort**(**s2i**(l))

We wish to verify that

- any memory allocation throughout an invocation of **main** is performed during the initial construction of the freelist, and in particular that
- during the execution of **List.sort**(l), all invocations of **make** are executed on a non-empty freelist, i.e. no call to **new** is performed

Type-based analysis of Camelot programs

Type system by Hofmann and Jost (POPL 2003):

- Input: program containing a function **start**: `string list -> unit`

Output: a *linear function* s such that **start**($\mathbb{l}$) will not call **new** when evaluated in a heap h where

- $\mathbb{l}$ points in h to a linear list of some length n
- the freelist which forms a part of h is well-formed, does not overlap with $\mathbb{l}$, and has length not less than $s(n)$
- How does this work?
 - Annotate types with freelist annotations for each constructor: $\mathbf{L}(k)$
 - Judgements $\Gamma, n \vdash e : T, m$ include information about *initial* and *final* size of freelist
 - Express final size of freelist as function of the size of the output
 - Complement this type system with some method for preventing deallocation of live cells (linear typing, usage aspects, layered sharing,...)
- Example: $\text{List.ins} : 1, \mathbf{I} \times \mathbf{L}(0) \rightarrow \mathbf{L}(0), 0$ $\text{List.sort} : 0, \mathbf{L}(0) \rightarrow \mathbf{L}(0), 0$

Consumer-side verification

- Formalise operational semantics of (virtual) machine language (like FPCC)
- Use a general-purpose program logic: soundness & (relative) completeness formally proven, little automation
- Derive special logics formally in theorem prover:
 - Interpret high-level typing judgements as a formulae in the core logic (fixed assertion format), guided by correctness statement of type system
 - Derive lemmas for low-level code fragments in the core logic. Guiding motivation:
$$\Gamma, n \vdash e : T, m \rightsquigarrow \triangleright \text{compile}(e) : \llbracket \Gamma, n, T, m \rrbracket$$

But: in principle, we don't need to prove that the implication holds
 - Implement tactic that "replays" typing derivations
 - Syntax-directedness of rules limits necessity for proof search
 - Simple side conditions: (in-)equalities over numbers, set inclusions. . .
- Invariants (method specifications) supplied by producer via certificate
- Expansion of statements in fixed assertion format into core logic (and op. semantics) always possible, but usually not necessary (TCB negotiation)

Demo

- JVM integration
 - Modified security manager: execution permitted only after successful verification
 - Hand-crafted Isabelle-files for program representation and certificate
- Prototypical implementation of certificate generation
 - Not yet fully finished (needs debugging, evaluation, polishing . . . !!!)
 - Examples: standard functions over (integer) lists: insertion sort, append, reverse, double. More functions (heapsort, flatten. . .) have been verified for manually created certificates
 - producer: input: prog.cmlt output: prog.class & Certificate.thy (contains method specifications and flow information)
 - consumer: input: prog.class output: Consumer1.thy (contains tactic and verification of method bodies) & Consumer2.thy (contains top-level correctness theorems, with standard proofs)

Discussion

Current & future work:

- Full integration of software components (CertGen, MRG-JVM, ...)
- Resource policies (wrapper, other policies)
- Generalise existing system of derived assertions (sharing, usage-aspects, separation), and evaluate on bigger examples
- Extract stand-alone proof checker from Isabelle tactic and derived proof rules
- Derive specialised logics and certificate generation for other resources: frame stack, time, limits and separation conditions on method parameters
- Termination logic & certification

Conclusion:

- MRG-motto: use structure available in HLL: program structure & type structure
- CertGen by interpreting (resource) type systems in program logic
- Chain of abstractions: operational semantics $\rightarrow$ general program logic $\rightarrow$ derived specialised logics with automation
- Development backed up by implementation in Isabelle/HOL

Extra slide: Certificates and automated verification

Producer-generated certificate:

- Content: method-level specifications in derived-assertions form
- Representation: Isabelle/HOL script that invokes a standard tactic **proveMe**

Consumer side:

- Tactic **proveMe** that
 - invokes derived proof rules (syntax-directed) and
 - discharges side conditions (set inclusions, arithmetic (in-)equalities).
 - Methods verified once, combination for mutual recursion via cut rule and parameter adaptation
 - Functions (basic blocks) verified once, via optimised treatment of merge points that combines imperative (dominator property) and functional (function parameters) viewpoints
 - Currently verified programs: functions over lists and trees (append, flatten, insertion sort & heap sort, ...)

Author: Pierre Corbineau

Title: A reflexive tactic for first-order reasoning in Coq

Abstract:

We propose a new implementation of a Coq tactic for first-order reasoning, using reflection. We produce a concrete structure for proof traces that is checked by a Coq function, which we proved correct. This method provides significant improvements regarding efficiency, and size of the proof-terms produced.

Author: Conor McBride

Title: Epigram

Abstract:

Epigram (<http://www.dur.ac.uk/CARG/epigram>)

is a high-level functional programming language which

elaborates into type theory. It was designed by myself and James McKinna.

It comes equipped with a (prototype) interactive type-directed editor. I

shall demonstrate the system's current capabilities.

TYPE SETUPS for Predicate Logic

Peter Aczel
Manchester

(with some help from
Joao Belo - MSc thesis)

I. Motivations (4)

II. Type Setups (4)

III. DS-logic over a type setup (3)

IV. Some signatures for DS-logic (1)

Types 2004

Paris

December 2004

Type Setups - Motivation 1

- There are advantages to logic-enriched type theories where predicate logic is simply added on top of a type theory.

This allows the freedom to interpret or reinterpret the logic in different ways; e.g.

- props-as-types
- props as objects of type Prop (in CS)
- sheaf reinterpretations
- $\neg\neg$ reinterpretation
- ...

- What is needed from a type theory in order to logic-enrich it?

Answer : A type setup

Type Setups - Motivation 2

- We want to generalise many-sorted logic to dependently-sorted logic (DS-logic)
- The notion of a type setup captures the needed properties of sorts, terms and atomic formulae for DS-logic
- The details of how expressions are formed using a signature of sort/term/prop - constructors is abstracted away
- We keep variables and contexts of variable declarations as in dependent type theory.
- We axiomatize how substitutions work.

a type theory with
predicate symbols

a signature of
sort/item/prop
constructors

↓ ↓
a type setup

↓
a λ -category with atomic props

λ -category = { category with families
contextual category
category with attributes
type category
⋮

Many roughly equivalent category
theoretic notions with many authors.

Some References

J. Cartmell (Ph.D thesis, 1978
APAL, 1986)

Generalised Algebraic Theories and
Contextual categories.

P. Dybjer (Types, 1995)

Internal Type Theory
(categories with families)

M. Makkai (unpublished? monograph, 1995)
FOLDS

(DS-logic without function symbols)

A type setup for DS-logic consists of

- a category of contexts Γ and substitution maps $\sigma: \Delta \rightarrow \Gamma$,
- for each context Γ
 - a set $Ty(\Gamma)$ of Γ -types
 - a set $Tm(\Gamma, A)$ of Γ -terms of type A for each Γ -type A
 - a set $At(\Gamma)$ of atomic Γ -formulae
- for each substitution map $\sigma: \Delta \rightarrow \Gamma$ substitution operations

$$\begin{aligned} A \in Ty(\Gamma) &\longmapsto A\sigma \in Ty(\Delta) \\ a \in Tm(\Gamma, A) &\longmapsto a\sigma \in Tm(\Delta, A\sigma) \\ \theta \in At(\Gamma) &\longmapsto \theta\sigma \in At(\Delta) \end{aligned}$$

These must satisfy TS1 – TS4

TS1 : Substitution Action

- For each identity $\text{id}_\Gamma : \Gamma \rightarrow \Gamma$

$$\begin{array}{ll} A \text{id}_\Gamma = A & (A \in \text{Ty}(\Gamma)) \\ a \text{id}_\Gamma = a & (a \in \text{Tm}(\Gamma, A)) \\ \theta \text{id}_\Gamma = \theta & (\theta \in \text{At}(\Gamma)) \end{array}$$

- For maps $\sigma : \Delta \rightarrow \Gamma, \tau : \mathcal{L} \rightarrow \Delta$

$$\begin{array}{ll} A(\sigma \circ \tau) = (A\sigma)\tau & (A \in \text{Ty}(\Gamma)) \\ a(\sigma \circ \tau) = (a\sigma)\tau & (a \in \text{Tm}(\Gamma, A)) \\ \theta(\sigma \circ \tau) = (\theta\sigma)\tau & (\theta \in \text{At}(\Gamma)) \end{array}$$

TS2: Contexts

- Contexts Γ have the form

$$x_1: A_1, \dots, x_n: A_n$$

where

$x_1, \dots, x_n$ are distinct variables

$$A_i \in \text{Ty}(\underbrace{x_1: A_1, \dots, x_{i-1}: A_{i-1}}_{\Gamma_{<i}}) \quad (i=1, \dots, n)$$

- Then

$$\begin{aligned} A_i &\in \text{Ty}(\Gamma) \\ x_i &\in \text{Tm}(\Gamma, A_i) \end{aligned} \quad (i=1, \dots, n)$$

TS3: Substitution Maps

- Substitution maps $\sigma: \Delta \rightarrow \Gamma$ have the form

$$[x_1 := a_1, \dots, x_n := a_n]_{\Delta \rightarrow \Gamma}$$

where

$$a_i \in \text{Tm}(\Delta, A_i[x_1 := a_1, \dots, x_{i-1} := a_{i-1}]_{\Delta \rightarrow \Gamma_{<i}})$$

- Then

$$x_i \sigma = a_i \quad (i=1, \dots, n)$$

and is the unique such map.

TS4: Weakening

- $\Gamma \subseteq \Delta$ if every declaration in Γ is in Δ

- If $\Gamma \subseteq \Delta$ then

$$Ty(\Gamma) \subseteq Ty(\Delta)$$

$$Tm(\Gamma, A) \subseteq Tm(\Delta, A) \quad (A \in Ty(\Gamma))$$

$$At(\Gamma) \subseteq At(\Delta)$$

and

$$A\pi = A$$

$$(A \in Ty(\Gamma))$$

$$a\pi = a$$

$$(a \in Tm(\Gamma, A))$$

$$\theta\pi = \theta$$

$$(\theta \in At(\Gamma))$$

when

$$\pi = [x_1 := x_1, \dots, x_n := x_n]_{\Delta \rightarrow \Gamma} : \Delta \rightarrow \Gamma$$

where Γ is $(x_1 : A_1, \dots, x_n : A_n)$

DS-logic over a type setup

- Γ -formulae φ

$$\varphi ::= 0 \mid \perp \mid (\varphi_1 \rightarrow \varphi_2) \mid (\forall x:A) \varphi_0$$

0 atomic Γ -formula

φ_1, φ_2 Γ -formulae

φ_0 $(\Gamma, x:A)$ -formula

- Sequents $(\Gamma) \Phi \Rightarrow \varphi$

Φ list of Γ -formulae

- Inference Rules

standard for classical logic

e.g.

$$\frac{(\Gamma, x:A) \Phi \Rightarrow \varphi_0}{(\Gamma) \Phi \Rightarrow (\forall x:A) \varphi_0}$$

$$(\Gamma) \Phi \Rightarrow (\forall x:A) \varphi_0$$

$$\frac{(\Gamma) \Phi \Rightarrow (\forall x:A) \varphi_0}{(\Gamma) \Phi \Rightarrow \varphi_0[a/x]}$$

$$(\Gamma) \Phi \Rightarrow \varphi_0[a/x]$$

a is a Γ -term of type A

$$[a/x] = [x_1 := x_1, \dots, x_n := x_n, x := a]_{\Gamma \rightarrow (\Gamma, x:A)}$$

$$: \Gamma \rightarrow (\Gamma, x:A)$$

where Γ is $(x_1:A_1, \dots, x_n:A_n)$

Semantics for DS-logic

- An interpretation $\mathcal{A}$ for a type setup consists of, for each context Γ
 - a set $\llbracket \Gamma \rrbracket^{\mathcal{A}}$ of tuples $\vec{u}$
 - a function $\llbracket \sigma \rrbracket^{\mathcal{A}} : \llbracket \Delta \rrbracket^{\mathcal{A}} \rightarrow \llbracket \Gamma \rrbracket^{\mathcal{A}}$ for each $\sigma : \Delta \rightarrow \Gamma$
 - for each $\vec{u} \in \llbracket \Gamma \rrbracket^{\mathcal{A}}$

$$\begin{cases} \text{a set } \llbracket A \rrbracket_{\Gamma}^{\mathcal{A}}(\vec{u}) & (A \in \text{Ty}(\Gamma)) \\ \llbracket a \rrbracket_{\Gamma, A}^{\mathcal{A}}(\vec{u}) \in \llbracket A \rrbracket_{\Gamma}^{\mathcal{A}}(\vec{u}) & (a \in \text{Trm}(\Gamma, A)) \\ \llbracket \theta \rrbracket_{\Gamma}^{\mathcal{A}} \subseteq \llbracket \Gamma \rrbracket^{\mathcal{A}} & (\theta \in \text{At}(\Gamma)) \end{cases}$$

satisfying . . .

- We can define " $\mathcal{A} \models_{\Gamma} \varphi[\vec{u}]$ " as usual

e.g. $\mathcal{A} \models_{\Gamma} \theta[\vec{u}]$ iff $\vec{u} \in \llbracket \theta \rrbracket_{\Gamma}^{\mathcal{A}}$

$\mathcal{A} \models_{\Gamma} (\forall x:A) \varphi_0[\vec{u}]$ iff

for all $v \in \llbracket A \rrbracket_{\Gamma}^{\mathcal{A}}(\vec{u})$

$\mathcal{A} \models_{\Gamma, x:A} \varphi[\vec{u}, v]$

III:

Soundness-Completeness Theorem

$\Sigma \vdash \varphi$ iff $(\) \Phi \Rightarrow \varphi$ can be proved
for some list Φ of sentences
from Σ

$\Sigma \models \varphi$ iff $(\forall \psi \in \Sigma)(A \models_{\text{cl}} \psi) \Rightarrow A \models_{\text{cl}} \varphi$
for every interpretation A

Theorem $\Sigma \vdash \varphi$ iff $\Sigma \models \varphi$

Proof: As usual
e.g. use Henkin method
via Extension Lemma:

If Σ is a consistent set of sentences
of a type setup Π and
 $(\exists x:A) \varphi_0 \in \Sigma$ then Π can be
extended to a type setup $\Pi'[c:A]$
with c a new closed term of
type A so that $\Sigma \cup \{\varphi_0[c/x]\}$
is a consistent set of sentences
of $\Pi'[c:A]$.

Some Signatures for DS-logic

- the notion of a category

Obj **sort**

$\text{Hom}(a, b : \text{Obj})$ **sort**

$\text{Id}(a : \text{Obj}) : \text{Hom}(a, a)$

$\text{Comp}(a, b, c : \text{Obj},$
 $f : \text{Hom}(b, c),$
 $g : \text{Hom}(a, b)$
 $) : \text{Hom}(a, c)$

- an unusual, but coherent, signature

1) $F(y : F(c))$ **sort**

2) $c : F(c)$

We can derive

i) $\Rightarrow c : F(c)$ by 2)

ii) $\Rightarrow F(c)$ **sort** by 1) and i)

iii) $y : F(c) \Rightarrow F(y)$ **sort** by 1) and ii)

1) is coherent by iii)

2) is coherent by ii)

Author: Olivier Hermant

Title: Semantical Cut Elimination in Intuitionistic Deduction Modulo

Abstract:

In this talk, we will show techniques that allows to show cut elimination for a wide range of rewrite systems in deduction modulo. The frame is Intuitionnistic Sequent Calculus Modulo, and we show soundness and completeness of the cut-free calculus w.r.t. Kripke Structures, one of the well-known semantics of intuitionnistic calculi. We show these results for some wide conditions on the rewrite system, and at last exhibit a system ehancing cut redundancy, although non-normalizing.

Semantic Cut Elimination in Intuitionistic Deduction Modulo

Olivier Hermant, INRIA Futurs, Palaiseau
[http ://pauillac.inria.fr/~ohermant](http://pauillac.inria.fr/~ohermant)

Types 2004
Dec. 15-18 Jouy-en-Josas

Outline

- The deduction system
- Soundness, Completeness and Cut Elimination
- Two conditions and the proofs
- Putting the two conditions together

Deduction Rules

Some Rules of Sequent Calculus

$\frac{}{\Gamma, P \vdash P} \text{axiom}$	$\frac{\Gamma, P \vdash R \quad \Gamma \vdash P}{\Gamma \vdash R} \text{cut}$
$\frac{\Gamma \vdash P \quad \Gamma Q \vdash R}{\Gamma P \Rightarrow Q \vdash R} \Rightarrow \text{-l}$	$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} \Rightarrow \text{-r}$
$\frac{\Gamma, \{t/x\}P \vdash R}{\Gamma, \forall x P \vdash R} \forall\text{-l}$	$\frac{\Gamma \vdash \{c/x\}P}{\Gamma \vdash \forall x P} \forall^*\text{-r}$

We want to add rewrite rules on terms and on propositions :

$$x * y = 0 \quad \rightarrow \quad (x = 0) \vee (y = 0)$$

$$(x + y) + z \quad \rightarrow \quad x + (y + z)$$

$$x * 0 \quad \rightarrow \quad 0$$

Deduction Rules

$\mathcal{R}$ is a set of Rewrite Rules

Some Rules of Sequent Calculus Modulo

$\frac{}{\Gamma, P \vdash_{\mathcal{R}} Q} \text{axiom, } P \equiv Q$	$\frac{\Gamma, P \vdash_{\mathcal{R}} R \quad \Gamma \vdash_{\mathcal{R}} Q}{\Gamma \vdash_{\mathcal{R}} R} \text{cut, } P \equiv Q$
$\frac{\Gamma \vdash_{\mathcal{R}} P \quad \Gamma, Q \vdash_{\mathcal{R}} R}{\Gamma S \vdash_{\mathcal{R}} R} \wedge\text{-l, } S \equiv P \Rightarrow Q$	$\frac{\Gamma, P \vdash_{\mathcal{R}} Q}{\Gamma \vdash_{\mathcal{R}} S} \Rightarrow\text{-r, } S \equiv P \Rightarrow Q$
$\frac{\Gamma, \{t/x\}P \vdash_{\mathcal{R}} R}{\Gamma, Q \vdash_{\mathcal{R}} R} \forall\text{-l, } Q \equiv \forall xP$	$\frac{\Gamma \vdash_{\mathcal{R}} \{c/x\}P}{\Gamma \vdash_{\mathcal{R}} Q} \forall^*\text{-r, } Q \equiv \forall xP$

In the general case, cut elimination doesn't hold (even in confluent terminating cases) :

$$A \rightarrow B \wedge \neg A$$

$\Rightarrow$ We need some (general) conditions on the Rewrite System to ensure cut-elimination.

Classical case

Theorem[Soundness] : If $\Gamma \vdash_{\mathcal{R}} \Delta$ (with possible cuts) then $\Gamma \models \Delta$.

Theorem[Completeness] : If $\mathcal{T}$ is a cut free-consistent theory, it has a model

Corollary[Cut elimination] : If $\Gamma \vdash_{\mathcal{R}} \Delta$ then $\Gamma \models \Delta$ hence $\Gamma \vdash_{\mathcal{R}}^{cf} \Delta$.

In the intui-

tionistic case, we need other definitions than the usual ones :

A-consistency	:	$\Gamma \not\vdash_{\mathcal{R}}^{cf} A$
A-completeness	:	$P \in \Gamma \text{ or } \Gamma, P \vdash_{\mathcal{R}}^{cf} A$
A-Henkin witnesses	:	$\Gamma, \exists x P \not\vdash_{\mathcal{R}}^{cf} A \Rightarrow \{c/x\}P \in \Gamma$

Theorem[Soundness] : If $\Gamma \vdash_{\mathcal{R}} P$ (with possible cuts)

then $\Gamma \models P$.

Theorem[Completeness] : If $\mathcal{T}$ is a P -cut free-consistent

theory, it has a model that is not a model of P .

Corollary[Cut elimination] : If $\Gamma \vdash_{\mathcal{R}} P$ then $\Gamma \vdash_{\mathcal{R}}^{cf} P$.

Proof : if $\Gamma \vdash_{\mathcal{R}} P$, by soundness, we have $\Gamma \models P$. By

completeness theorem, this means that Γ is

P -cut free-inconsistent, i.e. $\Gamma \vdash_{\mathcal{R}}^{cf} P$.

Our notion of model : Kripke Model, extended to deduction modulo with the condition :

$$\text{if } P \equiv Q \text{ then } \alpha \Vdash P \Leftrightarrow \alpha \Vdash Q$$

A Kripke Structure :

- a partially ordered set of “worlds” $K, \leq$
- a domain D for each $\alpha \in K$, $D(\alpha)$. D is monotone wrt $\leq$.
- a forcing relation $\Vdash$ defined by induction on the propositions. e.g.

$$\alpha \Vdash P \Rightarrow Q \text{ iff } \forall \beta \geq \alpha \quad \beta \Vdash P \Rightarrow \beta \Vdash Q$$

Completion of an A -consistent theory $\mathcal{T}$

Set $\Gamma_0 = \mathcal{T}$, enumerate all the propositions of the language extended with new fresh constants :

$$P_0, \dots, P_n, \dots$$

At each step, check if $\Gamma_n, P_n \vdash_{\mathcal{R}}^{cf} A$ or not, and define

- $\Gamma_{n+1} = \Gamma_n$ if yes
- $\Gamma_{n+1} = \Gamma_n \cup \{P_n\}$ if no

Add a Henkin witness if P_n is an existential formula.

Take $\Gamma = \bigcup_{n=0}^{\infty} \Gamma_n$.

Γ is A -complete, A -consistent, admits A -Henkin witnesses.

All of this is valid under the only hypothesis of confluence of $\mathcal{R}$

First condition

We will consider a rewrite system that is :

- confluent
- terminating
- compatible with a well-founded order $\succ$ having the subformula property.

E.g. the rule $A[x, 0] \longrightarrow B[x] \Rightarrow C$ is compatible with such an order.

We prove the completeness theorem : given a A -consistent theory $\mathcal{T}$ expressed in a language $\mathcal{L}_0$, we construct a Kripke Structure and find a node $\alpha \Vdash \mathcal{T}$ and $\alpha \nVdash A$

Model Construction

Consider a denumerable family of set of new constants C_n .
Define the languages $L_0, \dots, L_n$ such that $L_{n+1} = L_n \cup C_n$,

- $K = \{\Gamma\}$, s.t. for some proposition A , Γ is an A -consistent, A -complete theory of some L_i , admitting A -Henkin witnesses.
- $\leq$ is the inclusion.
- $D(\Gamma) = \text{clos}(L_i)$
- if A is a normal atom, $\Gamma \Vdash A$ iff $A \in \Gamma$.
- if A is a non-normal atom, set $\Gamma \Vdash A$ iff $\Gamma \Vdash A \downarrow$.
- define as usual the interpretation of non atomic propositions.

Definition is well-founded, thanks to $\succ$. This is a Kripke Structure for $\mathcal{R}$, and $\Gamma \Vdash \mathcal{T}$, $\Gamma \nVdash A$

Application : Quantifier-free rewrite systems

We consider only rules $A \rightarrow Q$ where Q doesn't contain quantifiers. We need confluence and termination of the set of rules.

The pair $\langle \#_{\forall, \exists}, \#_{\wedge, \vee, \Rightarrow, \neg} \rangle$ is a well-founded order on normal terms.

Extend it on propositions : $A \succ B$ if

- $A \downarrow \succ B \downarrow$
- $A \downarrow = B \downarrow$ and $A \rightarrow^+ B$

Second condition

We consider a positive rewrite system $\mathcal{R}$:

in a rewrite rule $A \rightarrow P$ atoms of P occurs positively. For example :

$$A \longrightarrow \forall x A$$

$$A \longrightarrow (\neg B) \Rightarrow C$$

Model construction

As before, we define a family of languages.

- $K = \{\Gamma\}$, Γ is an A -consistent, A -complete theory of some L_i , admitting A -Henkin witnesses, ordered by $\subset$
- $D(\Gamma) = \text{clos}(L_i)$
- in a world Γ , we define the truth value of **all** atoms, and extend it on all propositions.
 - If $B \in \Gamma$ is atomic, we let $\Gamma \Vdash B$
 - if $\Gamma \vdash_{\mathcal{R}}^{cf} B$ and $\Gamma, B \vdash_{\mathcal{R}}^{cf} A$ we let $\Gamma \Vdash B$.
 - else, $\Gamma \not\vdash_{\mathcal{R}}^{cf} B$, we let $\Gamma \nVdash B$.

It is a Kripke Structure, $\Gamma \Vdash \Gamma$ and $\Gamma \nVdash A$.

But is it a model of the rewrite rule? The key lemma :

Lemma 1

$$\begin{array}{ccc}
 \Gamma, P^+ \vdash_{\mathcal{R}}^{cf} A & & \Gamma, Q^- \vdash_{\mathcal{R}}^{cf} A \\
 \Gamma \vdash_{\mathcal{R}}^{cf} P^+ & & \Gamma \vdash_{\mathcal{R}}^{cf} Q^- \\
 & \text{implies} & \\
 \Gamma \Vdash P^+ & & \Gamma \nVdash Q^-
 \end{array}$$

Two conditions together

- $R = \mathcal{R}_{\succ} \cup \mathcal{R}_{+}$
- where $\mathcal{R}_{\succ}$ is compatible with a wfo
- and $\mathcal{R}_{+}$ a positive rewrite system such that
- for any atomic proposition A normal for $\mathcal{R}_{\succ}$, any P , if $A \equiv_{\mathcal{R}_{+}} P$ then any instance of any atom of P is normal for $\mathcal{R}_{\succ}$.

Example :

$$\begin{array}{ll} A \rightarrow (\forall x B) \wedge C & \text{compatible with an order} \\ B(0) \rightarrow \quad \forall x B & \text{a positive rewrite rule} \end{array}$$

Model Construction

We define the Kripke Structure as usual except of the forcing relation :

- if A is a normal atom for $\mathcal{R}_\succ$, $\Gamma \Vdash A$ iff $\Gamma \vdash_{\mathcal{R}}^{cf} A$.
- if A is a non-normal atom for $\mathcal{R}_\succ$, set $\Gamma \Vdash A$ if $\Gamma \Vdash A \downarrow_\succ$.
- in the non-atomic case, set the forcing relation according to the Kripke Structure definition

Model Construction

- We have that $\Gamma \Vdash \Gamma$ and $\Gamma \nVdash A$
- We get a model for $\mathcal{R}_\succ$: proof as in the order case.
- We prove as in the positive case, the lemma :

Lemma 2

$$\begin{array}{cc} \Gamma, P^+ \vdash_{\mathcal{R}}^{cf} A & \Gamma, Q^- \vdash_{\mathcal{R}}^{cf} A \\ \Gamma \vdash_{\mathcal{R}}^{cf} P^+ & \Gamma \vdash_{\mathcal{R}}^{cf} Q^- \\ \text{implies} & \\ \Gamma \Vdash P^+ & \Gamma \nVdash Q^- \end{array}$$

and the Kripke Structures yields a model for $\mathcal{R}_+$ too. $\square$

Conclusion and Perspectives

- link with strong normalization and pre-model construction ([Dowek, Werner])
 - normalization is NOT cut elimination
 - however, how to transform pre-models into Kripke Structure? ([Dowek], [Coquand, Gallier])
- extend this result to the intuitionistic first-order expression of HOL.

Deduction rules of the Intuitionistic Sequent Calculus Modulo

$\frac{}{\Gamma, P \vdash_{\mathcal{R}} Q}$ axiom if $P \equiv Q$	$\frac{\Gamma, P \vdash_{\mathcal{R}} Q \quad \Gamma \vdash_{\mathcal{R}} R}{\Gamma \vdash_{\mathcal{R}} Q}$ cut if $P \equiv R$
$\frac{\Gamma, P, R \vdash_{\mathcal{R}} Q}{\Gamma, P \vdash_{\mathcal{R}} Q}$ contr-l if $P \equiv R$	$\frac{}{\Gamma, P \vdash_{\mathcal{R}} Q}$ $\perp$ -l if $P \equiv \perp$
$\frac{\Gamma \vdash_{\mathcal{R}} Q}{\Gamma, P \vdash_{\mathcal{R}} Q}$ weak-l	$\frac{\Gamma \vdash_{\mathcal{R}}}{\Gamma \vdash_{\mathcal{R}} P}$ weak-r
$\frac{\Gamma, P, Q \vdash_{\mathcal{R}} R}{\Gamma, S \vdash_{\mathcal{R}} R}$ $\wedge$ -l if $P \wedge Q \equiv S$	$\frac{\Gamma \vdash_{\mathcal{R}} P \quad \Gamma \vdash_{\mathcal{R}} Q}{\Gamma \vdash_{\mathcal{R}} R}$ $\wedge$ -r if $P \wedge Q \equiv R$
$\frac{\Gamma, P \vdash_{\mathcal{R}} R \quad \Gamma, Q \vdash_{\mathcal{R}} R}{\Gamma, S \vdash_{\mathcal{R}} R}$ $\vee$ -l if $P \vee Q \equiv S$	$\frac{\Gamma \vdash_{\mathcal{R}} P}{\Gamma \vdash_{\mathcal{R}} S}$ $\vee$ -r if $P \vee Q \equiv R$ $\frac{\Gamma \vdash_{\mathcal{R}} Q}{\Gamma \vdash_{\mathcal{R}} R}$ $\vee$ -r if $P \vee Q \equiv R$
$\frac{\Gamma \vdash_{\mathcal{R}} P \quad \Gamma, Q \vdash_{\mathcal{R}} R}{\Gamma, S \vdash_{\mathcal{R}} R}$ $\Rightarrow$ -l if $P \Rightarrow Q \equiv S$	$\frac{\Gamma, P \vdash_{\mathcal{R}} Q}{\Gamma \vdash_{\mathcal{R}} S}$ $\Rightarrow$ -r if $P \Rightarrow Q \equiv S$
$\frac{\Gamma \vdash_{\mathcal{R}} P}{\Gamma, Q \vdash_{\mathcal{R}}}$ $\neg$ -l if $\neg P \equiv Q$	$\frac{\Gamma, P \vdash_{\mathcal{R}}}{\Gamma \vdash_{\mathcal{R}} Q}$ $\neg$ -r if $\neg P \equiv Q$
$\frac{\Gamma, \{t/x\}P \vdash_{\mathcal{R}} Q}{\Gamma, R \vdash_{\mathcal{R}} Q}$ $\forall$ -l if $\forall xP \equiv R$	$\frac{\Gamma \vdash_{\mathcal{R}} \{c/x\}P}{\Gamma \vdash_{\mathcal{R}} Q}$ $\forall^*$ -r, if $\forall xP \equiv Q$
$\frac{\Gamma, \{c/x\}P \vdash_{\mathcal{R}} Q}{\Gamma, R \vdash_{\mathcal{R}} Q}$ $\exists^*$ -l, if $\exists xP \equiv R$	$\frac{\Gamma \vdash_{\mathcal{R}} \{t/x\}P}{\Gamma \vdash_{\mathcal{R}} Q}$ $\exists$ -r if $\exists xP \equiv Q$

Author: Michel Parigot

Title: On constructive existence

Abstract:

I will discuss a characterization of constructive existence in classical logic due to Matthias Baaz and Christian Fermueller. I will give a simple proof of it based on cut elimination which allows to extend this characterization to any logic having a reasonable cut elimination theorem.

Authors: Gilles Dowek, Alexandre Miquel

Title: A cut elimination theorem for set theory

Abstract:

We show how to express Zermelo's set theory in deduction modulo (i.e. by replacing its axioms by rewrite rules) in such a way that the corresponding notion of proof enjoys the cut-elimination property. For that, we first rephrase set theory as a theory of pointed graphs (following a paradigm due to Aczel) by interpreting set-theoretic equality as bisimilarity, and show that in this setting, Zermelo's axioms can be decomposed into graph-theoretic primitives that can be turned into rewrite rules. We then show that the theory we obtain in deduction modulo is a conservative extension of (a minor extension of) Zermelo's theory. Finally, we build a premodel of this theory from which we deduce the cut-elimination property for its intuitionistic fragment.

Cut elimination for set theory

Gilles Dowek & Alexandre Miquel

Deduction modulo

Replace the axiom $\forall x (x + 0 = x)$ by the rule $x + 0 \longrightarrow x$

A theory: set of axioms **and computation rules**

$$\frac{}{\forall x \ x = x \vdash \forall x \ x = x} \text{Axiom}$$
$$\frac{}{\forall x \ x = x \vdash 2 \times 2 = 4} \text{\(\forall\)-elim}$$
$$\frac{}{\forall x \ x = x \vdash \exists y \ 2 \times y = 4} \text{\(\exists\)-intro}$$

Cut free proofs

In the empty theory: a cut free proof ends with an introduction rule (hence consistency, independence, disjunction, witness, ...)

Lost if we add axioms

Recovered in deduction modulo **with rewrite rules only**

Try to express theories

- with rewrite rules only,
- in such a way that cut elimination holds.

Crabbé's counterexample (1974)

Replace the axiom

$$x \in \cup(a) \Leftrightarrow \exists y (x \in y \wedge y \in a)$$

with the rewrite rule

$$x \in \cup(a) \longrightarrow \exists y (x \in y \wedge y \in a)$$

cut elimination holds (positive)

But not with the rule

$$x \in C \longrightarrow x \in a \wedge \neg(x \in x)$$

That is the question

Does the problem come

- from set theory *per se*
- or from the way axioms are oriented ?

More bad news

Besides the existence axioms, another axiom in set theory

Extensionality

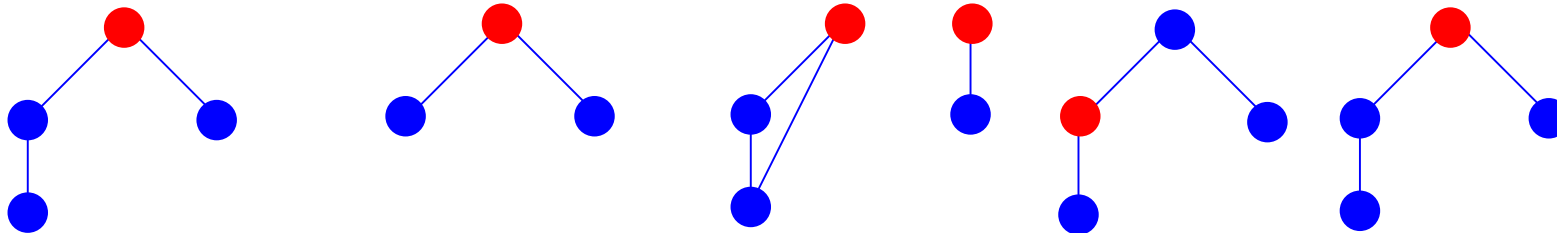
$$(\forall x (x \in a \Leftrightarrow x \in b)) \Rightarrow a = b$$

No clue to orient this axiom

Another presentation of set theory (Aczel, ...)

A new dogma for the XXIst century:

*Mathematics should not be founded on the notion of **set** but on that of **(directed pointed) graph***



A formal theory of graphs

Two sorts (Nodes, Graphs)

Forget (temporarily) about $\in$

A new predicate: η

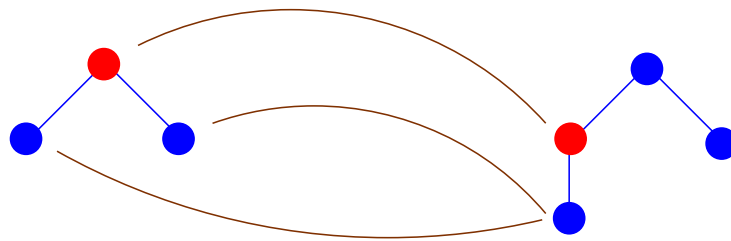
$x \eta_a y$ means “there is an edge from x to y in graph a ”

More symbols: $root(a)$, a/x

Equality and membership

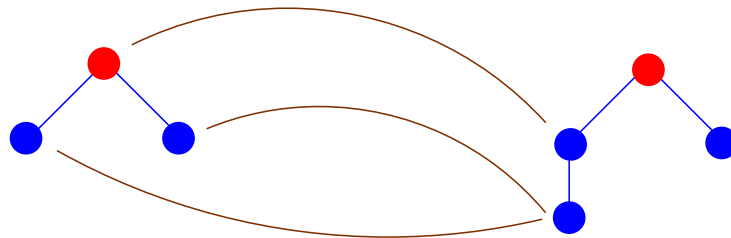
Equality as usual on nodes, but not very useful on graphs

A more useful relation: bisimilarity $\approx$



Membership derived notion: immediate subtrees up to bisimilarity

$$a \in b \xrightarrow{\text{red}} \exists x (x \eta_b \text{root}(b) \wedge a \approx (b/x))$$



Substitutivity

$\approx$ is $\approx$ -substitutive and $\in$ -substitutive

We **could** take the quotient by $\approx$

Fixing the extensionality problem

If a and b have the same elements

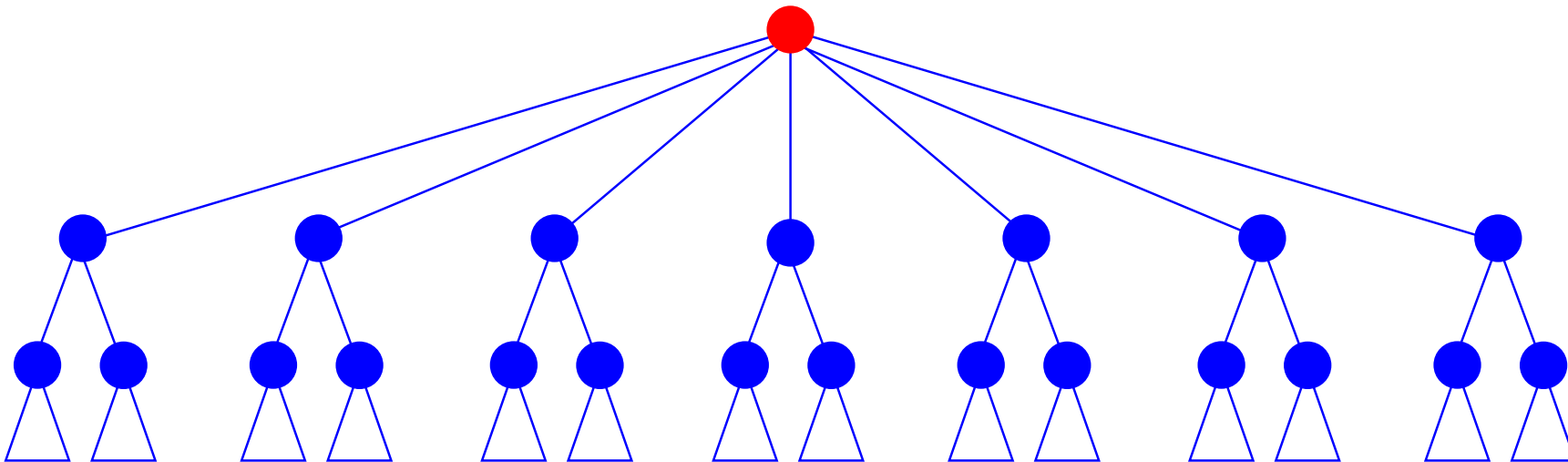
Same direct subtrees (up to $\approx$)

Then $a \approx b$

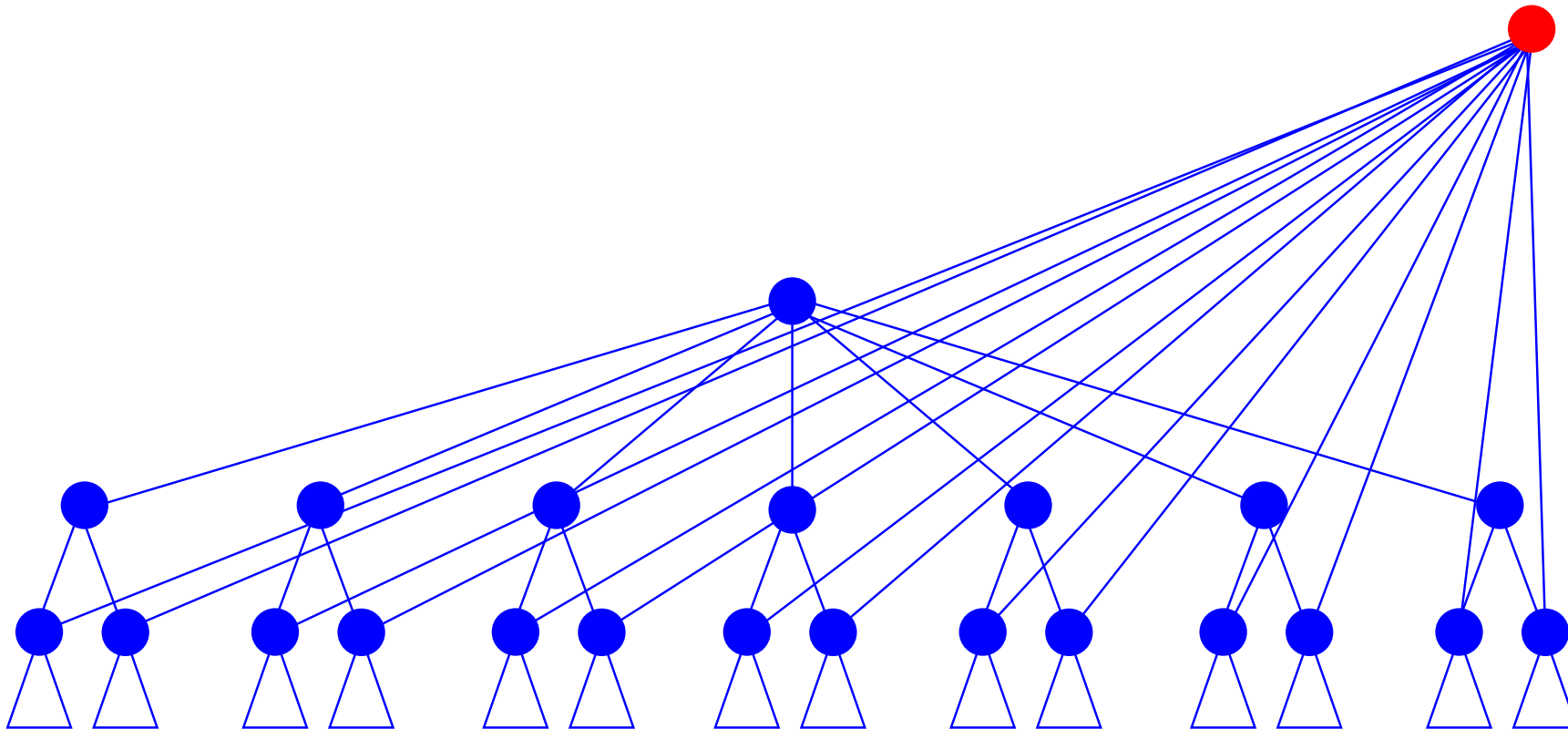
Usual approach: define equality so that it is substitutive (Leibniz equality) and then add the (embarassing) extensionality axiom

Graph theoretic approach: define equality so that it is extensional and prove it is (partially) substitutive

The usual properties of sets (e.g. $\cup$)



The usual properties of sets (e.g. $\cup$)



Delocalisation

Difficulty: pick a node for the root of $\cup(a)$ not in a

An non surjective injection i and an element 0 out its image

$$i'(i(x)) \longrightarrow x$$

$$I(i(x)) \longrightarrow \top \qquad I(0) \longrightarrow \perp$$

Delocalise the full graph a by i and use 0 for root of $\cup(a)$

$$x \ \eta_{\cup(a)} \ x' \longrightarrow$$

$$(\exists y \exists y' \ (x = i(y) \wedge x' = i(y') \wedge y \ \eta_a \ y'))$$

$$\vee (\exists y \exists z \ (x = i(y) \wedge x' = 0 \wedge y \ \eta_a \ z \wedge z \ \eta_a \ \text{root}(a)))$$

$\mathbb{Z}^{\text{mod}}$

4 sorts

26 symbols (including some schemes)

30 rules (including some schemes)

- 1/3 for delocalisation (i)
- 1/3 for the constructions of set theory ($\cup$)
- 1/3 general rules ($\in$)

Equivalence

Z^{mod} is equivalent to

Zermelo's set theory + Strong extensionality + Transitive closure

Between Z and ZF + Foundation

Define two translations (one is trivial, the other interesting)

Cut elimination

To each P we associate a proposition P^* such as P provable in Z^{mod} implies P^* provable in Z

To each P we associate a proposition $p \Vdash P$ such as p is a proof of P in Z^{mod} implies $p \Vdash P$ provable in Z

Saturated graph: to each edge is associated a reducibility candidate

Theorem: If Z^{st} has a ω -model then all proofs in Z^{mod} are strongly normalizing

Author: Stefano Berardi

Title: Continuations, Backtracking and Limits: A Notion of Construction for Classical Logic

Abstract:

We give an overview of a research line started in cooperation with Prof. Hayashi Proof Animation group. Aim of this research is to describe Classical Principle in term of ``constructions'', of a general kind, learning a value by trial-and-error, rather than computing it exactly.

This notion of ``construction" may be equivalently described in many ways. We may use notions from programming: continuations and backtracking. We may use notion from game theory: Coquand's game interpretation. We may use notions from topology: Hayashi's limit interpretation.

Using this notion of construction, we may define, without using Excluded Middle, models of Classical Arithmetic and non-recursive maps. These models are algebraic structures, in the ordinary sense: a set of individual equipped with maps and a notion of equality.

Semantic of

<<Backtracking>

(Ongoing work with
S. Hayashi, T. Coquand)

Jouy-en-Josas, France, December 2004

Stefano Berardi, Semantic of Computation group
C.S. Dept., Turin University, <http://www.di.unito.it/~stefano>



Abstract of the talk

- “*Backtracking*” is the possibility, for a given computation, to come back to a previous state and restart from it, forgetting everything that took place after it.
- We introduce a new notion, “*1-backtracking*”. It is a particular case of backtracking in which forgetting is *irreversible*. If we forget a state we can never restore it back.
- We introduce a game theoretical model for 1-backtracking.

Games in Set Theory

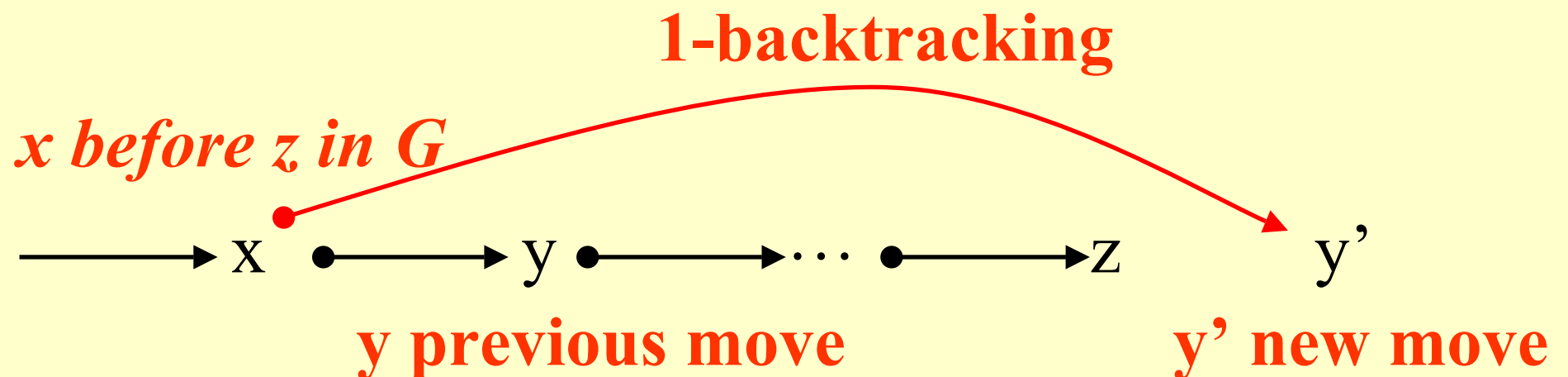
A game $G = (T, R, \text{turn}, W_E, W_A)$ between two players, E (Eloise) and A (Abelard), consists of:

1. a tree T with a father/child relation R .
2. A map $\text{turn}: T \rightarrow \{E, A\}$.
3. A partition (W_E, W_A) of infinite branches of T .

A play is any finite or infinite branch of T , starting from the root of T . In each node x of the branch, the player $\text{turn}(x)$ must select a child of x in T , otherwise his opponent wins. If a play continues forever, the winner is E if the play is in W_E , and it is A if the play is in W_A .

Games with 1-Backtracking

- Given a game G , we define a game **bck(G)**. In $\text{bck}(G)$, a player can come back from the last position z of the play to some previous position x , *provided x is before z in G* . Then he can replace the move y he did from x by some new move y' .



Infinite backtracking is loosing

- A player is allowed to backtrack to a given position x_i of the play only finitely many times.
- A player backtracking infinitely many times to the same position x_i loses.
- If $\mathcal{E}$ backtracks infinitely many times to some x_i and $\mathcal{A}$ infinitely many times to some x_j , the loser is the player backtracking infinitely many times to a position with smaller index.

Defining $\text{bck}(G)$

- **Positions of $\text{bck}(G)$.** All finite successions $\langle s_0, \dots, s_n \rangle$ over G , such that: *(i)* s_0 initial position of G *(ii)* any s_{i+1} is a child in G of some s_j , with $j \leq i$, *s_j ancestor in G of s_i* , and $\text{turn}(s_j) = \text{turn}(s_i)$.
- For each infinite play $\sigma = \langle s_0, \dots, s_n, \dots \rangle$ of $\text{bck}(G)$ we define a (finite or infinite) play $\sigma^{(1)} = \langle t_0, \dots, t_n, \dots \rangle$ of G by recursion on n :
 1. t_0 = initial position of G .
 2. t_{n+1} = last child of t_n in σ , provided t_n has a last child in σ . Otherwise $\sigma^{(1)}$ ends.

Defining $\text{bck}(G)$

- **Turn.** The player on turn on a position $\langle s_0, \dots, s_n \rangle$ of $\text{bck}(G)$ is the player on turn on s_n .
- **Winner of a infinite play.** The winner of an infinite play σ of $\text{bck}(G)$ is the winner of the (finite or infinite) play $\sigma^{(1)}$ of G associated to it.

1-Backtracking and Limit Computable Mathematic

- Let A be any arithmetical formula in the connectives $\exists, \vee, \forall, \wedge$. Let $G(A)$ be the Tarski game associated to A . Let LCM be Hayashi's Limit Computable Mathematic (or “Arithmetic with incremental learning”).
- **Theorem.** A is true in LCM if and only if $\mathcal{E}$ has a recursive winning strategy on $\text{bck}(G(A))$.

*1-backtracking characterizes
the set of formulas we can “learn”*

1-Backtracking and Recursive Degrees

- Let O be any recursive degree, G any game, p any player.
- **Theorem.** p has a winning strategy of degree O' for G if and only if p has a winning strategy of degree O for $\text{bck}(G)$.
- In the definition of a winning strategy:

1-backtracking = 1 quantifier less

1- Backtracking and Excluded Middle

- Let A be any arithmetical formula in the connectives $\exists, \vee, \forall, \wedge$. Let $G(A)$ be the Tarski game associated to A . Let HA be Intuitionistic Arithmetic. Let 1-EM be Excluded Middle for degree 1 formulas.
- **Theorem.** $\mathcal{E}$ has a winning strategy for $G(A)$ if, and only if, $HA + \omega\text{-rule} + 1\text{-EM} \vdash A$.

1-backtracking characterizes the set of intuitionistic consequences of 1-EM.

1-Backtracking and (unlimited) Backtracking

- 1-backtracking can be nested, even a transfinite number of times. For each game G and each ordinal α we can define $\text{bck}^\alpha(G)$.
- For any game H *with alternating players and no infinite plays*, T. Coquand defined $\text{Bck}(H)$, the game with *unlimited* backtracking over H .
- **Theorem.** $\text{bck}(H)$ is a subgame of $\text{Bck}(H)$, and $\text{Bck}(H)$ is a quotient of a subgame of $\text{bck}^{\omega_1}(H)$.

*Unlimited backtracking can be
obtained by iterating 1-backtracking.*

References

- **S. Hayashi and M. Nakata**, “*Towards Limit Computable Mathematics*”, Types for Proofs and Programs, LNCS 2277, pp. 125-144, 2001.
- **T. Coquand**, “*A semantics of evidence for classical arithmetic*”, Journal of Symbolic Logic 60, pp. 325-337, 1995.

Authors: Delia Kesner, Stéphane Lengrand

Title: A logical model for explicit operators

Abstract:

We present a simple term language with explicit operators for erasure, duplication and substitution enjoying a sound and complete correspondence with the intuitionistic fragment of Linear Logic's Proof Nets. We establish the good operational behaviour of the language by means of some fundamental properties such as confluence, preservation of strong normalisation, strong normalisation of well-typed terms and step by step simulation. This formalism is the first calculus with explicit substitutions having full composition and preserving strong normalisation.

Explicit operators and proof-nets

Delia Kesner, Stéphane Lengrand

PPS, Paris VII

18 December 2004

TYPES 2004

- Introduction
- Syntax and Typing in λ_{lr}
- Reduction in λ_{lr}
- Connexion with λ -calculus
- Strong Normalisation of λ_{lr} & Proof-nets modulo

Explicit substitutions and Strong Normalisation

ES used to compute β -reduction

$$(x.u)v \rightarrow u\langle x = v \rangle \quad + \text{ propagation of } u\langle x = v \rangle$$

but final instantiation can be delayed

$\Rightarrow$ ES used for unification (Dowek, Hardin, Kirchner)

Composition?

$$t\langle y = v \rangle\langle x = u \rangle \longrightarrow t\langle x = u \rangle\langle y = v\langle x = u \rangle \rangle$$

Not terminating.

$$\text{even } t\langle y = v \rangle\langle x = u \rangle \longrightarrow t\langle y = v\langle x = u \rangle \rangle \quad \text{if } x \notin t \text{ (Mellies)}$$

β -reduction and cut-elimination in proof-nets

Di Cosmo, Guerrini:

With equivalence classes of proof-nets (associativity of contraction, ...), cut-elimination

-is still terminating

-simulates β -reduction

But proof-nets have full composition!!

Full composition=

$$t\langle x = v \rangle \longrightarrow^* t\{x = v\}$$

even when t contains non-evaluated substitutions

Towards a terminating ES-calc with full composition

w_s : partial composition + strong normalisation

(Guillaume, David)

Does not simulate β -reduction exactly.

Mellies' counter-example avoided using *labels*

What are labels?

Typed case:

label = weakening of linear logic

“resource-aware” logic

Untyped case?...

... Resource control:

weakening

$\Rightarrow$ erasure operator

contraction?

$\Rightarrow$ duplication operator

$\Rightarrow$ lxr

Syntax & Typing

Syntax

$(Terms)$	$t, t' ::=$	x	<i>variable</i>
		$x.t$	<i>abstraction</i>
		$t\ t'$	<i>application</i>

Syntax

$(Terms)$	$t, t' ::=$	x	<i>variable</i>
		$x.t$	<i>abstraction</i>
		$t\ t'$	<i>application</i>
		$t\langle x = t' \rangle$	<i>substitution</i>

Syntax

$(Terms)$	$t, t' ::=$	x	<i>variable</i>
		$x.t$	<i>abstraction</i>
		$t\ t'$	<i>application</i>
		$t\langle x = t' \rangle$	<i>substitution</i>

$$(B) \quad (x.t)\ u \longrightarrow t\langle x = u \rangle$$

Syntax

$(Terms)$	$t, t' ::=$	x	<i>variable</i>
		$x.t$	<i>abstraction</i>
		$t\ t'$	<i>application</i>
		$t\langle x = t'\rangle$	<i>substitution</i>
		$W_x(t)$	<i>weakening</i>
		$C_x^{y,z}(t)$	<i>contraction</i>

$$(B) \quad (x.t)\ u \longrightarrow t\langle x = u\rangle$$

Syntax

$(Terms)$	$t, t' ::=$	x	<i>variable</i>
		$ \quad x.t$	<i>abstraction</i>
		$ \quad t \ t'$	<i>application</i>
		$ \quad t\langle x = t' \rangle$	<i>substitution</i>
		$ \quad W_x(t)$	<i>weakening</i>
		$ \quad C_x^{y,z}(t)$	<i>contraction</i>

$$(B) \quad (\ x.t) \ u \longrightarrow t\langle x = u \rangle$$

We only consider *well-formed* terms

- Linearity
- Compulsory presence

+ Barendregt's convention

$$\frac{}{x : A \vdash x : A} \quad (Axiom) \quad \frac{\Gamma, x : B \vdash t : A \quad \Delta \vdash M : B}{\Gamma, \Delta \vdash t \langle x = M \rangle : A} \quad (Subs)$$

$$\frac{\Gamma \vdash t : A \rightarrow B \quad \Delta \vdash v : A}{\Gamma, \Delta \vdash (t \ v) : B} \quad (App) \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash x.t : A \rightarrow B} \quad (Lambda)$$

$$\frac{\Gamma, x : A, y : A \vdash M : B}{\Gamma, z : A \vdash C_z^{x,y}(M) : B} \quad (Cont) \quad \frac{\Gamma \vdash t : A}{\Gamma, x : B \vdash W_x(t) : A} \quad (Weak)$$

Congruence & reduction

Congruence & propagation

Commutativity of substitutions:

$$t\langle x = u \rangle \langle y = v \rangle \equiv t\langle y = v \rangle \langle x = u \rangle$$

$$\text{if } y \notin FV(u) \ \& \ x \notin FV(v)$$

+ AC of contraction + C of weakening

System x

(Var)	$x\langle x = u \rangle$	$\longrightarrow$	u	
$(Weak1)$	$W_x(t)\langle x = u \rangle$	$\longrightarrow$	$W_{FV(u)}(t)$	
$(Cont1)$	$C_x^{y,z}(t)\langle x = u \rangle$	$\longrightarrow$	$C_{FV(u)}^{FV(u_1), FV(u_2)}(t\langle y = u_1 \rangle\langle z = u_2 \rangle)$	
	$C[t]\langle x = u \rangle$	$\longrightarrow$	$C[t\langle x = u \rangle]$	$x \in t$
$(Comp)$	$t\langle y = v \rangle\langle x = u \rangle$	$\longrightarrow$	$t\langle y = v\langle x = u \rangle \rangle$	$x \in FV(v)$

System r

	$C[W_x(t)]$	$\longrightarrow$	$W_x(C[t])$	x not bound by $C[]$
	$C_x^{y,z}(C[t])$	$\longrightarrow$	$C[C_x^{y,z}(t)]$	$y, z \in t$
$(Merge)$	$C_w^{y,z}(W_y(t))$	$\longrightarrow$	$R_z^w(t)$	
$(Cross)$	$C_w^{y,z}(W_x(t))$	$\longrightarrow$	$W_x(C_w^{y,z}(t))$	$x \neq y, \quad x \neq z$

Properties

Full composition

Free variables are preserved:

If $s \longrightarrow_{B_{\text{xr}}} s'$, then $FV(s) = FV(s')$. (Interface preserving)

Subject reduction:

If $\Gamma \vdash s : A$ et $s \longrightarrow_{B_{\text{xr}}} s'$, then $\Gamma \vdash s' : A$.

xr is convergent

Bxr is confluent

breaks Melliès' counter-example of non-terminating composition

Confluence on open terms? (terms with meta-variables)

Using composition. still checking all critical pairs

Connexion with λ -calculus

Encodings

$\mathcal{B}()$ hides resource control

$$\begin{array}{ccccc}
 & & \xrightarrow{\mathcal{B}()} & & \\
 t \twoheadrightarrow_{\text{xr}}^* W_{\Pi}(\mathcal{A}(\mathcal{B}(t))) & \lambda\text{lr} & & \lambda & t = \mathcal{B}(\mathcal{A}(t)) \\
 & & \xleftarrow{\mathcal{A}()} & &
 \end{array}$$

$\mathcal{A}()$ introduces resource operators

Simulations (1)

λ_{lr} λ

$$\begin{array}{ccc} t & & \mathcal{B}(t) \\ \downarrow B & \xrightarrow{\mathcal{B}()} & \downarrow \beta_* \\ t' & & \mathcal{B}(t') \end{array}$$

Simulations (2)

λ_{lr} λ

$$\begin{array}{ccc} t & & \mathcal{B}(t) \\ \downarrow_{\text{lr}} & \xrightarrow{\mathcal{B}()} & || \\ t' & & \mathcal{B}(t') \end{array}$$

Simulations (3)

$\lambda \mathbf{l} \mathbf{x} \mathbf{r}$ λ

$\mathcal{A}(t)$ t

$\downarrow B \mathbf{x} \mathbf{r} +$ $\xleftarrow{\mathcal{A}(\cdot)}$ $\downarrow \beta$

$W_{\Pi}(\mathcal{A}(t'))$ t'

Properties

Preservation of typing:

If $\Gamma, FV(t) \vdash t : A$ then $FV(t) \vdash W_\Gamma(\mathcal{A}(t)) : A$

If $\Gamma \vdash t : A$ then $\Gamma \vdash \mathcal{B}(t) : A$

Preservation of Strong Normalisation (PSN):

If $M \in SN^\beta$ then $\mathcal{A}(M) \in SN^{B_{\text{sr}}}$.

Proof-nets & Strong normalisation

Either directly from PSN, or...

Encoding types:

$$\begin{aligned} A^* &= A && \text{if } A \text{ is an atomic type} \\ (A \rightarrow B)^* &= ?((A^*)^\perp) \wp B^* \end{aligned}$$

Encoding terms:

... Sound and complete encoding $T(-)$ in proof-nets modulo

Simulation

- If $t \equiv t'$ then $T(t) \equiv_E T(t')$.
- If $t \longrightarrow_B t'$ then $T(t) \longrightarrow^+_{R_E} T(t')$
- If $t \longrightarrow_{\text{lr}} t'$ then $T(t) \longrightarrow^*_{R_E} T(t')$

Typed lr is strongly normalising.

Intersection types

Simply-typed +

$$\frac{\Gamma \vdash t : A_1 \quad \Gamma \vdash t : A_2}{\Gamma \vdash t : A_1 \cap A_2} \quad (\cap - I) \quad \frac{\Gamma \vdash t : A_1 \cap A_2}{\Gamma \vdash t : A_i} \quad (\cap - E_i)$$

to characterise strong normalisation: $t \in SN$ if and only if $\Gamma \vdash t : A$

true in λ -calculus (Dezani, Coppo)

false in $x: x\langle y = zz \rangle \langle z = \Omega \rangle \longrightarrow x\langle z = \Omega \rangle \longrightarrow x$ is SN

but same typing as $(\lambda y.x)(zz)\langle z = \Omega \rangle \longrightarrow (\lambda y.x)(\Omega\Omega)$ not SN

should be **true** again in λ_{lr} , because of composition.

Proof: SN from PSN (Proof-nets do not have intersection types)

Conclusion & future work

The λ_{lr} -calculus, in brief:

- An explicit substitution calculus *à la* λ_{x} , **confluent**...
...with explicit resource handling.
...simulating λ -calculus.
- Convergence of the propagation of resource operators
- Strong version of **composition**

Typed λ_{lr} -calculus:

Clear interpretation in linear logic - cut-elimination

$\implies$ **Strong Normalisation**

Future works

- Study strategies in `lxr`
- Resource control and optimal reductions
- Get rid of α -conversion? (c.f. Van Oostrom)
- Implementation: director strings?

References

- [CKP00] Roberto Di Cosmo, Delia Kesner, and Emmanuel Polonovski. Proof nets and explicit substitutions. *Lecture Notes in Computer Science*, 1784:63–??, 2000.
- [Gui99] B. Guillaume. *Un calcul de substitution avec Étiquettes*. PhD thesis, 1999.
- [HvO03] D. Hendriks and V. van Oostrom. Adbmal. In *Proceedings of the 19th Conference on Automated Deduction (CADE 19), Miami*, pages 136 – 150, July-August 2003. LNAI 2741.
- [Mel95] P.-A. Melliès. Typed λ -calculi with explicit substitution may not terminate. In M. Dezani, editor, *TLCA '95—Int. Conf. on Typed Lambda Calculus and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages

328–334, Edinburgh, Scotland, April 1995.
Springer-Verlag.

Congruence

- Associativity and commutativity of contractions:

$$\begin{aligned}C_w^{x,v}(C_x^{y,z}(t)) &\equiv C_w^{x,y}(C_x^{z,v}(t)) \\C_x^{y,z}(t) &\equiv C_x^{z,y}(t) \\C_{x'}^{y',z'}(C_x^{y,z}(t)) &\equiv C_x^{y,z}(C_{x'}^{y',z'}(t))\end{aligned}$$

- Commutativity of weakenings:

$$W_x(W_y(t)) \equiv W_y(W_x(t))$$

- Commutativity of substitutions:

$$t\langle x = u \rangle \langle y = v \rangle \equiv t\langle y = v \rangle \langle x = u \rangle$$

$$\text{si } y \notin FV(u) \text{ et } x \notin FV(v)$$

Authors: Luís Cruz-Filipe, Pierre Letouzey

Title: Executing Extracted Programs

Abstract:

It is a well-known fact that algorithms are often hidden inside mathematical proofs. If these proofs are formalized inside a proof assistant, then a mechanism called extraction can generate the corresponding programs automatically. In previous work, it has been explained how a program has been (with difficulty) produced from a formalization of the Fundamental Theorem of Algebra inside the Coq proof assistant. This program theoretically allows one to compute approximations of polynomial roots. But there is currently quite a gap between theory and practice. In this sequel work, we focus on studying the complexity of this program, and tried to analyze the reasons of its inefficiency by looking at selected sub-problems, like computation of series giving values of Euler constant e or $\sqrt{2}$.

Executing Extracted Programs

FTA vs. Extraction : Round II

Luís Cruz-Filipe¹

Pierre Letouzey²

¹ Center for Logic and Computation, IST Lisbon

² Mathematik Institut, LMU Munich

FTA : Fundamental Theorem of Algebra

“All non-constant polynomial over $\mathbb{C}$ has at least one root”

One impressive development (40 000 lines) at Nijmegen by Geuvers, Barendregt, Wiedijk, Pollack, Zwanenburg & Niqui.

Constructive proof $\Rightarrow$ a effective root search method

But extraction wasn't planned at first.

$\Rightarrow$ Huge efforts in order to obtain a program

Recall : FTA vs. Extraction, Round I

Cf. last TYPES talk and TPHOLs03 paper by C.F. & Spitters :

As result, a reasonable extracted source code (250kb)

But after :

- tweaking the logic of FTA : Prop vs. Set
- improving several proofs of FTA : real model, division, ...
- removing one bug in extraction : abusive constant unfolding

⇒ Already quite an achievement

New : FTA vs. Extraction, Round II

Is this program usable ? No ! Not yet ?

The whole FTA program is out of understanding.

⇒ Study of easier subproblems :

- computations of series limits, especially $e = \sum_0^{\infty} \frac{1}{n!}$
- computations via the IVT, in particular $\sqrt{2}$

⇒ No complete solutions yet, but some progress / new clues

Euler e : first tests & improvements

Unary/binary number inefficiencies :

- in the representation of rational numbers :

At first, $\mathbb{Q} = \mathbb{Z} \times \mathbb{N}$

Now, $\mathbb{Q} = \mathbb{Z} \times \mathbb{N}^*$

- in the generation of real numbers :

$\text{nring} : \mathbb{N} \rightarrow \mathbb{R}$

vs.

$\text{zring} : \mathbb{Z} \rightarrow \mathbb{R}$

- in the structure of proofs : at first, $1/n!$ was using

$\forall k : \mathbb{N}, k > 0 \rightarrow \text{nring } k \neq 0$

Better :

$\forall z : \mathbb{Z}, z > 0 \rightarrow \text{zring } z \neq 0$

Euler e : why bother ?

In fact, the problems may look rather artificial :
usually, it's obvious to inject $\mathbb{Q} \rightarrow \mathbb{R}$.

But FTA is split in two parts :

- the mathematical proofs where assumptions over $\mathbb{R}$ are minimalist (0, 1, +, ... plus a few axioms)
- a model based on Cauchy sequence implementing this minimalist interface

Modularity dilemma : rich or poor interface ?

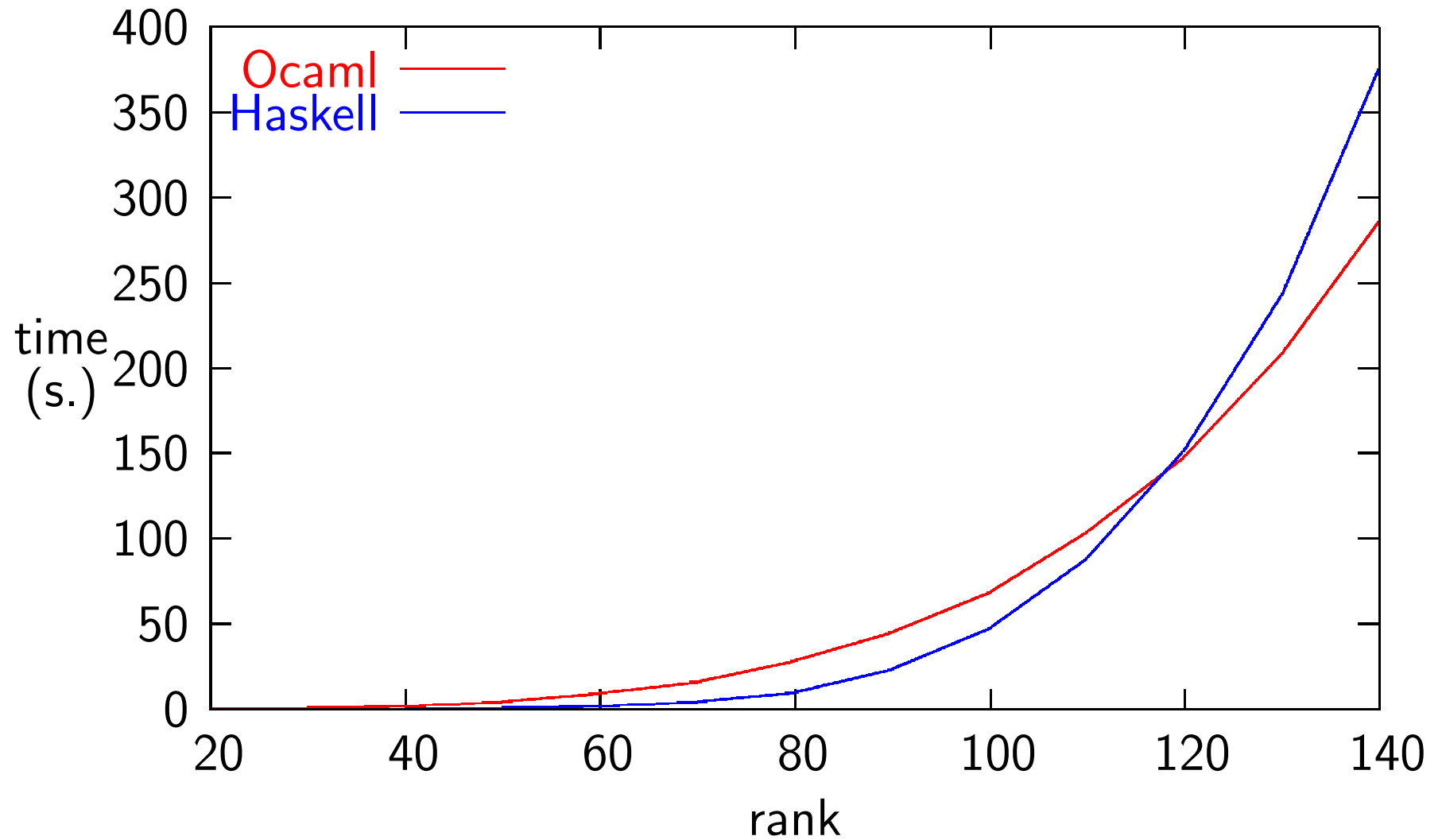
Euler e : the crucial change

We have finally shifted **in the model** a few critical functions, in order to access **definitions** of reals, of $\neq$, etc.

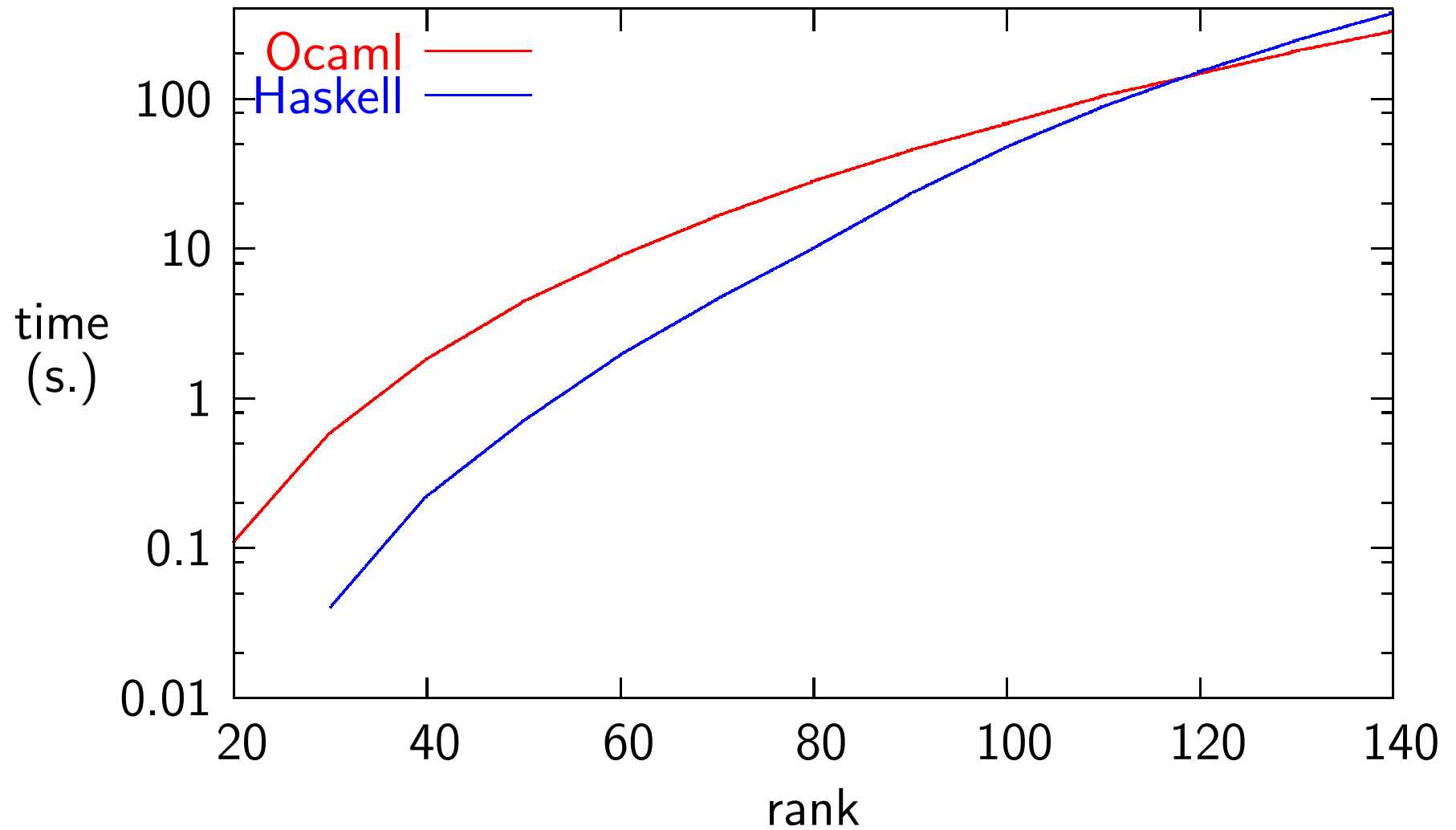
To use these optimized functions in the abstract part : axioms.

The extraction maps these axioms to the concrete functions.

Euler e : some statistics



Euler e : some statistics



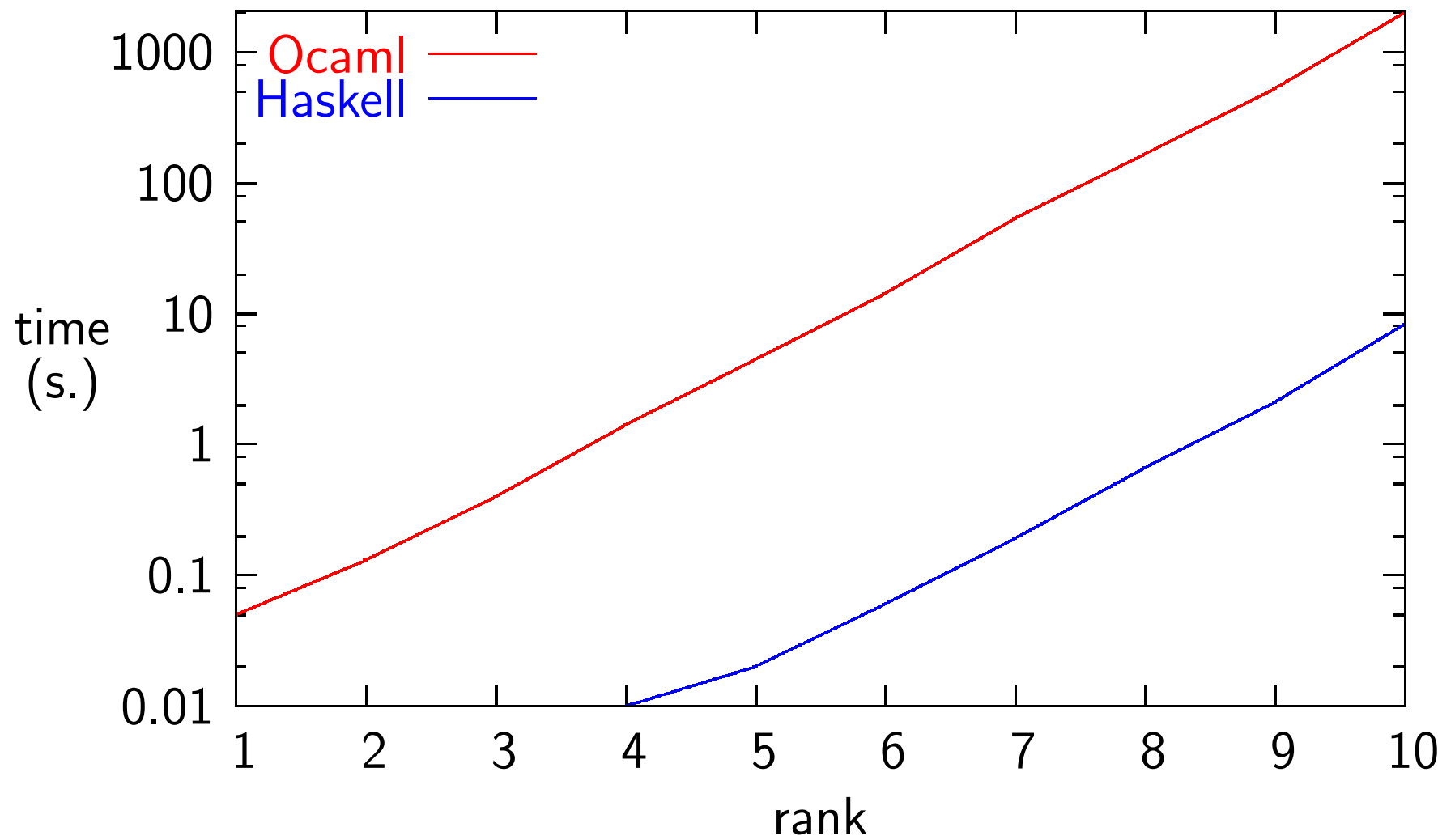
$\sqrt{2}$

The next stage in difficulty : real functions & IVT.

- A generic but awfully inefficient proof that every non-constant polynomial is ... locally non-constant, as required by the IVT. Here, instead, a simple monotonicity argument for $X^2 - 2$.
- As for Euler's e , some functions moved into the real number model.

But...

$\sqrt{2}$: some statistics



A new alternative $\sqrt{2}$ development

Small is ... dirty, but quick !

A short development inspired by H. Schwichtenberg : 400 lines of Coq, and 40 digits of $\sqrt{2}$ in 2 min !

Dirty :

- not finished (“sorry” !)
- really ad hoc. for example, only continuity of $X^2 - 2$.

But quite rich in lessons.

Could this approach be extended ? up to FTA ??

FTA : summary of current state

- A reasonable-sized extracted program, that compiles
- Root search : unusable
- Some encouraging intermediate results : series
- Can we extend up to the whole FTA ? Or would it be quicker to start anew ?

Conclusion : the quest for **good** proofs

What is a **good** proof ?

- efficient w.r.t extraction ?
- elegant : abstract / generic / short ... ?
- ... or just finished ?

The extraction is definitively an interesting method, but not a magic button.

Author: Helmut Schwichtenberg

Title: Program extraction in constructive analysis

Abstract:

We consider a formal arithmetical system with type and predicate parameters. Moreover inductive definitions with their introduction and elimination axioms are allowed. For this system we define extracted terms and the notion of (modified) realizability, and prove a soundness theorem. Then constructive analysis is developed in this setting, with special emphasis on low type level witnesses (which is possible when we assume separability). As an example we discuss the intermediate value theorem, and the form and usefulness of extracted programs from its proof.

Author: Stefan Berghofer

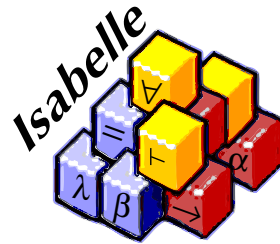
Title: Extracting a normalization algorithm from a proof of weak normalization for the simply-typed Lambda-calculus

Abstract:

Typed λ -calculi play an important role in computer science, both as a foundation for functional programming languages and for theorem provers. A particularly important result for typed λ -calculi is the normalization theorem, stating that each well-typed term can be reduced to a normal form. We describe the formalization of a proof of the weak normalization theorem for simply-typed λ -calculus based on de Bruijn indices in the theorem prover Isabelle/HOL. This proof is entirely constructive, and therefore contains an algorithm computing the normal form of a λ -term. We show how this algorithm, together with a proof of its correctness, can automatically be extracted from this proof within Isabelle/HOL.

Program extraction from a proof of weak normalization

Stefan Berghofer
Institut für Informatik, TU München



Background

Strong normalization

- Every reduction sequence starting from a well-typed term is **finite**
- Formalized using **accessible part**
- Extra work needed to obtain a normalization algorithm

$$e \vdash t : T \implies t \in acc (\rightarrow_{\beta}^{-1})$$

Weak normalization

- Every well-typed term has a normal form
- Proof corresponds to a **specific evaluation strategy**
- Can directly **extract** normalization algorithm from proof

$$e \vdash t : T \implies \exists t'. t \rightarrow_{\beta}^* t' \wedge t' \in NF$$

Problems

- Normalization proofs usually based on strong computability predicates (W. Tait, JSL 1967)

$$e \vdash s : T \Rightarrow U \wedge (\forall t. e \vdash t : T \wedge SC\ e\ t\ T \longrightarrow SC\ e\ U\ (s \circ t)) \longrightarrow SC\ e\ (T \Rightarrow U)\ s$$

- SC not admissible as an inductive definition, because SC occurs negatively
- SC must be defined by recursion on type expression
 - Requires strong elimination (i.e. definition of predicates by recursion)
 - May give rise to extracted programs using dependent types

Alternative approach (R. Matthes and F. Joachimski, AML 2003)

- Inductive characterization of normal forms
- Normalization proof is by induction on normal forms and type expressions
- Particularly well-suited for program extraction

Related work

- T. Altenkirch (TLCA 1993): A Formalization of the Strong Normalization Proof for System F in LEGO
- U. Berger (TLCA 1993): Extraction of a normalization by evaluation algorithm from a strong normalization proof in MINLOG
- R. Pollack (1994): The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions
- B. Barras, B. Werner: Coq in Coq
- C. Urban (2004): Formalization of a Tait-style strong normalization proof in Isabelle/HOL using nominal techniques
- T. Coquand (2004)
- H. Schwichtenberg (2004)

λ -terms with de Bruijn indices

datatype $dB = Var\ nat \mid dB \circ dB \mid Abs\ dB$

Notation

$$t^{\circ\circ} [t_1, \dots, t_n] = t \circ t_1 \circ \dots \circ t_n$$

$$t \uparrow i : \text{increment all indices } \geq i$$

$$t[u/i] : \text{substitute } u \text{ for variable } i \text{ in } t$$

$$t \rightarrow_{\beta} t' : \beta \text{ reduction}$$

Simple types

datatype $type = Atom\ nat \mid type \Rightarrow type$

Notation: $[T_1, \dots, T_n] \Rightarrow T = T_1 \Rightarrow \dots \Rightarrow T_n \Rightarrow T$

Type environments

- Modelled as functions from nat to $type$
- Inserting type T in environment e at position i :

$$e\langle i:T \rangle \equiv \lambda j. \text{ if } j < i \text{ then } e\ j \text{ else if } j = i \text{ then } T \text{ else } e\ (j - 1)$$

Typing judgement

$$VarT: e\ x = T \Longrightarrow e \vdash Var\ x : T$$

$$AbsT: e\langle 0:T \rangle \vdash t : U \Longrightarrow e \vdash Abs\ t : (T \Rightarrow U)$$

$$AppT: e \vdash s : T \Rightarrow U \Longrightarrow e \vdash t : T \Longrightarrow e \vdash (s \circ t) : U$$

Normal forms

$AppN: listall (\lambda t. t \in NF) ts \implies Var\ x \circ\circ\ ts \in NF$

$AbsN: t \in NF \implies Abs\ t \in NF$

where

$listall\ P\ xs \equiv (\forall i. i < length\ xs \longrightarrow P\ (xs\ !\ i))$

Computational content

datatype $NFT = Dummy \mid AppN\ (dB\ list)\ nat\ (nat \Rightarrow NFT) \mid AbsN\ dB\ NFT$

Type NFT used by **extracted program** to **represent normal forms**.

Main theorems

Well-typed substitution preserves existence of normal forms

lemma *subst-type-NF*: $t \in NF \implies e\langle i:U \rangle \vdash t : T \implies u \in NF \implies e \vdash u : U \implies \exists t'. t[u/i] \rightarrow_{\beta}^* t' \wedge t' \in NF$

Proof: Main induction on T , side induction on $t \in NF$

Type of program:

$dB \Rightarrow (nat \Rightarrow type) \Rightarrow nat \Rightarrow type \Rightarrow type \Rightarrow dB \Rightarrow NFT \Rightarrow NFT \Rightarrow dB \times NFT$

Well-typed terms have a normal form

theorem *type-NF*: $e \vdash_R t : T \implies \exists t'. t \rightarrow_{\beta}^* t' \wedge t' \in NF$

Proof: Induction on $e \vdash_R t : T$

Type of program: $rtypingT \Rightarrow dB \times NFT$

Correctness theorem:

$(x, e, t, T) \in rtypingR \implies t \rightarrow_{\beta}^* fst (type-NF\ x) \wedge fst (type-NF\ x) \in NF$

Note: Typing derivation computationally irrelevant in proof of *subst-type-NF*
relevant *type-NF*

Proof of the application case

How to normalize $(\underbrace{\text{Var } i}_{T'} \circ a \circ\circ as)[u/i] = u \circ a[u/i] \circ\circ as[u/i]$
 $T' \Rightarrow Ts \Rightarrow T$

1. Substitute and normalize tail of argument list
 (by side induction)

$$as[u/i] \rightarrow_{\beta}^* as'$$

2. Substitute and normalize first argument
 (by side induction)

$$a[u/i] \rightarrow_{\beta}^* a'$$

3. Apply head term to result term, normalize again
 (by main induction)

$$u \circ \underbrace{a'}_{T'} \rightarrow_{\beta}^* ua$$

4. Apply result term to normalized argument list,
 normalize again (by main induction)

$$\underbrace{ua}_{Ts \Rightarrow T} \circ\circ as' \rightarrow_{\beta}^* r$$

Computational content of lemma

$subst\text{-}type\text{-}NF \equiv \lambda x\ xa\ xb\ xc\ xd\ xe\ H\ Ha.$

$type\text{-}induct\text{-}P\ xc\ (\lambda x\ \textcolor{red}{H2}\ \textcolor{red}{H2a}\ xa\ xb\ xc\ xd\ xe\ H.$

$NFT\text{-}rec\ arbitrary$

$(\lambda ts\ xa\ xaa\ \textcolor{red}{r}\ xb\ xc\ xd\ xe\ H. var\text{-}app\text{-}typesE\text{-}P\ (xb\langle xe:x\rangle)\ xa\ ts$

$(\lambda Us. case\ nat\text{-}eq\text{-}dec\ xa\ xe\ of$

$Left \Rightarrow case\ ts\ of\ [] \Rightarrow (xd, H)$

$| a \# list \Rightarrow$

$case\ Us\ of\ [] \Rightarrow arbitrary$

$| T'' \# Ts \Rightarrow$

$let\ (x, y) = norm\text{-}list\ (\lambda t. t \uparrow 0)\ xd\ xb\ xe\ list\ Ts\ (\lambda t. lift\text{-}NF\ 0)\ H$

$(listall\text{-}conj2\text{-}P\text{-}Q\ list\ (\lambda i. (xaa\ (Suc\ i), \textcolor{red}{r}\ (Suc\ i))));$

$(xa, ya) = snd\ (xaa\ 0, \textcolor{red}{r}\ 0)\ xb\ T''\ xd\ xe\ H;$

$(xd, yb) = app\text{-}Var\text{-}NF\ 0\ (lift\text{-}NF\ 0\ H);$

$(xa, ya) = \textcolor{red}{H2}\ T''\ (Ts \Rightarrow xc)\ xd\ xb\ (Ts \Rightarrow xc)\ xa\ 0\ yb\ ya;$

$(x, y) = \textcolor{red}{H2a}\ T''\ (Ts \Rightarrow xc)\ (Var\ 0\ \circ\circ\ map\ (\lambda t. t \uparrow 0)\ x)\ xb\ xc$
 $xa\ 0\ (y\ 0)\ ya$

$in\ (x, y)$

$| Right \Rightarrow \dots)$

$(\lambda t\ x\ r\ xa\ xb\ xc\ xd\ H. \dots)$

$x\ xa\ xd\ xe\ xb\ H\ Ha$

dummy case

application case

head variable affected

empty argument list

non-empty argument list

ill-typed term

head variable not affected

abstraction case

Proof of the main theorem

theorem *type-NF*: **assumes** $T: e \vdash_R t : T$

shows $\exists t'. t \rightarrow_{\beta^*} t' \wedge t' \in NF$ **using** T

proof *induct*

case (*AppRT* $T\ U\ e\ s\ t$)

from *AppRT* **obtain** $s'\ t'$ **where** $sred: s \rightarrow_{\beta^*} s'$ **and** $sNF: s' \in NF$

and $tred: t \rightarrow_{\beta^*} t'$ **and** $tNF: t' \in NF$ **by** *rules*

have $\exists u. (Var\ 0 \circ t' \uparrow 0)[s'/0] \rightarrow_{\beta^*} u \wedge u \in NF$

proof (*rule subst-type-NF*)

$\vdots$

qed

then obtain u **where** $ured: s' \circ t' \rightarrow_{\beta^*} u$ **and** $unf: u \in NF$ **by** *simp rules*

from $sred\ tred$ **have** $s \circ t \rightarrow_{\beta^*} s' \circ t'$ **by** (*rule rtrancl-beta-App*)

hence $s \circ t \rightarrow_{\beta^*} u$ **using** $ured$ **by** (*rule rtrancl-trans*)

with unf **show** *?case* **by** *rules*

next

$\dots$

qed

Computational content of main theorem

$type\text{-}NF \equiv$

$\lambda H. rtypingT\text{-}rec (\lambda e x T. (Var x, Var\text{-}NF x))$

variable case

$(\lambda e T t U x r. let (\textcolor{red}{x}, \textcolor{red}{y}) = r in (Abs \textcolor{red}{x}, AbsN \textcolor{red}{x} \textcolor{red}{y}))$

abstraction case

$(\lambda e s T U t x xa \textcolor{red}{r} \textcolor{red}{ra}.$

application case

$let (\textcolor{red}{x}, \textcolor{red}{y}) = \textcolor{red}{r}; (xa, ya) = \textcolor{red}{ra};$

$in subst\text{-}type\text{-}NF (Var 0 \circ \textcolor{red}{xa} \uparrow 0) e 0 (T \Rightarrow U) U \textcolor{red}{x}$

$(AppN [\textcolor{red}{xa} \uparrow 0] 0 (listall\text{-}cons\text{-}P (lift\text{-}NF 0 \textcolor{red}{ya}) listall\text{-}nil\text{-}P)) \textcolor{red}{y})$

H

Benchmark: Multiplication of Church numerals

$$2 =_{def} (\lambda f. \lambda x. f (f x))$$

$$\star =_{def} (\lambda m. \lambda n. \lambda f. m (n f))$$

$$x \star 2 \rightarrow_{\beta}^* x'$$

Runtime of normalization function

x	extracted	hand-written
2	0.002	0.000
4	0.016	0.000
8	0.012	0.000
16	0.036	0.000
32	0.096	0.000
64	0.430	0.000

x	extracted	hand-written
128	2.615	0.000
256	27.786	0.001
512	364.193	0.002
1024	?	0.007
2048	?	0.010
4096	?	0.030

Done on a Pentium III with 1 GHz

Conclusion

- **Realistic** example for program extraction
- Relatively **compact** proof (800 lines of Isabelle code)
- Extracted program **reasonably efficient** for medium-size input

Future work

- Increase efficiency of normalization function
 - Eliminate **redundant arguments** (e.g. typing information)
May require **computationally irrelevant universal quantifiers**
 - Avoid re-examining terms
 - Avoid generating **garbage**
- Extract other normalization algorithms, e.g. **NBE**
- Other encodings, e.g. **nominal techniques**

Types 2004
15-18.12.4

A minimalist FOUNDATION
FOR
CONSTRUCTIVE MATHEMATICS

M. Huet & G. Sambin

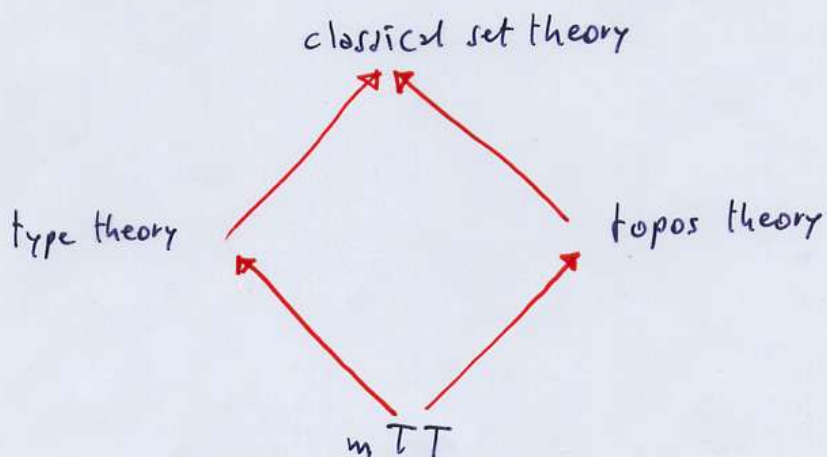
T satisfies proofs-as-programs
 $\Rightarrow T$ consistent with $AC+CT$

$$(\pm \text{Ext}) AC: \forall x \exists y R(x, y) \rightarrow \exists f \forall x R(x, f x)$$

$$CT: \forall f: N \rightarrow N \exists e \forall n (T(e, n, z) \rightarrow z = f n)$$

but $\left. \begin{array}{l} ZF \\ \Pi_1^1 L_{84} \\ \text{topos th.} \end{array} \right\} + AC + CT \vdash \perp$

$$\text{ExtPow} + AC \vdash EM$$



abandon propositions-as-sets

$\exists$ via the reflection principle

$$\Gamma, (\exists x \in D) A(x) \vdash \Delta \quad \text{iff} \quad \text{for every } d \in D, \\ \Gamma, A(d) \vdash \Delta$$

$$\Gamma, (\exists x \in D) A(x) \vdash \Delta \quad \text{iff} \quad \Gamma, z \in D, A(z) \vdash \Delta \\ z \text{ not free in } \Gamma, \Delta$$

$$\exists\text{-formation} \quad \frac{\Gamma, z \in D, A(z) \vdash \Delta}{\Gamma, (\exists x \in D) A(x) \vdash \Delta}$$

$\exists$ -implicit reflection

$$\frac{\Gamma, (\exists x \in D) A(x) \vdash \Delta}{\Gamma, z \in D, A(z) \vdash \Delta} \quad \begin{array}{l} \text{trivialize} \\ \Gamma = \emptyset \\ \Delta = (\exists x \in D) A(x) \end{array} \quad \begin{array}{l} \text{compose} \\ \text{cut } (\exists x \in D) A(x) \end{array}$$

$$\exists\text{-axiom} \quad z \in D, A(z) \vdash (\exists x \in D) A(x)$$

$\exists$ -explicit reflection

$$\frac{\Gamma \vdash z \in D \quad \Gamma' \vdash A(z)}{\Gamma, \Gamma' \vdash (\exists x \in D) A(x)} \quad \begin{array}{l} \text{compose} \\ \text{cut } z \in D \\ A(z) \end{array} \quad \begin{array}{l} \text{trivialize} \\ \Gamma = z \in D \\ \Gamma' = A(z) \end{array}$$

minimal Type Theory mTT

sets: exactly as in Martin-Löf

Σ -elimination

$$\frac{\begin{array}{l} [z \in (\Sigma_{x \in B} C(x)) \\ M(z) \text{ set} \end{array} \quad \begin{array}{l} [x \in B, y \in C(x)] \\ d \in (\Sigma_{x \in B} C(x)) \quad m(x, y) \in M(\langle x, y \rangle) \end{array}}{\Sigma_{\text{elim}}(d, m) \in M(d)}$$

propositions: intuitionistic logic, A prop

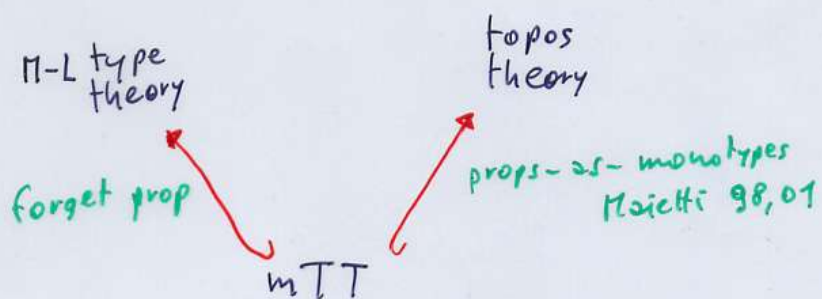
$\exists$ -elimination

$$\frac{\begin{array}{l} M \text{ prop} \\ d \in (\exists x \in B) C(x) \end{array} \quad \begin{array}{l} [x \in B, y \in C(x)] \\ m(x, y) \in M \end{array}}{\exists_{\text{elim}}(d, m) \in M}$$

link: propositions $\rightarrow$ sets

$$\frac{A \text{ prop}}{A \text{ set}}$$

Compatibility:



toolbox: that for M-L type th.

normalization: that for M-L type th.

type checking: open, but seems clear

realizability interpretation: open, but have suideo

other literature: Howard 69-79
Mitchell-Plotkin ?
Aczel-Gambino 02
??
...

Author: Aarne Ranta

Title: Declarative Language Definitions and Code Generation by Linearization

Abstract:

We present a compiler for a fragment of the C programming language, with an assembler notation for JVM (Java Virtual Machine) as target language. The compiler is implemented in a purely declarative way: its definition consists of an abstract syntax of programs and two concrete syntaxes matching the abstract syntax: one for C and one for JVM. From these grammar components, the compiler is derived by using the GF (Grammatical Framework) grammar tool: the front end consists of parsing and semantic checking in accordance to the C grammar, and the back end consists of linearization in accordance to the JVM grammar. The tool provides other functionalities as well, such as decompilation and interactive editing.

Declarative Language Definitions and Code Generation as Linearization

TYPES Meeting, Jouy-en-Josas, December 15–18, 2004.

Aarne Ranta

`aarne@cs.chalmers.se`

The Challenge

3. Write a C compiler. The input language is a subset of C. A valid input program should produce byte code that when executed runs the C program.

Lennart Augustsson, programming challenges posted to the participants of the Dagstuhl workshop on Dependent Types in Programming, 2004.

Example program: C source

```
int factr (int n) {
    int f ;
    if (n < 2) {
        f = 1 ;
    }
    else {
        f = n * factr (n-1) ;
    }
    return f ;
} ;

int main () {
    int n = 1 ;
    while (n < 11) {
        printf("%d",factr(n)) ;
        n = n+1 ;
    }
    return ;
} ;
```

Compiling the program

```
[aarne@localhost compiler]$ gfc -c factr.c
```

```
> > wrote file factr.o
```

```
Generated: factr.class
```

Generated JVM bytecode (Jasmin assembler format)

```
.class public factr
.super java/lang/Object
.method public <init>()V
aload_0
invokenonvirtual java/lang/Object/<init>()V
return
.end method
.method public static factr(I)I
.limit locals 3
.limit stack 1000 ;
; alloc i n i ;
; alloc i f ;
iload 0
ldc 2 ;
invokestatic runtime/ilt(II)I
ifeq FALSE_
ldc 1 ;
istore 1
```

```
goto TRUE_  
FALSE_:  
  iload 0  
  iload 0  
  ldc 1 ;  
  isub  
  invokestatic factr/factr(I)I  
  imul  
  istore 1  
TRUE_:  
  iload 1  
  ireturn  
.end method ;
```

```
.method public static main([Ljava/lang/String;)V  
.limit locals 2  
.limit stack 1000 ;  
; alloc i n ;  
ldc 1 ;  
istore 1
```

```
TEST_:
  iload 1
  ldc 11 ;
  invokestatic runtime/ilt(II)I
  ifeq END_
  iload 1
  invokestatic factr/factr(I)I
  invokestatic runtime/iprintf(I)V
  iload 1
  ldc 1 ;
  iadd
  istore 1
  goto TEST_
END_:
  return ;
.end method ;
```

Running the bytecode

```
[aarne@localhost compiler]$ java factr
```

1

2

6

24

120

720

5040

40320

362880

3628800

Compiler Passes

`x + 5`

Lexer

`[TV "x", TS "+", TI "5"]`

Parser

`EAdd (EVar "x") (ENum 5)`

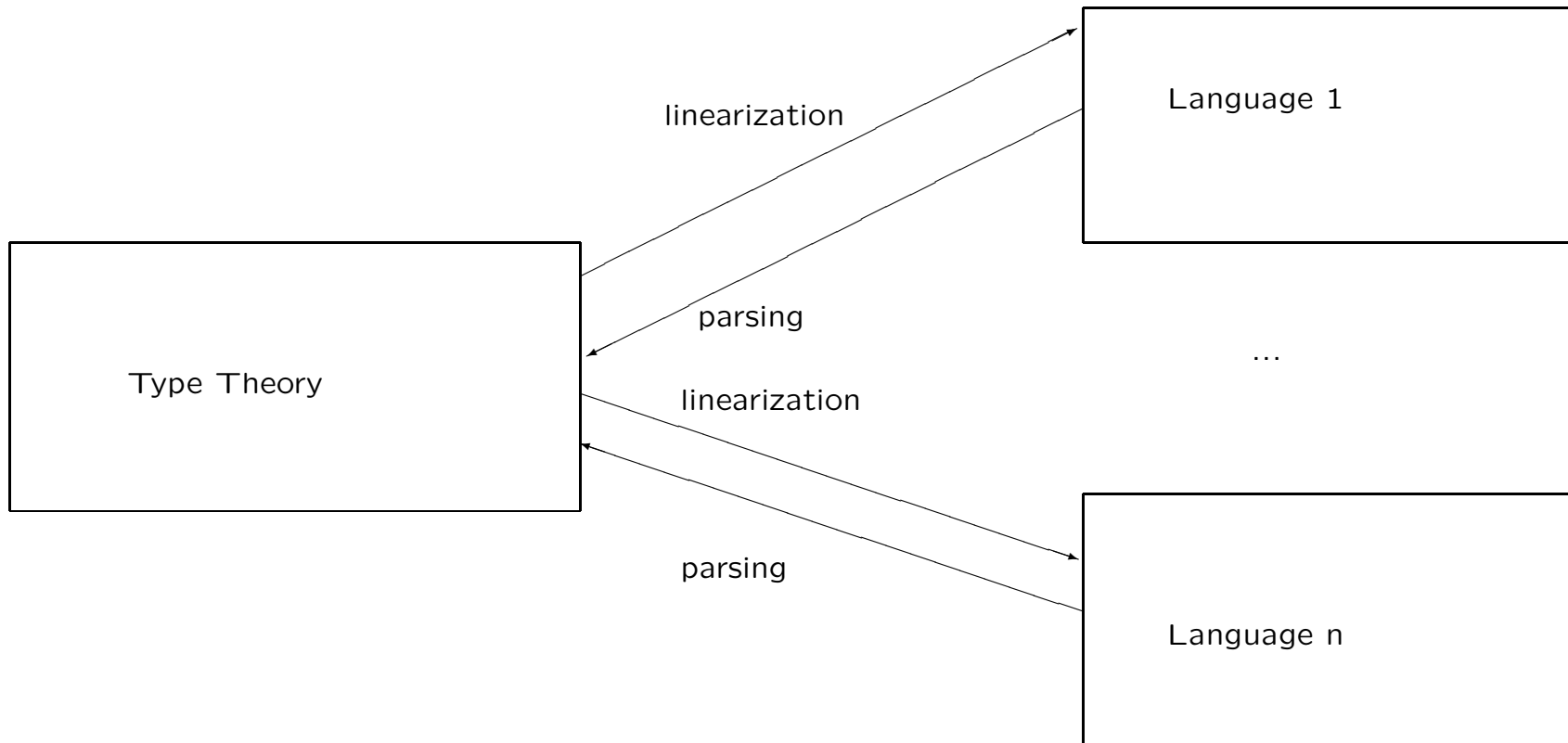
Type annotator

`EAddT TInt (EVarT TInt "x") (ENum TInt 5)`

Code generator

`iload_1 ; bipush 5 ; iadd`

GF = Grammatical Framework



Example GF rules

Abstract syntax:

```
fun Ev : Nat -> Prop ;
```

Concrete syntax of English:

```
lin Ev x = {s = x.s ++ "is" ++ "even"} ;
```

Concrete syntax of French:

```
lin Ev x = {  
  s = table {m => x.s ++  
    table {Ind  => "est" ; Subj => "soit"} ! m ++  
    table {Masc => "pair" ; Fem  => "paire"} ! x.g  
  }  
} ;
```

Abstract and concrete syntax

Abstract Syntax is defined in a Logical Framework:

```
cat C (x1 : A1) ... (xn : An)
fun f : A
```

Concrete syntax is defined by linearization:

```
lincat C = T
lin    f = t
```

Notice: linearization type T instead of just strings.

Linearization

Well-typedness:

$$\frac{f: A_1 \rightarrow \dots \rightarrow A_n \rightarrow A}{f^\circ: A_1^\circ \rightarrow \dots \rightarrow A_n^\circ \rightarrow A^\circ}$$

Linearization algorithm:

$$(f \ a_1 \ \dots \ a_n)^\circ = f^\circ \ a_1^\circ \ \dots \ a_n^\circ$$

Expressions: abstract syntax

Expressions of a type; the “class” of numeric types

cat

Typ ;

Exp Typ ;

IsNum Typ ;

fun

EAdd : (n : Typ) -> IsNum n -> Exp n -> Exp n -> Exp n ;

TInt : NumTyp ;

isNumInt : IsNumTyp TInt ;

The parsing and type annotation phases

`x + 5`

Lexer + parser

`EAdd ? ? (EVar ? x) (EInt 5)`

Type checker + metavariable solver

`EAdd TInt isNumInt (EVar TInt x) (EInt 5)`

Expressions: concrete syntax of C

Type: record with string and precedence

```
lincat Exp = {s : Str ; p : Prec} ;
```

```
oper    Prec : PType = Predef.Ints 4 ; -- = {0,1,2,3,4}
```

Example:

```
lin EInt n = {s = n.s ; p = 4} ;
```

Auxiliary operations for precedence

```
PrecExp : Type = {s : Str ; p : Prec} ;
```

```
usePrec : PrecExp -> Prec -> Str = \x,p ->  
  ifThenStr (less x.p p)  
    (paren x.s)  
    x.s ;
```

```
infixL : Prec -> Str -> (_,_ : PrecExp) -> PrecExp = \p,op,x,y ->  
  {s = usePrec x p ++ op ++ usePrec y (nextPrec p) ;  
   p = p  
  } ;
```


Some expression linearizations

lin

EMul _ _ = infixL 3 "*" ;

EAdd _ _ = infixL 2 "+" ;

ESub _ _ = infixL 2 "-" ;

Expressions: concrete syntax of JVM

lincat

Exp = {s : Str} ;

Typ = {s : Str ; t : TypIdent} ;

lin

EAdd t _ = binopt "add" t.t ; -- x ; y ; iadd ;

ESub t _ = binopt "sub" t.t ;

EMul t _ = binopt "mul" t.t ;

Auxiliary operations for JVM instructions

oper

```
binopt : Str -> TypIdent -> SS -> SS -> SS = \op, t -> {s =  
  x.s ++  
  y.s ++  
  typInstr op t  
} ;
```

```
typInstr : Str -> TypIdent -> Str = \instr,t -> case t of {  
  TIInt      => "i" + instr ;  
  TIFloat    => "f" + instr ;  
  TIDouble   => "d" + instr  
} ;
```

Declarations: abstract syntax

Use higher-order abstract syntax

cat

Stm ;

Var Typ ;

fun

Decl : (A : Typ) -> (Var A -> Stm) -> Stm ;

Using variables

fun

Assign : (A : Typ) -> Var A -> Exp A -> Stm -> Stm ;

EVar : (A : Typ) -> Var A -> Exp A ;

Example of binding

```
int x ;      Decl TInt (\x ->
x = 5 ;      Assign TInt x (EInt 5) (
return x ;    Return TInt (EVar TInt x)))
```

Linearization of variable bindings

We assume η -long form of syntax trees.

We add a record field for each variable symbol

$$(\lambda x_0 \rightarrow \cdots \rightarrow \lambda x_n \rightarrow b)^{\circ} = b^{\circ} ** \{\$ _0 = x_0^{\circ}; \dots; \$ _n = x_n^{\circ}\}$$

Variable bindings: an example record

The term

```
(\x -> Assign TInt x (EInt 5) (  
    Return TInt (EVar TInt x)))
```

linearizes in C to

```
{s = ["x = 5 ; return x ;"] ; $0 = "x"}
```


Variable bindings: an example rule

The declaration function

```
fun Decl : (A : Typ) -> (Var A -> Stm) -> Stm ;
```

has in C the linearization rule

```
lin Decl typ stm = {s = typ.s ++ stm.$0 ++ ";" ++ stm.s} ;
```

A problem with higher-order abstract syntax

With normal scoping rules of type theory, we permit

```
int x ;      Decl int    (\x ->
float x ;    Decl float (\x ->
x = 3.14 ;   Assign float x (EFloat 3 14) ...))
```

which is not allowed in C.

Top-level program structure

Program = sequence of function definitions
(potentially recursive ones)

```
int factr (int n) { ... }  
...
```

Bindings on two levels:

function symbols: `factr`

local parameters: `n`

Abstract syntax of program structure

cat

```
Program ;  
ListTyp ;  
Body ListTyp ;  
Fun ListTyp Typ ;
```

fun

```
Empty : Program ;  
Funct : (AS : ListTyp) -> (V : Typ) ->  
        (Fun AS V -> Body AS) -> Program ;  
FunctNil : (V : Typ) ->  
           Stm -> (Fun NilTyp V -> Program) -> Program ;  
RecOne   : (A : Typ) ->  
           (Var A -> Stm) -> Program -> Body (ConsTyp A NilTyp) ;  
RecCons  : (A : Typ) -> (AS : ListTyp) ->  
           (Var A -> Body AS) -> Program -> Body (ConsTyp A AS) ;
```

Loops: JVM code generatoin

```
lincat
  Stm = {s,s2,s3 : Str} ; -- code, storage size, labels
lin
  While exp loop =
    let
      test = "TEST_" ++ loop.s2 ;
      end   = "END_"   ++ loop.s2
    in instr1 (
      "label" ++ test ++ ";" ++
      exp.s ++
      "ifeq" ++ end ++ ";" ++
      loop.s ++
      "goto"  ++ test ++ ";" ++
      "label" ++ end
    ) ;
```

JVM code generation: the main problem

Variables are represented by stack addresses:

```
iload 1
```

```
istore 1
```

Solution: Symbolic JVM

Assembler format with variable symbols instead of stack addresses

easy linearization to Symbolic JVM

easy assembly of Symbolic JVM to Jasmin JVM

We wrote a Haskell program for the latter.

From symbolic JVM to Jasmin assembler

int x ;	alloc i x	; x gets address 0
int y ;	alloc i y	; y gets address 1
x = 5 ;	ldc 5	ldc 5
	istore x	istore 0
y = x ;	iload x	iload 0
	istore y	istore 1

Augustsson's Challenge 1

1. Write a printf function handling %d and %s.

```
fun Printf : (A : Typ) -> Exp A -> Stm -> Stm ;
```

```
lin Printf t e =  
  continues ("printf" ++ paren (t.s2 ++ "," ++ e.s)) ;
```

```
lin Printf t e =  
  instrc (e.s ++  
    "runtime" ++ typInstr "printf" t.t ++ paren (t.s) ++ "V"  
  ) ;
```

Code statistics

Code specific for this compiler

51	368	1677	Imper.gf
56	355	1764	ImperC.gf
93	535	2870	ImperJVM.gf
200	1258	6311	total

Code reusable in other compilers

85	584	2624	ResImper.gf
57	295	1835	Assemble.hs

What else we have achieved

Syntax editing (C and JVM in parallel!)

Decompilation (from Symbolic JVM)

Well-typedness of generated bytecode

Compiled parser ($GF \rightarrow BNFC \rightarrow \text{Happy/YACC}$)

What we cannot do

Non-compositional instruction selection

Prospects

Can this toy be extended to a tool?

I.e. a tool for implementing programming languages.

Author: Björn Bringert

Title: Building Multimodal Dialogue Systems in GF

Abstract:

Multimodal dialogue systems are interactive systems which support several modes of communication with the user, for example speech, mouse clicks and drawings. We present a method for writing multimodal grammars in the Grammatical Framework (GF) and for using such grammars in dialogue systems. Parallel multimodality, where the same information is represented in multiple modalities, is achieved by using multiple concrete syntaxes for the same abstract syntax. Integrated multimodality, where the different modalities are used together to represent some information, is done by using record types. We will also present an example multimodal dialogue system implemented using these techniques. This work is part of the TALK project (<http://www.talk-project.org/>).

Building Multimodal Dialogue Systems in Grammatical Framework

Björn Bringert
Chalmers University of Technology

with Aarne Ranta and Robin Cooper

EU TALK Project, IST-507802

What are Dialogue Systems?

- User interaction
- Involve natural language
- Examples:
 - automated flight information
 - car navigation systems
 - “smart home” systems
- Running example: Gothenburg tram map

What is Multimodality?

- Using multiple modes of communication:
 - Speech
 - Pointing
 - Drawings
 - Text
 - ...

What is Grammatical Framework?

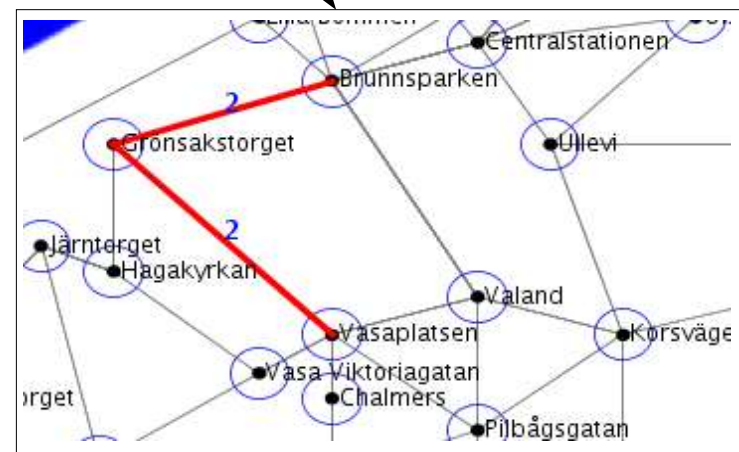
- Grammar formalism
- Based on type theory
- Abstract syntax
- Concrete syntax
 - Linearization functions
- Parsing for free

Parallel Multimodality

- Complete information in each modality:

Route 2 [Brunnsparken, Grönsakstorget, Vasaplatsen]

“Take line three
from Brunnsparken
to Vasaplatsen”



Parallel Multimodality in GF

- One concrete syntax / modality (like multilinguality):

- Abstract syntax:

```
fun Leg : Line -> Stop -> Stop -> Route
```

- English speech:

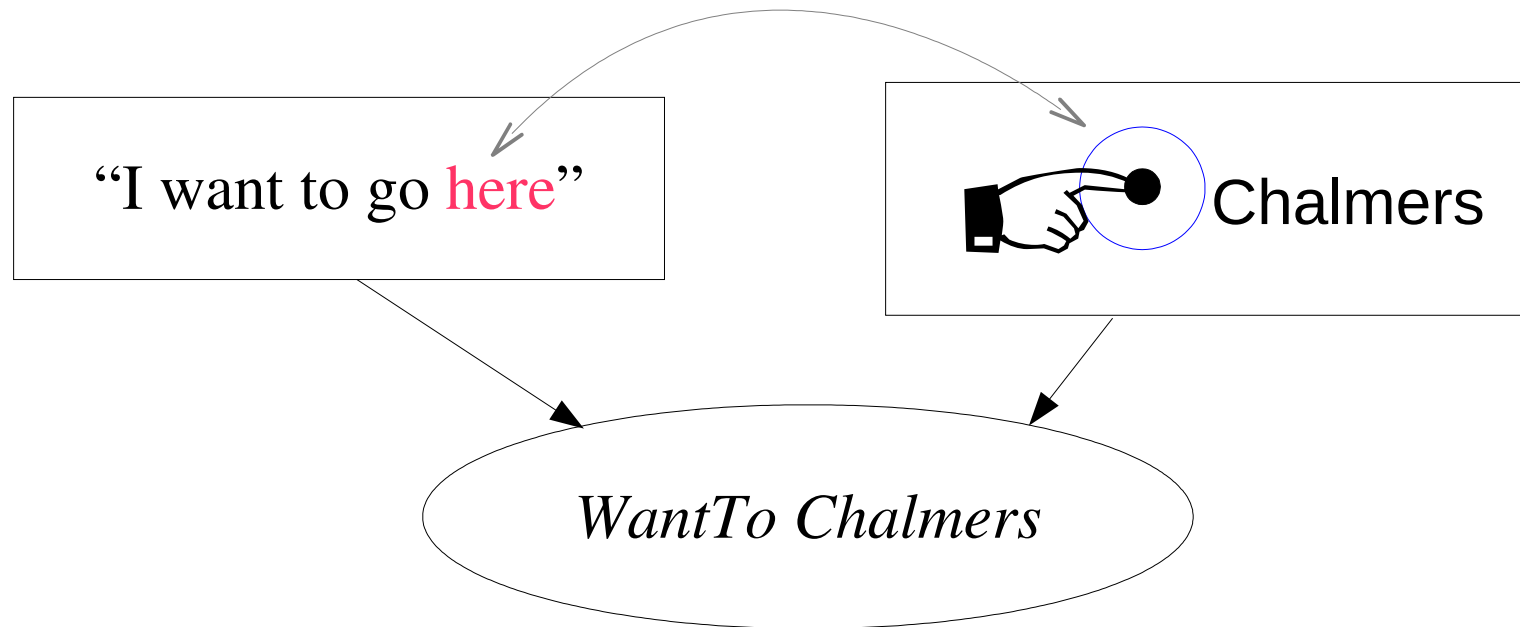
$$lin\ Leg\ lft = \{ s = \text{“Take”} ++ l.s ++ \text{“from”} ++ f.s \\ ++ \text{“to”} ++ t.s \}$$

- Map drawings:

$$\textit{lin Leg lft} = \{ s = \textit{“drawEdge(“} ++ f.s ++ \textit{“,”} \\ ++ t.s ++ \textit{“,”} \\ ++ l.s ++ \textit{“)”} \}$$

Integrated Multimodality

- Information presented by a combination of modalities:



Integrated Multimodality in GF

- Abstract syntax:

cat Input

cat Place

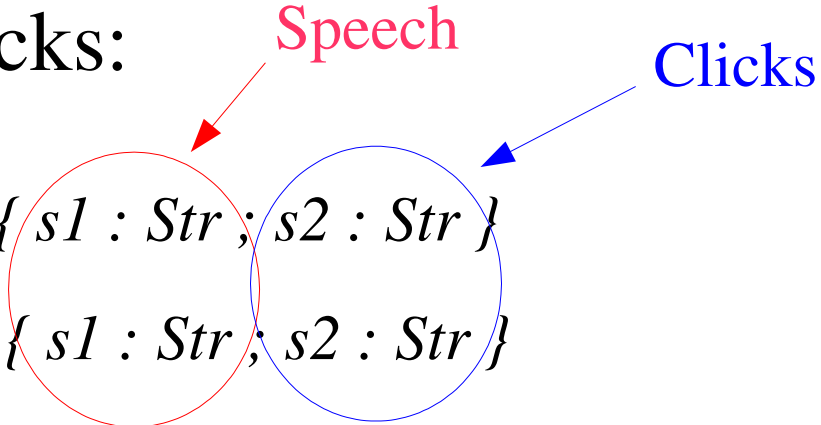
fun GoFromTo : Place -> Place -> Input

fun NamedPlace : String -> Place

fun ClickPlace : Click -> Place

Integrated Multimodality in GF

- English + clicks:



lincat Input = { *s1* : *Str* ; *s2* : *Str* }

lincat Place = { *s1* : *Str* ; *s2* : *Str* }

lin GoFromTo *x y* = {
 s1 = ["I want to go from"] ++ *x.s1* ++ "to" ++ *y.s1* ;
 s2 = *x.s2* ++ *y.s2*
}

lin NamedPlace *p* = { *s1* = *p.s* ; *s2* = "" }

lin ClickPlace *c* = { *s1* = "here" ; *s2* = *c.s* }

Demo: Gothenburg tram map

Author: Georgi Jojgov

Title: Towards a translation of Weak Type Theory into Dependent Type Theory

Abstract:

Weak Type Theory (WTT) is a formal language that allows us to write mathematical texts and check their conformity to very weak correctness criteria of linguistic nature. This makes WTT potentially suitable as a front-end language for tools for formal mathematics. To realize this potential, we need procedures to (semi-)automatically translate WTT texts into the languages of these tools. In this talk I will present an ongoing work on the translation of a fragment of WTT into dependent type theory with open terms and discuss the problems and the benefits of such a translation.



Towards a translation of Weak Type Theory into Dependent Type Theory

Georgi Jojgov

G.I.Jojgov@tue.nl

(work in progress)

TYPES 2004

Jouy-en-Josas, 18 December 2004

Motivation

Study the process of formalizing of informal math texts

- (Input) Informal math text in CML (e.g. English)
- (Output) Formalization in a proof assistant

We want as much as possible:

- Reliability
- Usability by Humans
- Computer Assistance

Motivation

Phased formalization:

Start with very weak correctness criterion
(To reduce the amount of work on the informal level)

Define one or more intermediate steps that require increasingly stronger correctness to be demonstrated.

The last stage is a completely formalized proof in a proof assistant.

Overview

- Phased Formalization using [Weak Type Theory](#)
- Translating Weak Type Theory into Type Theory with open terms.
- Examples and Problems



Phased Formalization using Weak Type Theory

What is Weak Type Theory

Formal language in the Automath tradition

- Identify structure: definitions, statements, etc.
- Rich in math primitives

$$\lambda x:A.M; \sum_{x:A} f(x); \iota x:A.\phi(x); \{x:A|\phi(x)\} \dots$$

- No built-in logic
- Enforces only linguistic correctness via weak types:

adjective, noun, term, set, statement

Examples

adjective

$Adj_{x:\text{nat}}.(0 < x) \rightarrow \text{"positive"}$

$Adj_{x:\text{nat}}.(\exists n:\text{nat}.2n = x) \rightarrow \text{"even"}$

statement

$\forall k:\text{Positive nat}.((2^k - 1) \text{ is prime}) \Rightarrow (2^{k-1}(2^k - 1) \text{ is perfect})$

term $\sqrt{b^2 - 4ac}, \lambda x:\text{nat}.x + x$

noun

$Noun_{x:\text{nat}} \text{prime}(n) \rightarrow \text{"a prime number"}$

$\text{positive even nat} \rightarrow \text{"a positive even number"}$

Weak Type Theory

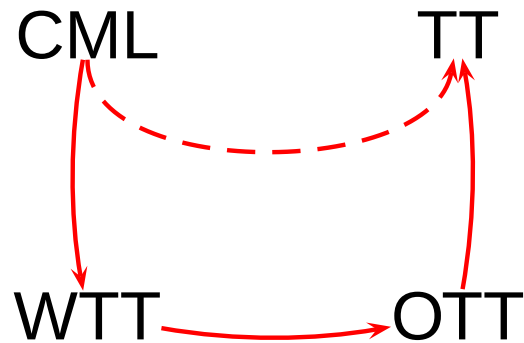
The Language

term	$\mathcal{T}$	$::=$	$x \mid c(\vec{\mathcal{P}}) \mid \lambda_{\mathcal{Z}} \mathcal{T} \mid \dots$
adjective	$\mathcal{A}$	$::=$	$c(\vec{\mathcal{P}}) \mid Adj_{\mathcal{Z}}(\mathcal{S}_p) \mid \dots$
noun	$\mathcal{N}$	$::=$	$c(\vec{\mathcal{P}}) \mid Noun_{\mathcal{Z}}(\mathcal{S}_p) \mid Abst_{\mathcal{Z}}(\mathcal{T}/\mathcal{N}/\mathcal{S}_s) \mid \mathcal{AN} \mid \mathcal{S}_s \downarrow \mid \dots$
set	$\mathcal{S}_s$	$::=$	$x \mid c(\vec{\mathcal{P}}) \mid Set_{\mathcal{Z}}(\mathcal{S}_p) \mid \mathcal{N} \uparrow \mid \dots$
statement	$\mathcal{S}_p$	$::=$	$x \mid c(\vec{\mathcal{P}}) \mid \mathcal{S}_p \rightarrow \mathcal{S}_p \mid \forall_{\mathcal{Z}}(\mathcal{S}_p) \mid \exists_{\mathcal{Z}}(\mathcal{S}_p) \mid \mathcal{T} \text{ is } \mathcal{A} \mid \mathcal{T} \text{ is } \mathcal{N} \mid \dots$
binder	$\mathcal{B}$	$::=$	$B_{\mathcal{Z}}.(\mathcal{T}/\mathcal{A}/\mathcal{N}/\mathcal{S}_s/\mathcal{S}_p)$
argument	$\mathcal{P}$	$::=$	$\mathcal{S}_s \mid \mathcal{S}_p \mid \mathcal{T}$
declaration	$\mathcal{Z}$	$::=$	$x:\mathbf{SET} \mid x:\mathbf{Prop} \mid x:\mathcal{N} \mid x:\mathcal{S}_s$
context	Γ	$::=$	$\emptyset \mid \Gamma, \mathcal{Z} \mid \Gamma, \mathcal{S}_p$
definition	$\mathcal{D}$	$::=$	$c(\vec{x}) := (\mathcal{T}/\mathcal{S}_p/\mathcal{S}_s/\mathcal{A}/\mathcal{N})$
line	l	$::=$	$\Gamma \triangleright \mathcal{D} \mid \Gamma \triangleright \mathcal{S}_p$
book	$\mathcal{B}$	$::=$	$\emptyset \mid \mathcal{B} \circ l$

The Formalization Path

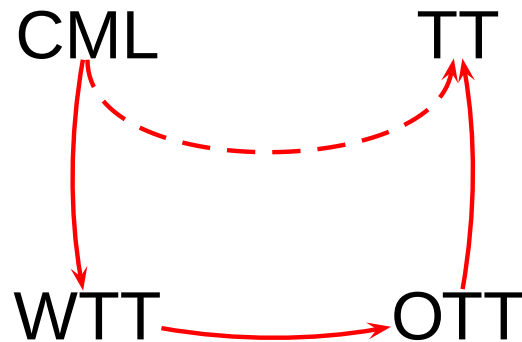


The Formalization Path



TT: Type Theory
OTT: TT with Open terms

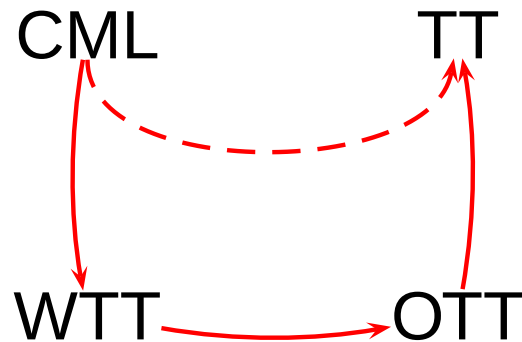
The Formalization Path



TT: Type Theory
OTT: TT with Open terms

- Write down the informal text into the formal language, considering only very weak (linguistic) correctness criteria

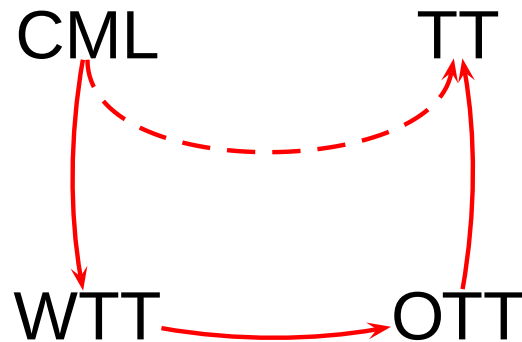
The Formalization Path



TT: Type Theory
OTT: TT with Open terms

- Write down the informal text into the formal language, considering only very weak (linguistic) correctness criteria
- Give "semantics" of the formal text. Here we make most of the "formalization" choices.

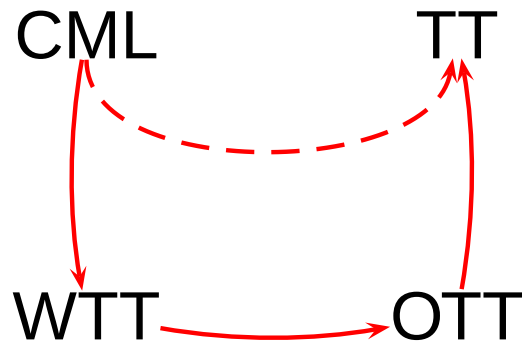
The Formalization Path



TT: Type Theory
OTT: TT with Open terms

- Write down the informal text into the formal language, considering only very weak (linguistic) correctness criteria
- Give "semantics" of the formal text. Here we make most of the "formalization" choices.
- Fill in implicit/missing details

Benefits



Reliability

Added value to the user

- Remain close to the original
- Getting faster "to the point"
- Opportunities for automation/checks/feedback
- Complements existing technologies
- Library of "models" of WTT



Translating WTT into Type Theory with Open Terms

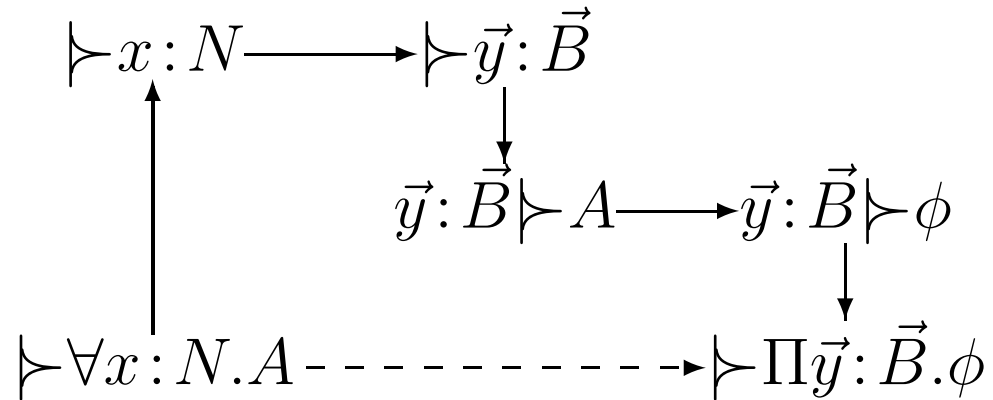
Translating Books and Lines

Books, lines:

$$\mathcal{B} := \emptyset \mid \mathcal{B}; \Delta \triangleright c(\vec{x}) := M \mid \mathcal{B}; \Delta \triangleright \psi$$

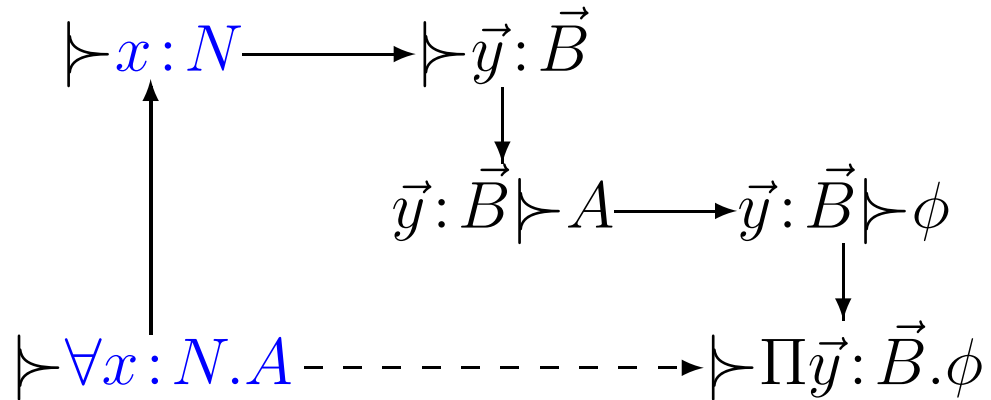
Books are mapped into TT contexts containing constants, meta-variables and definitions

Rule Diagrams



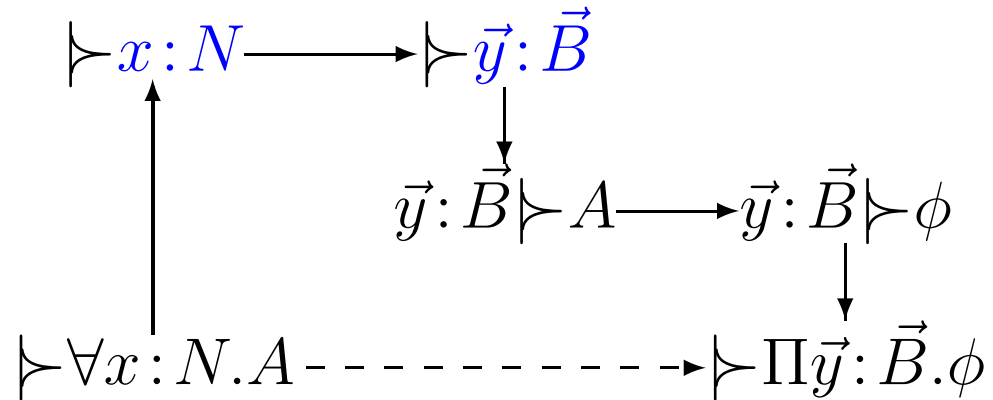
$$\frac{
 \begin{array}{l}
 \Gamma_1 \vdash x:N \longrightarrow \Gamma_2 \vdash \vec{y}:\vec{B} \\
 \Gamma_2, \vec{y}:\vec{B} \vdash A \longrightarrow \Gamma_3, \vec{y}:\vec{B} \vdash \phi
 \end{array}
 }{
 \Gamma_1 \vdash \forall x:N.A \longrightarrow \Gamma_3 \vdash \Pi \vec{y}:\vec{B}.\phi
 }$$

Rule Diagrams



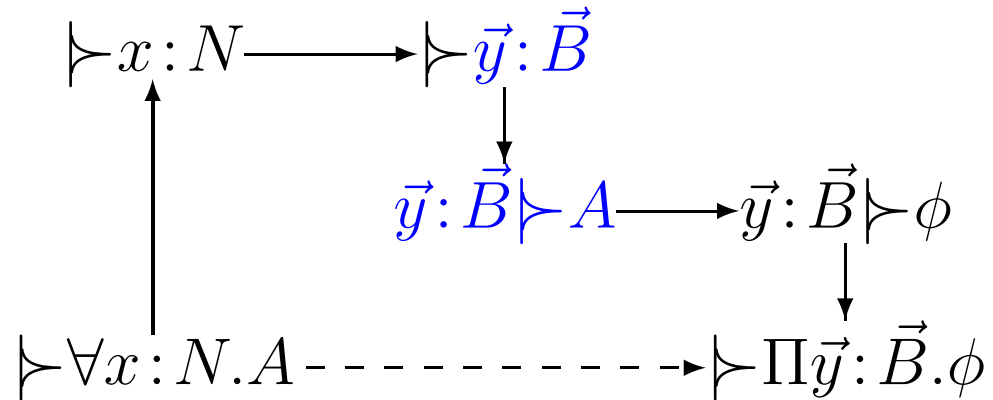
$$\frac{
 \begin{array}{l}
 \Gamma_1 \vdash x:N \longrightarrow \Gamma_2 \vdash \vec{y}:\vec{B} \\
 \Gamma_2, \vec{y}:\vec{B} \vdash A \longrightarrow \Gamma_3, \vec{y}:\vec{B} \vdash \phi
 \end{array}
 }{
 \Gamma_1 \vdash \forall x:N.A \longrightarrow \Gamma_3 \vdash \Pi \vec{y}:\vec{B}.\phi
 }$$

Rule Diagrams



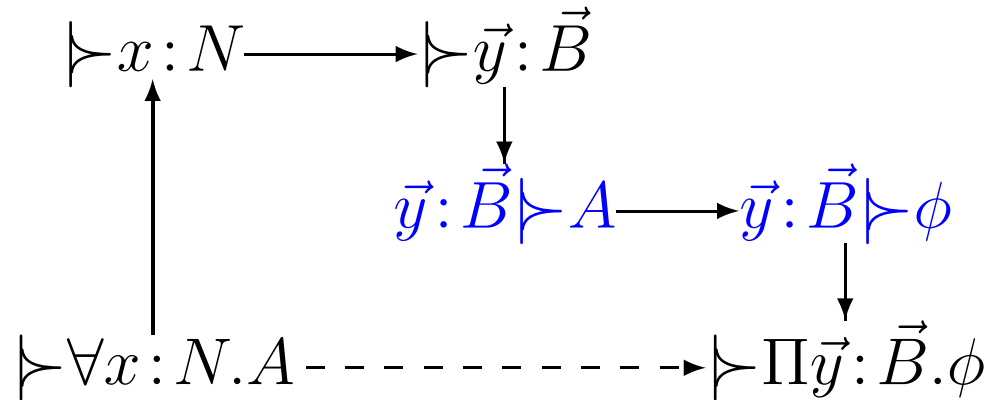
$$\frac{
 \begin{array}{l}
 \Gamma_1 \vdash x:N \longrightarrow \Gamma_2 \vdash \vec{y}:\vec{B} \\
 \Gamma_2, \vec{y}:\vec{B} \vdash A \longrightarrow \Gamma_3, \vec{y}:\vec{B} \vdash \phi
 \end{array}
 }{
 \Gamma_1 \vdash \forall x:N.A \longrightarrow \Gamma_3 \vdash \Pi \vec{y}:\vec{B}.\phi
 }$$

Rule Diagrams



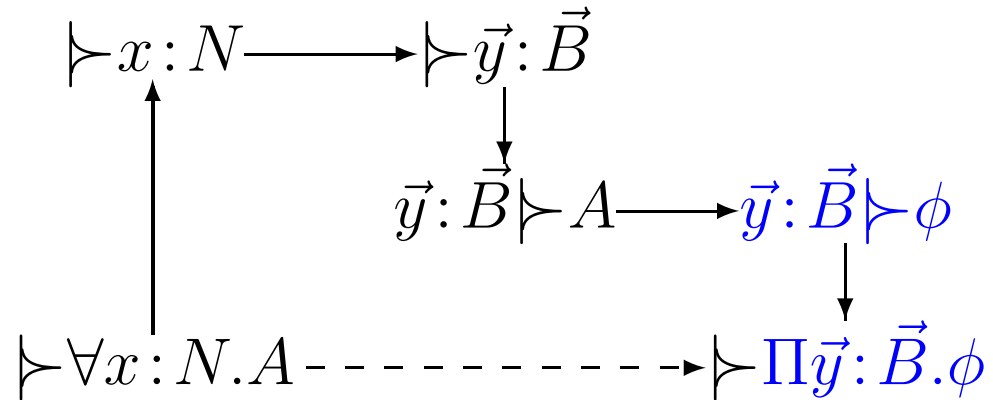
$$\begin{array}{c}
 \Gamma_1 \vdash x:N \longrightarrow \Gamma_2 \vdash \vec{y}:\vec{B} \\
 \Gamma_2, \vec{y}:\vec{B} \vdash A \longrightarrow \Gamma_3, \vec{y}:\vec{B} \vdash \phi \\
 \hline
 \Gamma_1 \vdash \forall x:N.A \longrightarrow \Gamma_3 \vdash \Pi \vec{y}:\vec{B}.\phi
 \end{array}$$

Rule Diagrams



$$\frac{
 \begin{array}{l}
 \Gamma_1 \vdash x:N \longrightarrow \Gamma_2 \vdash \vec{y}: \vec{B} \\
 \textcolor{red}{\Gamma}_2, \vec{y}: \vec{B} \vdash A \longrightarrow \textcolor{red}{\Gamma}_3, \vec{y}: \vec{B} \vdash \phi
 \end{array}
 }{
 \Gamma_1 \vdash \forall x:N. A \longrightarrow \Gamma_3 \vdash \Pi \vec{y}: \vec{B}. \phi
 }$$

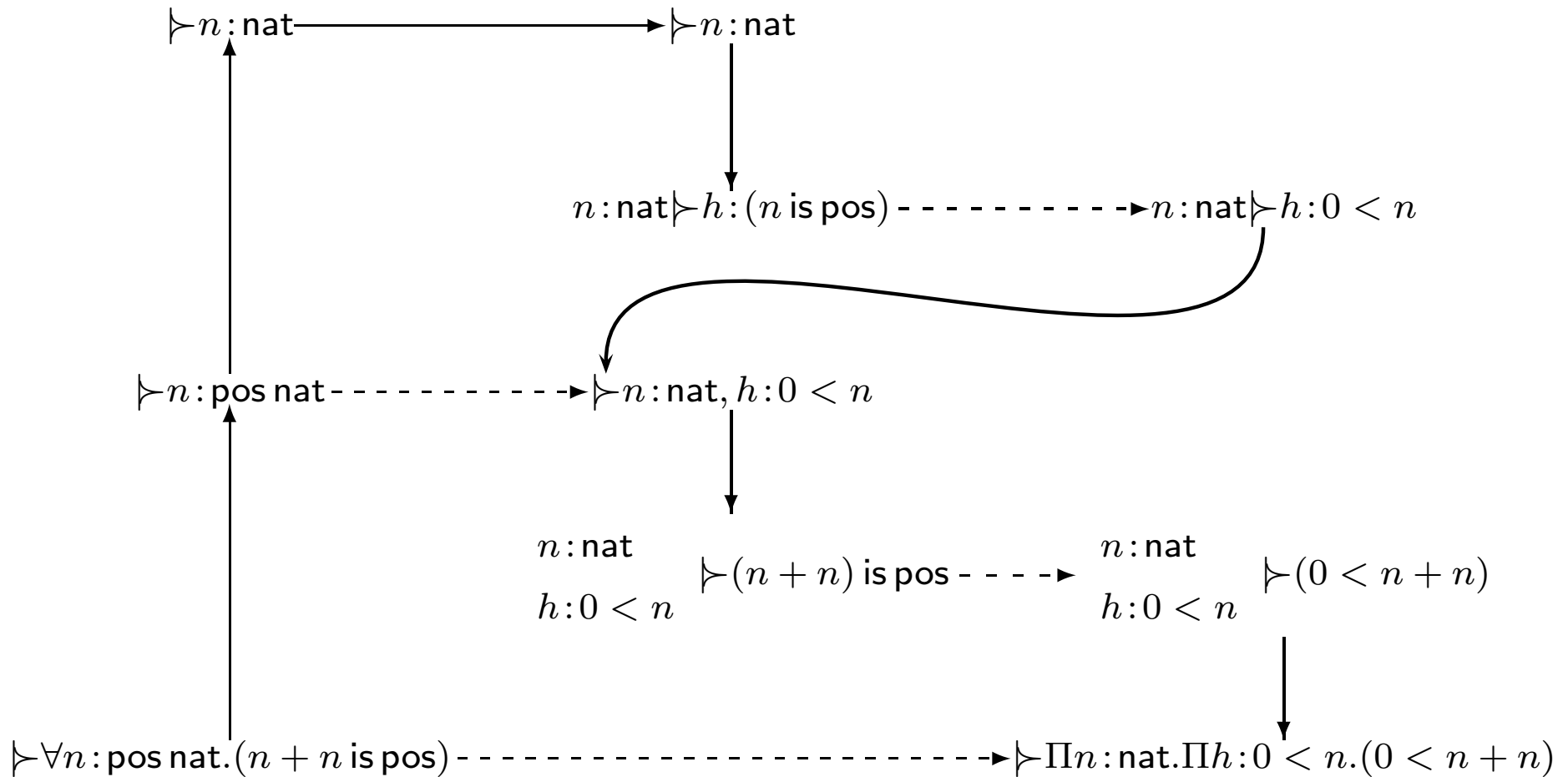
Rule Diagrams



$$\frac{
 \begin{array}{l}
 \Gamma_1 \vdash x:N \longrightarrow \Gamma_2 \vdash \vec{y}:\vec{B} \\
 \Gamma_2, \vec{y}:\vec{B} \vdash A \longrightarrow \textcolor{red}{\Gamma}_3, \vec{y}:\vec{B} \vdash \phi
 \end{array}
 }{
 \Gamma_1 \vdash \forall x:N.A \longrightarrow \textcolor{red}{\Gamma}_3 \vdash \Pi \vec{y}:\vec{B}.\phi
 }$$

Example

$\text{pos} := \text{Adj } n : \text{nat}. 0 < n$



"Demo": Export to Coq

```
|=> positive := Adj[n:nat] (lt [zero, n]).
```

```
x:positive[] nat |=> double := plus[x,x].
```

```
|:> Forall[n:positive[]~nat](double[n] is positive[]).
```

```
Definition positive := (fun n:nat=>(lt 0 n)).
```

```
Definition double:=
```

```
  fun x:nat=> fun H:(positive x)=> (plus x x).
```

```
Lemma PRF :
```

```
  forall n:nat,
```

```
  forall H:(positive n),
```

```
  (positive n).
```

```
Lemma STMLINE :
```

```
  (forall n:nat,
```

```
  (forall H:(positive n),
```

```
  (positive (double n (PRF n H)))).
```

"Demo": Export to Coq

```
|=> positive := Adj[n:nat] (lt [zero, n]).
```

```
x:positive[] nat |=> double := plus[x,x].
```

```
|:> Forall[n:positive[]~nat](double[n] is positive[]).
```

```
Definition positive := (fun n:nat=>(lt 0 n)).
```

```
Definition double:=
```

```
  fun x:nat=> fun H:(positive x)=> (plus x x).
```

```
Lemma PRF :
```

```
  forall n:nat,  
  forall H:(positive n),  
  (positive n).
```

```
Lemma STMLINE :
```

```
  (forall n:nat,  
  (forall H:(positive n),  
  (positive (double n (PRF n H))))
```

"Demo": Export to Coq

```
|=> positive := Adj[n:nat] (lt [zero, n]).
```

```
x:positive[] nat |=> double := plus[x,x].
```

```
|:> Forall[n:positive[]~nat](double[n] is positive[]).
```

```
Definition positive := (fun n:nat=>(lt 0 n)).
```

```
Definition double:=
```

```
  fun x:nat=> fun H:(positive x)=> (plus x x).
```

Lemma PRF :

forall n:nat,

forall H:(positive n),

(positive n).

Lemma STMLINE :

(forall n:nat,

(forall H:(positive n),

(positive (double n (PRF n H)))

"Demo": Export to Coq

```
x:nat | => quad := plus[  
  double[x], app[Lambda[y:nat](double[x]),x]].
```

```
Lemma PRF : forall x:nat,(positive x).  
Admitted.
```

```
Lemma PRF1 : forall x:nat, forall y:nat,(positive x).  
Admitted.
```

```
Definition quad := fun x:nat=> (plus  
  (double x (PRF x))  
  ((fun y:nat=>(double x (PRF1 x y))) x)).
```

Problems and Future Work

- How to define correctness criteria in general?
- One expects equal objects to be mapped to equal objects. The proof terms may break this - hence the need for unification of proof terms.
- How strong unification/matching do we need?
- Other models
- Debugging and testing the implementation with larger case study
- Modelling Meta-variables by Lemma's and Admitted is not the best solution.



Questions?