DEPARTMENT OF COMPUTER SCIENCE (DIKU)
UNIVERSITY OF COPENHAGEN

Proceedings of the 4th DIKU-IST Joint Workshop on Foundations of Software

Robert Glück, Fritz Henglein, Zhenjiang Hu, Masato Takeichi (Eds.)

February 2011

Technical Report no. 2011/01

ISSN: 0107-8283

Preface

These proceedings contain the contributions presented at the *4th DIKU-IST Joint Workshop on Foundations of Software* held at Tokyo, Japan, January 10-14, 2011. The workshop featured talks and discussions on program optimization and parallel programming, formal semantics and algorithms, reversible and bidirectional computing, and demonstrations of software prototypes.

After the success of the first three joint workshops, which took place at Dragør, Copenhagen (2005), Shonan Village, Japan (2006) and Roskilde, Denmark (2007) with proceedings as DIKU Technical Reports 05/07, 06/07 and 07/07, the 4th DIKU-IST workshop took again place in Japan. It aimed to provide a forum for presenting the latest research and promoting the research collaboration between the *Department of Computer Science (DIKU), University of Copenhagen*, and the *Graduate School of Information Science and Technology (IST), University of Tokyo*. In 2004, IST and the Faculty of Science at the University of Copenhagen entered an *Academic and Student Exchange Agreement* during the State Visit of Queen Margrethe II to Japan. Today, both universities are partners in the *International Alliance of Research Universities* (IARU).

Computer science provides one of the keys to the technologies of the 21st century. Its applications have found their way into all areas of daily life, often unnoticed by their users, and software has become a decisive factor for commercial success in many areas of modern business and society. This series of workshops between DIKU and IST is devoted to the scientific foundations of software. Theory and practice of programming languages are very important and visible research fields at both research institutions in Copenhagen and Tokyo. The objective of these joint workshops is to give researchers and graduate students at both institutions the opportunity to exchange the latest research ideas and to jointly engage in outstanding international research.

The workshop had about 50 participants from Japan and Denmark. The organizers would like to thank all speakers, participants, and the local organizers for making this meeting both successful and enjoyable. Special thanks to Masami Hagiya, Dean of IST, for supporting the workshop, Yasuo Kuniyoshi, Professor of IST, for the visit to the robotics laboratory, and to Kaori Sato of the Office of International Relations for organizing the student program.

Copenhagen and Tokyo, February 2011

Robert Glück
Fritz Henglein
Zhenjiang Hu
Masato Takeichi

Organization

The *DIKU-IST Joint Workshop on Foundations of Software* was organized by the Graduate School of Information Science and Technology, University of Tokyo, together with the Department of Computer Science, University of Copenhagen.

Meeting

Program committee:	Robert Glück Fritz Henglein Zhenjiang Hu Masato Takeichi
Local arrangements:	Kazuyuki Asada Keisuke Nakano
Proceedings:	Tom Hvitved

Sponsoring Institutions

This workshop was supported by Grant-in-Aid for Scientific Research (A) 192000-02, a joint research project at National Institute of Informatics, and The International Alliance of Research Universities (IARU) fund of the University of Copenhagen. Thanks also to the Danish National Advanced Technology Foundation (Højteknologifonden), the Danish Strategic Research Council (Det Strategiske Forskningsråd) and the Danish Free Research Council (Det Frie Forskningsråd).

Table of Contents

Session 5

Session 6

Session 7

Session 8

Session 9

Session 10

Session 11

Tokyo, Japan (January 2011)

The Fourth DIKU-IST Joint Workshop on Foundations of Software

Asakusa View Hotel and Univ. Tokyo

January 10-14, 2011

Academic Exchange Agreement Ceremony
between Faculty of Science, University of Copenhagen
and Graduate School of IST, University of Tokyo

November 15, 2004

Works after the 1st DIKU-IST Workshop at Copenhagen

September 24, 2005

Participants of the 2nd DIKU-IST Workshop at Shonan-Village, Japan

April 21-22, 2006

3rd DIKU-IST Joint Workshop on Foundations of Software

Fri & Sat, 5-6 October 2007, Comwell Hotel, Roskilde, Denmark

John Andersen
(Director of International Affairs, University of Copenhagen)

Masaki Okada
(Ambassador of Japan In Denmark)

40+ participants
(11 from IST, University of Tokyo)

The Fourth DIKU-IST Joint Workshop on Foundations of Software

10 – 14 January, 2011

Japan

- Important Dates
- Objective
- Theme
- Tentative Programme
- Venue / Accommodation
- Access
- Organization / Program Committee
- Local Organizers
- Sponsors
- Contact Information

Important Dates

- January 10-12: Workshop at Asakusa View Hotel
- January 13: Workshop at The University of Tokyo
- January 14: Free Discussion

Objective

After the success of the first, second, and third DIKU-IST workshops, the fourth DIKU-IST workshop aims to provide a forum for exchanging research ideas among researchers on programming languages and foundations of software, and promoting research collaboration between Department of Computer Science (DIKU) at University of Copenhagen and Graduate School of Information Science and Technology (IST) at University of Tokyo.

PROGRAMMING IN BIOMOLECULAR COMPUTATION

Lars Hartmann

Neil D. Jones

Jakob Grue Simonsen

+

Visualization by Søren Bjerregaard Vrist

(All now or recently at the University of Copenhagen)

DIKU-IST (January 2011)

Sources:

- June 2010 conference CS2BIO Computer Science to Biology
- Journal Scientific Annals of Computer Science (to appear)
- Festschrift for Carolyn Talcott (to appear)

UNIVERSALITY AND PROGRAMMING IN A BIOCHEMICAL SETTING

Turing completeness results for biomolecular computation:

- Cardelli, Chapman, Danos, Reif, Shapiro, Wolfram,. . .
- Net effect: any computable function can be computed, in some sense, by various biological mechanisms.
- Not completely compelling from a programming perspective.
- Our aim: a computation model where
  - "program" is clearly visible and natural, and
  - Turing completeness is not artificial or accidental, but a natural part of biomolecular computation

CONNECTIONS EXIST BETWEEN BIOLOGY AND COMPUTATION, but . . . WHERE ARE THE PROGRAMS?

Our proposal: a model of computation that is

- biologically plausible: semantics by chemical-like reaction rules;
- programmable (a bit like low-level computer machine code);
- uniform: new "hardware" not needed to solve new problems;
- stored-program: programs = data;
  programs are executable and compilable and interpretable
- universal: all computable functions can be computed
- Turing complete in a strong sense: $\exists$ a universal algorithm
  (able to execute any program, asymptotically efficient)

BUT WHERE ARE THE PROGRAMS?

In existing models of biomolecular computation

it's hard to see anything like a program that realises or directs a computational process.

- In cellular automata, "program" is expressed only in the initial cell configuration, or in the global transition function
- Many examples: given a problem, authors cleverly devise a biomolecular system that can solve this particular problem
- The algorithm being implemented is hidden in the details of the system's construction, hard to see.

Our purpose is to fill this gap,

- to establish a biologically feasible framework in which
- programs are first-class citizens.

— 3 —

OTHER COMPUTATIONAL FRAMEWORKS

Circuits, BDDs, finite automata: Nonuniform, Turing incomplete

Turing machine:

- Pro Visible program; complete; universal machine exists
- Con Asymptotically slow: universal machine takes time $O(n^2)$ to simulate a program running in time $O(n)$

Other program-based models: Post, Minsky, LISP, RAM, RASP...
Complex, biologically implausible

Cellular automata: von Neumann, LIFE, Wolfram,...

- Pro Can simulate a Turing machine
- Con Complex, biologically implausible (synchronisation!)
  There is no natural universal cellular automaton.
  It's very hard to see "the program".

— 4 —

"DIRECT" PROGRAM EXECUTION

Write $[\![\texttt{program}]\!]$ for the meaning or net effect of running `program`:

$$[\![\texttt{program}]\!](\texttt{data}_{in}) = \texttt{data}_{out}$$

- `program` is an active agent.
- It is activated (run) by applying the semantic function $[\![\_]\!]$.
- Some mechanism is needed to execute `program`, i.e., to apply $[\![]\!]$ to `program` and $\texttt{data}_{in}$:

  hardware ("wetware"?).

— 5 —

THE BIOLOGICAL WORLD IS NOT HARDWARE!

We must re-examine programming language assumptions. Computers have programmer-friendly conveniences, e.g.,

- A large address space of randomly accessible data
- Pointers to data, perhaps at a great "distance" from the current program or data
- address arithmetic, index registers,...
- Unbounded fan-in: many pointers to the same data item...

None of these is biologically plausible!

Workarounds are needed if we want to do biological programming.

— 6 —

FOR BIOLOGICAL PLAUSIBILITY

- There is no action at a distance: all effects achieved via chains of local interactions. Biological analog: signaling.
- There are no pointers to data (addresses, links, list pointers): To be acted on, a data value must be physically adjacent to an actuator. Biological analog: chemical bond between program and data.
- No nonlocal control transfer, e.g., unbounded GOTOs or remote procedure calls. Biological analog: a bond from one part of a program to another.
- A "yes": $\exists$ available resources to tap, i.e., energy to change the program control point, or to add data bonds.
  Biological analogs: ATP, oxygen, Brownian movement.

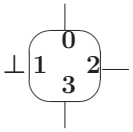
— 7 —

THE BLOB MODEL

Simplified view of a molecule and chemical interactions (Cardelli, Danos, Lanève,...).

Blobs are in a biological "soup" and are connected by symmetrical bonds linking their bond sites.

Picture of a blob: (Bond sites $0, 2$ and 3 are bound, and 1 is unbound)



A blob has 4 bond sites and 8 cargo bits (boolean values).

Here: Bond sites $0, 2$ and 3 are bound, and 1 is unbound.
Cargo bits omitted for brevity.

— 8 —

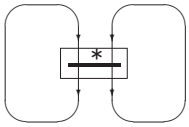
KEEPING THE FOCUS

How to structure a biologically feasible model of computation?

- Idea: keep current program counter and data cursor always close to a focus point where all actions occur.
- How? Continually shift both program and data, to keep the active bits near the focus.

Running program p: computing ⟦p⟧(d)

Program p Data d



□ = Focus point for control and data
(connects the APB and the ADB)

* = program-to-data bond: "the bug"

A MOVIE IS WORTH DURATION×FRAMERATE×1000 WORDS

(Circle.avi)

ABOUT INSTRUCTIONS:

Instruction form:

```
opcode parameters (bond0, bond1, bond2, bond3)
```

Why exactly 4 bonds?

- Predecessor (1 bond); true and false successors (2 bonds);
- plus one bond to link the APB to the ADB.

It's almost a von Neumann machine code, but...

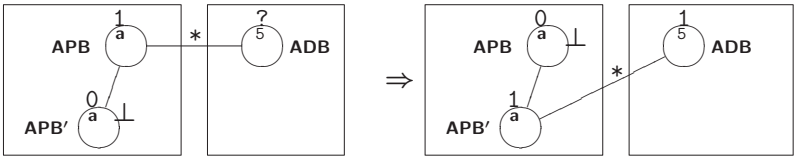
- A bond is a two-way link between two adjacent blobs.
- A bond is not an address.
- There is no address space as in conventional computer (and hence: no address decoding hardware).
- Also: no registers (use the cargo bits instead).

INSTRUCTIONS HAVE 8 BITS

Instruction	Description	Informal semantics (write :=: for a two-way interchange)
SCG v c	Set CarGo bit	ADB.c := v; APB := APB.2
JCG c	Jump CarGo bit	if ADB.c = 0 then APB := APB.3 else APB := APB.2
JB b	Jump Bond	if ADB.b = ⊥ then APB := APB.3 else APB := APB.2
CHD b	CHange Data	ADB := ADB.b; APB := APB.2
INS b1 b2	INSert new bond	ADB-new.b2 :=: ADB.b1; ADB-new.b1 :=: ADB.b1.bs;
	—	APB := APB.2
SBS b1 b2	SWap Bond Sites	ADB.b1 :=: ADB.b2; APB := APB.2
SWL b1 b2	SWap Links	ADB.b1 :=: ADB.b2.b1; APB := APB.2
SWP3 b1 b2	Swap bs3 on linked	ADB.b1.3 :=: ADB.b2.3; APB := APB.2
FIN	Fan IN	APB := APB.2 (two predecessors: bond sites 1 and 3)
EXT	EXiT program	

SCG,...,EXT: **Operation codes**
b, b1, b2: **Bond site numbers**
c: **Cargo site number**
v: **A one-bit value**

EXAMPLE: EFFECT OF SCG 1 5 (SET CARGO BIT 5 TO 1)



- "The bug" $\overset{*}{—}$ has moved:
  - before execution, it connected APB with ADB.
  - After: it connects successor APB′ with ADB.
- Also: activation bits 0, 1 have been swapped.

Instruction syntax: the 8-bit string 11001101 is grouped as

$$\overbrace{1}^{a}\ \overbrace{100}^{SCG}\ \overbrace{1}^{v}\ \overbrace{101}^{c}$$

SEMANTICS OF SCG 1 5 BY "SOMETHING LIKE" A CHEMICAL REACTION RULE

Instruction form: $\overbrace{1}^{a}\ \overbrace{100}^{SCG}\ \overbrace{1}^{v}\ \overbrace{101}^{c}$

$$\overbrace{B[1\ 100\ 1\ 101](*\text{---})}^{APB},\ \overbrace{B[0\text{------}](\perp\text{---})}^{APB'},\ \overbrace{B[0\text{----}x\text{--}](*\text{---})}^{ADB}$$

$$\Rightarrow$$

$$\underbrace{B[0\ 100\ 1\ 101](\perp\text{---})}_{APB},\ \underbrace{B[1\text{------}](*\text{---})}_{APB'},\ \underbrace{B[0\text{----}1\text{--}](*\text{---})}_{ADB}$$

(- = unchanged bond or cargo bit)

Similar style to: Danos and Laneve, Formal Molecular Biology.

TURING COMPLETENESS BY INTERPRETATION

Turing completeness is usually shown by simulation, e.,g.,

- for any 2CM program you build a biomolecular system such that ...

But: the biomolecular system is usually built by hand. The effect: hand computation of the $\exists$ quantifier in

$$\forall p \exists q ([\![p]\!]^L = [\![q]\!]^M)$$

In contrast, Turing's original "Universal machine" (UM) works by interpretation, where $\exists$ is realised by machine.

- The UM can execute any TM program, if coded on the UM's tape along with its input data.
- Our research follows Turing's line, in a biological context: It does simulation by general interpretation, and not by one-problem-at-a-time constructions.

PROGRAM EXECUTION BY INTERPRETATION

- $$[\![\texttt{interpreter}]\!](\texttt{program}, \texttt{data}_{in}) = \texttt{data}_{out}$$
- Now `program` is a passive data object: both `program` and $\texttt{data}_{in}$ are data for the interpreter.
- `program` is now executed by running the interpreter program.

  (Of course, some mechanism will be needed to run the interpreter, e.g., hard-, soft- or wetware.)
- Self-interpretation is possible, and useful in practice.
- The Universal Turing machine is a self-interpreter.

A "BLOB UNIVERSAL MACHINE"

We have programmed a self-interpreter for the blob formalism – analogous to Turing's original universal machine.

This gives: Turing-completeness in a new biological framework.

BIRDS-EYE VIEW OF THE SELF-INTERPRETER

(Not shown: Each 'finger' along the periphery has a connection to the main control in the center)

— 18 —

CONTRIBUTIONS OF THIS WORK

- Programmable bio-level computation where programs = data.
- Blob semantics by abstract biochemical reaction rules.
- All computable functions are blob-computable:
  - Can do with one fixed, set of reaction rules (defining a fixed instruction set, i.e., a "machine language")
  - Don't need new rule sets (i.e., biochemical architectures) to solve new problems; it's enough to write new programs.
- (Uniform) Turing-completeness
- Promise of tighter analogy between universality and self-reproduction.
- Interpreters and compilers make sense at biological level, may give useful operational and utilitarian tools.

— 19 —

WHERE TO NOW?

Some points to address:

- Find a true, biological (not just "feasible") implementation of the fixed set of reduction rules in vitro.
- Programs are currently similar to classical machine code; this requires programmer skill. Solution: Devise an intermediate-level blob programming language.
- Still to analyse: The time or energy cost of performing a single program step (may depend on program/data). An appropriate and realistic cost model should be found.
- Bonus: This could initiate a study of computational complexity in the blob world.

— 20 —

Reinvestigation of Symmetric Lambda Calculus

Yayoi Ueda Kenichi Asai
Ochanomizu University

Abstract

This paper presents a symmetric lambda calculus (SLC) in which both the duality between call-by-value and call-by-name and the duality between expressions and continuations hold at the same time. The idea of SLC was originally introduced by Filinski in 1989, but has not been investigated seriously since then. This paper first reformulates SLC using small-step reduction semantics in a completely symmetric way. We then show call-by-value and call-by-name evaluation strategies for SLC and prove that both enjoy various formal properties, such as subject reduction. Finally, we prove that the small-step semantics shown here coincides with Filinski's original definition of SLC using the functional correspondence approach: the two semantics are related via defunctionalization.

Keywords: symmetric lambda calculus, control operators, functional correspondence, defunctionalization

1 Introduction

Background and related work In 1990, Griffin [7] showed the correspondence between the law of double-negation elimination in classical logic and the control operator $\mathcal{C}$ introduced by Felleisen and Hieb [5]. Since then, many researchers explored the foundational theory for continuations and control operators. Parigot [8] introduced $\lambda\mu$-calculus and showed that it corresponds to the classical natural deduction. Curien and Herbelin [3] introduced $\overline{\lambda}\mu\tilde{\mu}$-calculus that exhibits symmetry between expressions and continuations as well as between call-by-value (CBV) and call-by-name (CBN) based on the classical sequent calculus. Selinger [12] gave a categorical semantics to $\lambda\mu$-calculus and showed categorical duality between CBV and CBN. These work led to Wadler's Dual Calculus [14] that shows CBV/CBN symmetry in a particularly clear way in a simple syntax and operational semantics. Tzevelekos [13] studied the Dual Calculus in detail and showed various syntactic properties.

Underlying these work is an intuition that expressions and continuations are dual. Surprisingly, this observation was made as early as in 1989 by Filinski [6]. He presented a symmetric lambda calculus (SLC) in which expressions and continuations are treated in a complete symmetry. Just like we abstract over expressions to receive an argument expression, we can abstract over continuations in SLC to capture the current continuation. This system has many implications. The CBV/CBN duality naturally arises from the priority of execution: to execute expressions first leads to CBV and continuations first CBN. The duality between types of expressions and continuations even suggests a relationship to classical logic via de Morgan's laws.

However, although most of the previous work mentioned Filinski's work, none of them actually investigated the relationship to SLC seriously. This is partly because Filinski's SLC lacks small-step reduction semantics. It makes it harder to compare Filinski's SLC to other calculi such as Dual Calculus, because they are often defined as small-step reduction semantics. This is unfortunate because SLC not only contains the standard lambda-calculus naturally but also presents control operators in a particularly simple and intuitive way.

The name symmetric lambda calculus was independently used by Barbanera and Berardi [2]. However, their work is quite different from Filinski's SLC. Their calculus includes a notion of symmetric application where either component of an application can be a function or an argument. They proved the strong normalization property of this calculus.

This work This paper investigates small-step reduction semantics for Filinski's SLC. It handles both non-deterministic, CBV, and CBN evaluation strategies. The non-deterministic semantics exhibits com-

$$
\begin{array}{lrcl}
\text{expression} & e & ::= & n \mid x \mid () \mid (e,\, e) \mid e \uparrow f \mid \lceil f \rceil \\
\text{function} & f & ::= & g \mid p \Rightarrow e \mid c \Leftarrow q \mid \overline{e} \mid \underline{c} \\
\text{continuation} & c & ::= & \bullet \mid y \mid \{\} \mid \{c,\, c\} \mid f \downarrow c \mid \lfloor f \rfloor \\
\text{pattern for } e & p & ::= & x \mid () \mid \lceil g \rceil \mid (p,\, p) \\
\text{pattern for } c & q & ::= & y \mid \{\} \mid \lfloor g \rfloor \mid \{q,\, q\}
\end{array}
$$

Figure 1: Syntax of SLC

plete symmetry, whereas CBV and CBN strategies are dual to each other. A type system for SLC is shown which satisfies various properties such as subject reduction. The paper then shows that the small-step semantics presented here and the original denotational semantics of SLC given by Filinski are in functional correspondence with each other. This work is also a non-trivial application of Danvy's functional correspondence approach [1, 4].

Overview In the next Section, we introduce SLC, its syntax, types, typing rules, non-deterministic reduction rules, and some properties. In Section 3, we show the CBV strategy and its properties such as uniqueness of evaluation. In Section 4, we show the CBN strategy and its properties. We defunctionalize Filinski's original CBV denotational semantics, and show the correspondence between the two SLC's in Section 5. Finally, we also defunctionalize Filinski's CBN semantics and show their correspondence in Section 6.

2 Symmetric Lambda Calculus

This section introduces the symmetric lambda calculus (SLC). A standard way to represent reduction semantics is to use an evaluation context: $E[M] \rightsquigarrow E[M']$ if $M \rightsquigarrow M'$. In SLC, we represent it with a configuration $\langle\ e \mid c\ \rangle$, where e is an expression (corresponding to M) and c is a continuation (corresponding to $E[\,]$). In addition, SLC has three-place configuration $\langle\ e \mid f \mid c\ \rangle$ which represents that an expression e is passed to a function f in a context c. In SLC, an evaluation context and searching for a redex are both explicit in a configuration. SLC calculates an expression and a continuation in a completely dual manner.

2.1 Syntax

Figure 1 shows the syntax of SLC.[1] An expression e is either a natural number n, a variable x, a unit $()$, a pair of expressions $(e,\, e)$, an application $e \uparrow f$, or a function treated as an expression $\lceil f \rceil$. An application $e \uparrow f$ passes an expression e to a function f. Unlike the standard lambda-calculus, a function f is placed after the argument e. In SLC, an expression and a function are distinct syntactic objects. To treat a function as an expression (to pass a higher-order function to another function, for example), a function is tagged with $\lceil\ \rceil$.

A continuation is defined as a dual notion of an expression. A continuation c is either an initial continuation $\bullet$, a continuation variable y, a counit $\{\}$, a pair of continuations $\{c,\, c\}$, a continuation application $f \downarrow c$, which represents a continuation that applies f before passing a value to c, or a function treated as a continuation $\lfloor f \rfloor$. An initial continuation $\bullet$ represents a continuation that receives a natural number n. When the initial continuation receives a natural number n, the computation finishes with the result n. The counit $\{\}$ represents a continuation that never receives any value.

A function is either a function variable g, an expression abstraction $p \Rightarrow e$ which binds the current expression to p and replaces the expression with e, a continuation abstraction $c \Leftarrow q$ which binds the current continuation to q and replaces the continuation with c, $\overline{e}$ an expression that evaluates to a function $\lceil f \rceil$, or $\underline{c}$ a continuation that evaluates to a function $\lfloor f \rfloor$.

[1] This syntax slightly deviates from Filinski's original syntax. Filinski used $f \uparrow e$ and $q \Rightarrow c$ instead of our $e \uparrow f$ and $c \Leftarrow q$, respectively. We changed the syntax because the new syntax exhibits beautiful symmetry when written in a small-step semantics.

$$
\begin{array}{rcll}
T & ::= & +A & \text{(types of expressions } e\text{)} \\
 & | & A \supset B & \text{(types of functions } f\text{)} \\
 & | & \neg B & \text{(types of continuations } c\text{)} \\
A, B & ::= & \bot \mid \top \mid \mathsf{int} \mid A \wedge B \mid A \vee B \mid A \to B \mid A - B &
\end{array}
$$

Figure 2: Types of SLC

An expression abstraction $p \Rightarrow e$ has a pattern p as its formal argument. If the pattern is a variable x, $x \Rightarrow e$ corresponds to $\lambda x.e$ in the standard lambda calculus. A pattern p can also be a unit, a pair, or a function pattern $\lceil g \rceil$. Dually, q represents a pattern for a continuation abstraction $c \Leftarrow q$.

We assume that the sets of variable names for expressions, continuations, and functions do not overlap. We use a meta variable var to represent any of the three kinds of variables.

As an example, Filinski represented call/cc in SLC as follows [6]:

$$\mathsf{call/cc} = (\lceil g \rceil \Rightarrow \lceil y \Leftarrow \_ \rceil \uparrow g) \downarrow y \Leftarrow y$$

where _ represents a dummy variable not used anywhere else. It grabs the current continuation in y and replaces the current continuation with $(\lceil g \rceil \Rightarrow \lceil y \Leftarrow \_ \rceil \uparrow g) \downarrow y$. The installed function $(\lceil g \rceil \Rightarrow \lceil y \Leftarrow \_ \rceil \uparrow g)$ receives a function g and passes the representation of the current continuation $\lceil y \Leftarrow \_ \rceil$ to g. The passed continuation, when applied, will replace the then continuation with y, achieving jump to the continuation captured in y. (See Section 2.4 for an example reduction sequence for call/cc.) Similarly, Felleisen's $\mathcal{C}$ operator can be defined as follows:

$$\mathcal{C} = (\lceil g \rceil \Rightarrow \lceil y \Leftarrow \_ \rceil \uparrow g) \downarrow \bullet \Leftarrow y$$

2.2 Types

Figure 2 shows types of SLC. Since SLC consists of three syntactic objects, the types of SLC consist of three types: an expression type $+A$, a continuation type $\neg B$, and a function type $A \supset B$.

The type $+A$ represents a type of an expression. The prefix $+$ indicates that it is a type of an expression. The type $\neg B$ represents a type of a continuation that receives a value of type $+B$. In this paper we use $\neg$ for a type of continuations. The type $A \supset B$ represents a type of functions that receive an expression of type $+A$ and return an expression of type $+B$. At the same time, it represents a type of functions that receive a continuation of type $\neg B$ and return a continuation of type $\neg A$. A function has an implicative and contrapositive type together.

A neutral type A, B forms the inner structure of types. A neutral type is either true $\top$, false $\bot$, integer int, conjunction $A \wedge B$, disjunction $A \vee B$, implication $A \to B$, or minus $A - B$.[2] The type $+\top$ is a type of () and $\neg\bot$ is a type of {}.

2.3 Typing Rules

The typing rules of SLC are shown in Figure 3. The left column shows typing rules for expressions, while the right for continuations. Let Γ be a type environment. We write $\Gamma \vdash e : +A$ to mean that an expression e has a type $+A$ under Γ, and similarly for a continuation and for a function.

The type environment Γ is defined as a set of type bindings $\mathsf{var} : T$, where all the variables occurring in a type environment have to be distinct. Given a type environment Γ, we define Γ_p as a type environment where bindings for variables in the pattern p are removed (Figure 4).

The typing rules for expressions are mostly standard. To support a nested pattern for abstractions, the judgement $\Gamma \vdash_p p : +A$ for expression patterns (and $\Gamma \vdash_q q : \neg B$ for continuation patterns) is introduced, which is defined in the second part of Figure 3. The typing rules for continuations are obtained by taking the dual of the ones for expressions. In the rule $\overline{\texttt{TOr}}$, a pair of continuations is given a disjunctive continuation type. Logically, it is a de Morgan dual of $\neg A \wedge \neg B$. The rule $\overline{\texttt{TApp}}$ says that $f \downarrow c$ as a whole can be regarded as a continuation of type $\neg A$: it receives a value of type $+A$, turns it

[2]The intuitive logical meaning of minus operator $A - B$ is $\neg(\neg B \to \neg A)$. Filinski expresses this type as $[A \leftarrow B]$ [6].

$$\frac{}{\Gamma_g \cup \{g : A \supset B\} \vdash g : A \supset B}\ \texttt{TFVar}$$

$$\frac{}{\Gamma_x \cup \{x : +A\} \vdash x : +A}\ \texttt{TVar} \qquad \frac{}{\Gamma_y \cup \{y : \neg B\} \vdash y : \neg B}\ \overline{\texttt{TVar}}$$

$$\frac{}{\Gamma \vdash n : +\mathsf{int}}\ \texttt{TInt} \quad \frac{}{\Gamma \vdash () : +\top}\ \texttt{TUnit} \qquad \frac{}{\Gamma \vdash \{\} : \neg\bot}\ \overline{\texttt{TUnit}} \quad \frac{}{\Gamma \vdash \bullet : \neg\mathsf{int}}\ \overline{\texttt{TInt}}$$

$$\frac{\Gamma \vdash e_1 : +A \quad \Gamma \vdash e_2 : +B}{\Gamma \vdash (e_1, e_2) : +(A \wedge B)}\ \texttt{TAnd} \qquad \frac{\Gamma \vdash c_1 : \neg A \quad \Gamma \vdash c_2 : \neg B}{\Gamma \vdash \{c_1, c_2\} : \neg(A \vee B)}\ \overline{\texttt{TOr}}$$

$$\frac{\Gamma \vdash_p p : +A \quad \Gamma \vdash e : +B}{\Gamma_p \vdash p \Rightarrow e : A \supset B}\ \texttt{TFun} \qquad \frac{\Gamma \vdash c : \neg A \quad \Gamma \vdash_q q : \neg B}{\Gamma_q \vdash c \Leftarrow q : A \supset B}\ \overline{\texttt{TFun}}$$

$$\frac{\Gamma \vdash e : +A \quad \Gamma \vdash f : A \supset B}{\Gamma \vdash e \uparrow f : +B}\ \texttt{TApp} \qquad \frac{\Gamma \vdash f : A \supset B \quad \Gamma \vdash c : \neg B}{\Gamma \vdash f \downarrow c : \neg A}\ \overline{\texttt{TApp}}$$

$$\frac{\Gamma \vdash f : A \supset B}{\Gamma \vdash \lceil f \rceil : +(A \to B)}\ \texttt{TFClo} \qquad \frac{\Gamma \vdash f : A \supset B}{\Gamma \vdash \lfloor f \rfloor : \neg(A - B)}\ \overline{\texttt{TFClo}}$$

$$\frac{\Gamma \vdash e : +(A \to B)}{\Gamma \vdash \overline{e} : A \supset B}\ \texttt{TCFun} \qquad \frac{\Gamma \vdash c : \neg(A - B)}{\Gamma \vdash \underline{c} : A \supset B}\ \overline{\texttt{TCFun}}$$

$$\frac{}{\Gamma \vdash_p () : +\top}\ \texttt{PUnit} \qquad \frac{}{\Gamma \vdash_q \{\} : \neg\bot}\ \overline{\texttt{QUnit}}$$

$$\frac{}{\Gamma_x \cup \{x : +A\} \vdash_p x : +A}\ \texttt{PVar} \qquad \frac{}{\Gamma_y \cup \{y : \neg B\} \vdash_q y : \neg B}\ \overline{\texttt{QVar}}$$

$$\frac{}{\Gamma_g \cup \{g : A \supset B\} \vdash_p \lceil g \rceil : +(A \to B)}\texttt{PFClo} \qquad \frac{}{\Gamma_g \cup \{g : A \supset B\} \vdash_q \lfloor g \rfloor : \neg(A - B)}\overline{\texttt{QFClo}}$$

$$\frac{\Gamma_{p_2} \vdash_p p_1 : +A \quad \Gamma_{p_1} \vdash_p p_2 : +B}{\Gamma \vdash_p (p_1, p_2) : +(A \wedge B)}\ \texttt{PAnd} \qquad \frac{\Gamma_{q_2} \vdash_q q_1 : \neg A \quad \Gamma_{q_1} \vdash_q q_2 : \neg B}{\Gamma \vdash_q \{q_1, q_2\} : \neg(A \vee B)}\ \overline{\texttt{QOr}}$$

$$\frac{\vdash e : +A \quad \vdash c : \neg A}{\vdash \langle\ e \mid c\ \rangle}\ \texttt{TProg1} \qquad \frac{\vdash e : +A \quad \vdash f : A \supset B \quad \vdash c : \neg B}{\vdash \langle\ e \mid f \mid c\ \rangle}\ \texttt{TProg2}$$

Figure 3: Typing Rules for SLC

$$\Gamma_{\mathsf{var}} = \{(\mathsf{var}' : T) \in \Gamma \mid \mathsf{var}' \neq \mathsf{var}\}$$

$$\begin{array}{rclcrcl} \Gamma_{()} & = & \Gamma & & \Gamma_{\{\}} & = & \Gamma \\ \Gamma_{\lceil g \rceil} & = & \Gamma_g & & \Gamma_{\lfloor g \rfloor} & = & \Gamma_g \\ \Gamma_{(p_1, p_2)} & = & (\Gamma_{p_1})_{p_2} & & \Gamma_{\{q_1, q_2\}} & = & (\Gamma_{q_1})_{q_2} \end{array}$$

Figure 4: Definition of Environment Removed Patterns

$$
\begin{array}{lrcll}
(\mathit{begin}) & e : +\mathsf{int} & \rightsquigarrow & \langle\ e \mid \bullet\ \rangle & \\
(\mathit{left}) & \langle\ (e_1,\, e_2) \mid c\ \rangle & \rightsquigarrow & \langle\ e_1 \mid (x \Rightarrow (x,\, e_2)) \downarrow c\ \rangle & \\
(\mathit{right}) & \langle\ (e_1,\, e_2) \mid c\ \rangle & \rightsquigarrow & \langle\ e_2 \mid (x \Rightarrow (e_1,\, x)) \downarrow c\ \rangle & \\
(\mathit{pop}) & \langle\ e \uparrow f \mid c\ \rangle & \rightsquigarrow & \langle\ e \mid f \mid c\ \rangle & \\
(\mathit{push}) & \langle\ e \mid \underline{f} \mid c\ \rangle & \rightsquigarrow & \langle\ e \mid f \downarrow c\ \rangle & \\
(\mathit{exc}) & \langle\ e \mid \overline{e'} \mid c\ \rangle & \rightsquigarrow & \langle\ e' \mid (\lceil g \rceil \Rightarrow e \uparrow g) \downarrow c\ \rangle & \\
(\beta L) & \langle\ e \mid p \Rightarrow e' \mid c\ \rangle & \rightsquigarrow & \langle\ e'\, [\mathcal{L}[\![p]\!] \mapsto e] \mid c\ \rangle & \\
(\beta R) & \langle\ e \mid p \Rightarrow e' \mid c\ \rangle & \rightsquigarrow & \langle\ e'\, [\mathcal{R}[\![p]\!] \mapsto e] \mid c\ \rangle & \text{if } e \text{ matches } p \\
(\overline{\beta R}) & \langle\ e \mid c' \Leftarrow q \mid c\ \rangle & \rightsquigarrow & \langle\ e \mid c'\, [\overline{\mathcal{R}}[\![q]\!] \mapsto c]\ \rangle & \text{if } c \text{ matches } q \\
(\overline{\beta L}) & \langle\ e \mid c' \Leftarrow q \mid c\ \rangle & \rightsquigarrow & \langle\ e \mid c'\, [\overline{\mathcal{L}}[\![q]\!] \mapsto c]\ \rangle & \\
(\overline{\mathit{exc}}) & \langle\ e \mid \underline{c'} \mid c\ \rangle & \rightsquigarrow & \langle\ e \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor) \mid c'\ \rangle & \\
(\overline{\mathit{push}}) & \langle\ e \mid f \mid c\ \rangle & \rightsquigarrow & \langle\ e \uparrow f \mid c\ \rangle & \\
(\overline{\mathit{pop}}) & \langle\ e \mid f \downarrow c\ \rangle & \rightsquigarrow & \langle\ e \mid f \mid c\ \rangle & \\
(\overline{\mathit{right}}) & \langle\ e \mid \{c_1,\, c_2\}\ \rangle & \rightsquigarrow & \langle\ e \uparrow (\{c_1,\, y\} \Leftarrow y) \mid c_2\ \rangle & \\
(\overline{\mathit{left}}) & \langle\ e \mid \{c_1,\, c_2\}\ \rangle & \rightsquigarrow & \langle\ e \uparrow (\{y,\, c_2\} \Leftarrow y) \mid c_1\ \rangle & \\
(\overline{\mathit{end}}) & \langle\ n \mid \bullet\ \rangle & \rightsquigarrow & n &
\end{array}
$$

Figure 5: Non-Deterministic Small-Step Semantics of SLC

into a value of type $+B$ using f, and passes the result to c. In other words, the function f transforms a continuation of type $\neg B$ to a continuation of type $\neg A$, effectively acting as a function from $\neg B$ to $\neg A$.

The rules `TProg1` and `TProg2` at the bottom of Figure 3 are the typing rules for configurations. A configuration is well-typed if all of its components are well-typed under an empty environment.

In this type system, the function call/cc has a function type $((A \to B) \to A) \supset A$ (what we call Peirce's Law).

2.4 Reduction Rules (Non-Deterministic)

In the pure lambda-calculus, reduction rules are written as a binary relation between terms because it consists of only terms. SLC operates not only on expressions but also on continuations. Therefore, reduction rules of SLC are written as a binary relation between configurations, $\langle\ e \mid c\ \rangle$ or $\langle\ e \mid f \mid c\ \rangle$. Figure 5 shows the (non-deterministic) reduction rules of SLC.

In the standard lambda-calculus, reduction proceeds by decomposing the input into a redex and a context, reducing the redex, and plugging the result into the context. In SLC, such decomposition and plugging are described within the reduction rules. If the expression e in a configuration $\langle\ e \mid c\ \rangle$ is an application, its function part is popped from the application via (*pop*) and the focus of reduction moves to the function part. In case we do not want to perform β-reduction of the popped function right now, the rule (*push*) pushes the function to the continuation part and the argument is further reduced. In case the function is not a value, on the other hand, the rule (*exc*) is used to exchange e and e' to move the focus to e'. After e' is reduced to $\lceil f \rceil$, the function $\lceil g \rceil \Rightarrow e \uparrow g$ brings it back into an application. Similarly, (*left*) and (*right*) reduce the left and right elements of a pair, respectively.

SLC has two kinds of β-reduction, eager one (βR) and lazy one (βL), depending on how to handle pattern matching. Eager β-reduction requires the argument to match the pattern at reduction time, deconstructs it, and binds variables in the pattern to each component using $\mathcal{R}$ defined in Figure 6. Lazy one does not deconstruct the argument but binds variables in the pattern to code that deconstructs the argument when the variables are used later (see the definition of $\mathcal{L}$ in Figure 6). The former is required to perform the real deconstruction of patterns, while the latter enables to defer pattern matching as long as possible (like the irrefutable pattern in Haskell).

The reduction rules for continuations are defined completely symmetrically to those for expressions. In particular, we can freely capture the current continuation using continuation abstractions, just as we can freely receive the current expression using expression abstractions.

As an example of reduction, we can capture the current continuation in x by passing a function $\lceil x \Rightarrow e \rceil$ to call/cc. See Figure 7.

$$
\begin{array}{rcl}
[\mathcal{R}[\![x]\!] \mapsto e] & = & [e/x] \\
[\mathcal{R}[\![()]\!] \mapsto e] & = & [] \\
[\mathcal{R}[\![\lceil g \rceil]\!] \mapsto \lceil f \rceil] & = & [f/g] \\
[\mathcal{R}[\![(p_1,\, p_2)]\!] \mapsto (e_1,\, e_2)] & = & [\mathcal{R}[\![p_1]\!] \mapsto e_1][\mathcal{R}[\![p_2]\!] \mapsto e_2] \\
[\overline{\mathcal{R}}[\![\{q_1,\, q_2\}]\!] \mapsto \{c_1,\, c_2\}] & = & [\overline{\mathcal{R}}[\![q_1]\!] \mapsto c_1][\overline{\mathcal{R}}[\![q_2]\!] \mapsto c_2] \\
[\overline{\mathcal{R}}[\![\lfloor g \rfloor]\!] \mapsto \lfloor f \rfloor] & = & [f/g] \\
[\overline{\mathcal{R}}[\![\{\}]\!] \mapsto c] & = & [] \\
[\overline{\mathcal{R}}[\![y]\!] \mapsto c] & = & [c/y]
\end{array}
$$

$$
\begin{array}{rcl}
[\mathcal{L}[\![x]\!] \mapsto e] & = & [e/x] \\
[\mathcal{L}[\![()]\!] \mapsto e] & = & [] \\
[\mathcal{L}[\![\lceil g \rceil]\!] \mapsto e] & = & [\overline{e}/g] \\
[\mathcal{L}[\![(p_1,\, p_2)]\!] \mapsto e] & = & [\mathcal{L}[\![p_1]\!] \mapsto e \uparrow ((x_1,\, x_2) \Rightarrow x_1)][\mathcal{L}[\![p_2]\!] \mapsto e \uparrow ((x_1,\, x_2) \Rightarrow x_1)] \\
[\overline{\mathcal{L}}[\![\{q_1,\, q_2\}]\!] \mapsto c] & = & [\overline{\mathcal{L}}[\![q_1]\!] \mapsto (y_1 \Leftarrow \{y_1,\, y_2\}) \downarrow c][\overline{\mathcal{L}}[\![q_2]\!] \mapsto (y_2 \Leftarrow \{y_1,\, y_2\}) \downarrow c] \\
[\overline{\mathcal{L}}[\![\lfloor g \rfloor]\!] \mapsto c] & = & [\underline{c}/g] \\
[\overline{\mathcal{L}}[\![\{\}]\!] \mapsto c] & = & [] \\
[\overline{\mathcal{L}}[\![y]\!] \mapsto c] & = & [c/y]
\end{array}
$$

Figure 6: Definition of Eager and Lazy Substitution

$$
\begin{array}{rll}
 & \langle\ \lceil x \Rightarrow e \rceil \mid \mathsf{call/cc} \mid c\ \rangle & \\
\rightsquigarrow & \langle\ \lceil x \Rightarrow e \rceil \mid (\lceil g \rceil \Rightarrow \lceil y \Leftarrow \_ \rceil \uparrow g) \downarrow y \Leftarrow y \mid c\ \rangle & (\text{definition of } \mathsf{call/cc}) \\
\rightsquigarrow & \langle\ \lceil x \Rightarrow e \rceil \mid (\lceil g \rceil \Rightarrow \lceil c \Leftarrow \_ \rceil \uparrow g) \downarrow c\ \rangle & (\overline{\beta R}) \\
\rightsquigarrow & \langle\ \lceil x \Rightarrow e \rceil \mid \lceil g \rceil \Rightarrow \lceil c \Leftarrow \_ \rceil \uparrow g \mid c\ \rangle & (\overline{pop}) \\
\rightsquigarrow & \langle\ \lceil c \Leftarrow \_ \rceil \uparrow (x \Rightarrow e) \mid c\ \rangle & (\beta R) \\
\rightsquigarrow & \langle\ \lceil c \Leftarrow \_ \rceil \mid x \Rightarrow e \mid c\ \rangle & (pop) \\
\rightsquigarrow & \langle\ e[\lceil c \Leftarrow \_ \rceil / x] \mid c\ \rangle & (\beta R)
\end{array}
$$

Figure 7: Example Reduction of $\mathsf{call/cc}$

There can be multiple rules that are applicable to the same configuration. Furthermore, the reduction rules are not Church-Rosser, that is, the result of evaluation can be different values. For example, the configuration $\langle\ 1 \uparrow (\bullet \Leftarrow y) \mid (x \Rightarrow 2) \downarrow \bullet\ \rangle$ has the following two different reductions:

$$
\begin{array}{rlll}
 & \langle\ 1 \uparrow (\bullet \Leftarrow y) \mid (x \Rightarrow 2) \downarrow \bullet\ \rangle & & \\
\rightsquigarrow & \langle\ 1 \mid \bullet \Leftarrow y \mid (x \Rightarrow 2) \downarrow \bullet\ \rangle & (pop) & \\
\rightsquigarrow & \langle\ 1 \mid \bullet\ \rangle & (\overline{\beta}) & \\
\rightsquigarrow & 1 & (\overline{end}) &
\end{array}
\qquad
\begin{array}{rlll}
 & \langle\ 1 \uparrow (\bullet \Leftarrow y) \mid (x \Rightarrow 2) \downarrow \bullet\ \rangle & & \\
\rightsquigarrow & \langle\ 1 \uparrow (\bullet \Leftarrow y) \mid (x \Rightarrow 2) \mid \bullet\ \rangle & (\overline{pop}) & \\
\rightsquigarrow & \langle\ 2 \mid \bullet\ \rangle & (\beta) & \\
\rightsquigarrow & 2 & (\overline{end}) &
\end{array}
$$

This is not surprising. Without fixing the evaluation strategy, the result can be different. In a functional language, for example, $(\lambda x.1)(3/0)$ is reduced to 1 in CBN and an error in CBV. To recover uniqueness of evaluation, we introduce CBV evaluation strategy into SLC in Section 3 and CBN one in Section 4.

2.5 Properties of non-deterministic SLC

Without specifying the evaluation strategy, the small-step reduction semantics for non-deterministic SLC is sound with respect to the type system. Let $\langle ... \rangle$ (possibly with a subscript) denote either a two-place configuration $\langle\ e \mid c\ \rangle$ or a three-place configuration $\langle\ e \mid f \mid c\ \rangle$. We can show the progress by simple case analysis and the preservation using the substitution lemma.

Theorem 2.1 (Progress)
If $\vdash \langle ... \rangle_1$, then $\langle ... \rangle_1 \rightsquigarrow \langle ... \rangle_2$ for some $\langle ... \rangle_2$ or $\langle ... \rangle_1 = \langle\ n \mid \bullet\ \rangle$ for some n.

$$
\begin{array}{rrcl}
\text{value} & v & ::= & n \mid x \mid () \mid (v, v) \mid \lceil f \rceil \mid [v \uparrow inl] \mid [v \uparrow inr] \mid [v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)] \\
\text{expression} & e & ::= & v \mid (e, e) \mid e \uparrow f \\
\text{function} & f & ::= & g \mid p \Rightarrow e \mid c \Leftarrow q \mid \overline{e} \mid \underline{c} \\
\text{continuation} & c & ::= & \bullet \mid y \mid \{\} \mid \{c, c\} \mid f \downarrow c \mid \lfloor f \rfloor \mid [inl \downarrow c] \mid [inr \downarrow c]
\end{array}
$$

Figure 8: Syntax of CBV SLC

$$
\frac{\Gamma \vdash v : +A}{\Gamma \vdash [v \uparrow inl] : +(A \vee B)}\ \texttt{TInl} \qquad \frac{\Gamma \vdash c : \neg(A \vee B)}{\Gamma \vdash [inl \downarrow c] : \neg A}\ \overline{\texttt{TInl}}
$$

$$
\frac{\Gamma \vdash v : +B}{\Gamma \vdash [v \uparrow inr] : +(A \vee B)}\ \texttt{TInr} \qquad \frac{\Gamma \vdash c : \neg(A \vee B)}{\Gamma \vdash [inr \downarrow c] : \neg B}\ \overline{\texttt{TInr}}
$$

$$
\frac{\Gamma \vdash v : +A \quad \Gamma \vdash c : \neg B}{\Gamma \vdash [v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)] : +(A - B)}\ \texttt{TCtx}
$$

Figure 9: Additional Typing Rules for CBV SLC

Theorem 2.2 (Preservation)
Assume $\vdash \langle ... \rangle_1$. If $\langle ... \rangle_1 \rightsquigarrow \langle ... \rangle_2$, then $\vdash \langle ... \rangle_2$.

3 CBV SLC

The reduction rules in Figure 5 are non-deterministic. In this section, we introduce the CBV evaluation strategy into SLC. Under CBV, an argument is evaluated to a value before β-reduction. In SLC, it means that the evaluation goes from the expression side of the configuration $\langle ... \rangle$, or from left to right of $\langle ... \rangle$. To enforce this evaluation strategy, we first introduce a value into the syntax of SLC as in Figure 8. In addition to integers, variables, units, pairs of values, and higher-order functions, the value contains three *frozen* values that are expressions enclosed in brackets. In the non-deterministic setting, they are operationally the same as the expressions without brackets using the following interpretation:

$$
\begin{array}{rcl}
inl & = & y_1 \Leftarrow \{y_1, y_2\} \\
inr & = & y_2 \Leftarrow \{y_1, y_2\}
\end{array}
$$

In the CBV setting, they are used to control the order of evaluation. Likewise, we have introduced two frozen continuations. The typing rules for the frozen values and frozen continuations are the same as those without brackets. They are summarized in Figure 9.

Currently, values include a function value of the form $\lceil \overline{e} \rceil$. If we want to exclude this case from values, we could separate f into value functions and non-value functions, and include only the former into values.

Figure 10 shows the reduction rules for CBV SLC. The rule names with the subscript v indicate the rules that are changed from non-deterministic ones (Figure 5). The rule names with primes are the reduction rules for frozen values/continuations. The rules marked with $*$ are ones directly obtained from Filinski's denotational semantics. We will explain it in detail in Section 5.2. Rules in Figure 10 are restriction of the reduction rules for the non-deterministic SLC in a way the evaluation order is fixed to CBV: evaluation goes from left to right in the configuration $\langle ... \rangle$. For example, the rule (βR_{v}) requires that the argument is fully evaluated (and hence the use of eager pattern deconstruction $\mathcal{R}$); the rules $(left_{\mathrm{v}})$ and $(right_{\mathrm{v}})$ force the left-to-right evaluation order for pairs. Rules for frozen values/continuations first arise in $(\overline{exc}_{\mathrm{v}})$ when a continuation appears as a function. In this case, we freeze v and c into $[v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)]$ and evaluate c'. Here, we cannot remove the bracket (as in $(\overline{exc})$ in

$$
\begin{array}{rrcll}
 & (begin) & e : +\mathsf{int} & \rightsquigarrow & \langle\ e \mid \bullet\ \rangle & \\
 & (left_{\mathrm{v}}) & \langle\ (e_1,\, e_2) \mid c\ \rangle & \rightsquigarrow & \langle\ e_1 \mid (x \Rightarrow (x,\, e_2)) \downarrow c\ \rangle & \text{if } e_1 \neq v \\
 & (right_{\mathrm{v}}) & \langle\ (v_1,\, e_2) \mid c\ \rangle & \rightsquigarrow & \langle\ e_2 \mid (x \Rightarrow (v_1,\, x)) \downarrow c\ \rangle & \text{if } e_2 \neq v \\
 & (pop) & \langle\ e \uparrow f \mid c\ \rangle & \rightsquigarrow & \langle\ e \mid f \mid c\ \rangle & \\
 & (push_{\mathrm{v}}) & \langle\ e \mid \underline{f} \mid c\ \rangle & \rightsquigarrow & \langle\ e \mid f \downarrow c\ \rangle & \text{if } e \neq v \\
* & (exc_{\mathrm{v}}) & \langle\ v \mid \overline{e'} \mid c\ \rangle & \rightsquigarrow & \langle\ e' \mid (\lceil g \rceil \Rightarrow v \uparrow g) \downarrow c\ \rangle & \\
* & (inl') & \langle\ [v \uparrow inl] \mid \{c_1,\, c_2\}\ \rangle & \rightsquigarrow & \langle\ v \mid c_1\ \rangle & \\
* & (inr') & \langle\ [v \uparrow inr] \mid \{c_1,\, c_2\}\ \rangle & \rightsquigarrow & \langle\ v \mid c_2\ \rangle & \\
* & (contx') & \langle\ [v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)] \mid \lfloor f \rfloor\ \rangle & \rightsquigarrow & \langle\ v \mid f \mid c\ \rangle & \\
* & (\beta R_{\mathrm{v}}) & \langle\ v \mid p \Rightarrow e' \mid c\ \rangle & \rightsquigarrow & \langle\ e'\,[\mathcal{R}[\![p]\!] \mapsto v] \mid c\ \rangle & \\
* & (\overline{\beta L_{\mathrm{v}}}) & \langle\ v \mid c' \Leftarrow q \mid c\ \rangle & \rightsquigarrow & \langle\ v \mid c'\,[\overline{\mathcal{L}}_{\mathrm{v}}[\![q]\!] \mapsto c]\ \rangle & \\
* & (\overline{inr'}) & \langle\ v \mid [inr \downarrow c]\ \rangle & \rightsquigarrow & \langle\ [v \uparrow inr] \mid c\ \rangle & \\
* & (\overline{inl'}) & \langle\ v \mid [inl \downarrow c]\ \rangle & \rightsquigarrow & \langle\ [v \uparrow inl] \mid c\ \rangle & \\
* & (\overline{exc_{\mathrm{v}}}) & \langle\ v \mid \underline{c'} \mid c\ \rangle & \rightsquigarrow & \langle\ [v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)] \mid c'\ \rangle & \\
* & (\overline{pop_{\mathrm{v}}}) & \langle\ v \mid f \downarrow c\ \rangle & \rightsquigarrow & \langle\ v \mid f \mid c\ \rangle & \\
* & (\overline{end}) & \langle\ n \mid \bullet\ \rangle & \rightsquigarrow & n &
\end{array}
$$

Figure 10: Small-Step Semantics for CBV SLC

$$
\begin{array}{rcl}
[\overline{\mathcal{L}}_{\mathrm{v}}[\![\{q_1,\, q_2\}]\!] \mapsto c] & = & [\overline{\mathcal{L}}_{\mathrm{v}}[\![q_1]\!] \mapsto [inl \downarrow c]][\overline{\mathcal{L}}_{\mathrm{v}}[\![q_2]\!] \mapsto [inr \downarrow c]] \\
[\overline{\mathcal{L}}_{\mathrm{v}}[\![\lfloor g \rfloor]\!] \mapsto c] & = & [\underline{c}/g] \\
[\overline{\mathcal{L}}_{\mathrm{v}}[\![\{\}]\!] \mapsto c] & = & [] \\
[\overline{\mathcal{L}}_{\mathrm{v}}[\![y]\!] \mapsto c] & = & [c/y]
\end{array}
$$

Figure 11: Definition of Lazy Substitution for CBV SLC

Figure 5), because it would lead to non-termination in CBV:

$$
\begin{array}{rcll}
\langle\ v \mid \underline{c'} \mid c\ \rangle & \rightsquigarrow & \langle\ v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor) \mid c'\ \rangle & (\overline{exc}) \\
 & \rightsquigarrow & \langle\ v \mid g \downarrow c \Leftarrow \lfloor g \rfloor \mid c'\ \rangle & (pop) \\
 & \rightsquigarrow & \langle\ v \mid \underline{c'} \downarrow c\ \rangle & (\overline{\beta L_{\mathrm{v}}}) \\
 & \rightsquigarrow & \langle\ v \mid \underline{c'} \mid c\ \rangle & (\overline{pop_{\mathrm{v}}})
\end{array}
$$

By temporarily freezing the application $[v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)]$ as a value, we enforce the evaluation of c'. The frozen value is destructed in $(contx')$ only when the evaluation of c' is finished. Filinski called this frozen value $[v \uparrow (g \downarrow c \Leftarrow \lfloor g \rfloor)]$ a *context*.

Because the evaluation goes from left to right, an interesting asymmetry arises between expression abstractions and continuation abstractions: although the argument to an expression abstraction is always a value in CBV, the argument continuation to a continuation abstraction is not a value in general. In other words, an expression abstraction is evaluated in CBV but a continuation abstraction is evaluated in CBN.

Because the argument continuation in $(\overline{\beta L_{\mathrm{v}}})$ is not evaluated yet, we cannot use the eager pattern deconstruction here. Instead, we use the CBV lazy pattern deconstruction shown in Figure 11. Unlike $\overline{\mathcal{L}}$ (in Figure 6), $\overline{\mathcal{L}}_{\mathrm{v}}$ introduces frozen injections ($[inl \downarrow c]$ and $[inr \downarrow c]$) to defer the pattern deconstruction. They force the evaluation of continuations only when the pattern deconstruction is actually needed in $(\overline{inl'})$ or $(\overline{inr'})$.

3.1 Properties of CBV SLC

Since the CBV reduction semantics is a special case of the non-deterministic reduction semantics, the preservation holds for CBV reduction semantics. As for the progress, we can show by simple case analysis that the CBV reduction semantics satisfies the following stronger property:

$$
\begin{array}{rrcl}
\text{expression} & e & ::= & n \mid x \mid () \mid (e,\, e) \mid e \uparrow f \mid \lceil f \rceil \mid [e \uparrow fst] \mid [e \uparrow snd] \\
\text{function} & f & ::= & g \mid p \Rightarrow e \mid c \Leftarrow q \mid \overline{e} \mid \underline{c} \\
\text{continuation} & c & ::= & k \mid \{c,\, c\} \mid f \downarrow c \\
\text{covalue} & k & ::= & \bullet \mid y \mid \{\} \mid \{k,\, k\} \mid \lfloor f \rfloor \mid [(\lceil g \rceil \Rightarrow e \uparrow g) \downarrow k] \mid [fst \downarrow k] \mid [snd \downarrow k]
\end{array}
$$

Figure 12: Syntax of CBN SLC

$$
\frac{\Gamma \vdash e : +(A \wedge B)}{\Gamma \vdash [e \uparrow fst] : +A}\ \texttt{TFst} \qquad \frac{\Gamma \vdash k : \neg A}{\Gamma \vdash [fst \downarrow k] : \neg(A \wedge B)}\ \overline{\texttt{TFst}}
$$

$$
\frac{\Gamma \vdash e : +(A \wedge B)}{\Gamma \vdash [e \uparrow snd] : +B}\ \texttt{TSnd} \qquad \frac{\Gamma \vdash k : \neg B}{\Gamma \vdash [snd \downarrow k] : \neg(A \wedge B)}\ \overline{\texttt{TSnd}}
$$

$$
\frac{\Gamma \vdash e : +A \quad \Gamma \vdash k : \neg B}{\Gamma \vdash [(\lceil g \rceil \Rightarrow k \uparrow g) \downarrow e] : \neg(A \to B)}\ \overline{\texttt{TCtx}}
$$

Figure 13: Additional Typing Rules of CBN SLC

Theorem 3.1 (The Uniqueness of Reduction in CBV SLC)
Under CBV, if $\vdash \langle ... \rangle$, then there is exactly one reduction rule applicable to $\langle ... \rangle$.

Furthermore, using the standard logical relation argument (as found in [9, Section 12]) tailored for SLC, we can show the following theorem:

Theorem 3.2 (Termination of Evaluation in CBV SLC)
Under CBV, if $\vdash e : +\mathsf{int}$, then there exists a unique n such that $e \rightsquigarrow^* n$.

4 CBN SLC

The CBN evaluation strategy evaluates expressions only when it is needed by its context. In other words, the evaluation goes from the continuation side, or from right to left of $\langle ... \rangle$. Because of this duality, CBN SLC can be mechanically obtained by repeating the construction of CBV SLC in the previous section with the roles of expressions and continuations swapped: the evaluation goes from the continuation side of the configuration $\langle ... \rangle$, or from right to left of $\langle ... \rangle$. The result is a mirror image of CBV SLC. Furthermore, the same properties hold for CBN SLC.

Dually to CBV SLC, we first introduce a *covalue* (that is a value of continuations) into the syntax of SLC as in Figure 12. It is defined as completely dual notion of the value v in CBV SLC. Therefore the covalue contains three frozen covalues that are continuations enclosed in brackets. In the non-deterministic setting, they are operationally the same as the continuations without brackets using the following interpretation:

$$
\begin{array}{rcl}
fst & = & (x_1,\, x_2) \Rightarrow x_1 \\
snd & = & (x_1,\, x_2) \Rightarrow x_1
\end{array}
$$

In the CBN setting, they are used to control the order of evaluation. Likewise, we have introduced two frozen expressions. Figure 13 shows typing rules for frozen covalues and frozen expressions.

Figure 14 shows the reduction rules for CBN SLC. The rule names with the subscript n indicate the rules that are changed from non-deterministic ones (Figure 5). The rule names with primes are the reduction rules for frozen expressions/covalues. The rules marked with $*$ are the ones directly obtained from Filinski's denotational semantics. We will explain it in detail in Section 6.2. Rules in Figure 14 are restriction of the reduction rules for the non-deterministic SLC in a way the evaluation order is fixed to CBN: evaluation goes from right to left in the configuration $\langle ... \rangle$.

$$
\begin{array}{cr@{\quad}rl@{\quad}l}
 & (\mathit{begin}) & e : +\mathsf{int} & \leadsto & \langle\, e \mid \bullet \,\rangle \\
* & (\mathit{pop}_{\mathrm{n}}) & \langle\, e \uparrow \underline{f} \mid k \,\rangle & \leadsto & \langle\, e \mid f \mid k \,\rangle \\
* & (\mathit{exc}_{\mathrm{n}}) & \langle\, e \mid \overline{e'} \mid k \,\rangle & \leadsto & \langle\, e' \mid [(\lceil g \rceil \Rightarrow e \uparrow g) \downarrow k] \,\rangle \\
* & (\mathit{fst}') & \langle\, [e \uparrow \mathit{fst}] \mid k \,\rangle & \leadsto & \langle\, e \mid [\mathit{fst} \downarrow k] \,\rangle \\
* & (\mathit{snd}') & \langle\, [e \uparrow \mathit{snd}] \mid k \,\rangle & \leadsto & \langle\, e \mid [\mathit{snd} \downarrow k] \,\rangle \\
* & (\beta L_{\mathrm{n}}) & \langle\, e \mid p \Rightarrow e' \mid k \,\rangle & \leadsto & \langle\, e'[\mathcal{L}_{\mathrm{n}}[\![p]\!] \mapsto e] \mid k \,\rangle \\
* & (\overline{\beta R_{\mathrm{n}}}) & \langle\, e \mid c' \Leftarrow q \mid k \,\rangle & \leadsto & \langle\, e \mid c'[\overline{\mathcal{R}}[\![q]\!] \mapsto k] \,\rangle \\
* & (\overline{\mathit{contx}'}) & \langle\, \lceil f \rceil \mid [(\lceil g \rceil \Rightarrow e \uparrow g) \downarrow k] \,\rangle & \leadsto & \langle\, e \mid f \mid k \,\rangle \\
* & (\overline{\mathit{snd}'}) & \langle\, (e_1, e_2) \mid [\mathit{snd} \downarrow k] \,\rangle & \leadsto & \langle\, e_2 \mid k \,\rangle \\
* & (\overline{\mathit{fst}'}) & \langle\, (e_1, e_2) \mid [\mathit{fst} \downarrow k] \,\rangle & \leadsto & \langle\, e_1 \mid k \,\rangle \\
* & (\overline{\mathit{exc}_{\mathrm{n}}}) & \langle\, e \mid \underline{c'} \mid k \,\rangle & \leadsto & \langle\, e \uparrow (g \downarrow k \Leftarrow \lfloor g \rfloor) \mid c' \,\rangle \\
 & (\overline{\mathit{push}_{\mathrm{n}}}) & \langle\, e \mid f \mid c \,\rangle & \leadsto & \langle\, e \uparrow f \mid c \,\rangle \quad \text{if } c \neq k \\
 & (\overline{\mathit{pop}}) & \langle\, e \mid f \downarrow c \,\rangle & \leadsto & \langle\, e \mid f \mid c \,\rangle \\
 & (\overline{\mathit{right}_{\mathrm{n}}}) & \langle\, e \mid \{c_1, c_2\} \,\rangle & \leadsto & \langle\, e \uparrow (\{c_1, y\} \Leftarrow y) \mid c_2 \,\rangle \quad \text{if } c_2 \neq k \\
 & (\overline{\mathit{left}_{\mathrm{n}}}) & \langle\, e \mid \{c_1, k_2\} \,\rangle & \leadsto & \langle\, e \uparrow (\{y, k_2\} \Leftarrow y) \mid c_1 \,\rangle \quad \text{if } c_1 \neq k \\
* & (\overline{\mathit{end}}) & \langle\, n \mid \bullet \,\rangle & \leadsto & n
\end{array}
$$

Figure 14: Small-Step Semantics for CBN SLC

$$
\begin{array}{rcl}
[\mathcal{L}_{\mathrm{n}}[\![x]\!] \mapsto e] & = & [e/x] \\
[\mathcal{L}_{\mathrm{n}}[\![()]\!] \mapsto e] & = & [] \\
[\mathcal{L}_{\mathrm{n}}[\![\lceil g \rceil]\!] \mapsto e] & = & [\overline{e}/g] \\
[\mathcal{L}_{\mathrm{n}}[\![(p_1, p_2)]\!] \mapsto e] & = & [\mathcal{L}_{\mathrm{n}}[\![p_1]\!] \mapsto [e \uparrow \mathit{fst}]][\mathcal{L}_{\mathrm{n}}[\![p_2]\!] \mapsto [e \uparrow \mathit{snd}]]
\end{array}
$$

Figure 15: Definition of Lazy Substitution for CBN SLC

4.1 Properties of CBN SLC

Since the CBN reduction semantics is a special case of the non-deterministic reduction semantics, the preservation holds for CBN reduction semantics. As for the progress, we can show by simple case analysis that the CBN reduction semantics satisfies the following stronger property:

Theorem 4.1 (The Uniqueness of Reduction in CBN SLC)
Under CBN, if $\vdash \langle ... \rangle$, then there is exactly one reduction rule applicable to $\langle ... \rangle$.

Furthermore, using the standard logical relation argument (as found in [9, Section 12]) tailored for SLC, we can show the following theorem:

Theorem 4.2 (Termination of Evaluation in CBN SLC)
Under CBN, if $\vdash e : +\mathsf{int}$, then there exists a unique n such that $e \leadsto^* n$.

5 Functional Correspondence for CBV SLC

In this section, we show that the CBV small-step semantics of SLC presented in Section 3 corresponds to Filinski's original definition of CBV SLC given as a denotational semantics. The method we use is the functional correspondence of interpreters and abstract machines developed by Danvy and his group [1, 4]: an abstract machine is a CPS-transformed defunctionalized interpreter. In our case, since Filinski's denotational semantics is already in (a kind of) CPS, the small-step semantics (an abstract machine) can be obtained by defunctionalizing the denotational semantics.

Figure 16 shows Filinski's denotational semantics for CBV SLC. Since the syntax of SLC consists of three kinds, the semantics also consists of three functions: $\mathcal{E}$, $\mathcal{C}$, and $\mathcal{F}$. In Filinski's formulation, the patterns do not contain the function pattern. The semantic functions are not expressed in the symmetric manner, because SLC is mapped into the standard (asymmetric) lambda-calculus. In the semantics, ρ

$$
\begin{array}{rcl}
Val & = & Int + Unit() + Pair(Val \times Val) + In_1(Val) + In_2(Val) + \\
 & & Closr(Val \to Cnt \to Ans) + Contx(Val \times Cont) \\
Cnt & = & Val \to Ans \\
Env & = & Ide \to (Val + Cnt)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E} : e \to Env \to Cnt & \to & Ans \\
\mathcal{E}[\![n]\!]\rho\kappa & = & \kappa\ n \\
\mathcal{E}[\![x]\!]\rho\kappa & = & let\ val(v) = \rho\ x\ in\ \kappa\ v \\
\mathcal{E}[\![(e_1,\ e_2)]\!]\rho\kappa & = & \mathcal{E}[\![e_1]\!]\,\rho\,(\lambda v_1.\mathcal{E}[\![e_2]\!]\,\rho\,(\lambda v_2.\kappa\ pair(v_1, v_2))) \\
\mathcal{E}[\![()]\!]\rho\kappa & = & \kappa\ unit() \\
\mathcal{E}[\![e \uparrow f]\!]\rho\kappa & = & \mathcal{E}[\![e]\!]\,\rho\,(\lambda v.\mathcal{F}[\![f]\!]\rho v\kappa) \\
\mathcal{E}[\![\lceil f \rceil]\!]\rho\kappa & = & \kappa\ closr(\lambda v\kappa'.\mathcal{F}[\![f]\!]\rho v\kappa')
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C} : c \to Env \to Val & \to & Ans \\
\mathcal{C}[\![y]\!]\rho v & = & let\ cnt(\kappa) = \rho\ y\ in\ \kappa\ v \\
\mathcal{C}[\![\{c_1,\ c_2\}]\!]\rho v & = & case\ v\ of\ in_1(v') : \mathcal{C}[\![c_1]\!]\rho v' \mid in_2(v') : \mathcal{C}[\![c_2]\!]\rho v'\ esac \\
\mathcal{C}[\![\{\}]\!]\rho v & = & case\ v\ of\ esac \\
\mathcal{C}[\![f \downarrow c]\!]\rho v & = & \mathcal{F}[\![f]\!]\,\rho\,v\,(\lambda v'.\mathcal{C}[\![c]\!]\rho v') \\
\mathcal{C}[\![\lfloor f \rfloor]\!]\rho v & = & let\ contx(v', \kappa) = v\ in\ \mathcal{F}[\![f]\!]\rho v'\kappa
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F} : f \to Env \to Val \to Cnt & \to & Ans \\
\mathcal{F}[\![X \Rightarrow e]\!]\rho v\kappa & = & \mathcal{E}[\![e]\!]\,([\mathcal{X}[\![X]\!] \mapsto v]\rho)\,\kappa \\
\mathcal{F}[\![c \Leftarrow Y]\!]\rho v\kappa & = & \mathcal{C}[\![c]\!]\,([\mathcal{Y}[\![Y]\!] \mapsto \kappa]\rho)\,v \\
\mathcal{F}[\![\overline{e}]\!]\rho v\kappa & = & \mathcal{E}[\![e]\!]\,\rho\,(\lambda v'.let\ closr(h) = v'\ in\ h\,v\,\kappa) \\
\mathcal{F}[\![\underline{c}]\!]\rho v\kappa & = & \mathcal{C}[\![c]\!]\,\rho\,contx(v, \kappa)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{X} : X \to Val \to Env & \to & Env \\
[\mathcal{X}[\![x]\!] \mapsto v]\rho & = & \rho[x \mapsto val(v)] \\
[\mathcal{X}[\![()]\!] \mapsto v]\rho & = & let\ unit() = v\ in\ \rho \\
[\mathcal{X}[\![(X_1,\ X_2)]\!] \mapsto v]\rho & = & let\ pair(v_1, v_2) = v\ in\ [\mathcal{X}[\![X_1]\!] \mapsto v_1]([\mathcal{X}[\![X_2]\!] \mapsto v_2]\rho)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{Y} : Y \to Cnt \to Env & \to & Env \\
[\mathcal{Y}[\![y]\!] \mapsto \kappa]\rho & = & \rho[y \mapsto cnt(\kappa)] \\
[\mathcal{Y}[\![\{\}]\!] \mapsto \kappa]\rho & = & \rho \\
[\mathcal{Y}[\![\{Y_1,\ Y_2\}]\!] \mapsto \kappa]\rho & = & [\mathcal{Y}[\![Y_1]\!] \mapsto \lambda v.\kappa\ in_1(v)]\ ([\mathcal{Y}[\![Y_2]\!] \mapsto \lambda v.\kappa\ in_2(v)]\rho)
\end{array}
$$

Figure 16: Filinski's Denotational Semantics for CBV SLC

represents an environment, v represents a value (the result of evaluation), and κ represents a *semantic* continuation. Filinski introduces a domain for each value type (e.g., $Pair$) together with a constructor for it (in lower case letter, e.g., *pair*).

We will not go into details of this denotational semantics. Rather, we regard this semantics as a definitional interpreter for SLC and defunctionalize it. Defunctionalization, introduced by Reynolds [10], is a whole-program transformation to remove higher-order functions. Every higher-order function in a program is replaced with a unique constructor whose arguments are free variables of the higher-order function. When the higher-order function is applied, a newly introduced apply function $\mathcal{A}$ is used instead, which, given the actual argument, will execute the body of the higher-order function. This way, all the higher-order functions are replaced with first-order data.

5.1 Defunctionalization

We defunctionalize two kinds of higher-order functions in Figure 16: continuations (κ of type Cnt) and closures (the argument of *closr*).[3] The result of defunctionalization is found in the upper left of Figure 17.

[3]The latter is sometimes called closure conversion.

Defunctionalized Semantics		Corresponding Abstract Machine
$\mathcal{E} : e \to Env \to Cnt \to Ans$		The rules coming from $\mathcal{E}$
$\mathcal{E}[\![n]\!]\rho\kappa = \mathcal{A}\ \kappa\ n$		$\langle\ [\![n]\!]\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ n \mid \kappa\ \rangle$
$\mathcal{E}[\![x]\!]\rho\kappa = let\ val(v) = \rho\ x\ in\ \mathcal{A}\ \kappa\ v$		$\langle\ [\![x]\!]\rho \mid \kappa\ \rangle \rightsquigarrow let\ val(v) = \rho\ x\ in \langle\ v \mid \kappa\ \rangle$
$\mathcal{E}[\![(e_1,\ e_2)]\!]\rho\kappa = \mathcal{E}[\![e_1]\!]\,\rho\,(\mathsf{Ep1}(e_2, \rho, \kappa))$	$\diamond$	$\langle\ [\![(e_1,\ e_2)]\!]\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ [\![e_1]\!]\rho \mid \mathsf{Ep1}(e_2, \rho, \kappa)\ \rangle$
$\mathcal{E}[\![()]\!]\rho\kappa = \mathcal{A}\ \kappa\ unit()$		$\langle\ [\![()]\!]\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ unit()\ \mid \kappa\ \rangle$
$\mathcal{E}[\![e \uparrow f]\!]\rho\kappa = \mathcal{E}[\![e]\!]\,\rho\,(\mathsf{Epp}(f, \rho, \kappa))$	$\diamond$	$\langle\ [\![e \uparrow f]\!]\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ [\![e]\!]\rho \mid \mathsf{Epp}(f, \rho, \kappa)\ \rangle$
$\mathcal{E}[\![\lceil f \rceil]\!]\rho\kappa = \mathcal{A}\ \kappa\ \mathsf{Closr}(f, \rho)$		$\langle\ [\![\lceil f \rceil]\!]\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ \mathsf{Closr}(f, \rho) \mid \kappa\ \rangle$
$\mathcal{C} : c \to Env \to Val \to Ans$		The rules coming from $\mathcal{C}$
$\mathcal{C}[\![y]\!]\rho v = let\ cnt(\kappa) = \rho\ y\ in\ \mathcal{A}\ \kappa\ v$		$\langle\ v \mid [\![y]\!]\rho\ \rangle \rightsquigarrow let\ cnt(\kappa) = \rho\ y\ in \langle\ v \mid \kappa\ \rangle$
$\mathcal{C}[\![\{c_1,\ c_2\}]\!]\rho\ in_1(v) = \mathcal{C}[\![c_1]\!]\rho v$	$*$	$\langle\ in_1(v) \mid [\![\{c_1,\ c_2\}]\!]\rho\ \rangle \rightsquigarrow \langle\ v \mid [\![c_1]\!]\rho\ \rangle$
$\mathcal{C}[\![\{c_1,\ c_2\}]\!]\rho\ in_2(v) = \mathcal{C}[\![c_2]\!]\rho v$	$*$	$\langle\ in_2(v) \mid [\![\{c_1,\ c_2\}]\!]\rho\ \rangle \rightsquigarrow \langle\ v \mid [\![c_2]\!]\rho\ \rangle$
$\mathcal{C}[\![\{\}]\!]\rho v = case\ v\ of\ esac$		$\langle\ v \mid [\![\{\}]\!]\rho\ \rangle \rightsquigarrow case\ v\ of\ esac$
$\mathcal{C}[\![f \downarrow c]\!]\rho v = \mathcal{F}[\![f]\!]\,\rho\,v\,(\mathsf{Cpp}(c, \rho))$	$*$	$\langle\ v \mid [\![f \downarrow c]\!]\rho\ \rangle \rightsquigarrow \langle\ v \mid [\![f]\!]\rho \mid \mathsf{Cpp}(c, \rho)\ \rangle$
$\mathcal{C}[\![\lfloor f \rfloor]\!]\rho\ contx(v, \kappa) = \mathcal{F}[\![f]\!]\rho v\kappa$	$*$	$\langle\ contx(v, \kappa) \mid [\![\lfloor f \rfloor]\!]\rho\ \rangle \rightsquigarrow \langle\ v \mid [\![f]\!]\rho \mid \kappa\ \rangle$
$\mathcal{F} : f \to Env \to Val \to Cnt \to Ans$		The rules coming from $\mathcal{F}$
$\mathcal{F}[\![X \Rightarrow e]\!]\rho v\kappa = \mathcal{E}[\![e]\!]\,([\mathcal{X}[\![X]\!] \mapsto v]\rho)\,\kappa$	$*$	$\langle\ v \mid [\![X \Rightarrow e]\!]\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ [\![e]\!]\,([\mathcal{X}[\![X]\!] \mapsto v]\rho) \mid \kappa\ \rangle$
$\mathcal{F}[\![c \Leftarrow Y]\!]\rho v\kappa = \mathcal{C}[\![c]\!]\,([\mathcal{Y}[\![Y]\!] \mapsto \kappa]\rho)\,v$	$*$	$\langle\ v \mid [\![c \Leftarrow Y]\!]\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ v \mid [\![c]\!]\,([\mathcal{Y}[\![Y]\!] \mapsto \kappa]\rho)\ \rangle$
$\mathcal{F}[\![\overline{e}]\!]\rho v\kappa = \mathcal{E}[\![e]\!]\,\rho\,(\mathsf{Openr}(v, \kappa))$	$*$	$\langle\ v \mid [\![\overline{e}]\!]\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ [\![e]\!]\rho \mid \mathsf{Openr}(v, \kappa)\ \rangle$
$\mathcal{F}[\![\underline{c}]\!]\rho v\kappa = \mathcal{C}[\![c]\!]\,\rho\ contx(v, \kappa)$	$*$	$\langle\ v \mid [\![\underline{c}]\!]\rho \mid \kappa\ \rangle \rightsquigarrow \langle\ contx(v, \kappa) \mid [\![c]\!]\rho\ \rangle$
$\mathcal{A} : Cnt \to Val \to Ans$		The rules coming from $\mathcal{A}$
$\mathcal{A}\ (\mathsf{Ep1}(e_2, \rho, \kappa))\ v_1 = \mathcal{E}[\![e_2]\!]\,\rho\,(\mathsf{Ep2}(v_1, \kappa))$	$\diamond$	$\langle\ v_1 \mid \mathsf{Ep1}(e_2, \rho, \kappa)\ \rangle \rightsquigarrow \langle\ [\![e_2]\!]\rho \mid \mathsf{Ep2}(v_1, \kappa)\ \rangle$
$\mathcal{A}\ (\mathsf{Ep2}(v_1, \kappa))\ v_2 = \mathcal{A}\ \kappa\ pair(v_1, v_2)$	$\diamond$	$\langle\ v_2 \mid \mathsf{Ep2}(v_1, \kappa)\ \rangle \rightsquigarrow \langle\ pair(v_1, v_2) \mid \kappa\ \rangle$
$\mathcal{A}\ (\mathsf{Epp}(f, \rho, \kappa))\ v = \mathcal{F}[\![f]\!]\rho v\kappa$	$*$	$\langle\ v \mid \mathsf{Epp}(f, \rho, \kappa)\ \rangle \rightsquigarrow \langle\ v \mid [\![f]\!]\rho \mid \kappa\ \rangle$
$\mathcal{A}\ (\mathsf{Cpp}(c, \rho))\ v = \mathcal{C}[\![c]\!]\rho v$		$\langle\ v \mid \mathsf{Cpp}(c, \rho)\ \rangle \rightsquigarrow \langle\ v \mid [\![c]\!]\rho\ \rangle$
$\mathcal{A}\ (\mathsf{Openr}(v, \kappa))\ \mathsf{Closr}(f, \rho) = \mathcal{F}[\![f]\!]\rho v\kappa$	$\diamond$	$\langle\ \mathsf{Closr}(f, \rho) \mid \mathsf{Openr}(v, \kappa)\ \rangle \rightsquigarrow \langle\ v \mid [\![f]\!]\rho \mid \kappa\ \rangle$
$\mathcal{A}\ (\mathsf{Inl}(\kappa))\ v = \mathcal{A}\ \kappa\ in_1(v)$	$*$	$\langle\ v \mid \mathsf{Inl}(\kappa)\ \rangle \rightsquigarrow \langle\ in_1(v) \mid \kappa\ \rangle$
$\mathcal{A}\ (\mathsf{Inr}(\kappa))\ v = \mathcal{A}\ \kappa\ in_2(v)$	$*$	$\langle\ v \mid \mathsf{Inr}(\kappa)\ \rangle \rightsquigarrow \langle\ in_2(v) \mid \kappa\ \rangle$
$\mathcal{A}\ \mathsf{Init}\ n = n$	$*$	$\langle\ n \mid \mathsf{Init}\ \rangle \rightsquigarrow n$

$$
\begin{array}{rcl}
\mathcal{X} : X \to Val \to Env & \to & Env \\
[\mathcal{X}[\![x]\!] \mapsto v]\rho & = & \rho[x \mapsto val(v)] \\
[\mathcal{X}[\![()]\!] \mapsto unit()]\rho & = & \rho \\
[\mathcal{X}[\![(X_1,\ X_2)]\!] \mapsto pair(v_1, v_2)]\rho & = & [\mathcal{X}[\![X_1]\!] \mapsto v_1]([\mathcal{X}[\![X_2]\!] \mapsto v_2]\rho)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{Y} : Y \to Cnt \to Env & \to & Env \\
[\mathcal{Y}[\![y]\!] \mapsto \kappa]\rho & = & \rho[y \mapsto cnt(\kappa)] \\
[\mathcal{Y}[\![\{\}]\!] \mapsto \kappa]\rho & = & \rho \\
[\mathcal{Y}[\![\{Y_1,\ Y_2\}]\!] \mapsto \kappa]\rho & = & [\mathcal{Y}[\![Y_1]\!] \mapsto \mathsf{Inl}(\kappa)]([\mathcal{Y}[\![Y_2]\!] \mapsto \mathsf{Inr}(\kappa)]\rho)
\end{array}
$$

Figure 17: Defunctionalized Semantics and Corresponding Abstract Machine (CBV)

$$
\begin{array}{rcl}
n & \mapsto & n \\
unit() & \mapsto & () \\
pair(v_1, v_2) & \mapsto & (v_1,\, v_2) \\
in_1(v) & \mapsto & [v \uparrow inl] \\
in_2(v) & \mapsto & [v \uparrow inr] \\
\mathsf{Closr}(f) & \mapsto & \lceil f \rceil \\
contx(v, \kappa) & \mapsto & [\kappa \uparrow (g \downarrow v \Leftarrow \lfloor g \rfloor)]
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{Init} & \mapsto & \bullet \\
\mathsf{Ep1}(e_2, \kappa) & \mapsto & (x_1 \Rightarrow (x_1,\, e_2)) \downarrow \kappa \\
\mathsf{Ep2}(v_1, \kappa) & \mapsto & (x_2 \Rightarrow (v_1,\, x_2)) \downarrow \kappa \\
\mathsf{Epp}(f, \kappa) & \mapsto & f \downarrow \kappa \\
\mathsf{Cpp}(c) & \mapsto & c \\
\mathsf{Openr}(v, \kappa) & \mapsto & (\lceil g \rceil \Rightarrow \kappa \uparrow g) \downarrow v \\
\mathsf{Inl}(\kappa) & \mapsto & [inl \downarrow \kappa] \\
\mathsf{Inl}(\kappa) & \mapsto & [inl \downarrow \kappa]
\end{array}
$$

Figure 18: Term Transformation for CBV Derivation

The translation is mechanical: whenever λ is used in Figure 16, it is replaced with a first-order data and the corresponding clause is added to the apply function $\mathcal{A}$. For example, $\lambda v.\mathcal{F}[\![f]\!]\rho v\kappa$ appearing in $\mathcal{E}[\![e \uparrow f]\!]\rho\kappa$ of Figure 16 is translated to $\mathsf{Epp}(f, \rho, \kappa)$ where the free variables of $\lambda v.\mathcal{F}[\![f]\!]\rho v\kappa$ becomes the arguments to Epp. The corresponding clause is added to the definition of $\mathcal{A}$:

$$\mathcal{A}\ (\mathsf{Epp}(f, \rho, \kappa))\ v = \mathcal{F}[\![f]\!]\rho v\kappa$$

so that the original body of $\lambda v.\mathcal{F}[\![f]\!]\rho v\kappa$ is executed. Since all the κ's now become a first-order data, its application is replaced with a call to the apply function $\mathcal{A}$. For example, $\kappa\ n$ in $\mathcal{E}[\![n]\!]\rho\kappa$ in Figure 16 is translated to $\mathcal{A}\ \kappa\ n$.

5.2 Rewriting to Abstract Machine Style

Because the four semantic functions, $\mathcal{E}$, $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{A}$, in the defunctionalized interpreter are mutually tail-recursive, the resulting interpreter can be directly regarded as an abstract machine. The right column of Figure 17 shows the corresponding abstract machine. It is obtained by mechanically changing the notation from $\mathcal{E}[\![e]\!]\rho\kappa$, $\mathcal{C}[\![c]\!]\rho v$, $\mathcal{F}[\![f]\!]\rho v\kappa$, and $\mathcal{A}\ \kappa\ v$, to $\langle\ [\![e]\!]\rho \mid \kappa\ \rangle$, $\langle\ v \mid [\![c]\!]\rho\ \rangle$, $\langle\ v \mid [\![f]\!]\rho \mid \kappa\ \rangle$, and $\langle\ v \mid \kappa\ \rangle$, respectively.

From this abstract machine, we can derive our small-step reduction semantics by two more simple transformations: replacing the environment with substitution and rewriting defunctionalized first-order data in SLC syntax.

The abstract machine in Figure 17 uses an environment to realize substitution. The role of an environment is to defer the substitution until the substituted variable is found. (As a slogan, "an environment is a lazy substitution.") We can simply remove all the environments (and semantic brackets) by performing substitution whenever the environment is extended, namely, at the first two rules for $\mathcal{F}$ in Figure 17. With this transformation, the first two rules for $\mathcal{F}$ becomes identical to the corresponding rules ((βR_{v}) and $\overline{(\beta L_{\mathrm{v}})}$) in Figure 10. After removing the environments, all the rules for variables become useless, because they will never be used. Furthermore, the rule for a number becomes useless, because it now becomes an identity transition: $\langle\ n \mid \kappa\ \rangle \rightsquigarrow \langle\ n \mid \kappa\ \rangle$.

The second transformation is to rewrite defunctionalized first-order data in SLC syntax. Rather than writing $\mathsf{Epp}(f, \kappa)$, for example, we can instead write $f \downarrow \kappa$, because the rule for $\mathsf{Epp}(f, \kappa)$ is:

$$\langle\ v \mid \mathsf{Epp}(f, \kappa)\ \rangle \rightsquigarrow \langle\ v \mid f \mid \kappa\ \rangle$$

By writing $\mathsf{Epp}(f, \kappa)$ as $f \downarrow \kappa$, the rule becomes identical to $(\overline{pop_{\mathrm{v}}})$. Likewise, we change the notation for all the first-order data as shown in Figure 18.

With the above two transformations, all the rules marked with $*$ in Figure 17 coincide with the marked rules in the CBV small-step reduction semantics in Figure 10. The rules with $\diamond$ do not coincide with the rules in Figure 10, but we can confirm that they can all be simulated by one or more reduction steps of Figure 10. Furthermore, since $\mathcal{X}$ and $\mathcal{Y}$ are transformed to $\mathcal{R}$ and $\overline{\mathcal{L}_{\mathrm{v}}}$ (except for function patterns), we conclude that the CBV small-step semantics in Figure 10 correctly implements Filinski's SLC.

$$
\begin{array}{rcl}
Val & = & Cnt \to Ans \\
Cnt & = & Bcont + Zero + Case(Cnt \times Cnt) + Pr_1(Cnt) + Pr_2(Cnt) + \\
 & & Contx(Val \to Cnt \to Ans) + Closr(Val \times Cnt) \\
Env & = & Ide \to Val + Cnt
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E} : e \to Env \to Cnt & \to & Ans \\
\mathcal{E}[\![n]\!]\rho k & = & let\ bcont = k\ in\ n \\
\mathcal{E}[\![x]\!]\rho k & = & let\ val(\nu) = \rho\ x\ in\ \nu\ k \\
\mathcal{E}[\![(e_1,\, e_2)]\!]\rho k & = & case\ k\ of\ pr_1(k') : \mathcal{E}[\![e_1]\!]\rho k' \mid pr_2(k') : \mathcal{E}[\![e_2]\!]\rho k' \\
\mathcal{E}[\![()]\!]\rho k & = & case\ k\ of\ esac \\
\mathcal{E}[\![e \uparrow f]\!]\rho k & = & \mathcal{F}[\![f]\!]\,\rho\,(\lambda k'.\mathcal{E}[\![e]\!]\rho k')\,k \\
\mathcal{E}[\![\lceil f \rceil]\!]\rho k & = & let\ closr(\nu, k') = k\ in\ \mathcal{F}[\![f]\!]\rho\nu k'
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C} : c \to Env \to Val & \to & Ans \\
\mathcal{C}[\![y]\!]\rho\nu & = & let\ cnt(k) = \rho\ y\ in\ \nu\ k \\
\mathcal{C}[\![\{c_1,\, c_2\}]\!]\rho\nu & = & \mathcal{C}[\![c_2]\!]\,\rho\,(\lambda k_2.\mathcal{C}[\![c_1]\!]\,\rho\,(\lambda k_1.\nu\ case(k_1, k_2))) \\
\mathcal{C}[\![\{\}]\!]\rho\nu & = & \nu\ zero() \\
\mathcal{C}[\![f \downarrow c]\!]\rho\nu & = & \mathcal{C}[\![c]\!]\,\rho\,(\lambda k.\mathcal{F}[\![f]\!]\rho\nu k) \\
\mathcal{C}[\![\lfloor f \rfloor]\!]\rho\nu & = & \nu\ contx(\lambda\nu' k.\mathcal{F}[\![f]\!]\rho\nu' k)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F} : f \to Env \to Val \to Cnt & \to & Ans \\
\mathcal{F}[\![p \Rightarrow e]\!]\rho\nu k & = & \mathcal{E}[\![e]\!]\,([\mathcal{X}[\![p]\!] \mapsto \nu]\rho)\,k \\
\mathcal{F}[\![c \Leftarrow Y]\!]\rho\nu k & = & \mathcal{C}[\![c]\!]\,([\mathcal{Y}[\![Y]\!] \mapsto k]\rho)\,\nu \\
\mathcal{F}[\![\overline{e}]\!]\rho\nu k & = & \mathcal{E}[\![e]\!]\,\rho\ closr(\nu, k) \\
\mathcal{F}[\![\underline{c}]\!]\rho\nu k & = & \mathcal{C}[\![c]\!]\,\rho\,(\lambda k'.let\ contx(h) = t\ in\ h\,\nu\,k')
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{X} : p \to Val \to Env & \to & Env \\
[\mathcal{X}[\![x]\!] \mapsto \nu]\rho & = & \rho[x \mapsto val(\nu)] \\
[\mathcal{X}[\![()]\!] \mapsto \nu]\rho & = & \rho \\
[\mathcal{X}[\![(p_1,\, p_2)]\!] \mapsto \nu]\rho & = & [\mathcal{X}[\![p_1]\!] \mapsto \lambda k.\nu\ pr_1(k)]\ ([\mathcal{X}[\![p_2]\!] \mapsto \lambda k.\nu\ pr_2(k)]\rho)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{Y} : q \to Cnt \to Env & \to & Env \\
[\mathcal{Y}[\![y]\!] \mapsto k]\rho & = & \rho[y \mapsto cnt(k)] \\
[\mathcal{Y}[\![\{\}]\!] \mapsto k]\rho & = & let\ zero() = k\ in\ \rho \\
[\mathcal{Y}[\![\{q_1,\, q_2\}]\!] \mapsto k]\rho & = & let\ case(c_1, c_2) = k\ in\ [\mathcal{Y}[\![q_1]\!] \mapsto c_1]\ ([\mathcal{Y}[\![q_2]\!] \mapsto c_2]\rho)
\end{array}
$$

Figure 19: Filinski's Denotational Semantics for CBN SLC

6 Functional Correspondence for CBN SLC

Filinski also presented denotational semantics for CBN SLC. Using the same method presented in the previous section, we can obtain the CBN small-step semantics. Although denotational semantics for CBV SLC and CBN SLC look quite different (because they are encoded in a standard non-symmetric lambda-calculus), the present results confirm that they are actually dual to each other. In this section, we show that the CBN small-step semantics of SLC presented in Section 4 corresponds to Filinski's original definition of CBN SLC given as a denotational semantics.

Figure 19 shows denotational semantics of Filinski's CBN SLC. In the semantics, k represents *covalues* (the result of evaluation), and ν represents *semantic* values (that is *not* a result). Filinski introduces a domain for each covalue type (e.g., $Case$) together with a constructor for it (in lower case letter, e.g., $case$).

Defunctionalized Semantics	Corresponding Abstract Machine
$\mathcal{E} : e \to Env \to Cnt \to Ans$	The rules coming from $\mathcal{E}$
$\mathcal{E}[\![n]\!]\rho\ bcont = n$	$*$ $\langle\ [\![n]\!]\rho \mid bcont\ \rangle \rightsquigarrow n$
$\mathcal{E}[\![x]\!]\rho k = let\ val(\nu) = \rho\ x\ in\ \mathcal{A}\ \nu\ k$	$\langle\ [\![x]\!]\rho \mid k\ \rangle \rightsquigarrow let\ val(\nu) = \rho\ x\ in \langle\ \nu \mid k\ \rangle$
$\mathcal{E}[\![(e_1,\ e_2)]\!]\rho\ pr_1(k) = \mathcal{E}[\![e_1]\!]\rho k$	$*$ $\langle\ [\![(e_1,\ e_2)]\!]\rho \mid pr_1(k)\ \rangle \rightsquigarrow \langle\ [\![e_1]\!]\rho \mid k\ \rangle$
$\mathcal{E}[\![(e_1,\ e_2)]\!]\rho\ pr_2(k) = \mathcal{E}[\![e_2]\!]\rho k$	$*$ $\langle\ [\![(e_1,\ e_2)]\!]\rho \mid pr_2(k)\ \rangle \rightsquigarrow \langle\ [\![e_2]\!]\rho \mid k\ \rangle$
$\mathcal{E}[\![()]\!]\rho k = case\ k\ of\ esac$	$\langle\ [\![()]\!]\rho \mid k\ \rangle \rightsquigarrow case\ k\ of\ esac$
$\mathcal{E}[\![e \uparrow f]\!]\rho k = \mathcal{F}[\![f]\!]\,\rho\,(\mathsf{Epp}(e,\rho))\,k$	$*$ $\langle\ [\![e \uparrow f]\!]\rho \mid k\ \rangle \rightsquigarrow \langle\ \mathsf{Epp}(e,\rho) \mid [\![f]\!]\rho \mid k\ \rangle$
$\mathcal{E}[\![\lceil f \rceil]\!]\rho\ closr(a,c) = \mathcal{F}[\![f]\!]\rho ac$	$*$ $\langle\ [\![\lceil f \rceil]\!]\rho \mid closr(\nu,k)\ \rangle \rightsquigarrow \langle\ \nu \mid [\![f]\!]\rho \mid k\ \rangle$
$\mathcal{C} : c \to Env \to Val \to Ans$	The rules coming from $\mathcal{C}$
$\mathcal{C}[\![y]\!]\rho\nu = let\ cnt(k) = \rho\ y\ in\ \mathcal{A}\ \nu\ k$	$\langle\ \nu \mid [\![y]\!]\rho\ \rangle \rightsquigarrow let\ cnt(k) = \rho\ y\ in \langle\ \nu \mid k\ \rangle$
$\mathcal{C}[\![\{c_1,\ c_2\}]\!]\rho\nu = \mathcal{C}[\![c_2]\!]\,\rho\,(\mathsf{Cp2}(c_1,\rho,\nu))$	$\diamond$ $\langle\ \nu \mid [\![\{c_1,\ c_2\}]\!]\rho\ \rangle \rightsquigarrow \langle\ \mathsf{Cp2}(c_1,\rho,\nu) \mid [\![c_2]\!]\rho\ \rangle$
$\mathcal{C}[\![\{\}]\!]\rho\nu = \mathcal{A}\ \nu\ zero()$	$\langle\ \nu \mid [\![\{\}]\!]\rho\ \rangle \rightsquigarrow \langle\ \nu \mid zero()\ \rangle$
$\mathcal{C}[\![f \downarrow c]\!]\rho\nu = \mathcal{C}[\![c]\!]\,\rho\,(\mathsf{Cpp}(f,\rho,\nu))$	$\diamond$ $\langle\ \nu \mid [\![f \downarrow c]\!]\rho\ \rangle \rightsquigarrow \langle\ \mathsf{Cpp}(f,\rho,\nu) \mid [\![c]\!]\rho\ \rangle$
$\mathcal{C}[\![\lfloor f \rfloor]\!]\rho\nu = \mathcal{A}\ \nu\ \mathsf{Contx}(f,\rho)$	$\langle\ \nu \mid [\![\lfloor f \rfloor]\!]\rho\ \rangle \rightsquigarrow \langle\ \nu \mid \mathsf{Contx}(f,\rho)\ \rangle$
$\mathcal{F} : f \to Env \to Val \to Cnt \to Ans$	The rules coming from $\mathcal{F}$
$\mathcal{F}[\![p \Rightarrow e]\!]\rho\nu k = \mathcal{E}[\![e]\!]\,([\mathcal{X}[\![p]\!] \mapsto \nu]\rho)\,k$	$*$ $\langle\ \nu \mid [\![p \Rightarrow e]\!]\rho \mid k\ \rangle \rightsquigarrow \langle\ [\![e]\!]\,([\mathcal{X}[\![p]\!] \mapsto \nu]\rho) \mid k\ \rangle$
$\mathcal{F}[\![c \Leftarrow q]\!]\rho\nu k = \mathcal{C}[\![c]\!]\,([\mathcal{Y}[\![q]\!] \mapsto k]\rho)\,\nu$	$*$ $\langle\ \nu \mid [\![c \Leftarrow q]\!]\rho \mid k\ \rangle \rightsquigarrow \langle\ \nu \mid [\![c]\!]\,([\mathcal{Y}[\![q]\!] \mapsto k]\rho)\ \rangle$
$\mathcal{F}[\![\overline{e}]\!]\rho\nu k = \mathcal{E}[\![e]\!]\,\rho\,closr(\nu,k)$	$*$ $\langle\ \nu \mid [\![\overline{e}]\!]\rho \mid k\ \rangle \rightsquigarrow \langle\ [\![e]\!]\rho \mid contx(\nu,k)\ \rangle$
$\mathcal{F}[\![\underline{c}]\!]\rho\nu k = \mathcal{C}[\![c]\!]\,\rho\,(\mathsf{Plug}(\nu,k))$	$*$ $\langle\ \nu \mid [\![\underline{c}]\!]\rho \mid k\ \rangle \rightsquigarrow \langle\ \mathsf{Plug}(\nu,k) \mid [\![c]\!]\rho\ \rangle$
$\mathcal{A} : Val \to Cnt \to Ans$	The rules coming from $\mathcal{A}$
$\mathcal{A}\ (\mathsf{Epp}(e,\rho))\ k = \mathcal{E}[\![e]\!]\rho k$	$\langle\ \mathsf{Epp}(e,\rho) \mid k\ \rangle \rightsquigarrow \langle\ [\![e]\!]\rho \mid k\ \rangle$
$\mathcal{A}\ (\mathsf{Cp2}(c_1,\rho,\nu))\ k_2 = \mathcal{C}[\![c_1]\!]\,\rho\,(\mathsf{Cp1}(k_2,\nu))$	$\diamond$ $\langle\ \mathsf{Cp1}(c_2,\rho,\nu) \mid k_1\ \rangle \rightsquigarrow \langle\ \mathsf{Cp2}(k_1,\nu) \mid [\![c_2]\!]\rho\ \rangle$
$\mathcal{A}\ (\mathsf{Cp1}(k_2,\nu))\ k_1 = \mathcal{A}\ \nu\ case(k_1,k_2)$	$\diamond$ $\langle\ \mathsf{Cp2}(k_1,\nu) \mid k_2\ \rangle \rightsquigarrow \langle\ \nu \mid case(k_1,k_2)\ \rangle$
$\mathcal{A}\ (\mathsf{Cpp}(f,\rho,\nu))\ k = \mathcal{F}[\![f]\!]\rho\nu k$	$*$ $\langle\ \mathsf{Cpp}(f,\rho,\nu) \mid k\ \rangle \rightsquigarrow \langle\ \nu \mid [\![f]\!]\rho \mid k\ \rangle$
$\mathcal{A}\ (\mathsf{Plug}(\nu,k))\ \mathsf{Contx}(f,\rho) = \mathcal{F}[\![f]\!]\rho k\nu$	$\diamond$ $\langle\ \mathsf{Plug}(\nu,k) \mid \mathsf{Contx}(f,\rho)\ \rangle \rightsquigarrow \langle\ \nu \mid [\![f]\!]\rho \mid k\ \rangle$
$\mathcal{A}\ (\mathsf{Fst}(\nu))\ k = \mathcal{A}\ \nu\ pr_1(k)$	$*$ $\langle\ \mathsf{Fst}(\nu) \mid k\ \rangle \rightsquigarrow \langle\ \nu \mid pr_1(k)\ \rangle$
$\mathcal{A}\ (\mathsf{Snd}(\nu))\ k = \mathcal{A}\ \nu\ pr_2(k)$	$*$ $\langle\ \mathsf{Snd}(\nu) \mid k\ \rangle \rightsquigarrow \langle\ \nu \mid pr_2(k)\ \rangle$

$$
\begin{array}{rcl}
\mathcal{X} : p \to Val \to Env & \to & Env \\
[\mathcal{X}[\![x]\!] \mapsto \nu]\rho & = & \rho[x \mapsto val(\nu)] \\
[\mathcal{X}[\![()]\!] \mapsto \nu]\rho & = & \rho \\
[\mathcal{X}[\![(p_1,\ p_2)]\!] \mapsto \nu]\rho & = & [\mathcal{X}[\![p_1]\!] \mapsto \mathsf{Fst}(\nu)]([\mathcal{X}[\![p_2]\!] \mapsto \mathsf{Snd}(\nu)]\rho)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{Y} : q \to Cnt \to Env & \to & Env \\
[\mathcal{Y}[\![y]\!] \mapsto k]\rho & = & \rho[y \mapsto cnt(\nu)] \\
[\mathcal{Y}[\![\{\}]\!] \mapsto \{\}]\rho & = & \rho \\
[\mathcal{Y}[\![\{q_1,\ q_2\}]\!] \mapsto case(k_1,k_2)]\rho & = & [\mathcal{Y}[\![q_1]\!] \mapsto k_1]\ ([\mathcal{Y}[\![q_2]\!] \mapsto k_2]\rho)
\end{array}
$$

Figure 20: Defunctionalized Semantics and Corresponding Abstract Machine (CBN)

$$
\begin{array}{rcl@{\qquad}rcl}
n & \mapsto & n & \mathit{bcont} & \mapsto & \bullet \\
\mathsf{Epp}(e) & \mapsto & e & \mathit{zero}() & \mapsto & \{\} \\
\mathsf{Cp1}(k_2, \nu) & \mapsto & \nu \uparrow (\{y_1,\, k_2\} \Leftarrow y_1) & \mathit{pair}(k_1, k_2) & \mapsto & \{k_1,\, k_2\} \\
\mathsf{Cp2}(c_1, \nu) & \mapsto & \nu \uparrow (\{c_1,\, y_2\} \Leftarrow y_2) & \mathit{pr}_1(k) & \mapsto & [\mathit{fst} \downarrow k] \\
\mathsf{Cpp}(f, \nu) & \mapsto & \nu \uparrow f & \mathit{pr}_2(k) & \mapsto & [\mathit{snd} \downarrow k] \\
\mathsf{Plug}(\nu, k) & \mapsto & \nu \uparrow (g \downarrow k \Leftarrow \lfloor g \rfloor) & \mathsf{Contx}(f) & \mapsto & \lfloor f \rfloor \\
\mathsf{Fst}(\nu) & \mapsto & [\nu \uparrow \mathit{fst}] & \mathit{closr}(\nu, k) & \mapsto & [(\lceil g \rceil \Rightarrow k \uparrow g) \downarrow \nu] \\
\mathsf{Snd}(\nu) & \mapsto & [\nu \uparrow \mathit{snd}] & & &
\end{array}
$$

Figure 21: Term Transformations for CBN Derivation

6.1 Defunctionalization

We defunctionalize two kinds of higher-order functions in Figure 19: values (ν of type Val) and contexts (the argument of *contx*). The result of defunctionalization is found in the upper left of Figure 20. The translation is mechanical and dual to the CBV derivation in Section 5.1. For example, though we have defunctionalized a semantic continuation for $e \uparrow f$ in Section 5.1, here, we need to defunctionalize a semantic value for $f \downarrow c$. We translate the semantic value $\lambda k.\mathcal{F}[\![F]\!]\rho\nu k$ to $\mathsf{Cpp}(f, \rho, \nu)$. The corresponding clause is added to the definition of $\mathcal{A}$:

$$\mathcal{A}\ (\mathsf{Cpp}(f, \rho, \nu))\ k = \mathcal{F}[\![f]\!]\rho\nu k$$

6.2 Rewriting to Abstract Machine Style

Because the four semantic functions, $\mathcal{E}$, $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{A}$, in the defunctionalized interpreter are mutually tail-recursive, the resulting interpreter can be directly regarded as an abstract machine. The right column of Figure 20 shows the corresponding abstract machine. It is obtained by mechanically changing the notation from $\mathcal{E}[\![e]\!]\rho k$, $\mathcal{C}[\![c]\!]\rho\nu$, $\mathcal{F}[\![f]\!]\rho\nu k$, and $\mathcal{A}\ \nu\ k$, to $\langle\ [\![e]\!]\rho \mid k\ \rangle$, $\langle\ \nu \mid [\![c]\!]\rho\ \rangle$, $\langle\ \nu \mid [\![f]\!]\rho \mid k\ \rangle$, and $\langle\ \nu \mid k\ \rangle$, respectively.

From this abstract machine, we can derive our small-step reduction semantics by two more simple transformations: replacing the environment with substitution and rewriting defunctionalized first-order data in SLC syntax. First, we simply remove all the environments (and semantic brackets) by performing substitution whenever the environment is extended. After removing the environments, all the rules for variables become useless, because they will never be used. Secondly, we change the notation for all the first-order data as shown in Figure 21.

With the above two transformations, all the rules marked with $*$ in Figure 20 coincide with the marked rules in the CBN small-step reduction semantics in Figure 14. The rules with $\diamond$ do not coincide with the rules in Figure 14, but we can confirm that they can all be simulated by one or more reduction steps of Figure 14. Furthermore, since $\mathcal{X}$ and $\mathcal{Y}$ are transformed to $\mathcal{L}_{\mathrm{n}}$ and $\overline{\mathcal{R}}$ (except for function patterns), we conclude that the CBN small-step semantics in Figure 14 correctly implements Filinski's SLC.

7 Future Direction

In this paper, we have presented a small-step reduction semantics for Filinski's symmetric lambda calculus (SLC). Now that we obtained a small-step reduction semantics for SLC, we can express and compare various other calculi in terms of SLC. Our preliminary work shows that we can naturally encode Felleisen's $\mathcal{C}$ operator as well as $\lambda\mu$-calculus into SLC [11]. We are also interested in if delimited continuations can be introduced into the SLC framework.

References

[1] Ager, M. S., D. Biernacki, O. Danvy, and J. Midtgaard "A functional correspondence between evaluators and abstract machines," *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pp. 8–19 (September 2003).

[2] Barbanera, F., and S. Berardi "A Symmetric Lambda Calculus for Classical Program Extraction," *Information and Computation*, Vol. 125, No. 2, pp. 103–117, Academic Press (March 1996).

[3] Curien, P.-L., and H. Herbelin "The Duality of Computation," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pp. 233–243 (September 2000).

[4] Danvy, O. "Defunctionalized interpreters for programming languages," *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pp. 131–142 (September 2008).

[5] Felleisen, M., and R. Hieb "The Revised Report on the Syntactic Theories of Sequential Control and State," *Theoretical Computer Science*, Vol. 103, No. 2, pp. 235–271 (September 1992).

[6] Filinski, A. "Declarative Continuations and Categorical Duality," Master's thesis, DIKU Report 89/11, University of Copenhagen (August 1989).

[7] Griffin, T. "A Formulae-as-Types Notion of Control," *Conference Record of the 17th ACM Symposium on Principles of Programming Languages*, pp. 47–58 (January 1990).

[8] Parigot, M. "$\lambda\mu$-calculus: An Algorithmic Interpretation of Classical Natural Deduction," In A. Voronkov, editor, *Logic Programming and Automated Reasoning (LNCS 624)*, pp. 190–201 (July 1992).

[9] Pierce, B. C. *Types and Programming Languages*, Cambridge: MIT Press (2002).

[10] Reynolds, J. C. "Definitional Interpreters for Higher-Order Programming Languages," *Proceedings of the ACM National Conference*, Vol. 2, pp. 717–740, (August 1972), reprinted in *Higher-Order and Symbolic Computation*, Vol. 11, No. 4, pp. 363–397, Kluwer Academic Publishers (December 1998).

[11] Sakaue, S., and K. Asai "The Foundations of Symmetric Lambda Calculus," *Computer Software*, Vol. 26, No. 2, pp. 3-17 (May 2009). (in Japanese).

[12] Selinger, P. "Control Categories and Duality: on the Categorical Semantics of the Lambda-Mu Calculus," *Mathematical Structures in Computer Science*, Vol. 11, No. 2, pp. 207-260 (March 2001).

[13] Tzevelekos, N. "Investigations on the Dual Calculus," *Theoretical Computer Science*, Vol. 360, Nos. 1–3, pp. 289–326 (August 2006).

[14] Wadler, P. "Call-by-value is dual to call-by-name," *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pp. 189–201 (August 2003).

Functional Derivation of Small-Step Semantics for Symmetric Lambda Calculus (SLC)

Kenichi Asai
joint work with Yayoi Ueda

Ochanomizu University (Tokyo, Japan)

January 10, 2011

What is Symmetric Lambda Calculus (SLC)?

Symmetric extension of λ-calculus proposed by Filinski in 1989

- Two kinds of duality:

Expressions	produce data.	CBV	expressions first
Functions	transform data.		
Continuations	consume data.	CBN	continuations first

$$\mathbf{1} \longrightarrow A \longrightarrow B \longrightarrow \mathbf{0}$$

- Given: denotational semantics, translation to SCL.

Good calculus for dealing with continuations.

SLC: Syntax

A configuration is either $\langle\, e \mid c \,\rangle$ or $\langle\, e \mid f \mid c \,\rangle$, where:

$$\begin{array}{rrcl}
\text{expression} & e & ::= & \circ_T \mid x \mid (e, e) \mid \lceil f \rceil \mid e \uparrow f \\
\text{function} & f & ::= & g \mid x \Rightarrow e \mid (x_1, x_2) \Rightarrow e \mid \lceil g \rceil \Rightarrow e \mid \overline{e} \\
 & & & h \mid c \Leftarrow y \mid c \Leftarrow (y_1, y_2) \mid c \Leftarrow \lfloor h \rfloor \mid \underline{c} \\
\text{continuation} & c & ::= & \bullet_T \mid y \mid (c, c) \mid \lfloor f \rfloor \mid f \downarrow c
\end{array}$$

Example:

$$\begin{array}{rl}
 & \langle\, \mathbf{1} \uparrow x_1 \Rightarrow x_1 + \mathbf{2} \uparrow x_2 \Rightarrow x_2 * \mathbf{4} \mid \bullet_{\mathsf{int}} \,\rangle \\
\leadsto & \langle\, \mathbf{1} \uparrow x_1 \Rightarrow x_1 + \mathbf{2} \mid x_2 \Rightarrow x_2 * \mathbf{4} \mid \bullet_{\mathsf{int}} \,\rangle \\
\leadsto & \langle\, \mathbf{1} \uparrow x_1 \Rightarrow x_1 + \mathbf{2} \mid x_2 \Rightarrow x_2 * \mathbf{4} \downarrow \bullet_{\mathsf{int}} \,\rangle \\
\leadsto & \langle\, \mathbf{1} \mid x_1 \Rightarrow x_1 + \mathbf{2} \mid x_2 \Rightarrow x_2 * \mathbf{4} \downarrow \bullet_{\mathsf{int}} \,\rangle \\
\leadsto & \langle\, \mathbf{1} + \mathbf{2} \mid x_2 \Rightarrow x_2 * \mathbf{4} \downarrow \bullet_{\mathsf{int}} \,\rangle \\
\leadsto & \langle\, \mathbf{3} \mid x_2 \Rightarrow x_2 * \mathbf{4} \downarrow \bullet_{\mathsf{int}} \,\rangle \\
\leadsto & \langle\, \mathbf{3} \mid x_2 \Rightarrow x_2 * \mathbf{4} \mid \bullet_{\mathsf{int}} \,\rangle \\
\leadsto^* & \langle\, \mathbf{12} \mid \bullet_{\mathsf{int}} \,\rangle
\end{array}$$

Reduction rules (non-deterministic)

$$
\begin{array}{rrcl}
(pop) & \langle\, e \uparrow f \mid c \,\rangle & \rightsquigarrow & \langle\, e \mid f \mid c \,\rangle \\
(push) & \langle\, e \mid f \mid c \,\rangle & \rightsquigarrow & \langle\, e \mid f \downarrow c \,\rangle \\
(left) & \langle\, (e_1,\, e_2) \mid c \,\rangle & \rightsquigarrow & \langle\, e_1 \mid x_1 \Rightarrow (x_1,\, e_2) \mid c \,\rangle \\
(right) & \langle\, (e_1,\, e_2) \mid c \,\rangle & \rightsquigarrow & \langle\, e_2 \mid x_2 \Rightarrow (e_1,\, x_2) \mid c \,\rangle \\
(exc) & \langle\, e \mid \overline{e'} \mid c \,\rangle & \rightsquigarrow & \langle\, e' \mid \lceil g \rceil \Rightarrow e \uparrow g \mid c \,\rangle \\
(\beta) & \langle\, e \mid x \Rightarrow e' \mid c \,\rangle & \rightsquigarrow & \langle\, e'[e/x] \mid c \,\rangle \\
(\beta_p) & \langle\, (e_1,\, e_2) \mid (x_1,\, x_2) \Rightarrow e' \mid c \,\rangle & \rightsquigarrow & \langle\, e'[e_1/x_1,\, e_2/x_2] \mid c \,\rangle \\
(\beta_f) & \langle\, \lceil f \rceil \mid \lceil g \rceil \Rightarrow e' \mid c \,\rangle & \rightsquigarrow & \langle\, e'[f/g] \mid c \,\rangle \\
(\overline{\beta_f}) & \langle\, e \mid c' \Leftarrow \lfloor h \rfloor \mid \lfloor f \rfloor \,\rangle & \rightsquigarrow & \langle\, e \mid c'[f/h] \,\rangle \\
(\overline{\beta_p}) & \langle\, e \mid c' \Leftarrow (y_1,\, y_2) \mid (c_1,\, c_2) \,\rangle & \rightsquigarrow & \langle\, e \mid c'[c_1/y_1,\, c_2/y_2] \,\rangle \\
(\overline{\beta}) & \langle\, e \mid c' \Leftarrow y \mid c \,\rangle & \rightsquigarrow & \langle\, e \mid c'[c/y] \,\rangle \\
(\overline{exc}) & \langle\, e \mid \underline{c'} \mid c \,\rangle & \rightsquigarrow & \langle\, e \mid h \downarrow c \Leftarrow \lfloor h \rfloor \mid c' \,\rangle \\
(\overline{right}) & \langle\, e \mid (c_1,\, c_2) \,\rangle & \rightsquigarrow & \langle\, e \mid (c_1,\, y_2) \Leftarrow y_2 \mid c_2 \,\rangle \\
(\overline{left}) & \langle\, e \mid (c_1,\, c_2) \,\rangle & \rightsquigarrow & \langle\, e \mid (y_1,\, c_2) \Leftarrow y_1 \mid c_1 \,\rangle \\
(\overline{push}) & \langle\, e \mid f \mid c \,\rangle & \rightsquigarrow & \langle\, e \uparrow f \mid c \,\rangle \\
(\overline{pop}) & \langle\, e \mid f \downarrow c \,\rangle & \rightsquigarrow & \langle\, e \mid f \mid c \,\rangle
\end{array}
$$

Denotational Semantics (CBV)

$$
\begin{array}{rcl}
\mathcal{E}[\![x]\!]\rho\kappa & = & let\ val(v) = \rho\ x\ in\ \kappa\ v \\
\mathcal{E}[\![(E_1, E_2)]\!]\rho\kappa & = & \mathcal{E}[\![E_1]\!]\,\rho\,(\lambda v_1.\mathcal{E}[\![E_2]\!]\,\rho\,(\lambda v_2.\kappa\ (pair(v_1, v_2)))) \\
\mathcal{E}[\![F \uparrow E]\!]\rho\kappa & = & \mathcal{E}[\![E]\!]\,\rho\,(\lambda v.\mathcal{F}[\![F]\!]\rho v\kappa) \\
\mathcal{E}[\![\lceil F \rceil]\!]\rho\kappa & = & \kappa\ (closr(\lambda vc.\mathcal{F}[\![F]\!]\rho vc))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}[\![y]\!]\rho v & = & let\ cnt(\kappa) = \rho\ y\ in\ \kappa\ v \\
\mathcal{C}[\![\{C_1, C_2\}]\!]\rho v & = & case\ v\ of\ in_1(t)\ \ \mathcal{C}[\![C_1]\!]\rho t\ \mid\ in_2(t)\ \ \mathcal{C}[\![C_2]\!]\rho t \\
\mathcal{C}[\![F \downarrow C]\!]\rho v & = & \mathcal{F}[\![F]\!]\,\rho\,v\,(\lambda t.\mathcal{C}[\![C]\!]\rho t) \\
\mathcal{C}[\![\lfloor F \rfloor]\!]\rho v & = & let\ contx(a, c) = v\ in\ \mathcal{F}[\![F]\!]\rho ac
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F}[\![X \Rightarrow E]\!]\rho v\kappa & = & \mathcal{E}[\![E]\!]\,([\mathcal{X}[\![X]\!] \mapsto v]\rho)\,\kappa \\
\mathcal{F}[\![Y \Leftarrow C]\!]\rho v\kappa & = & \mathcal{C}[\![C]\!]\,([\mathcal{Y}[\![Y]\!] \mapsto \kappa]\rho)\,v \\
\mathcal{F}[\![\overline{E}]\!]\rho v\kappa & = & \mathcal{E}[\![E]\!]\,\rho\,(\lambda t.let\ closr(f) = t\ in\ f\ v\ \kappa) \\
\mathcal{F}[\![\underline{C}]\!]\rho v\kappa & = & \mathcal{C}[\![C]\!]\,\rho\,(contx(v, \kappa))
\end{array}
$$

Two Semantics

- Denotational Semantics
- Small-Step Semantics

Are they the same? — Yes (...well, almost).

We can connect them via
functional correspondence [Danvy].

Functional Correspondence

Interpreter (big-step semantics)

⇓ CPS transformation

CPS interpreter

⇓ Defunctionalization

Abstract machine (small-step semantics)

Functional Correspondence of SLC

CBV DS CBN DS

⇓ ⇓

CBV SSS → ND SSS ← CBN SSS

Subtle Point: CBN pair destruction

In Haskell:

```
f  (x, y) = 0    //   refutable pattern
g ~(x, y) = 0    // irrefutable pattern

*Main> undefined
*** Exception: Prelude.undefined
*Main> f undefined
*** Exception: Prelude.undefined
*Main> g undefined
0
*Main>
```

Subtle Point: CBN pair destruction

$$\langle\, e \mid (x_1, x_2) \Rightarrow e' \mid c \,\rangle$$

- Should we evaluate e to a pair?

$$\begin{array}{rcl} \langle\, e \mid (x_1, x_2) \Rightarrow e' \mid c \,\rangle & \rightsquigarrow & \langle\, e \mid [(x_1, x_2) \Rightarrow e' \downarrow c] \,\rangle \\ \langle\, (e_1, e_2) \mid [(x_1, x_2) \Rightarrow e' \downarrow c] \,\rangle & \rightsquigarrow & \langle\, e'[e_i/x_i] \mid c \,\rangle \end{array}$$

- Should we bind x_i to $[e \uparrow \pi_i]$?

$$\begin{array}{rcl} \langle\, e \mid (x_1, x_2) \Rightarrow e' \mid c \,\rangle & \rightsquigarrow & \langle\, e'[[e \uparrow \pi_i]/x_i] \mid c \,\rangle \\ \langle\, [e \uparrow \pi_i] \mid c \,\rangle & \rightsquigarrow & \langle\, e \mid [\pi_i \downarrow c] \,\rangle \end{array}$$

Summary

- Small-step semantics for SLC is shown.
- Its CBV/CBN variants coincide with the original CBV/CBN denotational semantics.
- The original SLC uses irrefutable patterns.

- Relationship to other calculi?
  ($\lambda\mu\tilde{\mu}$-calculus, Dual calculus, etc.)
- Delimited continuations?

Report on a Self-applicable Online Partial Evaluator for a Recursive Flowchart Language

Robert Glück

DIKU, Department of Computer Science, University of Copenhagen,
DK-2100 Copenhagen, Denmark

Summary

We describe the design and implementation of a *self-applicable online partial evaluator* for a flowchart language with recursive calls. Self-application of the online partial evaluator converts interpreters into compilers and produces an online compiler generator, all of which are as efficient as those known from the offline partial evaluation literature (*e.g.*, [1, 6, 8–11]). This result is remarkable because it is assumed that online techniques *unavoidably* lead to *inefficient* and *overgeneralized* program generators [8]. The online partial evaluator does not require partial evaluation techniques that are stronger than those already known. Instead it requires a reorganization of the algorithm to achieve successful self-application according to all three Futamura projections [4].

Offline partial evaluation was invented in 1984 [9] specifically to solve the problem of self-application and to perform all three Futamura projections. Even though the first partial evaluators were all online and in spite of a number of attempts during the last decades, the self-application of an online partial evaluator has been an open question. The purpose of this investigation is not to determine which line of partial evaluation is better, but to show how the problem can be solved. The solution sought is a specialization algorithm that is (i) *complete*, (ii) *purely online*, that is, without binding-time analysis prior to specialization, and (iii) *fully self-applicable* in that all three Futamura projections yield efficient residual programs (*e.g.*, compilers, compiler generators). The online partial evaluator that we developed satisfies all three criteria.

The offline partial evaluator `mix` for a flowchart language described by Gomard and Jones [6] is well suited as a reference for our online partial evaluator because `mix` does not follow the local binding-time annotations of a subject program, but instead bases its specialization decisions on a fixed global division of the program variables into static and dynamic, this division being precomputed by a monovariant binding-time analysis. An important advantage of using a flowchart language is that partial evaluation for flowchart languages is very well documented in the literature (*e.g.*, [2, 3, 6–8]), which should make our results easily accessible and comparable. Flowchart languages also represent the core of many realistic imperative languages.

Our online partial evaluator presented differs from `mix` in two important ways:

(1) the division of program variables is maintained as an *updatable set of variable names* at specialization time, and

(2) *recursive polyvariant specialization* is performed instead of the traditional iterative version using an accumulating parameter (the pending list) [5].

Although the design of the self-applicable online partial evaluator is based on a number of known techniques, by combining them in a new way this synergetic effect can be produced.

Successful self-application of the online partial evaluator according to all three Futamura projections is demonstrated, amongst other things, by converting an interpreter for Turing machines into a compiler, an Ackermann program into a generating extension, and the partial evaluator itself into an online compiler generator. Self-application of the online partial evaluator can yield generating extensions that are more optimizing than those produced by the offline partial evaluator `mix`. The generating extension of the Ackermann program for example can precompute the function, thereby producing more optimized residual programs.

We believe that the online partial evaluator for the flowchart language presented here provides the clearest solution to date.

References

1. A. Bondorf, O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.
2. M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21(5):473–484, 1984.
3. N. H. Christensen, R. Glück. Offline partial evaluation can be as accurate as online partial evaluation. *ACM TOPLAS*, 26(1):191–220, 2004.
4. Y. Futamura. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
5. R. Glück. An experiment with the fourth Futamura projection. In A. Pnueli, I. Virbitskaite, A. Voronkov (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 5947, 135–150. Springer-Verlag, 2010.
6. C. K. Gomard, N. D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12(3):123–144, 1991.
7. J. Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In J. Hatcliff, T. Æ. Mogensen, P. Thiemann (eds.), *Partial Evaluation. Practice and Theory*, LNCS 1706, 20–82. Springer-Verlag, 1999.
8. N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
9. N. D. Jones, P. Sestoft, H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud (ed.), *Rewriting Techniques and Applications. Proceedings*, LNCS 202, 124–140. Springer-Verlag, 1985.
10. T. Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 325–347. North-Holland, 1988.
11. S. A. Romanenko. The specializer Unmix, 1990. Program and documentation available from `ftp://ftp.diku.dk/pub/diku/dists/jones-book/Romanenko/`.

A Universal Reversible Turing Machine

(joint work w/ Robert Glück)

Holger Bock Axelsen
funkstar@diku.dk

4th DIKU-IST Joint Workshop on Foundations of Software, Tokyo, Japan

January 10, 2011

Overview

- Reversible Turing machines
- Universality for RTMs
- A first principles URTM

Turing machines

Definition (Turing machine)

A TM T is a tuple $(Q, \Sigma, \delta, b, q_s, q_f)$ where Q is a finite set of states, Σ is a finite set of tape symbols, $b \in \Sigma$ is the blank symbol,

$$\delta \subseteq (Q \times [(\Sigma \times \Sigma) \cup \{\leftarrow, \downarrow, \rightarrow\}] \times Q)$$

is a partial relation defining the transition relation, $q_s \in Q$ is the starting state, and $q_f \in Q$ is the final state. There must be *no* transitions leading out of q_f nor into q_s.

Triple format for transition rules

$$\delta \subseteq (Q \times [(\Sigma \times \Sigma) \cup \{\leftarrow, \downarrow, \rightarrow\}] \times Q)$$

The form of a triple format rule in δ is either:

- a symbol rule $(q, (s, s'), q')$ where $s, s' \in \Sigma$, or
- a move rule (q, d, q') where $d \in \{\leftarrow, \downarrow, \rightarrow\}$.

(Triples can be converted to the usual quintuples, and vice versa. We use it for convenience.)

Reversible Turing machines (RTMs)

Intuition: RTMs are those where each configuration has a *unique* successor and predecessor configuration.

Definition (Reversible Turing machine)

A TM T is *reversible* iff it is (locally) forward and backward deterministic.

Local forward/backward determinism

Definition (Local forward determinism)

A TM T is *local forward deterministic* iff for any distinct pair of triples $(q_1, a_1, q_1') \in \delta$ and $(q_2, a_2, q_2') \in \delta$, if $q_1 = q_2$ then $a_1 = (s_1, s_1')$ and $a_2 = (s_2, s_2')$, and $s_1 \neq s_2$.

Definition (Local backward determinism)

A TM T is *local backward deterministic* iff for any distinct pair of triples $(q_1, a_1, q_1') \in \delta$ and $(q_2, a_2, q_2') \in \delta$, if $q_1' = q_2'$ then $a_1 = (s_1, s_1')$ and $a_2 = (s_2, s_2')$, and $s_1' \neq s_2'$.

Local backward determinism: Examples

- $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{q}, (\mathtt{a}, \mathtt{c}), \mathtt{p})$ respects bwd determinism.
- $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{r}, (\mathtt{c}, \mathtt{b}), \mathtt{p})$ breaks bwd determinism.
- $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{r}, \rightarrow, \mathtt{p})$ breaks bwd determinism.

RTM computability

Some important results:

- RTMs compute *injective functions*, only.
- *All* injective computable function are computable with RTMs.
- 1-tape, 3-symbol RTMs are sufficient.
- RTMs can be easily inverted...

Classical universality

A universal TM U is defined as a *self-interpreter* for Turing machines:

$$[\![U]\!](\ulcorner T \urcorner, x) = [\![T]\!](x) \ .$$

Here, $\ulcorner T \urcorner \in \Sigma^*$ is a Gödel number representing some TM T.

Problem: Does *not* work for RTMs - $[\![U]\!]$ is non-injective.

RTM-universality

An RTM-universal TM U_R is defined by

$$[\![U_R]\!](\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, [\![T]\!](x)) \ .$$

where $\ulcorner T \urcorner \in \Sigma^*$ is a Gödel number representing some RTM T.

$[\![U_R]\!]$ is injective and computable $\Rightarrow$ computable by some RTM.

Why a first-principles approach?

Previous approaches (Bennett, Morita) rely on *reversible simulations* ("reversibilization") of irreversible machines. Asymptotically *very* costly: As much space as time!

The URTM we give has better complexity: (Program dependent) constant factor slowdown, same space as interpreted program. Theoretical basis for the robustness of reversible models.

URTM overview

Scope:

- Interprets 1-tape, 3-symbol RTMs ($T = \{Q, \{b, 0, 1\}, \delta, b, q_s, q_f\}$).

Structure:

- Work tape: Identical to T's tape.
- Program tape: Contains the program $\ulcorner T \urcorner$.
- State tape: Encoding of T's internal state, q_c.

Program encoding $\ulcorner T \urcorner$

A program is a string $\ulcorner T \urcorner$ over $\Sigma = \{b, 0, 1, \mathtt{B}, \mathtt{S}, \mathtt{M}, \#\}$. $\ulcorner T \urcorner$ lists the rules δ of T, with q_s rule first, q_f rule last.

$$\begin{aligned} trans(q,(s,s'),q') &= \mathtt{S}\#e_Q(q)\#e_\Sigma(s)e_\Sigma(s')\#rev(e_Q(q'))\#\mathtt{S} \\ trans(q,d,q') &= \mathtt{M}\#e_Q(q)\#e_D(d)\#rev(e_Q(q'))\#\mathtt{M} \end{aligned}$$

$e_Q : Q \to \{0,1\}^{\lceil \log |Q| \rceil}$ is an injective binary encoding of states.

$$e_\Sigma(s) = \begin{cases} \mathtt{B} & \text{if } s = b \\ s & \text{otherwise} \end{cases} \qquad e_D(d) = \begin{cases} 10 & \text{if } d = \leftarrow \\ \mathtt{BB} & \text{if } d = \downarrow \\ 01 & \text{if } d = \rightarrow \end{cases}$$

$rev : \Sigma \to \Sigma$ reverses a string.

Program encoding, example

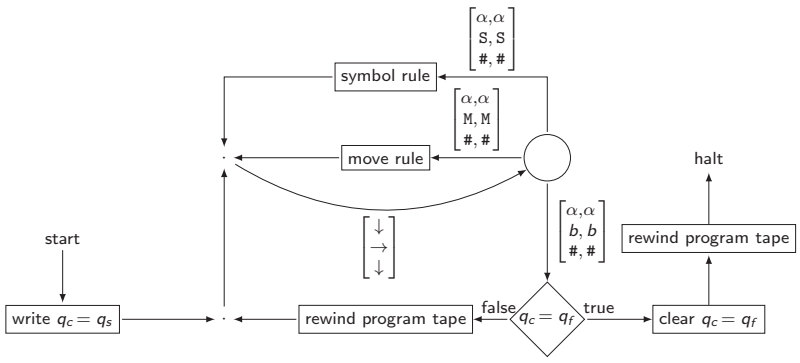
RTM $T = (\{q_0, q_1, q_2, q_3\}, \{b, 0, 1\}, \delta, b, q_0, q_3)$

$$\delta = \{(q_0, \rightarrow, q_1), (q_1, (0,1), q_2), (q_1, (1,0), q_2), (q_2, \leftarrow, q_3)\}$$

$\ulcorner T \urcorner = \mathtt{M}\#00\#01\#10\#\mathtt{MS}\#01\#01\#01\#\mathtt{SS}\#01\#10\#01\#\mathtt{SM}\#10\#10\#11\#\mathtt{M}$

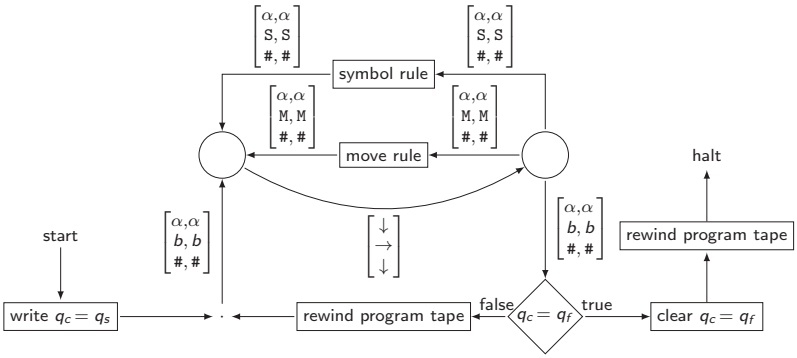
e_Q is given as $q_0 \mapsto 00$, $q_1 \mapsto 01$, $q_2 \mapsto 10$ and $q_3 \mapsto 11$.

URTM program



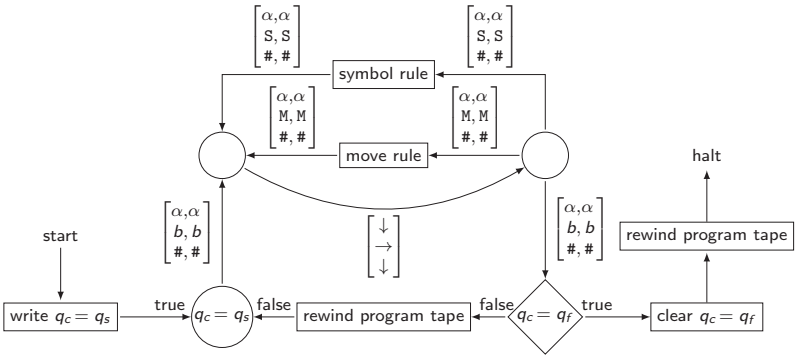
Problem: Lots of irreversibilities in control flow.

URTM program



Use enclosing S,M to join paths after rule tests.

URTM program



Works because q_s is only visited once.

String comparison

A key functionality we need to implement is string comparison.

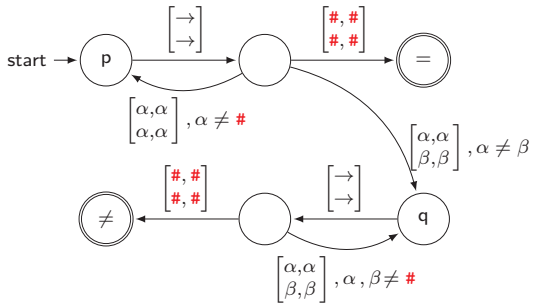
Assume a 2-tape structure

- $\#s_1 \cdots s_n\#$
- $\#t_1 \cdots t_n\#$

with tape heads on the *leftmost* #.

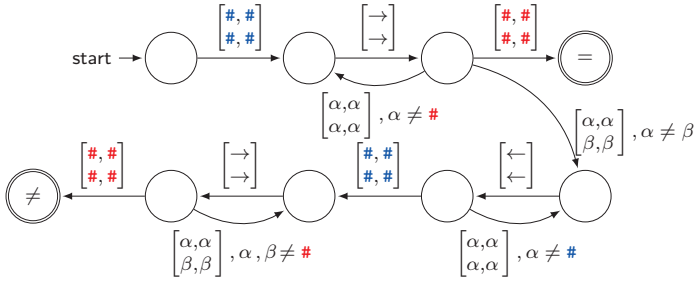
From internal state q, we want to pass over the strings, ending in internal state $q_=$ if the strings match, and in internal state $q_\neq$ if they don't, with the tape heads in either case on the *rightmost* #.

Irreversible string comparison of $\#s_1 \cdots s_n\#$ $\#t_1 \cdots t_n\#$

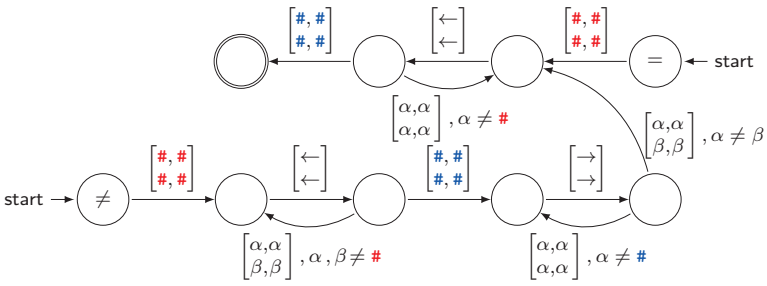


Irreversibility at state q (and probably p).

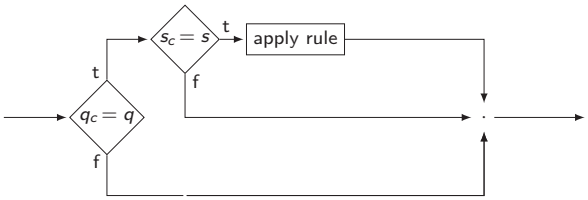
Reversible string comparison of $\#s_1 \cdots s_n\#$ $\#t_1 \cdots t_n\#$



Inverse string comparison of $\#s_1 \cdots s_n\#$ $\#t_1 \cdots t_n\#$ = join

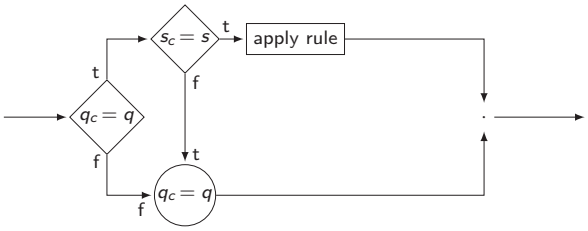


Testing a symbol rule $(q, (s, s'), q')$



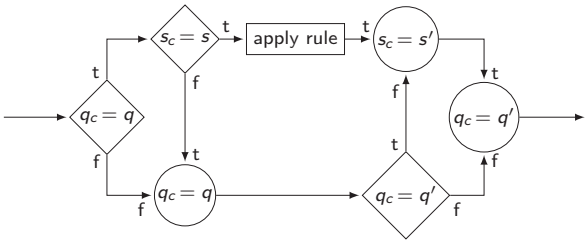
Must resolve the join in control flow.

Testing a symbol rule $(q, (s, s'), q')$



This assertion works for all T. Still irreversible...

Testing a symbol rule $(q, (s, s'), q')$



Only possible because T is reversible!

Bonus: Inverse interpretation

URTM can work as a (reversible) *inverse interpreter* with no extra overhead. A reversible inverse interpreter is a program *rinvint* s.t.

$$[\![rinvint]\!](p, y) = (p, x) \quad \text{i} \quad \text{ff}[\![p]\!](x) = y$$

We intentionally designed the program encoding s.t.

$$rev(\ulcorner T \urcorner) = \ulcorner T^{-1} \urcorner$$

Let R perform this *string reversal*. (Time linear in size of $\ulcorner T \urcorner$.)

$$[\![R \circ U \circ R]\!](\ulcorner T \urcorner, y) \quad = \quad (\ulcorner T \urcorner, [\![T^{-1}]\!](x))$$

Conclusion

First URTM with

- Only constant factor slowdown (proportional to $length(\ulcorner T \urcorner)$.)
- No space overhead (unlike all previous approaches.)
- Inverse interpretation for free.

The RTMs can thus simulate themselves efficiently.

References:
H. B. Axelsen and R. Glück: *What do reversible programs compute?* FOSSACS 2011, LNCS 6604, pp. 42–56, Springer, 2011.
H. B. Axelsen and R. Glück: *A Simple and Efficient Universal Reversible Turing Machine*. LATA 2011. To appear.

The Fourth DIKU-IST Joint Workshop on Foundations of Software
January 10th, 2011

Generating Input-Erasing Efficient Clean Reversible Programs for Injective Functions

TETSUO YOKOYAMA
Nanzan University, Japan

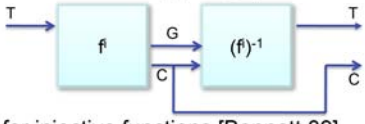
Existing Work and Our Approach

- A clean reversible simulation of injective programs is possible [Bennett89].
  - 2 computation + 2 inverse computation
- We argue that for a certain class of injective programs 1 computation + 1 uncomputation are necessary.
  - Twice as fast as Bennett's.
  - Demonstration by developing lossless encoders and decoders

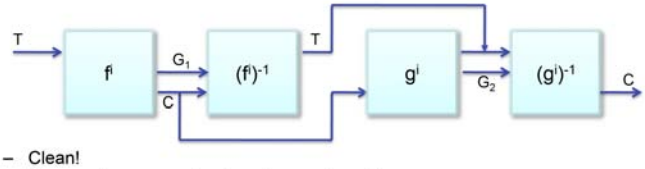
A new view on developing efficient reversible simulations for a certain class of injective functions

Reversible Simulations

- Bennett spying method [Bennett 73]
  - Garbage G is removed but T (garbage) remained.



- Bennett method for injective functions [Bennett 89]



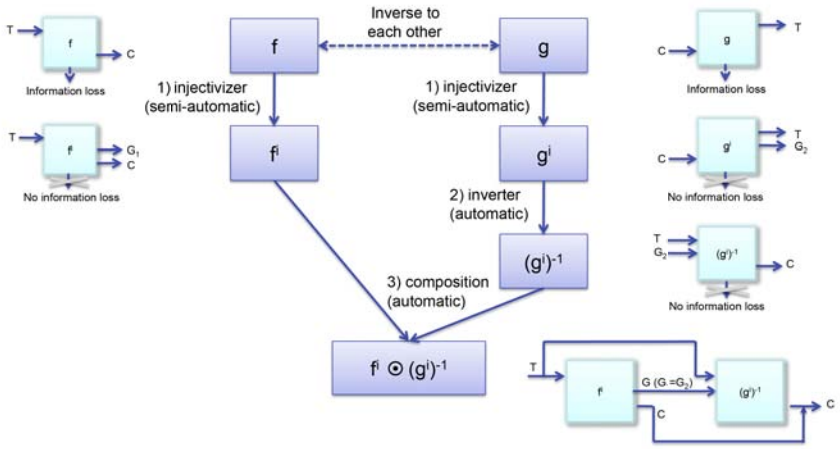
  - Clean!
  - In general, we cannot reduce the number of boxes.
  - How can we optimize it for a certain class of problems?

Reversible Simulations

- *The new method* (for domain specific problems)

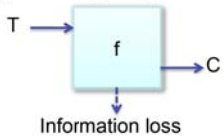
  - Twice as fast as Bennett's general method.
  - Still, clean!
  - G_1 must be equal to G_2.

The development process of the proposed clean reversible simulation

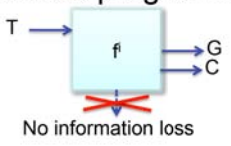


Injectivization at programming language level

- Injective program



- Injectivized program of f



- Ex.
  - Injective program
    - with information loss

```
procedure p
  if e
  then ...
  else ...
```

Postcondition inference is in general undecidable

  - Trivial injectivization
    - without information loss

```
procedure p^i
  if e
  then ...
       push(1,G)
  else ...
       push(2,G)
  fi top(G) = 1
```

Case Study: Run-length encoding (1)

A A A A A B B B B B B B B A A A…
⇒ A 5 B 9 A 3…

Reversibilization

Orthogonalization by adding a garbage variable n.
No human invention required.

```
procedure rl_gen_code                                   f^i
 from  n=0
 loop  //Count # of the same characters
       from len=0
       do    len += 1
             popfront(text,c)
             n += 1
       loop c ^= front(text) //Zero-clear c
       until empty(text) || c!=front(text)

       //Output the encoded text
       pushback(code,c)
       pushback(code,len)
 until empty(text)
```

The variable n after computation is the same.

```
procedure rl_gen_text                                   g^i
 from n=0
 loop  popfront(code,c)
       popfront(code,len)

       //Copy c for len times
       from empty(text) || c!=front(text)
       loop len -= 1
             pushfront(text,c)
             n += 1
       until len=0
 until  empty(code)
```

f^i(Text) = (Code, n)
g^i(Code) = (Text, n)

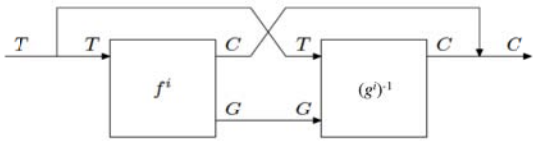
One Procedure, Two meaning (2)

call rng_gen_text

uncall rng_gen_text

```
procedure rl_gen_text
 from n=0
 loop  popfront(code,c)
       popfront(code,len)

       //Copy c for len times
       from empty(text) || c!=front(text)
       loop len -= 1
             pushfront(text,c)
             n += 1
       until len=0
until empty(code)
```

Run-length encoding (3)



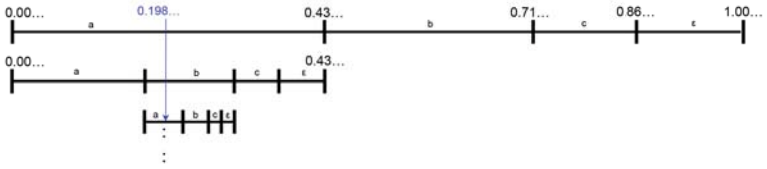
(a) The proposed input-erasing reversible simulation, in the case that the forward and backward garbage are the same.

```
procedure rl_encode
  ... copy text ...
  call    rl_gen_code                    f^i
  ... restore text and backup code  ...
  uncall rl_gen_text                     (g^i)^-1
  ... erase backup code …
```

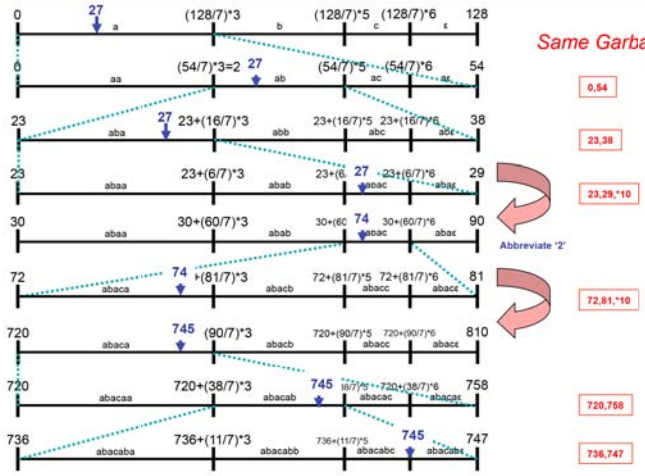
Case study: Range coding (Arithmetic coding)

1. Given a text, count the numbers of characters and make a model.
2. Mapping from texts to floating numbers and return the number.

- Example
  - Encoding: “abacab” → model(a=3,b=2,c=1) + 0.198…
  - Decoding: model(a=3,b=2,c=1) + 0.198… → “abacab”



Range Coding: Encoding “abacabε” ⇒ “2745”
Decoding “2745” ⇒ “abacabε”



(1)

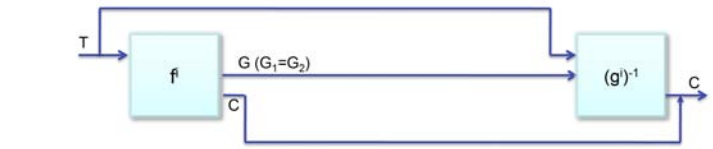
```
procedure rng_gen_code                                   f
  rng ^= range_size
  call count_freq
  from  n=0
  loop  popfront(text,t)
        pushback(gbg,low)
        pushback(gbg,rng)
        // ...set new low and new rng…

        if rng <= threshold then
          call expand_rng
          tmp ^= 1
        fi tmp = 1
        pushback(gbg,tmp)
        n += 1
        pushback(gbg,t)
  until empty(text)
  pushback(gbg,rng)
  pushback(gbg,n)
  code ^= low
  pushback(gbg,low)
```

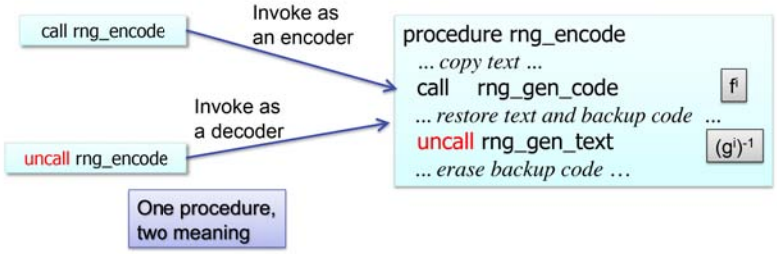
```
procedure rng_gen_text                                   g⁻¹
  rng ^= range_size

  from  n=0
  loop  //  set decoded character to t, and
        //  pushback t to text…
        pushback(gbg,low)
        pushback(gbg,rng)
        //...set new low and new rng
        if rng <= threshold then
          call expand_rng
          tmp ^= 1
        fi tmp=1
        pushback(gbg,tmp)
        n += 1
        pushback(gbg,t)
  until back(text) = ε
  pushback(gbg,rng)
  pushback(gbg,n)
  code ^= low
  pushback(gbg,low)
  uncall count_freq
```

Semi-automatic!

- Count the number of loops.
- Zero-clearing of variables were performed by gbg stack.

Wrap them up! (3)





Concluding remarks

- The efficient (semi-)automatic reversible simulations for every single problem domain are important.
  - There are huge amount of heritage from irreversible programming.
  - Programming reversibly from scratch is involved.
- We proposed an optimized clean Bennett method.
  - The execution time of the mechanically translated reversible algorithm was halved.
- As each programming paradigm does, reversible programming needs its own programming methodology.
  - We showed a power of the proposed method by successfully demonstrating the optimization of lossless data encoding and decoding.

References

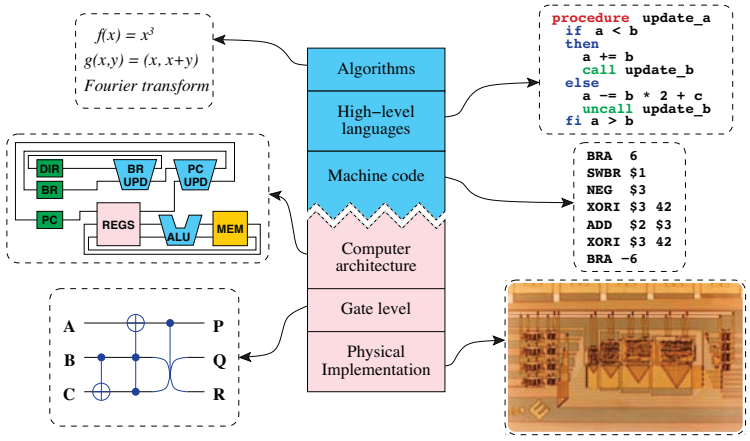
- Yokoyama T., Axelsen H.B., Glück R., Optimization of Input-Erasing Clean Reversible Simulation for Injective Functions, Journal of Multiple-Valued Logic and Soft Computing, 2011. To appear.
- Yokoyama T., Glück R., A reversible programming language and its invertible self-interpreter. In: Partial Evaluation and Program Manipulation. Proceedings. 144-153, ACM Press 2007.
- Axelsen H.B., Glück R., Yokoyama T., Reversible machine code and its abstract processor architecture. In: Diekert V., et al. (eds.), Computer Science - Theory and Applications. Lecture Notes in Computer Science, Vol. 4649, 56-69, Springer-Verlag 2007.
- Yokoyama T., Axelsen H.B., Glück R., Principles of a reversible programming language. In: Conference on Computing Frontiers. Proceedings. 43-54, ACM Press 2008.
- Yokoyama T., Axelsen H.B., Glück R., Reversible Flowchart Languages and the Structured Reversible Program Theorem. In: International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science, Vol. 5126, pp. 258-270, Springer-Verlag, 2008.

Reversible Coroutines

Poul J. Clementsen, Robert Glück & Holger B. Axelsen
DIKU, University of Copenhagen, Denmark

4th DIKU-IST Joint Workshop on Foundations of Software
Tokyo, Japan, January 2011

Reversible Computing



Motivation

Current **reversible programming languages** have few program constructs because of the constraints posed by reversibility.

We show how this expressiveness can be greatly extended by adding a **coroutine construct** without losing reversibility.

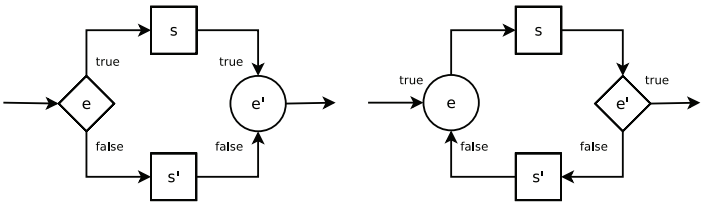
Reversible Programming Language

Characteristics

- Forward and backward deterministic
- Reversible update of variables `x += e`
- Local inversion
- Control flow operators require assertions

Reversible Programming Language

Reversible Structured Control Flow



(a) Conditional (b) Loop

Janus Programming Language

Example (Fibonacci pair)

procedure *main*
```
    x1 += 3
    x2 += 5
    uncall fib
```

procedure *fib*
```
    x1 += 1
    x2 += 1
    from   x1 = x2
    do     n -= 1
    loop   x1 += x2
           x1 <=> x2
    until  n = 0
```

n	x1	x2
0	3	5
4	0	0

Janus Programming Language

Abstract Syntax

$$
\begin{array}{rcl}
\mathit{prog} & ::= & (x\,\texttt{[}\,c\,\texttt{]})^*\ p^+ \\
p & ::= & \texttt{procedure}\ \mathit{id}\ s^+ \\
s & ::= & \texttt{x +=}\ e \mid \texttt{x -=}\ e \mid \texttt{x <=> x} \mid \\
 & & \texttt{if}\ e\ \texttt{then}\ s^*\ \texttt{else}\ s^*\ \texttt{fi}\ e \mid \\
 & & \texttt{from}\ e\ \texttt{do}\ s^*\ \texttt{loop}\ s^*\ \texttt{until}\ e \mid \\
 & & \texttt{call}\ \mathit{id} \mid \texttt{uncall}\ \mathit{id} \\
e & ::= & c \mid x \mid e \otimes e \\
\otimes & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \cdots
\end{array}
$$

Reversible Coroutine

Motivation for looking at reversible coroutine

- Coroutines are well-suited for defining
  - Cooperative tasks
  - Iterators
  - Generators
  - Infinite lists
  - Pipes
- Only reversible subroutine exist
- Gain expressiveness in a disciplined way

Reversible Coroutines in Janus

Example (Fibonacci pair generator w/ coroutine)

```
procedure main
    n += 2
    call fib_co
    call fib_co
    uncall fib_co
```

```
coroutine fib_co
    x1 += 1
    x2 += 1
    from   x1 = x2
    do     n -= 1
    loop   yield
           x1 += x2
           x1 <=> x2
    until  n = 0
```

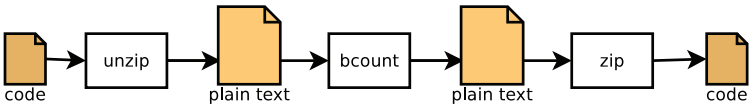
n	x1	x2
2	0	0
2	1	1
1	1	1
1	1	2
0	1	2
1	1	2
1	1	1

Application

Application of reversible coroutines

- Are coroutines well-suited for defining program components?
  - Generators
  - Iterators
  - Cooperative tasks
  - Pipes

Reversible Pipeline using Coroutine



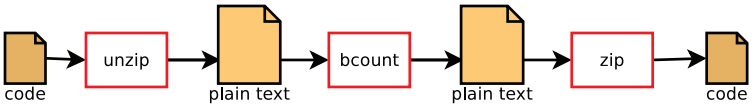
Example (Unix Pipe)

```
cat code | unzip | bcount | zip > code'
```

Pipeline implementation

- Communicate through circular buffer
- Define coroutines zip, unzip and bcount and methodology for communicating with pipes
- Pipeline scheduler linking the coroutines

Pipeline Implementation



Pipe Coroutines Methodology

```
coroutine proc(IN, OUT)
    from    empty(OUT)
    do      yield
    loop    get(IN) <=> in
            body
            put(OUT) <=> out
    until   empty(IN)
```

Pipeline Implementation

```
procedure pipeline
    // ... initialize buffer code ...
    call unzip(a, b)
    call bcount(b, c)
    call zip(c, d)
    from empty(b) && empty(c) && empty(d)
    do   call unzip(a, b)
         call bcount(b, c)
         call zip(c, d)
    until empty(a) && empty(b) && empty(c)
    call bcount(b, c)
    call zip(c, d)

coroutine bcount(in, out)
  from empty(out)
  do   yield
  loop from empty(out)
       do get(in) <=> chr
          if chr = 'b'
          then count += 1
          fi chr = 'b'
          put(out) <=> chr
       until empty(in)
  until empty(in)
  put(out) <=> count
```

```
coroutine unzip(in, out)
    from empty(out)
    do   yield
    loop get(in) <=> code
         get(in) <=> length
         from ppeek(out) != code || empty(out)
         do   length -= 1
         loop put(out) ^= code
         until length = 0
         put(out) <=> code
    until empty(in)

coroutine zip(in, out)
    from empty(out)
    do   yield
    loop get(in) <=> code
         from length = 0
         do   length += 1
         loop get(in) ^= code
         until gpeek(in) != code || empty(in)
         put(out) <=> code
         put(out) <=> length
    until empty(in)
```


Translation

Translation to a Reversible Instruction Set Architecture

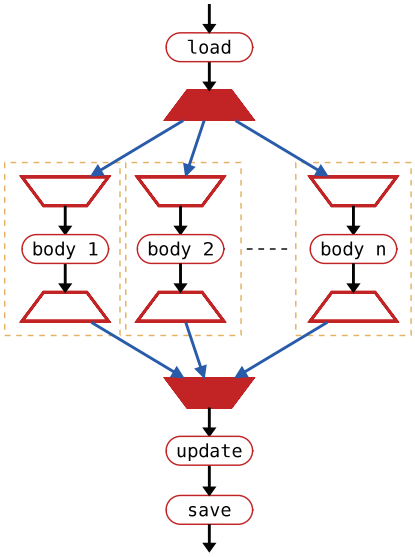
- Preserve reversibility properties of the language
- Exploit reversibility properties of the low-level language
- Gain properties of an implementation in reversible hardware, *i.e.*, lower heat dissipation

Translation

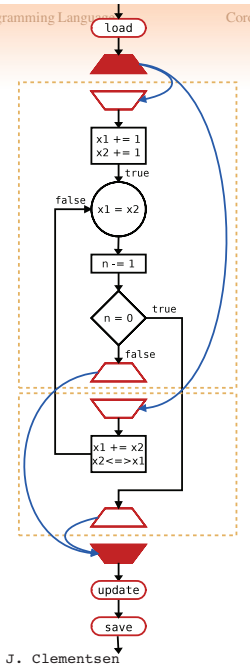
Reversible Coroutine Translation Challenges

- Granularity is finer
- Control flow
- Coroutine state
- Clean translation (*i.e.*, w/o trace data)

Coroutine Translation



Coroutine Translation



coroutine fib_{co}

```
x1 += 1
x2 += 1
from x1 = x2
do   n -= 1
loop yield
     x1 += x2
     x1 <=> x2
until n = 0
```


Conclusion

- Coroutines are a powerful way of gaining expressiveness in a disciplined way
- First step: Generator, pipeline and cooperative processes
- Adds first stateful construct in reversible languages
- Clean translation scheme

Future Work

- Expressive powers of coroutines need to be examined further, *i.e.*, defining cooperative tasks and state machines
- Generator and pipeline examples presented need to be refined and generalized
- Refine coroutine notation to include, *e.g.*, instances and local variables

Monadic Effects in Operational Semantics

Based on **Monads in Action**, POPL 2010

Andrzej Filinski
andrzej@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

4th DIKU-IST Workshop on Foundations of Software
January 10-14, Tokyo

Background and overview

- Two common approaches to uniformly specifying effectful operations in functional languages:
  1. Monad of computations + pure definitions of operations
     - Translate client program using monad components, plug in operation definitions, evaluate by core semantics
     - Typically used in Haskell-like settings
  2. Stylized evaluation contexts + context-rewriting operations
     - Formalize context shapes, extend core semantics with new rules for effectful operations
     - Typically used in ML/Scheme-like settings
- This talk: *generically* obtaining (2) from (1)
  - ... without giving up the monadic equational theory!
- In paper: details for full *multi*monadic metalanguage
  - Here: for single effect only; expressed in Haskell subset (slightly oversimplified)
  - There is a complementary story for ML-like settings

Monadic Haskell with Int-carrying exceptions

$$
\begin{array}{lcl}
t & ::= & \overbrace{\mathsf{Int} \mid t_1 \to t_2 \mid \mathsf{Either}\ t_1\ t_2}^{\textit{Core lang.}} \mid \mathsf{M}_\varepsilon\ t \\
\varepsilon & ::= & \mathit{id} \mid \mathit{ex} \mid \cdots \\
e & ::= & \left.\begin{array}{l} n \mid x \mid \lambda x \to e \mid e_1\ e_2 \mid \mathsf{Left}\ e \mid \mathsf{Right}\ e \\ \mid \mathbf{case}\ e_0\ \mathbf{of}\ \{\mathsf{Left}\ x \to e_1; \mathsf{Right}\ y \to e_2\} \end{array}\right\}\textit{Core lang.} \\
 & & \mid \mathbf{return}^\varepsilon\ e \mid \mathbf{do}^\varepsilon\ x \leftarrow e_1; e_2 \mid \mathbf{raise}\ e \mid \mathbf{try}^\varepsilon\ e_1\ \mathbf{with}\ x \to e_2
\end{array}
$$

Typing judgment $\Gamma \vdash e :: t$. Usual rules for core constructs +

$$\frac{\Gamma \vdash e :: t}{\Gamma \vdash \mathbf{return}^\varepsilon\ e :: \mathsf{M}_\varepsilon\ t} \qquad \frac{\Gamma \vdash e_1 :: \mathsf{M}_\varepsilon\ t_1 \quad \Gamma, x\colon t_1 \vdash e_2 :: \mathsf{M}_\varepsilon\ t_2}{\Gamma \vdash \mathbf{do}^\varepsilon\ x \leftarrow e_1; e_2 :: \mathsf{M}_\varepsilon\ t_2}$$

$$\frac{\Gamma \vdash e :: \mathsf{Int}}{\Gamma \vdash \mathbf{raise}\ e :: \mathsf{M}_{ex}\ t} \qquad \frac{\Gamma \vdash e_1 :: \mathsf{M}_{ex}\ t \quad \Gamma, x\colon \mathsf{Int} \vdash e_2 :: \mathsf{M}_\varepsilon\ t}{\Gamma \vdash \mathbf{try}^\varepsilon\ e_1\ \mathbf{with}\ x \to e_2 :: \mathsf{M}_\varepsilon\ t}$$

Superscripts ε on **return**, **do**, **try** added by overloading resolution.

Specifying exceptions with a monad

Transparent/*concrete* definition of *exception monad*:

$$\textbf{type } \mathsf{T}_{ex}\ a = \mathsf{Either}\ a\ \mathsf{Int} \quad \text{-- not } \textbf{newtype}$$
$$\mathsf{unit}_{ex} :: a \to \mathsf{T}_{ex}\ a$$
$$\mathsf{unit}_{ex}\ a = \mathsf{Left}\ a$$
$$\mathsf{bind}_{ex} :: \mathsf{T}_{ex}\ a \to (a \to \mathsf{T}_{ex}\ b) \to \mathsf{T}_{ex}\ b$$
$$\mathsf{bind}_{ex}\ t\ f = \textbf{case } t \textbf{ of } \{\mathsf{Left}\ a \to f\ a;\ \mathsf{Right}\ n \to \mathsf{Right}\ n\}$$

Used to implement *abstract* effect of exceptions:

$$\textbf{newtype } \mathsf{M}_\varepsilon\ a = \mathsf{Reflect}_\varepsilon\ (\mathsf{T}_\varepsilon\ a) \qquad \mathsf{Reflect}_\varepsilon :: \mathsf{T}_\varepsilon\ a \to \mathsf{M}_\varepsilon\ a$$
$$\mathsf{reify}_\varepsilon\ (\mathsf{Reflect}_\varepsilon\ t) = t \quad \text{-- aka. } \mathsf{run}_\varepsilon \qquad \mathsf{reify}_\varepsilon :: \mathsf{M}_\varepsilon\ a \to \mathsf{T}_\varepsilon\ a$$
$$\textbf{return}^\varepsilon\ e = \mathsf{Reflect}_\varepsilon\ (\mathsf{unit}_\varepsilon\ e)$$
$$\textbf{do}^\varepsilon\ x \leftarrow e_1; e_2 = \mathsf{Reflect}_\varepsilon\ (\mathsf{bind}_\varepsilon\ (\mathsf{reify}_\varepsilon\ e_1)\ (\lambda x \to \mathsf{reify}_\varepsilon\ e_2))$$
$$\textbf{raise}\ e \equiv \mathsf{Reflect}_{ex}\ (\mathsf{Right}\ e) \quad \text{-- definitional abbrevs.}$$
$$\textbf{try}^\varepsilon\ e_1\ \textbf{with}\ x \to e_2 \equiv$$
$$\quad \textbf{case } \mathsf{reify}_{ex}\ e_1 \textbf{ of } \{\mathsf{Left}\ a \to \textbf{return}^\varepsilon\ a;\ \mathsf{Right}\ x \to e_2\}$$

Standard (def.-unfolding) operational semantics

$$e ::= (\mathit{core}) \mid \textbf{return}^\varepsilon\ e \mid \textbf{do}^\varepsilon\ x \leftarrow e_1; e_2 \mid \mathsf{Reflect}_\varepsilon\ e \mid \mathsf{reify}_\varepsilon\ e$$

Reduction judgment $\boxed{e \longrightarrow e'}$ for *closed* terms:

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{}{(\lambda x \to e_1)\ e_2 \longrightarrow e_1[e_2/x]}$$

$$\frac{e_0 \longrightarrow e_0'}{\textbf{case } e_0 \textbf{ of } \{\cdots\} \longrightarrow \textbf{case } e_0' \textbf{ of } \{\cdots\}}$$

$$\frac{}{\textbf{case } \mathsf{Left}\ e \textbf{ of } \{\mathsf{Left}\ x_1 \to e_1;\ \mathsf{Right}\ x_2 \to e_2\} \longrightarrow e_1[e/x]}\ (+symm)$$

$$\frac{}{\textbf{return}^\varepsilon\ e \longrightarrow \mathsf{Reflect}_\varepsilon\ (\mathsf{unit}_\varepsilon\ e)}$$

$$\frac{}{\textbf{do}^\varepsilon\ x \leftarrow e_1; e_2 \longrightarrow \mathsf{Reflect}_\varepsilon\ (\mathsf{bind}_\varepsilon\ (\mathsf{reify}_\varepsilon\ e_1)\ (\lambda x \to \mathsf{reify}_\varepsilon\ e_2))}$$

$$\frac{e \longrightarrow e'}{\mathsf{reify}_\varepsilon\ e \longrightarrow \mathsf{reify}_\varepsilon\ e'} \qquad \frac{}{\mathsf{reify}_\varepsilon\ (\mathsf{Reflect}_\varepsilon\ e) \longrightarrow e}$$

Note: tags on $\textbf{return}^\varepsilon$, $\textbf{do}^\varepsilon$ play essential role in behavior.
Note: code for $\mathsf{unit}_\varepsilon$, $\mathsf{bind}_\varepsilon$ traversed on every $\textbf{return}^\varepsilon$, $\textbf{do}^\varepsilon$.

Equational theory

If $(\mathsf{T}_\varepsilon, \mathsf{unit}_\varepsilon, \mathsf{bind}_\varepsilon)$ satisfy monad laws, get additional valid reasoning principles for observational equivalence:

$$\begin{aligned}
(\lambda x \to e_1)\ e_2 &= e_1[e_2/x] \\
\lambda x \to e\ x &= e \quad (x \notin FV(e)) \\
&\vdots \\
\textbf{do}^\varepsilon\ x \leftarrow \textbf{return}^\varepsilon\ e_1; e_2 &= e_2[e_1/x] \\
\textbf{do}^\varepsilon\ x \leftarrow e; \textbf{return}^\varepsilon\ x &= e \\
\textbf{do}^\varepsilon\ y \leftarrow (\textbf{do}^\varepsilon\ x \leftarrow e_1; e_2); e_3 &= \textbf{do}^\varepsilon\ x \leftarrow e_1; \textbf{do}^\varepsilon\ y \leftarrow e_2; e_3 \quad (x \notin FV(e_3)) \\
\\
\mathsf{reify}_\varepsilon\ (\textbf{return}^\varepsilon\ e) &= \mathsf{unit}_\varepsilon\ e \\
\mathsf{reify}_\varepsilon\ (\textbf{do}^\varepsilon\ x \leftarrow e_1; e_2) &= \mathsf{bind}_\varepsilon\ (\mathsf{reify}_\varepsilon\ e_1)\ (\lambda x \to \mathsf{reify}_\varepsilon\ e_2) \\
\mathsf{reify}_\varepsilon\ (\mathsf{Reflect}_\varepsilon\ e) &= e \\
\mathsf{Reflect}_\varepsilon\ (\mathsf{reify}_\varepsilon\ e) &= e
\end{aligned}$$

Can we operationalize these equations in a different way?

New (effectful) operational semantics

$e ::= (core) \mid \mathbf{return}\ e \mid \mathbf{do}\ x \leftarrow e_1; e_2 \mid \mathrm{reify}_\varepsilon\ e \mid \mathbf{do}\ x \leftarrow \mathrm{Reflect}_\varepsilon\ e_1; e_2$

$\mathrm{reflect}_\varepsilon\ e \equiv \mathbf{do}\ x \leftarrow \mathrm{Reflect}_\varepsilon\ e; \mathbf{return}\ x$ -- for convenience

$\boxed{e \longrightarrow e'}$

(Unmodified rules for the core constructs)

$$\frac{e_1 \longrightarrow e_1'}{\mathbf{do}\ x \leftarrow e_1; e_2 \longrightarrow \mathbf{do}\ x \leftarrow e_1'; e_2} \qquad \frac{}{\mathbf{do}\ x \leftarrow \mathbf{return}\ e_1; e_2 \longrightarrow e_2[e_1/x]}$$

$$\frac{}{\mathbf{do}\ y \leftarrow (\mathbf{do}\ x \leftarrow \mathrm{Reflect}_\varepsilon\ e_1; e_2); e_3 \longrightarrow \mathbf{do}\ x \leftarrow \mathrm{Reflect}_\varepsilon\ e_1; \mathbf{do}\ y \leftarrow e_2; e_3}\ (*)$$

$$\frac{e \longrightarrow e'}{\mathrm{reify}_\varepsilon\ e \longrightarrow \mathrm{reify}_\varepsilon\ e'} \qquad \frac{}{\mathrm{reify}_\varepsilon\ (\mathbf{return}\ e) \longrightarrow \mathrm{unit}_\varepsilon\ e}$$

$$\frac{}{\mathrm{reify}_\varepsilon\ (\mathbf{do}\ x \leftarrow \mathrm{Reflect}_\varepsilon\ e_1; e_2) \longrightarrow \mathrm{bind}_\varepsilon\ e_1\ (\lambda x \to \mathrm{reify}_\varepsilon\ e_2)}\ (*)$$

Note: no effect-tags needed on **return**, **do**.

(∗)-rules: connect $\mathrm{Reflect}_\varepsilon$ to nearest dynamically enclosing $\mathrm{reify}_\varepsilon$

Properties of reduction semantics

- **Sound:** if $e \longrightarrow e'$, then $e = e'$ in equational theory.
- **Deterministic:** if $e \longrightarrow e'$ and $e \longrightarrow e''$, then $e' \equiv e''$.
- **Type-preserving:** if $\cdot \vdash e :: t$ and $e \longrightarrow e'$, then $\cdot \vdash e' :: t$.
- **Non-sticking:** if $\cdot \vdash e :: t$, then either e canonical, or $e \longrightarrow e'$ for some e'. Canonical forms are:

$$\overbrace{n}^{Int} \mid \overbrace{\lambda x \to e}^{t_1 \to t_2} \mid \overbrace{\mathrm{Left}\ e \mid \mathrm{Right}\ e}^{Either\ t_1\ t_2} \mid \overbrace{\mathbf{return}\ e \mid \mathbf{do}\ x \leftarrow \mathrm{Reflect}_\varepsilon\ e_1; e_2}^{\mathrm{M}_\varepsilon\ t}$$

  Note: a canonical M-computation is either effect-free, or an *immediate* effect invocation.

- In particular, a closed term of type Int (but using monadic effects internally) must eventually reduce to an n, or diverge.

Evaluation-context formulation of new semantics

General and restricted evaluation contexts:

$$E ::= [] \mid E\ e \mid \mathbf{case}\ E\ \mathbf{of}\ \{\cdots\} \mid \mathbf{do}\ x \leftarrow E; e \mid \mathrm{reify}_\varepsilon\ E$$
$$F ::= [] \mid \mathbf{do}\ x \leftarrow F; e \qquad \text{(in particular, no } \mathrm{reify}_\varepsilon\ F\text{)}$$

Bigger-step judgment $\boxed{e \longrightarrow\!\!\!\!\!\rightarrow e'}$:

$$\frac{e \longrightarrow\!\!\!\!\!\rightarrow e'}{E[e] \longrightarrow\!\!\!\!\!\rightarrow E[e']} \qquad \frac{}{(\lambda x \to e_1)\ e_2 \longrightarrow\!\!\!\!\!\rightarrow e_1[e_2/x]}$$

$$\frac{}{\mathbf{case}\ \mathrm{Left}\ e\ \mathbf{of}\ \{\mathrm{Left}\ x_1 \to e_1; \mathrm{Right}\ x_2 \to e_2\} \longrightarrow\!\!\!\!\!\rightarrow e_1[e/x]}\ (+symm)$$

$$\frac{}{\mathbf{do}\ x \leftarrow \mathbf{return}\ e_1; e_2 \longrightarrow\!\!\!\!\!\rightarrow e_2[e_1/x]} \qquad \frac{}{\mathrm{reify}_\varepsilon\ (\mathbf{return}\ e) \longrightarrow\!\!\!\!\!\rightarrow \mathrm{unit}_\varepsilon\ e}$$

$$\frac{}{\mathrm{reify}_\varepsilon\ (F[\mathrm{reflect}_\varepsilon\ e]) \longrightarrow\!\!\!\!\!\rightarrow \mathrm{bind}_\varepsilon\ e\ (\lambda x \to \mathrm{reify}_\varepsilon\ (F[\mathbf{return}\ x]))}\ (*)$$

Sound: if $e \longrightarrow\!\!\!\!\!\rightarrow e'$ then $e \longrightarrow^+ e'$.

Example: exceptions (again)

Recall definitions of exception monad and operations:

$$\begin{aligned} \mathsf{T}_{ex}\, a &= \text{Either } a \text{ Int};\ \mathsf{unit}_{ex}\, a = \text{Left } a;\ \mathsf{bind}_{ex} = \cdots \\ \textbf{raise}\ e &\equiv \mathsf{reflect}_{ex}\,(\text{Right } e) \\ \textbf{try}\ e_1\ \textbf{with}\ x \to e_2 &\equiv \\ \multicolumn{2}{r}{\textbf{case}\ \mathsf{reify}_{ex}\, e_1\ \textbf{of}\ \{\text{Left } a \to \textbf{return}\ a;\ \text{Right } x \to e_2\}} \end{aligned}$$

Gives *derivable* typing and reduction rules:

$$\frac{\Gamma \vdash e :: \text{Int}}{\Gamma \vdash \textbf{raise}\ e :: \mathsf{M}_{ex}\, t} \qquad \frac{\Gamma \vdash e_1 :: \mathsf{M}_{ex}\, t \qquad \Gamma, x : \text{Int} \vdash e_2 :: \mathsf{M}_{\varepsilon}\, t}{\Gamma \vdash \textbf{try}\ e_1\ \textbf{with}\ x \to e_2 :: \mathsf{M}_{\varepsilon}\, t}$$

$$\frac{e_1 \longrightarrow e_1'}{\textbf{try}\ e_1\ \textbf{with}\ x \to e_2 \longrightarrow \textbf{try}\ e_1'\ \textbf{with}\ x \to e_2}$$

$$\frac{}{\textbf{try return}\ e_0\ \textbf{with}\ x \to e_2 \longrightarrow^{+} \textbf{return}\ e_0}$$

$$\frac{}{\textbf{try}\ F[\textbf{raise}\ e_0]\ \textbf{with}\ x \to e_2 \longrightarrow^{+} e_2[e_0/x]}$$

Example: state

Standard definitions of state monad and operations:

$$\begin{aligned} \mathsf{T}_{st}\, a &= \text{Int} \to (a, \text{Int});\ \mathsf{unit}_{st}\, a = \lambda s \to (a, s);\ \mathsf{bind}_{st} = \cdots \\ \textbf{getst} &\equiv \mathsf{reflect}_{st}\,(\lambda s \to (s, s)) \\ \textbf{setst}\ e &\equiv \mathsf{reflect}_{st}\,(\lambda s \to ((), e)) \\ \textbf{withst}\ e_1\ \textbf{do}\ e_2 &\equiv \textbf{let}\ (a, s') = (\mathsf{reify}_{st}\, e_2)\, e_1\ \textbf{in}\ a \\ &\qquad \text{-- run } e_2 \text{ in initial state } e_1 \end{aligned}$$

Derived typing and reduction rules:

$$\frac{}{\Gamma \vdash \textbf{getst} :: \mathsf{M}_{st}\, \text{Int}} \qquad \frac{\Gamma \vdash e :: \text{Int}}{\Gamma \vdash \textbf{setst}\ e :: \mathsf{M}_{st}\, ()} \qquad \frac{\Gamma \vdash e_1 :: \text{Int} \Gamma \quad \vdash e_2 :: \mathsf{M}_{st}\, t}{\Gamma \vdash \textbf{withst}\ e_1\ \textbf{do}\ e_2 :: t}$$

$$\frac{e_2 \longrightarrow e_2'}{\textbf{withst}\ e_1\ \textbf{do}\ e_2 \longrightarrow \textbf{withst}\ e_1\ \textbf{do}\ e_2'} \qquad \frac{}{\textbf{withst}\ e_1\ \textbf{do return}\ e_2 \longrightarrow^{+} e_2}$$

$$\frac{}{\textbf{withst}\ e\ \textbf{do}\ F[\textbf{getst}] \longrightarrow^{+} \textbf{withst}\ e\ \textbf{do}\ F[\textbf{return}\ e]}$$

$$\frac{}{\textbf{withst}\ e\ \textbf{do}\ F[\textbf{setst}\ e'] \longrightarrow^{+} \textbf{withst}\ e'\ \textbf{do}\ F[\textbf{return}\ ()]}$$

Summary

- Monadic definitions of effects can be given *direct* operational interpretation. Curry-style type system: don't need to reconstruct types before evaluation.
- Rational reconstruction of evaluation-context semantics.
  - Related construction: taking implementation type $\mathsf{M}_{\varepsilon}\, a$ to be a delimited-continuations monad $\forall o.(a \to \mathsf{T}_{\varepsilon}\, o) \to \mathsf{T}_{\varepsilon}\, o$.
    $\Rightarrow$ embedding arbitrary monadic effects in Scheme.
- In paper:
  - Full core language with product, sum, function, recursive, and generalized-effect types; effect-subtyping.
  - Explicit syntax for effect definitions with layering.
  - Precise formulation of semantics (explicit and context-based), type system, type soundness – all mechanized in Twelf.
- Current work: correspondence between (domain-theoretic) denotational and operational semantics for monadic effects.

Report on using Generic Multiset Discrimination for a Probability Monad

Ken Friis Larsen
Department of Computer Science
University of Copenhagen
kflarsen@diku.dk

4th DIKU-IST Workshop

Probability Monad

- Our offset is the formulation by Erwig and Kollmansberger (2006)
- That probability distributions form a monad is not new Giry (1981)

Probability Monad in Haskell

```
type Probability = Float
newtype Dist a = D {unD::[(a,Probability)]}

instance Monad Dist where
  return x = D[(x, 1.0)]
  dist >>= f = dist `bind` f

bind :: Dist a -> (a -> Dist b) -> Dist b
```

Constructing Distributions

```
weightedCases::[(a, Probability)] -> Dist a
countedCases :: [(a, Int)] -> Dist a
uniform :: [a] -> Dist a
```

Example: Some Dice

```
die = uniform [1..6]

twoDice = do
    n1 <- die
    n2 <- die
    return (n1+n2)
```

Example: Traffic Light

```
data Light = Red | Green | Yellow

stopLight :: Dist Light
stopLight =
   countedCases[(Red,    50)
               ,(Yellow, 10)
               ,(Green,  40)]
```

Example: Driver Behavior

```
data Action = Stop | Drive

cautious :: Light -> Dist Action
cautious Red    = return Stop
cautious Green  = return Drive
cautious Yellow = countedCases[(Stop,9),(Drive,1)]

aggressive :: Light -> Dist Action
aggressive Red    = countedCases[(Stop,9),(Drive,1)]
aggressive Yellow = countedCases[(Stop,1),(Drive,9)]
aggressive Green  = return Drive
```

Implementing `bind`

```
bind :: Dist a -> (a -> Dist b) -> Dist b
bind (D d) cond =
  D [(y, p*q) | (x, p) <- d
              , (y, q) <- unD (cond x)]
```

We Got a Problem

```
> unD twoDice
[(2,2.777778e-2),(3,2.777778e-2),(4,2.777778e-2),
(5,2.777778e-2),(6,2.777778e-2),(7,2.777778e-2),
(3,2.777778e-2),(4,2.777778e-2),(5,2.777778e-2),
(6,2.777778e-2),(7,2.777778e-2),(8,2.777778e-2),
(4,2.777778e-2),(5,2.777778e-2),(6,2.777778e-2),
(7,2.777778e-2),(8,2.777778e-2),(9,2.777778e-2),
(5,2.777778e-2),(6,2.777778e-2),(7,2.777778e-2),
(8,2.777778e-2),(9,2.777778e-2),(10,2.777778e-2),
(6,2.777778e-2),(7,2.777778e-2),(8,2.777778e-2),
(9,2.777778e-2),(10,2.777778e-2),(11,2.777778e-2),
(7,2.777778e-2),(8,2.777778e-2),(9,2.777778e-2),
(10,2.777778e-2),(11,2.777778e-2),(12,2.777778e-2)]
```

Let's Fix it

```
bind (D d) cond = D d'
  where
    raw = [(y, p*q) | (x, p) <- d
                    , (y, q) <- unD (cond x)]
    sorted = sortBy (\(x,p)(y,q) -> compare x y) raw
    grouped = groupBy (\(x,p)(y,q) -> x==y) sorted
    d' = map (\g -> (fst$ head g, sum$ map snd g))
             grouped
```

We Got an Other Problem

```
Could not deduce (Ord b) from the context ()
      arising from a use of `bind'
    Possible fix:
      add (Ord b) to the context of the type
      signature for `>>='
    In the expression: bind dist f
    In the definition of `>>=': dist >>= f = bind
dist f
    In the instance declaration for `Monad Dist'
```

Using RMonad

```
data instance Constraints Dist a =
         Ord a => DistC
instance Ord a => Suitable Dist a where
  constraints = DistC

instance RMonad Dist where
  return  x = D[(x, 1)]
  d >>= k   = Suitable.withResConstraints $
              \DistC -> d `bind` k
```

Generic Multiset Discrimination

- We can also use Henglein's library for Generic multiset discrimination
- Key ingredient: generic symbolic equality relations
- Key function (for this talk):
  `part :: Equiv t -> [t] -> [[t]]`

GMSD With RMonad

```
class Eqv t where
   equiv :: Equiv t
data instance Constraints Dist a =
   Eqv a => DistC
instance Eqv a => Suitable Dist a where
  constraints = DistC

instance RMonad Dist where
  return  x = D[Pr x 1]
  d >>= k   = Suitable.withResConstraints $
              \DistC -> d `bind` k
```

`bind` with GMSD

```
bind :: Eqv a =>
        Dist t -> (t -> Dist a) -> Dist a

bind (D d) cond = D flattened
  where
     raw = [Pr y (p*q) | Pr x p <- d
                       , Pr y q <- unD (cond x)]
     grouped = Disc.part equiv raw
     flattened = [Pr (pfst$ head p)
                     (sum$ map psnd p)
                  | p <- grouped]
```

Near-future Work

- Performance evaluation
- Simulation framework
- Example from biology
  - Tree grows
  - Predator/Prey models

Future Work

- Monte Carlo simulation
- Examples from finance

Distant-future Work

- Symbolic representation of distributions
  - Perhaps extend beyond finite discrete distributions
- Parallel execution of simulations

A Trace-based Model for Multi-Party Contracts[1]

Tom Hvitved
hvitved@diku.dk

DIKU-IST Workshop 2011
January 11, 2011

[1] Joint work with Felix Klaedtke and Eugen Zălinescu

Background

- Multi-party contract: legally binding agreement between individuals or companies that describes the commitments of each contract participant.
- For enterprises: contracts serve as the external interface to their clients. Consequently crucial to monitor execution of contracts for breaches, and to comply with them.
- Aberdeen Group studies: "the average savings of transactions that are compliant with contracts is 22%".

Example

- **Paragraph 1.** Seller agrees to transfer and deliver to Buyer, on or before 2011-01-01, the goods: 1 laser printer.
- **Paragraph 2.** Buyer agrees to accept the goods and pay a total of €200 for them according to the terms further set out below.
- **Paragraph 3.** Buyer agrees to pay for the goods half upon receipt, with the remainder due within 30 days of delivery.
- **Paragraph 4.** If Buyer fails to pay the second half within 30 days, an additional fine of 10% has to be paid within 14 days.
- **Paragraph 5.** Upon receipt, Buyer has 14 days to return the goods to Seller in original, unopened packaging. Within 7 days thereafter, Seller has to repay the total amount to Buyer.

Trace-based Contract Model

- Abstract model for contracts.
- Contracts are *driven* by events.
- Depending on the sequence of events, the outcome of a contract can either be fulfillment or non-fulfillment.
- Such outcome may be *suggestive* of an obligation or a deadline not being met—but the model does not rely on such high-level notions.
- (Deterministic) blame assignment.
- Fundamental breaches: first breach matters.
- Compositionality (contract conjunction, contract disjunction).

Definitions (I)

- P: contract parties, A: actions, Ts: timestamps (for simplicity, Ts $:= \mathbb{N}$).
- Event: $\epsilon = (\tau, \alpha)$, where $\tau \in$ Ts, $\alpha \in$ A.
- ts$((\tau, \alpha)) := \tau$.
- Trace: finite or infinite sequence of events that satisfies the following properties:
  (1) timestamps are non-decreasing, that is, ts$(\sigma[i]) \leq$ ts$(\sigma[j])$, for all integers $i, j \in \mathbb{N}$ with $0 \leq i \leq j < |\sigma|$.
  (2) infinite traces have *progress*, that is, if $|\sigma| = \infty$ then for all timestamps $\tau \in$ Ts, there is an integer $i \in \mathbb{N}$ such that ts$(\sigma[i]) \geq \tau$.
- Example: $(\text{2011-01-01}, \mathrm{delivery})(\text{2011-01-01}, \mathrm{payment})$.

Definitions (II)

- Intuitively: contract is a subset of traces.
- BUT: we generalize the traditional binary outcome to incorporate blame assignment.
- Verdicts: V $:= \{\top\} \cup \{(\tau, B) \mid \tau \in$ Ts and $B \subseteq_{\text{fin}}$ P$\}$.
- Contract conformance, $\top$, is not associated with a point in time.
- Require that all breaches be associated with a *finite* point in time.
- $|B| > 1$: simultaneous breach, e.g., a barter deal.

Definitions (III)

- A contract between parties $P \subseteq_{\text{fin}} \mathsf{P}$: a function $\mathsf{c} : \mathsf{Tr} \to \mathsf{V}$, where

  if $\mathsf{c}(\sigma) = (\tau, B)$ then $B \subseteq P$,
  if $\mathsf{c}(\sigma) = (\tau, B)$ then $\mathsf{c}(\sigma') = (\tau, B)$, for all $\sigma' \in \mathsf{Tr}$ with $\sigma_\tau = \sigma'_\tau$.

- (σ_τ is restriction of σ to events with time stamp at most τ.)
- Contracts are deterministic.
- Traces are considered *complete*. Do not model: origin of events, agreement of events.
- Result: Verdict on infinite traces uniquely determined from verdicts on finite traces (contract conformance is a safety property).

Example

- Paragraph 1 as a contract, $\mathsf{c}_1 : \mathsf{Tr} \to \mathsf{V}$ between $\{\mathrm{Seller}\}$:

$$\mathsf{c}_1(\sigma) := \begin{cases} \top & \text{if } \sigma[i] = (\tau, \mathrm{delivery}), \\ & \quad \text{for some } 0 \le i < |\sigma| \text{ and } \tau \le \tau_d, \\ (\tau_d, \{\mathrm{Seller}\}) & \text{otherwise.} \end{cases}$$

- $\tau_d := 2011\text{-}01\text{-}01$.
- $\mathsf{c}_1(\varepsilon) = (2011\text{-}01\text{-}01, \{\mathsf{Seller}\})$.
- $\mathsf{c}_1((2011\text{-}01\text{-}01, \mathrm{delivery})) = \top$.

Contract Conjunction

- Present in virtually all contracts.
- Assume ν_i is verdict of c_i, $i = 1, 2$.
- Combined verdict:

$$\nu_1 \wedge \nu_2 := \begin{cases} \top & \text{if } \nu_1 = \nu_2 = \top, \\ \nu_1 & \text{if either } \nu_2 = \top, \\ & \quad \text{or } \nu_1 = (\tau_1, B_1),\ \nu_2 = (\tau_2, B_2), \text{ and } \tau_1 < \tau_2, \\ \nu_2 & \text{if either } \nu_1 = \top, \\ & \quad \text{or } \nu_1 = (\tau_1, B_1),\ \nu_2 = (\tau_2, B_2), \text{ and } \tau_1 > \tau_2, \\ (\tau, B) & \text{if } \nu_1 = (\tau, B_1),\ \nu_2 = (\tau, B_2), \text{ and } B = B_1 \cup B_2. \end{cases}$$

- (Fundamental breaches.)
- $(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma) := \mathsf{c}_1(\sigma) \wedge \mathsf{c}_2(\sigma)$.
- Result: c_1 (c_2) contract between parties P_1 (P_2) then $\mathsf{c}_1 \wedge \mathsf{c}_2$ contract between parties $P_1 \cup P_2$.

Contract Disjunction

- Assume ν_i is verdict of c_i, $i = 1, 2$, and that if $\nu_1 = (\tau, B_1)$ and $\nu_2 = (\tau, B_2)$ then $B_1 = B_2$ (deterministic blame assignment).
- Combined verdict:

$$\nu_1 \vee \nu_2 := \begin{cases} \top & \text{if } \nu_1 = \top \text{ or } \nu_2 = \top, \\ (\tau_1, B_1) & \text{if } \nu_1 = (\tau_1, B_1), \nu_2 = (\tau_2, B_2) \text{ and } \tau_1 > \tau_2, \\ (\tau_2, B_2) & \text{if } \nu_1 = (\tau_1, B_1), \nu_2 = (\tau_2, B_2) \text{ and } \tau_1 < \tau_2, \\ (\tau, B) & \text{if } \nu_1 = \nu_2 = (\tau, B). \end{cases}$$

- Inherently non-deterministic operator.
- $(\mathsf{c}_1 \vee \mathsf{c}_2)(\sigma) := \mathsf{c}_1(\sigma) \vee \mathsf{c}_2(\sigma)$.
- Result: c_1 (c_2) contract between parties P_1 (P_2) then $\mathsf{c}_1 \vee \mathsf{c}_2$ contract between parties $P_1 \cup P_2$.

Contract Composition

- $(C, \vee, \wedge)$ is a distributive lattice with unit element $\mathsf{c}_\top$, provided unique blame assignment.

$$\begin{aligned}
\nu_1 \wedge \nu_2 &= \nu_2 \wedge \nu_1 && \text{(Commutativity)} \\
\nu_1' \vee \nu_2' &= \nu_2' \vee \nu_1' && \text{(Commutativity)} \\
\nu_1 \wedge (\nu_2 \wedge \nu_3) &= (\nu_1 \wedge \nu_2) \wedge \nu_3 && \text{(Associativity)} \\
\nu_1' \vee (\nu_2' \vee \nu_3') &= (\nu_1' \vee \nu_2') \vee \nu_3' && \text{(Associativity)} \\
\nu_1' \vee (\nu_1' \wedge \nu_2') &= \nu_1' && \text{(Absorption)} \\
\nu_1' \wedge (\nu_1' \vee \nu_2') &= \nu_1' && \text{(Absorption)} \\
\nu_1' \vee (\nu_2' \wedge \nu_3') &= (\nu_1' \vee \nu_2') \wedge (\nu_1' \vee \nu_3') && \text{(Distributivity)} \\
\nu_1 \wedge (\nu_2' \vee \nu_3') &= (\nu_1 \wedge \nu_2') \vee (\nu_1 \wedge \nu_3') && \text{(Distributivity)} \\
\top \wedge \nu &= \nu \wedge \top = \nu && \text{(Unit)} \\
\top \vee \nu &= \nu \vee \top = \top && \text{(Unit)}
\end{aligned}$$

Run-time Monitoring

- Run-time monitor: partial, finite traces. Most be computable.
- Many-valued semantics: $\mathsf{V}_\star := \{\nu_\star \mid \nu \in \mathsf{V}\}$, $\star \in \{!, ?\}$.
- $\mathsf{mon} : \mathsf{Tr}_{\mathrm{fin}} \to \mathsf{V}_! \cup \mathsf{V}_?$ that satisfies

$$\mathsf{mon}(\sigma) = \begin{cases} \top_! & \text{if } \mathsf{c}(\sigma') = \top \text{ whenever } \sigma \sqsubset \sigma', \\ (\tau, B)_! & \text{if } \mathsf{c}(\sigma') = (\tau, B) \text{ whenever } \sigma \sqsubset \sigma', \\ \top_? & \text{if } \mathsf{c}(\sigma) = \top \text{ and } \mathsf{c}(\sigma') \neq \top \text{ for some } \sigma \sqsubset \sigma', \\ (\tau, B)_? & \text{if } \mathsf{c}(\sigma) = (\tau, B) \text{ and } \mathsf{c}(\sigma') \neq (\tau, B) \text{ for some } \sigma \sqsubset \sigma'. \end{cases}$$

- Impartiality and anticipation.
- Previous example:
  - $\mathsf{mon}(\varepsilon) = (\text{2011-01-01}, \mathrm{Seller})_?$.
  - $\mathsf{mon}((\text{2011-01-01}, \mathrm{delivery})) = \top_!$.
  - $\mathsf{mon}((\text{2011-01-02}, \mathrm{delivery})) = (\text{2011-01-01}, \mathrm{Seller})_!$.

A Contract Specification Language

- Writing contracts directly in the abstract model is cumbersome.
- CSL provides syntax for writing contracts.
- CSL specifications denote contracts via a reduction semantics $\Rightarrow$ deterministic blame assignment + compositionality.
- CSL supports (a) history sensitive contracts, (b) conditional commitments, (c) contract templates, (d) absolute and relative temporal constraints, (e) reparation clauses, (f) in-place arithmetic, and (g) potentially infinite contracts.

A Contract Specification Language

- Signature: $S = (\mathcal{K}, \mathsf{ar}, \mathcal{T})$, $\mathsf{ar} : \mathcal{K} \to \mathcal{T}^*$.
- Actions refined:
  $A := \{k(\vec{v}) \mid k \in \mathcal{K}, \mathsf{ar}(k) = (t_1, \ldots, t_n), \vec{v} \in [\![t_1]\!] \times \cdots \times [\![t_n]\!]\}$.
- Syntax:

$$
\begin{array}{rcl}
s & ::= & \textbf{letrec } \{f_i(\vec{x}_i)\langle \vec{y}_i \rangle = c_i\}_{i=1}^{n} \textbf{ in } c \textbf{ starting } e \\
c & ::= & \textbf{fulfillment} \\
 & \mid & \langle e_1 \rangle\ k(\vec{x}) \textbf{ where } e_2 \textbf{ due } d \textbf{ remaining } z \textbf{ then } c \\
 & \mid & \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \textbf{ remaining } z \textbf{ then } c_1 \textbf{ else } c_2 \\
 & \mid & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \\
 & \mid & c_1 \textbf{ and } c_2 \\
 & \mid & c_1 \textbf{ or } c_2 \\
 & \mid & f(\vec{e}_1)\langle \vec{e}_2 \rangle \\
e & ::= & x \mid v \mid \neg e \mid e_1 \star e_2 \mid e_1 \prec e_2 \\
d & ::= & \textbf{after } e_1 \textbf{ within } e_2
\end{array}
$$

Example, revisited

letrec *sale*(*deliveryDeadline*, *goods*, *payment*)⟨*buyer*, *seller*⟩ =
⟨*seller*⟩ *Deliver*(*s*,*r*,*g*) **where** $s = seller \wedge r = buyer \wedge g = goods$
due within *deliveryDeadline*
then
⟨*buyer*⟩ *Payment*(*s*,*r*,*a*) **where** $s = buyer \wedge r = seller \wedge a = payment/2$
due immediately
then
((⟨*buyer*⟩ *Payment*(*s*,*r*,*a*) **where** $s = buyer \wedge r = seller \wedge a = payment/2$
due within $30D$
or
⟨*buyer*⟩ *Payment*(*s*,*r*,*a*) **where** $s = buyer \wedge r = seller \wedge a = (payment/2) * 110/100$
due within $14D$ **after** $30D$)
and
if *Return*(*s*,*r*,*g*) **where** $s = buyer \wedge r = seller \wedge g = goods$ **due within** $14D$ **then**
⟨*seller*⟩ *Payment*(*s*,*r*,*a*) **where** $s = seller \wedge r = buyer \wedge a = payment$
due within $7D$)
in
sale(0, `"Laser printer"`, 200)⟨Buyer, Seller⟩ **starting** 2011−01−01

Some Notes on Semantics (I)

- Type system:
  - Δ: template environment, Λ: party environment, Γ: variable environment.
  - $\Delta, \Lambda, \Gamma \vdash c : \mathrm{Clause}\langle P \rangle$.
  - $\vdash s : \mathrm{Contract}\langle P \rangle$.

$$\frac{\begin{array}{ccc}\Gamma' := \Gamma[\vec{x} \mapsto \mathsf{ar}(k)] & \Lambda_1 \vdash e_1 : P_1 & \\ \Gamma_2 := \Gamma'[z \mapsto \mathrm{Int}] & \Gamma' \vdash e : \mathrm{Bool} & \\ & \Gamma \vdash d : \mathrm{Deadline} & \Delta, \Lambda_2, \Gamma_2 \vdash c : \mathrm{Clause}\langle P_2 \rangle\end{array}}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash \langle e_1 \rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c : \mathrm{Clause}\langle P_1 \cup P_2 \rangle}$$

$$\frac{|\Lambda_1 \cup \Lambda_2| + |P_1 \cup P_2| \leq 1 \qquad \Delta, \Lambda_1, \Gamma \vdash c_1 : \mathrm{Clause}\langle P_1 \rangle \qquad \Delta, \Lambda_2, \Gamma \vdash c_2 : \mathrm{Clause}\langle P_2 \rangle}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash c_1\ \textbf{or}\ c_2 : \mathrm{Clause}\langle P_1 \cup P_2 \rangle}$$

Some Notes on Semantics (II)

- Reduction semantics:
  - $D, \tau \vdash c \xrightarrow{\epsilon} \mathbf{c}$, $\mathbf{c}$ either c' or (τ, B).

$$\frac{e[\vec{v}/\vec{x}] \Downarrow \textbf{true} \qquad d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau_1 \leq \tau' \leq \tau_2}{D, \tau \vdash \langle p \rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c \xrightarrow{(\tau', k(\vec{v}))} c[\vec{v}/\vec{x}, \tau_2 - \tau'/z]}$$

$$\frac{d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau' > \tau_2}{D, \tau \vdash \langle p \rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c \xrightarrow{(\tau', k'(\vec{v}))} (\max(\tau, \tau_2), \{p\})}$$

  - $s \xrightarrow{\epsilon} \mathbf{s}$, $\mathbf{s}$ either s' or (τ, B).

$$\frac{e \Downarrow \tau \qquad D, \tau \vdash c \xrightarrow{\epsilon} (\tau', B)}{\textbf{letrec}\ D\ \textbf{in}\ c\ \textbf{starting}\ e \xrightarrow{\epsilon} (\tau', B)}$$

$$\frac{e \Downarrow \tau \qquad D, \tau \vdash c \xrightarrow{\epsilon} c' \qquad \mathsf{ts}(\epsilon) = \tau'}{\textbf{letrec}\ D\ \textbf{in}\ c\ \textbf{starting}\ e \xrightarrow{\epsilon} \textbf{letrec}\ D\ \textbf{in}\ c'\ \textbf{starting}\ \tau'}$$

Some Notes on Semantics (III)

- Progress: if $\vdash s : \mathrm{Contract}\langle P \rangle$ then for all events ϵ, there is a unique residue $\mathbf{s}$ such that $s \xrightarrow{\epsilon} \mathbf{s}$. Furthermore, whenever $\mathbf{s} = (\tau, B)$ then $B \subseteq P$.
- Preservation: if $\vdash s : \mathrm{Contract}\langle P \rangle$ and $s \xrightarrow{\epsilon} s'$ then $\vdash s : \mathrm{Contract}\langle P' \rangle$ with $P' \subseteq P$.
- Result: if $\vdash s : \mathrm{Contract}\langle P \rangle$ then $[\![s]\!]$ is a contract between parties P.
- ($[\![\cdot]\!]$ defined using $s \xrightarrow{\epsilon} s'$.)

Some Notes on Semantics (IV)

- Reduction semantics gives rise to incremental run-time monitoring.
- But full anticipation not supported (expression language may "hide" anticipated verdicts).

Summary

- Abstract, trace-based contract model. Focus: blame assignment.
- Abstract definition of contract conjunction, contract disjunction, and run-time monitoring.
- Contract specification language, CSL.
- Incremental run-time monitoring of CSL specifications.
- Various "real-world" contracts formalized in CSL.

Future Work

- Fragments of CSL: non-negative deadlines, fulfillable obligations.
- Full run-time monitoring.
- Contract analysis:
  - Satisfiability (related to full run-time monitoring).
  - Satisfiability w.r.t. a particular party.
  - Contract portfolios.
  - Non-deterministic semantic model: determinism $\Rightarrow$ decision procedure.
- Other applications (web services, network protocols)?
- (Implementation.)

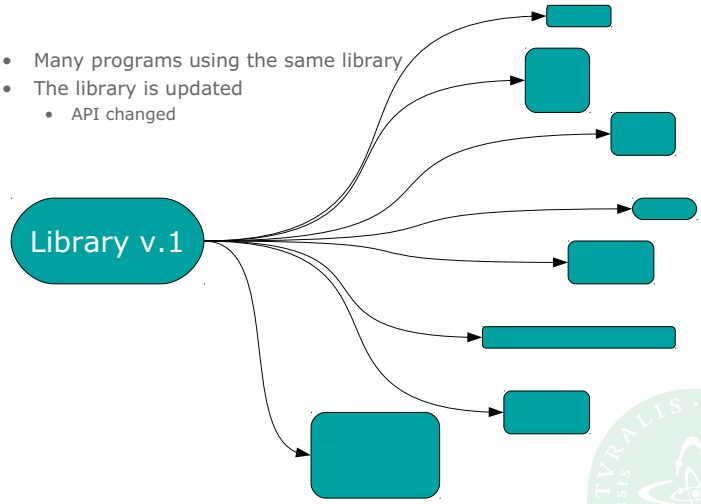
Semantic Patch Inference

Jesper Andersen
University of (wonderful) Copenhagen
Computer Science Department (DIKU)

Joint work with
Julia Lawall
David Lo
Siau-Cheng Khoo

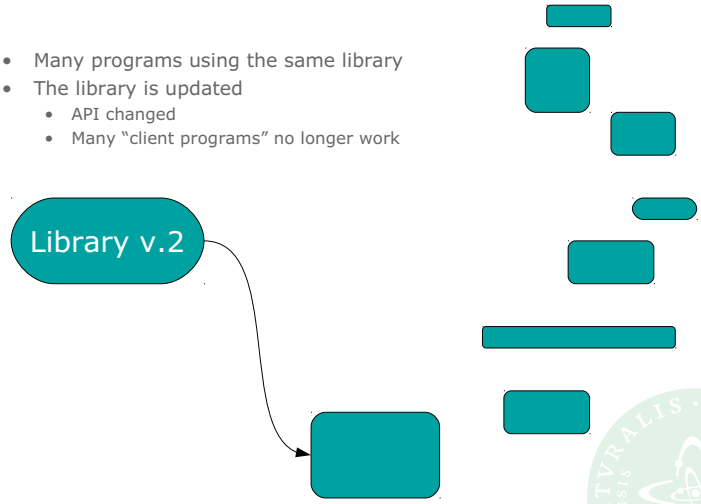
Context : collateral evolution

- Many programs using the same library
- The library is updated
  - API changed



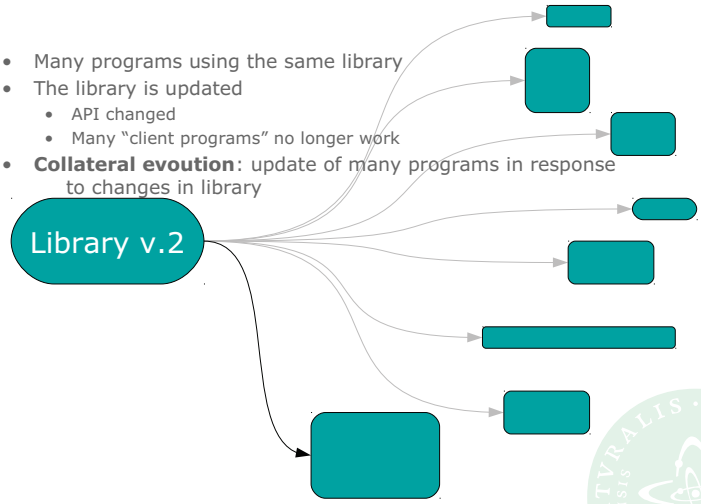
Context : collateral evolution

- Many programs using the same library
- The library is updated
  - API changed
  - Many "client programs" no longer work



Context : collateral evolution

- Many programs using the same library
- The library is updated
  - API changed
  - Many "client programs" no longer work
- **Collateral evoution**: update of many programs in response to changes in library

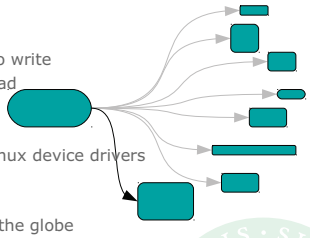


Problem : getting it right in the first place

- Performing a collateral evolution by hand is …

… tedious
- Many locations have to be updated in similar fashion

… error-prone
- Humans are not very good automatons
- Tricky reg-exp. based scripts are hard to write
  - Even more difficult for others to read



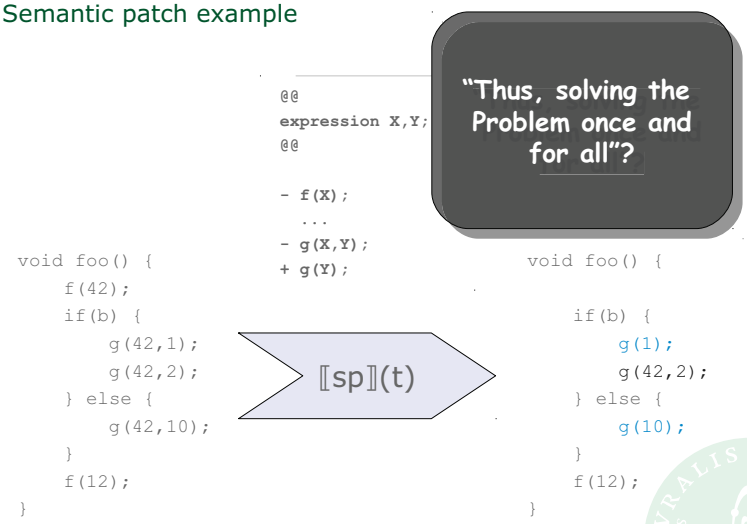
… done very frequently in development of Linux device drivers
- Often done mostly correct
- Varied development environments
  - Developers are distributed around the globe

Coccinelle

- A program transformation and query engine
- Used for
  - Collateral evolution of Linux device drivers
  - Bug finding & fixing in systems code
- Provides the specification language **SmPL**
  - WYSIWYG approach
  - Abstract irrelevant details
  - Control-flow sensitive

```
@@
expression X,Y;
@@

- f(X);
  ...
- g(X,Y);
+ g(Y);
```

Semantic patch example



Problem : who will write the semantic patches?

The library developer

- Real code is more natural to work with
- Gap between library evolution and driver collateral evolution
- In what context do the changes (not) apply?

"I wish I *could* write a semantic patch"

The driver developer

- Real code updates are too low level
- What details are (not) important?

"If only someone *had* written a semantic patch"

The contribution : `spdiff`

- A method for automatic inference of semantic patches from example updates
  - Library developers can work on real code...
  - Driver developers can apply `spdiff` to real updates...
- A formalization of subpatches
  - Ensures that inferred patches "do the right thing"
  - Gives higher-level description of changes than standard diff
- Implemented in the tool "`spdiff`"
  - Applicable to "historic" data from Linux revision system

Simple example : "I wish I could write a semantic patch"

```
int foo(int a) {
  int x = 42;
  x = bar();
  foz(x);
  bar();
  return x;
}
```

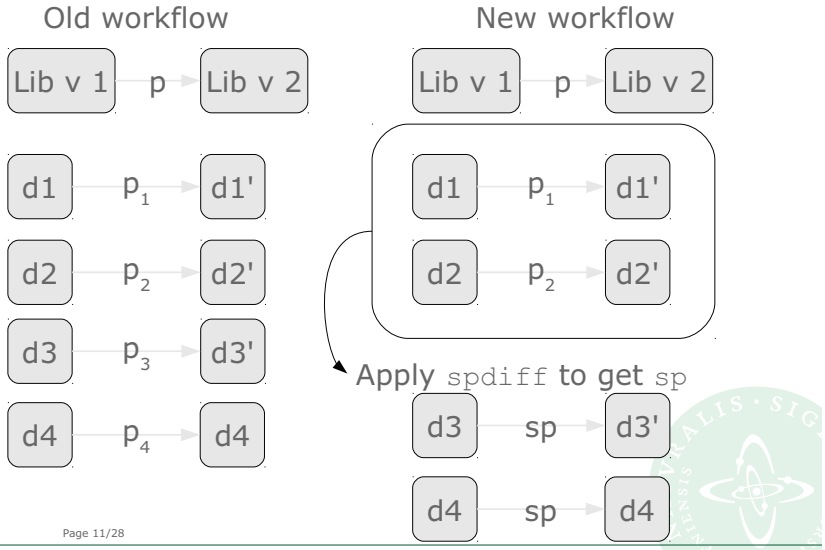
```
@@
expression X0;
@@
  int X0 = X1;
  ...
+ log_assign(X0);
  X0=bar();
+ returned(X0);
  ...
  return X0;
```

```
int foo(int a) {
  int x = 42+42;
  log_assign(x);
  x = bar();
  returned(x);
  foz(x);
  bar();
  return x;
}
```

```
int foo(int *q) {
  int y = *q + *q;
  *q = bar();
  y = bar();
  return y;
}
```

```
int foo(int *q) {
  int y = *q + *q;
  *q = bar();
  log_assign(y);
  y = bar();
  returned(y);
  return y;
}
```


Library devs. can work on (less) real code...



Driver devs.: "I wish someone had written a semantic patch"

```
commit 0723d4a8064d52ae44ef2e19749a190b08846be3
Author: FUJITA Tomonori <fujita.tomonori@lab.ntt.co.jp>
Date:   Thu Oct 9 11:25:42 2008 +0900

    [SCSI] 3w-xxxx: remove unnecessary local_irq_save/restore for scsi
    sg copy API

    Since the commit 50bed2e2862a8f3a4f7d683d0d27292e71ef18b9 (sg:
    disable interrupts inside sg_copy_buffer), no need to disable
    interrupts before calling scsi_sg_copy_from_buffer. So we can
    simplify tw_transfer_internal, which disables interrupts just for
    scsi_sg_copy_from_buffer.
```

"If only someone had written a semantic patch"

Patch **understanding**

Driver devs.: "I wish someone had written a semantic patch"

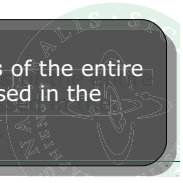
```
 drivers/scsi/3w-xxxx.c |    7 +------
 1 files changed, 1 insertions(+), 6 deletions(-)

diff --git a/drivers/scsi/3w-xxxx.c b/drivers/scsi/3w-xxxx.c
index a0537f0..c03f1d2 100644
--- a/drivers/scsi/3w-xxxx.c
+++ b/drivers/scsi/3w-xxxx.c
@@ -1466,12 +1466,7 @@ static int tw_scsiop_inquiry(TW_Device_Extension *tw_dev,
     int request_id)
 static void tw_transfer_internal(TW_Device_Extension *tw_dev, int request_id,
                                  void *data, unsigned int len)
 {
-       struct scsi_cmnd *cmd = tw_dev->srb[request_id];
-       unsigned long flags;
-
-       local_irq_save(flags);
-       scsi_sg_copy_from_buffer(cmd, data, len);
-       local_irq_restore(flags);
+       scsi_sg_copy_from_buffer(tw_dev->srb[request_id], data, len);
 }

 /* This function is called by the isr to complete an inquiry command */
```


Driver devs.: "I wish someone had written a semantic patch"

```
@@
expression X0;
@@
- unsigned long X0;
  ...
- local_irq_save(X0);
  ...
  scsi_sg_copy_from_buffer(X1,X2,X3);
  ...
- local_irq_restore(X0);
```

```
 drivers/scsi/3w-xxxx.c |
 1 files changed, 1 insertio

diff --git a/drivers/scsi/3
index a0537f0..c03f1d2 100644
--- a/drivers/scsi/3w-xxxx.c
+++ b/drivers/scsi/3w-xxxx.c
@@ -1466,12 +1466,7 @@ stat
     int request_id)
 static void tw_transfer_internal(TW_Device_Extension *tw_dev, int request_id,
                                  void *data, unsigned int len)
 {
-       struct scsi_cmnd *cmd = tw_dev->srb[request_id];
-       unsigned long flags;
-
-       local_irq_save(flags);
-       scsi_sg_copy_from_buffer(cmd, data, len);
-       local_irq_restore(flags);
+       scsi_sg_copy_from_buffer(tw_dev->srb[request_id], data, len);
 }

 /* This function is called by the isr t
```

Only the **common parts** of the entire transformation is expressed in the inferred semantic patch.



Formalization of subpatches

Main concept:

Transformation part:

$sp \preccurlyeq (t,t')$

sp is part of turning t into t'

~

"does the right thing w.r.t. (t,t')"

spdiff finds a semantic patch that "does the right thing"

Given a set of examples $C = \{(t_1,t_1'), \ldots, (t_n,t_n')\}$

- Find the sp such that

1) sp does "the right thing" w.r.t. C

Denoted : $sp \preccurlyeq C \Leftrightarrow \forall (t,t') \in C : sp \preccurlyeq (t,t')$

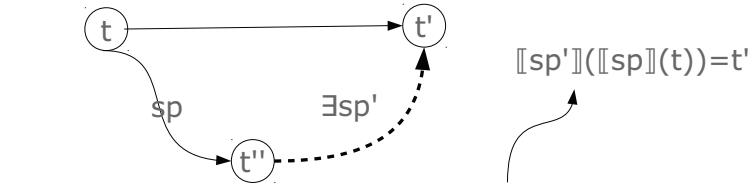
2) sp expresses as much of the common changes as possible

$\forall sp' : sp' \preccurlyeq C \Rightarrow sp' \preccurlyeq sp$

Doing the right thing

Transformation part:
sp ≼ (t,t') ⇔ "sp does the right thing" (???)

First suggestion: (t,t') can be **decomposed** into sp and sp'



⟦sp'⟧(⟦sp⟧(t))=t'

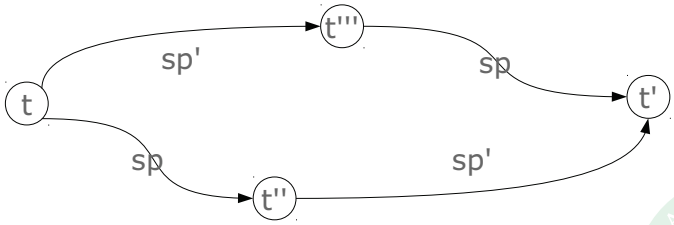
We need to restrict sp'
or else ∀sp : sp ≼ (t,t')

sp' should not "undo"
things done by sp

Doing the right thing (cont'd)

Transformation part:
sp ≼ (t,t') ⇔ "commutative parts"

Second suggestion: (t,t') can be decomposed into sp and sp'
and they commute with each other

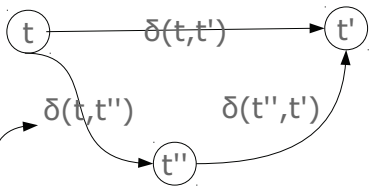


⟦sp⟧(⟦sp'⟧(t)) = ⟦sp'⟧(⟦sp⟧(t)) = t'

Doing the right thing (final)

Transformation part:
sp ≼ (t,t') ⇔ ⟦sp⟧(t) = t'' & δ(t,t'') + δ(t'',t') = δ(t,t')

Third suggestion: "do as little work as necessary"



δ(t,t'')+δ(t'',t') = δ(t,t')

δ(a,b) : some tree edit distance (metric)

"If a subterm changed, changing it again is
more costly than changing it only once"

Formalization of subpatches

What is it that "spdiff" do? How does it work?

Given a set of examples, C = {(t1,t1'), ..., (tn,tn')}

"Finds largest common transformation"

Find the sp such that:

1) sp does "the right thing" for each pair (t,t') ∈ C

   Denoted : $sp \preccurlyeq C \Leftrightarrow \forall (t,t') \in C : sp \preccurlyeq (t,t')$

2) sp expresses as much of the common changes as possible (subsumption relation)

   $\forall sp' : sp' \preccurlyeq C \Rightarrow sp' \preccurlyeq sp$

2 algorithms have been implemented...

2 algorithms for semantic patch inference

Context-free patches

Notation

gp ::= p ⇝ p | gp;gp

Application

⟦p⇝p'⟧(t) = replace p with p'

⟦gp;gp'⟧(t) = ⟦gp'⟧(⟦gp⟧(t))

Example

```
seq_path(X1,
         X2->X3.mnt,
         X2->X3.dentry,
         X4)
⇝
seq_path(X1,&X2->X3,X4)
```

Context-sensitive patches

Notation

sp ::= O sp | ε

O ::= p | … | -p | +p

Application

Example

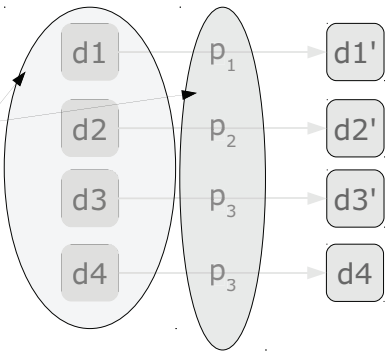
```
- unsigned long X8;
  ...
- local_irq_save(X8);
  ...
  scsi_sg_copy_from_buffer
             (X5,X6,X7);
  ...
- local_irq_restore(X8);
```


Algorithm for context-sensitive patches

Algorithm

Three main parts

1. Identification of changes
   - ➔ Graph differences
2. Inference of common characteristics
   - ➔ Context-sensitive patterns
3. Construction of context-sensitive patches
   - ➔ Iteratively grow patches from patterns and diffs



Growing semantic patches

Identify changes

```
int foo(int a) {
  int x = 42;
  x = bar();
  foz(x);
  bar();
  return x;
}
```

```
- int x = 42;
+ int x = 42+42;
+ log_assign(x);
  x = bar();
+ returned(x);
  foz(x);
  bar();
  return x;
}
```

```
int foo(int a) {
int x = 42+42;
log_assign(x);
x = bar();
returned(x);
foz(x);
bar();
return x;
}
```

Chunkify...

+p*(p|-p)+p*

```
- signed int x = 42;
+ int x = 42 + 42;

+ log_assign(x);
  x = bar();
+ returned(x);

+ returned(x);
  foz(x);
```

Page 23/28

Find context-sensitive patterns

2 steps:

- Find common node patterns
- **Grow** patterns from node patterns

```
int X0 = X1;
X2 = bar();
return X3;
```

```
int foo(int a) {
  int x = 42;
  x = bar();
  foz(x);
  bar();
  return x;
}
```

```
int foo(int *q) {
  int y = *q + *q;
  *q = bar();
  y = bar();
  return y;
}
```

Page 24/28

Find context-sensitive patterns

Grow pattern refine...

```
X2 = bar();
```

```
int X0 = X1;
...
X2 = bar();
...
return X3;
```

```
int X0 = X1;
...
X0 = bar();
...
return X0;
```

```
return X3;
```

```
int foo(int a) {
  int x = 42;
  x = bar();
  foz(x);
  bar();
  return x;
}
```

```
int foo(int *q) {
  int y = *q + *q;
  *q = bar();
  y = bar();
  return y;
}
```

Page 25/28

Construction of patches...

```
- signed int x = 42;
+ int x = 42 + 42;


+ log_assign(x);
  x = bar();
+ returned(x);


+ returned(x);
  foz(x);
```

```
- int X0 = X1;
+ int X0 = X1 + X1;
  X0 = bar();
  X0 = bar();
  return X0;
  return X0;
```

```
int foo(int *q) {
  int y = *q + *q;
  *q = bar();
  y = bar();
  return y;
}
```

Wrong:
One decl. did not change!

Construction of patches...

```
- signed int x = 42;
+ int x = 42 + 42;


+ log_assign(x);
  x = bar();
+ returned(x);


+ returned(x);
  foz(x);
```

```
  int X0 = X1;
  ...
+ log_assign(X0);
  X0 = bar();
+ returned(X0);
  ...
  return X0;
```

Maximality:
Use subsumption relation to return only largest...

Summary

In this talk

- Short description of collateral evolution
- Simple definition of "transformation parts"
  - Independent of transformation language
- Use of transformation parts to implement semantic patch inference

Not in this talk / future work

- Dealing with actual real-world changesets
- Bug-hunting
  - Searching for common contexts for backports
- Integration in IDEs
- Parameterize over subject language
- Relate to …
  - Semi-inversion
  - Logic programming

A Short Review: Left Inverses vs. Right Inverses

Kazutaka Matsuda (Tohoku University)
Jan. 11, 2011
DIKU-IST Workshop

This Talk

- A short review on inverse computation
- To tell difference between
  - left inverse construction
    - left inverse: what we call inverse
  - right inverse construction
    - right inverse: output -> a possible input
- To address issues in right inverse construction

(Left) Inverse

Definition

g is called a left inverse of f if
f(x) = y implies g(y) = x for every x and y

- Only injective function has a left inverse.
- e.g.)
  - inc is a left inverse of dec
    ← inc(dec(x)) = inc(x-1) = x

inc(x) = x +1
dec(x) = x - 1

Left Inverses are Useful

- Deserialization obtained from serialization
  - classic example: runlength
- Derivation of bidirectional transformation [Matsuda+07]

4

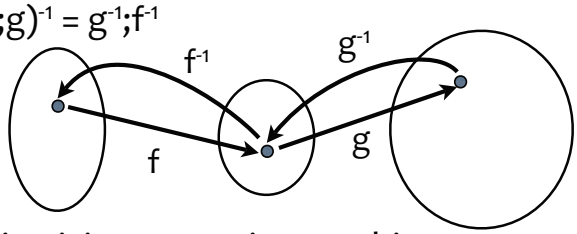
Approaches to Left Invs

- Compositional approach
- Derivation approach

5

Compositional Construction

- Compositional approach to left-inverse [Mu+04][Foster+05,07]
  - $(f;g)^{-1} = g^{-1};f^{-1}$

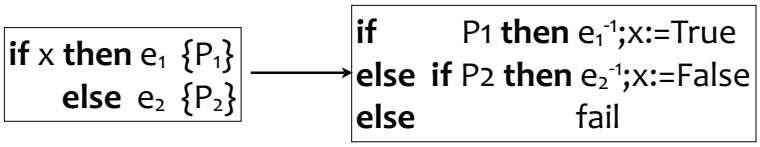


  - Injectivity-preserving combinators ×, +, ...
  - Predefined primitives

6

Derivational Construction

- Several known methods
  - [Matsuda+10] [Nishida&Sakai09,10] [Glück&Kawabe04,05][Glück&Kawabe05] [Mogensen05] ... [Gries81]
- Injectivity analysis for correctness and for efficiency



7

Short Summary

- Left inverse construction
  - Compositional: Injectivity
  - Derivation: Injectivity analysis
  - Complementary

8

Dual story: right inverses?

- Can we follow the successful stories about left-inverses?

Yes or No

9

Right Inverse

Definition

g is called a right inverse of f if
g(y) = x implies f(x) = y for every x and y

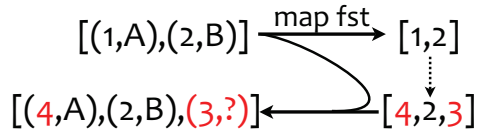
- Every function has (partial) right inverse.
- e.g.)
  - half(double(x)) = x
  - half(double'(x)) = x

half(x) = floor(x/2)
double(x) = 2*x
double'(x) = 2*x + 1

10

Right Inverses are Useful

- To handle lost values in bidirectional transformation [Foster+07,08, ...]



- Data abstraction [Wang+10]
  - Join list and list
- Derivation of parallel programs [Gibbons96][Morita+07][Morihata+09]

11

Rest of Talk

- Methods for right inverses
  - What are keys?
- Problems
- Conclusion

12

Construction of Right Inverses

- Compositional approaches
  [Wang+10][Foster+08;Bohannon+08]

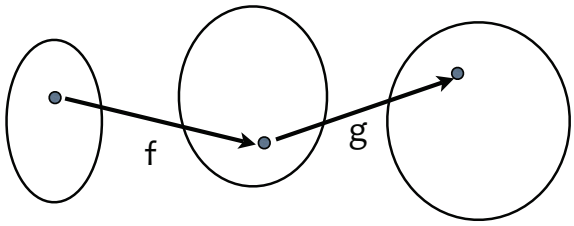
- Derivational approaches
  [Matsuda+10][Glück&Abramov02]
  [Morita+07]

13

Quiz

- f': a right-inverse of f
- g': a right-inverse of g

- Is (g';f') is a right inverse of (f;g)?

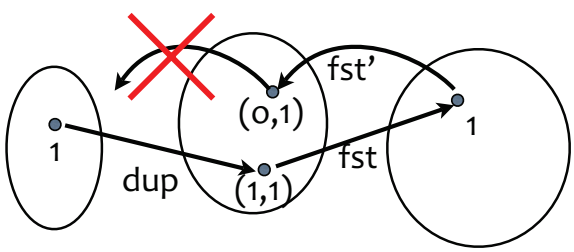


14

No!

- dup': a right-inverse of dup
- fst': a right-inverse of fst

dup(x) = (x,x)
dup'(x,x) = x
fst(x,y) = x
fst'(x) = (0,x)

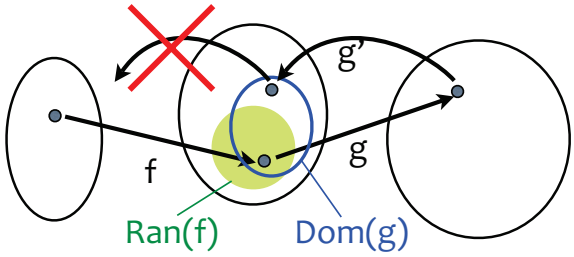


15

Reason to fail

- f': a right-inverse of f
- g': a right-inverse of g

- Ran(f) ⊈ Dom(g)

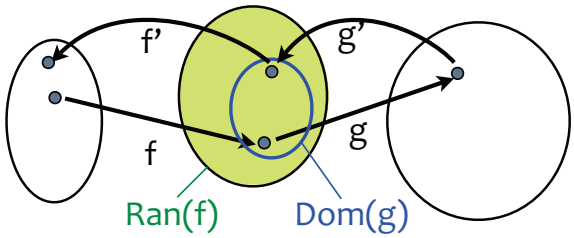


16

Surjectivity is a Key

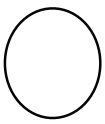
- f': a right-inverse of surjective f
- g': a right-inverse of surjective g

- (g';f') is a right inverse of (f;g).



17

Surjectivity in Existing Work

- [Wang+10] [Foster+08;Bohannon+08]
  - Compositional Composition
  - Surjectivity is guaranteed
    - via surjectivity-preserving combinators
      - including function compositon
  - Difference: Objects ( in prev slide)
    - [Wang+10]
      Algebraic Datatype
    - [Foster+08;Bohannon+08]
      Regular-set (with Quotient)

18

Derivational Construction

- [Matsuda+10]
- [Glück&Abramov02]
- [Morita+07]

19

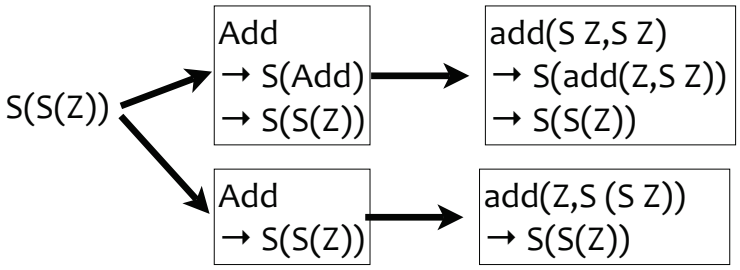
[Matsuda+10]

- ∃x.f(x) = v

```
add(Z,   y) = y
add(S x,y) = S(add(x,y))
```

∃xy.add(x,y) = v ⇔ Add →* v

```
Add → _
Add → S(Add)
```

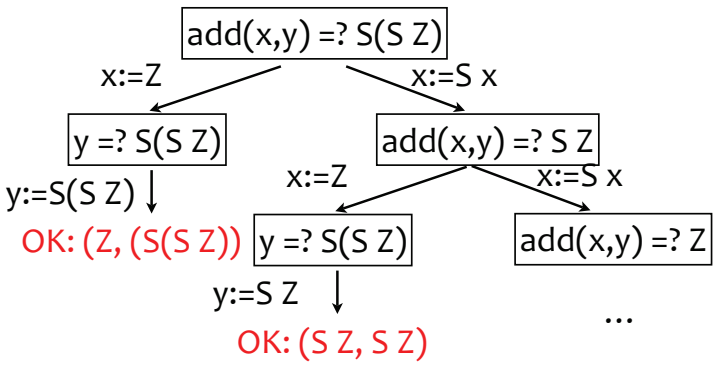


20

[Glück&Abramov02]

- ∃x.f(x) = v

```
add(Z,   y) = y
add(S x,y) = S(add(x,y))
```

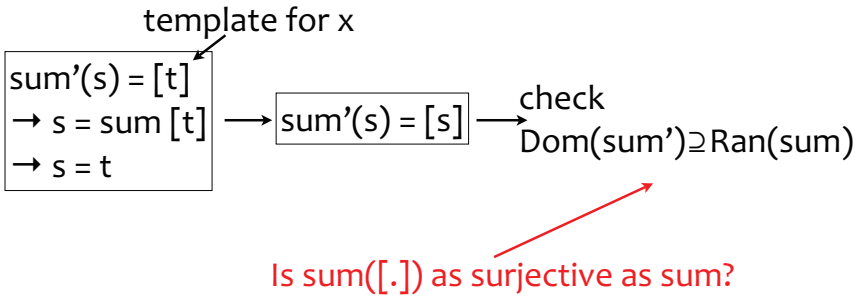


21

[Morita+06]

- ∃x.f(x) = v

```
sum [x]   = x
sum (a:x) = a + sum x
```



22

Candidate of Key Technique

- All depend on ∃x.f(x) = v
  - [Matsuda+10]
    - Any f is surjective to some regular set
    - Right inverse is given by automaton
  - [Glück&Abramov02]
    - Any f is surjective to some RE set
    - Right inverse does not necessary terminate
  - [Morita+07]
    - Surjectivity check
    - They asks whether still f([x]) is surjective

23

Summary

- Left inverse derivation
  - Compositional: Injectivity
  - Derivation: Injectivity analysis
- Right inverse derivation
  - Compositional: Surjectivity
  - Derivation: Surjectivity check or Exact Range?

24

Problems

- Left-inverse derivation uses injectivity analysis
- Little discussion on right-inverse derivation with surjectivity analysis?
  - [Morita+07] is notable exception

25

Problems

- Functions are rarely surjective
  - Objects (A, B of f :: A → B) are simple enough to check A=B
  - Range(f) is usually more complex
    - even Range(f) ⊇B is undecidable
- Derivation method is rarely useful to define primitives in compositional approach.
  - notable exceptions [Wang+10] [Foster+08;Bohannon+08]

26

Conclusion

- Left inverse construction
  - Compositional: Injectivity
  - Derivation: Injectivity analysis
  - Complementary
- Right inverse construction
  - Compositional: Surjectivity
  - Derivation: Surjectivity check or Exact Range?
  - Not complementary?

27

Towards a Software Model Checker for ML

Naoki Kobayashi
Tohoku University

Joint work with:
Ryosuke Sato and Hiroshi Unno (Tohoku University)
in collaboration with
Luke Ong (Oxford), Naoshi Tabuchi and Takeshi Tsukada (Tohoku)

This Talk

- Overview of our project on program verification, based on higher-order model checking (or, the model checking of higher-order recursion schemes)
  - What is higher-order model checking?
  - How are higher-order model checking problems solved?
  - How can software model checkers can be constructed on top of a higher-order model checker?

Goal: Software model checker for ML

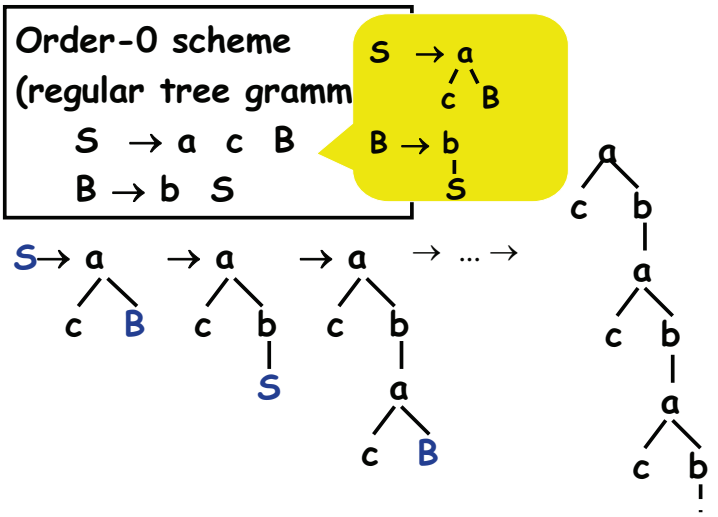
Outline

- Introduction to higher-order model checking
  - What are higher-order recursion schemes?
  - What are model checking problems?
  - How related to program verification?
- Model checking functional programs
  - Predicate abstraction and CEGAR
  - Automata-based abstractions for recursive data structures
- Discussion and conclusion

Higher-Order Recursion Scheme

♦Grammar for generating an infinite tree

Order-0 scheme
(regular tree gramm
S → a c B
B → b S

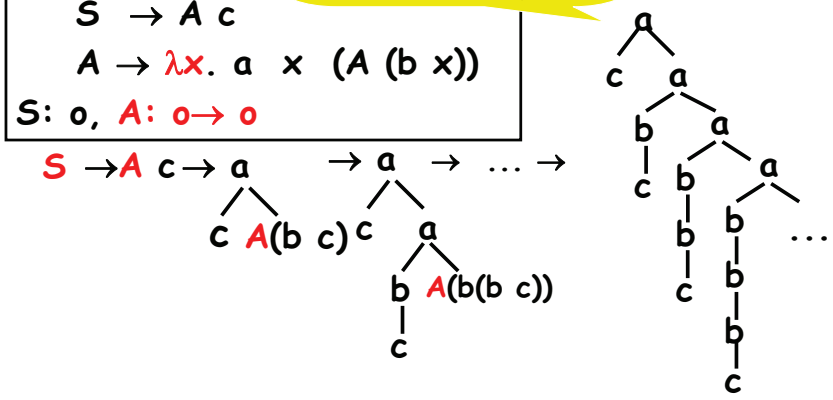


Higher-Order Recursion Scheme

♦Grammar for ...ite tree

Tree whose paths are labeled by $a^{m+1}\ b^m\ c$

Order-1 scheme
S → A c
A → λx. a x (A (b x))
S: o, A: o→ o



Higher-Order Recursion Scheme

♦Grammar for generating an infinite tree

Order-1 scheme
S → A c
A → λx. a x (A (b x))
S: o, A: o→ o

Higher-order recursion schemes
≈
Call-by-name simply-typed λ-calculus
+
recursion, tree constructors

Model Checking Recursion Schemes

Given
- G: higher-order recursion scheme
- A: alternating parity tree automaton (APT) (a formula of modal μ-calculus or MSO),

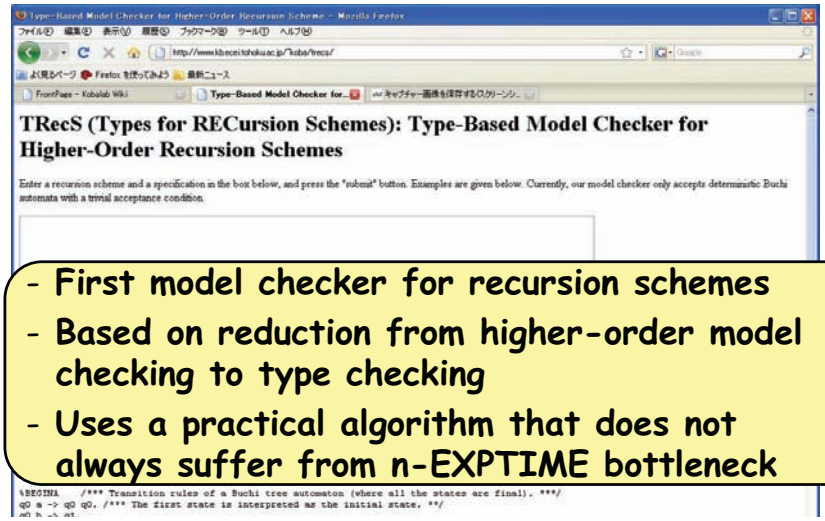
does A accept Tree(G)?

e.g.
- Does every finite path end with "c"?
- Does "a" occur eventually whenever "b" occurs?

n-EXPTIME-complete [Ong, LICS06] (for order-n recursion scheme) $n\left\{\begin{array}{l} 2^{p(x)} \\ 2^{2^{\cdot^{\cdot}}} \\ 2 \end{array}\right.$

TRecS [K., PPDP09]

http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/



- First model checker for recursion schemes
- Based on reduction from higher-order model checking to type checking
- Uses a practical algorithm that does not always suffer from n-EXPTIME bottleneck

Application: model-checking higher-order boolean programs

Theorem:
Given a closed term M of (call-by-name or call-by-value) simply-typed λ-calculus with:
- -recursion
- -finite base types (including booleans and constant "fail")
- -non-determinism,

it is decidable whether $M \rightarrow^* fail$

Proof: Translate M into a recursion scheme G s.t. $M \rightarrow^* fail$ if and only if G generates a tree containing "fail".

Example

```
fun repeatEven f x = if * then x else f (repeatOdd f x)
fun repeatOdd f x = f (repeatEven f x)
fun main( ) = if (repeatEven not true) then ( ) else fail
```

↓

```
S → RepeatEven C Not True
C b → if b e fail
RepeatEven k f x → if* (k x) (RepeatOdd (f k) f x)
RepeatOdd k f x → RepeatEven (f k) f x
Not k b → If b (k False) (k True)
If b x y → b x y
True x y → x
False x y → y
```

Comparison with Other Model Checking

Program Classes	Verification Methods	
Programs with while-loops	Finite state model checking	
Programs with 1st-order recursion	Pushdown model checking	infinite state model checking
Higher-order functional programs	Recursion scheme model checking	

Outline

- ♦ Introduction to higher-order model checking
  - What are higher-order recursion schemes?
  - What are model checking problems?
  - How related to program verification?
- ♦ Model checking functional programs
  - Predicate abstraction and CEGAR
  - Automata-based abstractions for recursive data structures
- ♦ Discussion and conclusion

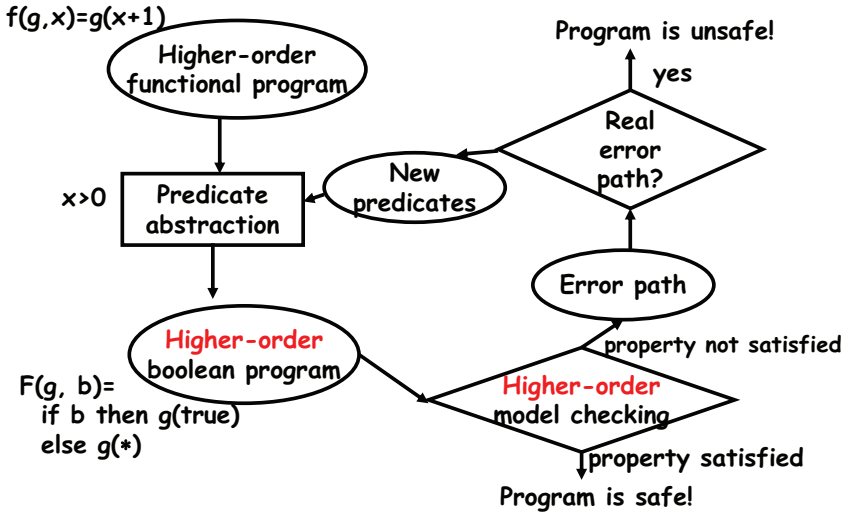
Recursion schemes as models of higher-order programs?

+ simply-typed λ-calculus
+ recursion
+ tree constructors
+ finite data domains (via Church encoding; true = λx.λy.x, false=λx.λy.y)
- infinite data domains (integers, lists, trees,…)
- advanced types (polymorphism, recursive types, object types, …)
- imperative features/concurrency

Dealing with Infinite Data Domains

- From recursion schemes to higher-order multi-tree transducers (HMTT), to deal with algebraic data types (lists, trees, …) [K.,Tabuchi&Unno, POPL 2010]
- Predicate abstraction and CEGAR (c.f. BLAST, SLAM, …)

Predicate Abstraction and CEGAR for Higher-Order Model Checking



What are challenges?

- Predicate abstraction
  - How to choose predicates for each term, in such a way that the resulting HOBP is consistent?

E.g.
```
fun f g x = ... g (x+1) ...
fun h y z = ...
fun main() = ... f (h 0) u ...
```

The same predicate should be used for z and u+1.

- CEGAR
  - How to find new predicates to abstract each term to guarantee progress (i.e. any spurious counterexample is eliminated)?

Our Approach

- Predicate abstraction

Abstraction types to express abstraction interface:

e.g. f: (x:int[λx.x>0] → int[λy.y>x])

Assuming the argument x is abstracted using predicate x>0, the return value y should be abstracted using y>x.

f(x) = if x>0 then x+1 else ...
=> F(b)= if b then true else ...

- CEGAR

Reduction from abstraction type finding problem to a refinement type inference problem for SHP (straightline higher-order program).

Example (predicate abstraction)

```
let sum n k = if n<=0 then k 0
              else sum (n-1) (fun x-> k(x+n))
in sum m (fun x-> assert(x>=m))
```

sum: (n:int[]→ (int[λx.x>=n] →★) →★)

```
let sum ( ) k = if * then k true
                else sum ( ) (fun b-> k true)
in sum ( ) (fun b-> assert(b))
```

Our Approach

♦ Predicate abstraction

Abastraction types to express abstraction interface:

e.g. f: (x:int[λx.x>0] → int[λy.y>x])

Assuming the argument x is abstracted using predicate x>0, the return value y should be abstracted using y>x.

f(x) = if x>0 then x+1 else ...

=> f'(b)= if b then true else ...

♦ CEGAR

Reduction from abstraction type finding problem to a refinement type inference problem for SHP (straightline higher-order program).

Example (predicate discovery)

```
let sum n k = if n<=0 then k 0
              else sum (n-1) (fun x-> k(x+n))
in sum m (fun x-> assert(x>=m))
```

sum: (n:int[]→ (int[ ] →★) →★)

```
let sum ( ) k = if * then k ( )
                else sum ( ) (fun ( )-> k ( ))
in sum ( ) (fun ( )-> assert(*))
```

spurious error path (with k = (fun ( )-> assert(*))):
sum ( ) k → if * then k( ) else ... → k( ) → assert(*) → fail

Example (predicate discovery)

```
let sum n k = if n<=0 then k 0
              else sum (n-1) (fun x-> k(x+n))
in sum m (fun x-> assert(x>=m))
```

Spurious error path:
sum ( ) k → if * then k( ) else ... → k( ) → assert(*) → fail

Straightline higher-order program (SHP):

```
let sum n k = assume(n<=0); k 0
in sum m (fun x -> asume(not(x>=m)); fail)
```

[Unno&K. PPDP09]

Typing for SHP:

sum: (n:int → ({x:int | x>=n} → ★) → ★

Experiments

	cycle	Time (sec)
mc91	2	0.07
ackermann	3	0.15
a-cppr	6	3.40
a-max	5	4.78
l-zipmap	4	0.20
l-zipunzip	3	0.12
repeat	3	0.15
a-max-e	2	0.13

Arrays encoded by:

```
let mk_array n i =
    assert(0<=i && i<n); 0
let update i n a x =
    a(i);
    let a' j = if i=j then x else a(i)
    in a'
```

(Environment: Intel(R) Xeon(R) 3Ghz with 8GB memory)

Outline

- Introduction to higher-order model checking
  - What are higher-order recursion schemes?
  - What are model checking problems?
  - How related to program verification?
- Model checking functional programs
  - Predicate abstraction and CEGAR
  - Automata-based abstractions for recursive data structures
- Discussion and conclusion

Remaining challenges

- More efficient higher-order model checker
  - practical fixed-parameter linear time algorithm (c.f. [K., FoSSaCS 2011])
  - efficient implementation techniques (e.g. BDD)
- Supporting more language features
  - recursive data structures
    - abstraction by automata/transducers
  - recursive types
  - objects (software model checker for Java)
- More verification power
  - Abstractions using higher-order predicates
  - Inference of auxiliary arguments

Conclusion

- New program verification method based on higher-order model checking
  - Many attractive features
    - Sound, complete, and fully automatic for certain classes of higher-order programs and verification problems
    - Subsumes first-order/pushdown model checking
    - Integration of types and model checking
      - types as certificates
      - counterexamples
  - Many interesting and challenging topics

References

- K., POPL09
  From program verification to model-checking, and typing
- K.&Ong, ICALP09 Complexity of model checking
- K.&Ong, LICS09 From model-checking to type checking
- K., PPDP09 First practical higher-order model-checking algorithm
- K., Tabuchi & Unno, POPL10
  Extension to transducers and its applications
- Tsukada & K., FoSSaCS 10
  Extension to deal with more advanced types
- Unno, Tabuchi & K., APLAS 2010
  Extension of POPL10 work to deal with arbitrary tree-processing programs
- K., FoSSaCS 2011
  Practical fixed-parameter linear time algorithm for higher-order model checking

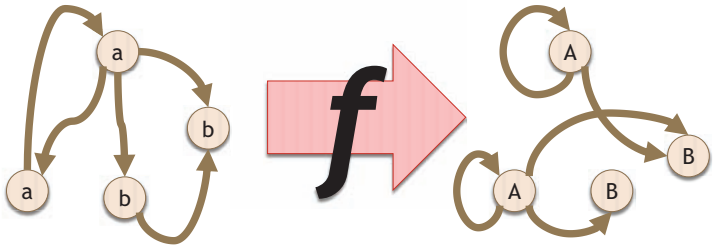
Modal-μ Definable Graph Transduction

Kazuhiro Inaba
National Institute of Informatics, Japan

4th DIKU-IST Workshop, 2011

What We Want to Do

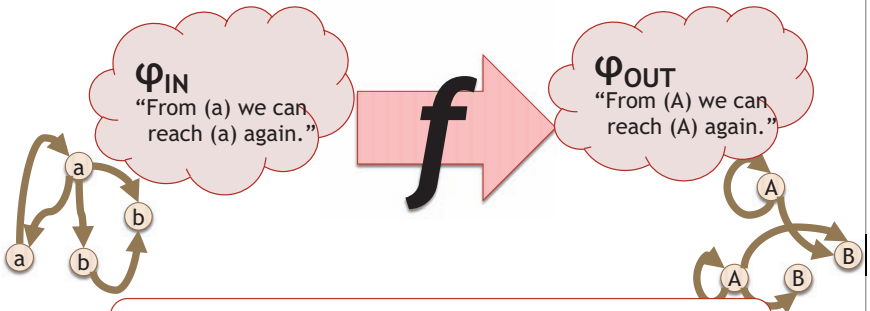
- Verification of **Graph-to-Graph Transformations**



- e.g., Queries on Graph-Structured Database
  or Transformations of XML with "id" links

What We Want to Do

- Verification of Graph-to-Graph Transformations with respect to **input/output specifications**



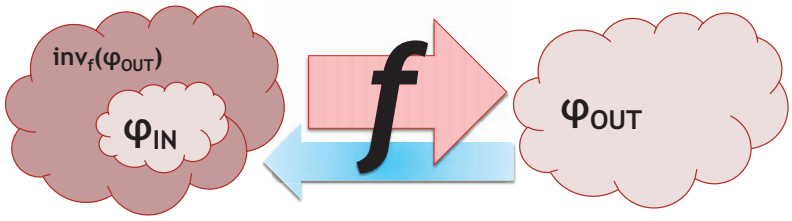
verify whether or not:
for any graph G, $G \models \varphi_{IN} \Rightarrow f(G) \models \varphi_{OUT}$

4/35

Verification by Pre-Image

(a.k.a. "weakest precondition" or "inverse type inference")

Given **f** and φ_{OUT} , compute $\mathbf{inv_f(\varphi_{OUT})}$ such that:
for any graph G, $\mathbf{f(G) \models \varphi_{OUT}}$ **iff** $\mathbf{G \models inv_f(\varphi_{OUT})}$



Then "**for any graph G,** $G \models \varphi_{IN} \Rightarrow f(G) \models \varphi_{OUT}$"
iff "**for any graph G,** $G \models (\varphi_{IN} \rightarrow inv_f(\varphi_{OUT}))$"
i.e., $\varphi_{IN} \rightarrow inv_f(\varphi_{OUT})$ is ***valid***

5/35

To Be More Concrete...

- Which logic can we use for specifying $\varphi_{IN/OUT}$?
  - Must be strong enough to express useful conditions.
  - Must be weak enough to have **decidable validity**.
- What kind of transformation ***f*** can be verified ?
  - We must be able to **compute the pre-image**.

6/35

Our Approach

- Take **Modal-μ Calculus** as the specification logic
  - (At least for trees) capture all existing XML-Schemas
- Define a new model of graph transformation called ***Modal-μ Definable Transduction***
  - Pre-image of modal-μ sentence can be fully automatically computed
  - Expressive enough to capture (unnested) structural recursion on graphs

Related Work

- MSO (Monadic 2nd-Order Logic) Definable Transduction
  - Overall structure is more or less the same.
  - Ours is a proposal to use **Modal-μ instead of MSO**
- Hoare-Style Verification of Imperative Programs
  - Ours don't deal with pointers or destructive updates.
  - Rather, it is more suitable for **functional programs**
  - **Structural recursion** is handled without any annotations

```
fun f( {$l: $x} ) = {cap($l) : g($x)}
fun g( {_: $x} ) = f($x)
{ φIN }  f  { φOUT }
```

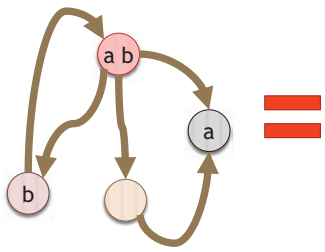
```
{ φIN }
  p := root
  while  p != null do
    q := p.next
    p.next := p.next.next
    p := q
  end
{ φOUT }
```


Outline

- Two Kinds of Logics on Graphs
  - Predicate Logics
  - Modal Logics
  - **Why Modal-μ ?**
- Review: Predicate-Logic Based Approach
  - MSO-Definable Graph Transduction [Courcelle 94]
- **Our Work:**
  - **Modal-μ Definable Graph Transduction**
  - **Computation of Pre-Image**

Graphs (in Today's Talk)

- Σ : Finite Nonempty Alphabet
- G = (V, E, π)
  - V — Set of Nodes
  - $E \subseteq V \times V$ — Set of Directed Edges
  - $\pi : V \to 2^{\Sigma}$ — Labels on Nodes



$\Sigma = \{a, b\}$

$V = \{$ a b, a, b, $\}$

$\pi =$

$\to \{a,b\}$

$\to \{a\}$

$\to \{b\}$

$\to \{\}$

Predicate Logics on Graphs

φ ::=
| False | ¬ φ | φ ∨ φ
| σ(x) (for σ∈Σ) "node x is labeled σ"
| edge(x, y) "an edge connects x to y"
| ∃x. φ "there's x that makes ψ hold"

FO

| ∃S. φ "there's a set S that makes ψ hold"
| x ∈ S "x is in S"

MSO

We can define True, φ∧φ , φ→φ, ∀x.φ, and ∀S.φ.

Semantics

- For a graph G=(V,E,π) and an environment Γ : Var→V
  - **G, Γ ⊨ σ(x)** iff σ ∈ π(Γ(x))
    "node x is labeled σ"
  - **G, Γ ⊨ edge(x, y)** iff (Γ(x), Γ(y)) ∈ E
    "an edge connects x to y"
  - **G, Γ ⊨ ∃x.φ** iff there's v∈V s.t. G,Γ[x:v] ⊨ φ

  …

Modal Logics on Graphs

ψ ::=
| False | ¬ φ | φ ∨ φ
| σ (for σ∈Σ) "current node is labeled σ"
| ◇φ "current node has an outgoing edge whose destination satisfies φ"

M

| X
| μX.φ "least fixpoint" (X must be in even # of ¬)

Mμ

We Can Define: □φ (dual of ◇) and νX.φ (GreatestFixPt)

13/35

Semantics

- For a graph G=(V,E,π), an environment Γ : Var→2^V, and the current node v ∈ V

  - **G, v, Γ ⊨ σ** iff σ ∈ π(v)
    "current node is labeled σ"
  - **G, v, Γ ⊨ ◇φ** iff there's w (v,w)∈E & G,w,Γ ⊨ φ
    "current node has an outgoing edge whose destination satisfies φ"
  - **G, v, Γ ⊨ μY. φ** iff v ∈ LFP(F)
    where F(A) = {w∈V | G, w, Γ[Y:A] ⊨ φ}

  ...

14/35

Examples

- "From the node x, we can reach a σ-node"
  ∀S. ((x∈S ∧ ∀y.∀z.(y∈S ∧ (edge(y,z)→z∈S))) → ∃y. (y ∈ S ∧ σ(y)))
- "Confluence"
  ∀y. ∀z. (edge(x,y) ∧ edge(x,z) → ∃w. (edge(y,w) ∧ edge(z,w)))
- "From the current node, we can reach a σ-node"
  μY. (σ ∨ ◇Y)
- "Confluence"
  (No way to express it in Modal-μ)

15/35

MSO Definable (1-copying) Transduction [Courcelle 94]

A set of MSO formulas **T** =

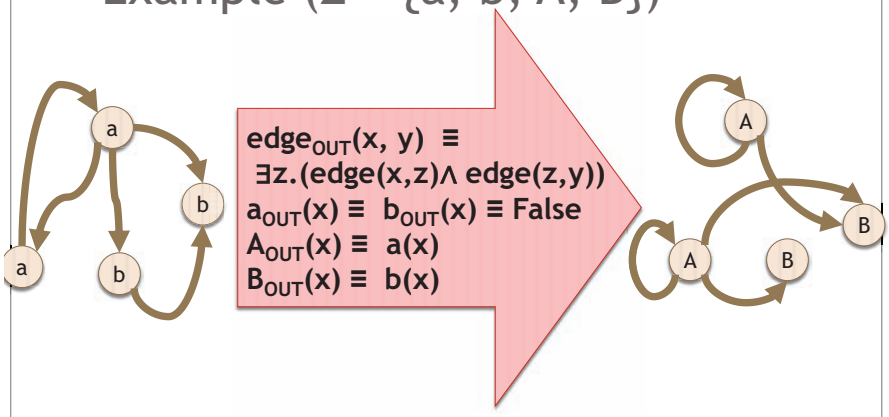
- $\sigma_{OUT}(x)$ for each σ∈Σ
- $edge_{OUT}(x,y)$

defines a transformation f_T converting
G = (V, E, π) into **G' = (V, E', π')** where

- **π'(v) = { σ | G, x:v ⊨ $\sigma_{OUT}(x)$ }**
- **E' = { (v, w) | G, x:v, y:w ⊨ $edge_{OUT}(x,y)$ }**

16/35

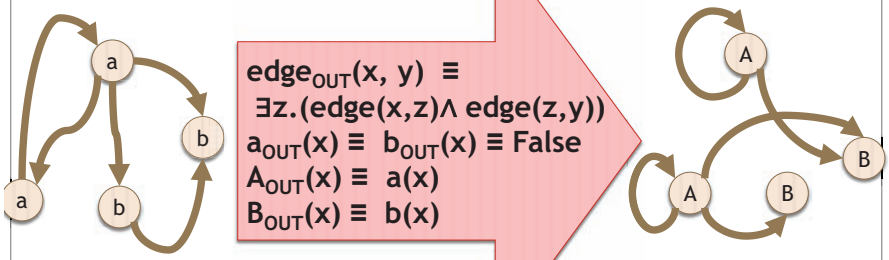
Example (Σ = {a, b, A, B})



$edge_{OUT}(x, y) \equiv \exists z.(edge(x,z) \wedge edge(z,y))$
$a_{OUT}(x) \equiv b_{OUT}(x) \equiv False$
$A_{OUT}(x) \equiv a(x)$
$B_{OUT}(x) \equiv b(x)$

17/35

Pre-Image is Easily Obtained



$edge_{OUT}(x, y) \equiv \exists z.(edge(x,z) \wedge edge(z,y))$
$a_{OUT}(x) \equiv b_{OUT}(x) \equiv False$
$A_{OUT}(x) \equiv a(x)$
$B_{OUT}(x) \equiv b(x)$

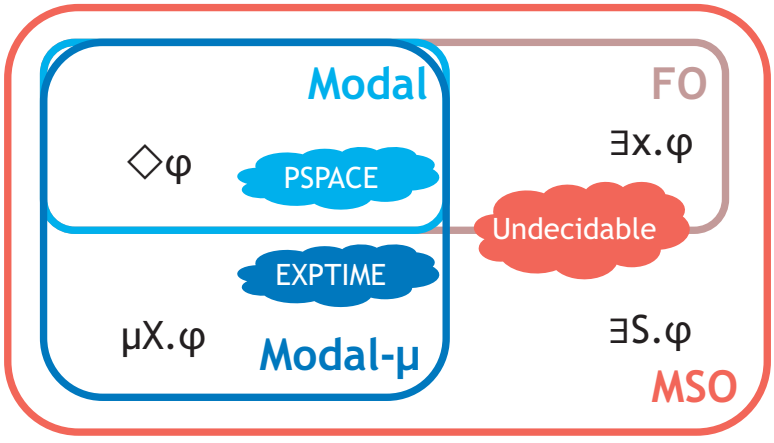
$\forall x.\ a(x) \rightarrow \exists y.\ \exists z.(edge(x,z) \wedge edge(z,y)) \wedge a(y)$

Inline Expansion

$\forall x.\ A(x) \rightarrow \exists y.\ edge(x,y) \wedge A(y)$

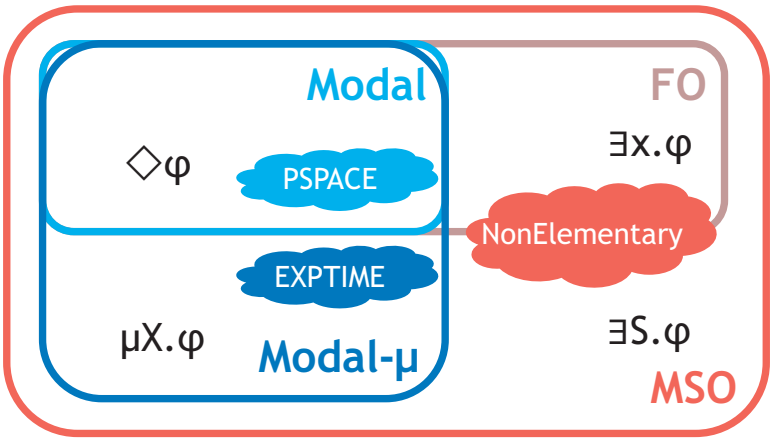
18/35

Expressiveness & Complexity



19/35

Expressiveness & Complexity (on "tree-like" graphs)



20/35

Modal-μ and MSO

- Complexity of Validity Checking
  - **Modal-μ : EXPTIME-complete**
  - MSO : Undecidable (Even in Trees, HyperEXP)
- Expressive Power
  - Modal-μ = ***Bisimulation-Invariant* Subset of MSO** [Janin & Walukiewicz 96]
  - "Bisimulation-Invariant" ≃ "Physical equality of pointers cannot be checked"
  - Not a severe restriction for purely functional programs!

21/35

Modal-μ Definable (1-copying) Transduction

A set of Modal-μ formulas T =

- σ_{OUT} for each $\sigma \in \Sigma$
- $edge_{OUT}$ an ***existential*** formula Fv={X}

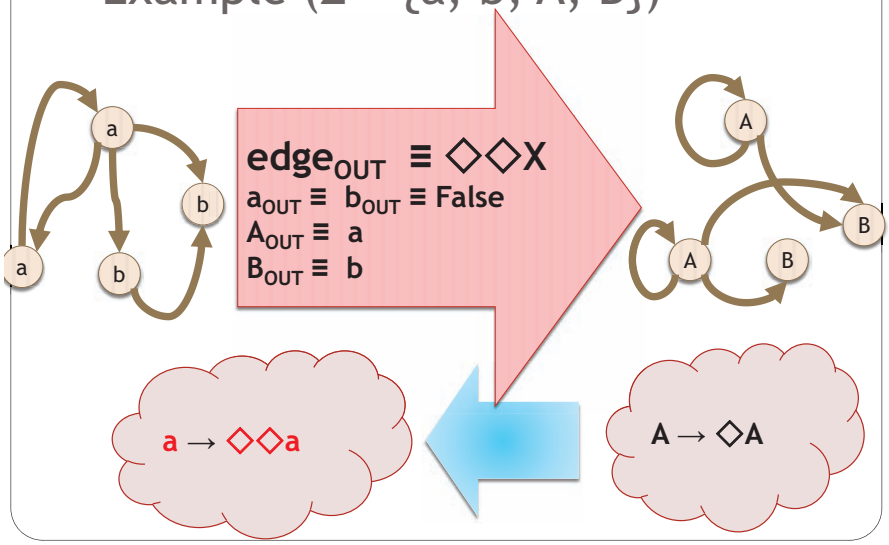
defines a transformation $\mathbf{f_T}$ converting

G = (V, E, π) into **G' = (V, E', π')** where

- **$\pi'(v) = \{ \sigma \mid G, v \models \sigma_{OUT} \}$**
- **$E' = \{ (v, w) \mid G, v, X:\{w\} \models edge_{OUT} \}$**

22/35

Example (Σ = {a, b, A, B})



23/35

Existential Formula

- A formula e with one free variable X is ***existential***, if

for all G=(V,E,π), v∈V, P⊆V
G, v, X:P ⊨ e iff ∃w∈P. G, v, X:{w} ⊨ e

- Examples:
  - "X ∨ True" is not existential (Consider P={}).
  - "◇X" is existential.
  - "□X" is not (when v is a leaf node …).
  - "σ" is not, but "X ∧ σ" is.

24/35

Syntactic Condition

for all G=(V,E,π), v∈V, P⊆V
G, v, X:P ⊨ e iff ∃w∈P. G, v, X:{w} ⊨ e

- Theorem:
  e is existential if it is in the following syntax

e ::= False | X | Y | e ∨ e | ◇e | μY. e
| e ∧ φ where φ is any formula without free variables

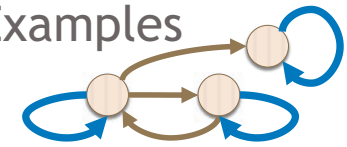
(True, ¬, σ, □, and GFP must be "guarded" by _ ∧ _)

OPEN QUESTION: is this a necessary condition ?
(i.e., do all existential formulas have logically-equivalent forms in this syntax?)

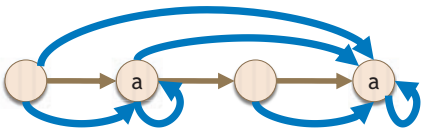
25/35

More Examples

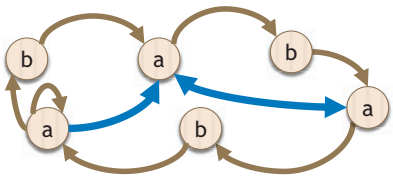
- $edge_{OUT} \equiv X$



- $edge_{OUT} \equiv \mu Y.\ ((X \wedge a) \vee \Diamond Y)$

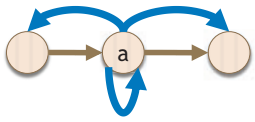


- $edge_{OUT} \equiv \mu Y.\ ((X \wedge a \wedge \Box b) \vee (\neg a \wedge \Diamond Y)$

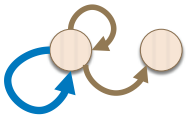


(Non-Examples)

$edge_{OUT} \equiv a$



$edge_{OUT} \equiv X \wedge \Diamond X$



26/35

Pre-Image Computation

For $T = (\sigma_{OUT}, e_{OUT})$, define

- $inv(\ False\ ) = False$
- $inv(\ \neg\ \varphi\ ) = \neg\ inv(\ \varphi\ )$
- $inv(\ \varphi_1 \vee \varphi_2\ ) = inv(\ \varphi_1\ ) \vee inv(\ \varphi_2\ )$
- **$inv(\ \sigma\ ) = \sigma_{OUT}$**
- **$inv(\ \Diamond\ \varphi\ ) = edge_{OUT}\ [X\ /\ inv(\varphi)]$**
- $inv(\ Y\ ) = Y$
- $inv(\ \mu Y.\ \varphi\ ) = \mu Y.\ inv(\varphi)$

Theorem: $f_T(G), v \models \varphi$ iff $G, v \models inv(\varphi)$

27/35

Proof of the Theorem

Theorem: $f_T(G), v \models \varphi$ iff $G, v \models inv(\varphi)$

- By Induction on φ. The essential case is:

  $G, v \models inv(\Diamond\varphi)$

  iff $G,v \models edge_{OUT}\ [X\ /\ inv(\varphi)]$ (definition of inv)

  iff $\exists w\ (G,v,X:\{w\} \models edge_{OUT}$ and $G,w \models inv(\varphi))$ (ext)

  iff $\exists w\ ((v,w)$ in E' and $G,w \models inv(\varphi))$ (def of E')

  iff $\exists w\ ((v,w)$ in E' and $f_T(G),w \models \varphi)$ (IH)

  iff **$f_T(G), v \models \Diamond\varphi$** (definition of $\Diamond$)

n-copying Modal-μ Definable Transduction

A set of Modal-μ formulas T =

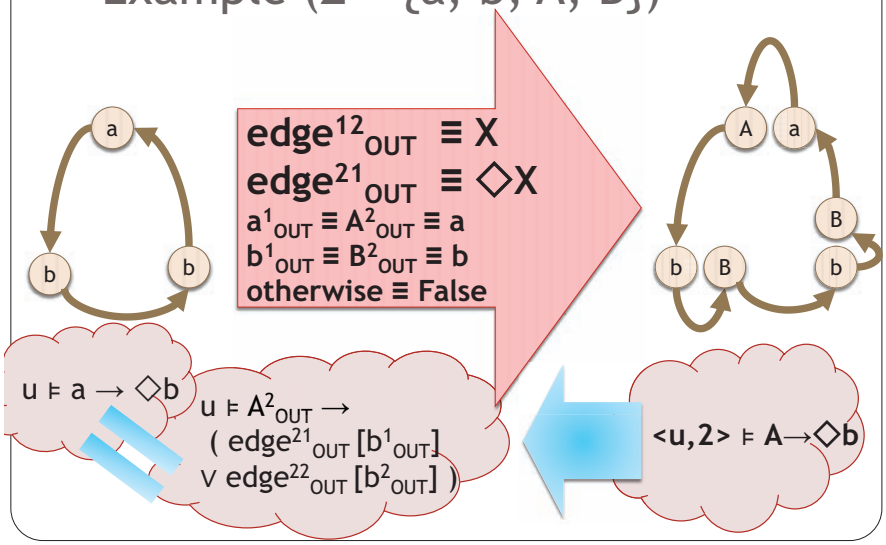
- σ^k_{OUT} for each $\sigma \in \Sigma$, $k \in \{1 .. n\}$
- $edge^{ik}_{OUT}$ for each i, $k \in \{1 .. n\}$: ***existential***

defines a transformation f_T converting

G = (V, E, π) into **G' = (V*{1..n}, E', π')** where

- **$\pi'(\langle v,k \rangle) = \{ \sigma \mid G, v \models \sigma^k_{OUT} \}$**
- **$E' = \{ (\langle v,i \rangle, \langle w,k \rangle) \mid G, v, X:\{w\} \models edge^{ik}_{OUT} \}$**

Example (Σ = {a, b, A, B})



Example

- **Mutual structural recursion** (without accumulating parameters) can be dealt with.
  - For the detail of structural recursion over graphs, see [Buneman, Fernandez & Suciu 00]
- fun ev(a → x) = $\overset{1}{A} \to \overset{2}{A}$ → od(x)
- fun ev(b → x) = $\overset{1}{B}$ → od(x)
- fun od(a → x) = $\overset{3}{A}$ → ev(x)
- fun od(b → x) = $\overset{3}{B} \to \overset{4}{B}$ → ev(x)

$edge^{12}_{OUT} \equiv a \wedge X$ $edge^{23}_{OUT} \equiv a \wedge \Diamond X$
$edge^{13}_{OUT} \equiv a \wedge \Diamond X$
$edge^{31}_{OUT} \equiv b \wedge \Diamond X$
$edge^{34}_{OUT} \equiv b \wedge X$ $edge^{41}_{OUT} \equiv b \wedge \Diamond X$

Pre-Image Computation

- inv_k (False, Δ) = False
- inv_k (¬φ , Δ) = ¬ inv_k (φ, Δ)
- inv_k ($\varphi_1 \vee \varphi_2$, Δ) = inv_k (φ_1, Δ) ∨ inv_k (φ_2, Δ)
- inv_k (σ, Δ) = σ^k_{OUT}
- inv_k (◇φ, Δ) = $\bigvee_{j \in \{1..n\}} edge^{kj}_{OUT}$ [X / inv_j(φ, Δ)]
- inv_k (Y, Δ) = Y_k if $k \in S$
- inv_k (Y, Δ) = μY_k. inv_k(φ, Δ[Y→<S∪{k},φ>]) where (S,φ) = Δ(Y)
- inv_k (μY.φ , Δ) = μY_k. inv_k(φ, Δ[Y→<{k},φ>])

Thm: $f_T(G)$, <v,k> ⊨ φ iff G,v ⊨ inv_k(φ, {})

Example

$edge^{11}_{OUT} \equiv edge^{12}_{OUT} \equiv edge^{21}_{OUT} \equiv edge^{22}_{OUT} \equiv \Diamond X$

$a^1_{OUT} \equiv a^2_{OUT} \equiv a$

OPEN QUESTION: can inv(μ) be shorter than (n-1)!+1 ?

- f(G), <v,1> ⊨ μY. (a ∧ ◇Y)
- G, v ⊨ μY_1. inv_1(a ∧ ◇Y)
- G, v ⊨ μY_1. a ∧ (◇inv_1(Y) ∨ ◇inv_2(Y))
- G, v ⊨ μY_1. a ∧ (◇Y_1 ∨ ◇μY_2.inv_2(a∧◇Y))
- G, v ⊨ μY_1. a ∧ (◇Y_1 ∨ ◇μY_2. a∧(◇inv_1(Y)∨◇inv_2(Y))
- G, v ⊨ μY_1. a ∧ (◇Y_1 ∨ ◇μY_2. a∧(◇Y_1∨◇Y_2))

Some Useful Results

Theorem:
Modal-μ Definable Transduction is closed under composition.

Construction is analogous to inv(φ).

Theorem:
Modal-μ Definable Transduction
⇔ MSO Definable & Bisimulation-Invariant.

It is known that Bisimulation-Invariant MSO transduction is equal to structural recursion [Colcombet & Löding 04].

Conclusion

- Modal-μ Definable Transduction
  - Pre-Image of a modal-μ sentence is computable
  - Structural recursion is expressible
  - (Not in the talk)
    - Node-erasing transformations
    - Edge-labeled graphs
    - Transformations with multiple inputs/outputs
- Future Work
  - Implementation
  - Addition of **Backward Modality**
    - (G,v ⊨ ◆φ iff there's **(w,v)∈E** s.t. G,w ⊨ φ)
  - **Syntactic necessary condition for $edge_{OUT}$**
  - More concise formula for **inv(μY.φ)**

References

[Trakhtenbrot 50] **Impossibility of an Algorithm for the Decision Problem for Finite Classes**
- Satisfiability of FO on graphs is undecidable

[Meyer 74] **Weak monadic second order theory of successor is not elementary-recursive**
- Satisfiability of MSO on finite strings is Non-Elementary

[Robertson 74] **Structure of Complexity in the Weak Monadic Second-Order Theories of the Natural Numbers**
- Satisfiability of FO[<] on finite strings is Non-Elementary

[Lander 77] **The Computational Complexity of Provability in Systems of Propositional Modal Logic**
- Satisfiability of Modal Logic on graphs is PSPACE-complete

[Emereson & Jutla 88] **The Complexity of Tree Automata and Logics of Programs**
- Satisiability of Modal-μ on graphs is EXPTIME-complete

[van Benthem 86] **Essays in Logical Semantics**
- FO ∩ Bisim = Modal

[Janin & Walukiewicz 96] **On the Expressive Completeness of the Propositional mu-Calculus with Respect to Monadic Second Order Logic**
- MSO ∩ Bisim = Modal-μ

[Colcombet & Löding 04] **On the Expressiveness of Deterministic Transducers over Infinite Trees**
- MSO-Definable Graph Transduction ∩ Bisim = Structural Recursion

[Courcelle 94] **Monadic Second-Order Definable Graph Transductions: A Survey**
- On MSO-Definable Transduction

Determining the Valid Parameters for the Weight-Balanced Tree Algorithm

Yoichi Hirai
IST, the University of Tokyo

January 11, 2011

(joint work with Kazuhiko Yamamoto
at Internet Initiative Japan)

History of Weight-Balanced Tree Algorithm

1972	proposed by Nievergelt and Reingold
1992	valid parameter determination and an example implementation of set operations by Adams
–1995	an implementation by Adams in MIT-Scheme
1998	another Scheme implementation in SLIB
–2002	Data.Map and Data.Set implementation in Hackage

History of Weight-Balanced Tree Algorithm

1972	proposed by Nievergelt and Reingold
1992	valid parameter determination and an example implementation of set operations by Adams (buggy)
–1995	an implementation by Adams in MIT-Scheme (buggy)
1998	another Scheme implementation in SLIB (buggy)
–2002	Data.Map (buggy) and Data.Set implementation in Hackage
2010-08-03	A bug found in `Data.Map` a tree got unbalanced after a delete

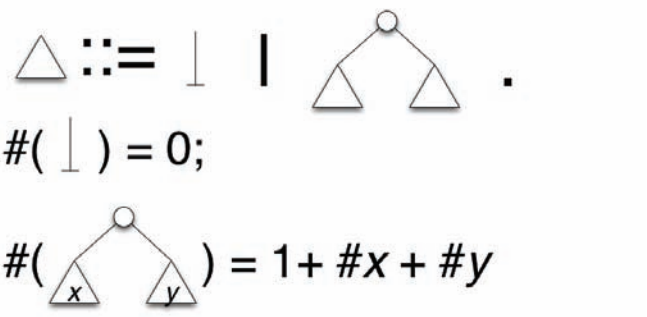
buggy: violating the restriction posed by the algorithm design.

Question

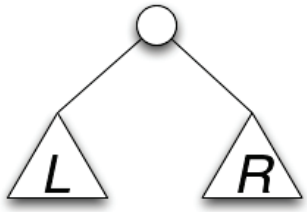
- There is a parametrized binary tree algorithm
  - Δ decides whether a tree is balanced
  - Γ chooses a balancing strategy: single/double rotation
- Wrong parameters break balance (found to be a common mistake)

Q. What parameters are correct?

Weight-Balanced Tree Algorithm (Nievergelt and Reingold, 1972)



The balance condition with Δ(wrong version)

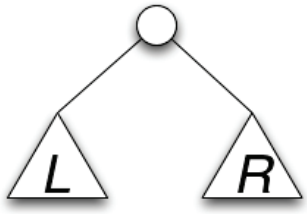


The top node is balanced iff

$$\#L \leq \Delta \times \#R$$

and vice versa

The balance condition withΔ

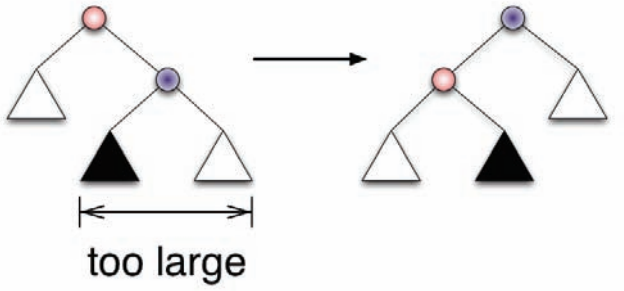


The top node is balanced iff

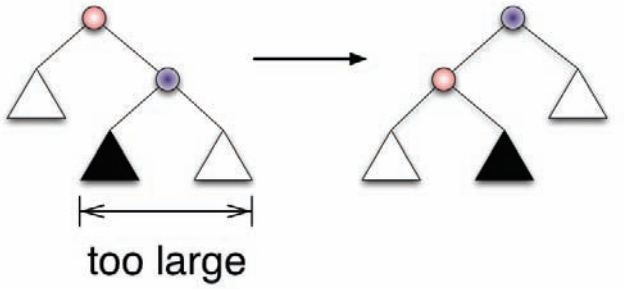
$$\#L + 1 \leq \Delta \times (\#R + 1)$$

and vice versa

Balancing Strategy 1: Single Rotation



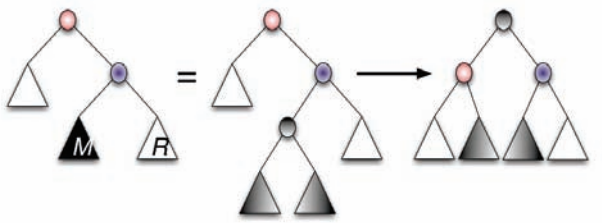
Balancing Strategy 1: Single Rotation is Not Enough



...when the black part is too large, this cannot balance the whole.

Balancing strategy withΓ

choose double rotation



when $\#M + 1 \geq \Gamma \times (\#R + 1)$

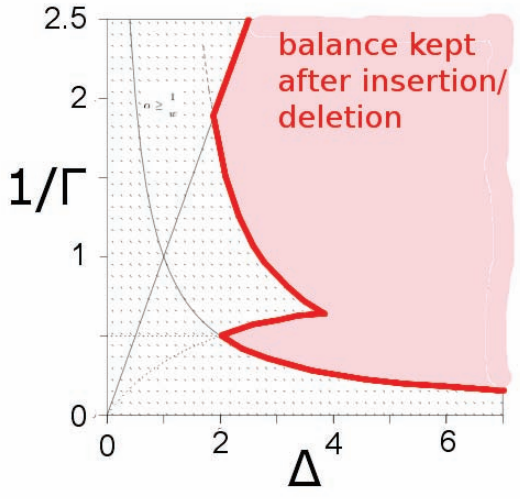
Two Parameters and One Question

- Δ for balance condition
- Γ for choosing single or double

Q. Under what (Δ,Γ) every insert/delete keeps balance?

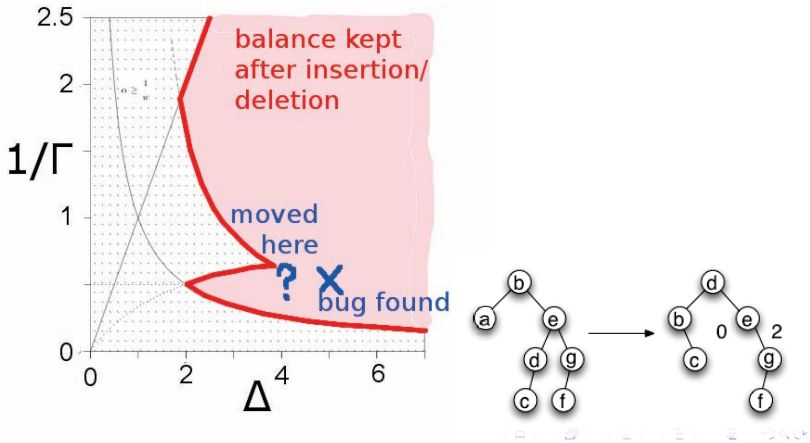
Nievergelt and Reingold has $(1 + \sqrt{2}, \sqrt{2})$. Comparing $(1 + \sqrt{2})m$ and n is costly (taking square?).

1992: Adams's Analysis



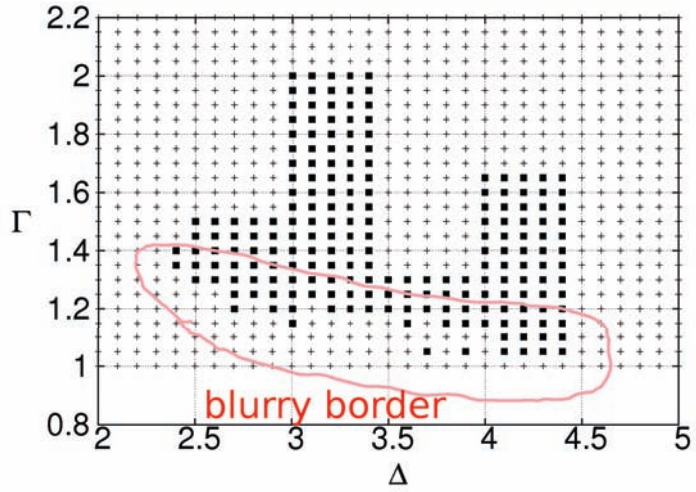
2010: Shadow of Doubt

On Hackage, the bug ticket was closed after they changed the parameters.

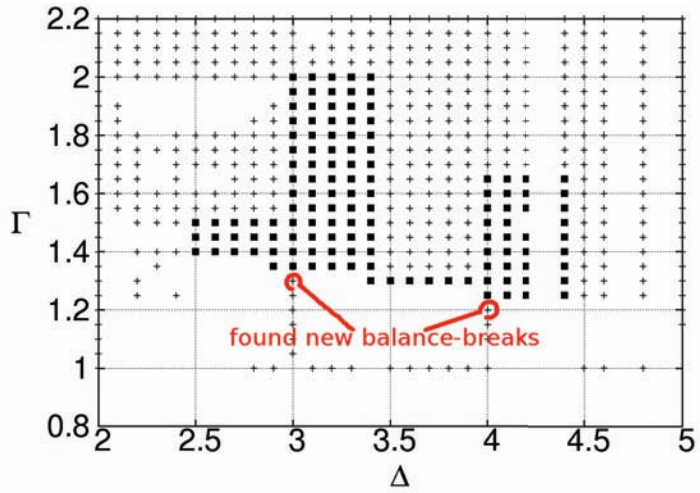


QuickCheck Result by Kazu Yamamoto

apparently bounded.

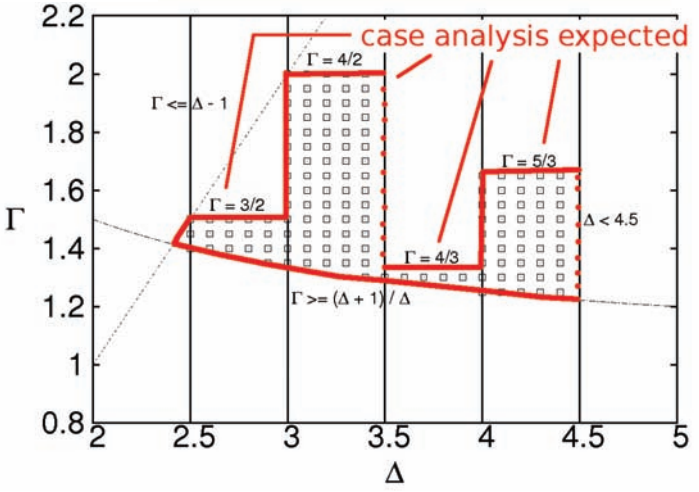


Omega: Presburger Solver



copied Coq responce changed / into && and so on.

Conjecture



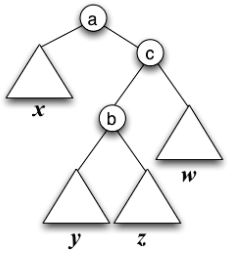
(sound) in the area, balance never broken

(complete) outside the area, balance can be broken

Completeness Proof

1. given any parameter pair (just) below the lower boundary,
2. take large enough z so that
   $z \geq \frac{\Gamma\Delta-\Delta+\Gamma}{1+\Delta-\Gamma\Delta} \sim \frac{\text{positive}}{\text{small positive}}$
3. deletion in the left subtree breaks the balance

$$w = \lfloor \Delta(z+1) \rfloor, \quad y = w-1, \quad r = y+z+w+2, \quad x = \left\lceil \frac{r+1}{\Delta} \right\rceil - 1.$$



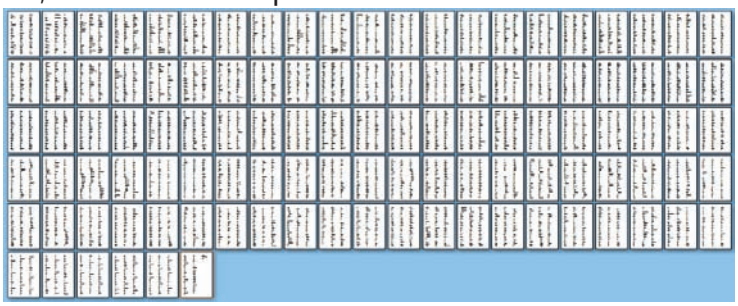
Soundness Proof

Goal: Within the range, the balance is never broken.
Reasons for using proof assistant Coq rather than proving by hand:

- many cases on parameter pairs
- every operation on every tree
- nested if in every operation
- balance condition of every subtree

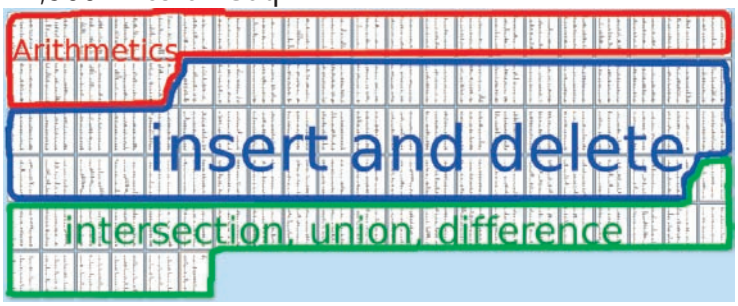
The Soundness Proof

12,900 lines of Coq.



The soundness proof

12,900 lines of Coq.



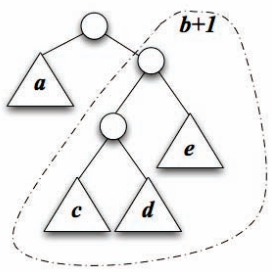
Required Time

- 62 days from 2010-08-18 to 2010-10-18
- 208 lines per day among other things

Arithmetic Lemmas

```
Lemma NR_deep_insert:
  good_params ->
  forall (a b c d e: Z),
    ???????
    (a#1) (b#1) (c#1) (d#1) (e#1).
```

Informally: a, b, c, d, e are



← the tree just before rotation

Arithmetic Lemma: Expanded

```
good_params ->
forall a b c d e : Z,
balance (a # 1) (b # 1) ->
~ balance (a # 1) ((b # 1) + 1) ->
(c #1) + (d #1) + 1 + (e #1) + 1 == (b #1) + 1 ->
balance (c # 1) (d # 1) ->
balance ((c # 1) + (d # 1) + 1) (e # 1) ->

if NR_isSingleSize ((c # 1) + (d # 1) + 1) (e # 1)
then
 balance (a # 1) ((c # 1) + (d # 1) + 1) /\
 balance (c # 1) (d # 1) /\
 balance ((a #1) + (c #1) + (d #1) + 1 + 1) (e # 1)
else
 balance (a #1) (c #1) /\ balance (d #1) (e #1) /\
 balance ((a #1) + (c #1) + 1) ((d #1) + (e #1) + 1)
```

Arithmetic Lemma: Proof Structure

- case analysis for Δ
- case analysis for special small trees
  needed because of this kind of reasoning:
  $b > 2.5$ implies $b \geq 3$ because $b \in \mathbb{Z}$
- reducing $\mathbb{Q}$ argument into $\mathbb{Z}$ argument
- `psatz` tactics

Arithmetics is Not Enough

We have to define binary trees.

```
Module FSet (X: OrderedType).

Definition Size := Z.
Definition k := X.t.

Inductive FSet :=
| Tip: FSet
| Bin: Size -> k -> FSet -> FSet -> FSet.
```

Arithmetics is Not Enough

We have to define insertion.

```
Fixpoint insert (kx:k) (t:FSet): FSet :=
  match t with
    | Tip => singleton kx
    | Bin _ ky l r =>
      match X.compare kx ky with
        | GT _ => balanceL ky l (insert kx r)
        | LT _ => balanceR ky (insert kx l) r
        | EQ _ => bin kx l r
      end
  end.
```

Arithmetics is Not Enough

```
Definition balanceR (kx: k) (l r:FSet) :=
  match (isBalanced r l) with
    | true => bin kx l r
    | false => rotateR kx l r
  end.
```

Arithmetics is Not Enough

```
Definition rotateR :
k -> FSet -> FSet -> FSet :=
  fun k l r =>
    match l with
      | Tip => assert_false k l r
      | Bin _ _ ll lr =>
        if isSingle lr ll then
          singleR k l r
        else
          doubleR k l r
    end.
```

Insert and Delete

```
Lemma insert_balanced:
  forall (t: FSet) (kx: k),
   good_params ->
   Is_true (balanced t) -> validsize_rec t ->
   Is_true (balanced (insert kx t)).
Lemma delete_balance:
 good_params ->
 forall (x: k) (t: FSet),
   validsize_rec t ->
   Is_true (balanced t) ->
   Is_true (balanced (delete x t)).
```

- induction on t
- case whether size changed or not

Set Operations

(request by Milan Straka, the maintainer of the containers package in Hackage)

- difference, union: straightforward
- some trick required for defining intersection induction on two arguments

Intersection Used This Induction

```
Fixpoint pair (l r: Tree): TreePair :=
  match l with
    | Tip =>
      match r with
        | Tip => TipTip
        | Bin rl rr => TipBin rl rr
      end
    | Bin ll lr =>
      match r with
        | Tip => BinTip ll lr
        | Bin rl rr => BinBin (pair l rl) (pair lr r)
      end
  end.
```

Experimental Function command solved this

also, I packed two arguments in a single pair

Achievements

1972	Proposed by Nievergelt and Reingold
1992	parameter analysis and an example implementation of set operations by Adams
1995 or older	MIT-Scheme implementation by Adams (buggy)
1998-02-09	SLIB implementation (buggy)
–2002	Data.Map (buggy) and Data.Set implementation in Hackage
2010-08-03	A bug: tree got unbalanced
2010-10-19	Submitted a paper to *Journal of Functional Programming* with a Coq proof script
2010-12-18	SLIB incorporated our change

A sequel: a bug in Z3

Z3 is an SMT (satisfiability modulo theory) solver developed by Microsoft.

1. We tried using Z3 instead of Omega Presburger solver
2. Z3 gave a different result from Omega's result
3. Eventually, we found a bug in Z3 and reported it to Nicolaj Bjørner, whose team fixed it.

Bug highlight:

```
assertions.
satisfiable?    -> no
more assertions.
satisfiable?    -> yes
```

Small, One-Shot Project, but It Produced

- a bug fix for libraries
- new rigorous knowledge on a widely-used algorithm
- a modestly complex problem for SMT solvers

Adams, S. (1992).
Implementing sets efficiently in a functional language. Technical report CSTR 92-10.
Tech. rept. University of Southampton.

Nievergelt, J., & Reingold, E. M. (1972).
Binary search trees of bounded balance.
Pages 137–142 of: Proceedings of the fourth annual acm symposium on theory of computing.
ACM.

Pugh, W. (1991).
The Omega test: a fast and practical integer programming algorithm for dependence analysis.
Proc. of the 1991 ACM/IEEE conference on supercomputing.
ACM.

Adams, S. (1992).
Implementing sets efficiently in a functional language. Technical report CSTR 92-10.
Tech. rept. University of Southampton.

Nievergelt, J., & Reingold, E. M. (1972).
Binary search trees of bounded balance.
Pages 137–142 of: Proceedings of the fourth annual acm symposium on theory of computing.
ACM.

Pugh, W. (1991).
The Omega test: a fast and practical integer programming algorithm for dependence analysis.
Proc. of the 1991 ACM/IEEE conference on supercomputing.
ACM.

WBT is Chosen When Storing 2-Bit Information is Costly

AVL / Red-Black Each node stores small information:
height difference / node-state

Weighted Balance Trees Each node stores size of the subtree.
can be used for index operations

- 2 bits can be stored efficiently in LSBs of a pointer in IA32
  but common Haskell / Scheme implementations do not exploit this
- a binary tree with index operations can be used instead of updatable arrays.

An adversarial approach to interaction specifications (Work in progress)

DIKU-IST 2011 / Tokyo

Anders Starcke Henriksen
starcke@diku.dk

Department of Computer Science
University of Copenhagen

January 11, 2011

Motivation

The context for this talk:

- Specification and verification of *concurrent* and *distributed* systems.
- **Key concept**: *interaction*.

This is not a technical talk, we seek to motivate and explain two topics:

- Cooperative vs. adversarial specifications.
- Interactions viewed as games.

Introduction

An *interaction* happens when several parties perform certain *actions* affecting each other (communication scenarios, business processes, computer programs, . . .)

An *interaction specification* is a set of requirements about the interaction:

- Something must happen/something must not happen, choices, deadlines, . . .
- Examples: Protocol specifications, business process languages, workflows, session types . . .

Programming by contract - scenario

Programming by contract (PBC) facilitates modular construction of software, by means of formal specifications.

- Consider a code producer assigned with the task of producing a program satisfying some specification.
- The task is broken down into several modules, each with a formally specified interface (typically with *pre- and postconditions*).
- Several different programmers can now work on the different modules and then if all developed modules satisfy their specifications, the full program will also satisfy its specification.

A cooperative world?

Most formal specifications are done in a *cooperative* way:

- All participants work to fulfill a set of common goals.
  - Every programmer want the final program to work.
- If the goals are not formalized, the participants are expected to follow the *intentions* behind.
  - If the specification does not rule out infinite loops, no programmer should just implement the trivially looping program.
- There is no notion of what happens after a failure to follow the specification
  - In case of a programming error, the module must be corrected before the main program can work.
- Often based on partial correctness.
  - Enables runtime monitoring of specifications.

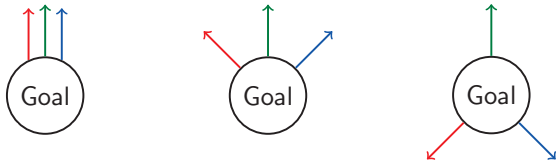
Problems with cooperative settings

The cooperative methodology contains hidden problems:

- It only takes one participant with conflicting goals to "topple the house of cards" (e.g. one participant who refuses to answer anything in a partial correctness setting).
- How to decide whether someone has followed the intentions?
- What if the specification is not followed? Failures do happen.
- If I am allowed to loop - why not loop always (partial correctness), what can an eventually guarantee be used for (except fulfilling another eventually guarantee?)

Problems with cooperative settings - cont.

Participants may seem to have the same goals, but in reality slightly conflicting goals:



Ideal. Actual. Actual?

The original specification could have been a compromise, which is not captured in the cooperative setting.

PBC scenario - cooperative problems

The scenario seems natural at first, but several problems exist:

- The programmers need to fulfill the specification, but is there anything that forces the programmer to use the best solution, or even a good solution? what happens if suboptimal solutions are used?
- How are the programmers paid (lines of code, number of finished modules, ...)? would they prefer a quick solution to a good one?
- What if a third party code is used, can the programmers just blame that? (e.g. a library picked by the programmer).

Adversarial composition

In an *adversarial setting* each participant is an autonomous entity:

- Each participant is potentially an adversary of the other participants.
- Each participant has its own set of goals, not necessary compatible with the goals of the other participants.
- A participant will defect from the "intended" path or even break a whole specification if it is more beneficial in terms of his own goals.
- A participant will still follow rules if the penalty for breaking them or the reward for following them, is large enough. (Cooperation by need)
- It is in general not enough for a participant to just reflect blame for a failure. The participant must assume *responsibility* by *quantifying* the penalty or reward.

PBC - adversarial view

The adversarial viewpoint illuminates the problems:

- If the programmers have their own goal, they might produce suboptimal code, unless they are penalized/rewarded accordingly.
- The consequences for breaking a specification must be clear, i.e. what is the risk. In an adversarial setting it could be broken at any time.
- In particular, we should never let a high-assurance service rely on a low-assurance one (e.g. a third party one).
- Instead of the traditional distinction between a module and its environment, one has to consider the environment not as a monolithic entity, but as a composite entity consisting of multiple parts.

Comparison - cooperative vs. adversarial

- In the cooperative setting the participants share the same goals, and therefore follow the specifications directly.
- In the adversarial setting the participants have their own goals, and are only responsible for actions in terms of rewards and penalties.
- In the cooperative setting the focus is on the "happy" path, and potential failures are ignored.
- In the adversarial setting the focus is on the "unhappy" path, and failures always have to be taken into account.

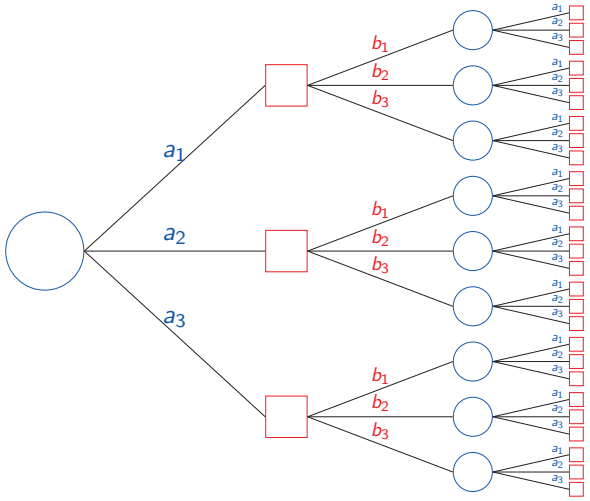
Apparently cooperative settings

Even apparently cooperative settings benefit from an adversarial viewpoint:

- Assuming responsibility in case of failure is more robust then just propagating blame.
- The cooperative participants can be seen as working together against a common adversary (e.g. the laws, Nature), whose goals not necessarily are compatible with the goals of the participants.
- No need to assume that the participants have the same goals.
- One can compare/analyse implementations in terms of the rewards/penalties.

Interaction as games

Adversarial interactions can be seen as *extensive-form games*:



Game-theoretic concepts

The realisation of the game-theoretic concepts are summarised in the following table:

Concept	Realisation
Players	Participants
Moves	Actions
Specifications	Game rules
Payoffs	Rewards/penalties
Strategies	Implementations (programs)
Winning strategies	Correct programs
Dominant strategies	Optimized programs

PBC - game-theoretic view

- To capture interaction, pre- and postconditions are generalized to extensive-form games.
- Because of the adversarial nature, using eventually guarantees as total correctness is not enough.
- Instead, we need a notion of *timed total correctness*, corresponding to real-time *deadlines* (similar to protocol specifications).
  - Enables runtime monitoring.
- Note: Deadlines are not restricted to interaction specifications. When more robust specifications are needed, they also make sense for pre- and postcondition type specifications.

Strategies

When program code is seen as the strategy, the running code can be seen as an *agent* operating on behalf of a *principal* (the programmer):

- When the strategy is chosen, the agent carries out the orders specified by the strategy and the principal can not influence the agent anymore.
  - The running code can not be changed at runtime, unless directly modelled as input.
- The agent does not have knowledge about or is responsible for the games of the principal, the agent might not be able to observe all moves during runtime (e.g. for a delegation scenario).

Formal models

Previous work has devised a concrete game-theoretical model[1]:

- Based on two-party games with quantifiable payoffs.
- Notion of conformance of a program (strategy), p, with respect to a set of specifications (games), written $\models p : s_1, \ldots, s_n$.
  - p will ensure that the total payoff is always non-negative, meaning that it will never be the first to break a specification.
- Compositionality theorem: If $\models p_1 : s, s_1$ and $\models p_2 : \overline{s}, s_2$ and only s mentions the internal communication between p_1 and p_2 then $\models p_1 \| p_2 : s_1, s_2$.
- $\overline{s}$ is the same specification as s but with the roles of the two players interchanged, $p_1 \| p_2$ are the programs running in parallel.

[1] A. S. Henriksen, T. Hvitved, A. Filinski, *A game-theoretic model for distributed programming by contract*.

Formal models - cont.

Current work is concerned with revising the model, key concepts include:

- Specifications based on events instead of actions.
- Hierarchical events (events definable from other events).
  - Compositionality of specification development, refinement of specifications.
- Quantitative vs. qualitative events.
  - Some properties are hard to quantify, but some specifications need to mention those properties alongside quantifiable events.
  - Mix quantitative and qualitative events in the same specification.

Perspective

This work was done in the context of the project *TrustCare: Trustworthy Pervasive Healthcare Services* [2].

- Trustworthy it-systems in the healthcare sector.

We believe that a game-theoretic model based on an adversarial approach would serve well as a foundation for not only interaction specifications for code, but also for e.g. workflows employed in the healthcare domain.

Early work was done in cooperation with Tom Hvitved:

- Ideas were applied to business processes as well.
- Current work separates from THV's work by being closer to game-theory (e.g. payoffs).

[2] www.trustcare.eu

Summary

It might be hard to convince people to use the adversarial model:

- Existing modelling approaches are cooperative, one has to reverse engineer the scenarios behind them.
- Many scenarios seem cooperative at the first glance.
- The adversarial model forces people to consider breaches, failures etc. - which might be very hard to account for.

Regardless, we think that the adversarial model:

- Gets the problems out in the open.
- Is a generalization of the cooperative setting.
- Is fundamentally how the world works.

Jigsaw Radix Conversion

Keisuke Nakano
The University of Electro-Communications

The 4th DIKU-IST workshop, 11.01.11.

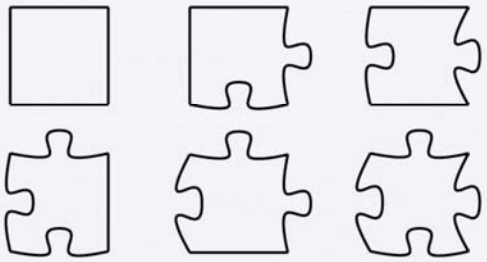
[Q] What is a binary representation for ternary representation "2 0 1" ?

- First compute the number.
  - $2 \cdot 3^2 + 0 \cdot 3^1 + 1 \cdot 3^0 = 19$.
  - Or $3 \cdot (3 \cdot 2 + 0) + 1 = 19$ with Horner scheme.
- Then divide it by 2 repeatedly.
  - $19 = 2 \cdot 9 + 1$;
  - $9 = 2 \cdot 4 + 1$;
  - $4 = 2 \cdot 2 + 0$;
  - $2 = 2 \cdot 1 + 0$; hence we have "1 0 0 1 1".

[Q] How can we solve it w/o +, -, ・, /, and % ?

A solution is ...

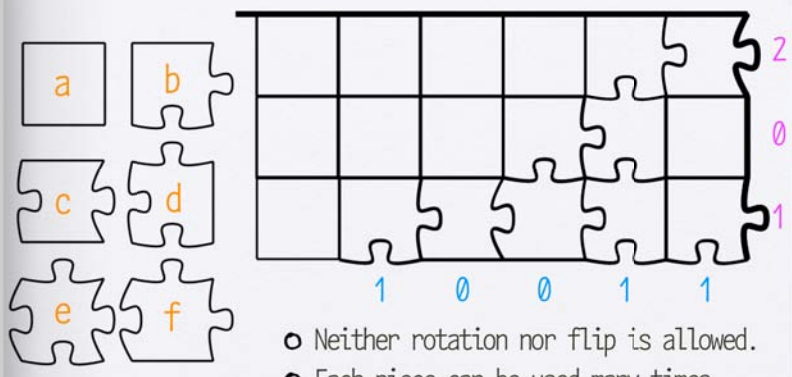
- 6 kinds of jigsaw pieces!



- Each curve stands for a number.

How to execute the radix conversion

[Q] What's a binary number for "201" in ternary?

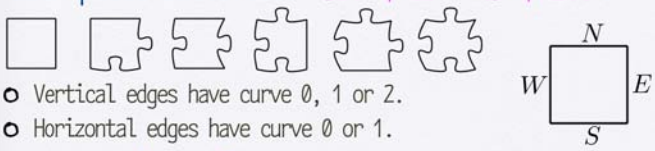


- Neither rotation nor flip is allowed.
- Each piece can be used many times.

4

Reveal the trick

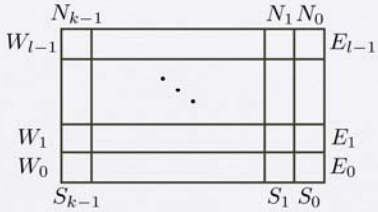
- Each piece satisfies: $3N + E = S + 2W$.



- Vertical edges have curve 0, 1 or 2.
- Horizontal edges have curve 0 or 1.

- Then any $(k \times l)$-board sides satisfy:

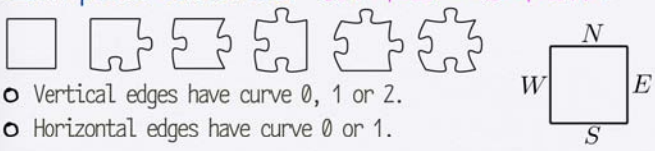
$$3^l \sum_{i=0}^{k-1} 2^i N_i + \sum_{j=0}^{l-1} 3^j E_j = \sum_{i=0}^{k-1} 2^i S_i + 2^k \sum_{j=0}^{l-1} 3^j W_j$$



5

Reveal the trick

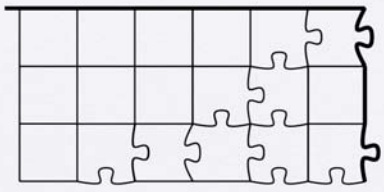
- Each piece satisfies: $3N + E = S + 2W$.



- Vertical edges have curve 0, 1 or 2.
- Horizontal edges have curve 0 or 1.

- Then any $(k \times l)$-board sides satisfy:

$$3^l \sum_{i=0}^{k-1} 2^i N_i + \sum_{j=0}^{l-1} 3^j E_j = \sum_{i=0}^{k-1} 2^i S_i + 2^k \sum_{j=0}^{l-1} 3^j W_j$$



5

Generalization

- m-to-n conversion requires mn kinds of pieces.
  - Each satisfies $mN + E = S + nW$
    - Horizontal: $N, S = 0, \ldots, n-1$
    - Vertical: $E, W = 0, \ldots, m-1$
  - Board sides satisfy

$$m^l \sum_{i=0}^{k-1} n^i N_i + \sum_{j=0}^{l-1} m^j E_j = \sum_{i=0}^{k-1} n^i S_i + n^k \sum_{j=0}^{l-1} m^j W_j$$

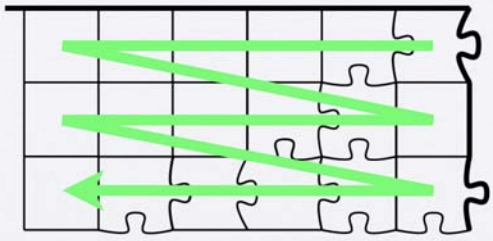
  - It can be generalized for n[th] gray and balanced-3.

6

Feature of the jigsaw method

- North/East edges determine South/West ones.

  - No other result is obtained.
  - There are many possible orders to place pieces.

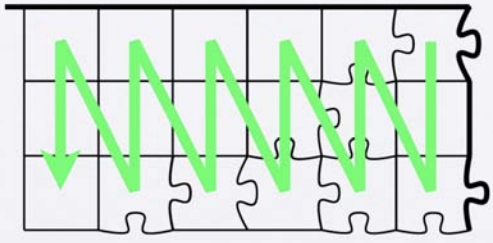


7

Feature of the jigsaw method

- North/East edges determine South/West ones.

  - No other result is obtained.
  - There are many possible orders to place pieces.

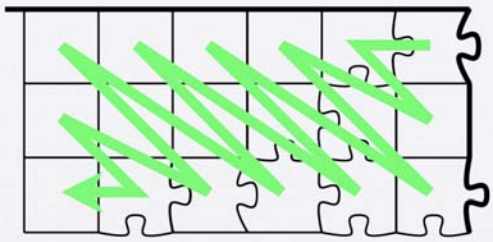


7

Feature of the jigsaw method

- North/East edges determine South/West ones.

  - No other result is obtained.
  - There are many possible orders to place pieces.

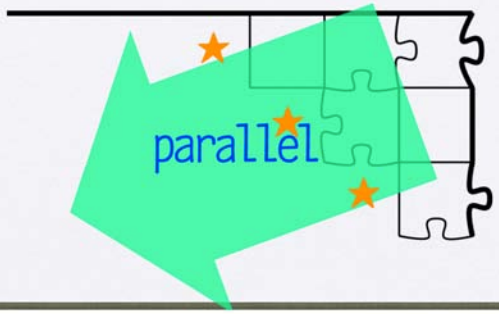


7

Feature of the jigsaw method

- North/East edges determine South/West ones.

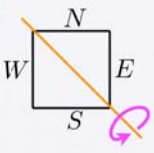
  - No other result is obtained.
  - There are many possible orders to place pieces.



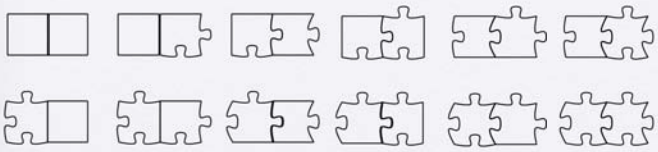
8

Other features

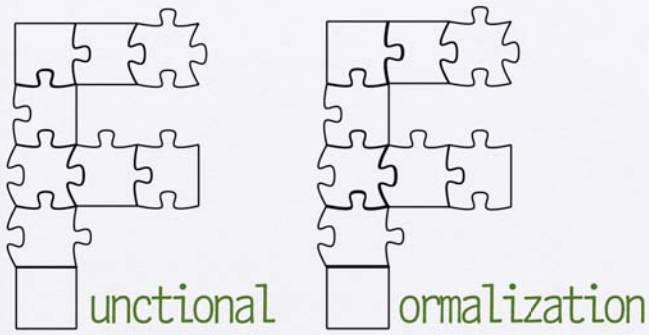
- Inversion (m-to-n conv. ⇒ n-to-m conv.)
  - $mN+E = S+nW \Leftrightarrow nW+S = E+mN$



- Base exponentation
  - Piece combination makes m^k-to-n^l radix conversion
  - 3-to-4 radix conversion (using 3-to-2 pieces)



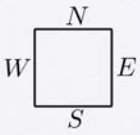
9



Set of jigsaw pieces as function

```
pset::(HCurve,VCurve)→(HCurve,VCurve)
```

- pset determines S and W from N and E.
  - HCurve - horizontal curves for N / S edges
  - VCurve - vertical curves for E / W edges



- Example for 3-to-2 base conversion
  - Set of curves

```
data HCurve = H0 | H1
data VCurve = V0 | V1 | V2
```

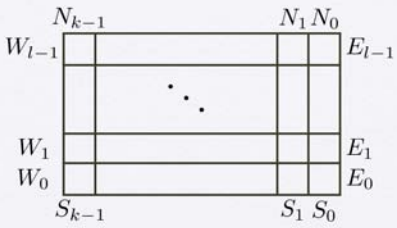
  - Set of pieces


```
pset (H0,V0) = (H0,V0)
pset (H0,V1) = (H1,V0)
pset (H0,V2) = (H0,V1)
pset (H1,V0) = (H1,V1)
pset (H1,V1) = (H0,V2)
pset (H1,V2) = (H1,V2)
```

11

Jigsaw function

```
jigsaw::((H,V)→(H,V))→([H],[V])→([H],[V])
```



H - HCurve
V - VCurve

- We first consider a bounded board.
  - It will be extended for jigsaw radix conversion.
- Jigsaw functions map N/E sides onto S/W sides.
  - (jigsaw pset):: ([H],[V]) → ([H],[V])

12

Auxiliary functions: pflip, mealy

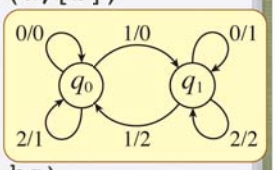
- Argument flipping

```
pflip::((a,b)→(a,b))→(b,a)→(b,a)
pflip f = swap . f . swap
  where swap (x,y) = (y,x)
```

- Mealy machine (automaton with output)

```
mealy_l::((a,b)→(a,b))→(a,[b])→(a,[b])
```

  - $(q_0,[1,2,0,1]) \Rightarrow (q_0,[0,2,1,2])$

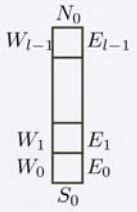



```
mealy_l f (a,[]) = (a,[])
mealy_l f (a,b:bs) = (a',b':bs')
  where (a'',b') = f (a,b)
        (a',bs') = mealy_l f (a'',bs)
```

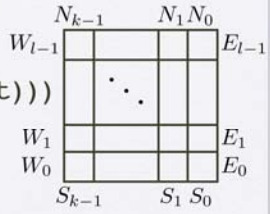
  - `mealy_r` is similarly defined.

Jigsaw constructed by Mealy

- 1-column jigsaw corresponds to mealy
  - `HCurve` - state; `VCurve` - input/output
  - `(mealy_r pset)::(H,[V])→(H,[V])`
    - `mealy_r::((a,b)→(a,b))→(a,[b])→(a,[b])`
    - `pset::(H,V)→(H,V)`



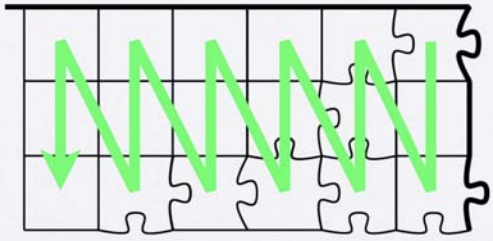
- `pflip(mealy pset)::([V],H)→([V],H)`
- `mealy_l(pflip(mealy_r pset))`
  `:: ([V],[H])→([V],[H])`
- `pflip(mealy_l(pflip(mealy_r pset)))`
  `:: ([H],[V])→([H],[V])`
  ≡ jigsaw pset



On piece placing strategy

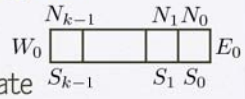
```
jigsaw = pflip . mealy_l . pflip . mealy_r
```

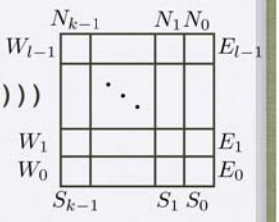
- It corresponds to a 'vertical' strategy.
  - First compute for each vertical line.
  - Then integrate them in horizontal way.



Jigsaw by Mealy, again

- Run 1-raw jigsaw by mealy
  - HCurve - input/output; VCurve - state
  - mealy_l(pflip pset)::(V,[H])→(V,[H])
    - mealy_l::((a,b)→(a,b))→(a,[b])→(a,[b])
    - pset::(H,V)→(H,V)
- pflip(mealy_l(pflip pset))
  ::([H],V)→([H],V)
- mealy_r(pflip(mealy_l(pflip pset)))
  ::([H],[V])→([H],[V])
  ≡ jigsaw pset





16

Jigsaw in vertical / horizontal

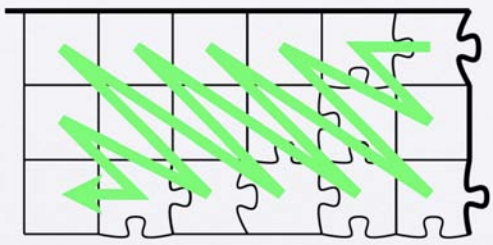
- ... beautiful symmetric equation !

```
  jigsaw

= pflip . mealy_l . pflip . mealy_r

= mealy_r . pflip . mealy_l . pflip
```

17

Other Piece Placing Strategies

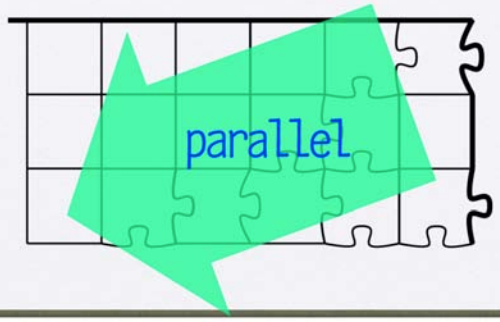
- Possibly neither horizontal nor vertical
  - Diagonal way
  - Random way
  - Parallel way



18

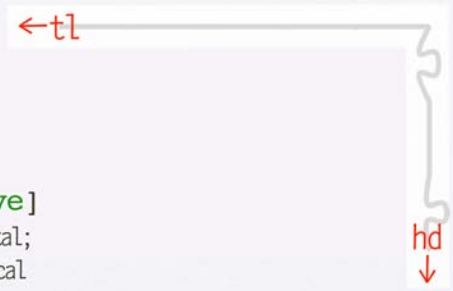
Other Piece Placing Strategies

- Possibly neither horizontal nor vertical
  - Diagonal way
  - Random way
  - Parallel way

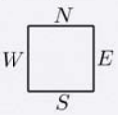


18

Formalization for arbitrary strategies



- Frontier-line pushing approach
  - `type FrLine = [Either HCurve VCurve]`
  - `Left _` (`L`) - horizontal; `Right _` (`R`) - vertical
- Placing a piece ≒ rewriting `[R` E`, L` N`]` to `[L` S`, R` W`]`
  - E, N, S, and W are defined by a piece.



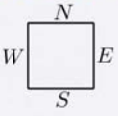
19

Formalization for arbitrary strategies

- Frontier-line pushing approach
  - `type FrLine = [Either HCurve VCurve]`
  - `Left _` (`L`) - horizontal; `Right _` (`R`) - vertical

`[R V1, R V0, R V2, L H0, L H0, L H0, L H0, L H0]`

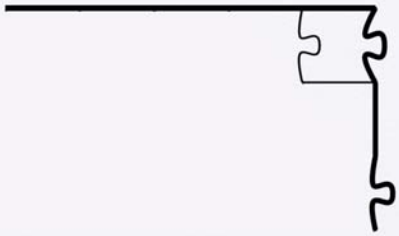
- Placing a piece ≒ rewriting `[R` E`, L` N`]` to `[L` S`, R` W`]`
  - E, N, S, and W are defined by a piece.



19

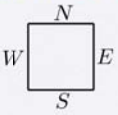
Formalization for arbitrary strategies

- Frontier-line pushing approach
  - `type FrLine = [Either HCurve VCurve]`
  - `Left _` (`L`) - horizontal; `Right _` (`R`) - vertical



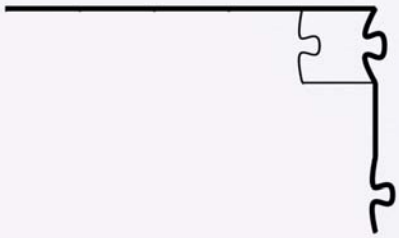
`[R V1, R V0, L H1, R V0, L H0, L H0, L H0, L H0]`

- Placing a piece ≒ rewriting `[R` E`, L` N`]` to `[L` S`, R` W`]`
  - E, N, S, and W are defined by a piece.



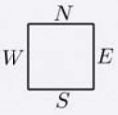
Formalization for arbitrary strategies

- Frontier-line pushing approach
  - `type FrLine = [Either HCurve VCurve]`
  - `Left _` (`L`) - horizontal; `Right _` (`R`) - vertical



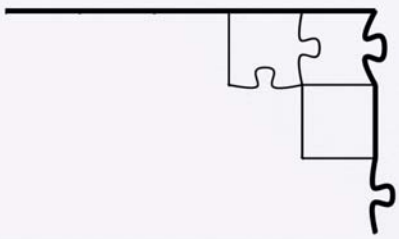
`[R V1, R V0, L H1, R V0, L H0, L H0, L H0, L H0]`

- Placing a piece ≒ rewriting `[R` E`, L` N`]` to `[L` S`, R` W`]`
  - E, N, S, and W are defined by a piece.



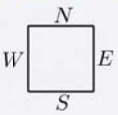
Formalization for arbitrary strategies

- Frontier-line pushing approach
  - `type FrLine = [Either HCurve VCurve]`
  - `Left _` (`L`) - horizontal; `Right _` (`R`) - vertical



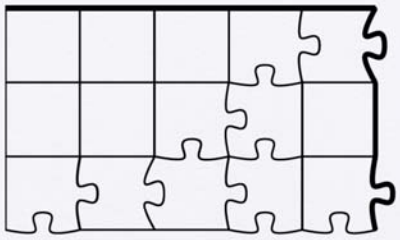
`[R V1, L H0, R V1, L H0, R V0, L H0, L H0, L H0]`

- Placing a piece ≒ rewriting `[R` E`, L` N`]` to `[L` S`, R` W`]`
  - E, N, S, and W are defined by a piece.



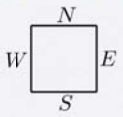
Formalization for arbitrary strategies

- Frontier-line pushing approach
  - `type FrLine = [Either HCurve VCurve]`
  - `Left _` (`L`) - horizontal; `Right _` (`R`) - vertical



`[L H1, L H1, L H0, L H0, L H1, R V0, R V0, R V0]`

- Placing a piece ≒ rewriting `[R` E`, L` N`]` to `[L` S`, R` W`]`
  - E, N, S, and W are defined by a piece.



19

Jigsaw in frontier-line approach

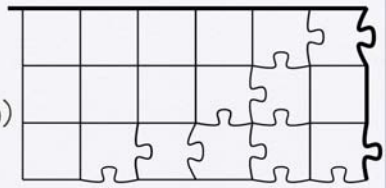
- With non-deterministic pattern (pseudo code)
  - It can runs in parallel.

```
push :: ((a,b)→(a,b)) → [Either a b] → [Either a b]
push f (zs1 ++ [Right b, Left a] ++ zs2) =
  push f (zs1 ++ [Left a', Right b'] ++ zs2)
    where (a',b') = f (a,b)
push f zs = zs

jigsaw :: ((H,V)→(H,V)) → ([H],[V])→([H],[V]))
jigsaw pset (hs,vs) =
  partition (push pset (map Right vs ++ map Left hs))

-- partition :: [Either a b] → ([a],[b])
-- It should work as you expected!
```

20

Infinitely wide board

- Board should be extended until straight-line.
  - How to extend for HCurve (`b0`)
  - When to stop for VCurve (`p`)



- Redefine only `mealy_l` function as `mealy_L`

```
mealy_L::(a→Bool)→b→((a,b)→(a,b))→(a,[b])→(a,[b])
mealy_L p b0 f (a,[]) =
  if p a then (a,[]) else (a',b':bs')
  where (a'',b') = f (a,b0)
        (a',bs') = mealy_L p b0 f (a'',[])
mealy_L p b0 f (a,b:bs) = (a',b':bs')
  where (a'',b') = f (a,b)
        (a',bs') = mealy_L p b0 f (a'',bs)
```

21

Inf. jigsaw in horizontal/vertical

- Similar equations hold!

```
  jigsawX p b0

= pflip . mealy_L (all p) b0 . pflip . mealy_r

= mealy_r . pflip . mealy_L p b0 . pflip
```

22

Frontier-line approach for inf.

- Just modify push function

```
pushX :: (a→Bool)→b→
  ((a,b)→(a,b))→[Either a b] → [Either a b]
pushX p b0 f (zs1 ++ [Right b, Left a] ++ zs2) =
  pushX p b0 f (zs1 ++ [Left a', Right b'] ++ zs2)
    where (a',b') = f (a,b)
pushX p b0 f zs | cond  = zs
-- cond holds iff no [R,L] and all L satisfy p
pushX p b0 f zs = pushX p b0 f (zs++[Left b0])

jigsawX :: (a→Bool)→b→
  ((H,V)→(H,V)) → ([H],[V])→([H],[V]))
jigsawX p b0 pset (hs,vs) =
  partition
    (pushX p b0 pset (map Right vs ++ map Left hs))
```

23

Jigsaw theorem

If g(z,y')=(y,z')
  ∧ f x = (y,x') ∧ f b0 = (_,b0)
    ⇒ f (g z x) = (y, g z' x'),

then jigsawX (...p...) b0
  ((id×fst.g⁻¹).f.g.swap)
    = unfoldl p f . foldr g b0

- cf. Streaming theorem [BirdGibbons03]

  If f z = (y,z') ⇒ f(g(z,x)) = (y, g(z',x))

  Then stream f g = unfoldr f . foldl g

24

Summary and further work

- We introduced Jigsaw radix conversion method.
  - The trick has been revealed.
  - Functional formalization is presented.
- Now working on:
  - Better formulation of jigsaw theorem
    - `jigsaw (.. f . g ..) = unfoldl f . foldr g`
  - Jigsaw fusion
    - Find `h` such that `jigsaw f . jigsaw g = jigsaw h`
    - `f` and `g` should have right inverse, I expect.

25

Other topics

- Application
  - Parsing lin. conj. grammars (like trellis automaton)
  - Sort (not efficient)
- Universality
  - Frontier-line simulates Turing machine
  - Related with Wang's tile (domino problem)
- (Parallel?) complexity
  - Depends on what to be primitive

26

Compositional Data Types

A Report from the Field

Patrick Bahr
paba@diku.dk

University of Copenhagen, Department of Computer Science

Fourth DIKU-IST Joint Workshop on
Foundations of Software,
January 10-14, 2011

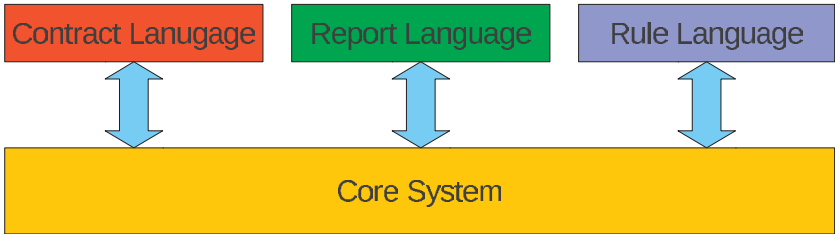
joint work with Tom Hvitved and Morten Ib Nielsen

Outline

Outline

The Setting

Domain-Specific Languages in POETS



- We have a number of domain-specific languages.
- Each pair shares some common sublanguage.
- All of them share a common language of values.
- We have the same situation on the type level!

How do we implement this system without duplicating code!

How Can we Compose Data Structures?

. . . and Functions Defined on Them?

- This is easy on non-recursive data structures.
- Composition by sum or product.

For recursively defined data structures this is different.

Example (A simple expression language)

```
data Expr = Val Int
          | Add Expr Expr

eval :: Expr -> Int
eval (Val x ) = x
eval (Add x y) = eval x + eval y
```


Compositional Data Types

Expression Problem [Phil Wadler]

The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety.

"Data Types à la Carte" by Wouter Swierstra (2008)

A solution to the expression problem: Decoupling!

- data types: decoupling of signature and term construction
  - isolated signature (expression data type without recursion)
  - explicit recursive construction of terms over arbitrary signatures
- functions: decoupling of pattern matching and recursion
  - functions are defined on signatures
  - recursion is added separately
- signatures (+ functions defined on them) can be composed

Decoupling Signature and Term Construction

The data type contains both the signature of operations and the inductive definition of terms over them through recursion.

```
data Expr  = Val Int
           | Add Expr Expr
```

Remove recursion from the definition

```
data Sig e = Val Int
           | Add e e
```

Recursion can be added separately

```
data Term f = Term (f (Term f))
```

`Term Sig` ≅ `Expr`

Combining Signatures

In order to extend expressions, we need a way to combine signatures.

Direct sum of signatures

Type constructor `:+:` of kind `(* -> *) -> (* -> *) -> (* -> *)`:

```
data (f :+: g) e = Inl (f e) | Inr (g e)
```

Example

```
data Sig e = Val Int
           | Add e e

data Val e = Val Int
data Add e = Add e e
```

`Val :+: Add` ≅ `Sig`

Separating Function Definition from Recursion

Compositional function definitions as algebras

In the same way as we defined the types:

- define functions on the signatures (non-recursive): `f a -> a`
- apply the resulting function recursively on the term: `Term f -> a`
- combine functions using type classes

Algebras

```
class Eval f where
  evalAlg :: f Int -> Int
```

Applying a function recursively to a term

```
algHom :: Functor f => (f a -> a) -> Term f -> a
algHom f (Term t) = f (fmap (algHom f ) t)
```

Defining Algebras

On the singleton signatures

```
instance Eval Val where
  evalAlg (Val x) = x
instance Eval Add where
  evalAlg (Add x y) = x + y
```

On sums of signatures

```
instance (Eval f , Eval g)
  => Eval (f :+: g) where
    evalAlg (Inl x) = evalAlg x
    evalAlg (Inr y) = evalAlg y
```

Applying the resulting unique homomorphism to terms

```
eval :: (Functor f, Eval f) => Term f -> Int
eval = algHom evalAlg
```


Outline

Using Compositional Data Types

Using Compositional Data Types in POETS

- Coarse-grained partition into only a few atomic signatures
  - one for base values
  - one for shared operations
  - operations for each individual language
  - syntactic sugar for each individual language
- similar on the type language

Now that we have this structure in place, can we make further use of it?

Products on Signatures

Annotate Syntax Trees, e.g. with source positions

- annotations are not part of the actual language
- annotations should be added separately (to the signature)
- functions that are agnostic to annotations should not care about them

Constant Products on Signatures

Type constructor `:*:` of kind `(* -> *) -> * -> (* -> *)`:

```
data (f :*: a) e = f e :*: a
```

Example

```
data Sig' e = Val Int SrcPos
            | Add e e SrcPos

                Sig' ≅ Sig :*: SrcPos
```


Dealing with Annotations

Strip away annotations

```
stripP :: (s :*: p) a -> s a
stripP (v :*: _) = v

stripPTerm :: (Functor s)
           => Term (s:*:p) -> Term s
stripPTerm = algHom (Term . stripP)
```

Ignoring annotations

```
liftPTerm :: (Functor s)
       => (Term s -> t) -> (Term (s :*: p) -> t)
liftPTerm f = f . stripPTerm
```

This can be extended to annotations on signature built with sums.

Limitations

Propagation of annotations

How can we lift a function `Term f -> Term g`
to a function `Term (f :*: p) -> Term (g :*: p)`?

- Even if function is given as algebra `a :: f (Term g) -> Term g` this does not work:
  `a . fmap stripP` is of type `f (Term (g :*: p)) -> Term g`
- We could derive an algebra from that, but then result has uniformly the same annotation.

Composition of algebras

Given two algebras `a :: f (Term g) -> Term g` and `b :: g B -> B`, how do we compose them to an algebra `f B -> B`?

- Straightforward composition `homAlg b . a` is of type `f (Term g) -> A`

An Example

Example (Syntactic Sugar)

```
type Exp = Core :+: Sugar

desugarAlg :: Exp (Term Core) -> Term Core

desugar :: Term Exp -> Term Core
desugar = algHom desugarAlg
```

Specialising Algebras

Problem

```
desugarAlg :: Exp (Term Core) -> Term Core
```

- Algebras are too general!
- We have to employ the fact that the domain consists of terms!
- We need something more polymorphic!

First attempt: Signature Transformation

```
desugarAlg :: Exp a -> Core a
```

- This is often too restrictive!
- Each "layer" of a term over Exp has to be transformed into exactly one "layer" of a term over Core.
  - $x > y \quad \rightsquigarrow \quad y < x$ ✔
  - $x - y \quad \rightsquigarrow \quad x + (-y)$ ✘

Contexts and Term Homomorphisms

Generalise terms to contexts

```
data Context f a = Term (f (Term f))
                 | Hole a
```

From signature transformations to term homomorphisms

```
d              E        Context Core a
```

Term homomorphisms

- `type TermHom f g = forall a . f a -> Context g a`
- Term homomorphisms (a.k.a. tree homomorphisms) are the term algebras that are defined uniformly. Hence, the polymorphism!

Applying term homomorphisms

```
termHom :: (Functor f, Functor g)
  => TermHom f g -> Term f -> Term g
```

Propagating Annotations

Propagating Annotations

```
constP :: (Functor f)
       => p -> Context f a -> Context (f :*: p) a
constP p (Hole a) = Hole a
constP p (Term t) = Term (fmap (constP p) t :*: p)

liftPTermAlg :: (Functor g)
     => TermHom f g -> TermHom (f :*: p) (g :*: p)
liftPTermAlg f (v :*: p) = constP p (f v)
```

composing term homomorphisms (and algebras)

```
compTermHom :: (Functor g, Functor h) =>
   TermHom g h -> TermHom f g -> TermHom f h
compAlg :: (Functor g) =>
   (g a -> a) -> TermHom f g -> (f a -> a)
```


Terms as Contexts without Holes

Contexts with GADTs

```
data Cxt :: * -> (* -> *) -> * -> * where
            Term :: f (Cxt h f a) -> Cxt h f a
            Hole :: a -> Cxt Hole f a

type Context = Cxt Hole
type Term f = Cxt NoHole f Nothing

data Hole
data NoHole
data Nothing
```

- ⇝ Generalise initial algebra semantics to free algebra semantics.
- ⇝ Terms & initial algebras are a special case.

Other Extensions

- monadic algebras
  - using generalised `sequence :: [m a] -> m [a]`
- (monadic) coalgebras
  - generating terms ⇝ e.g. for QuickCheck
- generic functions
  - e.g. size, querying, unification, matching ...
  - using generalised `foldl :: (a -> b -> a) -> a -> [a] -> b`
- generic term rewriting
  - e.g. for performing program transformations
- mutually recursive data types [Yakushev et al. 2009]
  - by adding additional type argument to the signatures
  - can be extended to rational sets of trees (by bottom-up tree automata on the type level)

Performance Impact

- Composable data types simplify function definitions, provide flexibility, reduce boilerplate code and avoid code duplication!
- But how does it affect runtime performance?

The setting

- Three signatures:
  - values: integers, Booleans, pairs
  - core language operations: $+$, $*$, if, $=$, $<$, $\wedge$, $\neg$, projections
  - syntactic sugar: negation, $-$, $>$, $\vee$, $\Rightarrow$
- a number of different typical functions:
  - type inference
  - evaluation to normal form,
  - "desugaring" (reduce syntactic sugar to the core language)
  - computing free variables
- We compare this to an ordinary implementation using standard data types and recursive functions.

Runtime Comparison

slowdown factors compared to standard data types

function	n=16	n=63	n=1290	n=111,279
desugarType	4.8	5.2	5.3	4.1
desugarType'	4.2	4.9	5.0	2.5
typeSugar	3.2	3.7	3.7	4.6
desugarEval	15	11	11	15
desugarEval'	13	10	9.8	8.8
evalSugar	12	9.4	7.4	18
desugarEvalPure	11	7.1	6.4	11
desugarEvalPure'	6.5	4.4	4.0	3.8
evalSugarPure	7.3	7.0	4.0	3.6
freeVars	1.3	1.6	1.4	1.6
desugar	0.33	0.08	$1.2 \cdot 10^{-3}$	$1.5 \cdot 10^{-5}$

- monadic functions are in blue
- underlined variants use composition of algebras

Outline

Applications for Compositional Data Types

Drawbacks

- not as straightforward as ordinary data types
- type errors are sometimes hard to decypher
- memory and runtime overhead

Benefits

- minimises code duplication
- functions on shared structures can be shared as well
- it is often more convenient to define functions
- more flexible (algebras can be easily modified / lifted)
- only little runtime overhead
- sometimes asymptotically faster that ordinary recursive functions on recursive data types

References

Wouter Swierstra.
Data types à la carte.
Journal of Functional Programming, 2008.

A. R. Yakushev, S. Holdermans, A. Löh and J. Jeuring.
Generic programming with fixed points for mutually recursive datatypes.
Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, 2009.

From Parametric Polymorphism to Balanced Tree Structures for Parallel Programming

Akimasa Morihata (RIEC, Tohoku Univ.)

Joint work with:
Kiminori Matsuzaki (Kochi Univ. Tech.)

Why Parallel Programming is Hard?

- C programmers love arrays and for-loops
- C++ programmers love vectors and iterators
- ML programmers love lists and foldl
- Haskell programmers love lists and foldr

All are inherently sequential!!

Parallel programming requires another data structure/recursion structure

Structure for Parallel Programming

```
sum [a]    = a
sum (a:x) = a + sum x
```

Bad (Sequential)

```
sum [a]       = a
sum (x ++ y) = sum x + sum y
```

Best!! (Optimally Parallel)

```
sum (Tip a)      = a
sum (Fork x y)  = sum x + sum y
```

Good (Parallel)

Balanced Trees for Parallel Program

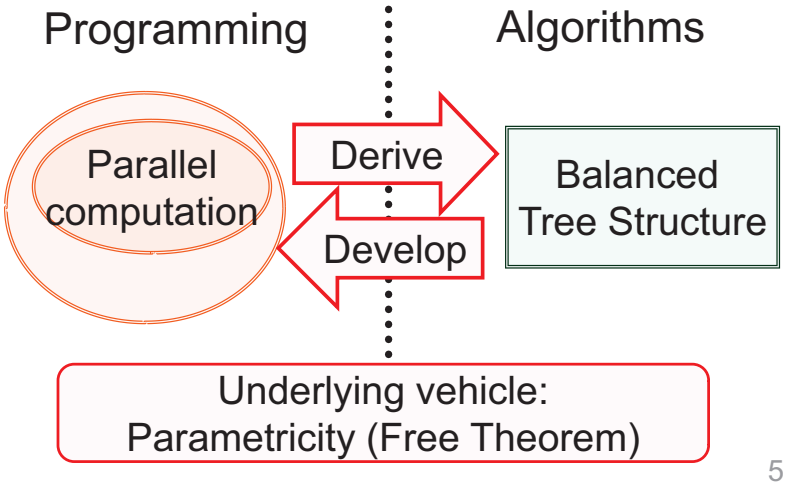
Parallel programmers loves **balanced trees!!**

A methodology for balanced tree is required

- How can we relate parallel algorithms and balanced trees?
- How can we develop programs on balanced trees?

4

Overview



5

Step 1: Extract Merging Operations

```
sum [a]       = a
sum (x ++ y) = sum x + sum y
```

merge results of independent sublists

➔ It raises a polymorphic function

$h :: \forall\beta.\ (A \to \beta) \to (\beta \to \beta \to \beta) \to [A] \to \beta$

Q. Does $h\ (\lambda a.\ a)\ (+) = sum$ hold for any h having this type?

7

Step 2: Specify Requirements

h :: $\forall\beta$. (A → β) → (β → β → β) → [A] → β

The type does not guarantee h (λa.a) (+) = sum
(cf. h f (⊕) (a:x) = f a ⊕ f a ⊕ h f (⊕) x)

Requirement: Don't discard, duplicate, shuffle.
⇔ h (λa. [a]) (++) x = x

Thm. For such h, h (λa. a) (+) = sum

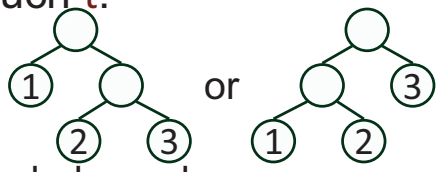
Proof. From parametricity

Step 3: Construct Balanced Trees

Given x :: [A], regard t = h Tip Fork x as a "*balanced tree representation*" of x

```
data BTree a = Tip a | Fork (BTree a) (BTree a)
```

- There are many such t:

h Tip Fork [1,2,3] ∼ 

➔ We can choose a balanced one

Summary: Development of Balanced Tree

- sum raises a family of polymorphic functions
  h :: $\forall\beta$. (A → β) → (β → β → β) → [A] → β
  ∧ h (λa. [a]) (++) x = x
- h Tip Fork yields a balanced tree based on the flexibility of choosing h
  – Indeed, it is the binary search tree
- Programs on the balanced trees should be efficient parallel programs

Programming on Balanced Trees

We would like to perform f :: [A] → B

➔ Work out f' such that

f' (h Tip Fork x) = f x = f (toList (h Tip Fork x))

by fusion (deforestation)

```
toList (Tip a)      = [a]
toList (Fork x y) = toList x ++ toList y
```

☆ No parallel programming issues are there

11

Example: scan

```
scan (⊕) e [a]    = [e]
scan (⊕) e (a:x) = e : scan (⊕) (e ⊕ a) x
```

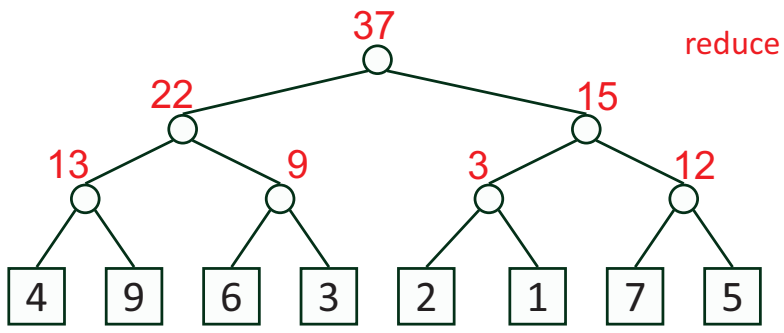
Let's derive scan' (⊕) e x = scan (⊕) e (toList x)

```
scan' (⊕) e (Tip a) = scan (⊕) e (toList (Tip a))
                    = [e]
scan' (⊕) e (Fork x y)
  = ...
  = scan' (⊕) e x ++ scan' (⊕) (e ⊕ reduce' (⊕) x) y
```

12

scan on Balanced Trees

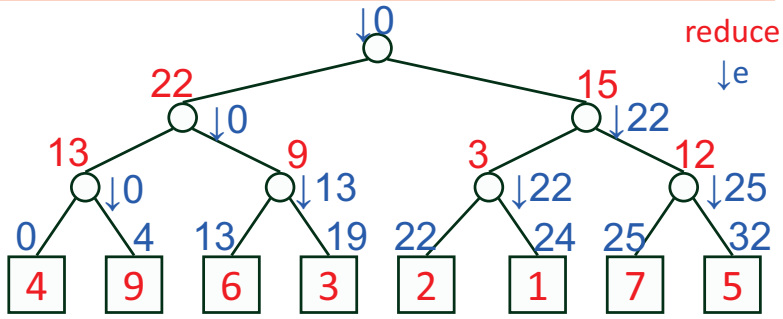
```
reduce' (+) (Tip a) = a
reduce' (+) (Fork x y) = reduce' (+) x + reduce' (+) y
```



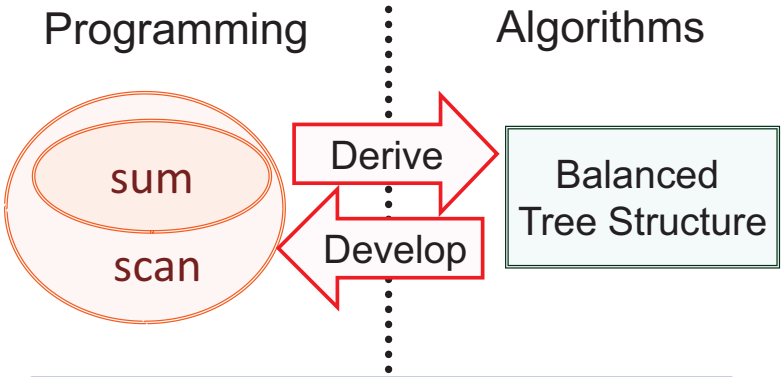
13

scan on Balanced Trees (Contd.)

```
scan' (+) e (Tip a) = [e]
scan' (+) e (Fork x y)
  = scan' (+) e x ++ scan' (+) (e + reduce' (+) e x) y
```



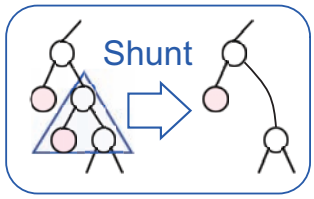
Summary



Our method scales from lists to trees

Shunt Contraction Algorithm
(Abrahamson et al.'89)

- Collapse a tree by primitive contraction operations called "Shunt"
  - time $O(n/p + \log p)$



- The process of collapse provides a good scheduling of gathering information
  - expression evaluation, register allocation, XPath querying …

Deriving Balanced Tree

Consider

h :: ∀β. (A→β) →

(β→β→β→β, β→β→β→β, β→β→β→β) →

Tree A → β s.t. h wrap connect x = x

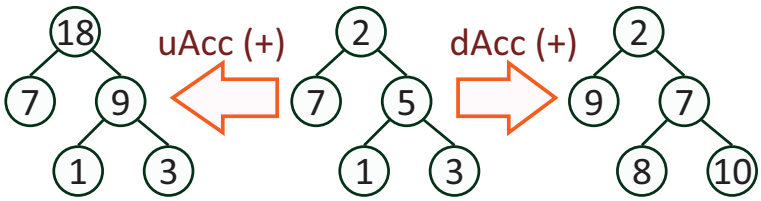
(connect reconstructs the tree according to Shunt)

h Leaf (NN, NL, NR) t

∼ ordered topology tree (Frederickson'97)

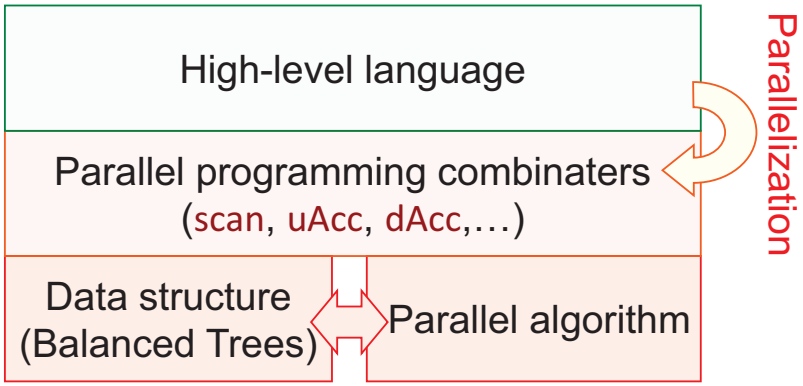
– height **O(log *n*)**, connect/cut: **O(log *n*)** time

Purely Functional Implementations of Tree Accumulations (Gibbons et al.'94)



- Known implementations (Gibbons et al.'94)
  Shunt contraction + stacks of pointers
- Our implementations:
  Balanced tree + upsweep + downsweep
  (as similar to scan)

Conclusion & Future Work



A Homomorphism-based MapReduce Framework for Systematic Parallel Programming

Yu Liu

The Graduate University for Advanced Studies

Jan 12, 2011

Outline

1. Motivations
2. Brief introduction of MapReduce
3. The Homomorphism-based Framework
4. Case Study: Parallel sum, Maximum prefix sum, Variance of numbers
5. Experimental Results

Motivation of This Talk

- Show how to make programming with MapReduce easier.

Motivation of This Talk

- Show how to make programming with MapReduce easier.
- Introduce an approach of automatic parallel program generating.

MapReduce Programming model

The Computation of MapReduce Framework

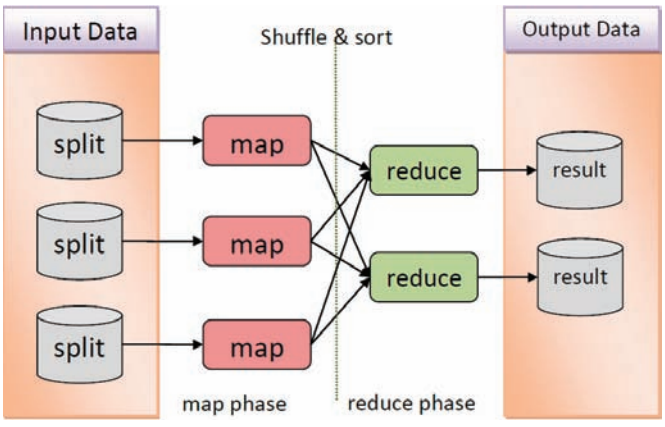
Google's MapReduce is a parallel-distributed programming model, together with an associated implementation, for processing very large data sets in a massively parallel manner.

Components of a MapReduce program (Hadoop)

- A *Mapper*;
- A *Partitioner* that can be used shuffling data;
- A *Combiner* that can be used doing local reduction;
- A *Reducer* ;
- A *Comparator* can be used while sorting or grouping;

MapReduce Programming model

MapReduce Data-processing flow



MapReduce Programming model

A simple functional specifcation of the MapReduce framework

$$
\begin{array}{l}
MapReduce :: ((k_1, v_1) \to [(k_2, v_2)]) \to ((k_2, [v_2]) \to v_3) \to \{(k_1, v_1)\} \to \{(k_2, v_3)\} \\
MapReduce\ mapper\ reducer\ set\_kv = \textbf{let}\ \ set\_list\_kv = map_{\mathbf{S}}\ mapper\ set\_kv \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad set\_k\_list\_v = groupByKey\ set\_list\_kv \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ map_{\mathbf{S}}\ (\lambda(k, vs).(k, reducer\ (k, vs)))\ set\_k\_list\_v
\end{array}
$$

Function $map_{\mathbf{S}}$ is a set version of the *map* function. Function $groupByKey :: \{[(k, v)]\} \to \{(k, [v])\}$ takes a set of list of key-value pairs (each pair is called a record) and groups the values of the same key into a list.

Maximum Prefix Sum problem

The Maximum Prefix Sum problem (mps) is to find the maximum prefix-summation in a list:

$$\underline{3, -1, 4, 1, -5, 9, 2}, -6, 5$$

This problem seems not obvious to solve this problem efficiently with MapReduce.

List Homomorphism

Function h is said to be a list homomorphism

If there are a function f and an associated operator $\odot$ such that for any list x and list y

$$
\begin{array}{lcl}
h\ [a] & = & f\ a \\
h\ (x +\!\!+\ y) & = & h(x) \odot h(y).
\end{array}
$$

Where $+\!\!+$ is the list concatenation.

For instance, the function sum can be described as a list homomorphism

$$
\begin{array}{lcl}
sum\ [a] & = & a \\
sum\ (x +\!\!+\ y) & = & sum\ x + sum\ y.
\end{array}
$$

List Homomorphism and Homomorphism Theorems

Leftwards function

Function h is leftwards if it is defined in the following form with function f and operator $\oplus$,

$$\begin{aligned} h\ [a] &= f\ a \\ h\ ([a] +\!\!+\ x) &= a \oplus h\ x. \end{aligned}$$

Rightwards function

Function h is rightwards if it is defined in the following form with function f and operator $\otimes$,

$$\begin{aligned} h\ [a] &= f\ a \\ h\ (x +\!\!+\ [a]) &= h\ x \otimes a. \end{aligned}$$

List Homomorphism and Homomorphism Theorems

Map and Reduce

For a given function f, the function of the form $([[\cdot] \circ f, +\!\!+])$ is a map function, and is written as $map\ f$.

The function of the form $([id, \odot])$ for some $\odot$ is a reduce function, and is written as $reduce\ (\odot)$.

The First Homomorphism Theorem

Any homomorphism can be written as the composition of a map and a reduce:

$$([f, \odot]) = reduce\ (\odot) \circ map\ f.$$

List Homomorphism and Homomorphism Theorems

The Third Homomorphism Theorem

Function h can be described as a list homomorphism, iff $\exists\ \odot$ and $\exists\ f$ such that:

$$h = ([f, \odot])$$

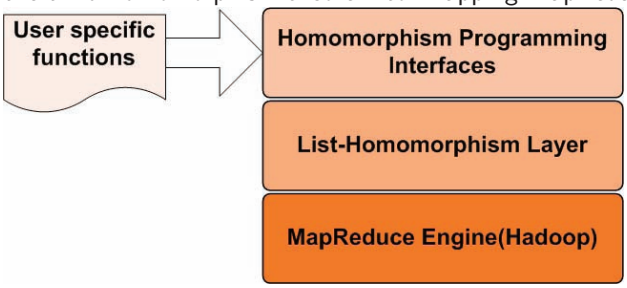
if and only if there exist f, $\oplus$, and $\oplus$ such that

$$\begin{aligned} h\ [a] &= f\ a \\ h\ ([a] +\!\!+\ x) &= a \oplus h\ x \\ h\ (x +\!\!+\ [b]) &= h\ x \otimes b. \end{aligned}$$

The third homomorphism gives a necessary and sufficient condition for the existence of a list homomorphism.

A homomorphism-based framework wrapping MapReduce

To make it easy for resolving problems such as mps by MapReduce. We using the knowledge of homomorphism especially the third homomorphism theorem to wrapping MapReduce model.



A homomorphism-based framework wrapping MapReduce

Basic Homomorphism-Programming Interface

$$\begin{aligned} \mathit{filter} &:: a \to b \\ \mathit{aggregator} &:: b \to b \to b. \end{aligned}$$

The implementlation on Hadoop

```
 * @author liuyu
 */
public interface BasicListHomomorphismInterface<T1 extends Object,T2 extends Object> {
    T2 filter(T1 val);
    T2 aggregator(ArrayList<T2> value);
}
```


A homomorphism-based framework wrapping MapReduce

A simple example of using this interface for computing the sum of a list

The implementlation on Hadoop

```
 */
public class SUM implements BasicListHomomorphismInterface<LongWritable, LongWritable> {

    private static final Log LOG = LogFactory.getLog(SUM.class);
    public static LongWritable sum(ArrayList<LongWritable> xs) {
        Long rst = 0l;
        for (LongWritable var : xs) {
            rst += var.get();
        }
        return new LongWritable(rst);
    }

    public LongWritable filter(LongWritable val) {
        return val;
    }

    public LongWritable aggregator(ArrayList<LongWritable> values) {
```

A homomorphism-based framework wrapping MapReduce

Programming Interface with Right Inverse

$$\begin{aligned} fold &:: [a] \to b \\ unfold &:: b \to [a]. \end{aligned}$$

The implementlation on Hadoop

```
public interface ThirdHomomorphismInterface<T1 extends Object,T2 extends Object>  {
    String folding_method ="fold";
    String unfolding_method  ="unfold";
    public   T2 fold(  ArrayList<T1> values);
    public   ArrayList<T1> unfold(T2 value);
}
```


A homomorphism-based framework wrapping MapReduce

A simple example of using this interface for computing the sum of a list

The implementlation on Hadoop

```
public class SUM2 implements ThirdHomomorphismInterface<LongWritable, LongWritable> {

    public LongWritable fold(ArrayList<LongWritable> values) {
        return sum(values);
    }

    public ArrayList<LongWritable> unfold(final LongWritable value) {
        return new ArrayList<LongWritable>() {
            { add(value); }
        };
    }

    public static LongWritable sum(ArrayList<LongWritable> xs) {
        Long rst = 0l;
        for (LongWritable var : xs) {
            rst += var.get();
        }
```


A homomorphism-based framework wrapping MapReduce

Requirements of using this interface in addition to the right-inverse property of *unfold* over *fold*.

Both leftwards and rightwards functions exist

$$\begin{aligned} fold([a] +\!\!+ x) &= fold([a] +\!\!+ unfold(fold(x))) \\ fold(x +\!\!+ [a]) &= fold(unfold(fold(x)) +\!\!+ [a]). \end{aligned}$$

The implementation of homomorphism framework upon Hadoop

To implement our programming interface with Hadoop, we need to consider how to represent lists in a distributed manner.

Set of pairs as list

We use integer as the index's type, the list $[a, b, c, d, e]$ is represented by $\{(3, d), (1, b), (2, c), (0, a), (4, e)\}$.

Set of pairs as distributed List

We can represent the above list as two sub-sets $\{((0, 1), b), ((0, 2), c), ((0, 0), a)\}$ and $\{((1, 3), d), ((1, 4), e)\}$, each in different data-nodes

The implementation of homomorphism framework upon Hadoop

The first homomorphism theorem implies that a list homomorphism can be implemented by MapReduce, at least two passes of MapReduce.

Defination of $hom_{\mathbf{MR}}$

$$hom_{\mathbf{MR}} :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to \{(ID, \alpha)\} \to \beta$$
$$hom_{\mathbf{MR}}\ f\ (\oplus) = getValue \circ MapReduce\ mapper2\ reducer2 \circ MapReduce\ mapper1\ reducer1$$

where

$$mapper1 :: (ID, \alpha)) \to [((ID, ID), \beta))]$$
$$mapper1\ (i, a) = [((pid, i), b)]$$

The implementation of homomorphism framework upon Hadoop

Defination of $hom_{\mathbf{MR}}$

$$reducer1 :: (ID, ID) \to [\beta] \to \beta$$
$$reducer1\ ((p, j), ias)) = hom\ f\ (\oplus)\ ias$$
$$mapper2 :: ((ID, ID), \beta) \to [((ID, ID), \beta)]$$
$$mapper2\ ((p, j), b) = [((0, j), b)]$$
$$reducer2 :: (ID, ID) \to [\beta] \to \beta$$
$$reducer2\ ((0, k), jbs) = hom\ (\oplus)\ jbs$$
$$getValue\ \{(0, b)\} = b$$

Where, $hom\ f\ (\oplus)$ denotes a sequential version of $([f, \oplus])$.

The *leftwards* and *rightwards*function

Derivation by right inverse

$$\begin{aligned} leftwards([a] +\!\!+ x) &= fold([a] +\!\!+ unfold(fold(x))) \\ rightwards(x +\!\!+ [a]) &= fold(unfold(fold\ x) +\!\!+ [a]). \end{aligned}$$

Now if for all xs,

$$rightwards\ xs = leftwards\ xs, \tag{1}$$

then a list homomorphism $([f, \oplus])$ that computes *fold* can be obtained automatically, where f and $\oplus$ are defined as follows:

Outline
Motivation
Brief introduction of background
The Design of Homomorphism-based Framework on MapReduce
Case Study
Performance Evaluation
Automatic Parallelization
Case Study

The *leftwards* and *rightwards*function

Derivation by right inverse

$$\begin{aligned} f\ a &= fold([a]) \\ a \oplus b &= fold(unfold\ a +\!\!+ unfold\ b). \end{aligned}$$

Equation (1) should be satisfied.

Outline
Motivation
Brief introduction of background
The Design of Homomorphism-based Framework on MapReduce
Case Study
Performance Evaluation
Automatic Parallelization
Case Study

Programming with this homomorphism framework
MPS

A sequential program

```
public static LongWritable mps(ArrayList<LongWritable> xs) {
    Long rst = 0l;
    Long tmp = 0l;
    for (LongWritable v : xs) {
        tmp += v.get();
        if (tmp > rst) {
            rst = tmp;
        }
    }
    return new LongWritable(rst);
}
```

Programming with this homomorphism framework
MPS

A tupled function

```
public static Pair<Long> mps_sum(ArrayList<LongWritable> xs) {
    Long mps = 0l;
    Long sum = 0l;
    for (LongWritable v : xs) {
        sum += v.get();
        if (sum > mps) {
            mps = sum;
        }
    }
    return new Pair<Long>(mps, sum);
}
```


MPS

$$
\begin{array}{lcl}
(mps \vartriangle sum)\ [a] & = & (a \uparrow 0, a) \\
(mps \vartriangle sum)\ (x +\!\!+ [a]) & = & \textbf{let}\ (m, s) = (mps \vartriangle sum)\ x\ \textbf{in}\ (m \uparrow (s +
\end{array}
$$

We use this tupled function as the *fold* function. The right inverse of the tupled function, $(mps \vartriangle sum)^{\circ}$:

$$(mps \vartriangle sum)^{\circ}\ (m, s) = [m, s - m]$$

Noting that for the any result (m, s) of the tupled function the inequality $m \geqslant s$ hold,

The implementation of homomorphism framework upon Hadoop
performance tests

Environment:Hardware

COE cluster in Tokyo University which has 192 computing nodes. We choose 16 , 8 , 4 , 2 and 1 node to run the MapReduce-MPS program. Each node has 2 Xeon(Nocona) CPU with 2GB RAM.

Environment:Software

- Linux2.6.26 ,Hadoop0.20.2 +HDFS
- Hadoop configuration: heap size= 1024MB
- maximum mapper pre node: 2
- maximum reducer pre node: 2

Performance

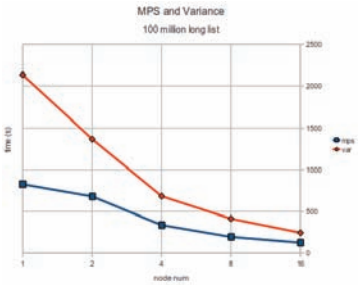
The input data

length of list	value type	file size	compressed
1 milllion	Long	29.5MB	zip
10 milllion	Long	295.5MB	zip
100 milllion	Long	2.95GB	zip

Performance

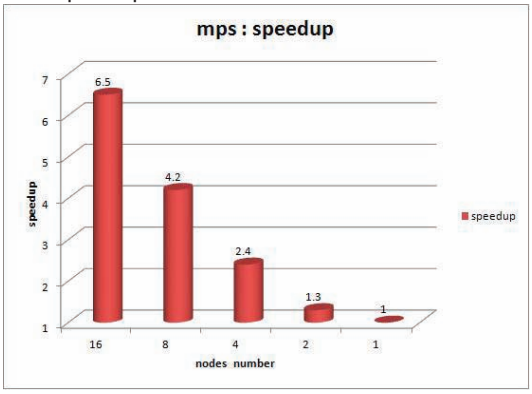
The time consuming of calculate 100 million-long list

(SequenceFile, *Pair* $<$ *Long* $>$):



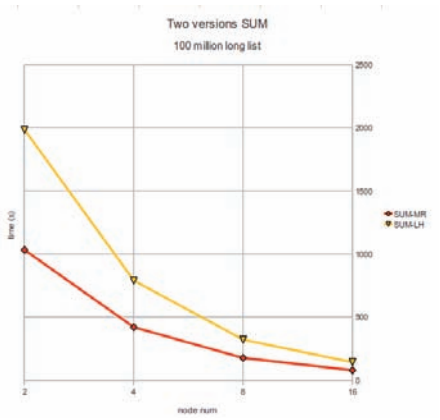
Performance

The speedup of 2-16 nodes:



Performance
Comparison of 2 version SUM

Comparison of 2-16 nodes:



Performance
Conclusions

The time curve indicate the system scalability with the number of computing nodes. The curve between x-axis 2 and 8 has biggest slope, when the curve reaches to 16, the slope decreased, that is because when there are more nodes, the overhead of communication increased. Totally, the curve shows the scalability is near-linear.
Overhead of 2 phases Map-Reduce.
Overhead of Java reflection.
Not support local reduction now (not implemented yet).

The end
Questions?

?

Specification and Implementation of ERP requirements

Mikkel Jønsson Thomsen
jonsson@diku.dk

Department of Computer Science
University of Copenhagen

January 12, 2011

Background

Enterprise Resource Planning (ERP) systems is a category of systems designed to aid the management of resources in a company. The ERP category include systems like Microsoft Dynamics NAV / AX, SAP.

Major ERP systems are centered around the Double-Entry Bookkeeping financial model, and are strongly tied to the database representation of data.

In 2004, 91% of the top 500 danish companies employed an ERP system. 40% were using an MS product, 23% a SAP product.

Modeling ERP requirements

The extraction and formulation of requirements for an ERP system is a very subjective discipline that often rely on the individual experts with extensive contextual knowledge of the industry (Rikhardsson *et al.*, 2004; de Carvalho *et al.*, 2009).

Deploying an ERP system is a huge task that scales with the size of the company deploying it. E.g. it is estimated that the cost of implementing an ERP system represents 1-6% of a company's turnover and that it takes an average of two to three years to complete.

The REA accounting model and the POETS ERP framework

McCarthy proposed the REA accounting model in 1982:

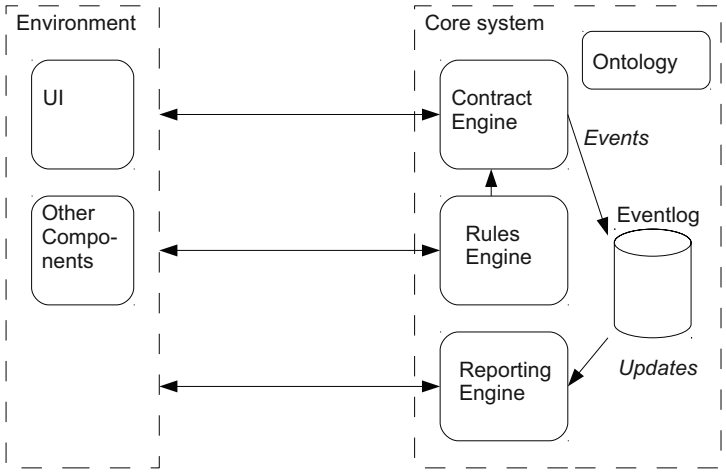
"An *Event* is some *Agent(s)* acting on some *Resource(s)*".

The Process Oriented Event-driven Transaction System (POETS)(Henglein *et al.*, 2009) is an implementation of this model.

"Shortening the distance between requirements and their formal expression for rapid system prototyping, implementation, and continuous adaptation to changing processes and information needs"

POETS(*Data Model*, *Work Flows*) => *ERP System*

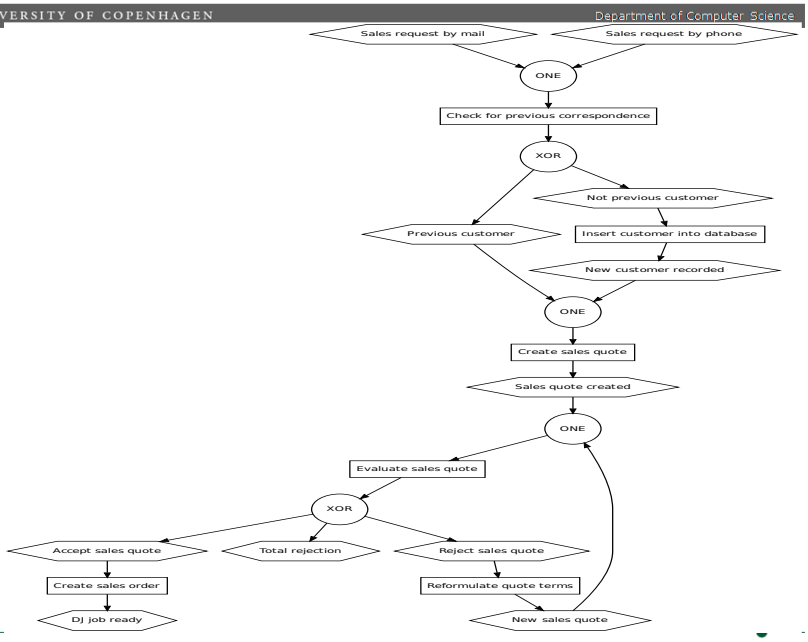
Architectural overview of POETS

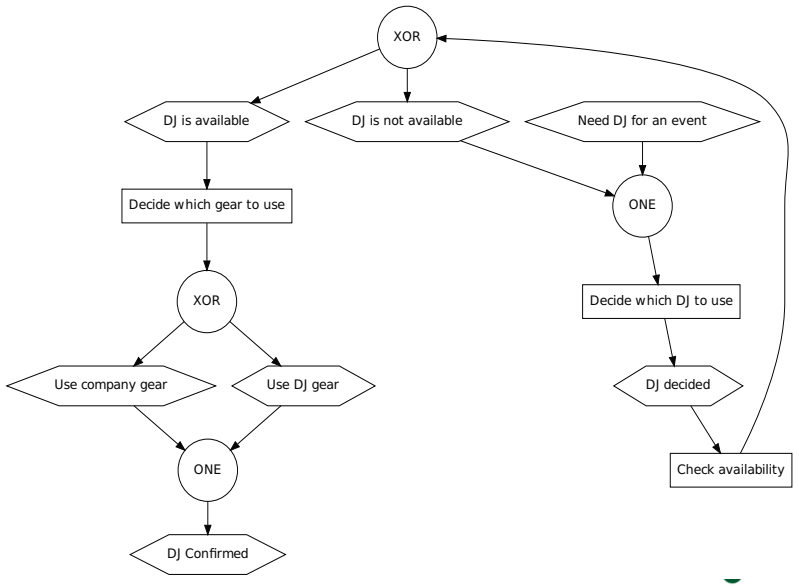


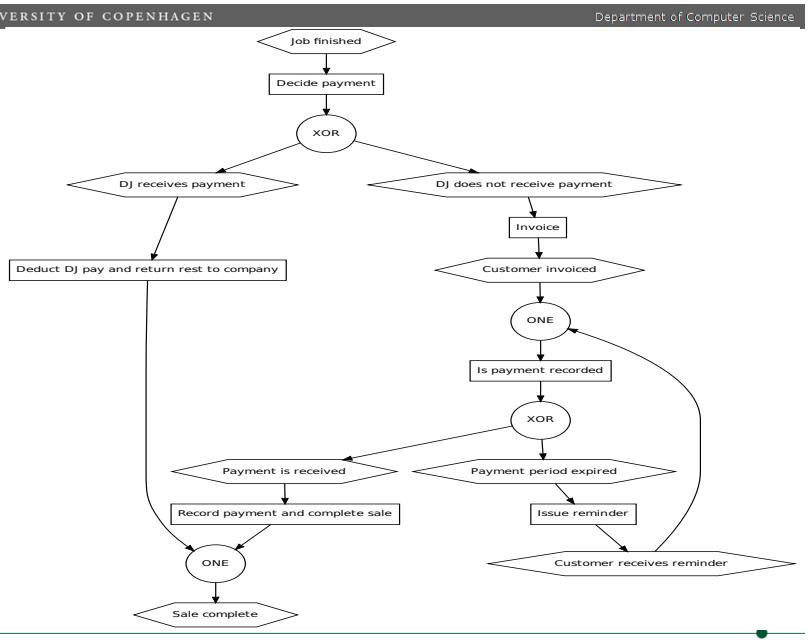
Formulation of a transaction process

Originates from a live case study.

- Create and Negotiate sales quote, including payment terms (Fixed price or hourly rate).
- Enter sales contract (binding).
- Select DJ.
- DJ plays the job.
- DJ must receive a payment for the job. Either directly from the customer or from the company
- Customer is invoiced the amount according to the payment terms.
- Company receives and registers the incoming payment.


Sales request by mail
Sales request by phone
ONE
Check for previous correspondence
XOR
Not previous customer
Previous customer
Insert customer into database
New customer recorded
ONE
Create sales quote
Sales quote created
ONE
Evaluate sales quote
XOR
Accept sales quote
Total rejection
Reject sales quote
Create sales order
Reformulate quote terms
DJ job ready
New sales quote


XOR
DJ is available
DJ is not available
Need DJ for an event
Decide which gear to use
ONE
XOR
Decide which DJ to use
Use company gear
Use DJ gear
DJ decided
ONE
Check availability
DJ Confirmed


Job finished
Decide payment
XOR
DJ receives payment
DJ does not receive payment
Invoice
Deduct DJ pay and return rest to company
Customer invoiced
ONE
Is payment recorded
XOR
Payment is received
Payment period expired
Record payment and complete sale
Issue reminder
ONE
Customer receives reminder
Sale complete

Modeling the Sales Process

What information is needed to model this?

- 3 Agents (Company, Customer, DJ).
- Information about the job, including date, size, type, place, etc.
- Payment details

What is needed to formalize this ontological data model?

- Entities (Records).
- Fields of Entities or simple types.
- Inheritance between entities.

Compact, easily readable syntax for formalization.

Formalizing an ontology language

S	::=	$EntityName_1$ $SupQuan$ $EntityName_2$ **.**
	|	$EntityName_1$ $PropQuan$ $EntityName_2$ $PropDecl$ **.**
	|	$EntityName$ **has comments** $Cstring$ **.**
	|	$EntityName$ **is abstract .**
	|	$EntityName$ $RuleExp$ **.**
$EntityName$	::=	$[A-Z]$ $([A-Za-z][1-9])^*$
$SupQuan$	::=	**is a** | **is an** | **is in**
$PropQuan$	::=	$SingleQuantity$ | $MultipleQuantity$
$PropDecl$	::=	**called** $FieldName$ | **named** $FieldName$ | ϵ
$FieldName$	::=	$[a-z]$ $[A-Za-z]^*$
$Cstring$	::=	**"**$([A-Za-z][\backslash n!.:;?..])^*$ **"**
$RuleExp$	::=	**validates ruleset** F | **is validated by** F
$SingleQuantity$	::=	**has a** | **has an**
$MultipleQuantity$	::=	**has a list of**

Data Model for Contract

```
DJContract is a Contract.
DJContract has a Company.
DJContract has a DJ called dj.
DJContract has a Customer.
DJContract has a FixedPrice called djSalary.
DJContract has a list of GearPackage called gear.
DJContract has a EventType.
DJContract has a int called size.
DJContract has a dateTime called start.
DJContract has a dateTime called tentativeEnd.
DJContract has an Address called eventAddress.
```

Formalizing the work flow

- Using multiparty contracts (Hvitved, 2009)
- Identify "Functions" and "Events" in the work flow.
- Identify deadlines and other temporal parameters.

Conclusion

Extracting / uncovering even the most basic requirements for ERP systems is not a trivial task.

Specialization of the POETS framework with regards to the requirements. This includes User Interfaces.

Specify a knowledge representation language for an ERP data model that is:

- Easily readable.
- Limited to the domain for which it is used.
- Strong enough to express a data model for an ERP system.

Future work

Deployment of a POETS implementation in a live setting will hopefully reveal whether the customizations are indeed easier to perform.

de Carvalho, Rogério Atem, Johansson, Björn, & Manhães, Rodrigo Soares. 2009.
Agile Software Development for Customizing ERPs.
Information Science Reference, IGI Global.
Chap. 2, pages 20–39.

Henglein, Fritz, Friis Larsen, Ken, Grue Simonsen, Jakob, & Stefansen, Christian. 2009.
POETS: Process-Oriented Event-driven Transaction System.
Journal of Logic and Algebraic Programming (JLAP), **78**(5), 381–401.

Hvitved, Tom. 2009.
Contracts in Programming and in Enterprise Systems.
Progress Report, Department of computer science, University of Copenhagen.

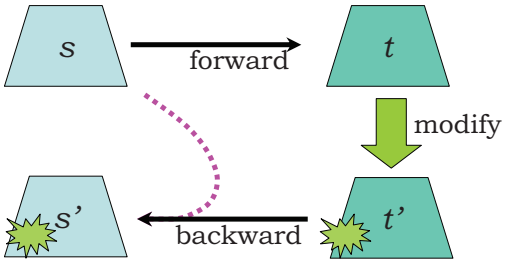
Rikhardsson, Pall, Møller, Charles, & Kræmmergaard, Pernille. 2004.
ERP: Enterprise Resource Planning. Danske erfaringer med implementering og anvendelse. 1 edn.
Børsens Forlag.

Marker-directed optimization of UnCAL graph algebra revisited: Optimizing bidirectional graph transformations

Soichiro Hidaka
National Institute of Informatics, Japan

The Fourth DIKU-IST Joint Workshop on Foundations of Software
12 Jan. 2011

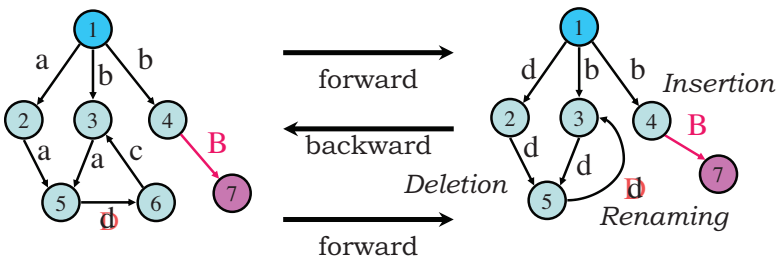
Bidirectional Transformation



2

Graph BT Example

- Replace ‘a’ by ‘d’ and contacts ‘c’



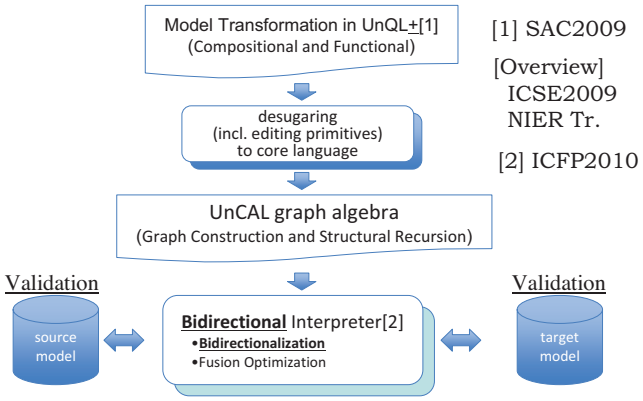
3

Background

- Performance is one of the big issues
  - We are highly motivated for optimization
- Preliminary framework for today's presentation had been mostly implemented

4

A Compositional Framework for Bidirectional Model Transformation



[1] SAC2009

[Overview] ICSE2009 NIER Tr.

[2] ICFP2010

5

UnCAL Structural Recursion in General Form

$$f\ \{\} = \{\}$$

$$f\ \{l:G\} = t(l,G)@\ f\ G$$

$$f\ (G_1 \cup G_2) = f(G_1) \cup (f\ G_2)$$

$$f = \mathbf{rec}(\lambda(\$l,\$g).t(\$l,\$g))$$

6

Example: a2d_xc

- Replace ‘a’ by ‘d’ and contracts ‘c’

```
a2d_xc($db) =
rec(λ($l,$g). if $l=a then {d:&}
              else if $l=c then {ε:&}
              else              {$l:&})($db)
```

7

Core UnCAL Language

$e ::= \{\} \mid \{L : e\} \mid e \cup e \mid \&x := e \mid \&y \mid ()$
$\mid e \oplus e \mid e \mathbin{@} e \mid \texttt{cycle}(e)$ { constructor }
$\mid \$g$ { graph variable }
$\mid \texttt{if } l = l \texttt{ then } e \texttt{ else } e$ { conditional }
$\mid \texttt{rec}(\lambda(\$l,\$g).e)(e)$ {structural recursion application }
$l ::= a$ { edge label }
$\mid \$l$ { label variable }

8

Optimization by forward transformation itself

- Non-necessary recursion can be eliminated
- We go back to bidirectional aspect in the conclusion

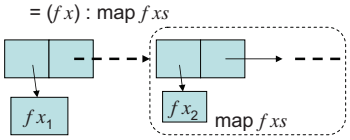
```
f {l:G}    = t(l,G)@ f G
```

9

Structural recursion and markers

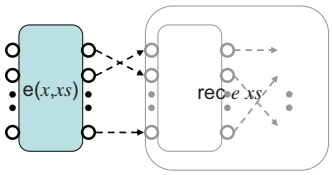
map f $(x{:}xs)$

= $(f\,x)$: map $f\,xs$



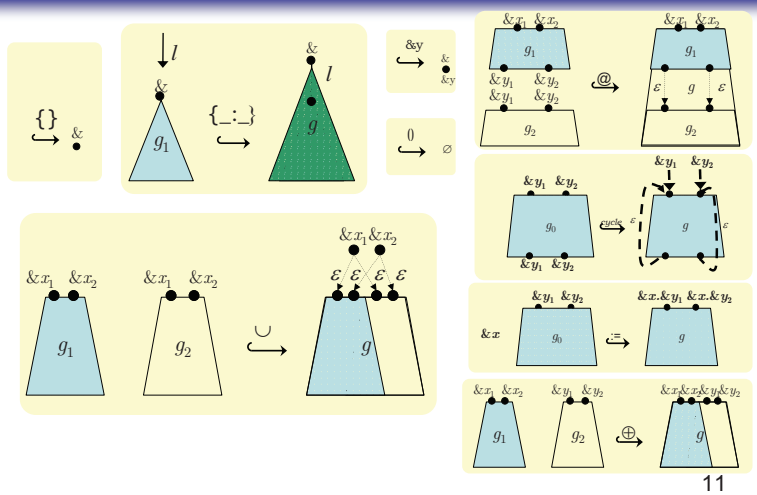
rec e $\{x{:}xs\}$

= $e(x,xs)$@rec $e\,xs$



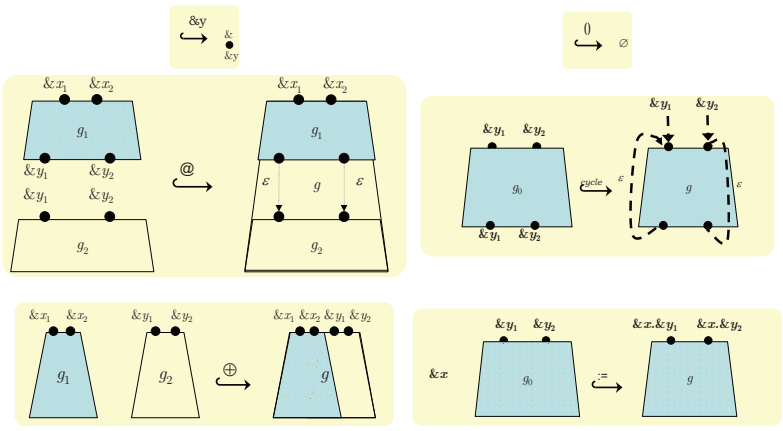
10

Graph Constructors



11

Graph Constructors



12

Revisiting marker analysis

- Mainly run-time analysis is mentioned
  - Avoid evaluating unnecessary subexpressions
- We can now statically compute and further simplify the transformation itself

13

Revisiting original marker analysis

[Buneman+98] TR version

$$\frac{a \in \mathcal{U}}{a : Label} \qquad \frac{y \text{ a label variable}}{y : Label} \qquad \frac{t \text{ a tree variable of type } Tree_{\mathcal{X}}}{t : Tree_{\mathcal{X}}}$$

$$\frac{}{\{\} : Tree_{\mathcal{X}}} \qquad \frac{X \in \mathcal{X}}{X : Tree_{\mathcal{X}}} \qquad \frac{l : Label \quad Q : Tree_{\mathcal{X}}}{\{l \Rightarrow Q\} : Tree_{\mathcal{X}}}$$

$$\frac{l_1 : Label \quad l_2 : Label}{l_1 = l_2 : Bool} \qquad \frac{l_1 : Label \quad \ldots \quad l_n : Label \quad p \text{ a predicate}}{p(l_1, \ldots, l_n) : Bool}$$

$$\frac{b : Bool \quad Q_1 : Tree_{\mathcal{X}} \quad Q_2 : Tree_{\mathcal{X}}}{\text{if } b \text{ then } Q_1 \text{ else } Q_2 : Tree_{\mathcal{X}}}$$

$$\frac{Q_1 : Tree_{\mathcal{Y}} \quad \ldots Q_m : Tree_{\mathcal{Y}}}{(X_1 := Q_1, \ldots, X_m := Q_m) : Tree_{\mathcal{Y}}^{\{X_1, \ldots, X_m\}}}$$

$$\frac{Q_1 : Tree_{\mathcal{X}} \quad Q_2 : Tree_{\mathcal{X}}}{Q_1 \cup Q_2 : Tree_{\mathcal{X}}} \qquad \frac{Q_1 : Tree_{\mathcal{X}} \quad Q_2 : Tree_{\mathcal{Y}}^{\mathcal{X}}}{Q \,@_{\mathcal{X}}\, Q_2 : Tree_{\mathcal{Y}}}$$

$$\frac{y \text{ label variable} \quad t \text{ tree variable of type } Tree_{\mathcal{Y}} \quad Q_1 : Tree_{\mathcal{X}}^{\mathcal{Y}} \quad Q_2 : Tree_{\mathcal{Y}}}{gext_{\mathcal{X}}(\lambda(y,t).Q_1)(Q_2) : Tree_{\mathcal{X}\cdot\mathcal{Y}}^{\mathcal{X}}}$$

14

Computation of markers

[Hidaka+08]

$$\frac{\Gamma \vdash e :: DB_{\mathcal{Y}}^{\mathcal{X}} \quad \mathcal{Y} \subseteq \mathcal{Y}'}{\Gamma \vdash e :: DB_{\mathcal{Y}'}^{\mathcal{X}}} \qquad \frac{}{\Gamma \vdash \{\} :: DB_{\emptyset}} \qquad \frac{\Gamma \vdash e_l :: Label \quad \Gamma \vdash e :: DB_{\mathcal{Y}}}{\Gamma \vdash \{e_l : e\} :: DB_{\mathcal{Y}}} \qquad \frac{\Gamma \vdash e_1 :: DB_{\mathcal{Y}_1}^{\mathcal{X}} \quad \Gamma \vdash e_2 :: DB_{\mathcal{Y}_2}^{\mathcal{X}}}{\Gamma \vdash e_1 \cup e_2 :: DB_{\mathcal{Y}_1 \cup \mathcal{Y}_2}^{\mathcal{X}}}$$

$$\frac{\Gamma \vdash e_1 :: DB_{\mathcal{Y}_1}^{\mathcal{X}_1} \quad \Gamma \vdash e_2 :: DB_{\mathcal{Y}_2}^{\mathcal{X}_2} \quad \mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset}{\Gamma \vdash e_1 \oplus e_2 :: DB_{\mathcal{Y}_1 \cup \mathcal{Y}_2}^{\mathcal{X}_1 \cup \mathcal{X}_2}} \qquad \frac{\Gamma \vdash e_1 :: DB_{\mathcal{Y}}^{\mathcal{X}} \quad \Gamma \vdash e_2 :: DB_{\mathcal{Z}}^{\mathcal{Y}'}}{\Gamma \vdash e_1 \,@\, e_2 :: DB_{\mathcal{Z}}^{\mathcal{X}}} \qquad \frac{\Gamma \vdash e :: DB_{\mathcal{Y}}^{\mathcal{X}} \quad \mathcal{X} \cap \mathcal{Y} = \mathcal{Z}}{\Gamma \vdash cycle(e) :: DB_{\mathcal{Y} \setminus \mathcal{Z}}^{\mathcal{X} \setminus \mathcal{Z}}} \qquad \frac{\Gamma \vdash e :: DB_{\mathcal{Y}}^{\mathcal{X}}}{\Gamma \vdash cycle_{orig}(e) :: DB_{\mathcal{Y} \setminus \mathcal{X}}^{\mathcal{X}}}$$

$$\frac{\Gamma(Var) = DB_{\mathcal{Y}}^{\mathcal{X}}}{\Gamma \vdash Var :: DB_{\mathcal{Y}}^{\mathcal{X}}} \qquad \frac{\begin{array}{c}\Gamma \vdash e_a :: DB_{\mathcal{Y}}^{\mathcal{X}} \\ \Gamma' = \Gamma\{L \mapsto Label\}\{T \mapsto DB_{\mathcal{Y}}\} \\ \Gamma' \vdash e_b :: DB_{\mathcal{Z}_o}^{\mathcal{Z}_i} \\ \mathcal{Z} = \mathcal{Z}_i \cup \mathcal{Z}_o\end{array}}{\Gamma \vdash rec(\lambda(L,T).e_b)(e_a) :: DB_{\mathcal{Y} \cdot \mathcal{Z}}^{\mathcal{X} \cdot \mathcal{Z}}} \qquad \frac{\Gamma \vdash e_b :: Bool \quad \Gamma \vdash e_t :: DB_{\mathcal{Y}_t}^{\mathcal{X}} \quad \Gamma \vdash e_f :: DB_{\mathcal{Y}_f}^{\mathcal{X}}}{\Gamma \vdash \text{if } e_b \text{ then } e_t \text{ else } e_f :: DB_{\mathcal{Y}_t \cup \mathcal{Y}_f}^{\mathcal{X}}}$$

15

Static estimation of markers for structural recursion

- Computation of structural recursion itself requires static computation of markers

16

Fusion rules and output marker analysis

- Removal of intermediate results in successive application of e_1 structural recursion.

$$\begin{cases} rec(e_2) \circ rec(e_1) = rec(rec(e_2) \circ e_1) \\ \quad \textbf{if } e_2(l,t) \textbf{ does not depend on } t \\ rec(e_2) \circ rec(e_1) \\ \quad = rec(\lambda(l,t).rec(e_2)(e_1(l,t) @ rec(e_1)(t))) \\ \quad \textbf{for arbitrary } e_2(l,t) \end{cases}$$

- If you know statically guarantee that e_1 does not produce any output marker, then the second rule fall backs to first rule, opening another optimization opportunities.

17

Non-recursive query

- Join translated to nested recursion
  - Finite step traversal only
  - Body of rec does not have recursion which means the body does not have output marker

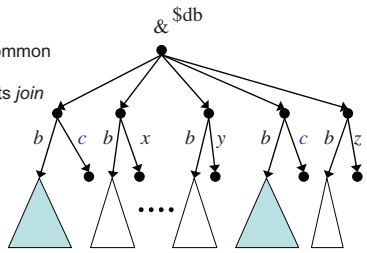
18

Join

- Want to return subgraphs under *b* with sibling *c*

```
sfun f({$l,$tl})=
  sfun fc({c:$tc}) =
    sfun fb({b:$tb}) = $tb
      in fb($tl)
  in fc($tl)
in f($db)
```

uses of common variables implements *join*



introduction of id function

19

Other rules for rec

[Buneman+00]

$$rec(e)(\{\}) = \{\}$$

$$rec(e)(\{l : d\}) = e(l,d) \mathbin{@} rec(e)$$

$$rec(e)(d_1 \cup d_2) = rec(e)(d_1) \cup rec(e)(d_2)$$

$$rec(e)(\&x := d) = \&x \bullet rec(d)$$

$$\frac{\mathcal{Z} = \{\&z_1,\ldots,\&z_p\}\ e \in DB^{\mathcal{Z}}_{\mathcal{Z}}}{rec(e)(\&y) = (\&z_1 := \&y \bullet \&z_1,\ldots,\&z_p := \&y \bullet \&z_p)}$$

$$rec(e)() = ()$$

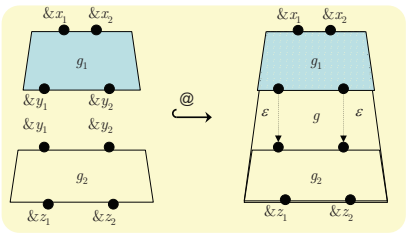
$$rec(e)(d_1 \oplus d_2) = rec(e)(d_1) \oplus rec(e)(d_2)$$

$$\frac{t \text{ does not occur free in } e}{rec(\lambda(l,t).e)(d_1 \mathbin{@} d_2) = rec(e)(d_1) \mathbin{@} rec(e)(d_2)}$$

$$\frac{t \text{ does not occur free in } e}{rec(\lambda(l,t).e)(cycle(d)) = cycle(rec(e)(d))}$$

20

Static marker analysis



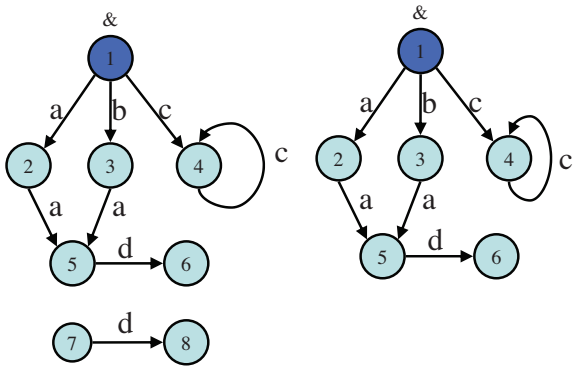
- How about "residual" markers?

$$\frac{d_1 \in DB^{\mathcal{X}}_{\mathcal{Y}}\ \ d_2 \in DB^{\mathcal{Y}}_{\mathcal{Z}}}{d_1 \mathbin{@} d_2 \in DB^{\mathcal{X}}_{\mathcal{Z}}}$$

21

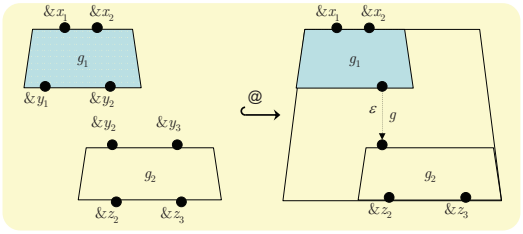
Bisimulation equivalence

- Unreachable parts may be disregarded



22

Residual marker



- Idiom for projection

$$\frac{e_1 :: DB^{\mathcal{X}}_{\varnothing}}{e_1 \,@\, e_2 \downarrow e_1} \qquad \frac{e_1 :: DB^{\mathcal{X}}_{\mathcal{Y}_1} \quad e_2 :: DB^{\mathcal{Y}_2}_{\mathcal{Z}} \quad \mathcal{Y}_1 \cap \mathcal{Y}_2 = \varnothing}{e_1 \,@\, e_2 \downarrow e_1}$$

Correct if the output markers in e_1 are eliminated!

23

Example fusion

```
select
 letrec sfun f2 ({name:T}) = {name:g2(T)}
          |  f2 ({L:T})    = f2(T)
  and   sfun g2 ({L:T})    = {L:g2(T)}
  in
  letrec sfun f1 ({primitiveDataType:T})
                = {primitiveDataType:g1(T)}
           |  f1 ({L:T})    = {L:f1(T)}
  and   sfun  g1 ({name:T})= {typeName:g1(T)}
          |   g1 ({L:T})    = {L:g1(T)}
  in f2(f1(db))
```

24

After desugaring

- composition of rec

```
&z1@rec(\ (L,T).
  if L = "name"
  then (&z1 := {"name": &z2},
        &z2 := {"name": &z2})
  else (&z1 := &z1, &z2 := {L: &z2}))
  (&z1@rec(\ (L,T).
    if L = "name"
    then (&z1 := {"name": &z1},
          &z2 := {"typeName": &z2})
    else if L = "primitiveDataType"
      then (&z1 := {"primitiveDataType": &z2},
            &z2 := {"primitiveDataType": &z2})
      else (&z1 := {L: &z1}, &z2 := {L: &z2}))
      (db))
```

25

After fusion rule applied

```
&z1@(&z2 := &z1&z2, &z1 := &z1&z1)@
 rec(\ (Sa1,T).
  if Sa1="name"
  then (&z1 := (&z1 := {"name": &z2},
                &z2 := {"name": &z2})
     @ (&z2 := &z1&z2, &z1 := &z1&z1),
       &z2 := (&z1 := &z1,
                &z2 := {"typeName": &z2})
     @ (&z2 := &z2&z2, &z1 := &z2&z1))
  else if Sa1 = "primitiveDataType"
    then (&z1 := (&z1 := &z1,
                  &z2 := {"primitiveDataType": &z2})
       @ (&z2 := &z2&z2, &z1 := &z2&z1),
         &z2 := (&z1 := &z1,
                  &z2 := {"primitiveDataType": &z2})
       @ (&z2 := &z2&z2, &z1 := &z2&z1))
    else (&z1 := llet L = Sa1 in
                 if L = "name"
                 then (&z1 := {"name": &z2},
                       &z2 := {"name": &z2})
                 else (&z1 := &z1, &z2 := {L: &z2})
                      @ (&z2 := &z1&z2, &z1 := &z1&z1),
                        &z2 := llet L = Sa1 in
                  if L = "name"
                  then (&z1 := {"name": &z2},
                        &z2 := {"name": &z2})
                  else (&z1 := &z1, &z2 := {L: &z2})
                     @ (&z2 := &z2&z2, &z1 := &z2&z1)))(db)
```

- Fusion rule 1 applied
- Hand optimization required
- We could “plug” the expression in the output marker of the left hand side of @

26

```
&z1@(&z2 := &z1&z2, &z1 := &z1&z1)@
 rec(\ (Sa1,T).
  if Sa1="name"
  then (&z1&z1 := {"name": &z1&z2},
        &z1&z2 := {"name": &z1&z2},
        &z2&z1 := &z2&z1,
        &z2&z2 := {"typeName": &z2&z2})
  else if Sa1 = "primitiveDataType"
    then (&z1&z1 := &z2&z1,
          &z1&z2 := {"primitiveDataType": &z2&z2},
          &z2&z1 :=  &z2&z1,
          &z2&z2 := {"primitiveDataType": &z2&z2}
    else (&z1 := llet L = Sa1 in
            if L = "name"
             then (&z1 := {"name": &z1&z2},
                   &z2 := {"name": &z1&z2})
              else (&z1 := &z1&z1,
                    &z2 := {L: &z1&z2}),
          &z2 := llet L = Sa1 in
            if L = "name"
            then (&z1 := {"name": &z2&z2},
                  &z2 := {"name": &z2&z2})
             else (&z1 := &z2&z1,
                   &z2 := {L: &z2&z2})
         ))(db)
```

27

Plugging expression to output marker expression

$$\frac{}{\{l : \&y\}\,@(\&y := e) \downarrow \{l : e\}}$$

- Further, we could eliminate the output marker expression if it will not be connected to other input nodes

Plugging rule for complex case: cycle

$$\frac{e :: DB^{\mathcal{X}}_{\mathcal{Y}} \quad y \notin (\mathcal{Y} \setminus \mathcal{X})}{cycle(e)\,@(\&x := e_1 \oplus \&y := e_2) \downarrow cycle(e[e_2/\&y])\,@(\&x := e_1)}$$

Preliminary performance results

- Customer to Order composed by selection
  - Without rewriting
    - Fwd evaluation took 0.05 CPU seconds
    - Bwd evaluation took 2.58 CPU seconds
  - Rewrite only to 2nd rule for fusion
    - Fwd evaluation took 0.06 CPU seconds
    - Bwd evaluation took 2.67 CPU seconds
  - After adding rewriting to lead to 1st rule of fusion
    - Fwd evaluation took 0.04 CPU seconds
    - Bwd evaluation took 1.30 CPU seconds

Conclusion

- Static marker computation defined to exploit new optimization opportunity.
- Further optimization based on “plugging” expression reasoned about.
- Future Work
  - Canonical form
    - $\&x_1 := e_1, \&x_2 := e_2, \ldots, \&x_n = e_n$
  - Other reasoning about what we can do statically
  - Analyze impact to reflectable updates

Functional Graph Transformation with Structural Recursion

(Ongoing work)

Hiroyuki Kato

National Institute of Informatics

The Fourth DIKU-IST Joint Workshop on Foundations of Software

January 12th, 2011

1

Graph Transformation

Graphs:

Natural direct representation of real world entities

WWW, GIS, images, videos, social networks,
Biological information, chemical information,
Models in software engineering and databases

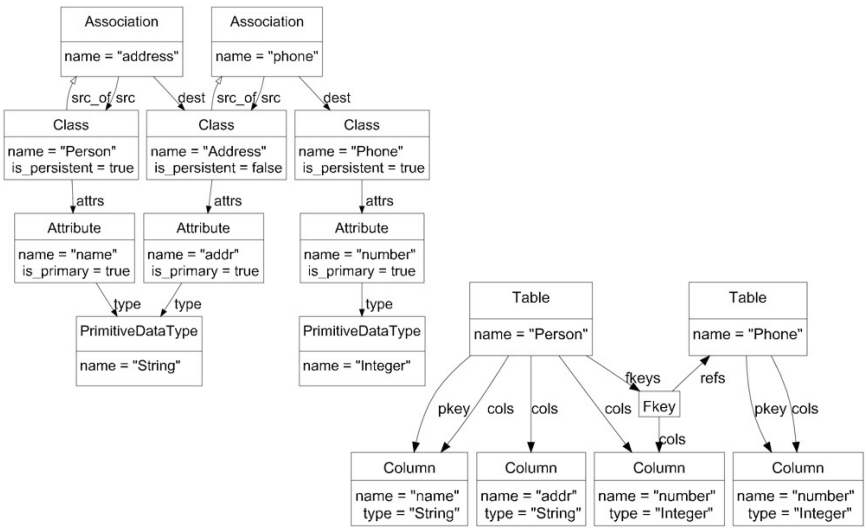
Use of Graphs:

Transformations of one graph into another.

Native graph transformation languages

2

Application: Class2RDB



3

UnQL: Graph Querying

UnQL [VLDB Journal '00] is a

well-defined graph query language

- Easy to use: select ... where ...
- Has a core graph algebra for arbitrary graph construction
- Use structural recursion for manipulating graphs
- Widely used in various applications [SAC'09][ICSE'09 NIER][ICFP'10]...

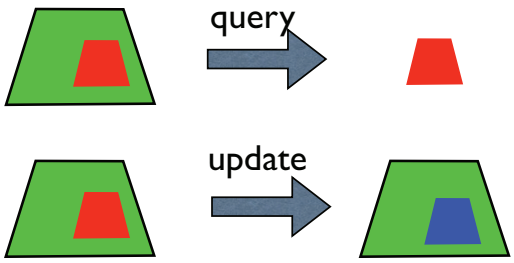
UnQL one or more graphs → one graph
SQL one or more relations → one relation
XQuery zero or more items → zero or more items

4

Goals

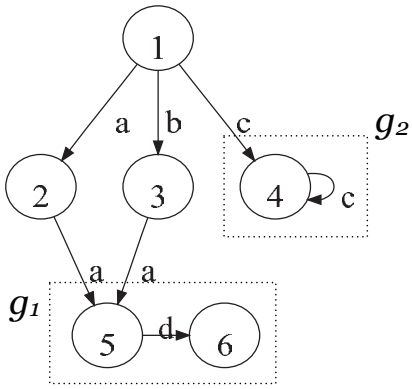
UnQL$^+$: a graph transformation language
an extension of UnQL

✳ A uniform mechanism for both querying and updating.
→ all updates are transformed into **structural recursion**.



5

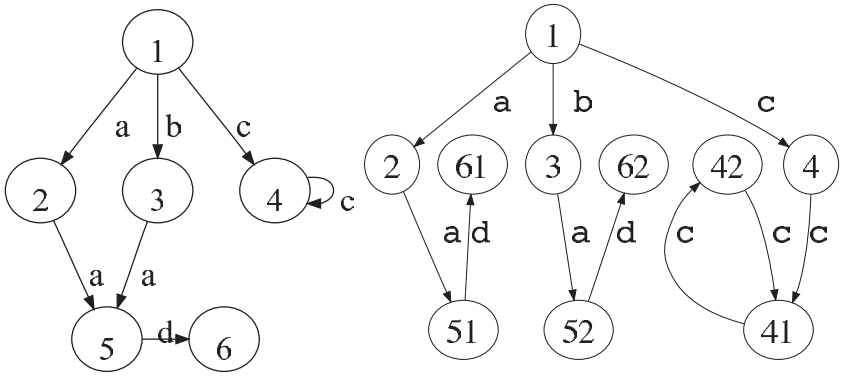
Edge-labelled Graphs



$g = \{a : \{a : g_1\}\} \cup \{b : \{a : g_1\}\} \cup \{c : g_2\}$
$g_1 = \{d : \{\}\}$
$g_2 = \{c : g_2\}$

6

Graph Equivalence based on Bisimulation



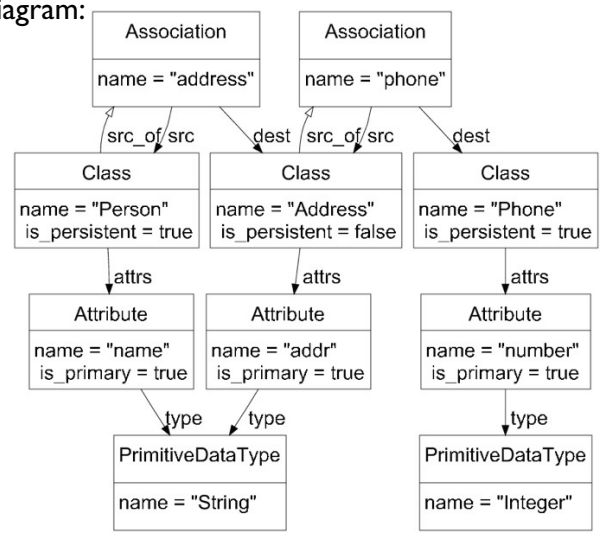
UnQL expressions are bisimulation generic.

f is bisimulation generic if $g_1 \equiv g'_1$, $g_2 \equiv g'_2$, ... implies $f(g_1,g_2,...) \equiv f(g'_1,g'_2,...)$

7

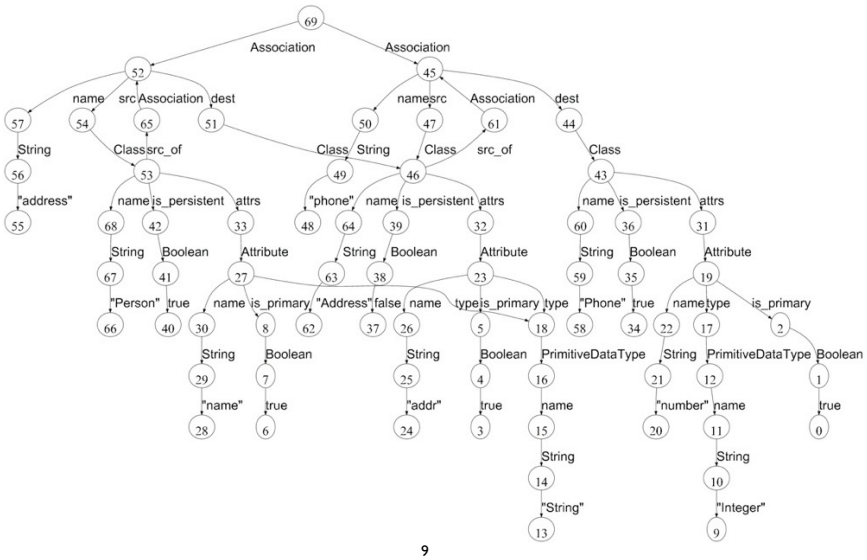
Graph Representation: An Example

A Class Diagram:



8

Graph Representation: An Example



9

Structural Recursion: Manipulating Graphs

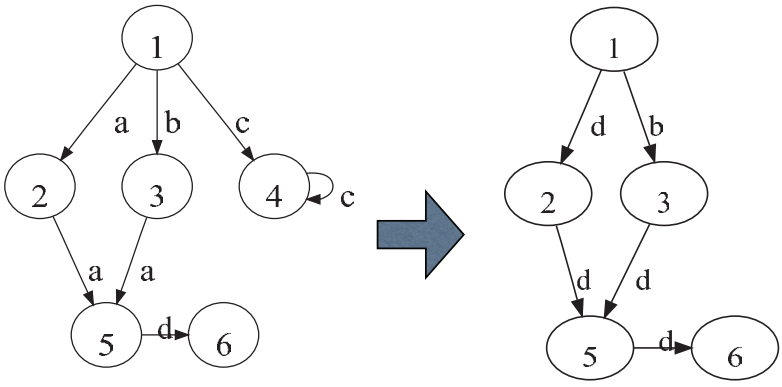
Structural Recursion:

$$f(\{\}) = \{\}$$
$$f(\{l : g\}) = e(l,g)@f(g)$$
$$f(g_1 \cup g_2) = f(g_1) \cup f(g_2)$$

Or written as:

$$\textbf{sfun}\ f(\{l : g\}) = e(l,g)@f(g)$$

Structural Recursion: An Example



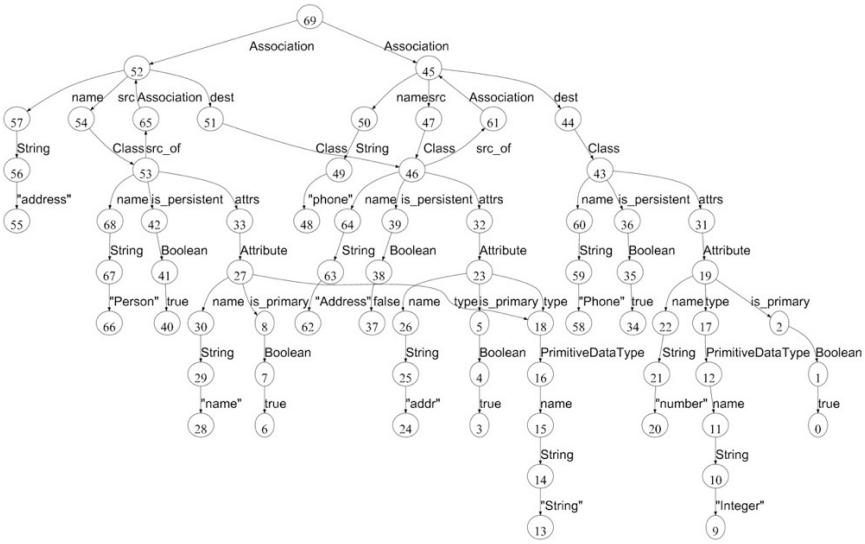
sfun *a2d_xc* ({$*l* : $*g*}) = if $*l* = a then {d : *a2d_xc*($*g*)}
else if $*l* = c then *a2d_xc*($*g*)
else {$*l* : *a2d_xc*($*g*)}

Structural Recursion in UnQL

- Can focus on one-dimensional graphs w.l.o.g.
  each node has at most one outgoing edge

$$f(g_1 \cup g_2) = f(g_1) \cup f(g_2)$$

- No accumulation parameters
  can not formulate one flat mutual recursion
  → nested structural recursion
  cannot refer outer sr function in nested sr

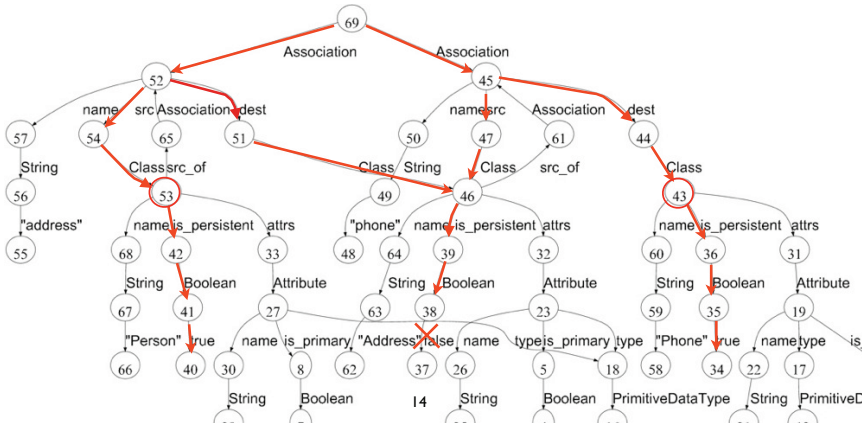


How to extract all persistent classes ?

13

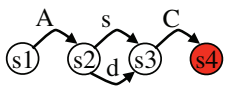
Graph Querying in UnQL

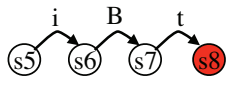
select $*class* where
{*Association*.(*src*|*dest*).*Class*: $*class*} in $*db*,
{*is_persistent*.*Boolean*.*true*:_} in $*class*



14

select $*class* where
{Association.(src|dest).Class: $*class*} in $*db*,
{is_persistent.Boolean.true:_} in $*class*



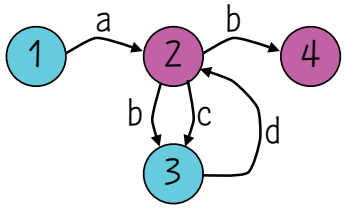



```
let sfun h1({A:$v}) = h2($v)
      | h1({$l:$v}) = {}
    sfun h2({s:$v}) = h3($v)
      | h2({d:$v}) = h3($v)
      | h2({$l:$v}) = {}
    sfun h3({C:$v}) = let sfun h5({i:$u}) = h6($u)
                          | h5({$l:$u}) = {}
                      sfun h6({B:$u}) = h7($u)
                          | h6({$l:$u}) = {}
                      sfun h7({t:$u}) = $v
                          | h7({$l:$u}) = {}
                      in h5($v)
      | h3({$l:$v}) = {}
in h1($db)
```

15

From RPP into structural recursion

`a.((b|c)).d)*.b?`




```
sfun h1({a:$t})  = h2($t) U $t
sfun h2({b:$t})  = h3($t) U h4($t) U $t
  |  h2({c:$t})  = h3($t)
sfun h3({d:$t})  = h2($t) U $t
sfun h4({$l:$t}) = {}
```

16

Language Design Policy

Two choices.

1. Updates are syntactically distinct from queries.
   + simple and clear semantics.
   + easier static analysis (e.g. typecheck).
   - less expressiveness.
   e.g. SQL, Flux

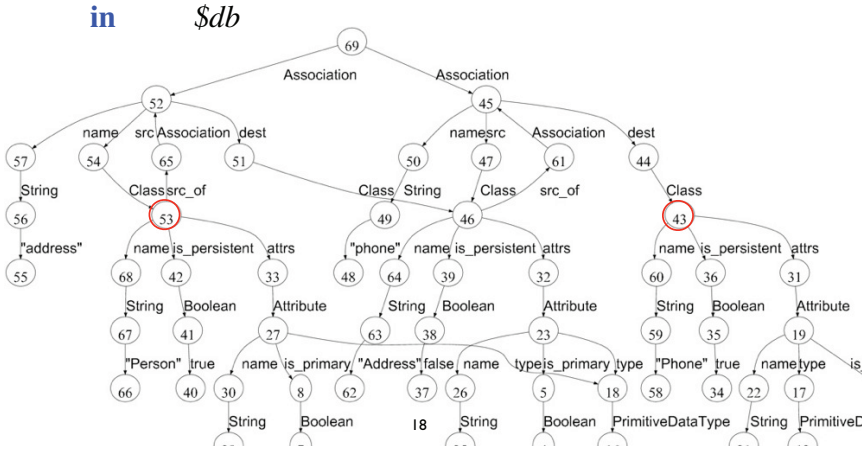
2. Update operations may appear within query exp.
   - complex semantics.
   - difficult static analysis.
   + more expressiveness.
   e.g. XQuery!, XQueryU

17

Graph deleting in UnQL+

How to delete all persistent classes ?

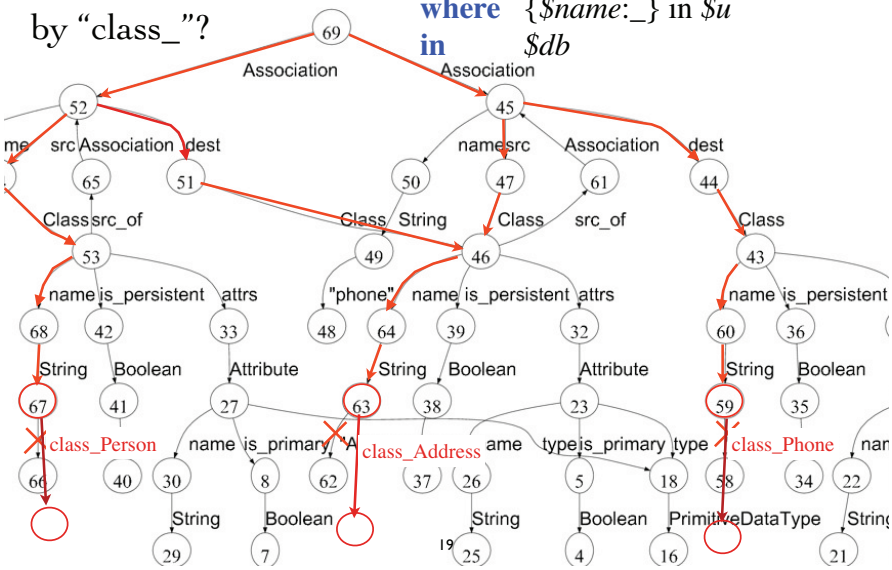
delete Association.(src|dest).Class → *$u*
where {is_persistent.Boolean.true:_} in *$u*
in *$db*



18

How to prefix every name of the class by "class_"?

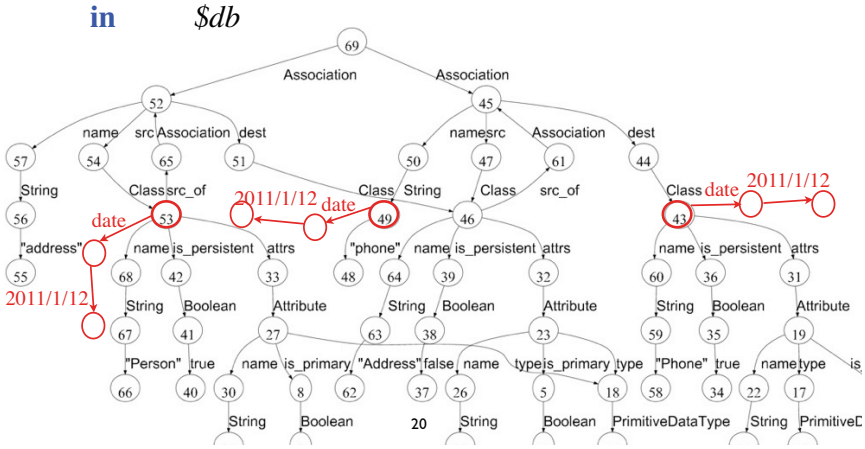
replace _*.Class.name.String → *$u*
by {("class_" + *$name*) : {}}
where {*$name*:_} in *$u*
in *$db*



Graph extending in UnQL+

How to add date information to each class ?

extend _*.Class → *$u*
with {date: {"2011/1/12":{}}}
in *$db*



Syntax of UnQL+ for updating

extend *r* → *var* **with** *e*
where *bs* **in** *e*

replace *r* → *tgt* [**under** *var*] **by** *e*
where *bs* **in** *e*

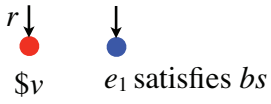
tgt ::= *var* | {*l*:*var*} | {*var*:*var*} | {*r*:*var*}

delete *r* → *tat* [**under** *var*]
where *bs* **in** *e*

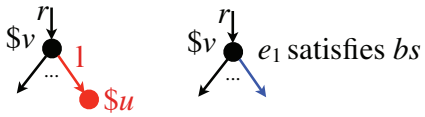
⇒

replace *r* → *tgt* [**under** *var*] **by** {}
where *bs* **in** *e*

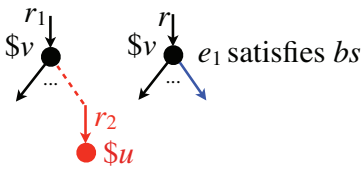
replace r → $v **by** e_1
where bs **in** e_2



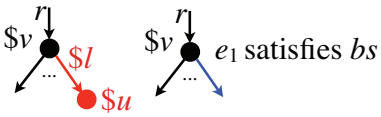
replace r → {l:$u} **[under** $v**]** **by** e_1
where bs **in** e_2



replace r_1 → {r_2:$u} **[under** $v**]** **by** e_1
where bs **in** e_2



replace r → {$l:$u} **[under** $v**]** **by** e_1
where bs **in** e_2

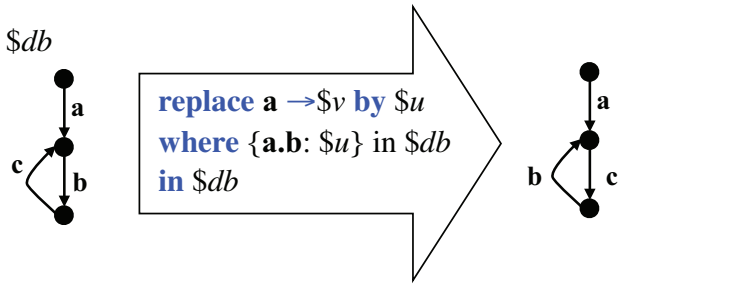


22

Snapshot Semantics

Two logical phases of processing:

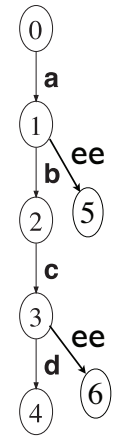
1. specifies nodes to be updated (snapshot)
2. updates are applied to the nodes



SQL, XQuery!, Flux, XQueryU, ...

23

Execution model for **extend**-exp.

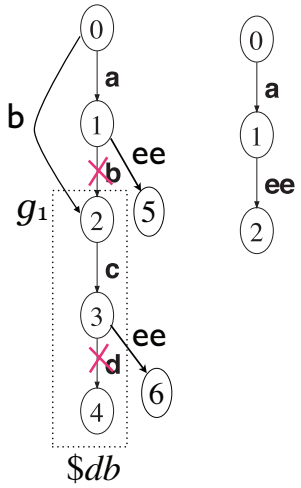


$db

extend _*.(a | c) → $t **with** {ee:{}}
in $db

24

Execution model for **replace**-exp.




```
replace _*.(a | c) → $t by {ee:{}}
in $db
```

The first matching node from the root is replaced.

$$f(\{b{:}g_1\} \cup \{a{:}\{b{:}g_1\}\}) = f(\{b{:}g_1\}) \cup f(\{a{:}\{b{:}g_1\}\})$$

Mapping to Structural Recursion

```
extend r → var with e
where bs in e
```

```
replace r → tgt [under $v] by e
where bs in e
```

- Constructing a DFA for r
- Associating a sr function with each state

How to preserve the context (unchanged part)

sfun h({l:g}) = {l : ...}

Mapping **extend**-exp. into SR

```
extend r → $v with e1
where bs in e2
```

A DFA for r: $(Q, \Sigma, \delta, q_0, F)$

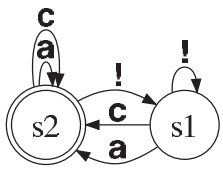
$Q = \{q_0,...,q_N\}$ $\quad \Sigma = \{l_0,...,l_M,!\}$

$\delta : Q \times \Sigma \mapsto Q$ $\quad \delta(q_i,l_k) \in F$

$h_i^{l_j}$: associated with $\delta(q_i, l_j)$

```
let sfun h0({l0:$v}) = ...
    ...
    sfun hi({l0:$v}) = {l0: hi^l0($v)}
      | ...
      | hi({lk:$v}) = {lk: hi^lk($v) ∪ select e1 where bs}
      | ...
      | hi({$l:$v}) = {$l: hi^!($v)}
    sfun ...
    ...
in s1($db)
```

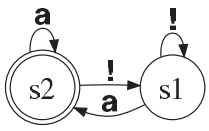
```
extend _*.(a | c) → $t with  {ee:{}}
where {b:$v} in $t
in $db
```




```
let
  sfun s1({a:$t})  = {a: s2($t) ∪ select {ee:{}}
                                  where {b:$v} in $t}
      | s1({c:$t})  = {c: s2($t) ∪ select ee:{}}
                                  where {b:$v} in $t}
      | s1({$l:$t}) = {$l: s1($t)}
  sfun s2({a:$t})  = {a: s2($t) ∪ select {ee:{}}
                                  where {b:$v} in $t}
      | s2({c:$t})  = {c: s2($t) ∪ select {ee:{}}
                                  where {b:$v} in ($t)}
      | s2({$l:$t}) = {$l: s1($t)}
in s1($db)
```

28

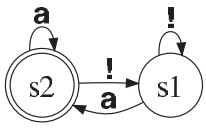
```
replace _*.a -> $u by {ee:{}}
where   {$l1:$g1} in $u,
        {$l2:$g2} in $g1,
        $l1=$l2
in      $db
```




```
let sfun s1({a:$u})  = {a: let sfun h1({$l1:$g1}) =
                               let sfun h2({$l2:$g2}) = if $l1=$l2
                                                        then {ee:{}}
                                                        else s2($u)
                               in h2($g1)
                           in h1($u)}
      | s1({$l:$u}) = {$l: s1($u)}
   sfun s2({a:$u})  = ...
       ....
in s1($db)
```

29

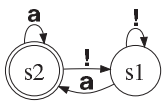
```
replace _*.a -> $u by {ee:{}}
where   {$l1:$g1} in $u,
        {$l2:$g2} in $g1,
        $l1=$l2
in      $db
```

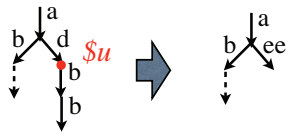



```
 let sfun s1({a:$u})  =
          if isEmpty(select {"found":{}}
                     where {$l1:$g1} in $u, {$l2:$g2} in $g1,  $l1=$l2)
          then {a:s2($u)}
          else {a: select {ee:{}}
                   where {$l1:$g1} in $u, {$l2:$g2} in $g1,  $l1=$l2}
      | s1({$l:$u}) = {$l: s1($u)}
   sfun s2({a:$u})  = ...
       ...
 in s1($db)
```

30

```
replace _*.a -> {d:$u} by {ee:{}}
where   {$l1:$g1} in $u,
        {$l2:$g2} in $g1,
        $l1=$l2
in      $db
```





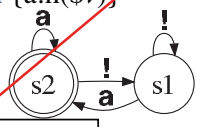

```
sfun s1({a:$v})  = let sfun h({d:$u}) =
                                if isEmpty(select {"found"} where ...)
                                then s2($v)
                                else {a: select {ee:{}} where ...}
                           | h({$l':$u}) = s2($v)
                   in {a:h($v)}
    | s1({$l:$v}) = {$l: s1($v)}
sfun s2({a:$v})   = let sfun h({d:$u}) = ...
    | s2({$l:$v}) = {$l:s1($v)}
```

31

```
sfun s1({a:$v})  = let sfun h({d:$u}) =
                                if isEmpty(select {"found"} where ...)
                                then s2($v)
                                else {a: select {ee:{}} where ...}
                           | h({$l':$u}) = s2($v) in {a:h($v)}
    | s1({$l:$v}) = {$l: s1($v)}
sfun s2({a:$v})   = let sfun h({d:$u}) = ...
    | s2({$l:$v}) = {$l:s1($v)}
```

s2($v)=s2({$l':$u}) s.t. $l' ≠ d
=s2({a:$u})={a:h($u)}
| s2({$l:$u})={$l:s1($u)}

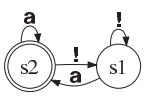
s2($v)=s2({d:$u})={d:s1($u)}

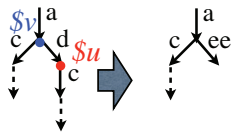



```
sfun s1({a:$v})  = {a:h($v)}
    | s1({$l:$v}) = {$l: s1($v)}
sfun h({d:$u}) = if isEmpty(select {"found"} where ...)
                 then {d:s1($u)}
                 else {a: select {ee:{}} where ...}
sfun h({a:$u}) = {a:h($u)}
sfun h({$l:$u}) = {$l:s1($u)}
sfun s2({a:$v})   = {a:h($v)}
    | s2({$l:$v}) = {$l:s1($v)}
```

32

```
replace _*.a -> {d:$u} under $v by {ee:{}}
where   {$l1:$g1} in $u,
        {$l2:$g2} in $v,
        $l1=$l2
in      $db
```






```
sfun s1({a:$v})  = let sfun h({d:$u}) =
                                if isEmpty(select {"found"} where ...)
                                then s2($v)
                                else {a: select {ee:{}} where ...}
                           | h({$l':$u}) = s2($v)
                   in {a:h($v)}
    | s1({$l:$v}) = {$l: s1($v)}
sfun s2({a:$v})   = let sfun h({d:$u}) = ...
    | s2({$l:$v}) = {$l:s1($v)}
```

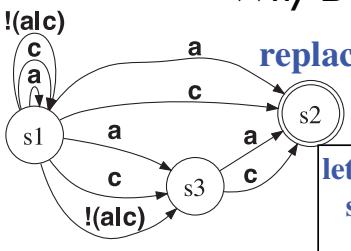
33

Class2RDB in UnQL+

```
select $tables where
  $tables in
    (select $tables where
      {Class:$class} in (select $asc where
                  {Association.(srcldest):$asc} in $db),
      {is_persistent.Boolean:true} in $class,
      $dests in (select {Class:$dest} where
              {(src_of.Association.dest.Class)+:$dest} in $class),
      $related in ({Class:$class} U $dests),
      $cols in (select {cols:{Column:{name:$cname,type:$ctype}}} where
              {Class.attrs.Attribute:{name:$cname,type:$ctype}} in $related),
      $tables in (select {Table:{name:$cname} U $cols} where
                {name:$cname} in $class),
      $tables in (extend Table -> $table with $pkeys U $fkeys in $tables where
                  {cols:$cols} in $table,
                  {Column.name.String:{$cname:{}}} in $cols,
                  $pkeys in (select {pkey:$cols} where
                          {attrs.Attribute:
                            {is_primary.Boolean:true,
                           name.String:{$pname:{}}}} in $class,
                   $cname = $pname),
                $fkeys in (select {fkeys:{Fkey:{cols:$cols, ref:$dcname}}} where
                          {Class:{is_persistent.Boolean:true,
                              attrs.Attribute.name.String:{$aname:{}},
                              name:$dcname}} in $dests,
                         $cname = $aname))),
  $tables in (replace Table.fkeys.Fkey.ref -> $ref by {Table:$table} in $tables where
              {Table:$table} in $tables,
             {String:{$rname:{}}} in $ref,
             {name.String:{$tname:{}}} in $table,
             $tname = $rname)
```


replace Table.fkeys.Fkey.ref -> $ref by {Table:$table} in $tables
where {Table:$table} in $tables,
{String:{$rname:{}}} in $ref,
{name.String:{$tname:{}}} in $table,
$tname = $rname

Why DFA instead of NFA

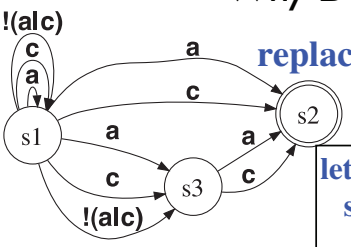


A NFA for _*.(a | c)

replace _*.(a | c)->*$t* **by** {ee:{}} **in** *$db*

```
let
  sfun h1({a:$t})  = {a: {ee:{}}}
       | h1({c:$t})  = {c: {ee:{}}}
       | h1({$l:$t}) = {$l: h1($t) ∪ h3($t)}
and
  sfun h2({$l:$t}) = {$l:$t}
and
 sfun h3({a:$t})   = {a: {ee:{}}}
      | h3({c:$t})   = {c: {ee:{}}}
      | h3({$l:$t})  = {$l:$t}
in h1($db)
```

Why DFA instead of NFA



replace _*.(a | c)->*$t* **by** {ee:{}} **in** *$db*

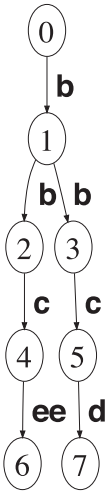
```
let
  sfun h1({a:$t})  = {a: {ee:{}}}
     | h1({c:$t})  = {c: {ee:{}}}
     | h1({$l:$t}) = {$l: h1($t) ∪ h3($t)}
 and
  sfun h2({$l:$t}) = {$l:$t}
 and
  sfun h3({a:$t})  = {a: {ee:{}}}
     | h3({c:$t})  = {c: {ee:{}}}
     | h3({$l:$t}) = {$l:$t}
in h1($db)
```

37

$db={b:{b:{c:{d:{}}}}}

replace _*.(a | c)->*$t* **by** {ee:{}} **in** *$db*


```
let
  sfun h1({a:$t})  = {a: {ee:{}}}
     | h1({c:$t})  = {c: {ee:{}}}
     | h1({$l:$t}) = {$l: h1($t) ∪ h3($t)}
 and
  sfun h2({$l:$t}) = {$l:$t}
 and
  sfun h3({a:$t})  = {a: {ee:{}}}
     | h3({c:$t})  = {c: {ee:{}}}
     | h3({$l:$t}) = {$l:$t}
in h1($db)
```



h1({b:{b:{c:{d:{}}}}})
={b: h1({b:{c:{d:{}}}})
∪ h3({b:{c:{d:{}}}})}
= ...

38

Conclusion

Adding graph updating into UnQL

All update expressions can be translated into structural recursion

- Monolithic operation for both querying and updating optimization friendly (fusion)
- bisimulation generic
- RPP → DFA → structural recursion

39

Related work

- James Cheney: FLUX: functional updates for XML. ICFP 2008:3-14

- Peter Buneman, Mary F. Fernandez, Dan Suciu: UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. VLDB J. 9(1): 76-110 (2000)

- Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, Keisuke Nakano: A compositional approach to bidirectional model transformation. ICSE Companion 2009: 235-238

- Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, Keisuke Nakano: Towards a compositional approach to model transformation for software development. SAC 2009: 468-475

Remaining work

- Mapping replace-exp. into sr without *isempty*.
  isempty introduces transitive closure.
  - use schema information (or, path-index).
  - sr extension with accumulation parameters.
  - use new graph algebra with simple sr.

Thank you

SEMANTIC STRUCTURES OF BIDIRECTIONAL PROGRAMMING

Kazuyuki Asada (The University of Tokyo)
12 January, 2011

Abstract

- This talk is about ...

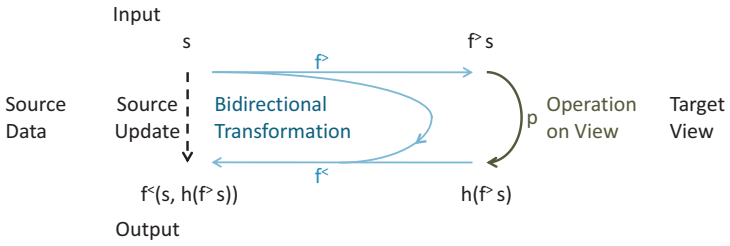
  some properties on some constructors
  (like products, coproducts, …)
  in Bidirectional Programming.

Part I

Introduction:

Bidirectional Programming

Bidirectional Program



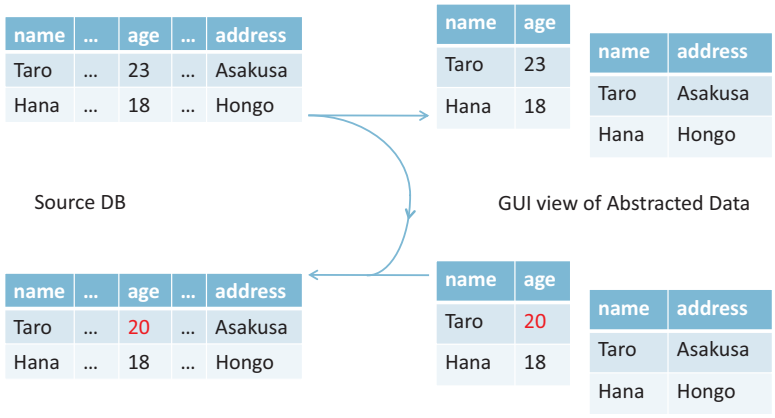
Bidirectional Programming Language

- Bidirectional Programming Language as Domain-Specific Language
  - Target: Relational DB, String, Tree, Graph, …
- Bidirectional Programming in General-Purpose Language
  - in Haskell, Ocaml, …

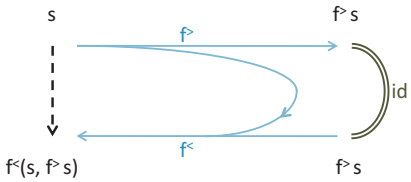
Example of Bidirectional Program

- Bidirectional Sort
  (assuming that veiw-operations keep list length)
- bsort$^{>}$: List int -> List int = foldl insert []
- bsort$^{<}$: List int × List int -> List int
  - Algorithm: an input [2 4 1 5] → its complement:
    ([], []) → ([2], [0]) → ([2 4], [1 0])
    → ([1 2 4], [0 1 0]) → ([1 2 4 5], [3 0 1 0])
  - bsort$^{<}$ [2 4 1 5] l = -- using [3 0 1 0]
    [(l-3-0-1).0 (l-3-0). 1 (l-3).0 l.3]
- bsort$^{<}$ [2 4 1 5] swap(bsort$^{>}$ [2 4 1 5]) = [1 4 2 5]

Example of Bidirectional Program



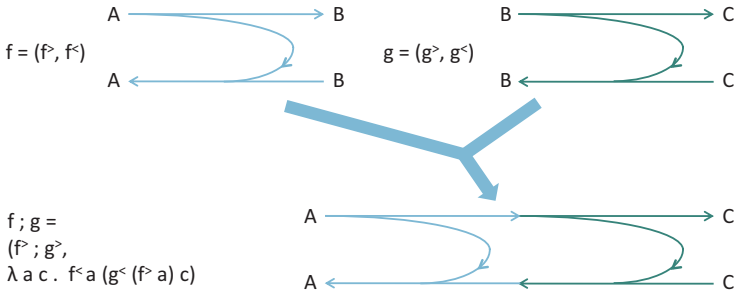
Stability (GetPut law)



$f = (f^{>}, f^{<})$: stable if $f^{<}(s, f^{>} s) = s$

Compositional Programming

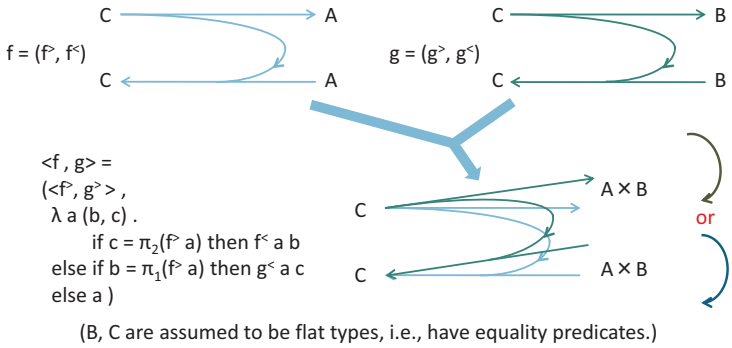
- Basic one: Sequential Composition



If f and g are stable, so is f;g .

Duplication-by-Split

- Dual view, but single update: [M. Takeichi]



(B, C are assumed to be flat types, i.e., have equality predicates.)

If f and g are stable, so is <f, g>.

Part II

(Informal) Semantics of Bidirectional Programming:

Monoidal Product, Trace, Product, and Coproduct

Categories of Bidirectional Programs

- Design choices for bidirectional programming:
  - Arbitrary typing? or "Diagonally" typing?

  - With or without stability law, or other laws such as PutGet.
- Here we consider three categories:

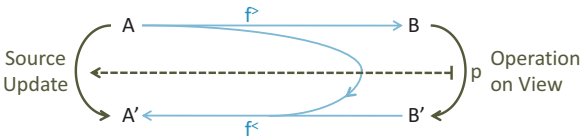
  $\mathrm{Bi}(C)$, $\mathrm{Bi}^{\Delta}(C)$, $\mathrm{Bi}^{\Delta}_{s}(C)$

Bi(C)

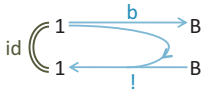
- For a cartesian category C,
  a category $Bi(C)$ consists of
  - objects: pairs of objects in C, and
  - morphisms from (A, A') to (B, B'):
    pairs of morphisms $f^{>} : A \to B$ and $f^{<} : A \times B' \to A'$.
  - Composition is the sequential composition.
  - Identity morphism on A is given by (id_A, π_2).

Reflection function

- For a morphism f: (A, A') -> (B, B'), we call the function $C(B, B') \to C(A, A')$ reflection function.



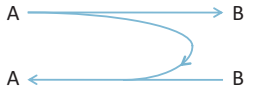
- This does not determine the original morphism $f = (f^{>}, f^{<}) : (A, A') \to (B, B')$, e.g.



  This has no information of "view", i.e., $f^{>}$.

$Bi^{\Delta}(C)$, $Bi^{\Delta}_{s}(C)$

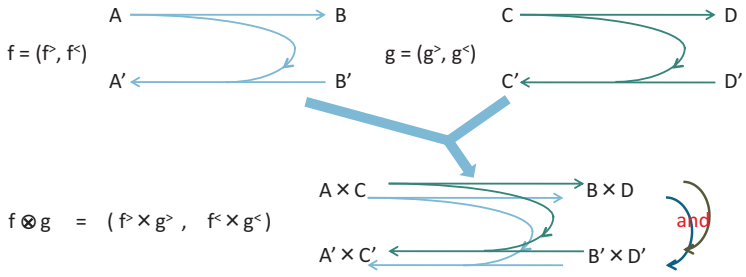
- For a cartesian category C,
  a category $Bi^{\Delta}(C)$ is defined as
  the full subcategory of Bi(C) determined by the objects {(A, A) | A in C}.



- For a morphism in $Bi^{\Delta}(C)$, we call it *stable* if its reflection function maps id_B to id_A .
- Identity in $Bi^{\Delta}_{s}(C)$ is stable, and composition preserves stability, so we have a subcategory $Bi^{\Delta}_{s}(C)$ of stable morphisms.

Monoidal Product

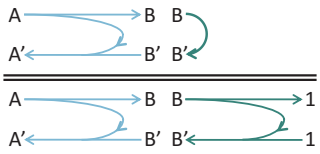
- Pararell Composition:



- If f and g are stable, so is f⊗g.
- This ⊗ is a functor and forms symmetric monoidal category. ($\frac{\text{monoidal category}}{\text{category}} = \frac{\text{monoid}}{\text{set}}$)

Monoidal unit and View operation

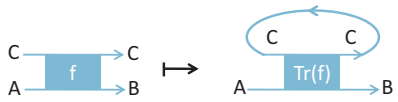
- The unit of the monoidal product is (1, 1).
- Bidirectional morphism from (B, B') to the unit (1,1) in Bi(C) bijectively correspond to ordinary morphism from B to B' in C.
- Then reflection function is just composition.



Trace

- For a monoidal category C, trace operator tr(-) is the following operator (with certain axioms).

$$\frac{f:\ A \otimes C \to B \otimes C}{tr(f):\ A \to B}$$



- Trace operator is for repeated computation.
  - Trace for (cartesian) products is equivalent to fixed point operator [Hyland, Hasegawa 97].
  - Trace for coproducts is called iterator (like while loop).

Cf. GoI

- Geometry of Interaction is originally studied for proof nets in linear logic [Girard 89].
- Categorically, it is understood by Int costruction:

  For a traced monoidal category $\mathcal{C}$,

  Int($\mathcal{C}$) consists of
  - objects: pairs (A, A') of objects in $\mathcal{C}$
  - morphisms from (A, A') to (B, B'):
    morphisms A⊗B' -> B⊗A' in $\mathcal{C}$.

- If $\mathcal{C}$ has trace, so does Int($\mathcal{C}$). (Moreover Compact closed category) [Joyal, Street and Verity 96]

Bi vs. Int

- Bi(-) construction can be considered for symmetric monoidal categories. (But here we skip.)
- For a cartesian category $\mathcal{C}$,

  a morphism f : (A, A') -> (B, B') in Int($\mathcal{C}$) is (essentially) a pair of morphism ($f^{>}$, $f^{<}$) where

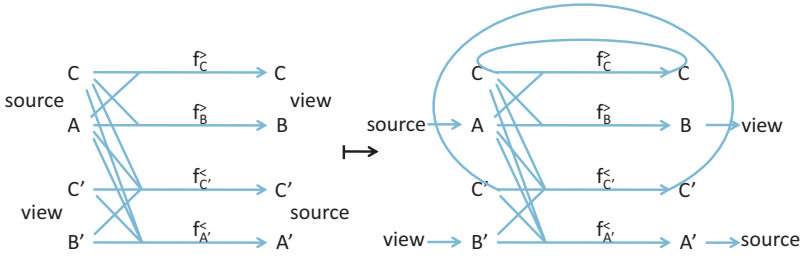
  $f^{>}$: A × B' -> B, $f^{<}$: A × B' -> A' in $\mathcal{C}$.

- Hence bidirectional program is such morphism that $f^{>}$ discard B' imformation, and there is a symmetric monoidal functor J: Bi($\mathcal{C}$) -> Int($\mathcal{C}$) (: not faithful).

Trace operator for Bi(C)

- Prop.

  For a traced symmetric monoidal category $\mathcal{C}$, (especially a cartesian category with fixed point operator,) we have a traced monoidal category Bi($\mathcal{C}$) such that the symmetric monoidal functor J: Bi($\mathcal{C}$) -> Int($\mathcal{C}$) preserves their traces.

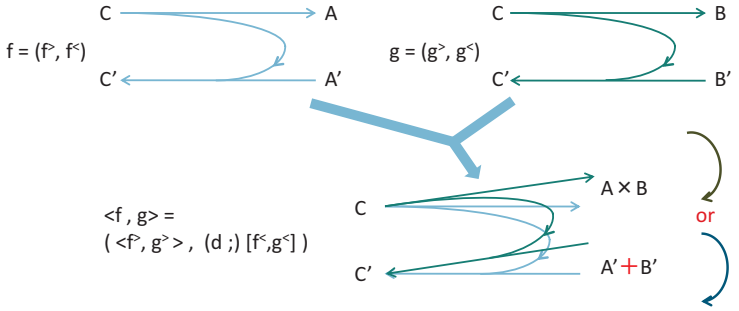


Duplication-by-Split by products

- Prop.

  In Bi(C), there are cartesian products:

  $(A,A') \times (B,B') := (A \times B, A' + B')$

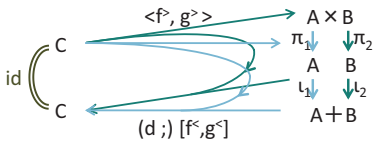


Products vs. Monoidal Products

	Dup.-by-Split by Products	Dup.-by-Split by Monoidal Prod.
Flat type (Eq)	Not required	Required
Property	Genuine cartesian	Only beta equality, w/o eta eq.
Where	in Bi(C)	$Bi_s^{\Delta}(C)$
Stability	Difficult to formalize*	Ok
Safety(?)	Safe(?)	Unsafe(?)

*: On stability of products, at least the following holds:
the reflection function maps $\pi_1; \iota_1$ and $\pi_2; \iota_2$ to id_C .

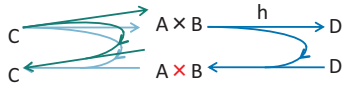
(cf. Here, $\pi_1; \iota_1$ and $\pi_2; \iota_2$ are pairing injective: i.e., $<\pi_1; \iota_1 , \pi_2; \iota_2 >$ is injective.)

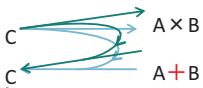


On Safety(?)

- For duplication by monoidal products, we can post-compose other bidirectional program h in $Bi^{\Delta}(C)$.
- Then, it is not rare at all that both B and C are changed via h.
- After that, final change on A is nothing.

  This might make users write unintended programs.

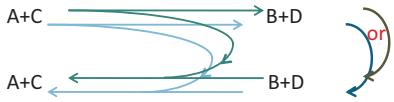




- On the other hand, with cartesian product, we can constrain not to post-compose, or if we program in Bi(C), no problem.

“Coproduct”

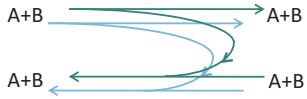
- We have a construction as the following:
  (A, A’)+(B, B’) := (A+B, A’+B’)
- What is this?
- … To make a mapping on morphisms, we define it in $Bi^{\Delta}(\mathcal{C})$:
  (A+C) × (B+D)



  ∼A×B+A×D+C×B+C×D -> A+A+C+C -> A+C
- Then Coproduct? Or, just monoidal product?

Semi-coproduct

- Unfortunately, + is not even monoidal product.
- Since, + is not even functor.
  This + does not preserve identity.



- However, + preserve composition in $Bi^{\Delta}_{s}(\mathcal{C})$.
  This kind of structure is called semi-functor.
- Moreover, + in $Bi^{\Delta}_{s}(\mathcal{C})$ is semi-coproduct.

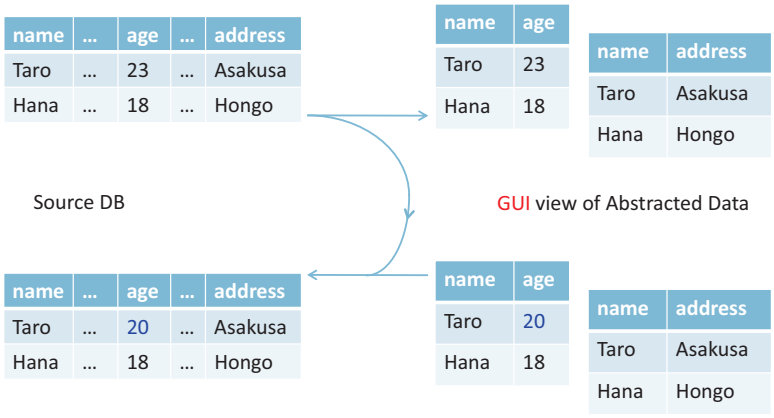
(Cf. [Hayashi 85])

Properties of Semi-coproduct

- For stable bidirectional programs
  $f_i: A_i \rightleftarrows B$ (i=1,2), there is $[f_1, f_2]: A_1+A_2 \rightleftarrows B$.
  For types A_i (i=1,2), there is $\iota_i: A_i \rightleftarrows A_1+A_2$.
- The following equations hold:

$$[f_1, f_2] \,;\, g = [f_1;g, f_2;g]$$
$$\iota_i \,;\, [f_1, f_2] = f_i \qquad (i=1,2)$$
$$[\iota_1, \iota_2] = id+id$$

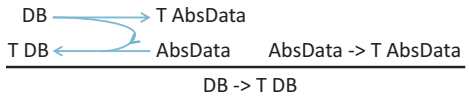
Example of Bidirectional Program



Bidirectional Programming and Monad

- Pure Bidirectional Programming:

- Bidirectional Programming with Monad T (e.g. I/O Monad), or in CBV Language:



Summary

- Optional "world"s for bidirectional programming:

$$\mathrm{Bi}(C) \hookleftarrow \mathrm{Bi}^{\Delta}(C) \hookleftarrow \mathrm{Bi}_{s}^{\Delta}(C)$$

- We have the following constructors on bidirectional programs:
  - Monoidal product in $\mathrm{Bi}(C)$, $\mathrm{Bi}^{\Delta}(C)$, $\mathrm{Bi}_{s}^{\Delta}(C)$
  - Trace operator in $\mathrm{Bi}(C)$, $\mathrm{Bi}^{\Delta}(C)$
  - Cartesian product in $\mathrm{Bi}(C)$
  - Semi-coproduct in $\mathrm{Bi}_{s}^{\Delta}(C)$
- Bidirectional programs easily accommodate with monad, and with arrow.

More

- The same thing might hold about trace w.r.t. coproducts: i.e.,
  "if C has iterator, then $\mathrm{Bi}^{\Delta}_{S}(C)$ has (semi-)iterator".
  However we first define the notion of semi-trace...

cf. Bidir. Progr. forms an Arrow?

- Bi(A,B):= (A -> B) × (A × B -> A)
- >>>, first operator seems exist,
  but is there arr: (A -> B) -> Bi(A, B) ?
- arr f = λa . (f a, λb . a)
  does not work,
  since this does not map identity to identity.
- There seems not to be arrow structure.

Regular expressions as types (Report and co-report)

Fritz Henglein

Department of Computer Science
University of Copenhagen
Email: henglein@diku.dk

DIKU-IST Workshop, Tokyo, 2011-01-12

Joint work with Lasse Nielsen, DIKU
TrustCare Project (trustcare.eu)
With ideas and support from Dexter Kozen and TIPL 2010 course participants

Most used embedded DSLs for programming

- MS Excel macro language
- SQL
- *Regular expressions*

Observe: They are *designed* to be domain-oriented, declarative and of limited expressive power.

Regular language

Definition (Regular language)

A *regular language* is a language (set of strings) over some finite alphabet A that is accepted by some finite automaton.

Regular expression

Definition (Regular expression)

A *regular expression (RE)* over finite alphabet A is an expression of the form

$$E, F ::= 0 \mid 1 \mid a \mid E|F \mid EF \mid E*$$

where $a \in A$

Language interpretation of regular expressions

Definition (Language interpretation)

The *language interpretation* of a regular expression E is the set of strings $\mathcal{L}[\![E]\!]$ defined by

$$\begin{array}{lcllcl} \mathcal{L}[\![0]\!] & = & \emptyset & \mathcal{L}[\![E|F]\!] & = & \mathcal{L}[\![E]\!] \cup \mathcal{L}[\![F]\!] \\ \mathcal{L}[\![1]\!] & = & \{\epsilon\} & \mathcal{L}[\![EF]\!] & = & \mathcal{L}[\![E]\!] \odot \mathcal{L}[\![F]\!] \\ \mathcal{L}[\![a]\!] & = & \{a\} & \mathcal{L}[\![E*]\!] & = & \bigcup_{i \geq 0} (\mathcal{L}[\![E]\!])^i \end{array}$$

where $S \odot T = \{s\,t \mid s \in S \wedge t \in T\}$, $E^0 = \{\epsilon\}, E^{i+1} = E\,E^i$.

Kleene's Theorem

Theorem (Kleene 1956)

A language is regular if and only it is denoted by a regular expression under its language interpretation.

Theory = Language interpretation

What we normally learn and teach about regular expressions in theory of computing classes:

- They're just a way to talk about finite state automata
- All equivalent regular expressions are interchangeable since they accept the same language.
- All equivalent automata are interchangeable since they accept the same language.
  - We might as well choose an efficient one (deterministic, minimal state): it processes its input in linear time and constant space.
- Complexity of containment and equivalence (PSPACE-complete).
- Closure properties, Myhill-Nerode Theorem, Pumping Lemma, Star-height problem ...

Observe: Assumes *language interpretation*!

Practice = Type interpretation

How regular expressions are used in programming[1]:

- Group matching: Does the RE match and where do (some of) its *sub-REs* match in the string?
- Substitution: Replace matched substrings by specified other strings
- Extensions: Backreferences, look-ahead, look-behind,...
- Lazy vs. greedy matching, possessive quantifiers, atomic grouping
- Optimization

Observe: Language interpretation (yes/no) inappropriate, need more refined interpretation

[1] in Perl and such

Example

`((ab)(c|d)|(abc))*`.

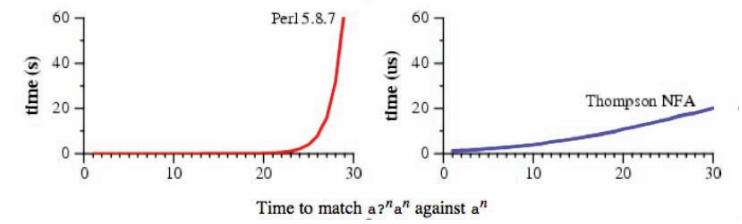
Match against `abdabc`.
For each parenthesized *group* a substring is returned.[a]

		PCRE	*POSIX*
\$1	=	*abc*	*abc*
\$2	=	*ab*	ϵ
\$3	=	*c*	ϵ
\$4	=	ϵ	*abc*

[a] Or special *null*-value

Intermezzo: Optimization??

Optimizing regular expressions = rewriting them to equivalent form that is more efficient for matching.[2]



Cox (2007)

- Perl-compliant regular expressions (what you get in Perl, Python, Ruby, Java) use *backtracking parsing*.
- Does not handle "problematic" regular expressions: E^* where E contains ϵ – will typically crash at run-time (stack overflow).

[2] Friedl, Mastering Regular Expressions, chapter 6: *Crafting an efficient expression*

Why discrepancy between theory and practice?

- Theory is *extensional*: About regular *languages*.
  - Does this string match the regular expression? Yes or no?
- Practice is *intensional*: About regular expressions as *grammars*.
  - Does this string match the regular expression and if so *how*—which parts of the string match which parts of the RE?
- Ideally: Regular expression matching = parsing + "catamorphic" processing of syntax tree
- Reality:
  - Naive backtracking matching, or
  - finite automaton + opportunistic instrumentation to get *some* parsing information (TCL (?), Laurikari 2000, Cox 2010).

Regular expression parsing

- Regular expression parsing: Construct parse tree for given string.
- Representation of parse tree: Regular expression as type

Example

Parse `abdabc` according to `((ab)(c|d)|(abc))*`.

- $p_1 = [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inr}\,(a, (b, c))]$
- $p_2 = [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inl}\,((a, b), \mathrm{inl}\,c)]$

- p_1, p_2 have *type* $\boxed{((a \times b) \times (c + d) + a \times (b \times c))\,\mathrm{list}}$.
- Compare with *regular expression* $\boxed{\texttt{((ab)(c|d)|(abc))*}}$.
- The *elements* of *type* E correspond to the *syntax trees* for strings parsed according to *regular expression* E!

Type interpretation

Definition (Type interpretation)

The *type interpretation* $\mathcal{T}[\![.]\!]$ compositionally maps a regular expression E to the corresponding simple type:

$$\begin{array}{rcll} \mathcal{T}[\![0]\!] & = & \emptyset & \text{empty type} \\ \mathcal{T}[\![1]\!] & = & \{()\} & \text{unit type} \\ \mathcal{T}[\![a]\!] & = & \{a\} & \text{singleton type} \\ \mathcal{T}[\![E \mid F]\!] & = & \mathcal{T}[\![E]\!] + \mathcal{T}[\![F]\!] & \text{sum type} \\ \mathcal{L}[\![E\ F]\!] & = & \mathcal{T}[\![E]\!] \times \mathcal{T}[\![F]\!] & \text{product type} \\ \mathcal{T}[\![E^*]\!] & = & \{[v_1, \ldots, v_n] \mid v_i \in \mathcal{T}[\![E]\!]\} & \text{list type} \end{array}$$

Flattening

Definition

The *flattening* function $\mathrm{flat}(.) : \mathrm{Val}(\mathcal{A}) \to \mathrm{Seq}(\mathcal{A})$ is defined as follows:

$$\begin{array}{rclrcl} \mathrm{flat}(()) & = & \epsilon & \mathrm{flat}(a) & = & a \\ \mathrm{flat}(\mathrm{inl}\, v) & = & \mathrm{flat}(v) & \mathrm{flat}(\mathrm{inr}\, w) & = & \mathrm{flat}(w) \\ \multicolumn{6}{c}{\mathrm{flat}((v, w)) = \mathrm{flat}(v)\,\mathrm{flat}(w)} \\ \multicolumn{6}{c}{\mathrm{flat}([v_1, \ldots, v_n]) = \mathrm{flat}(v_1) \ldots \mathrm{flat}(v_n)} \end{array}$$

Example

$$\begin{array}{rcl} \mathrm{flat}([\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inr}\,(a, (b, c))]) & = & abdabc \\ \mathrm{flat}([\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inl}\,((a, b), \mathrm{inl}\, c)]) & = & abdabc \end{array}$$

Regular expressions as types

Informally:

string s with syntax tree p according to *regular expression* E
$\cong$
string $\mathrm{flat}(v)$ of value v element of *simple type* E

Theorem

$$\mathcal{L}[\![E]\!] = \{\mathrm{flat}(v) \mid v \in \mathcal{T}[\![E]\!]\}$$

Membership testing versus parsing

Example

E = `((ab)(c|d)|(abc))*` E_d = `(ab(c|d))*`

- E_d is *unambiguous*: If $v, w \in \mathcal{T}[\![E_d]\!]$ and $\mathrm{flat}(v) = \mathrm{flat}(w)$ then $v = w$. (Each string in E_d has exactly one syntax tree.)
- E is *ambiguous*. (Recall p_1 and p_2.)
- E and E_d are *equivalent*: $\mathcal{L}[\![E]\!] = \mathcal{L}[\![E_d]\!]$
- E_d "represents" the minimal deterministic finite automaton for E.
- Matching (membership testing): Easy—use E_d.
- But: How to parse *according to* E using E_d?

Regular expression equivalence and containment

Sometimes we are interested in regular expression containment or equivalence.[3]

Definition

- E is *contained* in F if $\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$.
- E is *equivalent* to F if $\mathcal{L}[\![E]\!] = \mathcal{L}[\![F]\!]$.

Regular expression equivalence and containment are easily related: $E \leq F \Leftrightarrow E + F = F$ and $E = F \Leftrightarrow (E \leq F \wedge F \leq E)$.

[3] See e.g. Yasuhiko's talk.

Coercion

Definition (Coercion)

Partial coercion: Function $f : \mathcal{T}[\![E]\!] \to \mathcal{T}[\![F]\!]_\perp$ such that $f(v) = \perp$ or $\mathrm{flat}(v) = \mathrm{flat}(f(v))$.

Coercion: Function $f : \mathcal{T}[\![E]\!] \to \mathcal{T}[\![F]\!]$ such that $\mathrm{flat}(v) = \mathrm{flat}(f(v))$.

Intuition:

- A coercion is a *syntax tree transformer*:
  - It maps a *syntax tree* under regular expression E to a syntax tree under regular expression F for *same* string.
- Coercion = function where you "don't discard, duplicate, shuffle"[4]

[4] Recall Akimasa Morihata's talk

Example

$$f : ((a \times b) \times (c + d) + a \times (b \times c))\,\mathrm{list} \to (a \times (b \times (c + d)))\,\mathrm{list}$$

$$\begin{array}{lcl} f([]) & = & [] \\ f(\mathrm{inl}\,((x, y), z) :: l) & = & (x, (y, z)) :: f(l) \\ f(\mathrm{inr}\,(x, (y, z)) :: l) & = & (x, (y, \mathrm{inl}\,z)) :: f(l) \end{array}$$

- $\mathrm{flat}(f(v)) = \mathrm{flat}(v)$ for all $v : ((a \times b) \times (c + d) + a \times (b \times c))\,\mathrm{list}$.
- So f defines a *coercion* from E = `((ab)(c|d)|(abc))*` to E_d = `(ab(c|d))*`.
- f maps each *proof of membership* (= syntax tree) of a string s in regular language $\mathcal{L}[\![E]\!]$ to a proof of membership of string s in regular language $\mathcal{L}[\![E]\!]$.
- So f is a *constructive proof* that $\mathcal{L}[\![E]\!]$ is *contained* in $\mathcal{L}[\![F]\!]$!

Regular expression containment by coercion

Proposition

$\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$
if and only if
there exists a coercion from $\mathcal{T}[\![E]\!]$ *to* $\mathcal{T}[\![F]\!]$.

Idea:

- Come up with a sound and complete inference system for proving regular expression containments.
- Interpret it as a language for defininig *coercions*:
  - Soundness: Each proof term defines a coercion.
  - Completeness: For each valid regular expression containment there is at least one proof term.

A crash course on regular expression containment

- All classical *sound and complete axiomatizations* basically start with the axioms for *idempotent semirings*.
- Then they add various inference rules to capture the semantics of Kleene star.
- *Algorithms* for *deciding* containment are "coinductive" in nature:
  - transformation to automata or
  - regular expression containment rewriting
- The algorithms have little to do with the axiomatizations!
  - They do not produce a proof (derivation)
  - They cannot be thought of proof search in an axiomatization.

Our approach

Idea:

- Axiomatization =
  Idempotent semiring
  + finitary unrolling for Kleene-star
  + general coinduction rule (for completeness)
  - restriction on coinduction rule (for soundness)
- Each rule can be interpreted as natural *coercion constructor*.
- Algorithms for deciding containment can be thought of as strategies for proof search. They yield coercions, not just decisions (yes/no).

Idempotent semiring axioms

Proviso: $+$ for alternation, $\times$ for concatenation, $*$ for Kleene-star.

$$\begin{aligned}
E + (F + G) &= (E + F) + G \\
E + F &= F + E \\
E + 0 &= E \\
E + E &= E \\
E \times (F \times G) &= (E \times F) \times G \\
1 \times E &= E \\
E \times 1 &= E \\
E \times (F + G) &= (E \times F) + (E \times G) \\
(E + F) \times G &= (E \times G) + (F \times G) \\
0 \times E &= 0 \\
E \times 0 &= 0
\end{aligned}$$

Kleene-star

Finitary unrolling:

$$E^* = 1 + E \times E^*$$

General coinduction rule:

$$\frac{\begin{array}{c}[E = F] \\ \cdots \\ E = F\end{array}}{E = F}$$

- Fantastically powerful rule!
- Unfortunately unsound
- But "right idea" – just needs controlling.

Type-theoretic formulation: Idempotent semiring

With explicit proof terms, using judgement form (due to dispatch in coinduction rule) and containment instead of equivalence:

$$\Gamma \vdash \mathrm{shuffle} : E + (F + G) \leq (E + F) + G$$
$$\Gamma \vdash \mathrm{shuffle}^{-1} : E + (F + G) \leq (E + F) + G$$
$$\Gamma \vdash \mathrm{retag} : E + F \leq F + E$$
$$\Gamma \vdash \mathrm{untag} : E + E \leq E$$
$$\Gamma \vdash \mathrm{tagL} : E \leq E + F$$
$$\ldots$$
$$\Gamma \vdash \mathrm{proj} : E \times 1 \leq E$$
$$\Gamma \vdash \mathrm{proj}^{-1} : E \leq E \times 1$$
$$\Gamma \vdash \mathrm{distL} : E \times (F + G) \leq (E \times F) + (E \times G)$$
$$\Gamma \vdash \mathrm{distL}^{-1} : (E \times F) + (E \times G) \leq E \times (F + G)$$
$$\ldots$$

Primitive coercions

- Each axiom can be interpreted as a *coercion*; e.g.,

$$\begin{aligned} \mathrm{shuffle}(\mathrm{inl}\, x) &= \mathrm{inl}\,(\mathrm{inl}\, x) \\ \mathrm{shuffle}(\mathrm{inr}\,(\mathrm{inl}\, y)) &= \mathrm{inl}\,(\mathrm{inr}\, y) \\ \mathrm{shuffle}(\mathrm{inr}\,(\mathrm{inr}\, z)) &= \mathrm{inr}\, z \end{aligned}$$

- The (p, p^{-1}) pairs denote type isomorphisms: $p \circ p^{-1} = \mathrm{id}$ and $p^{-1} \circ p = \mathrm{id}$.
- $(\mathrm{tagL}\,, \mathrm{untag}\,)$ is an *embedding-projection* pair, but *not* an isomorphism even for $E \equiv F$: $\mathrm{untag} \circ \mathrm{tagL} = \mathrm{id}$, but $\mathrm{tagL} \circ \mathrm{untag} \neq \mathrm{id}$.

Type-theoretic formulation: Kleene-star, coinduction

$$\Gamma \vdash \mathrm{wrap} : 1 + E \times E^* \leq E^*$$
$$\Gamma \vdash \mathrm{wrap}^{-1} : E^* \leq 1 + E \times E^*$$

$$\frac{\Gamma, f : E \leq F \vdash c : E \leq F}{\Gamma \vdash \mathrm{fix} f.c : E \leq F} \quad (Sx)$$

- Interpret $(\mathrm{wrap}\,, \mathrm{wrap}^{-1})$ as isomorphism in accordance with isorecursive interpretation of lists.
- Interpret fix as *least fixed point operator*, that is, as *recursively* defined coercion: $\mathrm{fix} = Y(\lambda f.c)$.
- Add side-condition (Sx) that ensures that recursively defined coercions *terminate*.

The mother of all side conditions

Definition

Coercion c in $\Gamma \vdash c : E \leq F$ is *hereditarily total* if whenever its free variables are bound to (total!) coercions then it denotes a (total!) coercion.

Side condition S_0 (Total): $\boxed{\mathrm{fix} f.c \text{ is hereditarily total}}$

Theorem

It is undecidable whether $\Gamma \vdash c : E \leq F$ is hereditarily total.

Proved by Eijiro Sumii, Yasuhiko Minamide, Naoki Kobayashi, Atsushi Igarashi and Fritz Henglein at the IFIP TC 2 Working Group 2.8 meeting at Shirahama, Japan, April 11-16, 2010.

Other side conditions (informally)

Definition (Size of value)

- 0-size of $v = |\mathrm{flat}(v)|$ (length of underlying string)
- 1-size of v = 0-size of v + number of () occurring in v

Definition

- S_2: Guarantees that recursive calls in coercions are on values of smaller 1-size.
- S_4: Guarantees that recursive calls in coercions are on values of smaller 0-size.

Soundness and completeness

Theorem (Soundness and completeness)

For any of the side conditions S_0, S_2, S_4:

$$\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$$
if and only if
there exists c such that $\vdash c : E \leq F$

Theorem (Parametric soundness and completeness)

For side conditions S_0, S_2 (but not S_4):

For all E', F' we have $\mathcal{L}[\![E[E'/X]]\!] \subseteq \mathcal{L}[\![F[E'/X]]\!]$
if and only if
there exists c such that $\vdash c : [\forall X.]E \leq F$

Computational interpretation of proofs

It is possible to:

- code Salomaa's and Grabmeyer's axiomatizations of regular expression equivalence with side condition S_4.
- code Kozen's axiomatization of Kleene Algebra (= regular expression equivalence) with side condition S_2.

Significance:

- Provides computational interpretation of their proofs: Proofs = coercions (functions that do not discard, duplicate, shuffle)
- Shows that Kozen's axiomatization is parametric complete, but Salomaa's (F_1) and Grabmeyer's are not. (Message: Use 1-size, not 0-size.)
- Our axiomatization provides "more proofs": Better if you are looking for an *efficient* proof.
  - Raises question: How to *find* one?

So what?

Summary so far:

- A regular expression denotes a *type* ("right-regular type").
- A proof of regular expression containment denotes a coercion from one regular expression interpreted as a type to the other.

What is this good for?

Theoretical applications

Provides theoretical framework for *formulating* problems and (eventually) *solving* them:

1. Parametric completeness
2. Coercion synthesis
3. Oracle coding (see below)
4. Fast parsing (see below)
5. Ambiguity resolution
6. Regular expressions as refinement types for strings

Practical applications

First results:

- Bit-coded parse trees
- Bit-coded regular expression parsing

Bit coding

- Record binary choices for expanding a regular expression E into a particular string s.
- The sequence of choices (as bits) is the *bit coding* of s under E.

Example

Recall syntax trees p_1, p_2 for *abdabc* under $E = ((a \times b) \times (c + d) + a \times (b \times c))^*$.

- $p_1 = [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inr}\,(a, (b, c))]$
- $p_2 = [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inl}\,((a, b), \mathrm{inl}\,c)]$

We can *code* them by storing *only* their $\mathrm{inl}\,, \mathrm{inr}$ occurrences:

$$\begin{aligned} \mathrm{code}(p_1) &= 011 \\ \mathrm{code}(p_2) &= 0100 \end{aligned}$$

Bit decoding

There is a linear-time *polytypic* function decode that can reconstitute the syntax trees.

Theorem

$\mathrm{decode}_E(\mathrm{code}_E(v)) = v$ *for all* $v \in \mathcal{T}[\![E]\!]$.

Example

$$\begin{aligned} \mathrm{decode}_E(011) &= [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inr}\,(a, (b, c))] \\ \mathrm{decode}_E(0100) &= [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inl}\,((a, b), \mathrm{inl}\,c)] \end{aligned}$$

Bit coding: Applications

- Bit-coded parse trees are (typically substantially) smaller than serialized parse trees (e.g., source code).
  - Support efficient left-to-right traversal of parse trees.
  - Can be combined with statistical compression to obtain improved compression.
- Coercions can be automatically specialized to operate on bit codes instead of manifest parse trees (not worked out in detail yet); e.g.,

$$\begin{aligned} \mathrm{retag}(0d) &= 1d \\ \mathrm{retag}(1d) &= 0d \\ \mathrm{assoc}(d) &= d \end{aligned}$$

- *Right-regular grammars* yield better (more compressed) bit codes than regular expressions.

Ambiguity resolution

- All regular expression equivalences yield coercion isomorphisms, except for one: $(\mathrm{tagL}, \mathrm{untag}) : E = E + E$.
- This is where ambiguity is introduced/eliminated! Always choosing tagL (from left to right) favors the *left* alternative, as in Perl.
- Eager matching seems to correspond to choosing the *right* alternative in $E^* = 1 + E \times E*$; lazy matching to choosing the *left* alternative.

Open problem

Design an expressive annotation for regular expressions that specifies a choice function for deterministically choosing one of potentially multiple syntax trees for a string and that can (at a minimum) express POSIX and PCRE rules.

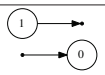
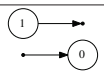
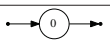
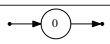
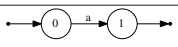
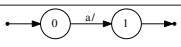
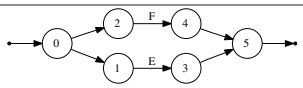
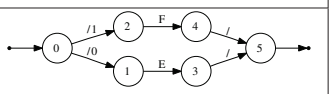
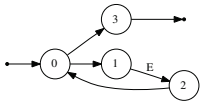
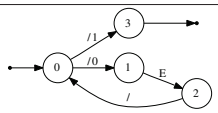
Bit coded regular expression parsing

- Problem:
  - Input: string s and regular expression E.
  - Output: (some) parse tree p such that $\mathrm{flat}(p) = s$.
- Goal: Output *bit coding* $\mathrm{code}_E(p)$ instead.
- Dual advantage:
  - Less space used for output.
  - Output faster to compute.
- How to do that? Mark the "turns" in Thompson NFA (they yield the bit coding)

DFASIM algorithm: Outline

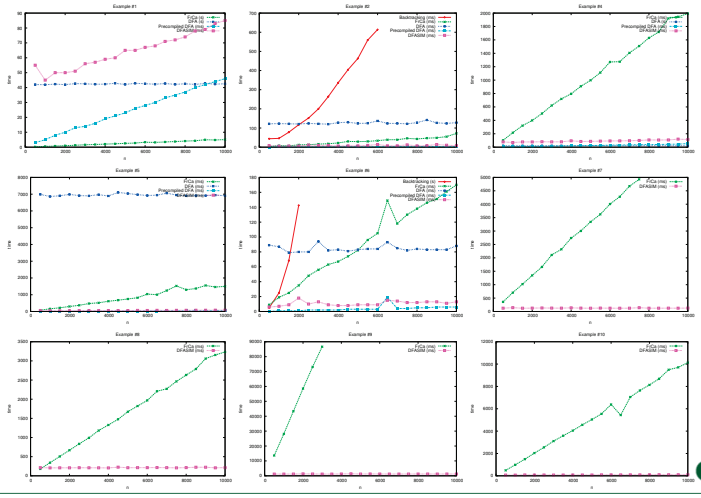
1. RE to NFA: Build Thompson-style NFA *with suitable output bits*
2. NFA to DFA: Perform *extended* DFA construction (only for states required by input string), with (multiple) bit sequence annotations on edges
3. Traverse accepting path from *right to left* to construct bit coding by concatenating bit sequences.

Thompson-style NFA generation with output bits

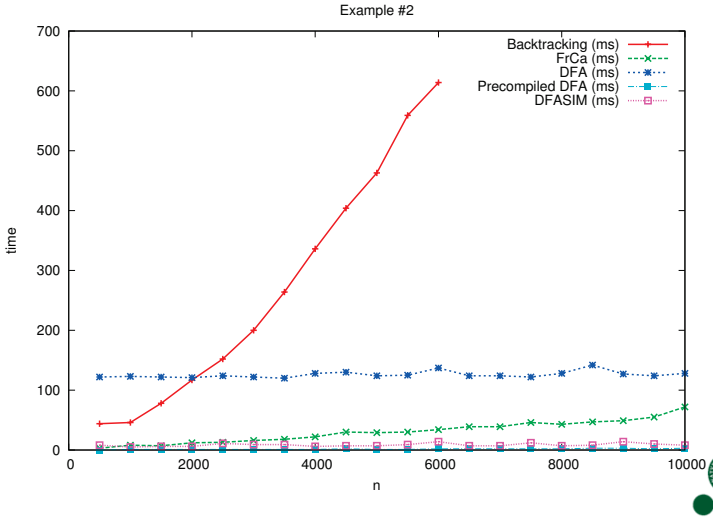
E	NFA	Extended NFA
0		
1		
a		
E F		
E | *F*		
*E**		

Benchmark experiments

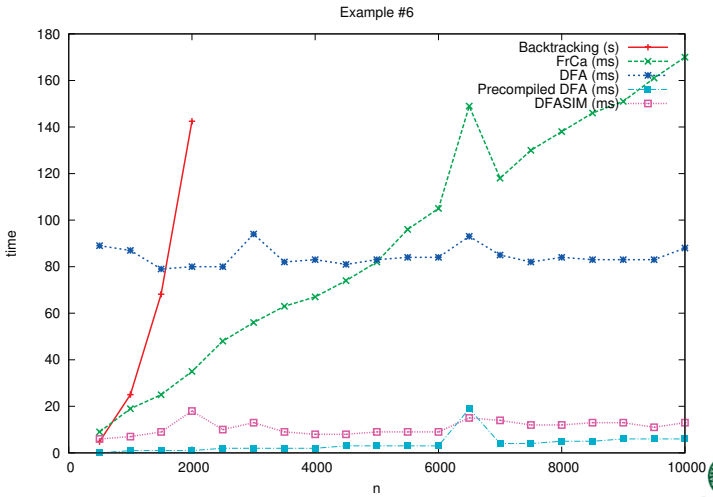
Benchmarks from Veanes et al. (2010)



Benchmark experiment #2



Benchmark experiment #6



References

- Henglein, Nielsen, "Regular Expression Containment: Coinductive Axiomatization and Computational Interpretation", POPL 2011
- Nielsen, Henglein, "Bit-coded Regular Expression Parsing", submitted to LATA 2011

Related work

- Frisch, Cardelli (2004): Regular *types* corresponding to regular expressions, linear-time parsing for REs;
- Hosoya et al. (2000-): Regular expression types, *proper* extension of regular types (!), axiomatization of tree containment
- Aanderaa (1965), Salomaa (1966), Krob (1990), Pratt (1990), Kozen (1994, 2008), Grabmeyer (2005), Rutten et al. (2008): RE axiomatizations (extensional)
- Rutten et al. (1998-): Coalgebraic approach to systems, including finite automata, *extensional*
- Brandt/Henglein (1998): Coinduction rule and computational interpretation for recursive types
- Cameron (1988), Jansson, Jeuring (1999): Bit coding for CFGs and algebraic types
- Cox (2010): RE2 regular expression library, TCL RE library (appear to be state of the Perl/POSIX-style "regex" libraries)

Regular expressions as refinement types for strings

- Add regular expressions as refinement types
- They're already there: Regular types! What needs to be added is coercion synthesis ($\sim$ deciding regular expression containment).
- Use bit coding for run-time representations and bit-coded coercions for bit transformations.

Open problem

Polymorphic regular type and coercion inference.

Related to Hosoya/Frisch/Castagna (2005), which is for regular *expression* types, however.

Future work

- Projection/substitution: efficient composition of parsing, containment (coercions) and catamorphic postprocessing.
- Analysis of ambiguity resolution.
- Build a PCRE- and RE2-killer library.
- Comparison of RE parsing with specialized algorithms (Knuth-Morris-Pratt, Boyer-Moor for single keyword; Aho-Corasick for multiple keywords)
- NFA constructions, relation to Fuh/Mishra S- and G-simplification?
- Proof-theoretic analysis of Krob-Boffa characterization of RE equivalence (50 page proof). Simplified proof?
- Regular expressions as refinement types: Type inference for script languages (Python, Thorn, etc.)

Workflows as Session Types

at DIKU/IST Workshop 2011

Lasse Nielsen

University of Copenhagen

January 13, 2011

Joint work with
Nobuko Yoshida, Imperial College
Kohei Honda, University of London

Project: TrustCare

Workflow models

Aim

Formal specification of cooperative procedures
Implementation (GUI) verification

Motivation

Clinical Practice Guidelines(CPGs):
Detailed descriptions of specific healthcare procedures.

Problem

Verification is either too rigid (one to one), or (too) unsafe

Idea

Use typechecking to verify specification compliance

Outline

What are Workflows

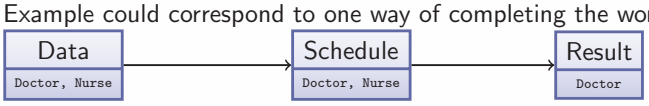
- Fixed sets of actions and participants
- Describes allowed sequences of actions performed by participants
- Used for describing Clinical Practice Guidelines (CPGs)
- Example: Doctor consultation

- Extended workflow as Process Matrix

		Roles			
Id	Name	Patient	Doctor	Nurse	Predecessors
1	Data	W	R	R	
2	Schedule	R	W	W	1
3	Result	R	W	N	2

What are Multiparty Session Types

- Fixed sets of channels and participants
- Describes allowed sequences of messages sent between participants
- Example: $1 \to 3{:}1\langle \text{String} \rangle$; $3 \to 1{:}2\langle \text{Date} \rangle$; $2 \to 1{:}3\langle \text{String} \rangle$; end
- Example could correspond to one way of completing the workflow



- Idea:
  If performing actions corresponds to sending messages on channels
  Then we can represent *some* workflows as session types

Asynchronous π-calculus Multiparty Session Types[2]

Problem: Cannot represent common decission
(Example of Social Interaction)

(Global Types)
$G ::= \mathtt{p} \to \mathtt{p}' : k\langle U \rangle.G'$
$\mid \mathtt{p} \to \mathtt{p}' : k\{l_i : G_i\}_{i \in I}$
$\mid \mu t.G$
$\mid t$
$\mid$ end
$\mid \{l : G_l\}_{l \in L; L'} \quad (L' \neq \emptyset)$[1]

(Message Types)
$U ::= \tilde{S}$
$\mid T@(\mathtt{p}, m, n)$

(Simple Types)
$S ::=$ bool
$\mid$ int
$\mid \ldots$
$\mid \langle G \rangle$

(Local Types)
$T ::= k!\langle U \rangle; T$
$\mid k?\langle U \rangle; T$
$\mid k \oplus \{l : T_l\}_{l \in L}$
$\mid k \,\&\, \{l : T_l\}_{l \in L}$
$\mid \mu t.T$
$\mid t$
$\mid$ end
$\mid \{l : T_l\}_{l \in L; L'}$

[1] EXPRESS'10: Nielsen, Yoshida and Honda
[2] POPL'08: Honda, Carbone and Yoshida

Process extensions

- Will not describe Asynchronous π-calculus with Multiparty sessions [POPL'08]
- New process construct: $\mathtt{sync}_{\tilde{s},n}\{l : P_l\}_{l \in L}$
  Represents participation in a common decission with n participants
- Synchronization step:

$$\frac{h \in \bigcap_{i=1}^{n} L_i}{\mathtt{sync}_{\tilde{s},n}\{l : P_{1l}\}_{l \in L_1} \mid \ldots \mid \mathtt{sync}_{\tilde{s},n}\{l : P_{nl}\}_{l \in L_n} \to P_{1h} \mid \ldots \mid P_{nh}}$$

- New typing rule:

$$\frac{\forall l \in L' : \Gamma \vdash P_l \triangleright \Delta, \tilde{s} : T_l @(\mathrm{p}, n) \quad L' \subseteq L \cup M \quad M \subseteq L'}{\Gamma \vdash \mathtt{sync}_{\tilde{s},n}\{l : P_l\}_{l \in L'} \triangleright \Delta, \tilde{s} : \{l : T_l\}_{l \in L;M} @(\mathrm{p}, n)}$$

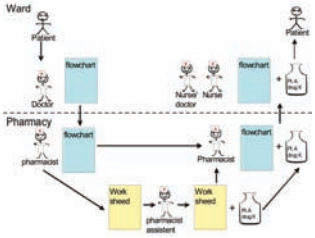
Results

- Theorem: Subject Reduction
  If $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ and $P \to P'$
  then $\Gamma \vdash P' \triangleright_{\tilde{s}} \Delta'$ where $\Delta \to^{0/1} \Delta'$.
- Communication safety
- Single Session Progress
- As expressive as the Process Matrix

Aπms

- Asynchronous π-calculus with Multiparty sessions and Symmetric sum
- First implementation of Asynchronous π-calculus
  with multiparty sessions and multiparty session types
  (It is however cheating!)
- Proof of concept implementation (slow and uses much memmory)
- Includes synchronisation and symmetric sum type extensions
- Aπms extends the theory with user interaction (GUIs)
- There is a translation from the Process Matrix workflow model to Aπms, which allows execution of workflows.
- Aπms and code examples are available from
  `www.thelas.dk/index.php/Apims`.

Real World Example[3]



Vunerabilities:
Bad faith implementations (Adversarial Model)
Typechecking does not ensure correct value is sent
Unimplemented Services (Gets stuck)
No progress for muli-session processes.

[3] From ProHealth'08

Future Work

- Improve Implementation (Compiler, Exploit parallelism)
- Encoding of Process Matrix has exponential size
- Add assertions by merging theory with [BHTY2010]
- Encode more workflow models (UML Activity Diagrams, GLIF, ...)
- Extend theory with temporal properties such as deadlines
- Encode in Adversarial Model

End of talk

- POPL08: Multiparty Asynchronous Session Types
  by Kohei Honda, Nobuko Yoshida and Marco Carbone (POPL 2008)
- BHTY2010: A theory of design-by-contract for distributed multiparty interactions
  by Laura Bocchi, Kohei Honda, Emilio Tuosto and Nobuko Yoshida (Concur 2010)
- SGI'09: A Game-Theoretic Model for Distributed Programming by Contract
  by Anders Starcke Henriksen, Tom Hvitved and Andrzej Filinski
- ProHealth'08: From Paper Based Clinical Practice Guidelines to Declarative Workflow Management
  by Karen Marie Lyng, Thomas Hildebrandt and Rakhava Rao Mukkamala
  Available from: http://www.trustcare.eu
- Aπms: `www.thelas.dk/index.php/Apims`
- Full version: `www.thelas.dk/index.php/Symmetric_Sum_Types`

Questions?

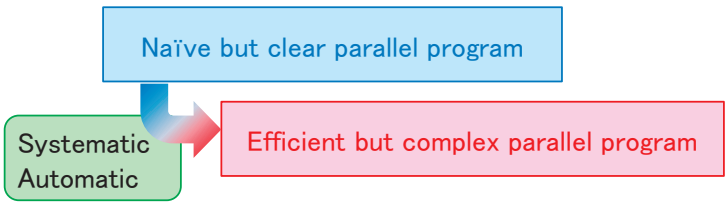
Semirings for Free!

An Algebraic Approach to Efficient Parallel Algorithms for Nested Reductions

Kento EMOTO (University of Tokyo)

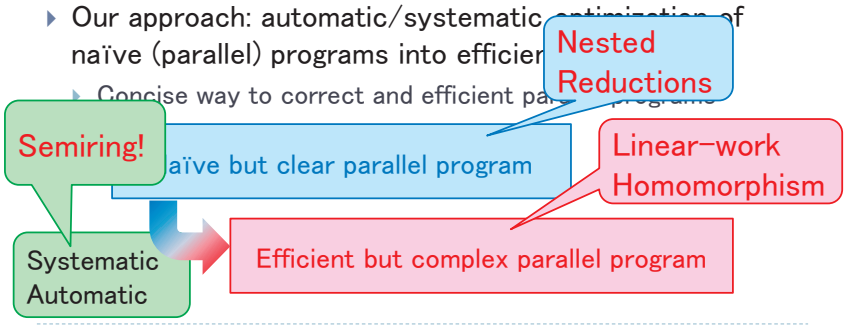
Background

- Parallelism is the only way to get fast computation
  - HW benders improve HW performance mainly by parallelism
- Parallel programming is however difficult for most users
- Our approach: automatic/systematic optimization of naïve (parallel) programs into efficient ones
  - Concise way to correct and efficient parallel programs



This Work

- Parallelism is the only way to get fast computation
  - HW benders improve HW performance mainly by parallelism
- Parallel programming is however difficult for most users
- Our approach: automatic/systematic optimization of naïve (parallel) programs into efficient ones
  - Concise way to correct and efficient parallel programs



Semiring $(\alpha, \oplus, \otimes)$

- Examples: $(\mathbb{R}, +, \times)$, $(\mathbb{Z}, \uparrow, +)$, $(\mathbb{N}, \uparrow, \times)$
- Definition
  $(\alpha, \oplus)$ is commutative monoid
  $(\alpha, \otimes)$ is monoid
  $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ (distributive)
  $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$
  $a \otimes \imath_\oplus = \imath_\oplus \otimes a = \imath_\oplus$ (zero)
  - Monoid $(\alpha, \oplus)$
    $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ (associative)
    $\imath_\oplus \oplus a = a \oplus \imath_\oplus = a$ (the identity)
  - Commutative monoid $(\alpha, \oplus)$
    $(\alpha, \oplus)$ is monoid
    $a \oplus b = b \oplus a$ (commutative)

Simple Problem: The Maximum Subsequence Sum

- Given a list x,
  find the maximum sum of all subsequences

$$\uparrow[\sum[a \mid a \leftarrow s] \mid s \leftarrow subs\ x]$$

$O(2^{|x|})$

- Example: x = [3, −4, 2]
  - subs x = [[3, −4, 2], [3, −4], [3, 2], [3], [−4, 2], [−4], [2], [ ]]
  - The answer is 5 computed as follows.
    5 = (3 + −4 + 2) ↑ (3 + −4) ↑ (3 + 2) ↑ 3 ↑ (−4 + 2) ↑ −4 ↑ 2 ↑ 0

Efficient Computation via Semiring

- Properties of the semiring give an efficient computation with a linear cost

$$\uparrow[\sum[a \mid a \leftarrow s] \mid s \leftarrow subs\ x] = \sum[a \uparrow 0 \mid a \leftarrow x]$$

$O(2^{|x|})$

List homomorphism with linear work
Parallel implementation on various frameworks
such as SkeTo, MapReduce, OpenMP, ⋯

The semiring gives the efficient parallel algorithm for the problem.

= (3 ↑ 0) + (−4 ↑ 0) + (2 ↑ 0)
= 3 + 0 + 2 = 5

Question:
Do Similar Problems Have Efficient Algorithms?

- Nested reduction:

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow gog\ x, p\ y]$$

- Examples
  - The even-sum maximum subsequence sum

$$\uparrow[\sum[a \mid a \leftarrow s] \mid s \leftarrow subs\ x, evensum\ s]$$

  - The even-sum maximum initial-segment sum

$$\uparrow[\sum[a \mid a \leftarrow s] \mid s \leftarrow inits\ x, evensum\ s]$$

  - k-maximum maximum initial-segment sum

$$\uparrow_k[[\sum[a \mid a \leftarrow s]] \mid s \leftarrow inits\ x, evensum\ s]$$

  - The number of subsequences matching a regular expression

$$\sum[1 \mid s \leftarrow subs\ x, p_{RE}\ s]$$

The Answer: **Yes!**
The Current Results of This Work

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow gog\ x, p\ y]$$

Semiring construction

$$\hat{\pi}(\hat{\bigoplus}[\hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y] \mid y \leftarrow subs'\ x])$$

The simple optimization

$$\hat{\pi}(\hat{\bigotimes}[\hat{f}'_{\hat{\oplus}}\ a \mid a \leftarrow x])$$

Linear cost list homomorphism can be run on SkeTo, MapReduce, ITBB, OpenMP, MPI, ⋯

- Optimizable nested reductions
  - gog = a finite range homomorphic (FRH) GoG
  - p = FRH predicates and their and/or/not
  - $(\alpha, \oplus, \otimes)$ is a semiring

Running Example:
Even-sum Maximum Subsequence Sum

- Let's build an efficient algorithm (homomorphism) for

$$\uparrow[\sum[f\ a \mid a \leftarrow s] \mid s \leftarrow subs\ x, evensum\ s]$$

- Remember the efficient algorithm for the simplest case
  - We can add partial results freely to get the total result

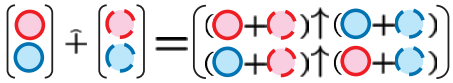
$$\begin{aligned}&\uparrow[\sum[f\ a \mid a \leftarrow s] \mid s \leftarrow subs\ (y +\!\!+ z)]\\ &= \sum[f\ a \uparrow 0 \mid a \leftarrow (y +\!\!+ z)]\\ &= \sum[f\ a \uparrow 0 \mid a \leftarrow y] + \sum[f\ a \uparrow 0 \mid a \leftarrow z]\end{aligned}$$

- Is EMSS of (y ++ z) a just sum of EMSSs of y and z?
  - **No.**

Possible States of Partial Subsequences

- Each sequence has two possible states: even and odd
  - States changes after summations

- Candidates of EMSS of (y ++ z) are
  - A sum of EMSSs of y and z, and
  - A sum of the odd-sum MSSs of y and z
- We need to use the following new merging operator

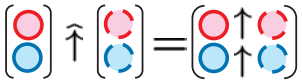


The Base Case Computation

- How is the computation of the base case x = [a]?
  - Remember that we need both EMSS and OMSS

$$
\begin{aligned}
f'\ a &= \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ \begin{pmatrix} f\ a \uparrow 0 \\ 0 \end{pmatrix}\ \textbf{else}\ \begin{pmatrix} 0 \\ f\ a \uparrow 0 \end{pmatrix} \\
&= \left(\textbf{if}\ \mathit{even}\ a\ \textbf{then}\ \begin{pmatrix} f\ a \\ 0 \end{pmatrix}\ \textbf{else}\ \begin{pmatrix} 0 \\ f\ a \end{pmatrix}\right) \hat{\uparrow} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
&= \hat{f}\ a \hat{\uparrow} \hat{0}
\end{aligned}
$$

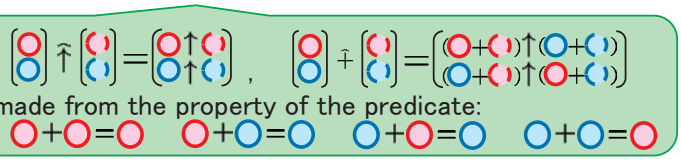
- We got the following new operator



Efficient Algorithm for Even-sum Maximum Subsequence Sum

- We have built the following algorithm for EMSS

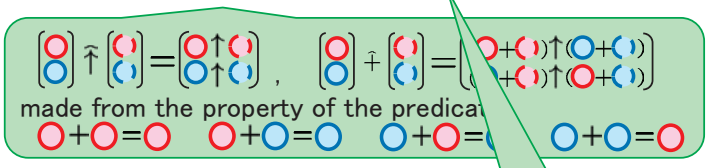
$$
\begin{aligned}
&\uparrow[\textstyle\sum[f\ a \mid a \leftarrow s] \mid s \leftarrow \mathit{subs}\ x, \mathit{evensum}\ s] \\
&= \pi(\hat{\textstyle\sum}[\hat{f}\ a \hat{\uparrow} \hat{0} \mid a \leftarrow x])
\end{aligned}
$$



Efficient Algorithm for Even-sum Maximum Subsequence Sum

- We have built the following algorithm for EMSS

$$\uparrow[\textstyle\sum[f\ a \mid a \leftarrow s] \mid s \leftarrow subs\ x, evensum\ s] = \pi(\hat{\sum}[\hat{f}\ a \mathbin{\hat{\uparrow}} \hat{0} \mid a \leftarrow x])$$



- C.f., the computation of MSS

$$\uparrow[\textstyle\sum[f\ a \mid a \leftarrow s] \mid s \leftarrow subs\ x] = \sum[f\ a \uparrow 0 \mid a \leftarrow x]$$

Completely the same!

Efficient Algorithm for Even-sum Maximum Subsequence Sum

- We have built the following algorithm for EMSS

$$\uparrow[\textstyle\sum[f\ a \mid a \leftarrow s] \mid s \leftarrow subs\ x, evensum\ s] = \pi(\hat{\sum}[\hat{f}\ a \mathbin{\hat{\uparrow}} \hat{0} \mid a \leftarrow x]) = \pi(\hat{\uparrow}[\hat{\sum}[\hat{f}\ a \mid a \leftarrow s] \mid s \leftarrow subs\ x])$$

Actually, $(\alpha^{\{even,odd\}}, \hat{\uparrow}, \hat{+})$ is a semiring!
The filter has been embedded into the semiring!

$$\pi(\hat{\sum}[\hat{f}\ a \mid a \leftarrow s]) = \textbf{if}\ evensum\ s\ \textbf{then}\ \textstyle\sum[f\ a \mid a \leftarrow s]\ \textbf{else}\ 0$$

Embedding Filters into Semirings

Theorem Given a semiring $(\alpha, \oplus, \otimes)$, a FRH predicate $p = accept_p \circ ([\oplus_p, f_p])$ (where $accept_p :: C_p \to \text{Bool}$), a list g, and a function f, the following equation holds.

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow g, p\ y] = \hat{\pi}(\hat{\bigoplus}[\hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y] \mid y \leftarrow g])$$

Here, $(\alpha^{C_p}, \hat{\oplus}, \hat{\otimes})$ is the lifted semiring of $(\alpha, \oplus, \otimes)$ with p, f is the lifted function of f with p, and $\hat{\pi}$ is the unlifer with p.

- Here, $([\oplus, f])\ x = \bigoplus[f\ a \mid a \leftarrow x]$ is a list homomorphism:

$$([\oplus, f])\ (x +\!\!+ y) = ([\oplus, f])\ x \oplus ([\oplus, f])\ y$$
$$([\oplus, f])\ [a] = f\ a$$

- FRH = finite range homomorphic
- E.g., $evensum = id \circ ([\text{XNOR}, even])$

Intuition of Lifted Semirings

- A FRH predicates = a monoid ∼ a group
- Analogy of group rings: group + ring = ring

$$a \mathbin{\hat{\oplus}} b = c \quad \text{where } c^{\langle k \rangle} = a^{\langle k \rangle} \oplus b^{\langle k \rangle}$$
$$a \mathbin{\hat{\otimes}} b = c \quad \text{where } c^{\langle k \rangle} = \bigoplus [a^{\langle i \rangle} \otimes b^{\langle j \rangle} \mid i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = k]$$

- Understanding as polynominal

$v \in \alpha^{C_p}$ can be seen as a polynominal: $v = v_1 \cdot c_1 \oplus v_2 \cdot c_2 \oplus \cdots \oplus v_{|C_p|} \cdot c_{|C_p|}$.
The sum $v \mathbin{\hat{\oplus}} v'$ is given as

$$\begin{aligned} v \mathbin{\hat{\oplus}} v' &= (v_1 \cdot c_1 \oplus v_2 \cdot c_2 \oplus \cdots \oplus v_{|C_p|} \cdot c_{|C_p|}) \mathbin{\hat{\oplus}} (v'_1 \cdot c_1 \oplus v'_2 \cdot c_2 \oplus \cdots \oplus v'_{|C_p|} \cdot c_{|C_p|}) \\ &= (v_1 \oplus v'_1) \cdot c_1 \oplus (v_2 \oplus v'_2) \cdot c_2 \oplus \cdots \oplus (v_{|C_p|} \oplus v'_{|C_p|}) \cdot c_{|C_p|} . \end{aligned}$$

The product $v \mathbin{\hat{\otimes}} v'$ is given as follows.

$$\begin{aligned} v \mathbin{\hat{\oplus}} v' &= (v_1 \cdot c_1 \oplus v_2 \cdot c_2 \oplus \cdots \oplus v_{|C_p|} \cdot c_{|C_p|}) \mathbin{\hat{\otimes}} (v'_1 \cdot c_1 \oplus v'_2 \cdot c_2 \oplus \cdots \oplus v'_{|C_p|} \cdot c_{|C_p|}) \\ &= (v_1 \otimes v'_1) \cdot (c_1 \oplus_p c_1) \oplus (v_1 \oplus v'_2) \cdot (c_1 \oplus_p c_2) \oplus \cdots \oplus (v_{|C_p|} \oplus v'_{|C_p|}) \cdot ((c_{|C_p|} \oplus_p c_{|C_p|}) \\ &= \bigoplus [a^{\langle i \rangle} \otimes b^{\langle j \rangle} \mid i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = 1] \cdot c_1 \oplus \cdots \\ &\quad \oplus \bigoplus [a^{\langle i \rangle} \otimes b^{\langle j \rangle} \mid i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = |C_p|] \cdot c_{|C_p|} \end{aligned}$$

Question: Do Similar Problems Have Efficient Algorithms?

- Nested reduction:

$$\bigoplus [\bigotimes [f\ a \mid a \leftarrow y] \mid y \leftarrow gog\ x, p\ y]$$

- Examples
  - The even-sum maximum subsequence sum **Done!**

$$\uparrow [\sum [a \mid a \leftarrow s] \mid s \leftarrow subs\ x, evensum\ s]$$

  - The even-sum maximum initial-segment sum **Next target**

$$\pi(\hat{\uparrow} [\hat{\sum} [\widehat{id}\ a \mid a \leftarrow s] \mid s \leftarrow inits\ x])$$

  - k-maximum maximum initial-segment sum

$$\uparrow_k [[\sum [a \mid a \leftarrow s]] \mid s \leftarrow inits\ x, evensum\ s]$$

  - The number of subsequences matching a regular expression

$$\sum [1 \mid s \leftarrow subs\ x, p_{RE}\ s]$$

GoGs by Predicates and Boolean Markings (based on the idea of [Sasano et al. 00])

Def. Given a FRH predicate $p = accept \circ ([\oplus, f])$, a FRH GoG gog is give as

$$gog\ x = [clean\ y \mid y \leftarrow subs'\ x, p\ y].$$

Here, $clean\ x = +\!+[\textbf{if } m \textbf{ then } [a] \textbf{ else } [] \mid (a, m) \leftarrow x]$ is a cleaner to remove all marks and elements marked false.

- *subs'* *x* generates all possible marking of x with Boolean:
- *subs'* [1,2,3] = [[(1, T), (2, T), (3, T)], [(1, T), (2, T), (3, F)], ⋯, [(1, F), (2, F), (3, T)], [(1, F), (2, F), (3, F)]]

- For example, we can define *inits* with

$$accept\ (i, s, n) = i\ ; \quad f\ (a, m) = (m, m, not\ m)$$
$$(i_1, a_1, n_1) \oplus (i_2, a_2, n_2) = ((i_1 \wedge n_2) \vee (a_1 \wedge i_2), a_1 \wedge a_2, n_1 \wedge n_2).$$

- We can similarly define *tails* to generate all tail segments.

Optimization for FRH GoGs

- Now, we can use the semiring construction

$$
\begin{aligned}
& \bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow gog\ x, p\ y] \\
= & \quad \{\ \text{Semiring construction}\ \} \\
& \hat{\pi}(\hat{\bigoplus}[\hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y] \mid y \leftarrow gog\ x]) \\
= & \quad \{\ \text{Definition of FRH GoGs}\ \} \\
& \hat{\pi}(\hat{\bigoplus}[\hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y] \mid y \leftarrow \\
& \quad +\!\!+[\chi_{++}[\textbf{if}\ m\ \textbf{then}\ [[a]]\ \textbf{else}\ [] \mid (a, m) \leftarrow x] \mid y \leftarrow subs'\ x, p'\ y]]) \\
= & \quad \{\ \text{Short-cut Futon}\ \} \\
& \hat{\pi}(\hat{\bigoplus}[\hat{\bigotimes}'[\hat{f}'\ a \mid a \leftarrow y] \mid y \leftarrow subs'\ x, p'\ y]) \\
= & \quad \{\ \text{Semiring construction}\ \} \\
& \hat{\pi}(\hat{\pi}'(\hat{\hat{\bigoplus}}[\hat{\hat{\bigotimes}}[\hat{\hat{f}}'a \mid a \leftarrow y] \mid y \leftarrow subs'\ x])) \\
= & \quad \{\ \text{The simple optimization}\ \} \\
& \hat{\pi}''(\hat{\hat{\bigotimes}}[\hat{\hat{f}}'_{\hat{\bigoplus}}\ a \mid a \leftarrow x])
\end{aligned}
$$

Question: Do Similar Problems Have Efficient Algorithms?

- Nested reduction:

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow gog\ x, p\ y]$$

- Examples
  - The even-sum maximum subsequence sum **Done!**

$$\uparrow[\sum[a \mid a \leftarrow s] \mid s \leftarrow subs\ x, evensum\ s]$$

  - The even-sum maximum initial-segment sum **Done!**

$$\uparrow[\sum[a \mid a \leftarrow s] \mid s \leftarrow inits\ x, evensum\ s]$$

  - k-maximum maximum initial-segment sum

$$\uparrow_k[[\sum[a \mid a \leftarrow s]] \mid s \leftarrow inits\ x, evensum\ s]$$

  - The number of subsequences matching a regular expression

$$\sum[1 \mid s \leftarrow subs\ x, p_{RE}\ s]$$

(stop over) Finding Solutions as well as Values

- The even-sum maximum initial-segment sum

$$\uparrow[\sum[a \mid a \leftarrow s] \mid s \leftarrow inits\ x, evensum\ s]$$

- We can compute the initial-segment contributing to the maximum sum:

$$\uparrow'[\sum'[(a, [a]) \mid a \leftarrow s] \mid s \leftarrow inits\ x, evensum\ s]$$

$$\textbf{where}\ (a, x) \uparrow' (b, y) = \textbf{if}\ a \geq b\ \textbf{then}\ (a, x)\ \textbf{else}\ (b, y)$$
$$(a, x) +' (b, y) = (a + b, x +\!\!+ y)$$

  - $((\alpha, [\alpha]), \uparrow', +')$ is a semiring
    - Segments which have the same sum are considered equivalent

(stop over)
Simple Querying (generate-and-test)

- We have a linear work algorithm for generate-and-test
  - Because $(\{[\alpha]\}, \uplus, \bigtriangledown_{++})$ is a semiring

$$\uplus[\bigtriangledown_{++}[\{[a]\} \mid a \leftarrow s] \mid s \leftarrow inits\ x, evensum\ s]$$
$$\textbf{where}\ u \bigtriangledown_{\oplus} v = \{a \oplus b \mid a \in u, b \in v\}$$

$\uplus$ is the bag concatenation.
$\{$ and $\}$ are brakets for bags (multi-sets).

  - We need to freeze the operators for efficiency

Question:
Do Similar Problems Have Efficient Algorithms?

- Nested reduction:

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow gog\ x, p\ y]$$

- Examples
  - The even-sum maximum subsequence sum **Done!**

$$\uparrow[\sum[a \mid a \leftarrow s] \mid s \leftarrow subs\ x, evensum\ s]$$

  - The even-sum maximum initial-segment sum **Done!**

$$\uparrow[\sum[a \mid a \leftarrow s] \mid s \leftarrow inits\ x, evensum\ s]$$

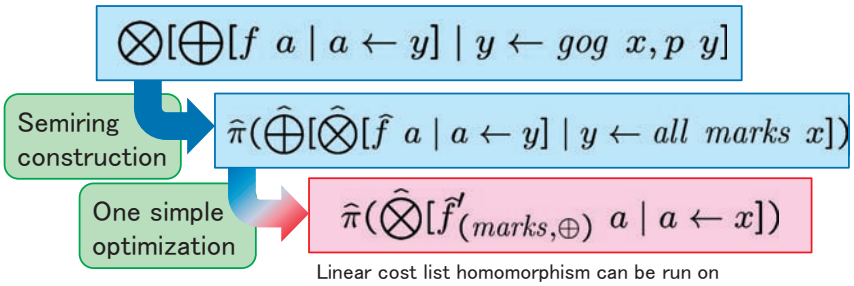
  - k-maximum maximum initial-segment sum **Done!**

$$\uparrow_k[\sum\nolimits_k[[a] \mid a \leftarrow s] \mid s \leftarrow inits\ x, evensum\ s]$$

  - The number of subsequences matching a regular expression **Done!**

$$\sum[\prod[w\ a \mid a \leftarrow s] \mid s \leftarrow subs\ x, p_{RE}\ s]$$

Summary of the Results



Linear cost list homomorphism can be run on SkeTo, MapReduce, ITBB, OpenMP, MPI, ⋯

- Optimizable nested reductions
  - GoG = concatMap-composition of FRH GoGs
  - p = FRH predicates and their and/or/not
  - $(\alpha, \oplus, \otimes)$ is a semiring

Related Work

- Maximum (Multi-) Marking Problem [Sasano et al. 00, 01, 05]
  - Deriving efficient sequential algorithm for nested reductions

$$\uparrow [\sum [f\ a \mid a \leftarrow y] \mid y \leftarrow subs'\ x, p\ y]$$

  - [Sasano et al. 05] deals with accumulative weight sum
  - Parallelization for the simplest case [Matsuzaki 07]

- Pre-order construction for max-sum problems [Morihata 09]
  - Similar framework focusing on semirings of pre-orders and monotonic summations
  - Generic formalization of generators

Conclusion and Future Work

- Automatic derivation of efficient algorithms for nested reductions with semirings, FRH GoGs and FRH predicates
  - Embedding predicates (filters) into semirings
  - Very clear optimization about all-marks generation
  - Serious study about optimization of nested homomorphisms
- Future work
  - Implementation of the derivation
    - Combinator library, DSL library, …
  - Optimization of lifted operators
    - FR foldr = FR homomorphism with a large number of states
  - Relational (accumulative) predicates
    - Correspond to accumulative weight function in MMP [Sasano et al. 05]
  - Apply semiring construction to combinatorial problems on graphs
    - $\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow g, p\ y] = \hat{\pi}(\widehat{\bigoplus}[\widehat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y] \mid y \leftarrow g])$

Reference

- Kento Emoto:
  An Algebraic Approach to Efficient Parallel Algorithms for Nested Reductions
  Technical Report METR 2011-01, 31 pages, Department of Mathematical Engineering and Information Physics, University of Tokyo, 2011.
- Available from Web:
  http://www.ipl.t.u-tokyo.ac.jp/~emoto/#METR2011-01
  or http://www.keisu.t.u-tokyo.ac.jp/research/techrep/2011.html

Semiring Fusion

Sebastian Fischer*
National Institute of Informatics, Tokyo

Programmers can write and reason about compositional programs that communicate via intermediate data easily, but processors can execute monolithic programs that do not allocate memory more efficiently. Short-cut fusion (Gill 1996; Gill et al. 1993) is a technique to eliminate intermediate data shared between composed functions.

Traditionally, short-cut fusion does not improve the complexity of the transformed algorithm. The generated program is more efficient only by a constant factor compared with the program it is derived from. Identifying the type of the eliminated intermediate date as free algebraic structure allows to apply short-cut fusion to improve the complexity of algorithms. Every algebraic structure, like monoid or semiring, gives rise to a short-cut fusion law that can be proved using a free theorem (Reynolds 1983; Wadler 1989). Using an algebraic structure that satisfies a distributive law gives rise to asymptotic improvement of algorithms transformed by short-cut fusion.

The key idea is that on the different sides of the distributive law "$a \cdot (b+c) = a \cdot b + a \cdot c$" the number of arithmetic operations are different. Semirings are an example for a distributive algebraic structure. The following slides describe short-cut fusion with the free semiring, multisets of lists.

References

Gill, Andrew. 1996. Cheap deforestation for non-strict functional languages. Ph.D. thesis, The University of Glasgow.

Gill, Andrew, John Launchbury, and Simon L. Peyton Jones. 1993. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 223–232. FPCA '93, New York, NY, USA: ACM.

Reynolds, John C. 1983. Types, abstraction and parametric polymorphism. In *IFIP Congress*, 513–523.

Wadler, Philip. 1989. Theorems for free! In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 347–359. FPCA '89, ACM Press.

*supported by a post-doc fellowship of the German Academic Exchange Service (DAAD)

Semiring Fusion

Sebastian Fischer

DAAD Research Fellow at NII, Tokyo

Fourth DIKU-IST Joint Workshop on Foundations of Software

Quiz

How many sublists[1] has $[1\,..\,n]$?

What is the sum of products over all sublists of $[2, -1, 3]$?

[1] some (not necessarily adjacent) elements in correct order

Preview

shortcut fusion

algebraic view on shortcut fusion

semirings as underlying structure

Linear algorithms to answer the posed questions can be obtained from seemingly exponential specifications.

Shortcut Fusion

Programmers can write easily

- compositional programs
- that communicate via intermediate data

Processors can execute efficiently

- tight loops
- that do not allocate memory

Simple Example

Without intermediate data:

```
factorial :: Int → Int
factorial n =
  if n < 1 then 1
  else n * factorial (n − 1)
```

Compositional: factorial = product ∘ downFrom

```
prod :: [Int] → Int
prod []       = 1
prod (n : ns) = n * prod ns
```

```
downFrom :: Int → [Int]
downFrom n =
  if n < 1 then []
  else n : downFrom (n − 1)
```

Abstracting from list constructors

```
genDownFrom :: Int → (Int → list → list) → list → list
genDownFrom n cons nil =
  if n < 1 then nil
  else cons n (genDownFrom (n − 1) cons nil)
```

```
build :: (∀list.(Int → list → list) → list → list) → [Int]
build gen = gen (:) []   -- foldr f e (build g) = g f e
```

```
prod (downFrom n) = foldr (*) 1 (build (genDownFrom n))
                  = genDownFrom n (*) 1
                  = factorial n   -- monolithic version
```

Algebraic View

Abstracting the list interface

```
class IntList list where
  cons :: Int → list → list
  nil  :: list
```

Homomorphisms / folds:

```
intListHom :: IntList list ⇒ [Int] → list
intListHom []       = nil
intListHom (n : ns) = cons n (intListHom ns)
```

build with overloaded function

```
instance IntList [Int] where
  cons = (:)
  nil  = []
```

```
buildIntList :: (∀list.IntList list ⇒ list) → [Int]
buildIntList gen = gen
```

Factorial again

```
downFromIntList :: IntList list ⇒ Int → list
downFromIntList n =
  if n < 1 then nil
  else cons n (downFromIntList (n − 1))
```

```
instance IntList Int where
  cons = (∗)
  nil  = 1
```

Free Theorem[2] for IntList generator

The Free Theorem for gen :: ∀list.IntList list ⇒ Int → list

$$\forall\, t_1,t_2 \in \mathsf{TYPES}(\textsc{IntList}),\ f :: t_1 \to t_2,\ f\ \textit{respects}\ \textsc{IntList}.$$
$$\forall\, x :: \textsc{Int}.\ f\ (gen\_t_1\ x) = gen\_t_2\ x$$

$$f\ \textit{respects}\ \textsc{IntList}\ \textit{if}$$
$$\forall\, x :: \textsc{Int}.$$
$$\forall\, y :: t_1.\ f\ (cons\_t_1\ x\ y) = cons\_t_2\ x\ (f\ y)$$
$$f\ nil\_t_1 = nil\_t_2$$

[2] http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi

Shortcut Fusion, revisited

- downFromIntList :: IntList list ⇒ Int → list
- intListHom respects IntList (is homomorphism)
- prod = intListHom
- downFrom = buildIntList ∘ downFromIntList

```
prod (downFrom n)
  = intListHom (buildIntList (downFromIntList n))
  = intListHom (downFromIntList n)   -- by definition
  = downFromIntList n                -- by free theorem
```

Monoids

Alternative list interface:

```
class Monoid m where
  one :: m
  (⊗) :: m → m → m
```

Must satisfy laws:

- one is unit of ⊗: one ⊗ x = x = x ⊗ one
- ⊗ is associative: x ⊗ (y ⊗ z) = (x ⊗ y) ⊗ z

Monoids and Homomorphisms

```
instance Monoid [a] where
  one = []
  (⊗) = (++)
instance Monoid Int where
  one = 1
  (⊗) = (*)
```

```
monoidHom :: Monoid m ⇒ (a → m) → [a] → m
monoidHom f []       = one
monoidHom f [x]      = f x
monoidHom f (xs ++ ys) = monoidHom f xs ⊗ monoidHom f ys
```

Monoid Generator and Free Theorem

```
downFromM :: Monoid m ⇒ (Int → m) → Int → m
downFromM f n =
  if n < 1 then one
  else f n ⊗ downFromM f (n − 1)
```

Free Theorem:

```
hom ∘ gen f = gen (hom ∘ f)
```

Consequence:

```
factorial
  = prod ∘ downFrom
  = monoidHom id ∘ downFromM (λx → [x])
  = downFromM (λx → monoidHom id [x])   -- by free theorem
  = downFromM id                        -- by definition
```

Associativity

Interesting:

- more freedom for implementation of homomorphisms
- parallel execution by nesting in a balanced way

Boring:

- same number of multiplications, regardless of nesting
- fusion improves efficiency only by a constant factor

Distributivity

Changes number of operations:

$$a \cdot (b + c) = a \cdot b + a \cdot c$$
$$(a + b) \cdot c = a \cdot c + b \cdot c$$

Extreme example:

$$\begin{aligned}(1 + x_1) \cdot \ldots \cdot (1 + x_n) = 1 \\ + x_1 + \ldots + x_n \\ + x_1 \cdot x_2 + \ldots + x_i \cdot x_k + \ldots + x_{n-1} \cdot x_n \\ \vdots \\ + x_1 \cdot \ldots \cdot x_n\end{aligned}$$

$O(n)$ vs. $O(2^n)$ operations

Semirings

Extension of monoids:

```
class Monoid s ⇒ Semiring s where
  zero :: s
  (⊕) :: s → s → s
```

Laws:

- zero is unit of $\oplus$
- $\oplus$ is associative and commutative
- $\otimes$ distributes over $\oplus$
- zero cancels multiplication

```
instance Semiring Int where zero = 0; (⊕) = (+)
```

Another Semiring

```
type Bag a   -- abstract type of multisets
instance Semiring (Bag [a]) where
  zero   = ∅
  a ⊕ b = a ∪ b
instance Monoid (Bag [a]) where
  one    = {[]}
  a ⊗ b = {x ++ y | x ∈ a, y ∈ b}
single :: a → Bag [a]
single x = {[x]}
```

Example:

$$\begin{aligned}&(\text{single } 1 \oplus \text{single } 2) \otimes \text{single } 3 \\ &\quad = (\{[1]\} \cup \{[2]\}) \otimes \{[3]\} \\ &\quad = \{[1],[2]\} \otimes \{[3]\} \\ &\quad = \{[1,3],[2,3]\}\end{aligned}$$

Homomorphisms

```
semiringHom :: Semiring s ⇒ (a → s) → Bag [a] → s
semiringHom f { }         = zero
semiringHom f {[]}        = one
semiringHom f (single x) = f x
semiringHom f (a ⊕ b)     = semiringHom f a ⊕ semiringHom f b
semiringHom f (a ⊗ b)     = semiringHom f a ⊗ semiringHom f b
```

Sublists

```
sublists :: [a] → Bag [a]
sublists []       = one
sublists (x : xs) = (one ⊕ single x) ⊗ sublists xs
```

Example:

$$
\begin{aligned}
&\texttt{sublists}\ [2,-1,3] \\
&\quad = \{[],[2]\} \otimes \{[],[-1]\} \otimes \{[],[3]\} \\
&\quad = \{[],[3],[-1],[-1,3],[2],[2,3],[2,-1],[2,-1,3]\}
\end{aligned}
$$

Counting sublists

```
size :: Bag [a] → Int
size = semiringHom (λx → 1)
```

$$
\begin{aligned}
&\texttt{size}\ (\texttt{sublists}\ [1,2,3]) \\
&\quad = \texttt{size}\ \{[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]\} \\
&\quad = 1+1+1+1+1*1+1*1+1*1+1*1*1 \\
&\quad = 1+1+1+1+1+1+1+1 \\
&\quad = 8
\end{aligned}
$$

$\texttt{size}\ (\texttt{sublists}\ [1\,..\,n]) = 2^n$

$O(2^n)$ operations

Sum of Products

```
prodSum :: Bag [Int] → Int
prodSum = semiringHom id
```

```
prodSum (sublists [2, −1, 3])
  = prodSum {[], [3], [−1], [−1, 3], [2], [2, 3], [2, −1], [2, −1, 3]}
  = 1 + 3 − 1 − 3 + 2 + 2 ∗ 3 − 2 ∗ 1 − 2 ∗ 1 ∗ 3
  = 1 + 3 − 1 − 3 + 2 + 6 − 2 − 6
  = 0
```

$O(2^n)$ operations

Generalized Sublists

```
genSublists :: Semiring s ⇒ (a → s) → [a] → s
genSublists f []       = one
genSublists f (x : xs) = (one ⊕ f x) ⊗ genSublists f xs
```

```
sublists = genSublists single
```

Semiring Fusion

Free Theorem:

```
hom ∘ gen f = gen (hom ∘ f)
```

Consequence:

```
semiringHom f ∘ gen single
  = gen (λx → semiringHom f (single x))   -- by free theorem
  = gen f                                 -- by definition
```

Size, efficiently

$$
\begin{aligned}
&(\mathsf{size} \circ \mathsf{sublists})\ [1,2,3] \\
&\quad = (\mathsf{semiringHom}\ (\lambda x \to 1) \circ \mathsf{genSublists}\ \mathsf{single})\ [1,2,3] \\
&\quad = \mathsf{genSublists}\ (\lambda x \to 1)\ [1,2,3] \\
&\quad = (1+1) * (1+1) * (1+1) \\
&\quad = 2 * 2 * 2 \\
&\quad = 8
\end{aligned}
$$

$O(n)$ operations

Sum of Products, efficiently

$$
\begin{aligned}
&(\mathsf{prodSum} \circ \mathsf{sublists})\ [2,-1,3] \\
&\quad = (\mathsf{semiringHom}\ \mathsf{id} \circ \mathsf{genSublists}\ \mathsf{single})\ [2,-1,3] \\
&\quad = \mathsf{genSublists}\ \mathsf{id}\ [2,-1,3] \\
&\quad = (1+2) * (1-1) * (1+3) \\
&\quad = 3 * 0 * 4 \\
&\quad = 0
\end{aligned}
$$

$O(n)$ operations

Review

shortcut fusion can be viewed through algebraic glasses

eliminated intermediate data is free algebraic structure

distributivity gives rise to asymptotic improvement

Partial Evaluation of Janus

Torben Ægidius Mogensen
DIKU
torbenm@diku.dk

February 15, 2011

Abstract

A reversible programming language is a programming language in which you can only write reversible programs, i.e., programs that can be run both forwards (computing outputs from inputs) and backwards (computing inputs from outputs). It is interesting to study reversible programs and languages because computations on reversible computers (computers that only allow reversible programs) in theory can be done using less energy than computations on traditional irreversible computers. Janus is a reversible, structured imperative programming language.

We present a partial evaluator for the full Janus language with the exception of procedure calls. The partial evaluator converts Janus programs into reversible flowcharts, specialises these using polyvariant specialisation and converts the result back to structured form. Reversibility adds some complications, which we address in the paper. We demonstrate the results by some small examples.

We believe this to be the first partial evaluator for a deterministic reversible programming language.

This paper was first presented at the DIKU-IST 2011 workshop in Tokyo and later presented in a slightly different form at PEPM'2011 in Austin, Texas.

1 Introduction

Reversible computation [14, 19, 3, 4, 9] can theoretically be done using less energy than irreversible computation, as erasure of information necessarily dissipates energy in the form of thermodynamic entropy [14]. Most studies use low-level computational models such as reversible Turing machines [18, 3], but some studies use structured reversible programming languages[16, 1, 24, 22, 23].

Partial evaluation [13, 11] is a technique for generating specialised programs by fixing the value of some of the inputs to more general programs. The intent is that the specialised programs are more efficient than the originals, and this has often been observed in practise. Specialised programs perform fewer computations (and, hence, use less energy), so they can be of interest to further reduce energy consumption.

The partial evaluator presented in this paper handles all of the Janus language except procedure calls, which we address in a follow-up paper.

2 The reversible language Janus

Janus is a structured reversible programming language originally designed for a class at Caltech [16]. A Janus program starts with a declaration of variables divided into inputs, outputs and other variables. Variables are either integer variables or arrays of integers. The size of an array is either a constant number or given by a previously declared input variable. The main part of a Janus program is a list of parameter-less procedure declarations and a sequence of reversible statements that can use conditionals, loops and procedure calls. A special feature is that procedures can be run backwards by calling them with the keyword `uncall` instead of `call`. A grammar for Janus is shown in figure 1 and figure 2 shows a few examples of Janus programs:

(a) Fibonacci. This Janus program takes a number n and returns both the nth and the $(n+1)$th Fibonacci numbers.

(b) Multiplication. This program takes two odd numbers `a` and `b` and returns both their product and `b` (unchanged) as outputs. Running this in reverse divides the product by `b`.

(c) Postfix interpreter. This program reads a postfix expression (represented as an array of numbers) and an array of input values and then outputs the expression and its result. Outputting the expression with the result is required for reversibility. Evaluation uses two stacks: An evaluation stack `stack` and a stack `garbage` that is used to ensure reversibility. Uncalling `calc` undoes the stack operations, so the stacks are, again, empty.

2.1 Informal semantics of Janus

We will only describe Janus informally and refer to [24] for a formal semantics.

Variables and array elements that are not inputs are initialised to 0 and variables and array elements that are not outputs are verified to be 0 when the program ends. A variable or array can be both input and output. Statements can take the following forms:

Update. The left-hand side of an update is either an integer variable or an element of an array. The update can either add or subtract the value of the right-hand side to this. The right-hand side can be any expression that does *not* contain the variable or array used on the left-hand side and if the left-hand side is an array element, the array can not be used in the expression specifying the index into the array. For example, the update `a[a[i]] += 1` is not legal. These restrictions ensures that the update can be reversed. Expressions can use the operators +, - and `/2`.

Swap. A statement of the form lv_1 `<=>` lv_2 swaps the contents of lv_1 and lv_2, which can be integer variables or array elements. It is possible to swap two elements of the same array, but the index expression of an array can contain none of the arrays or integer variables used in the swap statement. For example, while `a[i] <=> a[j]` is legal, `a[i] <=> i` is not. Again, the restriction is required for reversibility.

Sequence. Statements separated by `;` are executed in sequence.

Skip. No effect.

$$
\begin{array}{lcl}
\mathit{Prog} & \rightarrow & \mathit{Dec}^* \texttt{ -> } \mathit{Dec}^* \ (\texttt{with } \mathit{Dec}^*)^? \texttt{ ; } \mathit{Stat}\ \mathit{Proc}^* \\
\mathit{Dec} & \rightarrow & \textbf{id} \\
\mathit{Dec} & \rightarrow & \textbf{id } \texttt{[} \textbf{size } \texttt{]} \\
\mathit{Stat} & \rightarrow & \mathit{Lval} \texttt{ += } \mathit{Exp} \\
\mathit{Stat} & \rightarrow & \mathit{Lval} \texttt{ -= } \mathit{Exp} \\
\mathit{Stat} & \rightarrow & \mathit{Lval} \texttt{ <=> } \mathit{Lval} \\
\mathit{Stat} & \rightarrow & \mathit{Stat} \texttt{ ; } \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \texttt{skip} \\
\mathit{Stat} & \rightarrow & \texttt{if } \mathit{Cond} \texttt{ then } \mathit{Stat} \texttt{ else } \mathit{Stat} \texttt{ fi } \mathit{Cond} \\
\mathit{Stat} & \rightarrow & \texttt{from } \mathit{Cond} \texttt{ do } \mathit{Stat} \texttt{ loop } \mathit{Stat} \texttt{ until } \mathit{Cond} \\
\mathit{Stat} & \rightarrow & \texttt{call } \textbf{id} \\
\mathit{Stat} & \rightarrow & \texttt{uncall } \textbf{id} \\
\mathit{Lval} & \rightarrow & \textbf{id} \\
\mathit{Lval} & \rightarrow & \textbf{id } \texttt{[} \mathit{Exp} \texttt{]} \\
\mathit{Exp} & \rightarrow & \textbf{num} \\
\mathit{Exp} & \rightarrow & \mathit{Lval} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} \texttt{ + } \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} \texttt{ - } \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} \texttt{ /2} \\
\mathit{Exp} & \rightarrow & \texttt{(} \mathit{Exp} \texttt{)} \\
\mathit{Cond} & \rightarrow & \mathit{Exp} \texttt{ < } \mathit{Exp} \\
\mathit{Cond} & \rightarrow & \mathit{Exp} \texttt{ == } \mathit{Exp} \\
\mathit{Cond} & \rightarrow & \texttt{odd(}\mathit{Exp}\texttt{)} \\
\mathit{Cond} & \rightarrow & \texttt{! } \mathit{Cond} \\
\mathit{Cond} & \rightarrow & \mathit{Cond} \texttt{ \&\& } \mathit{Cond} \\
\mathit{Cond} & \rightarrow & \mathit{Cond} \texttt{ || } \mathit{Cond} \\
\mathit{Cond} & \rightarrow & \texttt{(} \mathit{Cond} \texttt{)} \\
\mathit{Proc} & \rightarrow & \texttt{procedure } \textbf{id } \mathit{Stat}
\end{array}
$$

Figure 1: Syntax of Janus

Conditional. A statement of the form `if` c_1 `then` s_1 `else` s_2 `fi` c_2 is executed by first evaluating c_1. If this is true, s_1 is executed and it is verified that c_2 is true. If c_1 is false, s_2 is executed and it is verified that c_2 is false. If the exit-assertion c_2 does not have the expected value, the program stops with an error message. A condition can compare numbers for equality (==), inequality (<) and test if a number is odd. Conditions can be combined by conjunction, disjunction and negation. The construction can be illustrated by the flowchart in figure 3(a), where a two-entry assertion is shown as a circle with two entry arrows marked with the expected truth value.

Loop. A statement of the form `from` c_1 `do` s_1 `loop` s_2 `until` c_2 is executed by first evaluating the assertion c_1. If this is false, the program stops with an error message, otherwise s_1 is executed and the c_2 is evaluated. If this is true, the loop terminates. Otherwise, s_2 is executed and c_1 is evaluated (again). If c_1 is true, the program stops with an error message, otherwise the loop repeats from s_1. The construction can be illustrated with the flowchart in figure 3(b).

Procedure call. A procedure call is either of the form `call` p or `uncall` p, where p is a procedure name. `call` p executes the body of p and returns to the place of the call.

```
n -> a b;

a += 0; b += 1;
from a==0 do
  n -= 1; a <=> b; b += a
loop skip
until n==0
```

(a) Fibonacci

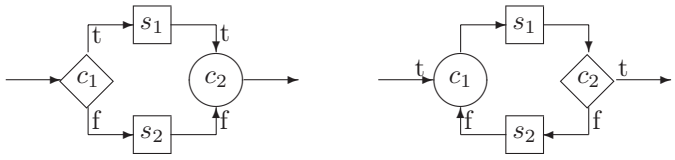
```
a b -> b prod with t v;

from 0==prod do
  if odd(a) then
    prod += b; t += a/2;
    a -= t+1; t -= a
  else
    t += a/2; a -= t; t -= a
  fi !(prod<b)
loop
  v += b; b += v; v -= b/2
until a==0;

from prod<b+b do
  v += b/2; b -= v; v -= b
loop skip
until odd(b)
```

(b) Multiplication

```
     sz exp[sz] i ins[i]
->   sz exp[sz] i ins[i] result
with pc sp stack[sz] gp garbage[sz];

call calc;
result += stack[0];
uncall calc

procedure calc
from pc==0 do
  if exp[pc]==0 then
    stack[sp] += exp[pc+1];
    sp += 1
  else
    if exp[pc]==1 then
      stack[sp] += ins[exp[pc+1]]; sp += 1
    else // exp[pc]==2
      sp -= 1; garbage[gp] <=> stack[sp];
      if exp[pc+1]==0 then
        stack[sp-1] += garbage[gp]
      else
        stack[sp-1] -= garbage[gp]
      fi exp[pc+1]==0;
      gp += 1
    fi exp[pc]==1
  fi exp[pc]==0;
  pc += 2
loop skip
until pc==sz
```

(c) Postfix interpreter

Figure 2: Janus programs



(a) Conditional (b) Loop

Figure 3: Flowchart diagrams for conditional and loop

`uncall` p executes the body of p *in reverse* and then returns to the place of the call. The sequence `call` p`;` `uncall` p has no net effect, as `uncall` p will undo all the state changes done by `call` p.

Note that a Janus program can either terminate normally, fail or be nonterminating. We regard all failures as equivalent, i.e., we do not differentiate between failing an assertion in a loop or conditional or failing due to a non-zero non-output variable.

2.2 Reverse execution

Statements can be executed both forwards and backwards (when a procedure is called with `uncall`). Backwards execution can be realised by syntactically reversing the statement and then executing it forwards. A program can be reversed by swapping input and output variables and reversing the body statement. The function R below shows how statements can be reversed.

$$
\begin{aligned}
R(lv \texttt{ += } e) &= lv \texttt{ -= } e \\
R(lv \texttt{ -= } e) &= lv \texttt{ += } e \\
R(lv_1 \texttt{ <=> } lv_2) &= lv_1 \texttt{ <=> } lv_2 \\
R(s_1;\ s_2) &= R(s_2);\ R(s_1) \\
R(\texttt{call } p) &= \texttt{uncall } p \\
R(\texttt{uncall } p) &= \texttt{call } p \\
R(\texttt{skip}) &= \texttt{skip} \\
R(\texttt{if } c_1 \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi } c_2) &= \texttt{if } c_2 \texttt{ then } R(s_1) \texttt{ else } R(s_2) \texttt{ fi } c_1 \\
R(\texttt{from } c_1 \texttt{ do } s_1 \texttt{ loop } s_2 \texttt{ until } c_2 &= \texttt{from } c_2 \texttt{ do } R(s_1) \texttt{ loop } R(s_2) \texttt{ until } c_1
\end{aligned}
$$

3 Partial evaluation

Partial evaluation has been studied for many languages. Good overviews can be found in [13] and [11].

Briefly stated, partial evaluation is about producing residual programs. If a program p takes two inputs and a r takes one input, r is a residual program of p with respect to a value s if, for any value d, running p with s and d as inputs yields the same result as running r with d as input. I.e., either both terminate and produce the same result, both fail with an error or both are nonterminating. A partial evaluator is a program that given (representations of) a program p and a value s can produce (a representation of) a program r that is residual to p with respect to s. Often, the definition of a partial evaluator is relaxed, so it is allowed to be nonterminating for some p and s. In this relaxed definition, the always nonterminating program is a partial evaluator, just not a very useful one.

These definitions generalise to cases where p has more inputs.

We will focus on *offline* partial evaluation: Inputs are classified in advance as known (static) or unknown (dynamic), and it is determined which parts of the program depend only on the static inputs and which parts may depend on the remaining (dynamic) inputs. When values for the static inputs are given, this classification is used to determine which parts of the program can be executed during specialisation and which parts must be included in specialised form in the residual program. Thus, the specialisation process is divided into the following steps:

1. Classify input variables as static (to be specialised away) or dynamic (to remain in the residual program).

2. Given the input classification, automatically classify all parts of the program as static or dynamic. This is called a *binding-time analysis*. Input variables that in step 1 were classified as static may be reclassified as dynamic. The binding-time analysis can ensure that all residual programs obey certain properties. We will use it to ensure that residual programs are reversible.

3. Given the full program classification and the values of the inputs classified as static, produce the specialised (residual) program by executing the static parts and copying instances of the dynamic parts into the residual program. The dynamic parts of the program may contain static values as constants, so the residual program can contain several copies of the same dynamic program part with different static values as constants. This phase is called *specialisation*.

The residual program can now be run with the remaining (dynamic) input values and will produce the same result as running the original program with all inputs. You can run the residual program repeatedly with different dynamic input values.

As an example of offline specialisation, consider the multiplication program shown in figure 2(b). We might want to specialise the program with the value 11 for `a` to get a one-input program that takes `b` and returns `b` and the product of 11 and `b`.

In this example, all outputs depend on the dynamic input and are, hence, outputs of the residual program. But in other programs there can be outputs that only depend on static inputs. If we follow the usual definition of a residual program, the residual program must return *all* outputs of the original program, including outputs that don't depend on the dynamic inputs and are, hence, the same every time. We will relax this requirement, so the partial evaluator will return both a residual program and the values of the outputs that depend only on static inputs, i.e., the static outputs. The residual program will, then, only return the outputs that depend on dynamic inputs, i.e., the dynamic outputs. The reason for this relaxation is that reversibility often requires a Janus program to return parts of its input along with the "interesting" output. For example, an interpreter written in Janus usually returns both the interpreted program and the result of running it. If we specialise the interpreter to a program, we don't want the residual program to return this program as part of its output. Hence, we define the following correctness criterion for offline partial evaluation for Janus:

Given a a program p and an initial classification of inputs as static or dynamic, the binding-time analysis must produce an annotated program where inputs and outputs are classified as static or dynamic. Some variables that were initially classified as static may now be classified as dynamic.

When, given the annotated program and values s of the static inputs, specialisation terminates with static output s' and a residual program r, the following must hold:

- If running p with inputs s and d terminates with outputs o, the part of o that was classified as static must have the value s' and if the part of the output that was classified as dynamic has the value d', then running r with d as input must terminate and return d' as output.

- If running p with inputs s and d stops with a failed assertion or does not terminate, running r with d as input must do the same.

These requirements can easily be fulfilled either by the binding-time analysis classifying all inputs and outputs as dynamic and let $r = p$ or by making the specialiser not produce any output. While we characterise such behaviours as correct, they are not desirable. But they

are not always possible to avoid: Consider a program with one input and one output. It is undecidable if the program will terminate without failure, so if we classify the output as static we risk nontermination or failure during specialisation. So to ensure termination, we must reclassify the output as dynamic, which may unnecessarily postpone computations until the residual program is executed. Analyses can determine if specialisation of a given program is guaranteed to terminate [10, 6, 12], but since this is undecidable, such analyses must necessarily be imprecise. We will not address this issue further in this paper but simply allow specialisation to sometimes not terminate.

4 Polyvariant partial evaluation

The standard approach to partial evaluation of imperative languages is polyvariant partial evaluation [5, 13, 11]. Polyvariant partial evaluation allows the same portion of the original program to be specialised to multiple different static states. This can completely change the structure of the program, so unless severe restrictions are imposed, it works best with programs that use unstructured control, i.e., programs that consist of basic blocks that each start with a label and end with a (possibly conditional) jump.

In polyvariant specialisation of basic blocks, the label of a basic block and the values of the static variables (the static state) are combined to make a new residual label. The statements in the basic block are then specialised with respect to this static state and a new static state is obtained. The jump at the end of the basic block is specialised in the following way:

- An unconditional jump to l is made into a residual jump to a residual label made by combining l with the static state.
- A jump with a static condition is made into a residual unconditional jump to a residual label made by combining the selected label with the static state.
- A jump with a dynamic condition is made into a residual conditional jump by specialising the condition to the static state and making two residual labels by combining the original labels with the static state.

If there are not already specialised basic blocks for the residual labels constructed above, these are now produced. When there are specialised basic blocks for all residual labels, the residual program is complete. It is possible that this process will not terminate.

4.1 Translation into flowchart form

A reversible flowchart language similar to Janus is described in [22]. We will convert the body of a Janus program and all the procedure bodies into lists of basic blocks. Each basic block consists of three parts: An entry point, a body and a jump. An entry point can be one of the following:

- `start`, that indicates the block where execution starts.
- A named label.
- A two-entry assertion consisting of a condition c and two named labels l_1 and l_2. This is written as `if` c `from` l_1 l_2. The condition c must be true if the basic block is entered by a jump to l_1 and false if the basic block is entered by a jump to l_2.

No label nor `start` can occur more than once in the entry points of all the basic blocks. The body of a basic block is either empty or any statement that does not contain structured control statements (conditionals and loops). The jump can be:

- `return`, that indicates the end of the program or procedure.
- An unconditional jump `goto` l.
- A conditional jump consisting of a condition c and two named labels l_1 and l_2. This is written as `if` c `goto` l_1 l_2. If c is true, the jump goes to l_1, otherwise to l_2.

No label nor `return` can occur more than once in the jumps of all the basic blocks. A basic block e: s; j, where e is an entry point, s a statement and j a jump is reversed into $R(j)$: $R(s)$; $R(e)$, where R is the statement-reversing function shown in section 2.2 extended to handle entry points and jumps:

$$\begin{array}{rcl@{\qquad}rcl} R(\texttt{start}) & = & \texttt{return} & R(\texttt{return}) & = & \texttt{start} \\ R(l) & = & \texttt{goto}\ l & R(\texttt{goto}\ l) & = & l \\ R(\texttt{if}\ c\ \texttt{from}\ l_1\ l_2) & = & \texttt{if}\ c\ \texttt{goto}\ l_1\ l_2 & R(\texttt{if}\ c\ \texttt{goto}\ l_1\ l_2) & = & \texttt{if}\ c\ \texttt{from}\ l_1\ l_2 \end{array}$$

A structured program is translated by first making the body statements of the program and procedures into single basic blocks by adding the entry point `start` and the jump `return`. These basic blocks may contain structured statements, so we translate them using the function T that translates a basic block that may contain structured statements into a set of basic blocks that do not:

$$\begin{aligned} &T(e:\ s;\ j) = \{e:\ s;\ j\} \\ &\qquad \text{if } s \text{ does not contain structured statements} \\ &T(e:\ s_1\texttt{;}\ \ s_2;\ j) = T(e:\ s_1;\ \texttt{goto}\ l)\ \cup\ T(l:\ s_2\ j) \\ &\qquad \text{where } l \text{ is a new label} \\ &T(e:\ \texttt{if}\ c_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ c_2;\ j) = \\ &\quad \{e:\ \texttt{if}\ c_1\ \texttt{goto}\ l_1\ l_2\} \\ &\quad \cup\ T(l_1:\ s_1;\ \texttt{goto}\ l_3)\ \cup\ T(l_2:\ s_2;\ \texttt{goto}\ l_4) \\ &\quad \cup\ \{\texttt{if}\ c_2\ \texttt{from}\ l_3\ l_4\texttt{:}\ j\} \\ &\qquad \text{where } l_1, l_2, l_3 \text{ and } l_4 \text{ are new labels} \\ &T(e:\ \texttt{from}\ c_1\ \texttt{do}\ s_1\ \texttt{loop}\ s_2\ \texttt{until}\ c_2;\ j) = \\ &\quad \{e:\ \texttt{goto}\ l_1\} \\ &\quad \cup\ T(\texttt{if}\ c_1\ \texttt{from}\ l_1\ l_2\texttt{:}\ s_1\texttt{;}\ \texttt{if}\ c_2\ \texttt{goto}\ l_3\ l_4) \\ &\quad \cup\ T(l_4:\ s_2;\ \texttt{goto}\ l_2)\ \cup\ \{l_3:\ j\} \\ &\qquad \text{where } l_1, l_2, l_3 \text{ and } l_4 \text{ are new labels} \end{aligned}$$

After the translation, there can be trivial basic blocks of the form (l_1 : `goto` l_2). We can eliminate such a block by making the jump to l_1 jump to l_2 instead. The program in figure 2(b) is translated into the flowchart program in figure 4(a).

4.2 Binding-time analysis

We will, in this paper, not handle procedure calls, so we assume that the body of a basic block is a sequence of updates and swaps. For simplicity, we use one global binding time for each variable. so binding time analysis is fairly straightforward:

```
a b  -> b prod with t v ;

start:
goto f_2

if 0==prod from f_2 a_2:
if odd(a) goto t1_3 e1_3

t1_3:
prod += b; t += a/2; a -= t+1; t -= a
goto t2_3

e1_3:
t += a/2; a -= t; t -= a
goto e2_3

if !(prod<b) from t2_3 e2_3:
if a==0 goto f_11 l_2

l_2:
v += b; b += v; v -= b/2
goto a_2

if prod<b+b from f_11 a_11:
v += b/2; b -= v; v -= b
if odd(b) goto u_11 a_11

u_11:
return
```

(a) flowchart

```
a ~b  -> ~b ~prod  with t ~v ;

start:
~goto f_2

~if ~0~=prod from f_2 a_2:
if odd(a) goto t1_3 e1_3

t1_3:
prod ~+= b~; t += a/2; a -= t+1; t -= a
~goto t2_3

e1_3:
t += a/2; a -= t; t -= a
~goto e2_3

~if ~!(prod~<b) from t2_3 e2_3:
if a==0 goto f_11 l_2

l_2:
v ~+= b~;~ b ~+= v~;~ v ~-= b~/2
~goto a_2

~if prod~<b~+b from f_11 a_11:
v ~+= b~/2~;~ b ~-= v~;~ v ~-= b
~if ~odd(b) goto u_11 a_11

~u_11:
return
```

(b) annotated flowchart

Figure 4: Multiplication program as flowchart and annotated flowchart

1. A statement, expression or condition that contains a dynamic variable is dynamic.
2. If an update statement is dynamic, then the variable or array element on the left-hand side is dynamic.
3. If a swap statement is dynamic, the variables or array elements on both sides are dynamic.
4. If one element of an array is dynamic, all elements of the array are dynamic.
5. An array indexed by a dynamic index expression is dynamic.
6. An `if-goto` or `if-from` with dynamic condition is dynamic.
7. A label used in a dynamic `if-goto` or `if-from` is dynamic.
8. An `if-from` with a dynamic label is dynamic.
9. An unconditional jump to a dynamic label is dynamic.
10. `start` and `return` are dynamic.
11. Static subexpressions of dynamic expressions or conditions are enclosed in "lift" operators that indicate that the static values will be inserted as constants during specialisation.

Note that there are three cases for arrays: A fully static array has static elements and is always indexed using static index expressions. A dynamic array has dynamic element values and is always indexed by dynamic index expressions (if not, a lift operator is used to make it so). But we may also have arrays that are statically indexed but contains dynamic element values. We explain specialisation of these in section 4.5

We start binding-time analysis from an initial classification of the input variables as static or dynamic and everything else as static. We then iterate applying the rules above until no changes occur.

We will later need to add extra rules to ensure reversibility of residual programs, but we need to describe the specialisation process first to identify the issues.

The multiplication program shown in figure 4(a) with `a` classified as static and `b` as dynamic results in the binding-time-annotated program shown in figure 4(b). A `~` indicates a dynamic declaration or operation and `^` is the lift operator.

4.3 Specialisation of flowchart programs

As mentioned in section 4, the basic idea is to create specialised basic blocks by combining labels with static state and specialising the statements of the basic block according to the static state (and updating the static state when static variables are changed). Additionally, static jumps can be unfolded. We use the following rules for specialisation:

- Static statements are executed to update the static state.
- Dynamic statements are made into residual statements by evaluating static subexpressions and inserting their values as constants in place of the expressions.
- `start` is specialised to `start`.
- An unconditional dynamic label is specialised by combining it with the static state to construct a new residual label. This is done by hashing the static state to a number and adding this number as a suffix to the label.
- A dynamic `if-from` is specialised into a residual `if-from` consisting of a residual condition and two residual labels constructed like above.
- Static labels and assertions are targets of static jumps, so we describe their treatment there.
- `return` is specialised to `return`. It is verified that all static variables that are not output variables have the value zero. If not, an error message is issued.
- An unconditional dynamic jump is specialised by combining its label with the static state to construct a new residual label, as described above for entry points. The specialised label is used to make a residual unconditional jump. If there is not already a specialised basic block for the specialised label, one is made.
- A dynamic `if-goto` is specialised into a residual `if-goto` consisting of a residual condition and two residual labels constructed like above. If there are not already specialised basic blocks for the specialised labels, these are made.
- An static unconditional jump is specialised by finding the basic block that has this label in its entry point. If the target basic block has an unconditional label as entry, the jump is unfolded by specialising the body of the target basic block and adding it to the body of the specialised basic block that contains the static jump and then specialising the jump of the target basic block. If the target basic block has a two-way assertion as entry, this will be static (by rules 8 and 9 of the binding-time analysis), so its condition will be evaluated and checked. If the assertion fails, an error is reported. Otherwise, the jump is unfolded in the same way a jump to an unconditional label is unfolded.

- A static `if-goto` is specialised by first evaluating the condition and selecting the label that corresponds to the result. It is then unfolded like an unconditional static or dynamic jump to this label, depending on whether the label is static or dynamic.

Specialisation starts by reading the values of the static input variables and setting the static state accordingly. Then the block with the `start` entry point is specialised with this state. This may trigger specialisation of more basic blocks with different static states. If this eventually terminates, there will be residual basic blocks for all residual jumps.

4.4 Making the residual program reversible

It is not hard to see that residual statements are reversible, but it is not clear that jumps and entry points are reversible. We will look at the requirements from section 4.1 in turn:

1. There must be exactly one entry point of the form `start`.

   There can be no jumps to `start`, so only one residual basic block can have the `start` entry point. So this property is preserved.

2. There must be exactly one jump of the form `return`.

   If the basic block that contains the `return` jump is specialised to two or more different static states, creating two or more residual basic blocks, there will be two or more `return` jumps in the residual program. So we need to remedy this.

3. Each named label must occur in exactly one entry point.

   When we make a residual jump to a specialised label, we check if a specialised basic block for this label already exists, so we will not produce multiple basic blocks with the same label. Also, if no specialised basic block exists for a residual jump, we will make one. Hence, each residual label will occur in exactly one entry point.

4. Each named label must occur in exactly one jump.

   This requirement forbids two situations: **No jumps to a label:** This may, actually happen: When we specialise a dynamic two-way assertion, we produce two residual labels even though we have seen only a residual jump to one of these. We may eventually see a jump to the other label, but there is no guarantee of this. So we may end up with a two-way assertion that has a jump to only one of its labels. **Two jumps to the same label:** This might occur if a basic block is specialised to two different static states but the updates to static variables make the static states at the end of these basic blocks identical. But since all static updates and swaps are reversible and a basic block is just a sequence of updates and swaps, this can not happen: Identical static end-states imply identical static start states.

Hence, we have two problems to fix: There may be several `return` jumps, and there may be a two-way assertion that has a jump to only one of its labels.

Multiple `return` jumps can occur only if the basic block containing the original `return` jump is specialised to several different static states. Since a basic block is reversible, this implies that there are several possible static states at the `return` jump itself, i.e., at the end of the program execution. So if we can ensure that there is only one possible static state at the end of program execution, we can avoid multiple `return` jumps.

All variables and array elements that are not part of the program output must, according to the Janus semantics, be 0 at the end of program execution. So for these, there is only one possible state at the end of execution. Output variables and arrays can, however, potentially have several possible values at the end program execution. If a static output variable or array can have several possible values, we get multiple specialised `return` jumps.

We avoid this by classifying all output variables that are modified anywhere in the program as dynamic, so no static output variables are ever updated. Static output variables will, hence, have the same value throughout execution.

It is common in partial evaluation to classify *all* outputs as dynamic, so even severe restrictions on static outputs is a relaxation compared to the usual case.

The issue of having a two-way entry with a jump to only one of its labels is not so easy to solve. The obvious solution would be to reduce the two-way assertion to an unconditional label, hence eliminating the label to which there is no jump. However, this will also eliminate the assertion, which may be required to preserve semantics (otherwise we might replace a failing execution by successful termination or nontermination). Another solution is to add a jump to the label that has none. This can be done in the following way: A residual basic block of the form `if` c `from` l_1 l_2`:` s`;` j where there is a jump only to l_1 is replaced by the two basic blocks l_1`:` `if true goto` l_3 l_2 and `if` c `from` l_3 l_2`:` s`;` j where l_3 is a new label. The case where there is a jump only to l_2 is handled in a symmetric way.

Introducing extra jumps may make residual programs slower than the original programs, so to avoid this we extend the Janus language with statements of the form `assert` c (with the obvious semantics). If there is a jump only to l_1, we replace the basic block `if` c `from` l_1 l_2`:` s`;` j by the basic block l_1`:` `assert` c`;` s`;` j. If there is a jump only to l_2, we replace
`if` c `from` l_1 l_2`:` s`;` j with l_2`:` `assert !`c`;` s`;` j.

The statement `assert` c reverses to itself and it is simple to specialise: It is classified as static if the condition is static, and a dynamic `assert` it is made into a residual `assert` by replacing the static subexpressions of c by constants.

We can eliminate unconditional jumps to unconditional labels by unfolding the jumps: The basic blocks e`:` s_1`;` `goto` l and l`:` s_2`;` j are combined to the single basic block e`:` s_1`;` s_2`;` j.

The program in figure 4(b) can be specialised to `a=11` to yield the residual program shown in figure 5. Note that some two-way assertions have been converted to one-way assertion statements and unconditional jumps to unconditional labels have been eliminated.

4.5 Partially static arrays

We specialise a array `a` of size n with dynamic elements and static indices into n integer variables named `a_`0 ... `a_`m where $m = n-1$. The l-value `a[`e`]` is specialised by evaluating the static expression e to the value i and returning the residual l-value `a_`i.

This technique is an instance of partially static data structures [17] and is well known from, e.g., C-mix as described in section 11.4.1 of [13]. It is useful for, among other things, specialising interpreters where a single array is used to hold the values of individual variables in the interpreted program. By splitting the array into scalar variables, the variables of the interpreted program become individual variables in the residual program obtained by specialising the interpreter to the interpreted program. For example, the arrays `ins[i]`, `stack[sz]` and `garbage[sz]` in the postfix interpreter in figure 2(c) will be partially static.

```
b  -> b prod with v ;

start:
assert 0==prod; prod += b;
assert !(prod<b); v += b; b += v; v -= b/2;
assert !(0==prod); prod += b;
assert !(prod<b); v += b; b += v; v -= b/2;
assert !(0==prod);
assert prod<b; v += b; b += v; v -= b/2;
assert !(0==prod); prod += b;
assert !(prod<b)
goto f_11_92636

if prod<b+b from f_11_92636 a_11_92636:
v += b/2; b -= v; v -= b
if odd(b) goto u_11_92636 a_11_92636

u_11_92636:
return
```

Figure 5: Specialised multiplication program

5 Recovering structured control

The residual program shown in figure 5 is not a Janus program, as it uses unstructured control where Janus uses structured control statements. In [22] it is shown that any reversible flowchart can be translated into a program that uses only the reversible control structures shown in section 2. The translation works by first numbering all labels from 1 to $n-1$. A flowchart program

in `->` *outs* `with` *others*`;` *blocks*

is translated into the structured program

in `->` *outs* `with` *others* `control;`
`from control==0 do` $S(blocks)$
`loop skip until control==`n`;`
`control -=` n

where S takes a set of basic blocks and returns a statement:

```
a -> b;

start: goto x

if b==0 from x y
a -= 1; b += 1
if a==0 goto z y

z: return
```

(a) Simple flowchart program

```
a -> b;

from b==0
  a -= 1; b += 1
loop skip
until a==0
```

(c) Natural structure

```
a -> b with control;

from control==0 do
  if control==0 then
    control+=1
  else
    if control==1 || control==2 then
      if control==1 then skip
      else control-=1 fi b==0;
      a -= 1; b += 1;
      if a==0 then control+=2
      else control+=1 fi control==3
    else
      control+=1
    fi control==2 || control==3
  fi control==1
loop skip
until control==4;
control-=4
```

(b) Structured with `control` variable

Figure 6: Two ways of structuring a flowchart program

$$
\begin{array}{ll}
S(\emptyset) & = \texttt{skip} \\
S(\{e\texttt{: } s\texttt{; } j\} \cup B) & = E\ s\texttt{;}\ C\ \texttt{else}\ S(B)\ \texttt{fi}\ J \\
& \quad \text{where} \quad (E,\, l) \;=\; S_E(e) \\
& \qquad\qquad\;\; (C,\, J) \;=\; S_J(j,\, l)
\end{array}
$$

$$
\begin{array}{ll}
S_E(\texttt{start}) & = (\texttt{if control==0 then},\ 0) \\
S_E(l) & = (\texttt{if control==}l\ \texttt{then},\ l) \\
S_E(\texttt{if}\ c\ \texttt{from}\ l_1\ l_2) & = (\texttt{if control==}l_1\ \texttt{||}\ \texttt{control==}l_2\ \texttt{then} \\
& \qquad \texttt{if control==}l_1\ \texttt{then skip} \\
& \qquad \texttt{else control +=}\ l_1\texttt{-}l_2\ \texttt{fi}\ c\texttt{;}\ , \\
& \quad l_1)
\end{array}
$$

$$
\begin{array}{ll}
S_J(\texttt{return},\, l) & = (\texttt{control +=}\ n\texttt{-}l,\ \texttt{control==}n) \\
S_J(\texttt{goto}\ l_1,\, l) & = (\texttt{control +=}\ l_1\texttt{-}l,\ \texttt{control==}l_2) \\
S_J(\texttt{if}\ c\ \texttt{goto}\ l_1\ l_2,\, l) & = (\texttt{if}\ c\ \texttt{then control +=}\ l_1\texttt{-}l \\
& \qquad \texttt{else control +=}\ l_2\texttt{-}l \\
& \qquad \texttt{fi control==}l_1\ , \\
& \quad \texttt{control==}l_1\ \texttt{||}\ \texttt{control==}l_2)
\end{array}
$$

As an example, the flowchart program in figure 6(a) is translated into the structured program in figure 6(b). This structure is, however, not natural and the addition of the extra `control` variable and the branching on this adds overhead. The flowchart program shown in figure 6(a) has a much simpler and more efficient structured equivalent shown in figure 6(c)

Ideally, we would like to have a translation that can find a structured program such that the sequences of variable updates and tests made by the structured program is the same as the sequence made by the flowchart program when both are executed with the same inputs. Intuitively, this means that no overhead is introduced by structuring the program.

There has (in the context of decompilation) been some work on recovering structured control from flowcharts [2, 21, 7, 8], but this has been done in a non-reversible setting. To our

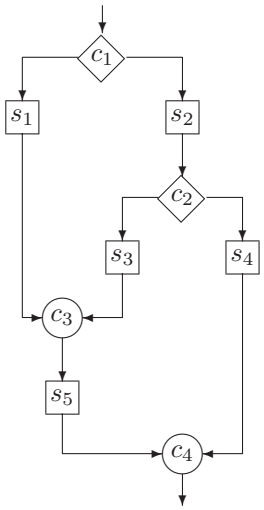


Figure 7: Unstructured flowchart

knowledge, there is no similar work on recovering reversible control.

It has been shown [2] that general control flow can not always be translated into structured control without adding extra variable updates, but again, this result is for general, irreversible control. Since reversible flowcharts are quite restricted compared to general flowcharts, it is not obvious that the result carries over: Unrestricted flowcharts that can not be structured might not be realisable as reversible flowcharts. We will, however, below show a similar result for reversible programs.

We first define evaluation equivalence of flowchart programs.

Definition 1 *Two reversible flowchart programs are evaluation equivalent if they both execute the same sequence of tests and state modifiers when run on the same inputs. We only consider terminating executions, so the sequences are finite. A state modifier is an update or swap. A test is a condition in an* `assert` *statement, a conditional jump or a two-way entry point. If a condition c evaluates to false, it is shown as* $\overline{c}$ *in the sequence. We consider* $\overline{c}$ *equivalent to* `!`c *and* $\overline{!c}$ *equivalent to c, so it is possible to swap the two branches of a conditional jump.*

A structured program is deemed to be evaluation equivalent to the flowchart program obtained by the translation shown in section 4.1 and, by transitivity, to all flowchart programs evaluation equivalent to this.

The proof in [2] that not all control flow can be made structured shows that a specific unstructured control-flow graph has no structured evaluation equivalent program (using a somewhat different notion of evaluation equivalence).

Consider the reversible flowchart in figure 7. The true exits/entries of conditional jumps and entry-points are to the left. There are three possible paths through this flowchart:

$$\begin{array}{ll} 1: & c_1,\ s_1,\ c_3,\ s_5,\ c_4 \\ 2: & \overline{c_1},\ s_2,\ c_2,\ s_3,\ \overline{c_3},\ s_5,\ c_4 \\ 3: & \overline{c_1},\ s_2,\ \overline{c_2},\ s_4,\ \overline{c_4} \end{array}$$

Let us see which structured programs can realise the above sequences.

Since all sequences start with c_1 or $\overline{c_1}$, c_1 must be the initial condition of an `if-then-else-fi` construct: An `assert` statement or the entry condition of a loop might have both outcomes, but would terminate execution after one of these. An `if-then-else-fi` construct must have a closing condition (assertion). The only condition (apart from c_1) that occurs in all the above sequences is c_4, so this must be it. So, the structured program must be of the form `if` c_1 `then` T `else` E `fi` c_4, though with the possibility that one or both of c_1 and c_4 is negated. We can without loss of generality assume that c_1 is not negated (we could swap the branches if it was), so E must be able to realise the rest of sequence 1. It ends in c_4, so the exit assertion must be c_4 (without negation). Sequence 2 and 3 start with $\overline{c_1}$, so they both go to the false branch. They, however, end with different values for the c_4 condition, which contradicts the assumption that they are in the same branch of the conditional.

So if we can find actual conditions and statements such that all three sequences can be realised, we have an example of a flowchart that has no structured evaluation equivalent using the control structures of Janus. If we use the following instances of the tests and conditions:

c_1	=	`a == 0`	s_1	=	`a += 1`
c_2	=	`a == 0`	s_2	=	`a -= 1`
c_3	=	`a == 1`	s_3	=	`a += 2`
c_4	=	`a < 2`	s_4	=	`a += 2`
			s_5	=	`a -= 1`

then sequence 1 is followed when `a` is 1 initially, sequence 2 is followed when `a` is 2 initially and sequence 3 is followed when `a` is 3 initially. Hence, we have a flowchart that is not evaluation equivalent to any structured program.

The above is just one example of a flowchart that has no evaluation equivalent structured program.

So we may sometimes need to introduce extra state modifiers or tests when re-structuring residual programs. But we want to avoid this whenever we can.

We use a restructuring methods based on recognising subsets of basic blocks that correspond to structured statements and then replace these subsets by basic blocks that use structured statements.

The following rules do this for 17 different patterns of basic blocks. We apply these until no rule applies to the remaining set of basic blocks.

1. If the set of basic blocks contains two blocks:

   (e: s_1; `goto` l) (l: s_2; j)

   combine these to the single basic block

   (e: s_1; s_2; j)

2. If the set of basic blocks contains four blocks:

   (e: s_1; `if` c_1 `goto` l_1 l_2) (l_1: s_2; `goto` l_3) (l_2: s_3; `goto` l_4) (`if` c_2 `from` l_3 l_4: s_4; j)

   combine these into the single basic block

   (e: s_1; `if` c_1 `then` s_2; `else` s_3; `fi` c_2; s_4; j)

3. If the set of basic blocks contains three blocks

   (e: s_1; `if` c_1 `goto` l_1 l_2) (l_1: s_2; `goto` l_3) (`if` c_2 `from` l_3 l_2: s_3; j)

   combine these into a single basic block

   (e: s_1; `if` c_1 `then` s_2; `else skip fi` c_2; s_3; j)

4. If the set of basic blocks contains three blocks
   (e: s_1; `if` c_1 `goto` l_1 l_2) (l_2: s_2; `goto` l_3) (`if` c_2 `from` l_1 l_3: s_3; j)
   combine these into a single basic block
   (e: s_1; `if` c_1 `then skip else` s_2; `fi` c_2; s_3; j)
5. If the set of basic blocks contains two blocks
   (e: s_1; `if` c_1 `goto` l_1 l_2) (`if` c_2 `from` l_1 l_2: s_2; j)
   combine these into a single basic block
   (e: s_1; `if` c_1 `then skip else skip fi` c_2; s_2; j)
6. If the set of basic blocks contains four blocks
   (e: s_1; `if` c_1 `goto` l_1 l_2) (l_1: s_2; `goto` l_3) (l_2: s_3; `goto` l_4) (`if` c_2 `from` l_4 l_3: s_4; j)
   combine these into a single basic block
   (e: s_1; `if` c_1 `then` s_2; `else` s_3; `fi !`c_2; s_4; j)
7. If the set of basic blocks contains three blocks
   (e: s_1; `if` c_1 `goto` l_1 l_2) (l_1: s_2; `goto` l_3) (`if` c_2 `from` l_2 l_3: s_3; j)
   combine these into a single basic block
   (e: s_1; `if` c_1 `then` s_2; `else skip fi !`c_2; s_3; j)
8. If the set of basic blocks contains three blocks
   (e: s_1; `if` c_1 `goto` l_1 l_2) (l_2: s_2; `goto` l_3) (`if` c_2 `from` l_3 l_1: s_3; j)
   combine these into a single basic block
   (e: s_1; `if` c_1 `then skip else` s_2; `fi !`c_2; s_3; j)
9. If the set of basic blocks contains two blocks
   (e: s_1; `if` c_1 `goto` l_1 l_2) (`if` c_2 `from` l_2 l_1: s_2; j)
   combine these into a single basic block
   (e: s_1; `if` c_1 `then skip else skip fi !`c_2; s_2; j)
10. If the set of basic blocks contains two blocks
    (`if` c_1 `from` l_1 l_2: s_1; `if` c_2 `goto` l_3 l_4) (l_4: s_2;`goto` l_2)
    combine these into a single basic block
    (l_1: `from` c_1 `do` s_1 `loop` s_2 `until` c_2; `goto` l_3)
11. If the set of basic blocks contains a block
    (`if` c_1 `from` l_1 l_2: s; `if` c_2 `goto` l_3 l_2)
    replace this by the basic block
    (l_1: `from` c_1 `do` s `loop skip until` c_2; `goto` l_3)
12. If the set of basic blocks contains two blocks
    (`if` c_1 `from` l_1 l_2: s_1; `if` c_2 `goto` l_4 l_3) (l_4: s_2;`goto` l_2)
    combine these into a single basic block
    (l_1: `from` c_1 `do` s_1 `loop` s_2 `until !`c_2; `goto` l_3)

13. If the set of basic blocks contains a block
    (`if` c_1 `from` l_1 l_2: s; `if` c_2 `goto` l_2 l_3)
    replace this by the basic block
    (l_1: `from` c_1 `do` s `loop skip until !`c_2; `goto` l_3)

14. If the set of basic blocks contains two blocks
    (`if` c_1 `from` l_2 l_1: s_1; `if` c_2 `goto` l_3 l_4) (l_4: s_2;`goto` l_2)
    combine these into a single basic block
    (l_1: `from !`c_1 `do` s_1 `loop` s_2 `until` c_2; `goto` l_3)

15. If the set of basic blocks contains a block
    (`if` c_1 `from` l_2 l_1: s; `if` c_2 `goto` l_3 l_2)
    replace this by the basic block
    (l_1: `from !`c_1 `do` s `loop skip until` c_2; `goto` l_3)

16. If the set of basic blocks contains two blocks
    (`if` c_1 `from` l_2 l_1: s_1; `if` c_2 `goto` l_4 l_3) (l_4: s_2;`goto` l_2)
    combine these into a single basic block
    (l_1: `from !`c_1 `do` s_1 `loop` s_2 `until !`c_2; `goto` l_3)

17. If the set of basic blocks contains a block
    (`if` c_1 `from` l_2 l_1: s; `if` c_2 `goto` l_2 l_3)
    replace this by the basic block
    (l_1: `from !`c_1 `do` s `loop skip until !`c_2; `goto` l_3)

The important rules are 1, 2 and 10, which are essentially inverses of the translation from section 4.1. The remaining rules are special cases where one branch of a conditional or part of a loop is empty or where labels in `if-goto` or `if-from` are swapped (so conditions need to be negated).

It is easy to see that these rules preserve (strong) evaluation equivalence. Since we negate some conditions, we need the equivalence of c and $\overline{\texttt{!}c}$ and of `!`c and $\bar{c}$ that we allowed in the definition of evaluation equivalence.

The reversibility requirement actually makes some things simpler than in the irreversible case, as we know that there can not be multiple jumps to the same label. In the irreversible case we would, for example, need a side condition to rule 1 saying that there are no other jumps to l. Also, since there are exactly two occurrences of each label, it is easy to verify that the rules do not overlap, so they can be applied in any order.

If application of the rules reduces the set of basic blocks to a single basic block (`start`: s; `return`), we have a single structured Janus statement s. If not, we can either declare failure to structure the program or add the extra `control` variable as described above. Currently, we admit failure and return a partially unstructured program, as our (extended) Janus compiler allows programs to mix structured and unstructured control. This is easy enough to change if we need to use an unextended Janus compiler.

As an example, figure 8 shows a restructured version of the residual program in figure 5. Also, if we translate any of the programs in figure 2 into flowchart form and restructure the result, we get the original structure back.

```
b  -> b prod with v ;

assert 0==prod; prod += b;
assert !(prod<b); v += b; b += v; v -= b/2;
assert !(0==prod); prod += b;
assert !(prod<b); v += b; b += v; v -= b/2;
assert !(0==prod);
assert prod<b; v += b; b += v; v -= b/2;
assert !(0==prod); prod += b;
assert !(prod<b);
from prod<b+b do
  v += b/2; b -= v; v -= b
loop
  skip
until odd(b)
```

Figure 8: Restructured specialised multiplication program

5.1 Replacing jumps by procedure calls

In [8], it is suggested that unstructured control that can not be reduced to structured control without adding variables can instead be replaced by mutually recursive procedure calls. The idea is that a tail call is very much like an unstructured `goto`. The method replaces every remaining basic block with a procedure and all remaining jumps by tail calls to these procedures. This works well because basic blocks in irreversible languages are headed by unconditional labels (that translate easily into procedure names) and both unconditional and conditional jumps can be translated into tail calls. In our reversible flowchart language, we have two issues not present in irreversible languages:

1. Basic blocks can be headed by two-way assertions.
2. In Janus, the branches of an `if-then-else-fi` are not in tail-call position, even if the whole conditional statement is. This is because the `fi`-condition needs to be tested after the chosen branch completes.

Two-way assertions are not difficult to handle: We just make two procedures that each test the condition (positively and negatively) and call a procedure for the common body. But we can not translate `if` c `goto` $l_1\, l_2$ into `if` c `then call` p_1 `else call` p_2 `fi` c' because there might be no suitable condition c', and even if there is, there is no obvious way to find it.

Hence, we believe that it is not workable to translate unstructured control flow into tail-recursive procedure calls.

6 Experiments

We have implemented (in Standard ML) a partial evaluator using the methods described in this paper.

We have used the partial evaluator to specialise a few Janus programs. For each of these, the table in figure 9 shows the number of non-blank lines in the original Janus program, the flowchart form of the program and the residual program.

`fib` is the program from figure 2(a). It is specialised to `n`=10, so the residual program is run without inputs. `encrypt` is a simple encryption program. The program is specialised to the key (which is used both for encryption and decryption). During specialisation, a loop is unrolled, so the residual program is quite large. `postfix` is the program from figure 2(c)

Program	Source lines	Flowchart lines	Residual lines
fib	9	12	33
multiply	23	30	28
encrypt	15	23	171
postfix	46	74	16
control	17	39	6
dfa	26	29	48

Program	Original steps	Residual steps
fib	147	95
multiply	160	108
encrypt	442	354
postfix	998	70
control	558	145
dfa	510	216

Figure 9: Size and speed

with the procedure calls manually unfolded (as the specialiser doesn't handle procedure calls yet). `postfix` uses a partially static array for the input values, so the residual program has individual variables for these. The stacks are also partially static, so the stack elements become individual variables. The interpreter is specialised with respect to a postfix expression with two inputs and five operations. `control` is the program from figure 6(b). This is specialised with no static inputs, but the `control` variable is static. The residual program is identical to the program in figure 6(c). The difference in running time show the overhead of structuring a program using a `control` variable. `dfa` is an interpreter for reversible DFAs. It is specialised to a DFA that recognises bit strings that are divisible by 3. The restructurer was not able to restructure the residual program, so the numbers for the residual program are for the flowchart form.

We compile Janus programs to MIPS code that is run on the simulator MARS [20]. Figure 9 show the instruction counts of the original and residual programs. As usual with partial evaluation, the most dramatic speedups are found when specialising interpreters.

7 Conclusion and future work

We have made a partial evaluator for the reversible language Janus with the exception of procedure calls. It is to our knowledge the first partial evaluator for a deterministic reversible programming language.[1]

Polyvariant program-point specialisation can be applied to Janus, but the reversibility requirement added some complications, which we have solved.

The residual programs produced by the specialiser use unstructured control flow where Janus uses structured control flow, so we have devised a method to restructure residual programs from flowchart form to using the reversible control structures of Janus. This is not always possible to do without introducing extra variables. Currently, the specialiser will return partially unstructured programs in such cases, but it is easy to add an extra pass to

[1] Pure logic languages can be considered reversible and partial evaluators for such exist [15].

structure these using a `control` variable. Since this adds overhead and our Janus compiler can handle unstructured programs, we have not done so.

We have in this paper not handled procedure calls. It is, as such, not difficult to specialise procedure calls: Procedures must, like the program, have a single `return` jump. For the program, we ensured this by disallowing modification of static output variables, so there is only one possible static state at program end (all non-output variables are zero at program end). There is no requirement that variables are zero at the end of a procedure, so to ensure that there is only one possible static state at procedure end, the equivalent solution is to disallow modification of `all` static variables inside a procedure. This means that a procedure call can not change static state, which makes specialisation easy. The restriction against modifying static state makes this of limited use, though. We will look at better ways of handling procedure calls in a future paper.

References

[1] Samson Abramsky. A structural approach to reversible computation. Manuscript, Oxford University Computing Laboratory, 2001.

[2] Edward A. Ashcroft and Z Manna. The translation of "go to" programs to "while" programs. Technical report, Stanford University, Stanford, CA, USA, 1971.

[3] Charles H. Bennett. Time/space trade-offs for reversible computation. *SIAM Journal on Computing*, 18(4):766–776, 1989.

[4] Harry Buhrman, John Tromp, and Paul Vitányi. Time and space bounds for reversible simulation. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming. Proceedings*, LNCS 2076, pages 1017–1027. Springer-Verlag, 2001.

[5] Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[6] Niels H. Christensen, Robert Glück, and Søren Laursen. Binding-time analysis in partial evaluation: one size does not fit all. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Perspectives of System Informatics. Proceedings*, LNCS 1755, pages 80–92. Springer-Verlag, 2000.

[7] Cristina Cifuentes. Reverse compilation techniques. PhD thesis, Queensland University of Technology, http://www.itee.uq.edu.au/~cristina/dcc.html#thesis, 1994.

[8] Cristina Cifuentes. Structuring decompiled graphs. In *Proceedings of the International Conference on Compiler Construction*, pages 91–105. Springer Verlag, 1996.

[9] Richard P. Feynman. *Feynman Lectures on Computation*, chapter 5 Reversible computation and the thermodynamics of computing, pages 137–184. Addison-Wesley, 1996.

[10] Arne J. Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of offline partial evaluation. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, LNCS 1181, pages 273–284. Springer-Verlag, 1996.

[11] John Hatcliff, Torben Æ Mogensen, and Peter Thiemann, editors. *Partial Evaluation. Practice and Theory*. LNCS 1706. Springer-Verlag, Berlin, Heidelberg, New York, 1999.

[12] Neil D. Jones and Arne Glenstrup. Program generation, termination, and binding-time analysis. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering. Proceedings*, LNCS 2487, pages 1–31. Springer-Verlag, 2002.

[13] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[14] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[15] Michael Leuschel. Logic program specialisation. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, pages 155–188, London, UK, 1999. Springer-Verlag.

[16] C. Lutz. Janus: a time-reversible language. A letter to Landauer. http://www.cise.uf1.edu/~mpf/rc/janus.html, 1986.

[17] Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[18] Kenichi Morita, Akihiro Shirasaki, and Yoshifumi Gono. A 1-tape 2-symbol reversible turing machine. *IEICE Transactions*, E72(3):223–228, 1989.

[19] Tommaso Toffoli. Reversible computing. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 632–644. Springer-Verlag, 1980.

[20] Kenneth Vollmar and Pete Sanderson. MARS: An education-oriented MIPS assembly language simulator. *ACM SIGCSE Bulletin*, 38(1):239–243, 2006.

[21] M. H. Williams. Generating structured flow diagrams: the nature of unstructuredness. *Computer Journal*, 20(1):45–50, 1977.

[22] Tetsuo Yokoyama, Holger Boch Axelsen, and Robert Glück. Reversible flowchart languages and the structured reversible program theorem. In Aceto L. et al., editor, *Automata, Languages and Programming (ICALP)*, LNCS 5126, pages 258–270. Springer-Verlag, 2008.

[23] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a reversible programming language. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 43–54, New York, NY, USA, 2008. ACM.

[24] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 144–153, New York, NY, USA, 2007. ACM.

On Determination of Backward Graph Transformation (Ongoing Work Report)

Zhenjiang Hu

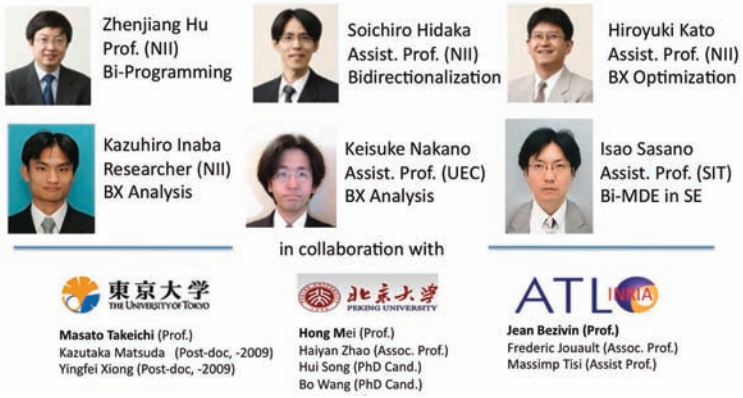
National Institute of Informatics

Joint Work with BiG Team Members

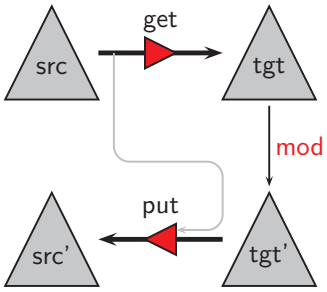
January 13, 2011

BiG: NII Grand Challenge Project (2008-)

A Bidirectional Graph (Model) Transformation Framework for Evolutional Software Development



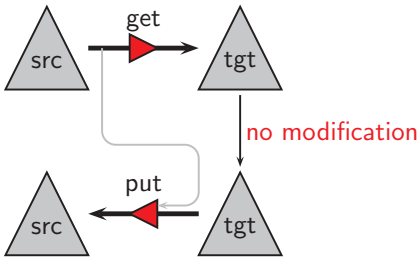
Bidirectional Transformation



It consists of a pair of computation forward and backward.

Stability

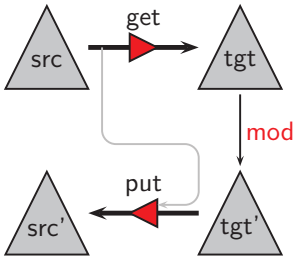
No change on the target implies no change on the source.



$$\mathrm{put}(\mathrm{get}(s), s) = s$$

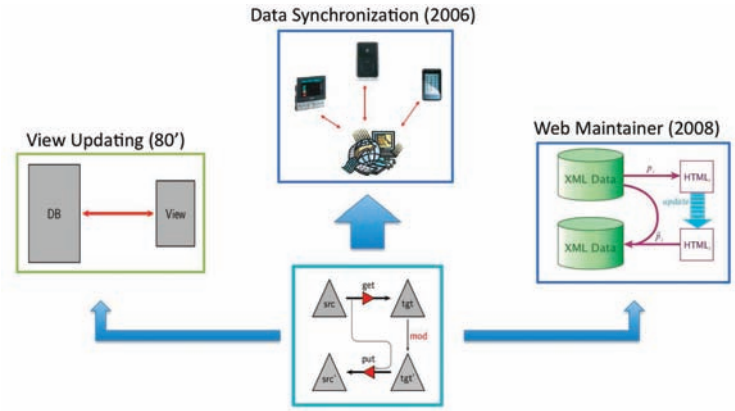
Reflectivity

Permitted changes on the target should be reflected to the source.



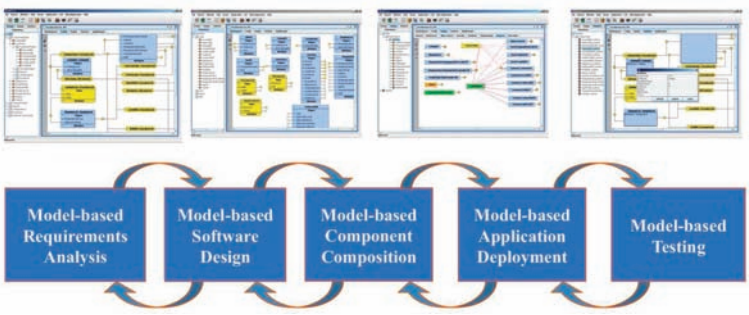
$$\mathrm{get}(\mathrm{put}(t', s)) = t'$$

Applications



Challenge: from Trees to Graphs

We aim at a language for bidirectional model-driven software developemnt (or roundtrip software development).

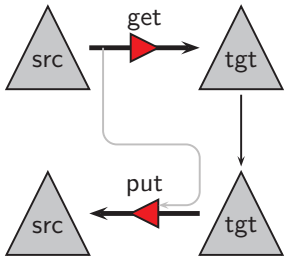


Two Approaches to Bidirectional Programming

- Design a domain-specific combinator library
  [Lens: POPL'95, POPL'98, ICFP'08, ICFP'10]
  - Primitive bidirectional computing functions
  - Combinators to compose bidirectional computations
- Automatic bidirectionalization of
  - A core ATL [Xiong+: ASE'07]
  - A first-order FPL [Matsuda+: ICFP'07]
  - A graph query language UnQL [Hidaka+: ICFP'10]

ASSUMPTION:
Each program has fixed computation behavior for *get* and *put*.

Not a practical assumption!



Since *get* is genereally non-injective, many suitable *put*s correspond to one *get*, each being useful in one context.

Inherited Ambiguity in Backward Transformation

Consider the relation input database *EMP*:

$$EMP = \{ \begin{array}{l} No \\ Name \\ Location \\ BBTeam \end{array} \}$$

No	Name	Location	BBTeam
1	Tanaka	Tokyo	Yes
2	Kato	Tohoku	Yes
3	Sato	Tokyo	No

Consider a forward transformation defined by

Select $*$
From EMP
Where Location $=$ "*Tokyo*"

which produces the view:

No	Name	Location	BBTeam
1	Tanaka	Tokyo	Yes
3	Sato	Tokyo	No

What is a suitable backward transformation if we delete employee #1 in the view?

$\Rightarrow$ delete employee #1

$\Rightarrow$ move employee #1 from Tokyo to Kyoto

Consider another forward transformation defined by

Select $*$
From EMP
Where BBTeam $=$ *Yes*

which produces the view:

No	Name	Location	BBTeam
1	Tanaka	Tokyo	Yes
3	Kato	Tohoku	Yes

What is a suitable backward transformation if we delete employee #1 in the view?

$\Rightarrow$ delete employee #1?

$\Rightarrow$ change the BBTeam of employee #1 to No?

Rather than fixing a backward transformation, is there a way to determine a suitable backward transformation at view definition time?

From Simon Peyton Jones:

If you find a new problem, there may have been some solutions (or ideas) given by DB people.

Keller's Dialog Approach

- Mapping a query (forward transformation) to the algebra consisting of selection, projection and join (Stanard).
- Enumerating finite backward transformations for selection, projection and join (PODS'85).
- Dialoging with view designers to choose a backward transformation statically by traversing over the syntactic tree of an algebra expression (VLDB'86).

Our Approach

Design a graph algebra such that

- it is powerful enough to be used as the base algebra for graph querying (transforming);
- each construct has enumerable choices for backward transformation.

Can we use the existing UnCAL as our graph algebra?

$$
\begin{array}{rcll}
e & ::= & \{\} \mid \{l : e\} \mid e \cup e \mid \&x := e \mid \&y \mid () & \\
& \mid & e \oplus e \mid e \mathbin{@} e \mid \mathbf{cycle}(e) & \{ \text{ constructor } \} \\
& \mid & \$g & \{ \text{ graph variable } \} \\
& \mid & \mathbf{if}\ l = l\ \mathbf{then}\ e\ \mathbf{else}\ e & \{ \text{ conditional } \} \\
& \mid & \mathbf{rec}(\lambda(\$l, \$g).e)(e) & \{ \text{ structural recursion application } \}
\end{array}
$$

We can easily enumerate suitable backward transformation for all constructs except for structural recursion [ICFP 2010]

For instance, **if** has several possiblities for backward transformation is some modification on the view is applied.

What is the problem with structural recursion?

Structural Recursion: Manipulating Graphs

Structural Recursion:

$$
\begin{array}{lcl}
f\ (\{\}) & = & \{\} \\
f\ (\{l : g\}) & = & e[l, g] \mathbin{@} f(g) \\
f\ (g_1 \cup g_2) & = & f(g_1) \cup f(g_2)
\end{array}
$$

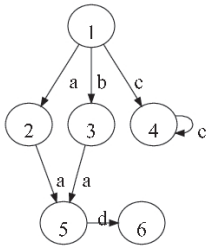
Or written as:

$$\mathbf{sfun}\ \ f\ (\{l : g\}) = e[l, g] \mathbin{@} f(g)$$

Or written as:

$$f = \mathbf{rec}(\lambda(l, g).e[l, g])$$

An Example



$$\textbf{sfun}\ \ a2d\_xc\ (\{l : g\}) \quad = \quad \begin{array}{l}\textbf{if}\ l = a\ \textbf{then}\ \{d : a2d\_xc(g)\}\\ \textbf{else if}\ l = c\ \textbf{then}\ a2d\_xc(g)\\ \textbf{else}\ \{l : a2d\_xc(g)\}\end{array}$$

i.e.,

$$a2d\_xc \quad = \quad \textbf{rec}(\lambda(l, g).\ \begin{array}{l}\textbf{if}\ l = a\ \textbf{then}\ \{d : \&\}\\ \textbf{else if}\ l = c\ \textbf{then}\ \&\\ \textbf{else}\ \{l : \&\})\end{array}$$

A structural recursion could be quite complicated

- Possible duplicated use of g:

$$\textbf{sfun}\ \ f\ (\{l : g\}) = e[l, g]\ @\ f(g)$$

$$f = \textbf{rec}(\lambda(l, g).e[l, g])$$

$\Rightarrow$ Backward transformation may introduce side-effect

- Nested recursion:

$$f = \textbf{rec}(\lambda(l, g).\ \cdots \textbf{rec}(\lambda(l', g').\ \cdots)\cdots)$$

$\Rightarrow$ Backward transformation needs to care about context

- Nested traversals over multiple graphs::

$$\underline{\mathbf{rec}(\lambda(l,g).\cdots\underline{\mathbf{rec}(\lambda(l,g).\cdots)(db_1)}\cdots)(db_2)}$$

$\Rightarrow$ Backward transformation needs to consider data joining.

Refining UnCAL

Is it possible to define a graph algebra which only considers the following simple structural recursion?

$$\mathbf{sfun}\ f\ (\{l:g\}) = e[l] \mathbin{@} f(g)$$

$$f = \mathbf{rec}(\lambda(l,g).e[l])$$

The backward behavior of this simple structural recursion is uniquely determined by that of $e[l]$.

Assumption: the unit for insertion/deletion on the view should match with $e[l]$.

Simplifying Structural Recursions

- Possible duplicated use of g:

$$\mathbf{sfun}\ f\ (\{l:g\}) = e[l,g] \mathbin{@} f(g)$$

$$f = \mathbf{rec}(\lambda(l,g).e[l,g])$$

$$\Uparrow$$

tupling f with id

- Nested recursion:

$$f = \mathbf{rec}(\lambda(l, g).\ \cdots \mathbf{rec}(\lambda(l', g').\ \cdots)\cdots)$$

$$\Uparrow$$

Unnesting / lambda lifting

$$\mu^f(X) = \{(v, w) \mid v \leftarrow X,\ w \leftarrow f(v)\}$$

- Nested traversals over multiple graphs:

$$\mathbf{rec}(\lambda(l, g).\ \cdots \underline{\mathbf{rec}(\lambda(l, g).\ \cdots)(db_1)}\cdots)(db_2)$$

$$\Uparrow$$

joining variables

$$X \bowtie Y = \{(v, w) \mid x \leftarrow X,\ w \leftarrow Y\}$$

Main Results

Theorem (Expressiveness)

Any UnQL expression can be compiled to the following graph algebra:

$$\begin{array}{rcl} t & ::= & \textit{any graph constructor} \\ & | & \$g \\ & | & \mathbf{if}\ l = l\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \\ & | & f\ t \\ & | & t_1 \bowtie t_2 \\ & | & \mu^f(t) \end{array}$$

where f is a simple structural recursion.

Theorem (Enumerability of Backward Transformations)

Suitable backward transformations for each construct of the following graph algebra is enumerable.

$$
\begin{array}{rcl}
t & ::= & \textit{any graph constructor} \\
& | & \$g \\
& | & \textbf{if } l = l \textbf{ then } t_1 \textbf{ else } t_2 \\
& | & f\ t \\
& | & t_1 \bowtie t_2 \\
& | & \mu^f(t)
\end{array}
$$

where f is a simple structural recursion.

Conclusion

- We show that Keller's dialogue approach to determining backward transformations can be generalized from relational data to graph data, by defining a new graph algebra:
  - sufficient expressive power
  - enumerable backward transformations

The new algebra can be considered as a combination of structural recursion with a general tree algebra.

More Information about BiG

The project page contains all published papers, system demo, and the source codes of the GRoundTram system  .

Author Index

After Work
Dinner at Asakusa