Datalogisk Institut
KØBENHAVNS UNIVERSITET




# Proceedings of the 2nd DIKU-IST Joint Workshop on Foundations of Software

Robert Glück and Zhenjiang Hu (Eds.)

**Technical Report no. 06/07**
**ISSN: 0107-8283**

**Dept. of Computer Science**
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

# Preface

These proceedings contain the contributions presented at the *2nd DIKU-IST Joint Workshop on Foundations of Software* held at Shonan Village Center, Kanagawa, Japan, April 21-22, 2006. The workshop featured talks and discussions on parallel and mobile computation, automatic software production, model checking techniques, and programming language design and implementation.

After the success of the first workshop, which took place near Copenhagen, the second workshop aimed to provide a forum for exchanging the latest research ideas 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 at Tokyo University and the Faculty of Science at the University of Copenhagen entered a *Five-Year Academic and Student Exchange Agreement* during the State Visit of Queen Margrethe II to Japan. The objective is to create a closer collaboration between the two groups of researchers, postdocs, and graduate students.

Software has become a decisive factor for commercial success in many areas of modern technology and business development. Despite tremendous progress in hardware, the production of software is manual, error prone, and costly. The exploding demand for software has led to a dramatic worldwide undersupply of skilled programmers, outsourcing, and to low software quality, which is evident to anyone today who uses software. The workshop invited contributions on all topics related to the foundations of software.

The workshop had about 45 participants from Japan and Denmark. The organizers would like to thank all speakers, participants, and the local organizers for making this meeting both a successful and enjoyable event. Special thanks to Masato Takeichi, Dean of IST, and to Stig Skelboe, Department Chairman of DIKU for their tireless support of this collaboration.

Copenhagen and Tokyo, June 2006

Robert Glück
Zhenjiang Hu

# 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 (Co-Chair) |
| | Masami Hagiya |
| | Fritz Henglein |
| | Shinichi Honiden |
| | Haruo Hosoya |
| | Zhenjiang Hu (Co-Chair) |
| | Neil Jones |
| | Julia Lawall |
| | Stig Skelboe |
| | Akihiko Takano |
| | Masato Takeichi |
| | Akinori Yonezawa |
| Local arrangements: | Carl Christian Frederiksen |
| | Johan Gade |
| | Masae Ito |
| | Keisuke Nakano |
| | Ruriko Watanabe |
| | Tetsuo Yokoyama |
| Proceedings: | Morten Fjord-Larsen |

## Sponsoring Institutions

This workshop was supported by the Superrobust Computation Project of the 21st Century COE Program "Information Science and Technology Strategic Core" of the University of Tokyo, and the Danish Research Agency, Forskningsrådet for Natur og Univers (FNU). We gratefully acknowledge the support of the Department of Computer Science, University of Copenhagen.

# Table of Contents


## Program Analysis

The Flow of Data and the Complexity of Algorithms . . . . . 1
*Neil D. Jones and Lars Kristiansen*

A Common Framework for Several Path-Summarizing Analyses . . . . . 9
*James Avery*

Abstraction of Programs Manipulating Pointers Using Modal Logics . . . . . 20
*Yoshinori Tanabe, Yoshifumi Yuasa, Toshifusa Sekizawa, and Koichi Takahashi*

Inferring Bidirectionality of Programs . . . . . 28
*Johan Gade, Zhenjiang Hu, Keisuke Nakano, and Kazutaka Matsuda*

## Parallel and Mobile Computation

Non-Equilibrium Green's Function Calculation for Electron Transport . . . 35
*Stig Skelboe*

Statistical State Estimation . . . . . 36
*Søren Hauberg*

Robust Routing Algorithm in Unstable Mobile Network . . . . . 48
*Satoshi Kataoka*

## Web and Agent Systems

Logiweb . . . . . 49
*Klaus Grue*

Towards Consistent Protocol-Based Agent Interactions . . . . . 50
*Jose Ghislain Quenum*

## Model Checking Techniques

Model Checking Networked Applications . . . . . 60
*Cyrille Artho, Armin Biere, Boris Zweimüller, Pierre-Loïc Garoche, Christian Sommer, and Shinichi Honiden*

Interprocedural Program Analysis for Java based on Weighted Pushdown Model Checking . . . . . 65
*Mizuhito Ogawa and Xin Li*

## Programming Techniques

Generic Sorting Multiset Discriminators . . . . . 71
*Fritz Henglein*

Hash-free Binary Decision Diagrams . . . . . 78
*Morten Ib Nielsen*

Aspect Mining Using a Structural Concern Model . . . . . 90
*Jan Hannemann and Hidehiko Masuhara*

## Program Transformation

A Method for Inversion of a Higher-Order Programming Language . . . . . 96
*Jesper Andersen*

A Parallelization Tool for Tree Reductions . . . . . 110
*Kiminori Matsuzaki and Noriyuki Ohkawa*

Weak Inversion for Parallelization . . . . . 116
*Akimasa Morihata and Zhenjiang Hu*

## Language Design and Implementation

Writing Practical Memory Management Code with a Strictly Typed Assembly Language . . . . . 123
*Toshiyuki Maeda and Akinori Yonezawa*

Static Types for Scripting . . . . . 137
*Torben Mogensen*

Waitomo: Web-Programming with Objects and Interfaces . . . . . 138
*Peter Thiemann*

## Formal Languages

Towards Deterministic Top-Down Parsing of LR Languages . . . . . 147
*Morten Fjord-Larsen*

The Density of Words in Formal Languages . . . . . 153
*Jakob Grue Simonsen*

## XML

Shuffle Expressions for Static XML Type-Checking . . . . . 154
*Tadahiro Suda, Haruo Hosoya, and Akihiko Tozawa*

XML Transformation Language Based on Monadic Second Order Logic . . 164
*Kazuhiro Inaba and Haruo Hosoya*

**Author Index** . . . . . 173



Shonan Village Center, Kanagawa, Japan (April 2006)

# THE FLOW OF DATA AND THE COMPLEXITY OF ALGORITHMS

Neil D. Jones (joint with Lars Kristiansen, Oslo)

Related work:

- Size-change termination (Lee, NDJ, Ben-Amram, Avery,...);
- Recognising ptime (C.C. Frederiksen + NDJ);
- Data Flow and Algorithm Complexity (L. Kristiansen+NDJ at CIE 2005).
- The Implicit Complexity community (many researchers).

A dream: a "grand unified theory" encompassing and simplifying the above

## SYNTAX: USUAL IMPERATIVE LANGUAGE

Expressions and Commands have forms given by the grammar

$$\begin{array}{rrl} \texttt{X} \in \textbf{Variable} & ::= & \texttt{X}_1 \mid \texttt{X}_2 \mid \texttt{X}_3 \mid \ldots \texttt{X}_n \\ \texttt{e} \in \textbf{Expression} & ::= & \texttt{X}_i \mid \texttt{e + e} \mid \texttt{e * e} \mid \texttt{ e otheroperation e} \\ \\ \texttt{C} \in \textbf{Command} & ::= & \texttt{skip} \mid \texttt{X}_i\texttt{:=e} \mid \texttt{C}_1\texttt{;C}_2 \mid \texttt{loop X \{C\}} \mid \texttt{if e then C else C} \end{array}$$

Convention:

- teletype font for syntactic objects (commands, expressions, variables $\texttt{X}_1, \texttt{X}_2, \texttt{X}_3, \ldots$), and
- *mathematical fonts* for semantic objects, e.g. the value $v$ currently assigned to a variable.
- The soundness of analysis result ($\vdash$ implies $\models$) holds for all programs (omit `while,...` in slides for brevity);



## SEMANTICS

Variable $\texttt{X}_i$ holds a natural number $x_i$ (possibly 0).

Command `loop X {C}` executes the command `C` for $m$ times in a row, where $m$ = initial value of `X`.

The command execution relation:

$$[\![\texttt{C}]\!](x_1, \ldots, x_n \rightsquigarrow x'_1, \ldots, x'_n)$$

variables $\texttt{X}_1, \ldots, \texttt{X}_n$ hold numbers $x_1, \ldots, x_n$ when the execution starts, implies
variables $\texttt{X}_1, \ldots, \texttt{X}_n$ hold numbers $x'_1, \ldots, x'_n$ when execution terminates.

Similar: the evaluation expression relation:

$$[\![\texttt{e}]\!](x_1, \ldots, x_n \rightsquigarrow v)$$

iff evaluation of `e`, when $\texttt{X}_1, \ldots, \texttt{X}_n$ as above, yields numeric value $v$.

For short: let $\vec{x}_n$ stand for $x_1, \ldots, x_n$, so we can write:

$$[\![\texttt{e}]\!](\vec{x}_n \rightsquigarrow v) \text{ or } [\![\texttt{C}]\!](\vec{x}_n \rightsquigarrow \vec{x}'_n)$$

## TO TRACE: POLYNOMIAL-BOUNDED VALUE GROWTH

Goal, given command C and variables $\mathtt{X}_i, \mathtt{X}_j$: discover polynomially-bounded data-flow relations between

- the initial value $x_i$ of $\mathtt{X}_i$ and the final value $x'_j$ of $\mathtt{X}_j$
- that hold whenever $[\![\mathtt{C}]\!](\vec{x}_n \rightsquigarrow \vec{x}'_n)$.

Some examples

- `loop Z { X := X+X }`: BAD $2^z = x'$
- `loop Z { X := X*Y }`: BAD $xy^z = x'$
- `loop Z { X := X+Y }`: GOOD $x + z * y = x'$
- `loop Z { X := X+Y; Y := X }`: BAD (why?...2 loops double $x$)

Our approach:

- Give up if non-polynomial growth
- Carefully trace data flow, e.g., copying versus adding versus multiplying
- Focus on self-dependence around loops

— 4 —

## THE PHENOMENON: VARIABLE VALUE GROWTH

Goal, given command C and variables $\mathtt{X}_i$ and $\mathtt{X}_j$, we wish to discover polynomially-bounded data-flow relations between

- the initial value $x_i$ of $\mathtt{X}_i$ and
- the final value $x'_j$ of $\mathtt{X}_j$
- that hold whenever $[\![\mathtt{C}]\!](\vec{x}_n \rightsquigarrow \vec{x}'_n)$.

For example, command `X := X*X` has bound $x^2 \geq x'$.

Concise matrix form (without exact polynomials):

$$\mathtt{loop}\,\mathtt{X}_3\,\{\mathtt{X}_1 := \mathtt{X}_1 + \mathtt{X}_2\} : \begin{pmatrix} m & \bot & \bot \\ p & m & \bot \\ p & \bot & m \end{pmatrix}$$

Net effect: $\mathtt{X}_1 := \mathtt{X}_1 + \mathtt{X}_3 * \mathtt{X}_2$

- $m$-variable: "max"
- $w$-variable: "weak polynomial"
- $p$-variable: "polynomial"

— 5 —

## $MWP$-BOUND

An $mwp$-bound (denoted $W, V, U, \ldots$) is a number-theoretic expression

$$\max(\vec{x}, q(\vec{y})) + p(\vec{z})$$

where $\vec{x}$, $\vec{y}$, and $\vec{z}$ are variables lists and $q$ and $p$ are monotone polynomials.

- $p$-variable: in $\vec{z}$ "polynomial"
- $w$-variable: in $\vec{y}$ (but not in $\vec{z}$) "weak polynomial"
- $m$-variable: in $\vec{x}$ (but not in $\vec{y}$ or $\vec{z}$) "max"

An example of a set of bounds:

$[\![\mathtt{loop}\,\mathtt{X}_3\,\{\mathtt{X}_1 := \mathtt{X}_1 + \mathtt{X}_2\}]\!](x_1, x_2, x_3 \rightsquigarrow x'_1, x'_2, x'_3) \Rightarrow W_1 \geq x'_1 \wedge W_2 \geq x'_2 \wedge W_3 \geq x$

where

$$\begin{aligned} W_1(x_1;;x_2,x_3) &= x_1 + x_2 \cdot x_3 \\ W_2(x_2;;) &= x_2 \\ W_3(x_3;;) &= x_3 \end{aligned}$$

— 6 —

## THE INFORMAL MEANING OF $MWP$-DESCRIPTIONS

Consider matrix entry $M_{ij}$.

- $M_{ij} = \bot$: source variable $\mathtt{X}_i$ is not used to compute target variable $\mathtt{X}_j$
- $M_{ij} = m$: source variable $\mathtt{X}_i$ might be copied to target variable $\mathtt{X}_j$.
  Example: $\mathtt{X}_j := \mathtt{X}_i$.
- $M_{ij} = w$: a polynomial function of source $\mathtt{X}_i$ might be copied to target $\mathtt{X}_j$. Here, $\mathtt{X}_j$ is not self-dependent.
  Example: $\mathtt{X}_j := \mathtt{X}_i * \mathtt{X}_i$.
- $M_{ij} = p$: a polynomial function of source $\mathtt{X}_i$ might be added to the target $\mathtt{X}_j$'s original value. Here, $\mathtt{X}_j$ may be self-dependent.
  Example: $\mathtt{X}_j := \mathtt{X}_j + (\mathtt{X}_i * \mathtt{X}_i)$.



## TRUE DECRIPTIONS BY VECTORS AND MATRICES

Definition Let $V = (\alpha_1, \alpha_2, \cdots, \alpha_n)$. Statement $\mathtt{e} : V$ is true, written $\models \mathtt{e} : V$, iff there exists an $mwp$-bound $W$ such that

1. $[\![\mathtt{e}]\!](\vec{x}_n \rightsquigarrow v)$ implies $W \geq v$
2. $\alpha_i = W^{(x_i)}$, for $i \in \{1, \ldots, n\}$.

Definition Let $M$ be an $n \times n$ matrix. Statement $\mathtt{C} : M$ is true, written $\models \mathtt{C} : M$, iff there exist $mwp$-bounds $W_1, \ldots, W_n$ such that

1. $[\![\mathtt{C}]\!](\vec{x}_n \rightsquigarrow \vec{x}'_n) \Rightarrow W_1 \geq x'_1 \wedge \ldots \wedge W_n \geq x'_n$
2. $M_{ij} = {W_j}^{(x_i)}$, for $i, j \in \{1, \ldots, n\}$.

Here the $W^{(x_i)}$'s abstract the polynomials out of the $W$ bounds:

$$W^{(x_i)} = \begin{cases} m & \text{if } x_i \text{ is an } m\text{-variable of } W \quad (x_i \text{ is in } \vec{x}) \\ w & \text{if } x_i \text{ is an } w\text{-variable of } W \quad (x_i \text{ is in } \vec{y}) \\ p & \text{if } x_i \text{ is an } p\text{-variable of } W \quad (x_i \text{ is in } \vec{z}) \\ \bot & \text{else (if } x_i \text{ is not in } W) \end{cases}$$



## VECTOR ALGEBRA TO MANIPULATE $MWP$-BOUNDS

Scalars.

- Elements: $\{\bot, m, w, p\}$ with order $\bot < m < w < p$.
- Operations:
  $+ =$ LUB (least upper bound),
  $\times = +$ except $x \times \bot = \bot \times x = \bot$.

Vectors. $n \times 1$, e.g. $V = (\alpha_1, \ldots, \alpha_n)$. Componentwise $\oplus$.

Scalar product:

$$\alpha V = (\alpha \times \alpha_1, \ldots, \alpha \times \alpha_n)$$

Notation. identify the vector $V$ with the set of all of its non-$\bot$ entries. E.g., the set

$$\{{}^{m}_{1}, {}^{m}_{3}, {}^{p}_{4}\}$$

is identified with the column vector

$$\begin{pmatrix} m \\ \bot \\ m \\ p \end{pmatrix}$$

## A CALCULUS TO PROVE EXPRESSION VALUE BOUNDS

$$(E1)\quad \vdash \mathtt{X}_i : \{^{m}_{i}\}$$

$$(E2)\quad \vdash \mathtt{e} : \{^{w}_{i} \mid i \in \text{variables}(\mathtt{e})\}$$

$$(E3)\quad \frac{\vdash \mathtt{e}_1 : V_1 \qquad \vdash \mathtt{e}_2 : V_2}{\vdash \mathtt{e}_1{+}\mathtt{e}_2 : pV_1 \oplus V_2}$$

$$(E4)\quad \frac{\vdash \mathtt{e}_1 : V_1 \qquad \vdash \mathtt{e}_2 : V_2}{\vdash \mathtt{e}_1{+}\mathtt{e}_2 : V_1 \oplus pV_2}$$

**Rule (E1):** we can label a single-variable expression by $m$.
**Rule (E2):** we can label all the variables in an expression by $w$'s.
**Rules $(E3), (E4)$:** An expression such as $\mathtt{X}_1{+}\mathtt{X}_2$ has several correct vectors:

$$\vdash \mathtt{X}_1{+}\mathtt{X}_2 : \begin{pmatrix} p \\ m \end{pmatrix} \quad \text{or} \quad \vdash \mathtt{X}_1{+}\mathtt{X}_2 : \begin{pmatrix} m \\ p \end{pmatrix} \quad \text{or} \quad \vdash \mathtt{X}_1{+}\mathtt{X}_2 : \begin{pmatrix} w \\ w \end{pmatrix}.$$



## $MWP$-BOUNDS ARE NOT UNIQUE

An $mwp$-bound: $\max(\vec{x}, q(\vec{y})) + p(\vec{z})$.

For example, expression $\mathtt{X}_1{+}\mathtt{X}_2$ has
three different valid $mwp$-bound descriptions $U, V, W$ where

$W(x_1;;x_1) \equiv \max(x_1, 0) + x_2$
$U(x_2;;x_1) \equiv \max(x_2, 0) + x_1$
$V(;x_1x_2;) \equiv \max(0, x_1 + x_2) + 0$

Here

- $W$ has $m$-variable $x_1$ and $p$-variable $x_2$;
- $U$ has $m$-variable $x_2$ and $p$-variable $x_1$;
- $V$ has $w$-variables $x_1, x_2$ and empty sets of $m$- and $p$-variables.

Why? to make it easier to recognise polynomial growth, for example

Analysable only if $\mathtt{X}_1$ is an $m$-variable: `loop X3 { X1 := X1 + X2 }`

Analysable only if $\mathtt{X}_2$ is an $m$-variable: `loop X3 { X2 := X1 + X2 }`

You can't have both!



## MATRIX ALGEBRA FOR $MWP$-BOUNDS ON COMMANDS

Matrices. $n \times n$.
Notation. $M_{ij} = \alpha$ written as $i \xrightarrow[M]{\alpha} j$.

Componentwise $\oplus$. Usual $\otimes$:

$$(M \otimes N)_{ij} = M_{i1} \times N_{1j} + \ldots + M_{ik} \times N_{kj} + \ldots + M_{in} \times N_{nj}$$

Matrix sum and conditional choice. If $\mathtt{C} : M$ and $\mathtt{C}' : N$ then

$$\texttt{if b then C else C}' : M \oplus N$$

Matrix product and sequential composition. If $\mathtt{C} : M$ and $\mathtt{C}' : N$ then

$$\mathtt{C};\mathtt{C}' : M \otimes N$$

## A LITTLE MORE MATRIX NOTATION

**Matrix modification notation.**

Let $M$ be a matrix and let $V$ be a vector. Then

$$M \overset{k}{\leftarrow} V$$

denotes the matrix obtained by replacing the $k$'th column in $M$ by the vector $V$.

**Matrix closure operation.**

A unary operation on matrices, denoted $*$ and called closure or star, is defined by the infinite sum

$$M^* \;=\; 1 \oplus M \oplus (M \otimes M) \oplus (M \otimes M \otimes M) \oplus (M \otimes M \otimes M \otimes M) \oplus \ldots$$

**Matrix closure and sequential iteration.** If $\mathtt{C} : M$ then $M^*$ describes effect of executing $\mathtt{C}$ for $0, 1, 2, \ldots$ times: any finite number of iterations.



## A CALCULUS TO PROVE COMMAND VALUE BOUNDS

**(S)**

$$\vdash \mathtt{skip} : 1$$

**(A)**

$$\frac{\vdash \mathtt{e} : V}{\vdash \mathtt{X}_j := \mathtt{e} : 1 \overset{j}{\leftarrow} V}$$

**(C)**

$$\frac{\vdash \mathtt{C}_1 : M_1 \qquad \vdash \mathtt{C}_2 : M_2}{\vdash \mathtt{C}_1 ; \mathtt{C}_2 : M_1 \otimes M_2}$$

**(I)**

$$\frac{\vdash \mathtt{C}_1 : M_1 \qquad \vdash \mathtt{C}_2 : M_2}{\vdash \mathtt{if\ e\ then\ C}_1 \mathtt{\ else\ C}_2 : M_1 \oplus M_2}$$

**(L)**

$$\frac{\vdash \mathtt{C} : M}{\vdash \mathtt{loop\ X}_\ell \ \{\mathtt{C}\} : M^* \oplus \{\, \ell \overset{p}{\rightarrow} j \mid \exists\, i\, [i \underset{M^*}{\overset{p}{\rightarrow}} j] \,\}} \text{(if diagonal}(M^*) = \text{all } m\text{'s)}$$



## WHAT'S HAPPENING IN THE LOOP RULE ?

**(L)**

$$\frac{\vdash \mathtt{C} : M}{\vdash \mathtt{loop\ X}_\ell \ \{\mathtt{C}\} : M^* \oplus \{\, \ell \overset{p}{\rightarrow} j \mid \exists\, i\, [i \underset{M^*}{\overset{p}{\rightarrow}} j] \,\}} \text{(if diagonal}(M^*) = \text{all } m\text{'s)}$$

1. Diagonal $i \underset{M^*}{\overset{w}{\rightarrow}} i$ or $i \underset{M^*}{\overset{p}{\rightarrow}} i$ are **forbidden**: possible non-polynomial growth.

2. If $i \underset{M^*}{\overset{p}{\rightarrow}} j$ then $\mathtt{X}_j$ is **made dependent on** the loop control variable $\mathtt{X}_\ell$.

3. Reasoning:
   - the value of loop control variable $\mathtt{X}_\ell$ determines
   - **the number of times** that variable $\mathtt{X}_j$ may be changed.

## EXAMPLES: THE LOOP RULE

1. Diagonal $i \xrightarrow[M^*]{w} i$ or $i \xrightarrow[M^*]{p} i$ are forbidden: possible non-polynomial growth.
2. If $i \xrightarrow[M^*]{p} j$ then $\mathtt{X}_j$ is made dependent on the loop control variable $\mathtt{X}_\ell$.
3. Example 1: $\mathtt{X}_1 \ := \ \mathtt{X}_1 \ + \ \mathtt{X}_2 : M$ where $M = \begin{pmatrix} m & \bot & \bot \\ p & m & \bot \\ \bot & \bot & m \end{pmatrix}$ has both $1 \xrightarrow[M^*]{m} 1$ and $2 \xrightarrow[M^*]{p} 1$.
4. Rule (L) makes $\mathtt{X}_1$ depend on $\mathtt{X}_3$. It adds $3 \xrightarrow{p} 1$, resulting in poly-time:
   $\mathtt{loop}\,\mathtt{X}_3\,\{\mathtt{X}_1 := \mathtt{X}_1 + \mathtt{X}_2\} : \begin{pmatrix} m & \bot & \bot \\ p & m & \bot \\ p & \bot & m \end{pmatrix}$
5. Example 2: $\mathtt{X}_1 \ := \ \mathtt{X}_1 \ + \ \mathtt{X}_2 : M$ where $M = \begin{pmatrix} p & \bot & \bot \\ m & m & \bot \\ \bot & \bot & m \end{pmatrix}$
6. Rule (L) cannot be applied at all (since $1 \xrightarrow[M^*]{p} 1$ ).

— 16 —

## SOUNDNESS

Theorem[Soundness] $\Gamma \vdash \mathtt{C} : M \ \Rightarrow \ \Gamma \models \mathtt{C} : M$.

Proved by induction on height of the proof of $\vdash \mathtt{C} : M$.

Key Lemma proves soundness of the loop rule.

Assume $\models C : M$ and $M^*$ has no diagonal edge $i \xrightarrow[M^*]{w} i$ or $i \xrightarrow[M^*]{p} i$.

Then, for $j = 1, \ldots, n$, there exist a fixed number $k$ and polynomials $p$ and $q$ such that for any $t$

$$[\![\mathtt{C}^t]\!](\vec{x}_n \rightsquigarrow \vec{x}_n) \ \Rightarrow \ x'_j \leq \max(\vec{u}, q(\vec{y})) + (t+2)^k p(\vec{z})$$

where

$$\vec{u} = \{x_i \mid i \xrightarrow[M^*]{m} j\} \text{ and } \vec{y} = \{x_i \mid i \xrightarrow[M^*]{w} j\} \text{ and } \vec{z} = \{x_i \mid i \xrightarrow[M^*]{p} j\}$$

Moreover, neither polynomial $p$ nor polynomial $q$ depends on $k$ or $t$.

In other words:

- $w$-variables have bounds that are independent of the loop iteration count.

— 17 —

## COMPLETENESS

Theorem[Completeness] Any feasible core command is derivable.

Not true: for any matrix $M$, $\models \mathtt{C} : M$ implies $\vdash \mathtt{C} : M$.

Proof sketch for [Completeness]: Assume core command $\mathtt{C}'$ is not derivable. (We will prove that $\mathtt{C}'$ is not feasible.)
Consider a smallest non-derivable subcommand of $\mathtt{C}$. It must have form $\mathtt{loop}\ \mathtt{X}_\ell\ \{\mathtt{C}\}$ or $\mathtt{while}\ ?\ \mathtt{do}\ \{\mathtt{C}\}$ where $\mathtt{C}$ is derivable. Prove that
$\exists t, k \in \mathbb{N} \forall \vec{x}_n \geq 2$

$$[\![\mathtt{C}^t]\!](\vec{x}_n \rightsquigarrow \vec{x}'_n) \ \ \wedge \ \ 2x_k \leq x'_k$$

where $\mathtt{C}^0 \equiv \mathtt{skip}$ and $\mathtt{C}^{t+1} \equiv \mathtt{C}; \mathtt{C}^t$.

(Phrased more lyrically: There exist $t \in \mathbb{N}$ and a variable $\mathtt{X}_k$ such that when $\mathtt{C}$ is executed $t$ times in a row on inputs $\geq 2$, the content of $\mathtt{X}_k$ will be at least doubled.)

Essential: a core command cannot decrease the value of a variable when its inputs are $\geq 1$ (otherwise it might multiply by zero).

— 18 —

## COMPLEXITY

Theorem The derivability problem is in $NP$.

Proof is by a "guess and verify" algorithm. Given a command C, we must decide whether $\vdash$ C : $M$ for some matrix $M$. If there is such a matrix, it can be expressed in $O(n^2)$ bits, where $n$ is the size of C.

- It is clearly PTIME-decidable whether the premises and conclusion of an inference rule are satisfied.
- The calculus rules are compositional in their syntactic arguments: matrix/vector for command or expression is defined using its substructures.

Thus exists a polynomial bound on the depth and total size of any proof tree ending in $\vdash$ C : $M$.
A nondeterministic algorithm: guess the matrices and vectors in such a tree, and check whether the inference rules of the caculus are satisfied at every node. Do this bottom-up, beginning with a root node $\vdash$ C : _.

CONJECTURE: The problem is $NP$-complete. Maybe $NP$-hardness construction analogous to "Complexity of . . . Dijkstra" (J. & Muchnick 1981).



## FILLING IN THE FRAMEWORK

- KJ (Kristiansen and Jones): matrix element values in $D = \{\bot, m, w, p\}$ where $m$ = maximum, $p$ = polynomially bounded.
  Minus: No inference if not polynomial-bounded. Tricky iteration treatment via "weak polynomial" dependence $w$.
- Size-change analysis (Lee-J-Ben-Amram): matrix entries in $D = \{\bot, >, \geq\}$. Analysis yields a set $\mathcal{M}$ of matrices.
  Each $M \in \mathcal{M}$ describes a definite data flow.
  Here $\oplus$ = set union, $\otimes$ = componentwise product.
- A possibility: strengthen, simplify KJ:
  - use sets of matrices over $D = \{\bot, m, p, \top\}$
  - treat self-dependence, iteration differently (like size-change)
- A dream: possible, I think. . . Develop a compositional set of inference rules to prove value bounds, runtime bounds, and termination
  Point: the three concepts are naturally inter-dependent.





## REFERENCES

References

[1] S.J. Bellantoni and S. Cook. A New Recursion-Theoretic Characterization of the Polytime Functions. Computational Complexity, 2:97–110 (1992).

[2] C.C. Frederiksen. Automatic runtime analysis for first order functional programs. Master Thesis, Dep. of Computer Science, University of Copenhagen, 2002.

[3] N.D. Jones and N. Bohr. Termination analysis of the untyped lambda-calculus. Rewriting Techniques and Applications, Springer LNCS Volume 3091 (2004), 1-23.

[4] L. Kristiansen and N.D. Jones. The Flow of Data and the Complexity of Algorithms (extended abstract). New Computational Paradigms (Computability in Europe) LNCS 3526 (2005), 263-274.

[5] L. Kristiansen and K.-H. Niggl. On the computational complexity of imperative programming languages. Theoretical Computer Science 318 (2004), 139-161.

[6] L. Kristiansen and K.-H. Niggl. The Garland Measure and Computational Complexity of Stack Programs. Electronic Notes in Theoretical Computer Science, Volume 90, Elsevier 2003.

[7] L. Kristiansen and P.J. Voda. Complexity classes and fragments of C. Information Processing Letters 88 (2003), 213-218.

[8] C.S. Lee, N. Jones, and A.M. Ben-Amram. The size-change principle for program termination. ACM Principles of Programming Languages ACM Press 2001, 81-92.

[9] K.-H. Niggl and H. Wunderlich. Certifying polynomial space and linear/polynomial space for imperative programs. Siam Journal on Computing.

# A Common Framework for Several Path-Summarizing Analyses

James Avery (avery@diku.dk)

Department of Computer Science
University of Copenhagen




Second DIKU-IST Workshop on Software Foundations, April 2006





## Outline

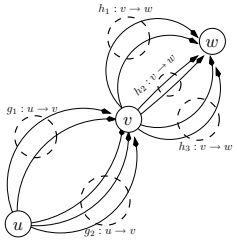
1. Introduction
2. SCT-like Analyses
3. A Common Domain
   - Definitions and Notation
   - Shared Properties
4. Conclusion & Future Work





## A Common Trait

- Finite abstract descriptions of the effects of infinitely many computation paths.
- Combinatorial aspect:




Level 1: Bundling effects of one-step transitions/calls.
Level 2: Composing these to summarize all possible effects of paths leading from pt. $A$ to pt. $B$.

## Benefits of Common Framework

1. A common domain may allow comparing the analyses to each other in a mathematically rigid fashion.
2. Results that apply to the general domain can be shared by all of the analyses.
3. A common "skeleton" of several analyses can work as a template for constructing new analyses that build on abstract summaries of path effects.
4. Implementing the common framework as a polymorphic library could allow fast prototyping.
5. It seems that the common structures that describe these analyses are well studied classical algebraic objects. **Pipe dream:** Apply classical algebraic results to show surprising, fun and/or useful properties of programs!





## The Scope of This Talk

- "Size-Change Termination"-like analyses.
- A common abstract domain.
- Some general properties.
- DISCLAIMER:

~~Work in Progress~~

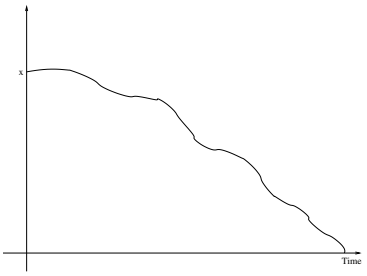


Work in Infancy



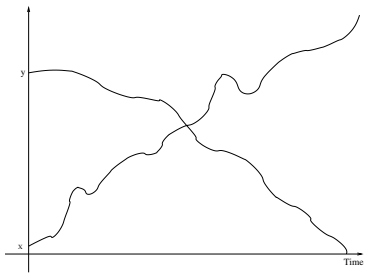


## SCT (Lee, Jones & Ben-Amram)




**Size-change** *termination analysis*: If any allegedly infinite computation would cause infinite descent in some well-founded variable, then the program terminates on all input.

## SCT-BA (Glenstrup & Jones, Copenhagen)

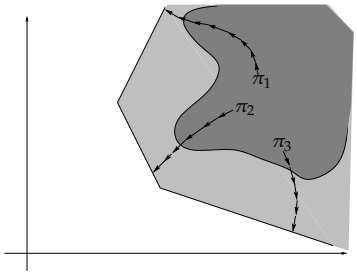



**Bounded Anchoring** *finiteness analysis*: If any computation that would cause $x$ to grow unboundedly **also** would cause infinite descent in some well-founded variable $y$, then for any input, $x$ has an upper bound.





## ESCT (Avery, Copenhagen)

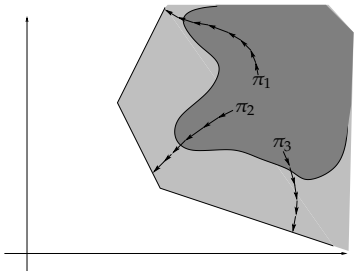



**ESCT** *termination analysis*: At each program point, discover a polyhedron $P$ enclosing all reachable stores. If every allegedly infinite computation would eventually leave $P$, then the program terminates on all input.





## ESCT-BA (Avery & Schou, Copenhagen)




**ESCT-BA** *finiteness analysis*: At each program point, discover a polyhedron $P$ enclosing reachable stores. If every computation that would cause $x$ to diverge unboundedly eventually leaves $P$, then $x$ is bounded.

## CCF Liveness (Frederiksen, Tokyo)

**FREDERIKSEN** *liveness analysis:* Infinite runs on combined program+Büchi automaton $P_{\Phi,\Psi}$ correspond to computations that are "fair" and falsify liveness property $\Psi$. Analysis uses a method similar to Avery's to show that no realizable infinite runs exist in order to conclude that $\Psi$ is eventually satisfied by all fair computations.





## Affine SCT (Anderson & Khoo, Singapore)

**Affine-Based size-change** *termination analysis*: Same overall goal as SCT, but uses Presburger formulæ to describe path effects. Finiteness is obtained by projection and widening.

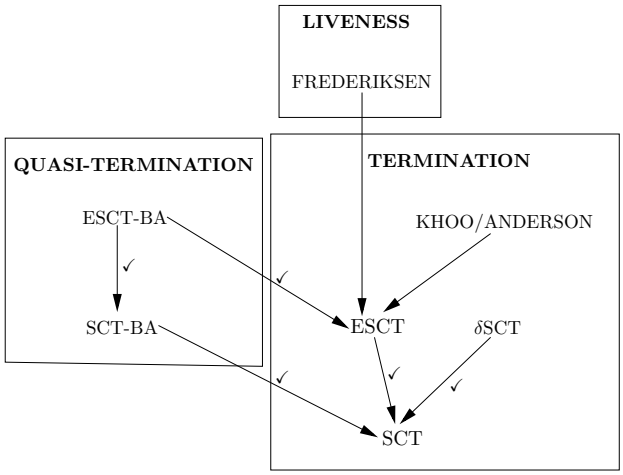




## $\delta$SCT (Ben-Amram, Tel-Aviv)

**$\delta$SCT** *termination analysis*: Same overall goal as SCT, but tracks integer differences between variable pairs. Decidable only for restricted problems that give finite closure.

## Expected Taxonomy








## Outline

1 Introduction
2 SCT-like Analyses
3 A Common Domain
- Definitions and Notation
- Shared Properties

4 Conclusion & Future Work





## Computation Model

- **Language**: L
- **Evaluation**: $[\![\mathrm{p}]\!]_{\mathrm{L}}\vec{x}_0 = (\sigma_i)_{i=0}^{B}$ with $B \in \mathbb{N} \cup \{\infty\}$, $\sigma_i \in \mathrm{States}_{\mathrm{p}}$. (E.g. $\sigma = (pc, env)$).
  It is deterministic, i.e. "current state determines remainder of computation".
- **Computations**:

$$\mathrm{Computations}_{\mathrm{p}} = \{[\![\mathrm{p}]\!]_{\mathrm{L}}\vec{x}_0 \mid \vec{x}_0 \in \mathit{Inputs}\}$$

- **Computation segments**:

$$\mathrm{Segments}_{\mathrm{p}} = \left\{ (\sigma_i)_{i=0}^{m} \;\middle|\; \begin{array}{c} \exists k \in \mathbb{N}, c \in \mathrm{Computations}_{\mathrm{p}} \\ s.t.\ c_{k+i} = \sigma_i \ \text{ for } 0 \le i \le m \end{array} \right\}$$

## Control Flow

- Abstract set $\text{Labels}_{\mathrm{p}}$ and partial map $\mathit{Label} : \text{States}_{\mathrm{p}} \rightharpoonup \text{Labels}_{\mathrm{p}}$ such that every state $\sigma$ that a program run goes through is "located" at a label $l = \mathit{Label}(\sigma)$.
- **Require:** No infinite run "gets stuck" on one label, i.e. if $(\sigma_i)_{i=0}^{\infty} \in \text{Computations}_{\mathrm{p}}$ and $\mathit{Label}(\sigma_m) = u$, there is an $n > m$ such that $\mathit{Label}(\sigma_n) \neq u$.





## Control Flow 2

For a computation segment $c = (c_i)_{i=0}^{m}$, write:

- $c : u \to v$ if $u = \mathit{Label}(c_0)$ and $v = \mathit{Label}(c_m)$
- $u \xrightarrow{*} v$, $u \xrightarrow{1} v$, ..., as usual.
- $\text{Onestep}_{\mathrm{p}} = \{c : u \xrightarrow{1} v \in \text{Segments}_{\mathrm{p}}\}$

The Control flow graph $CFG = (\text{Labels}_{\mathrm{p}}, E)$ of program $\mathrm{p}$ has arcs

$$E = \left\{ (u, v) \in \text{Labels}_{\mathrm{p}}^2 \;\middle|\; \exists c : u \xrightarrow{1} v \in \text{Onestep}_{\mathrm{p}} \right\}$$





## Abstracting Path Effects 1

Describe path effects by morphisms $\mathbb{G} = \hom(K)$ of some category $K$ with $\text{ob}(K) = \text{Labels}$.[1]

Relation "models", written $g \models c$, must have

$$\left.\begin{array}{c} g : u \to v \models c : u \xrightarrow{*} v \\ h : v \to w \models c' : v \xrightarrow{*} w \\ c; c' \in \text{Segments}_{\mathrm{p}} \end{array}\right\} \implies g; h \models c; c'$$

[1] NB: No category theory whatsoever coming up!

## Abstracting Path Effects 2

Let $\mathcal{G} : \mathrm{L} \to \wp(\mathbb{G})$ be our transition abstracting function. $\mathcal{G}$ assigns from syntax abstract descriptions of one-step transitions:

$$\forall c : u \xrightarrow{1} v \in \mathsf{Onestep}_{\mathrm{p}}\ \exists g \in \mathcal{G}(\mathrm{p})\ s.t.\ g \models c$$

For proof-purposes **we arbitrarily define**

$$\mathcal{G}^{(1)} : \mathsf{Onestep}_{\mathrm{p}} \to \mathcal{G}(\mathrm{p})$$

to choose one such $g$.

**We require**: $|\mathcal{G}(\mathrm{p})| < \infty$ and $|\mathcal{G}(\mathrm{p})^*| < \infty$.





## Outline

1. Introduction
2. SCT-like Analyses
3. A Common Domain
   - Definitions and Notation
   - Shared Properties
4. Conclusion & Future Work





## Idempotent Morphisms Describe Infinite Computations

**Theorem ($\mathrm{T}_1$: Idempotent morphisms characterize infinite computations)**

*Any infinite computation $c = (c_i)_{i=0}^{\infty}$ can be partitioned as $\overbrace{c_{p_0} \cdots c_{p_1-1}}^{c_{p_0 p_1}}; \overbrace{c_{p_1} \cdots c_{p_2-1}}^{c_{p_1 p_2}}; \cdots$ such that every segment after the first is modeled by the same idempotent morphism. I.e. there exists $g_0 \in \mathcal{G}(\mathrm{p})^*$ such that for all $i > 0$:*

1. $g_0 \models c_{p_i p_{i+1}}$, *and*
2. $g_0^2 = g_0$.

**Remark**

*If any label $u$ is passed infinitely many times by $c$, then we can choose $g_0 : u \to u$, i.e. let every segment begin and end at this $u$.*

## Proof of $T_1$ (1/2)

**Proof, modeled by the same morphism.**

Let $(c_i)_{i=0}^{\infty} \in \text{Computations}_{\mathrm{p}}$ and write $c_{mn} = (c_i)_{i=m}^{n-1}$. For each $g \in \mathcal{G}(\mathrm{p})^*$ define

$$P_g = \left\{ (m,n) \in \mathbb{N}^2 \;\middle|\; m < n \text{ and } g = \prod_{k=m}^{n-1} \mathcal{G}^{(1)}(c_{k,k+1}) \right\}$$

Because $|\mathcal{G}(\mathrm{p})^*| < \infty$, there are finitely many of these disjoint classes. By Ramsey's theorem there is a $g_0 \in \mathcal{G}(\mathrm{p})^*$ and infinite set $I \subseteq \mathbb{N}$ such that for all $m < n \in I$ we have $(m,n) \in P_{g_0}$, and specifically $g_0 \models c_{mn}$ as desired. □





## Proof of $T_1$ (2/2)

**Proof, idempotence.**

Now choose any $i < j < k \in I$. Then

$$g_0 = \prod_{l=i}^{k-1} \mathcal{G}^{(1)}(c_{l,l+1}) = \prod_{l=i}^{j-1} \mathcal{G}^{(1)}(c_{l,l+1}) \prod_{l=j}^{k-1} \mathcal{G}^{(1)}(c_{l,l+1}) = g_0^2$$

Thus $g_0$ must be idempotent. □





## A Rephrasing of the Halting Problem

**Observation**

*Computation $(c_i)_{i=0}^{B}$ is finite if and only if there exists a sub-sequence $(c_{k_i})_{i=0}^{B'}$ ($B' = \infty$ if $B = \infty$) and partial map $f : \text{States}_{\mathrm{p}} \rightharpoonup \mathbb{N}$ such that*

$$f(c_{k_i}) > f(c_{k_{i+1}})$$

*for all $i$.*

## Effect of Abstract Elements on Concrete Functions of State

**Definition (decreases)**

$g : u \to u$ is said to decrease $f : \text{States}_{\mathrm{p}} \rightharpoonup \mathbb{N}$ iff for every $(c_i)_{i=0}^{m} \in \text{Segments}_{\mathrm{p}}$ for which $g \models (c_i)_{i=0}^{m}$, we have

$$f(c_0) > f(c_m)$$

I.e. obvious meaning: "Abstract $g$ decreases $f$ iff every concrete computation segment corresponding to $g$ actually decreases $f$."





## General Size-change Termination Theorem (Undecidable)

**Theorem**

*Let $u$ be a label. If for every idempotent morphism $g : u \to u \in \mathcal{G}(\mathrm{p})^*$ there exists $f_g : \text{States}_{\mathrm{p}} \rightharpoonup \mathbb{N}$ such that $g$ decreases $f_g$, then no computation passes $u$ infinitely often.*

**Corollary**

*If this is true for every label $u$, then program $\mathrm{p}$ terminates on all input.*





## Getting Back to the Current Analyses

The analyses all use this property, but with different abstract domains $\mathbb{G}$ and different schemes for finding decreasing $f$'s.

## Quick Definitions

Let in the following:

1. $\mathbb{M}(R)$ be the matrix algebra over semiring $R$ with usual matrix sum and product, i.e.

$$(XY)_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj} \text{ when } X \in \mathbb{M}_{m\times n}(R), Y \in \mathbb{M}_{n\times p}(R)$$

$$(X+Y)_{ij} = X_{ij} + Y_{ij} \text{ when } X, Y \in \mathbb{M}_{m\times n}$$

2. $\mathbb{P}_m$ be the lattice of convex polyhedra of dimension $m$ over $\mathbb{Z}$.
3. $\mathfrak{a}^{\downarrow} = \{0, \overline{\downarrow}, \downarrow\}$ ("Must decrease"-arrows)
   $\mathfrak{a}^{\uparrow?} = \{0, \updownarrow, \uparrow\}$ ("May increase"-arrows), etc.
4. $A =$ source of $g$, $B =$ target of $g$.





## What are the Morphisms $\mathbb{G}$?

| | $g : u \to v$ is a member of |
|---|---|
| SCT | $\mathbb{M}_{\lvert A\rvert\times\lvert B\rvert}(\mathfrak{a}^{\downarrow})$ |
| SCT-BA | $\mathbb{M}_{\lvert A\rvert\times\lvert B\rvert}(\mathfrak{a}^{\downarrow}\times\mathfrak{a}^{\uparrow?})$ |
| ESCT | $\mathbb{P}_{\lvert A\rvert} \times \mathbb{M}_{\lvert A\rvert\times\lvert B\rvert}(\mathfrak{a}^{\downarrow}\times\mathfrak{a}^{\uparrow})$ |
| ESCT-BA | $\mathbb{P}_{\lvert A\rvert} \times \mathbb{M}_{\lvert A\rvert\times\lvert B\rvert}(\mathfrak{a}^{\downarrow}\times\mathfrak{a}^{\uparrow?}\times\mathfrak{a}^{\uparrow}\times\mathfrak{a}^{\downarrow?})$ |
| FREDERIKSEN | $\mathbb{P}_{\lvert A\rvert} \times \mathbb{M}_{\lvert A\rvert\times\lvert B\rvert}(\mathfrak{a}^{\downarrow}\times\mathfrak{a}^{\uparrow})$ |
| ANDERSON/KHOO | $\wp\left(\mathbb{P}_{\lvert A\rvert+\lvert B\rvert}\right)$ |
| $\delta$SCT | $\mathbb{M}(\mathbb{Z}\cup\{\infty\})$ |





## What do they Decrease?

| | Candidates to $f_g$ |
|---|---|
| SCT | $f(\vec{x}) = \lvert x_i\rvert$, $1 \le i \le \lvert A\rvert$. |
| SCT-BA | Same as SCT. |
| ESCT | $f(\vec{x}) = c_0 + \sum_{k=1}^{\lvert A\rvert} c_k x_k$ such that $f(\vec{x}) \ge 0$ contains $P$. |
| ESCT-BA | Same as ESCT. |
| FREDERIKSEN | Same as ESCT. |
| ANDERSON/KHOO | Same as SCT. |
| $\delta$SCT | Same as SCT. |

## Test on Idempotents

| | $g : u \to u = g^2$ "OK" iff |
|---|---|
| SCT | $\exists i : g_{ii} = \downarrow$. (In-situ decrease) |
| SCT-BA | Same as SCT or $\nexists i : g^{\uparrow}_{ii} = \uparrow$. (No "dangerous" vars) |
| ESCT | 1. $\forall k$ : $c_k > 0$ implies $g^{\downarrow}_{kk} = \overline{\downarrow}$ or $g^{\downarrow}_{kk} = \downarrow$; $c_k < 0$ implies $g^{\uparrow}_{kk} = \underline{\uparrow}$ or $g^{\uparrow}_{kk} = \uparrow$<br>2. $\exists k$ : $c_k > 0$ and $g^{\downarrow}_{kk} = \downarrow$ or $c_k < 0$ and $g^{\uparrow}_{kk} = \uparrow$ |
| ESCT-BA | Same as ESCT or $\nexists i : (g^{\uparrow?}_{ii} = \uparrow$ or $g^{\downarrow?} = \downarrow)$. (No "dangerous" vars). |
| FREDERIKSEN | Same as ESCT. |
| ANDERSON/KHOO | $\exists i : x'_i < x_i$. |
| $\delta$SCT | $\exists i : g_{ii} < 0$. |





## Reasoning

| | Consequence of $g : u \to u = g^2$ "OK" |
|---|---|
| SCT | No realizable infinite computation is described by $g$. |
| SCT-BA | No variable is increased unboundedly by a computation described by $g$. |
| ESCT | Same as SCT. |
| ESCT-BA | Same as SCT-BA. |
| FREDERIKSEN | No fair computation that doesn't satisfy liveness property $\Psi$ is described by $g$. |
| ANDERSON/KHOO | Same as SCT. |
| $\delta$SCT | Same as SCT. |





## Conclusion and Future Work

**Begun work on**:

1. A common domain may allow comparing the analyses to each other in a mathematically rigid fashion.
2. A common "skeleton" of several analyses can work as a template for constructing new analyses that build on abstract summaries of path effects.
3. Results that apply to the general domain can be shared by all of the analyses.

**Not yet looked at**:

1. Implementing the common framework as a polymorphic library could allow fast prototyping.
2. It seems that the common structures that describe these analyses are well studied classical algebraic objects. **Pipe dream:** Apply classical algebraic results to show surprising, fun and/or useful properties of programs!

# Abstraction of programs manipulating pointers using modal logics

Yoshinori TANABE (IST & AIST)

(Joint work with Yoshifumi YUASA, Toshifusa SEKIZAWA and Koichi TAKAHASHI (AIST) )

2nd DIKU-IST Joint Workshop on Foundations of Software
21 Apr., 2006

## Overview

- Analysis of programs manipulating pointers (shape analysis) in the predicate abstraction framework.
- We use formulae of modal logics as predicates.
- In previous study our logic was the two-way CTL with nominals (2CTLN). It was not strong enough to verify the Schorr-Waite algorithm, which is regarded as a benchmark for this type of analysis.

> *The Schorr-Waite algorithm is the first mountain that any formalism for pointer aliasing should climb.*
>
> —Richard Bornat

- In this on-going study we use a stronger logic: the alternation-free modal mu- calculus with nominals and the global modality (AFMNG).
- Both safety and liveness properties are handled.

Logic AFMNG

Schorr-Waite Algorithm

Verification Method

Conclusion

## Syntax of AFMNG

- AFMNG: Alternation Free Mu-calculus with Nominals and Global modality
- Parameters
  - PC ∋ p: Propositional Constant
  - Nom ∋ n: Nominal
  - BMod ∋ f: Basic Modalities
- Propositional Variables $X ::= X_1 \mid X_2 \mid \ldots$
- Modalities $m ::= o \mid f \mid \bar{f}$ o: global modality
- MNG $\varphi ::= p \mid n \mid X \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle m \rangle \varphi \mid \mu X \varphi$ (X is positive in φ)
- φ is alternation-free if it is equivallent to an NNF formula

..... μ X( ..... ν Y( ..... )..... ) ...

no free occurence of X

..... ν Z( ..... μ W( .....) .....) .....

no free occurence of Z

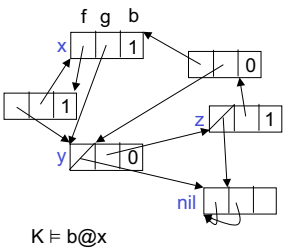
## Semantics of AFMNG

- Semantics are given by Kripke Structure (K,R,λ), where
  - K: universe
  - R: Mod → $2^{K \times K}$ relation defined for each modality
  - λ: PC ∪ Nom → $2^K$ Nominals are like predicate constants.
- $\lambda(\bar{n})$ is a singleton, for n ∈ Nom A nominal is satisfied at just one node.
- $R(f) = R(f)^{-1}$ f is the reverse modality of f
- $R(o) = K \times K$ o expresses the global relation.
- Others are same as the standard mu-calculus.
- Abbreviations etc
  - ∧, →, $[m]\varphi = \neg\langle m \rangle \neg\varphi$, $\nu X = \neg\mu X \neg \varphi[\neg X/X]$
  - $K, s' \models [o]\varphi \Leftrightarrow \forall s \in K\ K,s \models \varphi$ independent from s'
  - $K, s' \models \langle o \rangle \varphi \Leftrightarrow \exists s \in K\ K,s \models \varphi$ independent from s'
  - $\varphi@n = [o](n \rightarrow \varphi) \equiv \langle o \rangle(n \wedge \varphi)$ for n ∈ Nom. φ holds at the node pointed-to by n

## Heap as a Kripke Structure

```
struct Node {
  Node* f;
  Node* g;
  Bool  b;
};

Node* x,y,z;
```




PC = {b}
boolean field names as PC
Nom = {x,y,z , nil }
pointer variables as nominals
BMod = {f,g}
pointer fields as basic modalities

$K \models b@x$
b is set at node x.

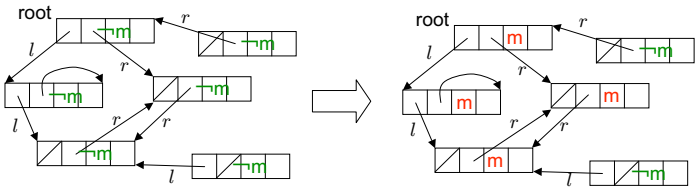
$K \models \langle \bar{f} \rangle b@y$
There is a f-parent of y where b is set.

$K \models (\mu X(\ y \vee \langle f \rangle X)) @ x$
y is f-reachable from x

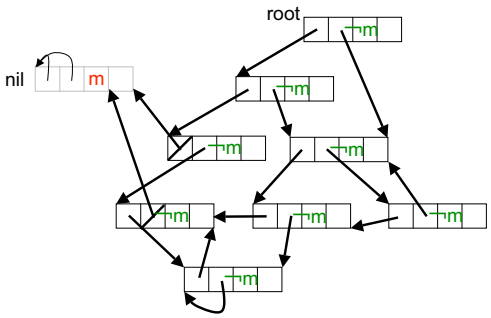
$K \models (\langle g \rangle \mu X(\ y \vee \langle g \rangle X)) @ y$
y is in a g-loop.

## The Schorr-Waite Algorithm

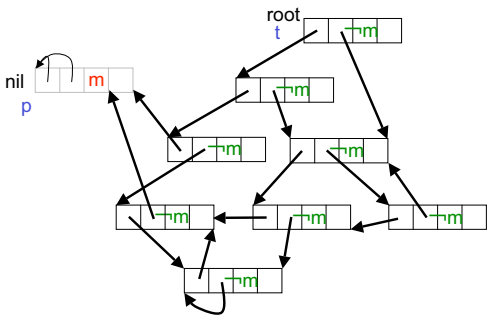
- Marks all nodes that are reachable from the root node in the manner of DFS.
- Does not use a stack to hold the nodes for backtracking, rewrites the pointers to remember the parent node instead.




## The Schorr-Waite Algorithm



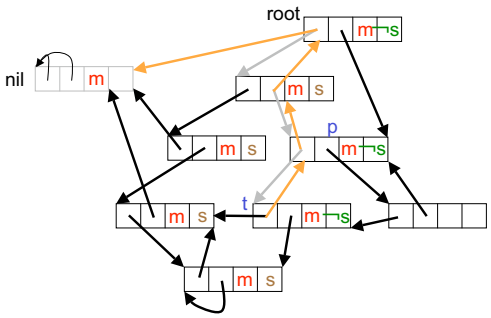

## The Schorr-Waite Algorithm (start)




conditions:

- p points to nil $\text{nil}@p$
- t points to root $\text{root}@t$
- every node is unmarked. $[o](\neg \text{nil} \rightarrow \neg m)$

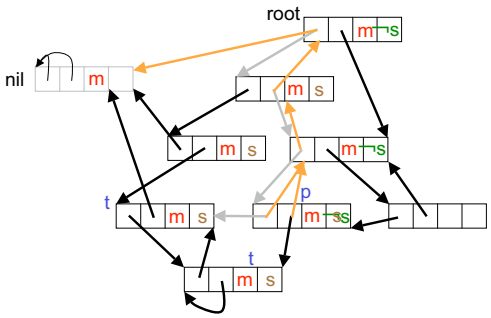
## The Schorr-Waite Algorithm (push)




conditions:

- t is unmarked $\neg m @ t$

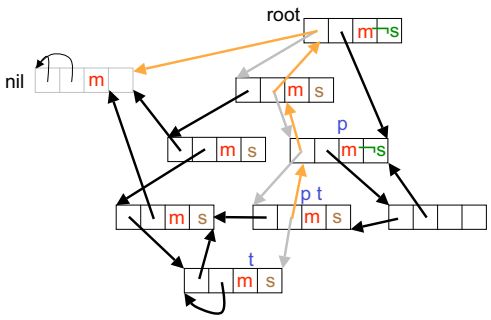
## The Schorr-Waite Algorithm (swing)




conditions:

- t is marked $m @ t$
- p is unswung $\neg s @ p$

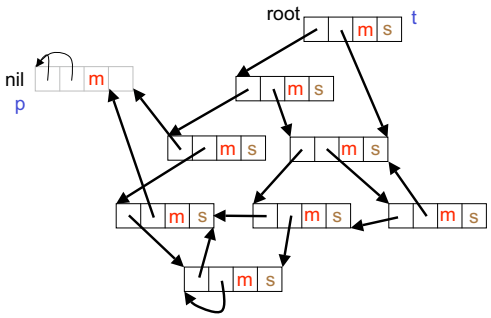
## The Schorr-Waite Algorithm (pop)




conditions:

- t is marked $m @ t$
- p is swung $s @ p$

## The Schorr-Waite Algorithm (termination)



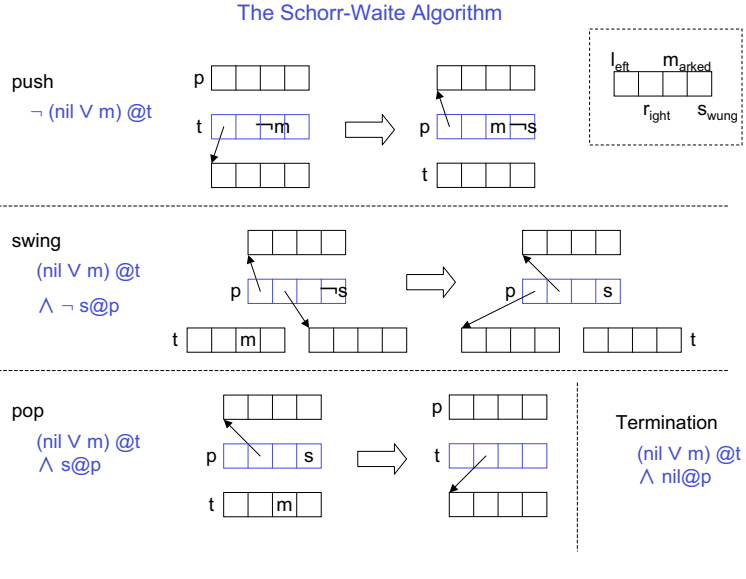

conditions:

- p points to nil $\mathrm{nil}@p$
- t is marked $m@t$

## The Schorr-Waite algorithm

```
void schorr_waite(node root) {
  node t = root; node p = NULL;
  while (p != NULL || (t != NULL && ! t->m)) {
    if (!(t == NULL || t->m)) {   /* push */
      node q = p; p = t; t = t->l;
      p->l = q; p->m = 1; p->c = 0;
    }else if (! p->c) {           /* swing */
      node q = t; t = p->r; p->r = p->l;
      p->l = q; p->c = 1;
    }else {                       /* pop */
      node q = t; t = p; p = p->r; t->r = q;
    }
  }
}
```

## The Schorr-Waite Algorithm



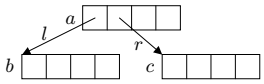
## Properties to Verify

- (liveness) The algorithm terminates for any heap structure.

- (safety) A node that is reachable from the root at the beginning is marked when the algorithm terminates.
- (safety) The "points-to" relation at the beginning is identical to that at the end

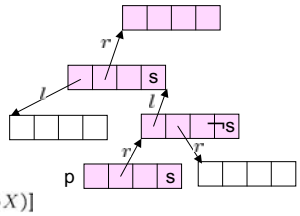
Take an arbitrary non-nil node $a$, which is reachable from the root at the beginning. Let $b$ and $c$ be the left and right child of $a$, resp., then at the end:

- $a$ is marked. ($m@a$)
- $b$ and $c$ is the left and right child of $a$, resp. ( $\langle l\rangle b@a,\ \langle r\rangle c@a$ )

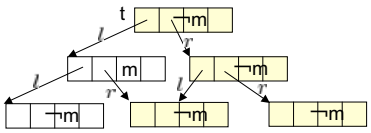


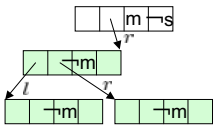

## Predicates

- a, b, c, p, t, nil, m, s, <l>b, <r>c, ...
- RPp ≡ $\mu X[p \vee \langle\bar{l}\rangle(\neg s \wedge X) \vee \langle\bar{r}\rangle(s \wedge X)]$
  reachable with "pop" relation from p

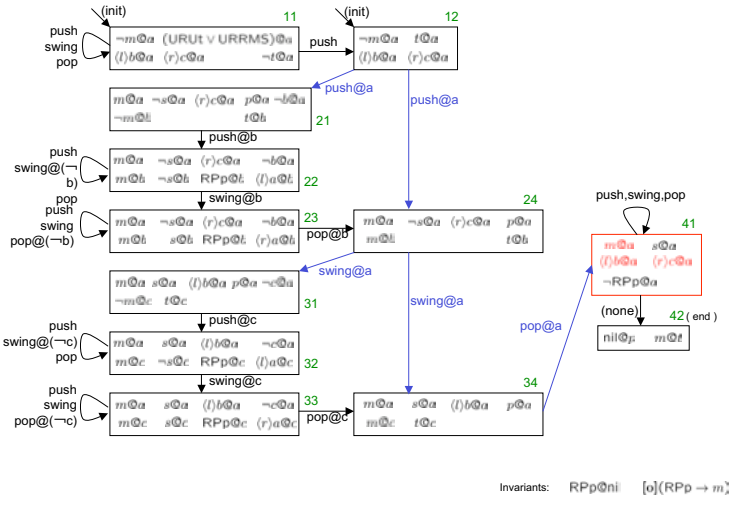



- URRMS ≡ $\mu X[\neg m \wedge (\langle\bar{r}\rangle(m \wedge \neg s) \vee \langle\bar{l}\rangle X \vee \langle\bar{r}\rangle X)]$
  unmarked-reachable from the right child of a marked and unswung node
- URUt ≡ $\mu X[\neg m \wedge (t \vee \langle\bar{l}\rangle X \vee \langle\bar{r}\rangle X)]$
  unmarked-reachable from unmarked t

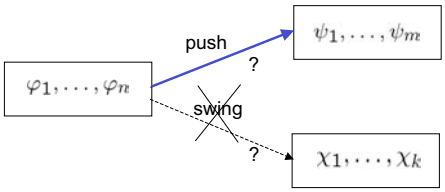



## The Abstract Transition Relation for the Safety Properties



## Deciding the Abstract Transition Relation




- If wp(push, $\psi$) $\wedge$ $\varphi$ is satisfiable
  and wp (swing, $\chi$) $\wedge$ $\varphi$ is NOT satisfiable ....

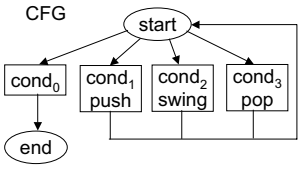
$$\begin{bmatrix} \varphi = \varphi_1 \wedge \ldots \wedge \varphi_n \\ \psi = \psi_1 \wedge \ldots \wedge \psi_m \\ \chi = \chi_1 \wedge \ldots \wedge \chi_k \end{bmatrix}$$

- AFMNG is
  - closed under taking weakest preconditions
  - decidable and has an effective decision procedure for satisfiability

## Termination

- Three ranking functions:

$$f_1 : \mathcal{S} \mapsto \{s \in S \mid \mathcal{S}, s \models \neg m\}$$
$$f_2 : \mathcal{S} \mapsto \{s \in S \mid \mathcal{S}, s \models \neg s\}$$
$$f_3 : \mathcal{S} \mapsto \{s \in S \mid \mathcal{S}, s \models \mathrm{RPp}\}$$




- Use the well-founded relation "$\supset$" on $2^S$.

| | $f_1$ | $f_2$ | $f_3$ |
|---|---|---|---|
| push | decreasing | --- | --- |
| swing | non-increasing | decreasing | non-increasing |
| pop | non-increasing | non-increasing | decreasing |

("non-increasing" means "decreasing or identical" )

- Using a lexicographic order, we can conclude that the algorithm terminates.
- How can we judge "non-increasing" and "decreasing"?

## Judging Non-increase and Decrease

- For operation op and formula $\varphi$, we define
  - NI(op, $\varphi$) = [o] ( wp(op, $\varphi$) $\rightarrow$ $\varphi$ )
  - D(op, $\varphi$) = NI(op, $\varphi$) $\wedge$ <o> ( wp(op, $\neg\varphi$) $\wedge$ $\varphi$)
- function f: $\mathcal{S} \mapsto$ { s $\in$ S | $\mathcal{S}$, s $\vDash$ $\varphi$ } is
  - non-increasing on op if NI(op, $\varphi$) is valid (i.e. its negation is not satisfiable)
  - decreasing on op if D(op, $\varphi$) is valid

Proof: Assume $\mathcal{S}_{pre}$ ---op--->$\mathcal{S}_{post}$.

If NI(op, $\varphi$) is valid, $\mathcal{S}_{pre} \vDash$ NI(op, $\varphi$) holds.

I.e. for any s $\in$ S $\mathcal{S}_{pre}$, s $\vDash$ wp(op, $\varphi$) $\Rightarrow$ $\mathcal{S}_{pre}$, s $\vDash$ $\varphi$

$\mathcal{S}_{post}$, s $\vDash$ $\varphi$ $\Rightarrow$ $\mathcal{S}_{pre}$, s $\vDash$ $\varphi$ ,

which means f($\mathcal{S}_{post}$) $\subseteq$ f($\mathcal{S}_{pre}$)

## Conclusion

- Analyzing programs manipulating pointers in the predicate abstraction framework using formulae of AFMNG, a modal logic, as predicates.
- Both safety and liveness properties are handled.
- Key issues are that the logic AFMNG is
  - decidable, has an effective decision procedure
  - closed under taking weakest preconditions for basic pointer manipulation
- Ongoing activity
  - a detailed procedure for deciding transition relation
  - an experimental implementation of the decision procedure for satisfiability of AFMNG
- Future work
  - extension of logic to handle more complicated properties / heap structure
    - bounded modalities
    - the downarrow binder
  - finding predicates for safety from counterexamples
  - finding predicates for liveness

## Related Work

- *Sagiv, Reps, Wilhelm: Parametric Shape Analysis via 3-valued Logic. ACM Transactions on Programming Languages and Systems, vol 24 2002, pp.217-298.* Shape analysis using abstract interpretation based on three valued logic. The logic for expressing the heap is FO+TC. The tool is called TVLA
- *Møller and Schwartzbach: The Pointer Assertion Logic Engine. PLDI'01.* Shape analysis that employs MSO as the logic for expressing the heap properties. The tool is called PALE.
- Balaban, Pnueli, Zuck: Shape Analysis by Predicate Abstraction. VMCAI 2005. Uses a decidable fragment of FO+TC as predicates. Both safety and liveness properties are handled.
- *John Reynolds: Separation Logic: A Logic for Shared Mutable Data Structures. LICS 2002. pp55-74.* An extension of Hoare logic for pointer manipulating programs.

# Inferring bidirectionality of programs
# — Work In Progress —
# 2nd DIKU-IST workshop

Johan Gade

joint work with Zhenjiang Hu, Keisuke Nakano and Kazutaka Matsuda

April 21, 2006

**A view from Mt. Fuji** **2**
- What is bidirectional transformations? . . . 3
- Example . . . 4
- Motivation . . . 5
- Objective . . . 6
- Example . . . 7

**Interaction model** **8**
- Summary . . . 9

**Language** **10**
- Characteristics . . . 11
- Syntax . . . 12
- Forward semantics . . . 13
- Backward semantics I . . . 14
- Backward semantics II . . . 15

**Conclusion** **16**
- Achievements & Future Goals . . . 17

## What is bidirectional transformations?

*bidirectional* |bidi rek sh onl|
adjective
functioning in two directions.
— New Oxford American Dictionary, 2nd Edition.

So a bidirectional transformation can transform data back and forth between different representations.

3 / 17

## Example

- Example: *Interactive editing*.
- Document preparation systems such as LaTeX can execute code to produce a Table of Contents (TOC).
- Given the input data — a document — it computes a list of section titles.
- We would like to modify one of the titles in the list — AND have the changed title reflected back into the document.
- A twist: We would also like to be able to *add* an entirely new title to the TOC — AND reflect the changes back to the input — creating a new section.
- Similarly. It should be possible to *delete* and entry and have the changes reflected on the input.

4 / 17

## Motivation

The need for two-way or *bidirectional* transformations often arise:

- Synchronizing data between different sources (calendars, bookmarks, etc.).
- Transformations between UML models and source code.
- Undoing and redoing operations in an editor (of any type of media).
- Compression and decompression of data.

However...

- The transformations are written *manually*.
- *Both ways!*

*This is error-prone!*

**Overall Goal:** Reduce the probability of errors.

5 / 17

## Objective

Design and implement a system that infers whether programs are *bidirectional*.

**Definition** (Bidirectional program — intuitively)**.** *A program $p$ is called* bidirectional *iff for any input $i$:*

$$p(i) \Rightarrow v_{orig} \text{ and modifying the result to } v_{mod}$$
$$\text{results in new inputs } i_{new} \text{ such that}$$
$$p(i_{new}) \Rightarrow v_{new} \text{ and } v_{new} \cong v_{mod}$$

A secondary objective is to obtain the new inputs $i_{\text{new}}$ automatically if $p$ is bidirectional.

**Note.** *Not inversion* since no inverse program is constructed. But an invertible program is naturally bidirectional.

6 / 17

### Example

Consider run-length encoding:

$$\begin{aligned} \mathrm{rle}([]) &= [] \\ \mathrm{rle}(x : xs) &= \mathrm{rle}_{\mathsf{aux}}(xs, x, 1) \end{aligned}$$

where

$$\begin{aligned} \mathrm{rle}_{\mathsf{aux}}([], elm, cnt) &= [\langle elm, cnt \rangle] \\ \mathrm{rle}_{\mathsf{aux}}(x : xs, elm, cnt) &= \mathbf{if}\ x = elm \\ &\quad \mathbf{then}\ \mathrm{rle}_{\mathsf{aux}}(xs, elm, cnt + 1) \\ &\quad \mathbf{else}\ \langle elm, cnt \rangle : \mathrm{rle}_{\mathsf{aux}}(xs, x, 1) \end{aligned}$$

E.g. $\mathrm{rle}[a, a, b, c, c, c] \Rightarrow [\langle a, 2 \rangle, \langle b, 1 \rangle, \langle c, 3 \rangle]$.

7 / 17

## Interaction model

8 / 17

### Summary

- The system works interactively. After each change made to the output, the system performs a synchronization between input and output.
- The following 3 kinds of modifications are considered.
  1. Updating / revising
  2. Deleting
  3. Inserting
- Not all of the modifications make sense for all types of structures. For instance deletions and insertions *only* make sense for lists.

9 / 17

## Characteristics

- First-order functional language with call-by-value forward semantics.
- *Lineарity*: All variables occur at most once in the body of a function.
- A special duplication expression if the value of a variable should be needed more than once: $\mathbf{let}\ \langle x_1, x_2\rangle = \mathbf{dup}\ e\ \mathbf{in}\ e'$.

11 / 17

## Syntax

| | | | |
|---|---|---|---|
| $p$ | $::=$ | $f_{\mathsf{def}}^{+}$ | Program |
| $f_{\mathsf{def}}$ | $::=$ | $f(x_1, \ldots, x_n) \equiv e$ | Function definition |
| $e$ | $::=$ | $v$ | Value |
| | $\mid$ | $x$ | Variable reference |
| | $\mid$ | $\langle e_1, e_2\rangle$ | Tuple |
| | $\mid$ | $f(e_1, \ldots, e_n)$ | Function application |
| | $\mid$ | $f_{\mathsf{prim}}(e_1, \ldots, e_n)$ | Application of primitive |
| | $\mid$ | $\mathbf{fst}\ e \mid \mathbf{snd}\ e$ | Tuple element extraction |
| | $\mid$ | $\mathbf{hd}\ e$ | Head element extraction |
| | $\mid$ | $\mathbf{tl}\ e$ | The remainder of the list |
| | $\mid$ | $e_1 : e_2$ | List construction |
| | $\mid$ | $\mathbf{isnil}\ e$ | List empty predicate |
| | $\mid$ | $\mathbf{let}\ \langle x_1, x_2\rangle = \mathbf{dup}\ e\ \mathbf{in}\ e$ | Variable duplication |
| | $\mid$ | $\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$ | Conditional expression |
| $v$ | $::=$ | $n \mid b \mid \langle v_1, v_2\rangle \mid v_1 : v_2 \mid \mathsf{nil}$ | Values |
| $n$ | $\in$ | $\mathbb{Z} = \{\ldots, -1, 0, 1, 2, \ldots\}$ | Integer numerals |
| $b$ | $\in$ | $\{\mathsf{True}, \mathsf{False}\}$ | Booleans |

12 / 17

## Forward semantics

$$\frac{}{\Sigma \vdash v \Rightarrow v} \qquad \frac{}{\Sigma \vdash x \Rightarrow \Sigma(x)} \qquad \frac{\Sigma \vdash e \Rightarrow \langle v_1, v_2 \rangle}{\Sigma \vdash \mathbf{fst}\ e \Rightarrow v_1} \qquad \frac{\Sigma \vdash e \Rightarrow \langle v_1, v_2 \rangle}{\Sigma \vdash \mathbf{snd}\ e \Rightarrow v_2}$$

$$\frac{\Sigma \vdash e_1 \Rightarrow v_1 \quad \Sigma \vdash e_2 \Rightarrow v_2}{\Sigma \vdash \langle e_1, e_2 \rangle \Rightarrow \langle v_1, v_2 \rangle} \qquad \frac{\Sigma \vdash e_1 \Rightarrow v_1 \quad \Sigma \vdash e_2 \Rightarrow v_2}{\Sigma \vdash e_1 : e_2 \Rightarrow v_1 : v_2} \qquad \frac{\Sigma \vdash e \Rightarrow v_1 : v_2}{\Sigma \vdash \mathbf{hd}\ e \Rightarrow v_1}$$

$$\frac{\Sigma \vdash e \Rightarrow v_1 : v_2}{\Sigma \vdash \mathbf{tl}\ e \Rightarrow v_2} \qquad \frac{\Sigma \vdash e \Rightarrow \mathsf{nil}}{\Sigma \vdash \mathbf{isnil}\ e \Rightarrow \mathsf{True}} \qquad \frac{\Sigma \vdash e \Rightarrow v_1 : v_2}{\Sigma \vdash \mathbf{isnil}\ e \Rightarrow \mathsf{False}}$$

$$\frac{\Sigma \vdash e_1 \Rightarrow v_1 \quad \cdots \quad \Sigma \vdash e_n \Rightarrow v_n \quad \Sigma[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash e \Rightarrow v}{\Sigma \vdash f(e_1, \ldots, e_n) \Rightarrow v} \text{ (where the def. of } f \text{ is } f(x_1, \ldots, x_n) \equiv e)$$

$$\frac{\Sigma \vdash e_1 \Rightarrow v_1 \quad \cdots \quad \Sigma \vdash e_n \Rightarrow v_n \quad [\![\Sigma[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash f_{\mathsf{prim}}(x_1, \ldots, x_n)]\!] \overset{\text{udef}}{=} v}{\Sigma \vdash f_{\mathsf{prim}}(e_1, \ldots, e_n) \Rightarrow v}$$

$$\frac{\Sigma \vdash e' \Rightarrow v' \quad \Sigma[x_1 \mapsto v', x_2 \mapsto v'] \vdash e \Rightarrow v}{\Sigma \vdash \mathbf{let}\ \langle x_1, x_2 \rangle = \mathbf{dup}\ e'\ \mathbf{in}\ e \Rightarrow v} \text{ (where } x_1, x_2 \notin \operatorname{dom}\Sigma, x_1 \neq x_2)$$

$$\frac{\Sigma \vdash e_1 \Rightarrow \mathsf{True} \quad \Sigma \vdash e_2 \Rightarrow v_2}{\Sigma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Rightarrow v_2} \qquad \frac{\Sigma \vdash e_1 \Rightarrow \mathsf{False} \quad \Sigma \vdash e_3 \Rightarrow v_3}{\Sigma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Rightarrow v_3}$$

13 / 17

## Backward semantics I

- Backward semantics purpose: Specify new input values corresponding to a modified result.
- Contrast: Forward semantics specifies *values* — backward semantics specifies *environments*.
- The semantics should satisfy:

$$\frac{\Sigma' \vdash e \Rightarrow v'' \qquad v' \sqsubseteq v''}{[\![\Sigma \vdash e]\!] \lhd v' \rightsquigarrow \Sigma'}$$

- Read: *Modifying* the result of evaluating $e$ under $\Sigma$ with the value $v'$ results in an environment $\Sigma'$, *provided* that $e$ evaluates, under $\Sigma'$, to a value $v''$ related to $v'$ by $\sqsubseteq$.
- The ordering $\sqsubseteq$ can be thought of as $=$, but because of the duplication expression it must be modified a little. E.g.:

$$f(x) = \mathbf{let}\ \langle x_1, x_2 \rangle = \mathbf{dup}\ x\ \mathbf{in}\ \langle x_1, x_2 \rangle$$

14 / 17

### Backward semantics II

$$\frac{\Sigma \vdash e \Rightarrow \langle v_1, v_2\rangle \quad [\![\Sigma \vdash e]\!] \lhd \langle v', v_2\rangle \rightsquigarrow \Sigma'}{[\![\Sigma \vdash \mathbf{fst}\ e]\!] \lhd v' \rightsquigarrow \Sigma'} \qquad \frac{\Sigma \vdash e \Rightarrow \langle v_1, v_2\rangle \quad [\![\Sigma \vdash e]\!] \lhd \langle v_1, v'\rangle \rightsquigarrow \Sigma'}{[\![\Sigma \vdash \mathbf{snd}\ e]\!] \lhd v' \rightsquigarrow \Sigma'}$$

$$\frac{[\![\Sigma \vdash e[e_1/x_1, \ldots, e_n/x_n]]\!] \lhd v' \rightsquigarrow \Sigma'}{[\![\Sigma \vdash f(e_1, \ldots, e_n)]\!] \lhd v' \rightsquigarrow \Sigma'} \quad \text{(where the def. of } f \text{ is } f(x_1, \ldots, x_n) \equiv e\text{)}$$

$$\frac{\begin{array}{c}\Sigma \vdash e_1 \Rightarrow v_1 \qquad \cdots \qquad \Sigma \vdash e_n \Rightarrow v_n \\ [\![\Sigma[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash f_{\mathsf{prim}}(x_1, \ldots, x_n)]\!] \lhd v' \overset{\mathsf{udef}}{=} \Sigma' \\ [\![\Sigma \vdash e_1]\!] \lhd \Sigma'(x_1) \rightsquigarrow \Sigma_1 \quad \cdots \quad [\![\Sigma \vdash e_n]\!] \lhd \Sigma'(x_n) \rightsquigarrow \Sigma_n\end{array}}{[\![\Sigma \vdash f_{\mathsf{prim}}(e_1, \ldots, e_n)]\!] \lhd v' \rightsquigarrow \biguplus_{i=1}^{n} \Sigma_i}$$

$$\frac{\begin{array}{c}\Sigma \vdash e' \Rightarrow v \qquad [\![\Sigma[x_1 \mapsto v, x_2 \mapsto v] \vdash e]\!] \lhd v' \rightsquigarrow \Sigma' \\ u = \mathrm{merge}\big(v, \Sigma'(x_1), \Sigma'(x_2)\big) \qquad [\![\Sigma \vdash e']\!] \lhd u \rightsquigarrow \Sigma''\end{array}}{[\![\Sigma \vdash \mathbf{let}\ \langle x_1, x_2\rangle = \mathbf{dup}\ e'\ \mathbf{in}\ e]\!] \lhd v' \rightsquigarrow (\Sigma' \uplus \Sigma'')\big|_{\mathrm{dom}\,\Sigma}}$$

$$\frac{\Sigma \vdash e_1 \Rightarrow \mathsf{True} \quad [\![\Sigma \vdash e_2]\!] \lhd v' \rightsquigarrow \Sigma' \quad \Sigma' \vdash e_1 \Rightarrow \mathsf{True}}{[\![\Sigma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] \lhd v' \rightsquigarrow \Sigma'}$$

$$\frac{\Sigma \vdash e_1 \Rightarrow \mathsf{False} \quad [\![\Sigma \vdash e_3]\!] \lhd v' \rightsquigarrow \Sigma' \quad \Sigma' \vdash e_1 \Rightarrow \mathsf{False}}{[\![\Sigma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] \lhd v' \rightsquigarrow \Sigma'}$$

15 / 17

## Conclusion

16 / 17

### Achievements & Future Goals

- A clean semantics (forward & backwards) of a general purpose functional programming language.
- An inference algorithm that determines whether a given program is bidirectional. (Probably type-based).
- A bidirectionalization algorithm that is *efficient* (using tags, typeinfo etc.) and *correct* (satisfies the semantics, but without necessarily following it directly).
- An implementation and some experimentation.

17 / 17

# Non-Equilibrium Green's Function Calculation for Electron Transport

Stig Skelboe

DIKU

**Abstract**

In mid 2005 a research project including Department of Computer Science, University of Copenhagen, Informatics and Mathematical Modelling, Technical University of Denmark and Atomistix was initiated. The aim of the project is to develop and implement scalable parallel algorithms for nano science simulations. We take the Atomistix Tool Kit (ATK) as a starting point. This presentation outlines the basic principles of ATK, emphasizes the computationally most demandig parts and discusses the possibilities for parallelization and vectorization. In the current approach, the main computational cost is in the evaluation of the discrete Hamiltonian, in the solution of the Poisson equation as part of the computation of the effective potential and finally in the inversion of the matrices required to compute the relevant part of Green's function. There are other demanding computations involved, and in order to assure scalability, as little as possible should be left sequential. With the research project being in a very early phase, most ideas are preliminary and untried.

# Statistical State Estimation

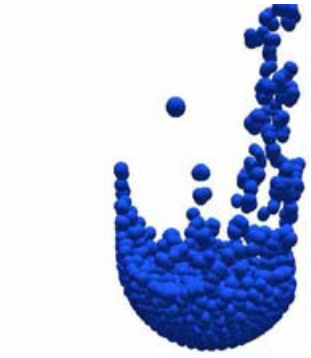
Søren Hauberg
hauberg@diku.dk

Bur first a commercial...

# The Image Group

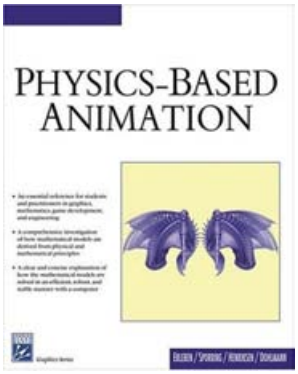
(http://image.diku.dk)

Graphics & Animation



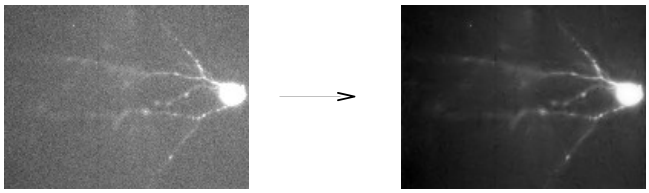
# The Image Group
(http://image.diku.dk)

Graphics & Animation




# The Image Group
(http://image.diku.dk)

Image Processing & Computer Vision



# The Image Group
(http://image.diku.dk)

Robotics



# Statistical State Estimation

Back to the talk...

**Q:** What is 'state'?

**A:** The parameters describing a system that changes continously with time.

**Q:** What is 'state'?

An example of 'state' would be the position of a moving object, such as a robot.

**Q:** If a robot was to tell you its current position what should it answer?

**A:** Latitude: 35.75, Longitude: 139.5?

The only problem with that answer is that we'll never get it right!

**Solution:**
*Be more vague!*

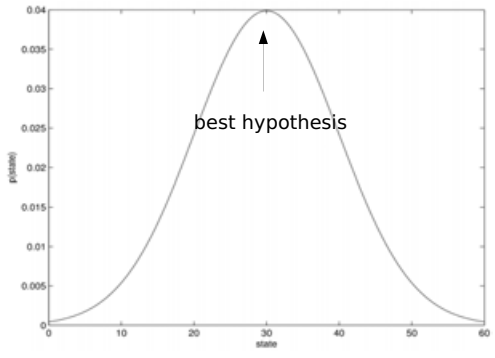
To formalize *vagueness* we use statistics.

('*with high propability my position is latitude: 35.75, longitude: 139.5, and with low propabilty I'm more than 500 km away*')

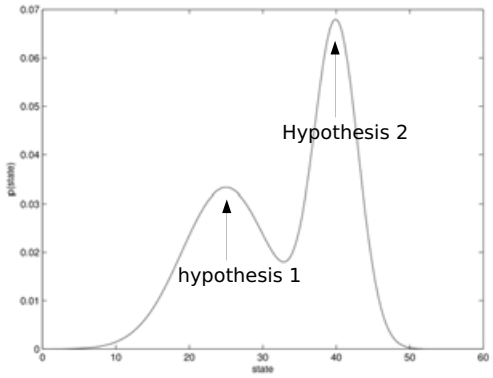
So, where is the robot?

The answer is a propability distribution, that hopefully peaks around the true position.

So, where is the robot?

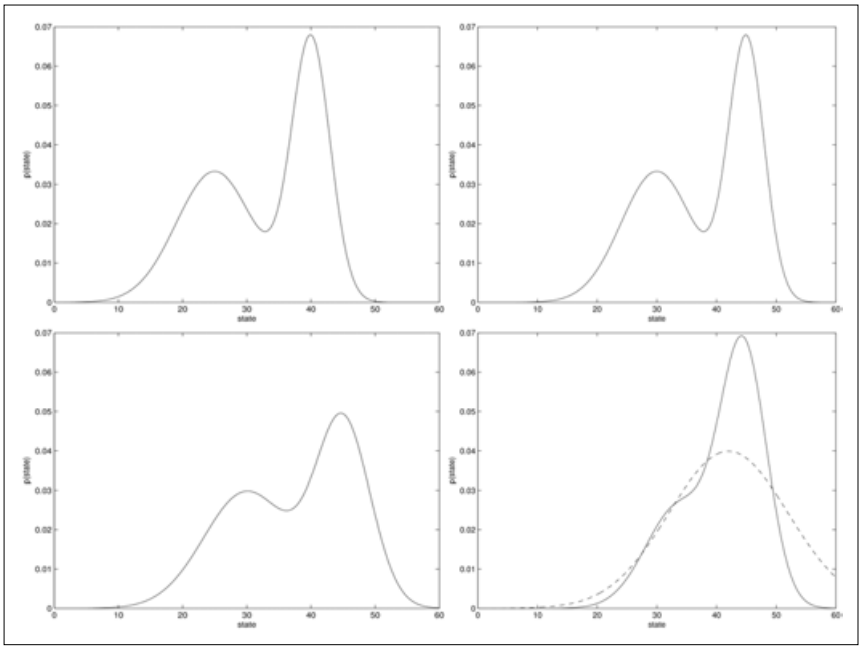


So, where is the robot?

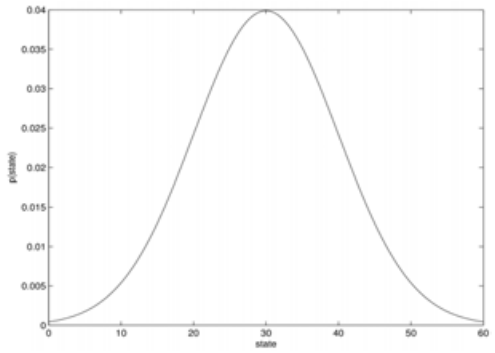



First, let's see the statistical process we want to work with.

Then, we can see how we'll implement it.



If we only want to model a mean and a variance, the principle of least commitment tell's us to use a normal distribution.




When using normal distributions, we have the classic (and excellent) recursive algorithm called *the Kalman Filter.*

However, normal distributions seldom describe the real world.

*(The 'real world' is much more complex)*

However, normal distributions seldom describe the real world.

*The normal distribution only supports* ***one*** *hypothesis.*
*(The distribution only has one top)*

So how do we represent *any* distribution?

*(What kind of data structure?)*

So how do we represent *any* distribution?

*No classic parameterized distribution can be used...*

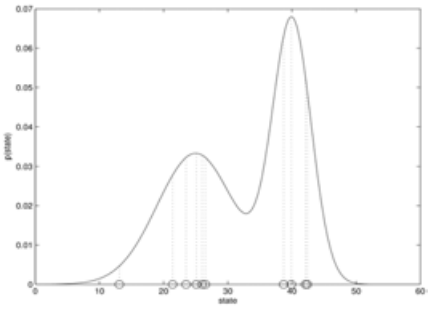
So how do we represent *any* distribution?

*...so we approximate a distribution using weighted samples from the distribution.*

This representation converges to the true distribution.

*(This makes a lot of analysis possible)*

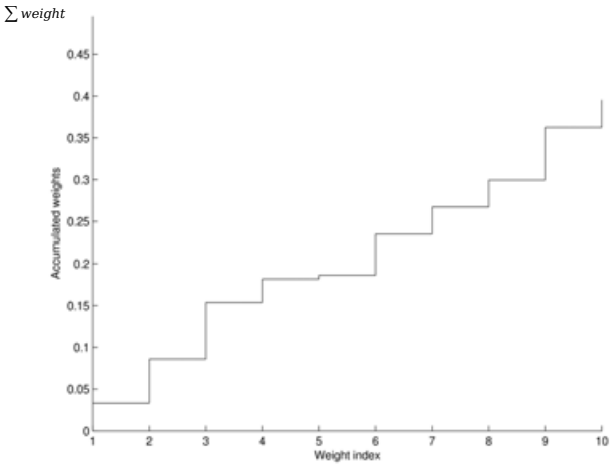
Using this representation, how do we implement the process?




So we need to ba able to draw samples from a distribution

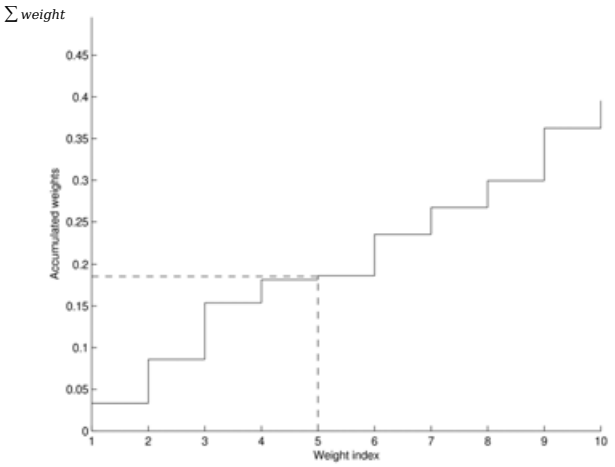
How is this done when the distribution is represented with weighted samples?

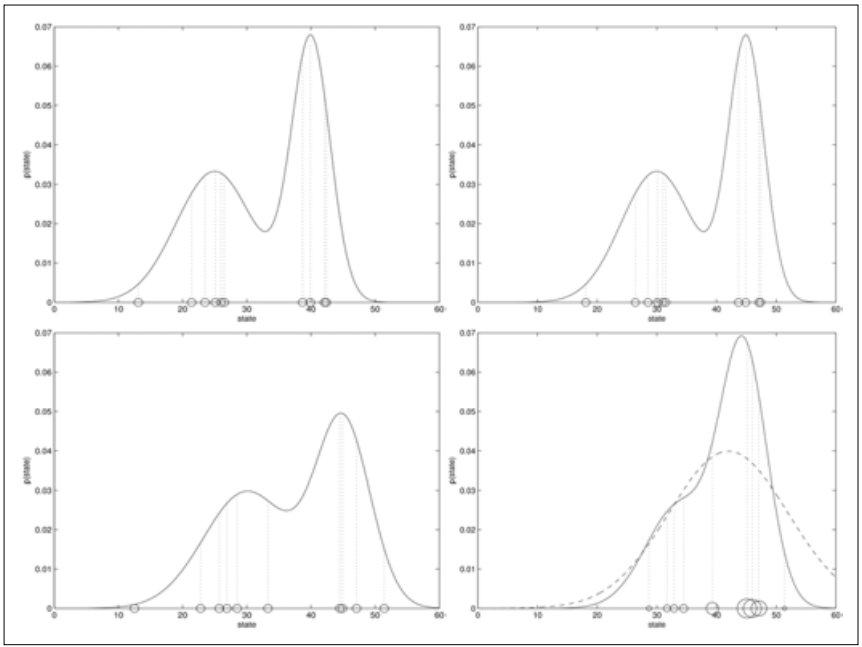
## First, we accumulate the weights...




(Note: this will always be a 1-dimensional function)

## ...then we draw a random sample



This data structure and algorithm is know as the Particle Filter.

## The basics of the Particle Filter:

- We have an amount of guesses.
- We add a random factor to each guess.
- We assign weights to each guess.
- In the next iteration, only the best guesses survive (stochasticly determined).

## An Example

The Particle Filter is used as the basis for the Human-Computer Interaction in a Computer Game

# Robust Routing Algorithm in Unstable Mobile Network

Satoshi Kataoka

IST

**Abstract**

The fundamental technologies of network have recently emerged as a premier research topic. They have great long term economic potential, ability to transform our lives, and pose many new system-building challenges. Some, such as tracking, location and routing are fundamental issues, in that many applications rely on them for needed information. In this paper, we address one of the fundamental problems, namely network routing coverage problem. In the mobile network the calculation power and the transmission speed of the device put on each node are weak. Therefore, a network system must be efficient at communication cost and calculation cost. Moreover, since the mobile network is very unstable, a robust communication algorithm is required of a network system. By combining computational geometry and graph theoretic techniques, specifically the Voronoi diagram and graph search algorithm, we establish the main highlight of the paper — robust mobile network routing.



Available under GNU GPL from http://logiweb.eu/

Klaus Grue

GRD-2006-04-21.UTC+9:15:30:00

## 1 Peano arithmetic

[**Theory** S]

[S **rule** MP: $\Pi\mathcal{A},\mathcal{B}\colon \mathcal{A}\Rightarrow\mathcal{B}\vdash\mathcal{A}\vdash\mathcal{B}$]

[S **rule** Gen: $\Pi\mathcal{X},\mathcal{A}\colon \mathcal{A}\vdash\forall\mathcal{X}\colon\mathcal{A}$]

[S **rule** Ded: $\Pi\mathcal{A},\mathcal{B}\colon \mathrm{Ded}(\mathcal{A},\mathcal{B})\Vdash\mathcal{A}\vdash\mathcal{B}$]

[S **rule** Neg: $\Pi\mathcal{A}\colon \neg\neg\mathcal{A}\vdash\mathcal{A}$]

[S **rule** S2: $\Pi\mathcal{A},\mathcal{B}\colon \mathcal{A}=\mathcal{B}\vdash\mathcal{A}'=\mathcal{B}'$]

[S **rule** S3: $\Pi\mathcal{A}\colon \neg 0=\mathcal{A}'$]

[S **rule** S4: $\Pi\mathcal{A},\mathcal{B}\colon \mathcal{A}'=\mathcal{B}'\vdash\mathcal{A}=\mathcal{B}$]

[S **rule** S5: $\Pi\mathcal{A}\colon \mathcal{A}+0=\mathcal{A}$]

[S **rule** S6: $\Pi\mathcal{A},\mathcal{B}\colon \mathcal{A}+\mathcal{B}'=(\mathcal{A}+\mathcal{B})'$]

[S **rule** S7: $\Pi\mathcal{A}\colon \mathcal{A}\cdot 0=0$]

[S **rule** S8: $\Pi\mathcal{A},\mathcal{B}\colon \mathcal{A}\cdot(\mathcal{B}')=(\mathcal{A}\cdot\mathcal{B})+\mathcal{A}$]

[S **rule** S1: $\Pi\mathcal{A},\mathcal{B},\mathcal{C}\colon \mathcal{A}=\mathcal{B}\vdash\mathcal{A}=\mathcal{C}\vdash\mathcal{B}=\mathcal{C}$]

[S **rule** S9: $\Pi\mathcal{X},\mathcal{A},\mathcal{B},\mathcal{C}\colon \langle\mathcal{B}{\equiv}\mathcal{A}|\mathcal{X}{:=}0\rangle\Vdash\langle\mathcal{C}{\equiv}\mathcal{A}|\mathcal{X}{:=}\mathcal{X}'\rangle\Vdash\mathcal{B}\vdash\mathcal{A}\Rightarrow\mathcal{C}\vdash\mathcal{A}$]

## 2 Example lemmas

[S **lemma** Prop 3.2a: $\Pi\mathcal{A}\colon \mathcal{A}=\mathcal{A}$]

[S **lemma** Prop 3.2b: $\Pi\mathcal{A},\mathcal{B}\colon \mathcal{A}=\mathcal{B}\vdash\mathcal{B}=\mathcal{A}$]

[S **lemma** Prop 3.2c: $\Pi\mathcal{A},\mathcal{B},\mathcal{C}\colon \mathcal{A}=\mathcal{B}\vdash\mathcal{B}=\mathcal{C}\vdash\mathcal{A}=\mathcal{C}$]

[S **lemma** Prop 3.2d: $\Pi\mathcal{A},\mathcal{B},\mathcal{C}\colon \mathcal{A}=\mathcal{C}\vdash\mathcal{B}=\mathcal{C}\vdash\mathcal{A}=\mathcal{B}$]

[S **lemma** Prop 3.2e: $\Pi\mathcal{A},\mathcal{B},\mathcal{C}\colon \mathcal{A}=\mathcal{B}\vdash\mathcal{A}+\mathcal{C}=\mathcal{B}+\mathcal{C}$]

[S **lemma** Prop 3.2f: $\Pi\mathcal{A}\colon \mathcal{A}=0+\mathcal{A}$]

[S **lemma** Prop 3.2g: $\Pi\mathcal{A},\mathcal{B}\colon \mathcal{A}'+\mathcal{B}=(\mathcal{A}+\mathcal{B})'$]

[S **lemma** Prop 3.2h: $\Pi\mathcal{A},\mathcal{B}\colon \mathcal{A}+\mathcal{B}=\mathcal{B}+\mathcal{A}$]

## 3 A lemma and a proof

[S **lemma** Prop $3.2f_2$: $\Pi\mathcal{A}\colon \mathcal{A}=0+\mathcal{A}\Rightarrow\mathcal{A}'=0+\mathcal{A}'$]

S **proof of** Prop $3.2f_2$:

| | | | |
|---|---|---|---|
| L01: | Arbitrary $\gg$ | $\mathcal{A}$ | ; |
| L02: | Block $\gg$ | Begin | ; |
| L03: | Arbitrary $\gg$ | $\mathcal{A}$ | ; |
| L04: | Premise $\gg$ | $\mathcal{A}=0+\mathcal{A}$ | ; |
| L05: | S2 $\triangleright$ L04 $\gg$ | $\mathcal{A}'=(0+\mathcal{A})'$ | ; |
| L06: | S6 $\gg$ | $0+\mathcal{A}'=(0+\mathcal{A})'$ | ; |
| L07: | Prop 3.2d $\triangleright$ L05 $\triangleright$ L06 $\gg$ | $\mathcal{A}'=0+\mathcal{A}'$ | ; |
| L08: | Block $\gg$ | End | ; |
| L09: | Ded $\triangleright$ L08 $\gg$ | $\mathcal{A}=0+\mathcal{A}\Rightarrow\mathcal{A}'=0+\mathcal{A}'$ | □ |

# Towards Consitent Protocol-based Agent Interactions

José Ghislain Quenum
joque@nii.ac.jp

National Institute of Informatics, Tokyo, Japan

April 21$^{st}$, 2006





## Outline

1 Context and Motivations

2 Abstract Models

3 Unification and Adaptation

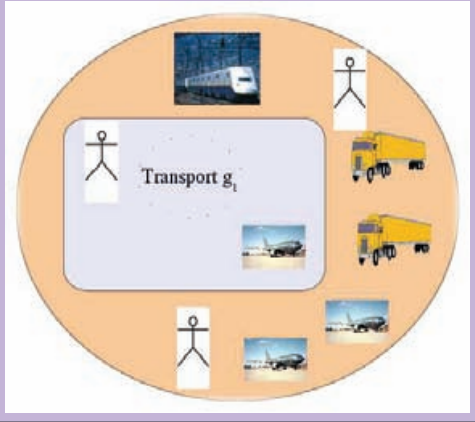
4 Conclusion





## Example

### TranspMAS

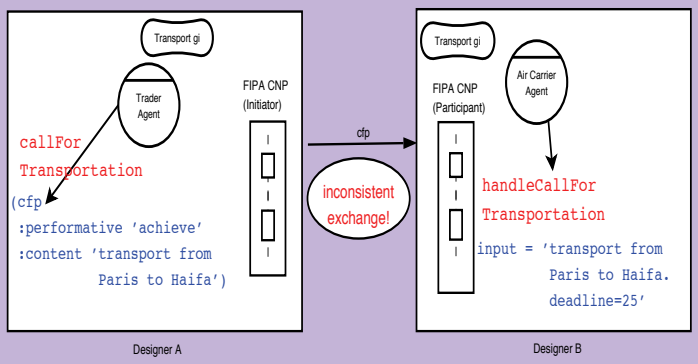



### Agents:

- $T_t$
- $AC_t$
- $RC_t$
- $LC_t$

## Generic Protocol Configuration

Manual configuration of generic protocols ⇨ inconsistent interactions








## Some Related Research

### Protocol Engineering and Design

- [Koning and Oudeyer, 2001] (Protocol Specification)
- [Paurobally and Cunningham, 2003] (Protocol Specification)
- [Jouvin, 2003] (Dynamic Role Delegation)

### Positionning our approach

- Generic protocols
- Protocol/Role configuration
- clean seperation in agent architecture



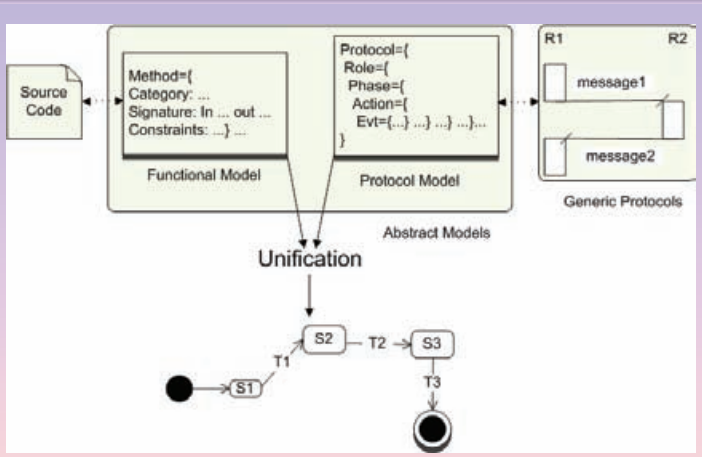


## Assumptions

- Openness and Heterogeneity
- Interaction protocol specifications
- Interactions through message exchange

# Protocol configuration

## Process







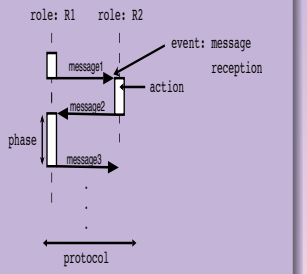

# Generic Protocol Abstract Model

## Concepts

- Protocol: $p \stackrel{\text{def}}{=} < \Theta, R, M, \Omega >$
- Role: $r \stackrel{\text{def}}{=} < \Theta_r, \Pi, A_g, V >, r \in R$
- Event: an atomic change
- Action: $a \stackrel{\text{def}}{=} < \nu, \Sigma, E >$
- Phase: $P_h \stackrel{\text{def}}{=} < A, \prec >$
- $\Omega = < r_1 \xrightarrow[a_1]{m_1} r_2, r_2 \xrightarrow[a_2, m_1]{m_2} r_1, r_1 \xrightarrow[a_3, m_2]{m_3} r_2 >$

View Specs

## Graphical Illustration








# Agent Functional Model

## Definitions

- Functional Model: $f \stackrel{\text{def}}{=} \{m_i\}$
- Method: $m_i \stackrel{\text{def}}{=} < \nu, \Sigma, C >$

## Configured Protocol Model

### Mixture of generic protocol & functional models

- user-defined data types
- No more concept of phase
- explicit scope for actions





## Unifying Actions and Methods

### Algorithm

1. Construct action graph $G_a$ View Graph
2. Construct method graph $G_m$ View Graph
3. Unify actions in $G_a$ and methods in $G_m$:
   - Category
   - Signature
   - Constraints/events





## Unifying Actions and Methods (Cont.)

### Matching rules

Let $\mathcal{M}$ be the set of methods paired with action $a_t$
Let $\mathcal{A}$ be the set of unpaired actions

- $\mathcal{M} = \{m_0\} \Rightarrow a_t \equiv m_0$
- $\mathcal{M} = \{m_0, m_1, \ldots, m_k\} \Rightarrow \textit{conflict}$
- $\mathcal{M} = \emptyset \Rightarrow a_t \in \mathcal{A}$

## Adapting Unpaired Actions

**Methodology**

Derive method specifications basing on that of the original action

**Three stages:**

- Category
- Signature: action signature, action graph & method graph
- Constraints: action graph

Illustrate





## Compiling into FSA

**Rule**

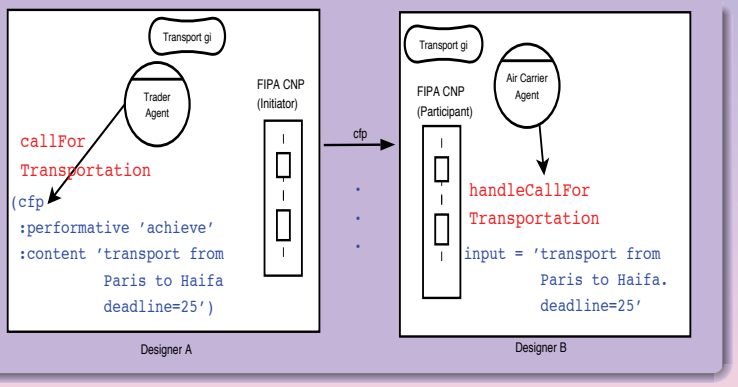
*Action $a_k$ is a transition $t_{ij}$ from $S_i$ to $S_j$ iff:*

source state *(1) $a_k$ has no predecessor: $S_i$ is an initial state*
*(2) $a_k$ follows $a_{k-1}$, transition to $S_i$*

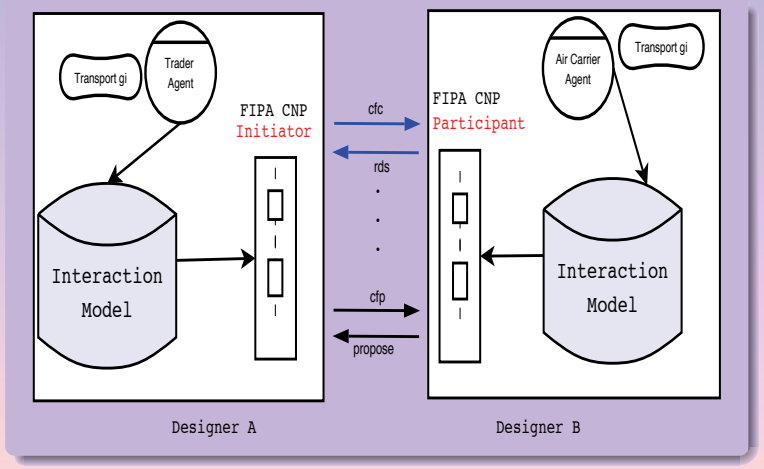
destination state *(1) $a_k$ precedes $a_{k+1}$, transition from $S_j$*
*(2) $a_k$ has no successor and $S_j$ is a final state*







## Towards Dynamic Protocol Selection








**Anything else?**

The End!!!



## Further Reading

D. Jouvin
Role Delegation and Dynamic Architectures in Conversational Multi-Agent Systems.
Ph.D Thesis, 2003.

J-L Koning and P-Y Oudeyer.
Introduction to pos: A protocol operational semantics.
International Journal on Cooperative Information Systems, 10(12):101–123, 2001.
Special Double Issue on Intelligent Information Agents: Theory and Applications.

## Further Reading (cont.)

Paurobally S. and Cunningham J.
Achieving common interaction protocols in open agent environments.
In Proceedings of AAMAS'03 Agentcities Workshop: challenges in open agents environments, Melbourne, Australia, August 2003.



## Specifying the Contract Net Protocol

### Contract Net Protocol

(protocol (protocolproperties (protocoldesc ident='cnpprot' title='ContractNet' location='KqmlCnp.xml') (protocolattr class='Request' participantcount='1' return='operationresult) (protkeyws '...')) (roles ...) (messagepatterns ...))



## Specifying the Contract Net Protocol (Cont.)

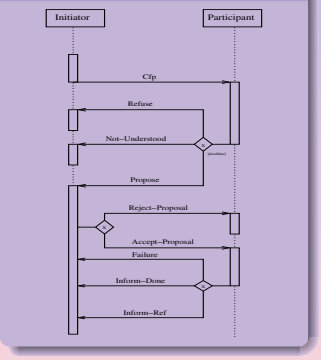
### Initiator Role

(role ident='initiator' (roleproperties (roledescriptors ident='initiator' name='Initiator') (roleattributes cardinality='1')) (variables (variable ident='bidsCol' type='collection') (variable ident='deliberations' type='map') (variable ident='deadline' type='date')) (actions(action category='compute' description='Deliberate' (signature (arg type='date' dir='in') (arg type='collection' dir='in')(arg type='map' dir='out')) (events (event type='change' dir='in' object='deadline' ident='evt0') (eventref dir='in' ident='evt5') (event type='change' dir='out' object='deliberations' ident='evt1')))) (phases ...))

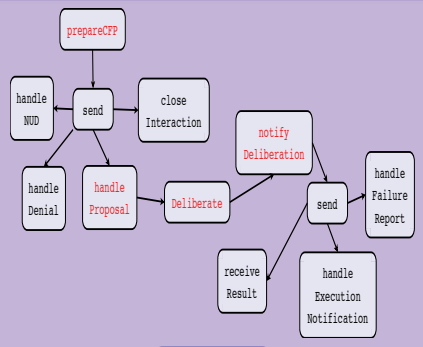
Back to Protocol

## Unification algorithm - Illustration

### CNP Sequence Diagram



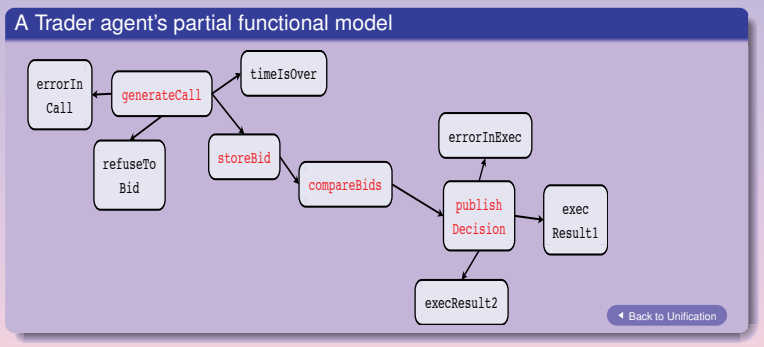

### Initiator role's action graph






## Unification algorithm - Illustration

### A Trader agent's partial functional model






## Unification algorithm - Illustration

### Action: PrepareCFP

```
Category = 'compute'
Signature =
(in = 'date')
(out = 'any')
```

### Method: generateCall

```
Category = 'compute'
Signature =
(in = 'date')
(out = 'CallForProposal')
```

- *CallForProposal*: subtype of *any*

## Unification algorithm - Illustration

**Action: PrepareCFP**

```
Category = 'compute'
Signature =
(in = 'date')
(out = 'any')
Events =
(in = 'variablecontent')
(out = 'messagecontent')
```

**Method: generateCall**

```
Category = 'compute'
Signature =
(in = 'date')
(out = 'any')
Constraints = {}
```

Back to Unification



## Adaptation algorithm - Illustration

**Action: Deliberate**

```
Category = 'compute'
Signature =
(in = 'date',
      'collection')
(out = 'map')
```

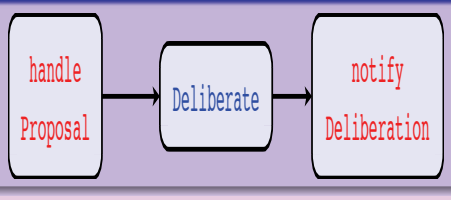
**Method: compareBids**

```
Category = 'compute'
Signature =
(in = '?', '?')
(out = '?')
```



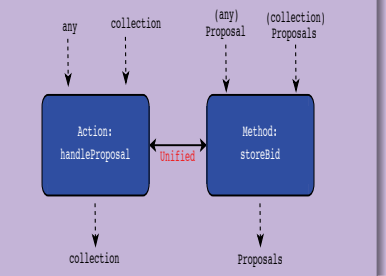
## Adaptation algorithm - Illustration

**Initiator role's action graph (partial)**

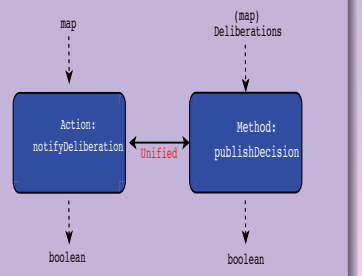


## Adaptation algorithm - Illustration

### handleProposal ≡ storeBid




### notifyDeliberation ≡ publishDecision






## Adaptation algorithm - Illustration

### Action: Deliberate

```
Category = 'compute'
Signature =
(in = 'date',
      'collection')
(out = 'map')
```

### Method: compareBids

```
Category = 'compute'
Signature =
(in = 'date', 'Proposals')
(out = 'Deliberations')
```



## Adaptation algorithm - Illustration

### Action: Deliberate

```
Category = 'compute'
Signature =
(in = 'date',
      'collection')
(out = 'map')

Events =
(in = 'change', 'change')
(out = 'change')
```

### Method: compareBids

```
Category = 'compute'
Signature =
(in = 'date', 'Proposals')
(out = 'Deliberations')

Constraints =
(status = 'executed'
  mth='storeBid')
```

Back to Adaptation

# Model Checking Networked Applications

**Cyrille Artho** [1]
cartho@nii.ac.jp

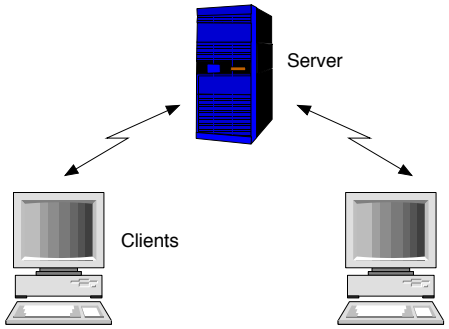
Armin Biere [2], Boris Zweimüller [3],
Pierre-Loïc Garoche [4], Christian Sommer [3], Shinichi Honiden [1]

[1] National Institute of Informatics, Tokyo, Japan
[2] Johannes Kepler Universität, Linz, Austria
[3] ETH Zürich, Zurich, Switzerland
[4] Institut de Recherche en Informatique de Toulouse, France

04/27/2006



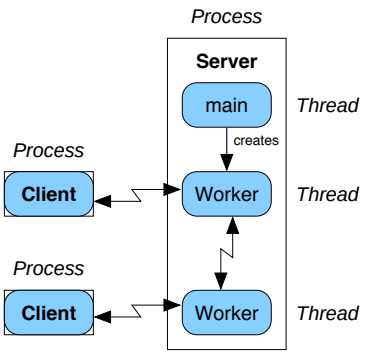
## Networked programs




- Multiple separate processes.
- Most SW model checkers can only handle a **single** process!

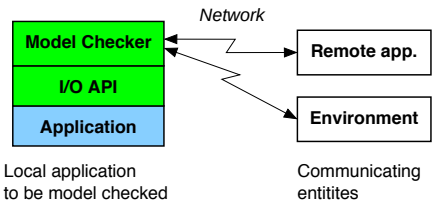


## Example application: chat server




- Main thread creates worker threads.
- Worker threads handle client conn.
- One worker thread per client.
- Example: 5 threads, 3 processes.

## Network communication




Model checker repeats I/O operations during state space exploration!

- Output of local application is sent several times.
- Input is expected several times.
- Communication with external applications won't quite work this way. . .



## Possible solutions

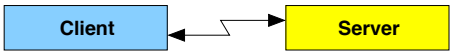
1. Stubs (prev. work)
2. Special I/O caching (new)
3. Centralization (extended)



## Stubs

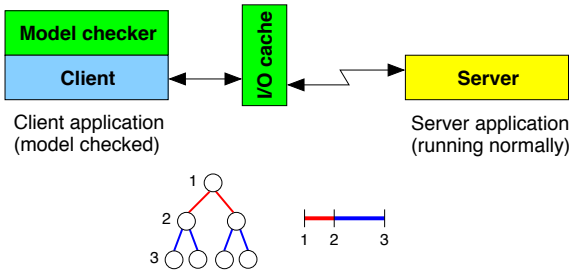
- Write (application-specific) stub for each I/O operation of each application.
- Model check each application individually.




(Conventional) test combining client and server

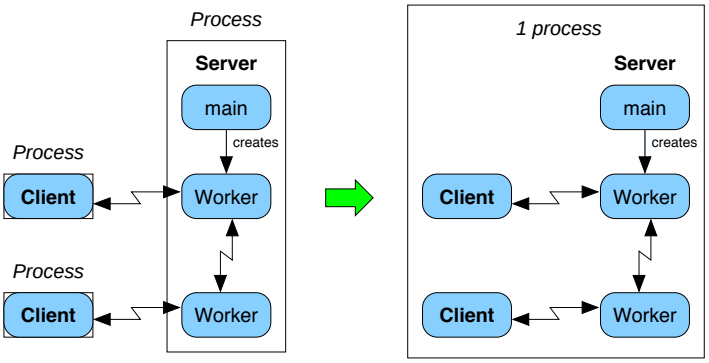
## MC-aware I/O caching




- Each I/O operation is captured by (MC-aware) cache layer.
- Duplicate send operations: „ignored” (not relayed to server).
- Duplicate read operations: previous output from server is replayed.
- Novel approach, work in progress.



## Centralization

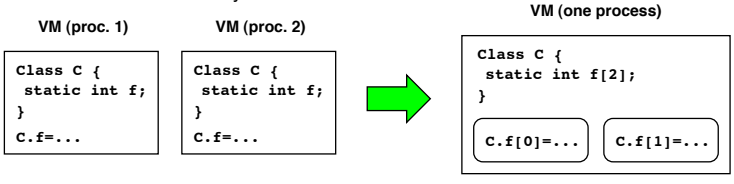



- All processes are merged into one process.
- Normal software model checker can be applied.



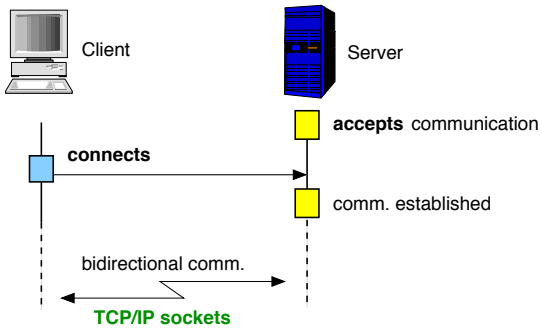
## Program transformations (after compilation)

1. Convert processes into threads (by wrapper thread).
2. Convert static fields to array of static fields.




3. Convert static synchronization (usage of Class descriptors as locks).
   Very similar to conversion of static fields.
4. Process shutdown.
   - Shutdown semantics (close files/sockets).
   - `System.exit` should not kill host VM, but only current centralized „process”.

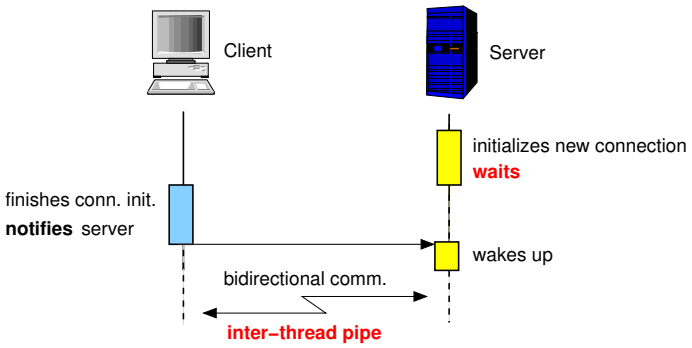
## Network model




- Blocking mechanism for accept/connect.
- Bidirectional, buffered communication channel.



## Centralized networking




- wait/notify: blocking call.
- Pipe: transparent replacement of transportation mechanism.



## Comparison

| Approach | Advantage | Disadvantage |
|---|---|---|
| Stubs | simple | limited automation/precision |
| I/O caching | scalable | requires certain program properties |
| Centralization | complete | large state space |

- Centralization is the only approach that currently works „out of the box”.

No “silver bullet”

## Conclusion

- Networked applications cannot be model checked normally.
- Several approaches possible: Centralization models all possibilities.
- Encouraging results with initial experiments.

If model checkers cannot handle I/O, we can make I/O handle model checking!



## Work in progress and future work

- Communication failures.
- Shutdown behavior.
- Alternatives to centralization.



## Modeling network failures

- Non-deterministic exception for each I/O operation.
- Problem: **Spurious deadlocks:**
  Method `connect` on the client side fails.
  Server may wait indefinitely for client.
- Special `Search` class suppresses such deadlocks.

# Interprocedual Program Analysis for Java based on Weighted Pushdown Model Checking

Mizuhito Ogawa (JAIST)
Collaborated with Xin Li
2006.4.21

## Program analysis as model checking

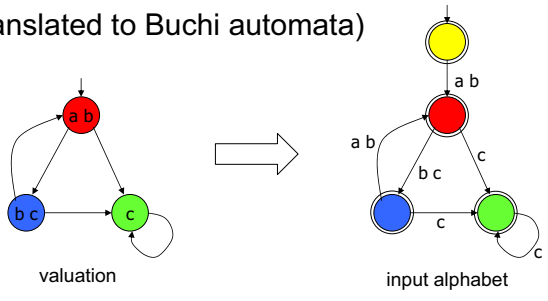
- Program analysis = Abstraction + MC
  - Steffen 1991, Schmidt 1997
  - practical language (e.g., Java, C, ML) is large.

Enables us to implement real analyzer with reasonably small cost

- Program analysis implementation
  = InterMediate Language + Abstraction + MC
  - IML is usually smaller (e.g., Jimple, …).
  - use existing tools for IML translation and MC.

## Model checking

- Find all states $s$ of $M$ such that $M,s \models \psi$ and check whether initial states are among these.
- Model $M$ : State transition system
  (→ translated to Buchi automata)



## (Finite) Model checking algorithm

- Model / specification (e.g., CTL*) can be translated to (Buchi) automata, $M$ and $S$.
- Model holds Spec. $\Leftrightarrow L(M) \subseteq L(S)$
  $\Leftrightarrow L(M) \cap L(S)^c = \phi$
- This holds even if $M$ is a pushdown system, by translation to a pushdown automaton.

## Pushdown system (PDS)

- **Def.** PDS $(P, \Gamma, \Delta, p_0, \omega)$ where
  $P$ : finite set of control locations
  $\Gamma$ : stack alphabet
  $\Delta \subseteq (P \times (\Gamma \cup \{\omega\})) \times (P \times (\Gamma^*))$ : transition
  $q_0$ : initial control location
  $\omega$: bottom stack symbol
- $<p,w> \in P \times \Gamma^*$ is called *configuration*
- $\nu : AP \rightarrow 2^{P \times \Gamma^*}$ is called *valuation*

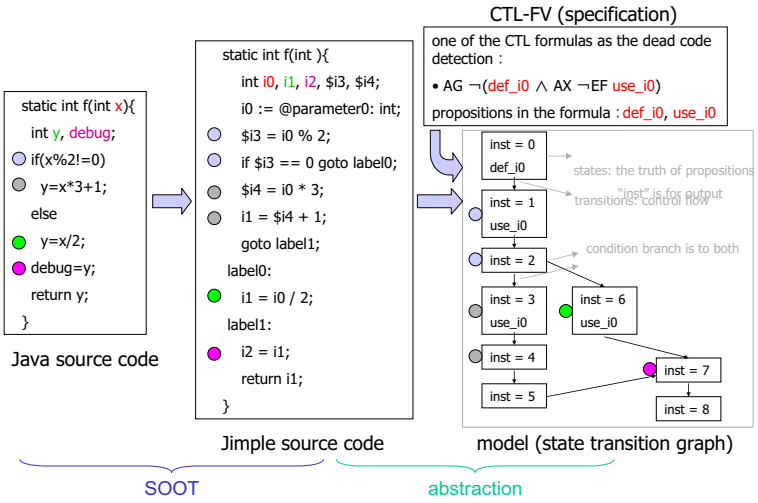
## Valuation $\nu : AP \rightarrow 2^{P \times \Gamma^*}$

- For general valuation, model checking is *undecidable*.
- Simple valuation : only topmost symbol in a stack is related (i.e., $\nu(\psi) = \{<q, \alpha> \mid q \in P, \alpha \in \Gamma\}$)
- Pushdown system M to (Buchi/Muller) PDA
  input alphabet $\Sigma = 2^{AP}$
  transition label at $<p, \alpha> = \nu^{-1}(<p, \alpha>)$

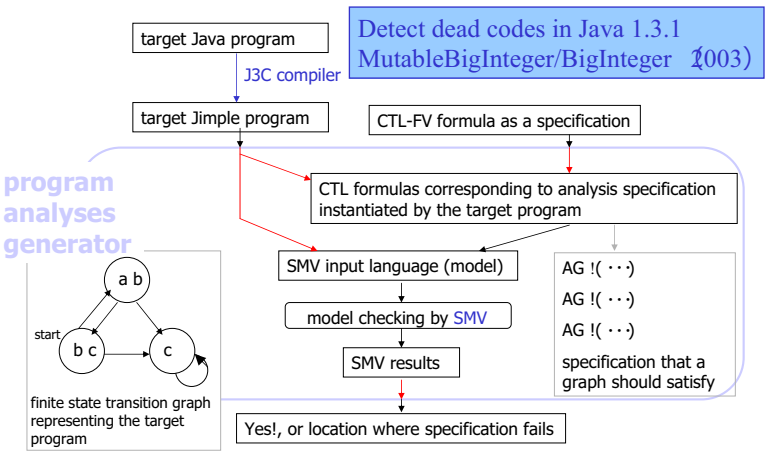
# Java program analysis implementation

- Java PA implementation = IML + Abstraction + MC
- Intraprocedual : SOOT + 700lines + SMV
  Dead code detection ($\neg use_x$ **W** $def_x$) in CTL-FV
  Design 2 months, Implementation 1 month (M2, 2003)
- Interprocedual : SOOT + 1500lines + WeightedPDS
  Useless code detection (PER-based abstraction)
  Design 8 months, Implementation 2 weeks (D2, 2006)

# IML translation and model generation




# Prototype Java analyzer on SMV

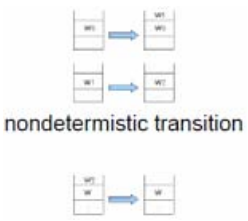


# Weighted PDS (T.Reps, et.al. 1996)

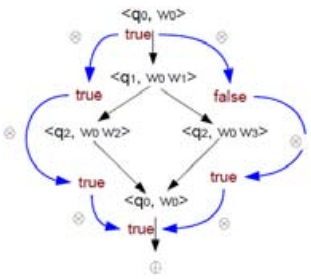
- **Def.** Bounded idempotent semiring $(D,+,\times,0,1)$ with $a \leqq b$ iff $a+b = a$
- **Def.** Weighted PDS = (P, D, f) where $f: \Delta \rightarrow D$ for PDS $P = (Q, \Gamma, \Delta, p_0, \omega)$
- Library available for generalized reachablity. (Weighted PDS, WPDS+)
- **Intuition.** Weight (in D) is a function (dataflow)
  - $D = [Abs \rightarrow Abs] \cup 0$ (error/inconsistency)
  - $\times$: composition
  - $+$: meet of paths, defined by $\leqq$ with $0$ as maximum

# Example of weighted PDS

$\langle q_0, w_0 \rangle \hookrightarrow \langle q_1, w_0 w_1 \rangle$
$\langle q_1, w_1 \rangle \hookrightarrow \langle q_2, w_2 \rangle$
$\langle q_1, w_1 \rangle \hookrightarrow \langle q_2, w_3 \rangle$
$\langle q_2, w_3 \rangle \hookrightarrow \langle q_0, \epsilon \rangle$
$\langle q_2, w_2 \rangle \hookrightarrow \langle q_0, \epsilon \rangle$



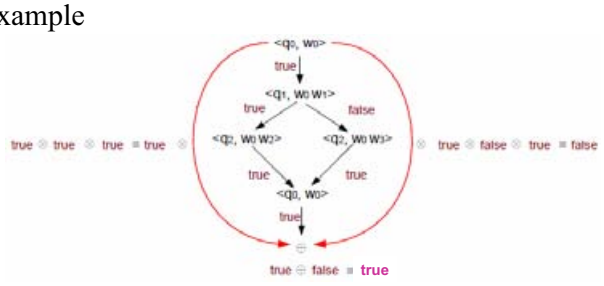

TRUE
TRUE
FALSE
TRUE
TRUE



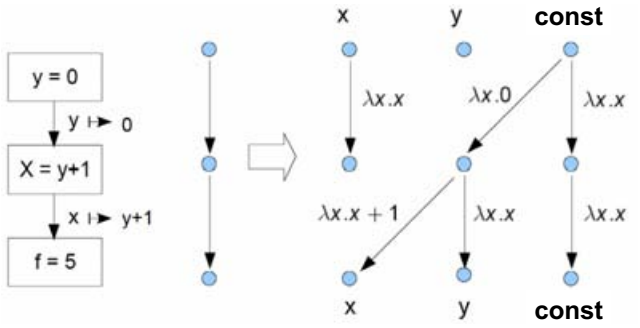

- TRUE (=1): say, propagate
- FALSE (=0): say, not propagate
- $\times = \wedge$, $+ = \vee$

# Generalized Reachablity

- Let $c \in P \times \Gamma^*, C \subseteq P \times \Gamma^*$
  s.t. $\delta(c,C) = +_{\sigma \in path(c, c'), c' \in C} f_n \times \ldots \times f_1$
  where $f_i$ is weight associated to $r_i$ with $\sigma = r_1 \ldots r_n$
- Example



# Encoding by exploded supergraph

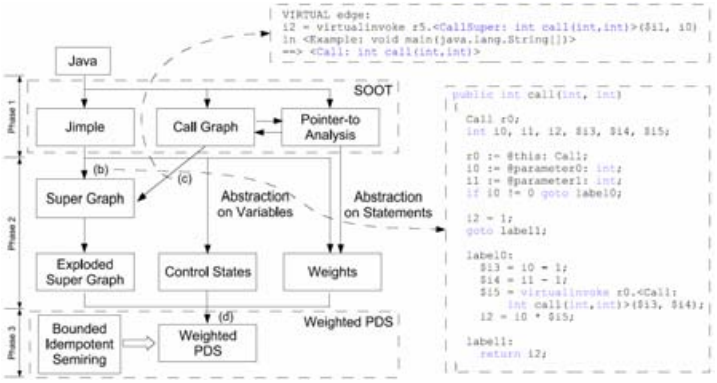



- Control location = variable
- Stack = (program counter ∪ return address)*
- Analysis is encoded as generalized reachability.

No encoding for *environment*

# Java useless code detection by WPDS

- PER abstraction ( ANY= {(x,y)}, D= {(x,x)}, Hunt 1991) :
  if ANY on a code does not affect the result (e.g., ID), then *useless.*




# Conclusion

- Case study on useless code detection implementation for Java by SOOT and Weighted PDS.
  (No evaluation / detailed experiments yet)

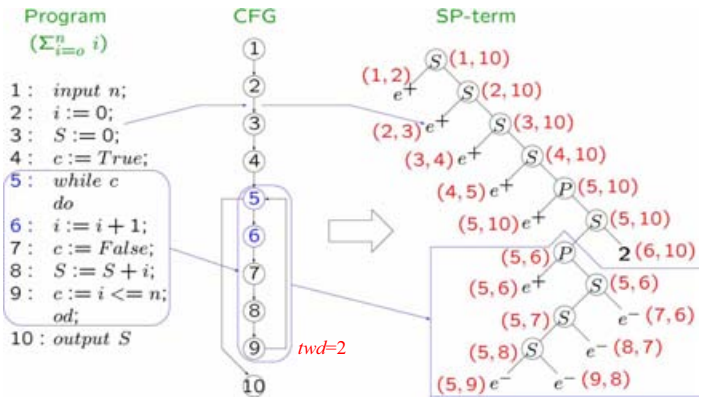
- Related work: Many for finite MC (e.g.,Bandera),
  Few for PDS MC (e.g., jMoped)

- Needs more case studys
  - call graph generation in SOOT is very slow.
  - call graph (exploded supergraph) generation
    by pushdown model checking ?

# Future Direction

- ML analyses
  - SML# compiler, collaboration with Ohori (Tohoku)
- Clarify PDS MC as abstract interpretation engine
  - Forward case seems OK. Backward?
- Scalable algorithm for Weighted PDS MC ?
  - Most of CFG is well-structured (bounded tree-width);
    Use of *iterative-free approach* (ICFP03)?

# CFG representation by SP term (*twd* =2)



# Generic Sorting Multiset Discriminators

## How to sort complex data in linear time

Fritz Henglein

Department of Computer Science
University of Copenhagen
Email: henglein@diku.dk

2nd DIKU-IST Workshop, Shonan Village, April 21-22, 2006







## Outline

1 Detour: My research interests
2 Sorting reloaded
3 Multiset discrimination
4 Sorted discrimination
5 Conclusion of talk







## Research interests

- Foundations of programming languages (algorithms, complexity, semantics)
- Program analysis (specifically type-based analysis)
- Applications of programming language technology



## Current projects

APPL: Applications and Principles of Programming Languages
- Duration: 2006-2009
- Topics: Novel language features, program inversion and specialization, domain-specific languages, computational logic

Plan-X: Software platform for value-oriented programming supporting transparency of distibution and mobility of data (P2P storage, routing, data synchronization, distributed XML technology)







## Current projects

3gERP: 3d generation enterprise resource planning systems
- Goal: Software technology for rapid development and configuration of shrink-wrap ERP systems
- Duration: 2006-2010
- Partners: Copenhagen Business School, Microsoft
- Upcoming 4 Ph.D. projects at DIKU: language-based security, formal process modeling, automatic incrementalization (real-time reporting), dynamic updating software updating







## Sorting algorithms

- Quicksort
- Mergesort
- Heapsort
- Insertion sort
- Bubble sort
- Bitonic sort
- Shell sort
- Zero-one mergesort
- AKS sorting network
- Bucket sort
- Radix/lexicographic sort
- . . .



## Myths and facts

Everybody knows: Sorting requires $O(n \log n)$. $O(n \log n)$ what? And does it really require that? Facts:

1. Given abstract total preorder (order) $(T, \leq)$, any sorting algorithm requires $\Omega(m \log m)$ applications of the comparison operator $\leq$ to sort an input of $m$ elements of type $T$.
2. There exist algorithms that, given any $(T, \leq)$, sort $m$ inputs using $O(m \log m)$ applications of the comparison operator.
3. Fact 1 does not imply that sorting requires $\Omega(n \log n)$ time where $n$ is the size of the input. $O(n)$ sorting algorithms for a large number of concrete orders exist (remainder of talk).
4. Fact 2 does not imply that those algorithms necessarily execute in worst-case time $O(n \log n)$ for non-constant size input elements. None of them do.







## Time complexities reconsidered

Assume $(T, \leq)$ such that time complexity of executing $x \leq y$ is $\Theta(|x| + |y|)$. Input: $[x_1, \ldots, x_m]$ of size $n = \Sigma_{i=1}^{n} |x_i|$.

- Quicksort: $\Theta(n^2)$ ($O(n \log n)$ randomized?!)
- Mergesort: $\Theta(n^2)$
- Heapsort: $\Theta(n^2)$
- Selection sort: $\Theta(n^3)$
- Insertion sort: $\Theta(n^2)$
- Bubble sort: $\Theta(n^2)$
- Bitonic sort: $\Theta(n \log^2 n)$
- Shell sort: $\Theta(n \log^2 n)$
- Zero-one mergesort: $\Theta(n \log^2 n)$
- AKS sorting network: ? (uniformly constructible?)
- Bucket/counting sort: not comparison-based
- Radix/lexicographic sort: not comparison-based







## Time complexities reconsidered

Proof ideas:

1. Consider one element of size $\Theta(n)$, the rest of size $O(1)$. How many comparisons performed on that one element?
2. Algorithm as sorting network: Maximum depth is upper bound on number of comparisons on each element.



## Paritioning and discrimination

Partitioning: Given concrete equivalence relation $(T, \cong)$, partition input $[x_1, \ldots, x_m]$ into $\cong$-equivalence classes.

Discrimination: For input $[(x_1, y_1), \ldots, (x_m, y_m)]$ partition the $y_i$ according to the corresponding $x_i$.

Examples:
partition Anagram ["AbC", "aaC", "BcA"] = [["AbC", "BcA"], ["aaC"]]
partition Char ['c', 'a', 'c'] = [['c', 'c'], ['a']]
discriminate Anagram [("AbC", 1), ("aaC", 2), ("BcA", 3)] = [[1, 3], [2]]
discriminate Char [('c', true), ('a', true), ('c', false)] = [[true, false], [true]]







## Language of typed equivalence relations

$$\frac{}{\text{Char}:\text{Equiv Char}} \qquad \frac{}{\text{Int } n:\text{Equiv Int}}$$

$$\frac{}{\text{Unit}:\text{Equiv Unit}} \qquad \frac{E_1:\text{Equiv } T_1 \quad E_2:\text{Equiv } T_2}{E_1 + E_2:\text{Equiv } (T_1 \times T_2)}$$

$$\frac{E_1:\text{Equiv } T_1 \quad E_2:\text{Equiv } T_2}{E_1 \times E_2:\text{Equiv } (T_1 \times T_2)} \qquad \frac{[X:\text{Equiv } T]E:\text{Equiv } T}{\mu X.E:\text{Equiv } T}$$

$$\frac{E:\text{Equiv } T}{\text{Bag}(E):\text{Equiv } (\text{Seq}(T))} \qquad \frac{E:\text{Equiv } T}{\text{Set}(E):\text{Equiv } (\text{Seq}(T))}$$







## Definable equivalence relations

$$\begin{aligned} \text{IgnoreCase} &= \text{Map toUpper Char} \\ \text{Anagram} &= \text{Bag}(\text{IgnoreCase}) \\ \text{Seq}(E) &= \mu X.\text{Unit} + (E \times X) \end{aligned}$$



## Linear-time discrimination

**Theorem (Top-down MSD)**

*For each derivable $E : \mathrm{Equiv}\ T$ there exists a discriminator that executes in worst-case linear time. The discriminators can be defined by induction on $E$.*

Linearity for top-down MSD only holds for unshared (unboxed) data (sequences, not lists with shared tails; trees, not dags). Linear time performance can be achieved for shared, acyclic data using bottom-up MSD.
$O(n \log n)$ performance can be achieved for shared, cyclic data (using different algorithmic strategy).







## Observations

- The equivalence relation combinators define (standard) orders if their arguments are (standard) orders; e.g., $O_1 \times O_2$ defines lexicographic ordering on pairs for given orderings $O_1, O_2$ on their components. $\mathrm{Seq}(O)$ defines lexicographic ordering on sequences for given ordering $O$ on elements.
- A (top-down) discriminator sort their input if the discriminators it calls on does so return the equivalence classes in the thus defined order:
- If the base discriminators (for atomic types) sort then all the discriminators sort.
- For base types (characters, fixed-size integers) we have linear-time sorting algorithms: bucket sort, radix sort.







## Language of typed orders

$$\frac{}{\mathrm{Char}:\mathrm{Order}\ \mathrm{Char}} \qquad \frac{}{\mathrm{Int}\ n:\mathrm{Order}\ \mathrm{Int}}$$

$$\frac{}{\mathrm{Unit}:\mathrm{Order}\ \mathrm{Unit}} \qquad \frac{O_1:\mathrm{Order}\ T_1 \quad O_2:\mathrm{Order}\ T_2}{O_1+O_2:\mathrm{Order}\ (T_1 \times T_2)}$$

$$\frac{O_1:\mathrm{Order}\ T_1 \quad O_2:\mathrm{Order}\ T_2}{O_1 \times O_2:\mathrm{Order}\ (T_1 \times T_2)} \qquad \frac{[X:\mathrm{Order}\ T]O:\mathrm{Order}\ T}{\mu X.O:\mathrm{Order}\ T}$$

$$\frac{O:\mathrm{Order}\ T}{\mathrm{Bag}(O):\mathrm{Order}\ (\mathrm{Seq}(T))} \qquad \frac{O:\mathrm{Order}\ T}{\mathrm{Set}(O):\mathrm{Order}\ (\mathrm{Seq}(T))}$$

Each derivable $O : T$ denotes an order (total preorder) on the elements of type $T$.



## Sorting discriminator

A *sorted partition* for (concrete) order $O$ is a partition listed in ascending order according to $O$. A *sorting discriminator* lists its blocks of results in sorted order according to the first components in the input.
Examples:
partition Anagram ["AbC", "aaC", "BcA"] = [["aaC"], ["AbC", "BcA"]]
partition Char ['c', 'a', 'c'] = [['a'], ['c', 'c']]
discriminate Anagram [("AbC", 1), ("aaC", 2), ("BcA", 3)] = [[2], [1,3]]
discriminate Char [('c', true), ('a', true), ('c', false)] = [[true], [true, false]]







## Linear-time sorting discrimination

**Theorem**

*For all derivable $O : T$, there is a linear-time sorting discriminator.*

**Corollary**

*For any $O : T$, lists of inputs of type $T$ can be sorted according to $O$ in linear time.*

Caveat:

- Bound only holds for unboxed data (no sharing); e.g., doesn't hold for input where elements are lists with shared tails.
- Same caveat applies to bounds for all other sorting algorithms!







## Generic definition

Discriminators can be defined inductively ("generically") on typed language denoting a practically useful class of total preorders using

- Generalized Algebraic Data Types (GADTs) and
- list comprehension

(See Haskell source code.)



## To sort or not to sort

- Bucket sort requires time $\Theta(n + N)$ where $N$ is the size of the array used. Basic discrimination (output not necessarily sorted according to index of array) requires only $\Theta(n)$ time. (NB: There may be many calls to bucket sort/discriminate with only a few elements in its argument in an application!)
- MSD (without sorting) works also for shared data if sorting is not required.
- MSD for circular data
- Sorting according to a particular (standard) order is often not required, only sorting according to *some* order that is common across a number of calls to sort. The latter can be implemented efficiently using MSD (weak sorting).







## Insights so far

- Powerful typed language of total preorders can be specified using Generalized Algebraic Data Types
- Linear time sorting discriminators can be defined inductively on the language of preorders.
- Sorting discriminators generalize both partitioning and sorting.







## Questions

- Sorting for boxed data — how and how hard?
- Can $O(n \log\log n)$ (Anderson et al., 1998) bound for $w$-bit RAM-model be extended to complex data?
- Is there an $O(n \log n)$ comparison-based sorting algorithm? (Can AKS-networks be generated fast enough? Probably not...)
- Find an example of nontrivial comparison function with cost $\Theta(|x| + |y|)$.
- For which variable-sized data is sorting used in practice? (Strings/sequences of atomic data and . . . ?)
- Can discrimination-based sorting be made competitive with Heapsort or Quicksort in terms of space consumption? (If so, there'd probably be little practical reason to use comparison-based sorting.)



# Hash free binary decision diagrams

M.I. Nielsen

Joint work with F. Henglein

Department of Computer Science
University of Copenhagen
<mortenib@diku.dk>

DIKU-IST 21 - 22 April, 2006







## Outline

1. BDD's a quick overview
2. Implementation
3. Motivation







### Definition

- *A Boolean variable* $x$ is a variable ranging over $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$. We will use $x_1, x_2, \ldots$ and $x, y, z, \ldots$ to denote Boolean variables.
- *A Boolean expression* is any nonempty expression generated from the following bnf grammar:

$$t ::= x \mid \mathsf{T} \mid \mathsf{F} \mid \neg t \mid t \wedge t \mid t \vee t \mid t \Rightarrow t \mid t \Leftrightarrow t$$



**Definition**

*if-then-else*-operator:
We define the *if-then-else-operator*, if $x\ t_1\ t_2$, by:

$$\text{if } x\ t_1\ t_2 = (x \wedge t_1) \vee (\neg x \wedge t_2)$$

A Boolean expression built using the following bnf grammar is said to be in *If-then-else Normal Form*.

$$\begin{array}{rcl} x & \in & \textit{Boolean variables} \\ i & ::= & \text{if } x\ i\ i \mid \mathsf{T} \mid \mathsf{F} \end{array}$$







**Lemma (Shannon expansion)**

*For all Boolean expressions $t$ and all Boolean variables $x$ we have:*

$$t \equiv \textit{if } x\ t[T/x]\ t[F/x]$$

Thus every Boolean expression is equivalent to an expression in INF.







Binary Decision Trees can represent Boolean expressions.

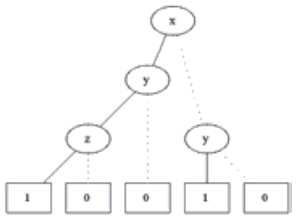



Figure: BDT representing the Boolean exp. $y \wedge (x \wedge z \vee \neg x)$.



## Generalization of BDTs

**Definition**

A *binary decision diagram*, is a finite dag with a unique initial node. All terminal nodes are labeled with T or F and all non-terminal nodes are labeled with Boolean variables. A non-terminal node, $u$, has exactly two out edges, one labeled T and one labeled F (represented as a solid and dotted line respectively).







## Redundancy problems

Four kinds of redundancy can occur in BDDs (and BDTs):
**R1** The same variable might occur several times on a path (from the root).

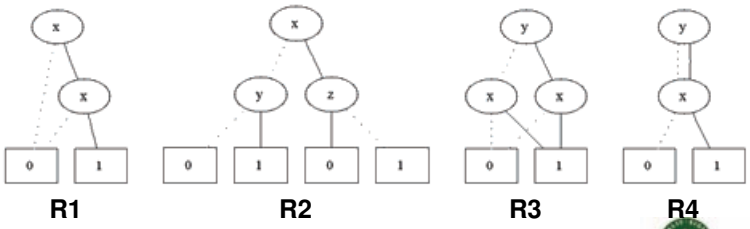



Figure: Redundancy in BDDs. R1, R3 and R4 represent the Boolean expression $x$ while R2 represents $(\neg x \wedge y) \vee (x \wedge \neg z)$.







## Redundancy problems

Four kinds of redundancy can occur in BDDs (and BDTs):
**R2** There might be duplicate terminal nodes.

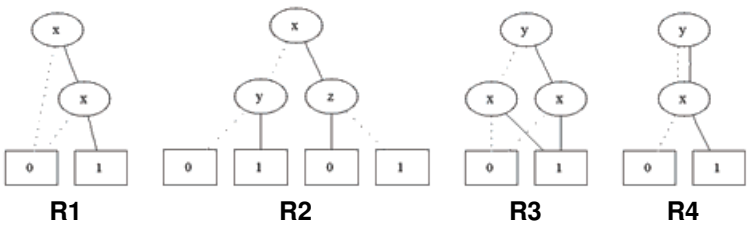



Figure: Redundancy in BDDs. R1, R3 and R4 represent the Boolean expression $x$ while R2 represents $(\neg x \wedge y) \vee (x \wedge \neg z)$.



## Redundancy problems

Four kinds of redundancy can occur in BDDs (and BDTs):
**R3** There might be duplicate non-terminal nodes.

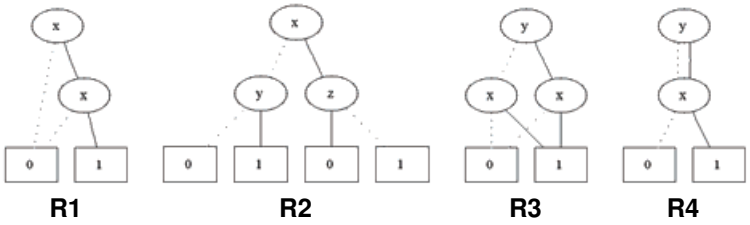



Figure: Redundancy in BDDs. R1, R3 and R4 represent the Boolean expression $x$ while R2 represents $(\neg x \wedge y) \vee (x \wedge \neg z)$.





## Redundancy problems

Four kinds of redundancy can occur in BDDs (and BDTs):
**R4** Both out edges of a non-terminal node can point to the same node.

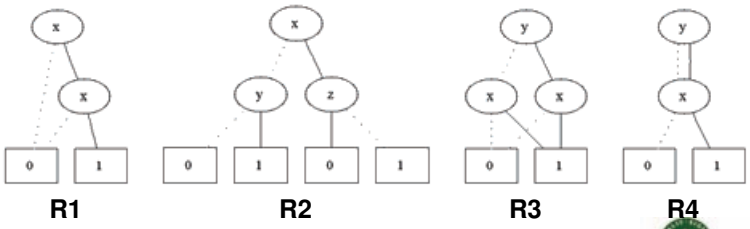



Figure: Redundancy in BDDs. R1, R3 and R4 represent the Boolean expression $x$ while R2 represents $(\neg x \wedge y) \vee (x \wedge \neg z)$.





## Reduced Ordered BDDs

**Definition**

Let $[x_1, x_2, \ldots, x_n]$ be an ordered list of variables with no duplicates, we shall call this a *variable order*. Let $B$ be a BDD. $B$ is called an *Ordered BDD*, henceforth OBDD, with respect to $[x_1, x_2, \ldots, x_n]$ if every variable in $B$ occur in the ordered list of variables and if for every $x_i$ followed by $x_j$ on any path in $B$ we have $i < j$. Furthermore if redundancies of type R2, R3 and R4 doesn't occur then $B$ is called a *Reduced OBDD*, henceforth BDD.

## Example BDD

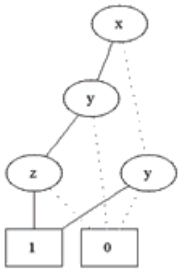



Figure: BDD representing the Boolean exp. $y \wedge (x \wedge z \vee \neg x)$ with respect to variable order $[x, y, z]$.





## Important theorems

**Theorem**

*Any Boolean expression, t, is equivalent to a ROBDD.*

**Theorem (Canonical forms theorem)**

*For any function $f : \mathbb{B}^n \to \mathbb{B}$ there exists exactly one ROBDD, B, with variable order $V = [x_1, x_2, \ldots, x_n]$ such that $f(x_1, x_2, \ldots, x_n) = f_B$, where $f_B$ denotes the (unique) function represented by B with respect to V.*





## The power of canonical forms

- Testing two expressions for equality is reduced to structural equivalence testing.
- A function is valid if and only if it is represented as $B_1$.
- A function is satisfiable if and only if it is not represented as $B_0$.
- Let $f$ and $g$ be functions $f, g : \mathbb{B}^n \to \mathbb{B}$ then $f \Rightarrow g$ if and only if $f \wedge \neg g$ is represented as $B_0$.

$B_1$: 1 $B_0$: 0

## BDD summary

- The canonical forms theorem makes it a powerfull representation.
- Often the representation is more compact than e.g. CNF and DNF.







## Common operations on BDDs

- restrict(T, $x$, $B_f$)
- satcount($B_f$)
- anysat($B_f$)
- allsat($B_f$)
- equal($B_f$, $B_g$)
- apply(op, $B_f$, $B_g$), op $\in$ACI-operators

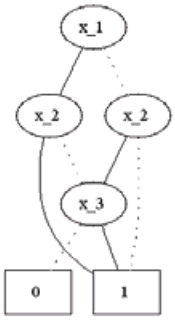






## Common operations on BDDs

- restrict(T, $x$, $B_f$)
- satcount($B_f$)
- anysat($B_f$)
- allsat($B_f$)
- equal($B_f$, $B_g$)
- apply(op, $B_f$, $B_g$), op $\in$ACI-operators

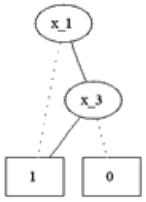



$B_{exa}$



## Common operations on BDDs

- restrict(T, $x$, $B_f$)
- satcount($B_f$)
- anysat($B_f$)
- allsat($B_f$)
- equal($B_f$, $B_g$)
- apply(op, $B_f$, $B_g$), op ∈ACI-operators




$B_{res}$ = restrict(F, $x_2$, $B_{exa}$)

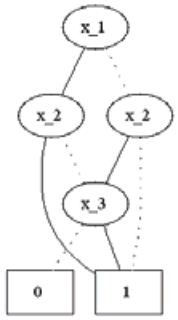






## Common operations on BDDs

- restrict(T, $x$, $B_f$)
- satcount($B_f$)
- anysat($B_f$)
- allsat($B_f$)
- equal($B_f$, $B_g$)
- apply(op, $B_f$, $B_g$), op ∈ACI-operators




satcount($B_{exa}$) = 6

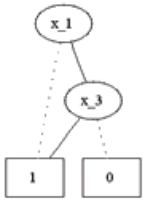






## Common operations on BDDs

- restrict(T, $x$, $B_f$)
- satcount($B_f$)
- anysat($B_f$)
- allsat($B_f$)
- equal($B_f$, $B_g$)
- apply(op, $B_f$, $B_g$), op ∈ACI-operators

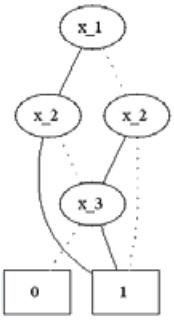



satcount($B_{res}$) = 3



## Common operations on BDDs

- restrict(T, $x$, $B_f$)
- satcount($B_f$)
- anysat($B_f$)
- allsat($B_f$)
- equal($B_f$, $B_g$)
- apply(op, $B_f$, $B_g$), op ∈ACI-operators




anysat($B_{exa}$) = $[x_1 = \mathrm{T}, x_2 = \mathrm{T}]$

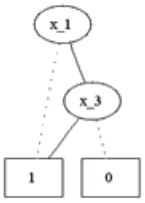






## Common operations on BDDs

- restrict(T, $x$, $B_f$)
- satcount($B_f$)
- anysat($B_f$)
- allsat($B_f$)
- equal($B_f$, $B_g$)
- apply(op, $B_f$, $B_g$), op ∈ACI-operators




allsat($B_{res}$) = $[[x_1 = \mathrm{F}], [x_1 = \mathrm{T}, x_3 = \mathrm{T}]]$

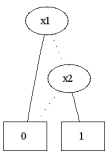




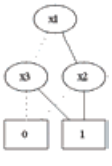


## Common operations on BDDs

- restrict(T, $x$, $B_f$)
- satcount($B_f$)
- anysat($B_f$)
- allsat($B_f$)
- equal($B_f$, $B_g$)
- apply(op, $B_f$, $B_g$), op ∈ACI-operators




$B_a$ represents $\neg x_1 \wedge x_2$

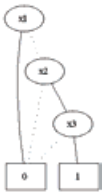



$B_b$ represents $(x_1 \wedge x_2) \vee (\neg x_1 \wedge x_3)$



## Common operations on BDDs

- restrict(T, $x$, $B_f$)
- satcount($B_f$)
- anysat($B_f$)
- allsat($B_f$)
- equal($B_f$, $B_g$)
- apply(op, $B_f$, $B_g$),
  op $\in$ACI-operators




apply($\wedge$, $B_a$, $B_b$) = $B_{a \wedge b}$







The key to efficient BDD design is to minimize memory usage and eliminate redundant computations. Memory usage can be divided into:

- Memory usage when idle (memory for BDD datastructure).
- Memory usage of operations on BDDs.







## apply - core algorithm

Apply is based on Shannon expansion

```
apply(b1,b2) {
  case b1 as if x_i n1 n2 {
    case b2 as if x_j n3 n4 {
      if x_i < x_j then
        newNode(x_i, apply(n1,b2), apply(n2,b2))
      else if x_j < x_i then
        newNode(x_j, apply(b1,n3), apply(b1,n4))
      else //x_i=x_j
        newNode(x_i, apply(n1,n3), apply(n2,n4))
    }
    .
    .
    .
```



## Hash based approach

- Uses value numbering (hashed consing) to represent BDDs.
- Uses memoization to avoid redundant computations.
- In apply recursive calls are used to compute new BDD.

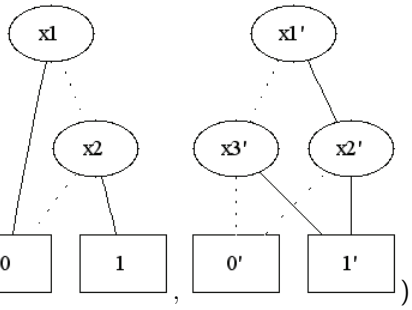






## The MSD way

Apply is implemented as a worklist algorithm.




$\text{apply}_{\wedge}($ , )

1. Worklist: $[x_1 : [\{x_1, x_1'\}]]$

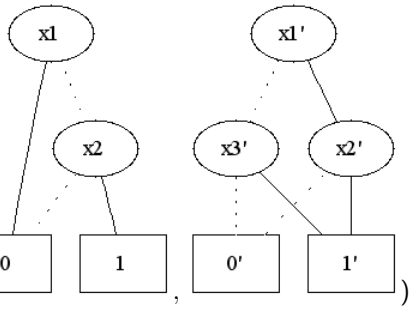






## The MSD way

Apply is implemented as a worklist algorithm.



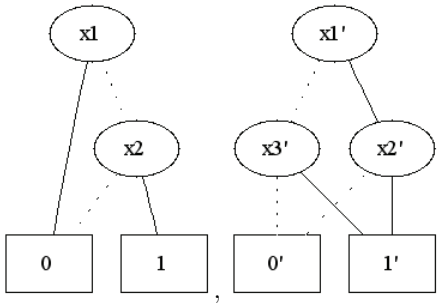

$\text{apply}_{\wedge}($ , )

2. Worklist: $[x_2 : [\{x_2, x_3'\}, \{0, x_2'\}]]$



## The MSD way

Apply is implemented as a worklist algorithm.




$\text{apply}_\wedge($ , )

3. Worklist: $[x_2 : [\{0, x_2'\}], x_3 : [\{0, x_3'\}, \{1, x_3'\}]]$

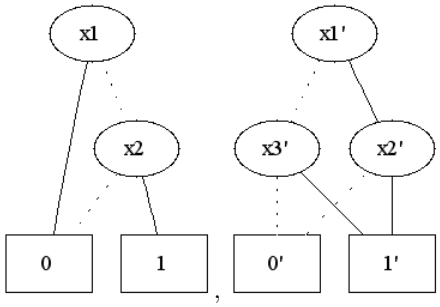






## The MSD way

Apply is implemented as a worklist algorithm.




$\text{apply}_\wedge($ , )

4. Worklist: $[x_3 : [\{0, x_3'\}, \{1, x_3'\}], 0 : [\{0, 0\}, \{0, 1\}]]$

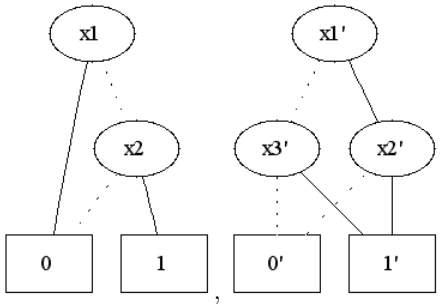






## The MSD way

Apply is implemented as a worklist algorithm.



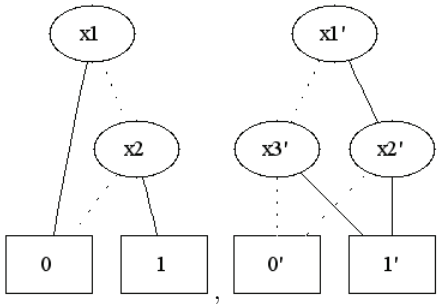

$\text{apply}_\wedge($ , )

5. Worklist: $[x_3 : [\{1, x_3'\}], 0 : [\{0, 0\}, \{0, 1\}, \{0, 0\}, \{0, 1\}]]$



## The MSD way

Apply is implemented as a worklist algorithm.




$\text{apply}_{\wedge}($ , )

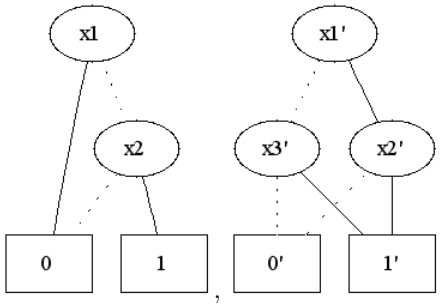
8. Worklist: $[0 : [\{0,0\},\{0,1\},\{0,0\},\{0,1\},\{1,0\}], 1 : \{1,1\}]$





## The MSD way

Apply is implemented as a worklist algorithm.




$\text{apply}_{\wedge}($ , )

9. Msd reduces worklist to: $[0 : [\{0,0\}], 1 : [\{1,1\}]]$





Why do we do this:

- We want to get experience with the msd technique by applying it to a well studied problem.
- We can avoid manual garbage collection.
- We believe the msd-way can provide new insight into more efficient implementations of BDDs.

Future work (a lot):

- Complete msd implementation and make benchmarks.
- Analyze our approach more formally - especially with respect to use of memory in apply.
- Apply the method to related problems.

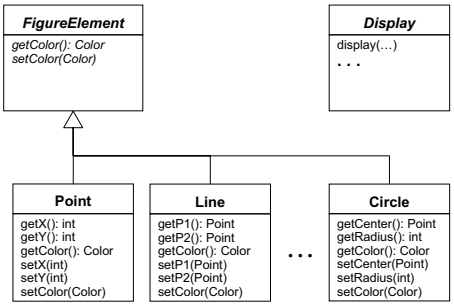
# *Aspect Mining Using a Structural Concern Model*

*Jan Hannemann, Hidehiko Masuhara*

**Programming Principles and Practices Group,**
**University of Tokyo**

## Example: A Graphical Figure Editor

- Typically, OO software is decomposed into classes corresponding to entities in application domain.

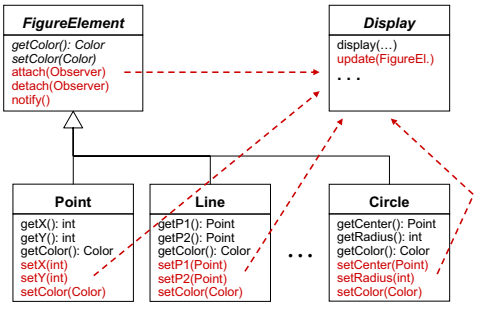



Example:
A figure editor



## Example: Adding Display Updates

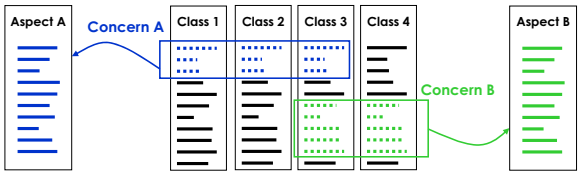
- Typically, OO software is decomposed into classes corresponding to entities in application domain.
- Certain concerns crosscut this primary decomposition




Example:
The display updating concern

# Aspect-Oriented Programming

- AOP allows to modularize crosscutting concerns
- Systems with better modularity are easier to reason about, maintain and evolve






# Aspect-Oriented Implementation

```
public aspect DisplayUpdate {

  LinkedList FigureElement.observers = new LinkedList();

  public void FigureElement.addObserver(Display o) {
    observers.add(o); }

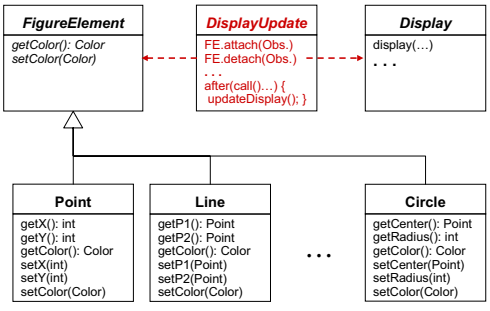
  public void FigureElement.removeObserver(Display o) {
    observers.remove(o); }

  after(FigureElement figure):
    call(FigureElement+.set*(..)) {
      Iterator iter = figure.observers.iterator();
      while ( iter.hasNext() ) {
        Display d = (Display) iter.next();
        d.update(figure); } }
}
```



# Example: Adding Display Updates

- Typically, OO software is decomposed into classes corresponding to entities in application domain.
- Certain concerns crosscut this primary decomposition



## Aspect Mining

- We know:
  - Software systems often have CCCs
    - Security, Synchronization, Logging,
    - Design Patterns, Idioms …
  - CCCs can be given better modularity using AOP
  - Replacement of CCCs with modular aspects is (semi-)automatable [Hannemann'05], [Binkely'05]
- The challenge: Aspect Mining
  - Identifying CCCs in software
  - Legacy systems can only benefit from AOP if CCCs are properly identified



## Related Work: Aspect Mining

- Manual mining
- Exploratory approaches
  - Aspect Mining Tool [Hannemann'00], Aspect Browser [Griswold'99], FEAT [Robillard'02], etc.
- Mining based on program analysis
  - Dynamic
    - Formal concept analysis of execution traces [Tonella'04, Tourwe'04], control flow [Krinke'04]
  - Static: abstract syntax tree
    - High fan-in methods [Marin'04], clone detection [Bruntink'04], search for natural language clues [Shepherd'06]

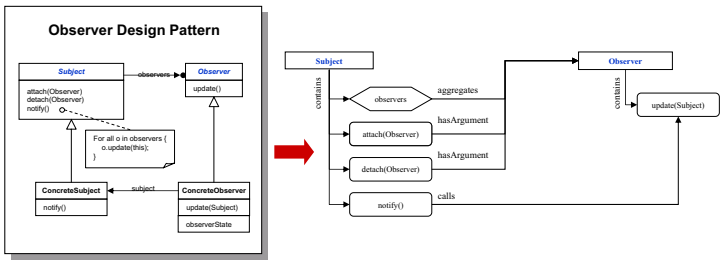


## Mining using a Structural Concern Model

- Other approaches: generic
  - Mine programs using generic CCC "indicators"
  - Hope matches are part of one or more CCCs
  - Little semantic information about matches
- Our approach: specific
  - Search for CCCs with known structures and known refactorings
  - Allows us to create precise, fully automated searches
  - Two ideas (+ working implentations)
    - Structural search over program's abstract syntax tree
    - Structural query over logic program database

## Structural Representations of Concerns

- Descriptions of CCCs in terms of role elements and their relations
  - Captures structural “blueprint”

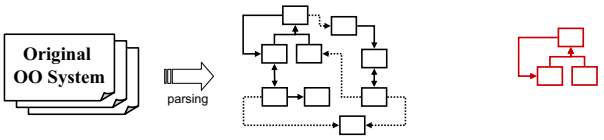





## Idea 1: Aspect Mining as Subgraph Matching

Based on prior work

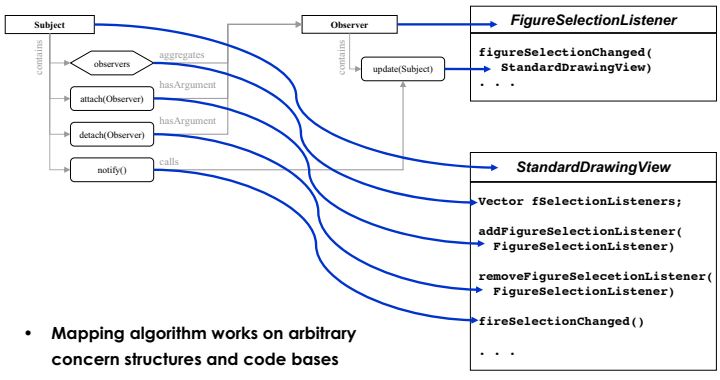
- Identifying occurrences of the concern structure in the program’s AST
- CCC structure is compared to AST structure




Source Code | Abstract Syntax Tree (AST) | Concern representation



## Mapping Structural Elements to Program Elements




- Mapping algorithm works on arbitrary concern structures and code bases

# Idea 1: Discussion

- Idea
  - Match CCC structure to program AST
- Status
  - Shown to work on non-trivial software for a variety of CCCs
- Advantages
  - Identified code is semantically related
  - Given a starting point (seed), the algorithm is efficient
- Disadvantages
  - Needs a starting point to be efficient



# Idea 2: Aspect Mining via Logic Queries

NEW

- Parse program into a database of facts, use logic query to identify CCCs
- CCC structure is reflected in logic constraints




Source Code | Database of program facts | Concern as logic query



# Idea 2: Discussion

- Idea
  - Logic query over database of program facts
- Status
  - Shown to work on non-trivial software for specific CCCs
- Advantages
  - Identified code is semantically related
  - Very fast for simple concern structures
- Disadvantages
  - Complex concern structures result in poor performance

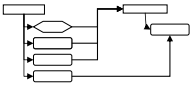
# Overview

**Problem**



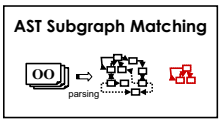

**Aspect Mining: identifying CCCs in legacy systems**

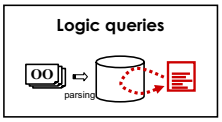
**Approach**



**Structural concern models allow for precise, automated mining**

**Ideas**






# Summary

- Advantages of using structural CCC model
  - Mining can be automated
  - Results are semantically meaningful
  - Results can serve as direct input to AOP refactoring
- Limitations
  - CCC structure must be known and expressive
  - Code structure must not vary too much from expected structure
  - Performance varies
- Current work focuses on
  - Extending the library of concerns
  - Improving performance

# A Method for Inversion of a Higher-order Functional Programming Language

Jesper Andersen
Computer Science Department, University of Copenhagen

April 21, 2006, 2nd. DIKU-IST joint workshop

## The map-program

Source program to invert:

```
fun map f = fn l => case l of
  Nil => Nil |
  Cons (x, xs) => Cons (f x, map f xs)

fun succ x = S x

fun main l = map succ l
```

## The inverted map-program

Inverse version:

```
fun map f = fn l => case l of
  Nil => Nil |
  Cons (x, xs) => Cons (f x, map f xs)

fun pred x = case x of S y => y

fun main l = map pred l
```

## Inversion In General

- Given program $p$ and semantic function $[\![\cdot]\!]$
- Derive inverse program $p^{-1}$ st.

$$[\![p]\!](x) = y \Leftrightarrow [\![p^{-1}]\!](y) = x$$

- Restrictions usually applied on either program

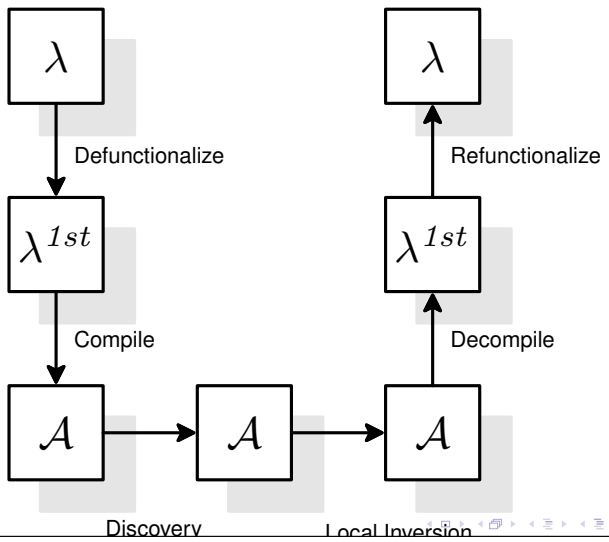
## Inverting Higher-order Functional Programs

See more 27

**Assumptions**

- One-to-one programs
- Disjoint branches in case-expressions
- Usage of explicit duplication and discarding operators

## Our approach



## Stack Programs

- Has CFG-like syntax
- Operates on a stack (may reorder elements)
- Linear usage of values
- *First-order* programs

$$\begin{array}{lcl} \mathit{succ} & \rightarrow & \text{S!} \\ \mathit{listSucc} & \rightarrow & \text{Nil? Nil!} \\ \mathit{listSucc} & \rightarrow & \text{Cons? Swp } \mathit{listSucc} \\ & & \text{Swp } \mathit{succ} \text{ Cons!} \end{array}$$

## Inversion Of Stack Programs

Local Inversion of stack programs

- Each instruction is "flipped"

$$\text{c!} \leftrightarrow \text{c?} \qquad f \leftrightarrow f^{-1} \qquad \text{Swp} \leftrightarrow \text{Swp}$$

- Reverse order of flipped instructions
- After local inversion we may have a non-deterministic program!

$$\begin{array}{lcl} f & \rightarrow & \text{Swp } \ldots \\ f & \rightarrow & \text{S! } \ldots \end{array}$$

## Defunctionalization (John Reynolds) See more 29

- Closures represented *within* source language

$$\begin{array}{lcl} \textbf{fn } \text{x} \Rightarrow \text{e} & \rightsquigarrow & \text{C0 (...)} \\ \textbf{fun } \text{f x} \Rightarrow \text{e} & \rightsquigarrow & \text{Cf (...)} \end{array}$$

- Introduce function to handle applications

$$\begin{array}{lcl} \text{(e1 e2)} & \rightsquigarrow & \text{(C0 D(e2))} \\ & \rightsquigarrow & \text{apply (C0, D(e2))} \end{array}$$

- Defunctionalization is a *whole-program* transformation

## Variants of defunctionalization

Untyped: Make just one **apply** function

Typed: Discriminate **apply** based on *type* of applied closure

Flows: Use *control flow* to specialize **apply** function

Which is better in the context of inversion?

## Sample Program

```
fun succ x = S x

fun pred x = case x of S y => y

fun mapp x = fn f => f x

fun entry x = mapp x succ
```

## Untyped Defunctionalization

```
fun apply (f, a) = case f of
  Csucc => S a
| Cpred => case a of S y => y
| Cmapp => C0 a
| C0 x => apply(a, x)

fun entry x =>
    apply(
      apply(Cmapp, x),
      Csucc
    )
```

Now we compile into the stack language...

## Compiled Version: untyped

Forward program:

$$
\begin{array}{lcl}
apply & \rightarrow & \text{Csucc? S!} \\
apply & \rightarrow & \text{Cpred? S?} \\
apply & \rightarrow & \text{Cmapp? C0!} \\
apply & \rightarrow & \text{C0? Swp } apply \\
entry & \rightarrow & \text{Cmapp! } \ldots \ apply
\end{array}
$$

After local inversion

$$
\begin{array}{lcl}
apply^{-1} & \rightarrow & \text{S? Csucc!} \\
apply^{-1} & \rightarrow & \text{S! Cpred!} \\
apply^{-1} & \rightarrow & \text{C0? Cmapp!} \\
apply^{-1} & \rightarrow & apply^{-1} \text{ Swp C0!} \\
entry & \rightarrow & apply^{-1} \ \ldots \ \text{Cmapp?}
\end{array}
$$

## Compiled: flow based

Forward program:

$$
\begin{array}{lcl}
apply_{Csucc} & \rightarrow & \text{Csucc? S!} \\
apply_{C0} & \rightarrow & \text{C0? Swp } apply_{Csucc} \\
apply_{Cmapp} & \rightarrow & \text{Cmapp? C0!} \\
entry & \rightarrow & \text{Cmapp! } \ldots \ apply_{Csucc}
\end{array}
$$

After local inversion

$$
\begin{array}{lcl}
apply^{-1}_{Csucc} & \rightarrow & \text{S? Csucc!} \\
apply^{-1}_{C0} & \rightarrow & apply^{-1}_{Csucc} \text{ Swp C0!} \\
apply^{-1}_{Cmapp} & \rightarrow & \text{C0? Cmapp!} \\
entry & \rightarrow & apply^{-1}_{C0} \ \ldots \ \text{Cmapp?}
\end{array}
$$

## Explicit duplication and discarding

- A one-to-one *program* may contain non one-to-one *functions*:

```
fun map f = fn x => case x of
  Nil => Nil |
  Cons (x, xs) => Cons
    (f x, map f xs)
fun main l = map succ l
```

>> map f [] = [] so f was lost

## Automatic Discovery

How to "discover" a value that was discarded:

- We analyse the *compiled inverted stack program*
- Insert copying and discarding instructions

$$\texttt{Dup}, \texttt{Dis}$$

$$\bullet\ \texttt{Dup}\ \ldots$$

$$\underbrace{\langle \ldots, x \rangle}_{\leftarrow}$$

## Automatic Discovery

How to "discover" a value that was discarded:

- We analyse the *compiled inverted stack program*
- Insert copying and discarding instructions

$$\texttt{Dup}, \texttt{Dis}$$

$$\texttt{Dup}\ \bullet\ \ldots$$

$$\underbrace{\langle \ldots, x, x \rangle}_{\leftarrow}$$

## Automatic Discovery

How to "discover" a value that was discarded:

- We analyse the *compiled inverted stack program*
- Insert copying and discarding instructions

$$\texttt{Dup}, \texttt{Dis}$$

$$\bullet\ \texttt{Dis}\ \ldots$$

$$\underbrace{\langle \ldots, x \rangle}_{\leftarrow}$$

## Automatic Discovery

How to "discover" a value that was discarded:

- We analyse the *compiled inverted stack program*
- Insert copying and discarding instructions

$$\texttt{Dup}, \texttt{Dis}$$

$$\texttt{Dis} \bullet \ldots$$

$$\underleftarrow{\langle \ldots \rangle}$$

## Example Discovery

Consider a *simple* K-combinator program: Functionally

```
fun kcomb (x, y) = x
fun main x = kcomb (x, x)
```

Stack Program

$$\begin{array}{lll} \mathit{kcomb} & \rightarrow & \text{Pair? Swp } \texttt{Dis} \\ \mathit{main} & \rightarrow & \texttt{Dup Pair! } \mathit{kcomb} \end{array}$$

Inverted

$$\begin{array}{lll} \mathit{kcomb}^{-1} & \rightarrow & \texttt{Dis}^{-1} \text{ Swp Pair!} \\ \mathit{main}^{-1} & \rightarrow & \mathit{kcomb}^{-1} \text{ Pair? Swp } \texttt{Dup}^{-1} \end{array}$$

## Example Discovery (cont'd)

Inverted

$$\begin{array}{lll} \mathit{kcomb}^{-1} & \rightarrow & \texttt{Dis}^{-1} \text{ Swp Pair!} \\ \mathit{main}^{-1} & \rightarrow & \bullet\ \mathit{kcomb}^{-1} \text{ Pair? Swp } \texttt{Dup}^{-1} \end{array}$$

Let's try to run the program with abstract input stack:

$$\underleftarrow{\langle x \rangle}$$

## Example Discovery (cont'd)

Inverted

$$\begin{array}{lll} kcomb^{-1} & \rightarrow & \bullet\ \texttt{Dis}^{-1}\ \text{Swp Pair!} \\ main^{-1} & \rightarrow & kcomb^{-1}\ \text{Pair? Swp}\ \texttt{Dup}^{-1} \end{array}$$

Let's try to run the program with abstract input stack:

$\underleftarrow{\langle x \rangle}$ What was the value discarded?

## Example Discovery (cont'd)

Inverted

$$\begin{array}{lll} kcomb^{-1} & \rightarrow & \texttt{Dis}^{-1}\ \bullet\ \text{Swp Pair!} \\ main^{-1} & \rightarrow & kcomb^{-1}\ \text{Pair? Swp}\ \texttt{Dup}^{-1} \end{array}$$

Let's try to run the program with abstract input stack:

$\underleftarrow{\langle x, \square \rangle}$ What was the value discarded?

Recall:

$$\begin{array}{lll} \langle \ldots, v \rangle; \texttt{Dis}\ is & \longrightarrow & \langle \ldots \rangle; is \\ \langle \ldots, v \rangle; \texttt{Dup}\ is & \longrightarrow & \langle \ldots v, v \rangle\ is \end{array}$$

## Example Discovery (cont'd)

Inverted

$$\begin{array}{lll} kcomb^{-1} & \rightarrow & \texttt{Dis}^{-1}\ \text{Swp}\ \bullet\ \text{Pair!} \\ main^{-1} & \rightarrow & kcomb^{-1}\ \text{Pair? Swp}\ \texttt{Dup}^{-1} \end{array}$$

Let's try to run the program with abstract input stack:

$\underleftarrow{\langle \square, x \rangle}$ What was the value discarded?

Recall:

$$\begin{array}{lll} \langle \ldots, v \rangle; \texttt{Dis}\ is & \longrightarrow & \langle \ldots \rangle; is \\ \langle \ldots, v \rangle; \texttt{Dup}\ is & \longrightarrow & \langle \ldots v, v \rangle\ is \end{array}$$

## Example Discovery (cont'd)

Inverted

$$\begin{array}{lcl} kcomb^{-1} & \rightarrow & \texttt{Dis}^{-1}\ \text{Swp Pair!} \bullet \\ main^{-1} & \rightarrow & kcomb^{-1}\ \text{Pair? Swp}\ \texttt{Dup}^{-1} \end{array}$$

Let's try to run the program with abstract input stack:

$\underleftarrow{\langle(\square, x)\rangle}$

What was the value discarded?

Recall:

$$\begin{array}{lcl} \langle \ldots, v\rangle; \texttt{Dis}\ is & \longrightarrow & \langle \ldots\rangle; is \\ \langle \ldots, v\rangle; \texttt{Dup}\ is & \longrightarrow & \langle \ldots v, v\rangle\ is \end{array}$$

## Example Discovery (cont'd)

Inverted

$$\begin{array}{lcl} kcomb^{-1} & \rightarrow & \texttt{Dis}^{-1}\ \text{Swp Pair!} \\ main^{-1} & \rightarrow & kcomb^{-1} \bullet \text{Pair? Swp}\ \texttt{Dup}^{-1} \end{array}$$

Let's try to run the program with abstract input stack:

$\underleftarrow{\langle(\square, x)\rangle}$

What was the value discarded?

Recall:

$$\begin{array}{lcl} \langle \ldots, v\rangle; \texttt{Dis}\ is & \longrightarrow & \langle \ldots\rangle; is \\ \langle \ldots, v\rangle; \texttt{Dup}\ is & \longrightarrow & \langle \ldots v, v\rangle\ is \end{array}$$

## Example Discovery (cont'd)

Inverted

$$\begin{array}{lcl} kcomb^{-1} & \rightarrow & \texttt{Dis}^{-1}\ \text{Swp Pair!} \\ main^{-1} & \rightarrow & kcomb^{-1}\ \text{Pair?} \bullet \text{Swp}\ \texttt{Dup}^{-1} \end{array}$$

Let's try to run the program with abstract input stack:

$\underleftarrow{\langle\square, x\rangle}$

What was the value discarded?

Recall:

$$\begin{array}{lcl} \langle \ldots, v\rangle; \texttt{Dis}\ is & \longrightarrow & \langle \ldots\rangle; is \\ \langle \ldots, v\rangle; \texttt{Dup}\ is & \longrightarrow & \langle \ldots v, v\rangle\ is \end{array}$$

## Example Discovery (cont'd)

Inverted

$$\begin{array}{lcl} kcomb^{-1} & \rightarrow & \texttt{Dis}^{-1}\ \text{Swp Pair!} \\ main^{-1} & \rightarrow & kcomb^{-1}\ \text{Pair? Swp} \bullet \texttt{Dup}^{-1} \end{array}$$

Let's try to run the program with abstract input stack:

$\underleftarrow{\langle x, \square \rangle}$ What was the value discarded?

Recall:

$$\begin{array}{lcl} \langle \ldots, v \rangle; \texttt{Dis}\ is & \longrightarrow & \langle \ldots \rangle; is \\ \langle \ldots, v \rangle; \texttt{Dup}\ is & \longrightarrow & \langle \ldots v, v \rangle\ is \end{array}$$

## Example Discovery (cont'd)

Inverted

$$\begin{array}{lcl} kcomb^{-1} & \rightarrow & \texttt{Dis}^{-1}\ \text{Swp Pair!} \\ main^{-1} & \rightarrow & kcomb^{-1}\ \text{Pair? Swp}\ \texttt{Dup}^{-1} \bullet \end{array}$$

Let's try to run the program with abstract input stack:

$\underleftarrow{\langle x \rangle}$ $\square = x \ldots$

Recall:

$$\begin{array}{lcl} \langle \ldots, v \rangle; \texttt{Dis}\ is & \longrightarrow & \langle \ldots \rangle; is \\ \langle \ldots, v \rangle; \texttt{Dup}\ is & \longrightarrow & \langle \ldots v, v \rangle\ is \end{array}$$

## Example Discovery (cont'd)

Inverted

$$\begin{array}{lcl} kcomb^{-1} & \rightarrow & \texttt{Dup}\ \text{Swp Pair!} \\ main^{-1} & \rightarrow & kcomb^{-1}\ \text{Pair? Swp}\ \texttt{Dis} \end{array}$$

Let's try to run the program with abstract input stack:

$\underleftarrow{\langle x \rangle}$ $\square = x$ so $\texttt{Dis}^{-1} = \texttt{Dup}$

Recall:

$$\begin{array}{lcl} \langle \ldots, v \rangle; \texttt{Dis}\ is & \longrightarrow & \langle \ldots \rangle; is \\ \langle \ldots, v \rangle; \texttt{Dup}\ is & \longrightarrow & \langle \ldots v, v \rangle\ is \end{array}$$

## Program state exploration method

The analysis *sketched* above is WIP

- Termination issues due to recursion
- Value of $\square$ must agree on all paths
- Derivation of preconditions could allow more precision of method

Still: The method works for the map program

## Summary of details

Defunctionalization:

- Defunctionalization suitable for inversion
- More "precise" defunctionalization enlarges class of invertible programs

Discovery:

- Simple
- Works for at least two programs

## Things to do in the future...

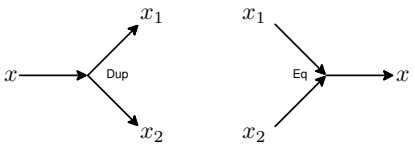
- Transformation back to higher order (Refunctionalization)
- More "precise" defunctionalization
- Develop theory of (dis)cover
- Analysis of stack programs
  - Optimizations based on grammar program traces
  - Determinization based on stack behaviour

## See you next year

# Mange tak!

## 1st. order functional inversion

- One-to-one programs
- Disjoint branches in case-expressions
- Well-formed programs
  - Patterns are non-overlapping and linear
  - No dead-variables
  - Usage of special duplication/equality test operator




## Properties of grammar programs

Given program $P$,

- Let $A, Q$ be it's set of actions (c!) and queries c?
- Let $A_f/Q_f$ be the set of actions and queries with respect to function $f$
- View $P$ as generator of traces $L \subseteq \mathcal{P}((Q \cup A)^*)$
- Pre-derivative: $x \backslash L = \{y \mid xy \in L\}$
- $\mathit{First}(L) = \{b \mid bx \in L\}$
- $L$ is *start-deterministic* if
  - $\mathit{First}(L) \subseteq Q$ or
  - $\mathit{First}(L) = \{a\}$

## Defunctionalization is *Meaning-preserving*

- Let $D(\cdot)$ denote defunctionalization
- Then

$$[\![p]\!](x) = y \Leftrightarrow [\![D(p)]\!](D(x)) = D(y)$$

- We shall restrict *program* inputs/outputs to first order

## Defunctionalization is *Meaning-preserving*

- Let $D(\cdot)$ denote defunctionalization
- Then

$$[\![p]\!](x) = y \Leftrightarrow [\![D(p)]\!](x) = y$$

- We shall restrict *program* inputs/outputs to first order

## Map inverted... optimized

```
fun applyC0 l = case l of
  Nil => (C0 C1, Nil) |
  Cons (S x, xs) => case applyC0 xs of
    (C0 e, d) => dis e
        [(C0 C1, Cons (b, d))]

fun main x = case applyC0 x of
  (C0 C1, b) => b
```

## Map inverted... further optimized

```
fun applyC0 l = case l of
  Nil => Nil |
  Cons (S x, xs) => Cons (x, applyC0 xs)

fun main x = applyC0 x
```

## Map inverted... “pattern-lifted”

```
fun patLift x = case x of (S y) => y

fun applyC0 l = fn f => case l of
  Nil => Nil  |
  Cons (x, xs) => Cons (f1 x, applyC0 f2 xs)

fun main x = applyC0 patLift x
```

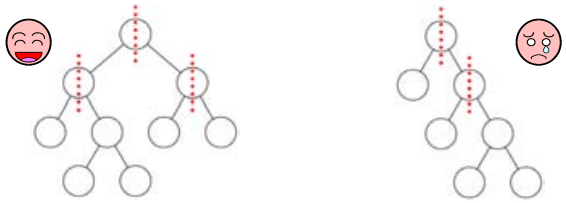
# A Parallelization Tool for Tree Reductions

Kiminori Matsuzaki
Noriyuki Ohkawa

IST, University of Tokyo

## Parallel Programming for Trees

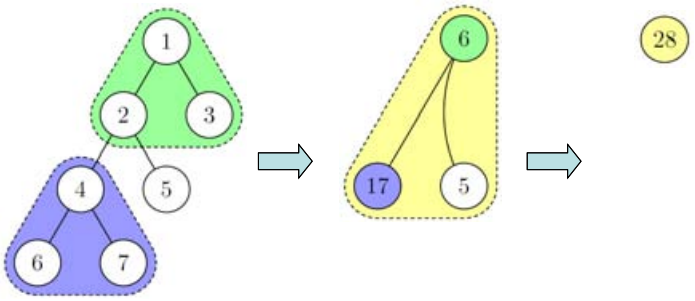
- Divide and Conquer
  - Simple and used widely
  - May be inefficient for ill-balanced trees



- Tree Contraction (in next page)

## Tree Contraction [Miller & Reif 85]

- Efficient algorithms for trees of any shapes




  - Additional functions for contractions

# SkeTo: Parallel Skeleton Library

http://www.ipl.t.u-tokyo.ac.jp/sketo/

Parallel Skeletons: Basic components for efficient parallel computations

– E.g. for tree reduction

```
tree_skeletons::reduce
(k_L, k_N, phi,
 psi_L, psi_R, G);
```




# Difficulties in Tree Contraction

Need for the six function parameters

How to develop those functions?

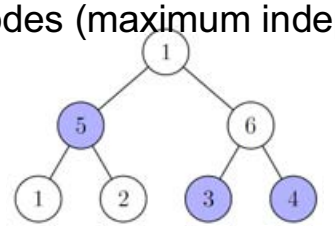
→ Propose tupled-ring theorem for many dynamic programming on trees

It is reluctant to write code for functions!

→ Develop a tool for generating parallel code from users' divide-and-conquer program

# Motivating Problem: Party Planning

Compute maximum of sums of disjoint nodes (maximum independent-set sum)




⇨ 12

$$k_N\ b \begin{pmatrix} s_l \\ n_l \end{pmatrix} \begin{pmatrix} s_r \\ n_r \end{pmatrix} = \begin{pmatrix} b + n_l + n_r \\ (s_l \uparrow n_l) + (s_r \uparrow n_r) \end{pmatrix}$$

(↑ returns the larger value)

## Motivating Problem: Party Planning

- Operators form a commutative semiring

$$k_N\ b \begin{pmatrix} s_l \\ n_l \end{pmatrix} \begin{pmatrix} s_r \\ n_r \end{pmatrix} = \begin{pmatrix} b + n_l + n_r \\ (s_l \uparrow n_l) + (s_r \uparrow n_r) \end{pmatrix}$$

- We can transform RHS as follows

$$\begin{pmatrix} b + n_l + n_r \\ (s_l \uparrow n_l) + (s_r \uparrow n_r) \end{pmatrix} = \begin{pmatrix} -\infty & b + n_r \\ s_r \uparrow n_r & s_r \uparrow n_r \end{pmatrix} \times_{\{+,\uparrow\}} \begin{pmatrix} s_l \\ n_l \end{pmatrix} = \begin{pmatrix} -\infty & b + n_l \\ s_l \uparrow n_l & s_l \uparrow n_l \end{pmatrix} \times_{\{+,\uparrow\}} \begin{pmatrix} s_r \\ n_r \end{pmatrix}$$

- Associativity of $\times_{\{+,\uparrow\}}$ enables parallelization

## Tupled-ring Form

Many dynamic programmings on trees have the following form (Tupled-ring form)

– Let $\otimes$ and $\oplus$ form a commutative semiring

$$(v_1, \ldots, v_k, \iota_\otimes)^T = \begin{pmatrix} a_{11} & \cdots & a_{1k} & a_{1(k+1)} \\ \vdots & \ddots & \vdots & \vdots \\ a_{k1} & \cdots & a_{kk} & a_{k(k+1)} \\ \iota_\oplus & \cdots & \iota_\oplus & \iota_\otimes \end{pmatrix} \times_{\otimes,\oplus} \begin{pmatrix} l_1 \\ \vdots \\ l_k \\ \iota_\otimes \end{pmatrix}$$

(Symmetric to $r_1, \ldots, r_k$)

## Tupled-ring Theorem

**Theorem (Tupled-ring theorem)**
From tree programs in tupled-ring form, functions for contractions can be derived systematically.

– Idea: Based on the associativity of matrix-multiplications

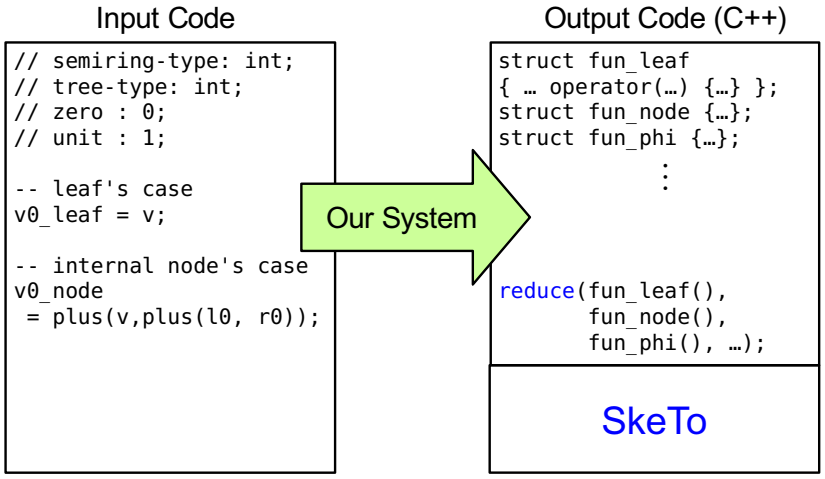
– For details: Please see our paper

# Applications

- Evaluation of computational trees with $+, -, \times$
  - On commutative semiring $\{Num, \times, +\}$
- Maximum weight-sum problems
  (e.g. Party Planning Problem)
  - On commutative semiring $\{Num, +, \uparrow\}$
- XPath Queries
  - On commutative semiring $\{Bool, \wedge, \vee\}$

# Implementation of Parallelization Tool

# Overview



```
pair party_planning(const tree& n) {
  if (n.isLeaf()) { return pair(n.v,0); }
  else {
    pair l = party_planning(n.left());
    pair r = party_planning(n.right());
    return pair(n.v + l.v1 + r.v1,
                max(l.v0,l.v1)+max(r.v0,r.v1));
  }
}
```

Retur ...ch case

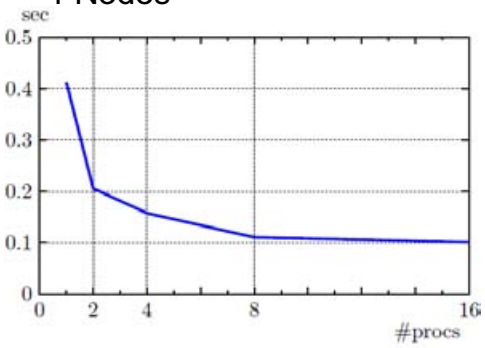
```
-- leaf's case
v0_leaf = v;
v1_leaf = 0;

-- internal node's case
v0_node = otimes(v,(otimes(l1,r1)));
v1_node = otimes(oplus(l0,l1),oplus(r0,r1));
```

# Experiments

- Party planning problem
  - 8,388,607 = $2^{23}$ – 1 Nodes

mpich 1.2.6
gcc 2.95
PC Cluster with Gb-Ether
Pentium4 3.0GHz, (HT on),
Mem 1GB




# Related Work

- Parallelization tools for arrays
  - [Fisher and Ghuloum 1994]
    - Loop parallelization
    - If-statement and one associative operators
  - [Xu, Khoo and Hu 2004]
    - Type-based approach
    - Multiple operators satisfying associativity and distributivity

## Conclusion

- Tupled-ring theorem
  - Idea: associativity of matrix-multiplication
  - Target: many DPs on trees
- A parallelization tool for tree programs
  - A prototype implementation
  - Less effort in writing parallel code
  - Generated code is reasonably fast

## Future Work

- Further implementation of the system
  - Involved parser to accept users' C++ code
  - For other computational patterns on trees (e.g. tree accumulations)

## Thank You

- Please visit our project's site!

  http://www.ipl.t.u-tokyo.ac.jp/sketo/



**Functional Pearl:**

# Weak Inversion for Parallelization

Akimasa Morihata  Zhenjiang Hu
(IST, University of Tokyo)

The Second DIKU-IST Workshop

## Parallelization

Sequential program

$$\begin{aligned} sum\ [a] &= a \\ sum\ (a : x) &= a + (sum\ x) \end{aligned}$$

⇓ Parallelization

Parallel program

$$\begin{aligned} sum\ [a] &= a \\ sum\ (l +\!\!+ r) &= (sum\ l) + (sum\ r) \end{aligned}$$

## List Homomorphism

$$\begin{aligned} (\!|\odot, f|\!)\ [a] &= f\ a \\ (\!|\odot, f|\!)\ (l +\!\!+ r) &= ((\!|\odot, f|\!)\ l) \odot ((\!|\odot, f|\!)\ r) \end{aligned}$$

$(\odot)$ : associative

e.g., $sum = (\!|+, id|\!)$

- List homomorphism is a general form for parallel computations on lists
- How to derive it from sequential programs?

# Third Homomorphism Theorem

**[Gibbons 1995]**

**Theorem.** (Third homomorphism theorem)

$$
\begin{array}{lcl}
h\ [a] & = & f\ a \\
h\ ([a] +\!\!+ x) & = & a \oplus (h\ x) \\
h\ (x +\!\!+ [a]) & = & (h\ x) \otimes a \\
& \Downarrow & \\
h & = & (\!|\odot, f|\!) \quad (\exists \odot : \text{associative})
\end{array}
$$

where $l \odot r = h\ (g\ l +\!\!+ g\ r)$

where $h \circ g \circ h = h$

- Both leftward and rightward (sequential)
  $\Rightarrow$ List homomorphism (parallel)

# The Course of Our Research

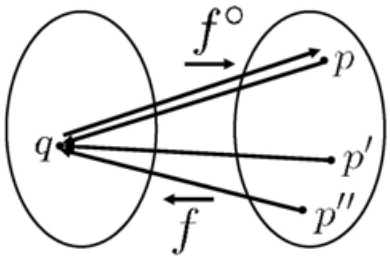
- We have struggled to obtain $g$ such that $h \circ g \circ h = h$ in a systematic manner
  - We have developed a notion of *weak inversion*
  - This work is still ongoing...
- We present how weak inversion derives parallel programs effectively
  - Demonstrate a new derivation of a parallel program for the maximum segment sum problem

# Weak Inversion

**Definition.** (Weak Inversion)

A weak inversion of $f$ is a function $f^\circ$ s.t. $\forall q \in \text{range}(f),\ f^\circ\ q = p \ \Rightarrow\ f\ p = q$

- $f^\circ$ restores one of original inputs of $f$ (from the result of $f$)

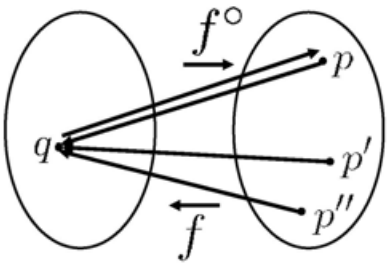


## Examples of Weak Inversion

$$
\begin{array}{lcl}
(n+)^\circ\ a & = & a-n \\
(+)^\circ\ a & = & (a,0) \\
sum^\circ\ a & = & [a] \\
sort^\circ\ x & = & x \\
head^\circ\ a & = & [a]
\end{array}
$$

$$(foldr1\ (\oplus))^\circ\ a\ =\ [a]$$

where
$$
\begin{array}{lcl}
foldr1\ (\oplus)\ [a] & = & a \\
foldr1\ (\oplus)\ (a:x) & = & a \oplus (foldr1\ (\oplus)\ x)
\end{array}
$$




## The Property of Weak Inversion

- Weak inversion is a relative of the inverse function
  - $f^\circ$ is the inverse of $f$ whenever $f$ is bijective
- Weak inversion exists at least one for any function
  - Inverse function exists at most one
- $f \circ f^\circ \circ f = f$
  - Weak inversion derives list homomorphisms

## A Simple Example : *sum*

$$
\begin{array}{lcl}
sum\ [a] & = & a \\
sum\ ([a] +\!\!+ x) & = & a + (sum\ x) \\
sum\ (x +\!\!+ [a]) & = & (sum\ x) + a
\end{array}
$$

⇓ Third homomorphism theorem

$$sum\ =\ (\!|\odot, id|\!)$$

where
$$
\begin{array}{lcl}
l \odot r & = & sum\ (sum^\circ\ l +\!\!+ sum^\circ\ r) \\
 & = & \{\ sum^\circ\ a = [a]\ \} \\
 & & sum\ ([l] +\!\!+ [r]) \\
 & = & sum\ [l,r] \\
 & = & l + r
\end{array}
$$

# The More Involved Example : Maximum Segment Sum

- Maximum Segment Sum problem (*mss*)

$$\begin{aligned} mss\ [2,-6,\boxed{5,-3,4,8},-6,5,-2] \\ = 5+(-3)+4+8 \\ = 14 \end{aligned}$$

  - *mss* is the problem to compute the maximum sums of all contiguous segments
  - How to parallelize it?

# Exercises before *mss* : *mis* & *mts*

- Maximum Initial-segment Sum (*mis*)

$$\begin{aligned} mis\ [\boxed{2,-6,5,-3,4,8},-6,5] \\ = 2+(-6)+5+(-3)+4+8 \\ = 10 \end{aligned}$$

- Maximum Tail-segment Sum (*mts*)

$$\begin{aligned} mts\ [2,-6,\boxed{5,-3,4,8,-6,5}] \\ = 5+(-3)+4+8+(-6)+5 \\ = 13 \end{aligned}$$

# Leftward & Rightward *mis* / *mts*

- *mis* is leftward but **NOT** rightward
  - *mts* is rightward but not leftward
- The following pair is both left / rightward

$$\begin{aligned} \langle mis \times sum \rangle\,[a] &= (a,a) \\ \langle mis \times sum \rangle\,([a] +\!\!+ x) &= \textbf{let } (i,s) = \langle mis \times sum \rangle\, x \\ &\quad \textbf{in } ((a+i)\uparrow a,\ a+s) \\ \langle mis \times sum \rangle\,(x +\!\!+ [a]) &= \textbf{let } (i,s) = \langle mis \times sum \rangle\, x \\ &\quad \textbf{in } (i\uparrow(s+a),\ s+a) \end{aligned}$$

## Constructing a Weak Inversion of *mis*

$$sum^{\circ}\ s\ =\ [s]$$

$$\Downarrow$$

$$\langle mis \times sum\rangle^{\circ}\ (i,s)\ =\ [i, s-i]$$

Note: $(s-i) \leq 0$ because $i \geq s$

e.g., $\langle mis \times sum\rangle^{\circ}\ (3,1) = [3,-2]$

$sum\ [3,-2]\ =\ 1$

$mis\ [3,-2]\ =\ 3$

## Deriving a Parallel Program for *mis*

$$\langle mis \times sum\rangle\ =\ (\!|\odot, f|\!)$$

$$f\ a\ =\ (a,a)$$

$$\begin{aligned}
&(i_l, s_l) \odot (i_r, s_r)\\
&\quad = (\!|\odot, f|\!)((\!|\odot, f|\!)^{\circ}\ (i_l, s_l) +\!\!+ (\!|\odot, f|\!)^{\circ}\ (i_r, s_r))\\
&\quad = (\!|\odot, f|\!)([i_l, s_l - i_l] +\!\!+ [i_r, s_r - i_r])\\
&\quad = (\!|\odot, f|\!)\ [i_l, s_l - i_l, i_r, s_r - i_r]\\
&\quad = (i_l \uparrow (s_l + i_r),\ \ s_l + s_r)
\end{aligned}$$

- This result is the same as a known efficient result

## Maximum Segment Sum Revisited

- Maximum Segment Sum problem (*mss*)

  $mss\ [2, -6, \boxed{5, -3, 4, 8,} -6, 5, -2]$
  $\quad = 14$

  – *mss* itself is **NOT** neither leftward nor rightward

  – The following tuple is both left / rightward

  $$\langle mss \times mis \times mts \times sum\rangle$$

  – We have studied about *mis*, *mts*, and *sum*

  - Can we reuse it?

# Constructing a Weak Inversion of *mss*

$$sum^\circ\ s\ =\ [s]$$
$$\langle mis \times sum\rangle^\circ\ (i,s)\ =\ [i, s-i]$$
$$\langle mis \times mts \times sum\rangle^\circ\ (i,t,s)\ =\ [i, s-t-i, t]$$
$$\Downarrow$$
$$\langle mss \times mis \times mts \times sum\rangle^\circ\ (m,i,t,s)$$
$$=\ [i-m,\ m,\ s-t-i,\ t]$$

Note: $m \geq i \sim t \geq s$ and $i+t \geq m+s$

$\langle mss \times mis \times mts \times sum\rangle^\circ\ (6,3,4,0)\ =\ [-3,6,-7,4]$

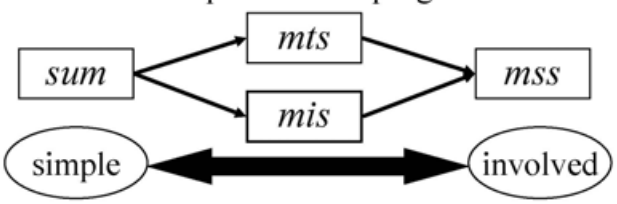
# Deriving a Parallel Program for *mss*

$$\langle mss \times mis \times mts \times sum\rangle\ =\ (\!|\odot, f|\!)$$
$$(m_l,i_l,t_l,s_l) \odot (m_r,i_r,t_r,s_r)$$
$$= (\!|\odot, f|\!)((\!|\odot, f|\!)^\circ\ (m_l,i_l,t_l,s_l) +\!\!+ (\!|\odot, f|\!)^\circ\ (m_r,i_r,t_r,s_r))$$
$$= (\!|\odot, f|\!)([i_l-m_l, m_l, s_l-t_l-i_l, t_l] +\!\!+ [i_r-m_r, m_r, s_r-t_r-i_r, t_r])$$
$$= \text{let } i^* = i_l \uparrow (s_l + i_r)$$
$$\quad t^* = (t_l + s_r) \uparrow t_r$$
$$\quad \text{in } (m_l \uparrow m_r \uparrow i^* \uparrow t^*,\ i^*,\ t^*,\ s_l + s_r)$$

- This result is the same as a known efficient result

# Conclusion

- We have studied weak inversion for parallelization
  - We demonstrated a new derivation of an efficient parallel *mss* program



## Our Work is still Ongoing!

- We need a more systematic way to derive weak inversions
  - How to deal with non-linear variables, tail-recursions, etc.?
  - $(f \circ g)^\circ \neq g^\circ \circ f^\circ$ !?
- There are many examples which are hard to be dealt with by our method
- Weak inversion would have other applications

# Writing practical memory management code with a strictly typed assembly language (Extended Version)

Toshiyuki Maeda
Graduate School of Information Science and Technology, The University of Tokyo
7-3-1, Hongo, Bunkyo-ku,
Tokyo, Japan
tosh@is.s.u-tokyo.ac.jp

Akinori Yonezawa
Graduate School of Information Science and Technology, The University of Tokyo
7-3-1, Hongo, Bunkyo-ku,
Tokyo, Japan
yonezawa@is.s.u-tokyo.ac.jp

## ABSTRACT

Memory management (e.g., malloc/free) cannot be implemented in traditional strictly typed programming languages because they do not allow programmers to reuse memory regions, in order to preserve memory safety. Therefore, they depend on external memory management facilities, such as garbage collection. Thus, many important programs that require explicit memory management (e.g., operating systems) have been written in weakly typed programming languages (e.g., C). To address the problem, we designed a new strictly and statically typed assembly language which is flexible and expressive enough to implement memory management. The key idea of our typed assembly language is that it supports variable-length arrays (the arrays whose size is not known until runtime) as language primitives, maintains integer constraints between variables, and keeps track of pointer aliases. Based on the idea, we implemented a prototype implementation of the language. We also implemented a small operating system kernel which provides a memory management facility with the language.

## 1. INTRODUCTION

Today, computers (PCs, cell-phones, etc.) are widely used in the world and their network becomes one of the indispensable social infrastructures. Therefore, the importance of ensuring safety of software is commonly recognized and many programs come to be written in strictly typed languages (e.g., Java [11], C# [4], Objective Caml [15]).

However, there still exist programs that have not been written in strictly typed languages. For example, existing OSes (e.g., Linux, FreeBSD, Windows XP and Solaris) are written in C and assembly languages. In addition, many Internet servers, for example, Apache httpd server, sendmail mail server and BIND DNS server are written in C. Therefore, many vulnerabilities that may result in a serious security breach (e.g., system crash and/or intrusion) are found in the programs, because it is very hard to ensure and/or verify their safety.

One of the reasons why the programs are written in weakly-typed languages despite the security problem is that existing strictly typed programming languages do not allow programmers to directly manage memory. In simple words, malloc/free cannot be implemented in the languages. For example, Java, C# and Objective Caml depend on an external memory management facility, garbage collection. In the languages, programmers cannot free or reuse memory regions explicitly.

To address the problem, we designed and implemented a strictly and statically typed language which is flexible and expressive enough to implement memory management (malloc/free), and actually implemented memory management code in the language. The language is a variant of Typed Assembly Language (TAL) [14, 13] extended with the support for variable-length arrays (the arrays whose size is not known until runtime) and integer constraints, and the idea of tracking aliases of pointers explicitly [20, 25].

There are two reasons why we chose TAL for writing memory management code. The first reason is that TAL can express low-level memory operations that are necessary for implementing memory management, because TAL is an ordinary assembly language (except for being typed). The second reason is that the TAL programs can be type-checkd at the level of binary executables by annotating the executables with the type information of TAL. This means that we can verify the safety of memory management code with the type-checker without its source code.

The rest of the paper is organized as follows. First, we argue what is required to implement memory management in Sec. 2. Next, we explain our language, called TALK, in Sec. 3. We also describe its type system and type-checking in the section. Next, we explain how memory management can be implemented using TALK in Sec. 4. Then, we briefly describe the prototype implementation of our language and the small toy OS kernel built with the implementation in Sec. 5. Last, we mention related work in Sec. 6 and conclude this paper in Sec. 7.

## 2. REQUISITES FOR IMPLEMENTING MEMORY MANAGEMENT

### 2.1 Variable-length arrays

First of all, memory management code must be able to handle memory. Typically, the memory consists of memory regions. At the lowest level, the memory region is just an array of bytes. One important point is that the array is a variable-length array because its size is not known until runtime. For example, let us think of the program that

is executed just after a system boots. The program cannot make any assumption about the size of the memory, because the amount of the available memory varies from system to system. Therefore, the type system is required to support variable-length arrays. Otherwise we cannot even know the size of the available memory, rather than implement memory management.

## 2.2 Integer constraints

The simplest representation of the variable-length array is a pair of the array and its size. In addition, we introduce special functions for accessing the pairs and do not allow programmers to directly access the pair. With this representation, the type system needs not to maintain the size of the arrays.

However, this approach has a big problem: we cannot implement an allocation function of the pair (the variable-length array) in the type system. This contradicts our goal, designing the statically typed programming language that is able to implement memory management without external trusted memory management facilities.

To avoid the problem, we need to introduce the idea of dependent type [27, 26] to the type system. For example, the type of a variable-length array of integers can be represented as `int`$[\alpha]$. Here, $[\alpha]$ indicates the size of the array, but the exact value, $\alpha$, is not known. The type is a kind of dependent type because it depends on the integer value $\alpha$.

In addition, we need to handle all integers with the dependent type (more specifically, singleton types), as well as the variable-length arrays, because the type information is removed and not available at runtime. For example, let us think of a program that accesses an element of the above array. Let us also suppose that a variable (say `x`) holds an integer index to the array. To ensure memory safety, the type system must be able to check whether the access is safe or not. That is, the type system must be able to check whether the value of `x` is smaller than $\alpha$.

To achieve this, the type system needs to keep track of the value of `x` with the singleton types. For example, the type of `x` must be `int`$(\beta)$, instead of `int`. Here, $\beta$ indicates the value of `x`, but the exact value is not known to the type system. In addition, the type system also needs to keep track of integer constraints (the constraint between $\alpha$ and $\beta$ in this case). For example, if the type system knows that $\alpha > \beta$, the array access is safe. Such constraints are generated by usual branch operations. For example, if the type of a variable `y` is $int(\alpha)$, the branch operation `if y > x {...} else {...}` generates the constraint $\alpha > \beta$ for the taken branch and $\alpha \leq \beta$ for the other branch. Here, $\alpha$ is the size of the variable-length arrays. Therefore, the type system knows that it is safe to access the array by the index `x` in the taken branch.

## 2.3 Alias tracking

The previous section argued how to represent memory in the type system with the variable-length arrays. As the next step, this section discusses how to manage the memory. From the viewpoint of type theory, memory management is almost the same as changing types of memory regions. For example, changing a type of a memory region from a pointer type to an integer type can be viewed as freeing the memory region that contains a pointer and reusing it for holding an integer. Fig. 1 is an example C code which performs this *memory reuse*. The function reuses the memory region pointed by pointer `x` (line 3).

```
1 void pointer_to_int(int** x)
2 {
3     int* y = (int*)x;
4     *y = 1;
5 }
```

**Figure 1: Example of C code that reuses a memory region**

However, existing strictly and statically typed programming languages do not allow programmers to change types of memory regions, because memory safety cannot be ensured. For example, using the function in Fig. 1, we can write a function as in Fig. 2. The function passes the type check of C, but it is apparently unsafe because it tries to dereference an integer which is no longer a pointer (line 4). Thus, memory management cannot be implemented in the existing strictly and statically typed languages and they depend on external memory management mechanisms, such as garbage collection.

```
1 void dangerous_func(int** x)
2 {
3     pointer_to_int(x);
4     **x;
5     ...
```

**Figure 2: Example of C code that breaches memory safety**

The essential problem is that the type system does not know that `y` in the function `pointer_to_int` and the argument `x` of function `dangerous_func` alias, that is, point to the same memory location.

To solve the problem, we need to introduce the idea of alias types [25] to the type system in order to keep track of aliases explicitly. The basic idea of the alias types is to change the representation of pointer types. In usual type systems, the type of a pointer is represented as the type of the memory region pointed by the pointer. In the alias type system, on the other hand, the type of the pointer is just the address of the memory region. The type of the memory region is separately maintained as memory type. The memory type is a map from addresses to the types of the memory regions at the addresses.

For example, based on the idea of alias types, the code in Fig. 1 can be rewritten as in Fig. 3. First, the type of the argument `x` is changed from `int**` to `ptr(p)` (line 2). `ptr(p)` indicates that `x` is a pointer which points to the address `p`. Next, the part surrounded by "[" and "]" represents the memory type. The memory type added before the function indicates the state of the memory before the function is called (line 1). The other memory type added after the function indicates the state of the memory after the function is executed (line 6). Here, the memory type before the function indicates that the memory region at the address `p` has pointer type `ptr(q)` and the memory region at the address `q` has the integer type. Thus, the type system

knows that x is a pointer to a pointer to an integer. Then, the body of the function stores an integer to the memory region at the address p (line 4). Therefore, the memory type after the function indicates that the memory region at the address p is an integer. (Please note that the alias type system ensures that p and q are different integers, because the memory type is a map.)

```
1 [ p --> ptr(q), q --> int]
2 void pointer_to_int(ptr(p) x)
3 {
4     *x = 1;
5 }
6 [ p --> int, q --> int]
```

**Figure 3: Example of pseudo code based on the idea of alias type**

In addition, the code of Fig. 2 can be rewritten as Fig. 4. The type check of the alias type system rejects the rewritten code at line 5, because after the function call (pointer_to_int, line 4), the type of the memory region is changed from a pointer type (ptr(q)) to the integer type (int). Thus, the alias type system allows programmers to reuse memory regions explicitly because it keeps track of aliases in the memory type.

```
1 [ p --> ptr(q), q --> int]
2 void dangerous_func(ptr(p) x)
3 {
4     pointer_to_int(x);
5     **x;
6     ...
```

**Figure 4: Example of pseudo code that causes a type error**

## 2.4 Split and Concatenation of Arrays

As described above, we can handle memory with the variable-length arrays and the integer constraints and reuse regions in the memory by explicitly tracking pointer aliases with alias types. However, we need one more mechanism in the type system to implement practical memory management. For example, let us suppose that free memory (not in use memory) is represented as an array of integers. Then, its memory type is represented as int$[a]$ (here we assume that $a > 0$). Now, let us think of the code in Fig. 5 that allocates one element from the top of the free memory and reuses it as a pointer to integer (line 4). The access to the array is obviously safe because $a > 0$. However, there is a problem in how to represent the memory type of the memory after the memory reuse. More specifically, the problem is that it is difficult to represent the variable-length array whose elements are integers except for its first element.

To solve the problem, we need a notion of split (and concatenation) of arrays in the type system. For example, memory type [ p --> int[a] ], which indicates that there is an array of size a at address p, can be split to memory type [ p --> int[a1], p2 --> int[a2] ], which indicates that there is one array of size a1 at address p and the other array

```
1 [ p --> int[a], q --> int] where a > 0
2 void alloc_and_reuse(ptr(p) x, ptr(q) y)
3 {
4     x[0] = y;
5     ...
```

**Figure 5: Example of pseudo code that allocates a pointer to integer from free memory (incomplete)**

of size a2 at address p2, where a = a1 + a2 and p2 = p + a1. In addition, the latter memory type can be concatenated back to the former memory type.

With the notion of the split of the arrays, the code of Fig. 5 can be rewritten as in Fig. 6. The split operation (line 4) splits the free memory into new array of size 1 at p and the rest of the free memory at p + 1. In this case, we can naturally represent the memory type of the free memory after line 5 as [ p --> ptr(q), (p + 1) --> int[a - 1] ].

```
1 [ p --> int[a], q --> int] where a > 0
2 void alloc_and_reuse(ptr(p) x, ptr(q) y)
3 {
4     split p, 1;
5     x[0] = y;
6     ...
```

**Figure 6: Example of pseudo code that allocates a pointer to integer from free memory (complete)**

# 3. OUR LANGUAGE: TALK

This section introduces our strictly typed assembly language for writing memory management code. Although the language explained in this section is based on a virtual CPU architecture, the actual implementation is based on the IA-32 [10] assembly language. In this section, we first explain its abstract machine and types. Then, its operational semantics and typing rules are described. The syntax of our language is shown in Fig. 7 and Fig.8.

## 3.1 Abstract Machine

A state of the abstract machine consists of program $P$, memory $M$, registers $R$ and instructions $I$. The instructions $I$ is explained in Sec. 3.3. The program $P$ is a map from label $l$ to the instructions $I$. The registers $R$ is a map from register $r$ to value $v$.

The memory $M$ is a map from an integer $n$ to heap value $h$. The heap value $h$ is array $a$ or stack $s$. The array $a$ consists of tuples $t$, and the tuples consist of values $v$. (roll $(t)$ and pack$_{[c_1,\ldots,c_n|M]}$ $(t)$ are introduced only for formal arguments of recursive types and existential types.) The value $v$ is integer $n$ or the label $l$. (The suffix of the label $l$ ($[c_1, \ldots, c_n/\Delta]$)is required only by type checkers. It has no meaning at runtime.) The stack $s$ consists of the values $v$. Strictly speaking, the stacks are not necessary because they can be represented by the arrays. However, in this paper, we introduce the stacks for ease of understating examples. In addition, we assume that the stacks are unbounded in

this paper. The actual implementation handles the bounded stacks properly.

$$
\begin{array}{llll}
(\textit{state}) & S & ::= & (P, M, R, I) \\
(\textit{prog.}) & P & ::= & \cdot \mid \{l \mapsto I\}P \\
(\textit{memory}) & M & ::= & \cdot \mid \{n \mapsto h\}M \\
(\textit{regs.}) & R & ::= & \{\mathtt{r1} \mapsto v_1, \ldots, \mathtt{rn} \mapsto v_n\} \\
(\textit{register}) & r & ::= & \mathtt{r1} \mid \ldots \mid \mathtt{rn} \\
(\textit{heap}) & h & ::= & a \mid s \\
(\textit{array}) & a & ::= & \langle t_1, \ldots, t_n \rangle \\
(\textit{tuple}) & t & ::= & \langle v_1, \ldots, v_n \rangle \mid \mathtt{roll}\,(t) \\
 & & & \mid \mathtt{pack}_{[c_1,\ldots,c_n \mid M]}\,(t) \\
(\textit{stack}) & s & ::= & \cdot \mid v :: s \\
(\textit{value}) & v & ::= & n \mid l\,[c_1, \ldots, c_n/\Delta] \\
(\textit{integer}) & n & & \\
(\textit{label}) & l & & \\
(\textit{insts.}) & I & ::= & \mathtt{ld}\,[r_s + n], r_d; I \mid \mathtt{st}\,r_s, [r_d + n]; I \\
 & & & \mid \mathtt{mov}\,r_s, r_d; I \mid \mathtt{movi}\,v, r_d; I \mid \mathtt{add}\,r_{s1}, r_{s2}, r_d; I \\
 & & & \mid \mathtt{sub}\,r_{s1}, r_{s2}, r_d; I \mid \mathtt{mul}\,r_{s1}, r_{s2}, r_d; I \mid \mathtt{push}\,r_s, [r_d]; I \\
 & & & \mid \mathtt{pop}\,[r_s], r_d; I \mid \mathtt{beq}\,r_{s1}, r_{s2}, r_d; I \mid \mathtt{ble}\,r_{s1}, r_{s2}, r_d; I \\
 & & & \mid \mathtt{jmp}\,r_d \mid \mathtt{apply}\,r\,[c_1, \ldots, c_n/\Delta]\,; I \\
 & & & \mid \mathtt{roll}_{\mu\eta[\Delta].\tau(c_1,\ldots,c_n)}\,i; I \mid \mathtt{unroll}\,i; I \\
 & & & \mid \mathtt{pack}_{[c_1,\ldots,c_n \mid \Sigma_1] as \exists\Delta.\mid C \mid [\Sigma_2].\tau}\,i; I \mid \mathtt{unpack}\,i\ \mathtt{with}\ \Delta; I \\
 & & & \mid \mathtt{split}\,i_1, i_2; I \mid \mathtt{concat}\,i_1, i_2, i_3; I \\
 & & & \mid \mathtt{tuple_split}\,i_1, n_2; I \mid \mathtt{tuple_concat}\,i_1, i_2; I
\end{array}
$$

**Figure 7: Syntax of Abstract Machine**

## 3.2 Types

The type of integers is represented by $i$. The integer type $i$ is integer constants $n$, type variables $\alpha$ or the result of integer arithmetic operations $i_1\ aop\ i_2$. For example, if a certain register $r$ has the integer type 3, the register $r$ holds the value 3. In addition, if two registers $r_1$ and $r_2$ have the same type $\alpha$, we know that $r_1 = r_2$, though the exact values of $r_1$ and $r_2$ is not known.

The type of memory is represented as $\Sigma$. The memory type $\Sigma$ consists of maps from the integer type $i$ to heap type $ht$, and type variables $\epsilon$. For example, $\{\mathtt{0x}c0345810 \mapsto ht\}$ represents memory in which there is only one heap value of the type $ht$ at the address $\mathtt{0x}c0345810$. On the other hand, $\{\mathtt{0x}c0345810 \mapsto ht\} \otimes \epsilon$ represents memory in which there is a heap value of the type $ht$ at the address $\mathtt{0x}c0345810$ and there may be some other data (or not). The type variable $\epsilon$ indicates that there may be some other data in the memory.

The heap type $ht$ is array type $at$ or stack type $st$. The array type $at$ is written as $\tau\,[i]$. This represents an array whose elements have the type $\tau$ and whose size is $i$. Because we can use type variables for representing sizes of arrays, we can deal with the variable-length arrays.

The type of elements of arrays is the tuple type $\tau$. There are three kinds of the tuple type. $\langle \sigma_1, \ldots, \sigma_n \rangle$ represents a type of tuples whose elements have types $\sigma_i$. $\exists\Delta.\,|C|\,[\Sigma]\,.\tau$ represents the type of a tuple which is packed as an existential type. (The details of existential types are explained later.) $\rho\,(c_1, \ldots, c_n)$ is a (parametric) recursive type for recursive data structures. The type of elements of tuples, $\sigma$, can be the integer type $i$ or label type $lt$.

The stack type $st$ consists of null stack type $(\cdot)$, a pair of the small value type $\sigma$ which represents the top of the stack and the stack type $st$ which represents the rest of the stack, and type variable $\gamma$. For example, $42 :: \gamma$ represents the type of a stack whose top element is an integer value (42) and the rest is unknown ($\gamma$).

The label type is written as $\forall\Delta.\,|C|\,[\Sigma]\,(\Gamma)$. It indicates a constraint condition that must be satisfied whenever a control flow reaches the label. First, $\Delta$ represents a set of type variables. This means that the instructions of the label is polymorphic over the type variables. Next, $C$ represents integer constraints. The instructions of the label are type-checked under the assumption that the constraints are satisfied, because our typing rules ensure that the constraints are satisfied at all the points of jumping to the label. Then, $\Sigma$ is the memory type described above. As with the integer constraints, the instructions of the label are type-checked under the assumption that the memory has the memory type $\Sigma$, because our typing rules ensure that the memory has the type $\Sigma$ at all the points of jumping to the label. Last, the registers type $\Gamma$ indicates the condition for registers that must be satisfied whenever execution reaches the label. For example, $\forall\alpha, \beta, \epsilon.\,|\beta \leq 128|\,[\epsilon \otimes \{\alpha \mapsto \langle 0 \rangle\,[\beta]\}]\,(\mathtt{r1} : \alpha)$ represents instructions that take a pointer (register $\mathtt{r1}$) to an array whose size is not greater than 128 and whose elements are 0.

Additionally, the existential type $\exists\Delta.\,|C|\,[\Sigma]\,\tau$ represents tuples that have the type $\tau$, and indicates that the integer constraints $C$ are satisfied and there exists memory whose type is $\Sigma$. For example, $\exists\alpha, \beta.\,|\beta \leq 128|\,[\{\alpha \mapsto \langle 0 \rangle\,[\beta]\}]\,\langle \alpha \rangle$ represents a tuple whose only element is a pointer to an array whose size is not greater than 128, and ensures that the array exists surely.

In addition, the actual implementation of our language supports a variant type, but we do not explain it in this paper for brevity.

The program type $\Phi$ represents the type of program $P$. It is a map from the label $l$ to the label type $lt$.

$$
\begin{array}{llll}
(\textit{label type}) & lt & ::= & \forall\Delta.\,|C|\,[\Sigma]\,(\Gamma) \\
(\textit{small type}) & \sigma & ::= & i \mid lt \\
(\textit{integer type}) & i & ::= & n \mid \alpha \mid i_1\ aop\ i_2 \\
(\textit{type var.}) & \delta & ::= & \alpha, \gamma, \epsilon \\
(\textit{type vars.}) & \Delta & ::= & \cdot \mid \delta, \Delta \\
(\textit{type}) & \tau & ::= & \langle \sigma_1, \ldots, \sigma_n \rangle \mid \exists\Delta.\,|C|\,[\Sigma]\,\tau \\
 & & & \mid \rho\,(c_1, \ldots, c_n) \\
(\textit{type scheme}) & \rho & ::= & \eta \mid \mu\eta\,[\Delta]\,.\tau \\
(\textit{array type}) & at & ::= & \tau\,[i] \mid \tau\ (= \tau\,[1]) \\
(\textit{stack type}) & st & ::= & \cdot \mid \sigma :: st \mid \gamma \\
(\textit{heap type}) & ht & ::= & at \mid st \\
(\textit{memory type}) & \Sigma & ::= & \cdot \mid \Sigma \otimes \{i \mapsto ht\} \mid \Sigma \otimes \epsilon \\
(\textit{regs. type}) & \Gamma & ::= & \cdot \mid \{r \mapsto \sigma\}\Gamma \\
(\textit{prog. type}) & \Phi & ::= & \cdot \mid \{l \mapsto lt\}\Phi \\
(\textit{constructor}) & c & ::= & i \mid \Sigma \mid st \\
(\textit{constraints}) & C & ::= & \cdot \mid i_1\ cop\ i_2 \\
 & & & \mid C \wedge C \mid C \vee C \mid \neg C \\
(\textit{compareop.}) & cop & ::= & = \mid < \mid \leq \mid > \mid \geq \\
(\textit{arithop.}) & aop & ::= & + \mid - \mid *
\end{array}
$$

**Figure 8: Syntax of Types**

## 3.3 Instructions and Operational Semantics

There are two kinds of instructions in our language. One is the ordinary instructions that update the state of the abstract machine. The other is the coerce instructions that update only the type information when type-checking. Fig. 9 and Fig. 10 represent their operational semantics. (In this paper, $e\,[b/a]$ represents a capture-avoiding substitution of $b$ for free variable $a$ in $e$. In addition, $e\,[b_1, b_2/a_1, a_2]$ is an abbreviation of $e\,[b_1/a_1, b_2/a_2]$.)

### 3.3.1 Ordinary Instructions

There are 12 ordinary instructions in our language. `ld` $[r_s + n], r_d$ is a load instruction which loads $n$th element of a tuple which resides in the address specified by the register $r_s$ and stores the element to the register $r_d$. `st` $r_s, [r_d + n]$ is a store instruction which stores the value of the register $r_s$ into $n$th element of a tuple which resides in the address specified by the register $r_d$.

`mov` $r_s, r_d$ is a register-copy instruction which just copies the value of the register $r_s$ to the register $r_d$. `movi` $v, r_d$ is a constant-load instruction which loads the value $v$ to the register $r_d$.

`add` $r_{s1}, r_{s2}, r_d$ is an add instruction which stores the sum of $r_{s1}$ and $r_{s2}$ into the register $r_d$. `sub` and `mul` is a subtraction and multiplication instruction, respectively. In our language, there is no reference types or pointer types. Memory addresses are only integers. Therefore, the address calculation are performed with the arithmetic instructions.

`push` $r_s, [r_d]$ is an instruction for manipulating memory stacks. It pushes the value stored in the register $r_s$ to the stack that resides in the address specified by the register $r_d$. Then, it decrements the register $r_d$ by one. `pop` $[r_s], r_d$ is the other instruction for manipulating memory stacks. It pops the top of the stack that resides in the address specified by the register $r_s$ and stores the value to the register $r_d$. Then, it increments the register $r_s$ by one.

`beq` $r_{s1}, r_{s2}, r_d$ is a branch instruction which jumps to the label specified by the register $r_d$ if $r_{s1} = r_{s2}$. `ble` $r_{s1}, r_{s2}, r_d$ is the other branch instruction which jumps to the label specified by the register $r_d$ if $r_{s1} \leq r_{s2}$.

`jmp` $r_d$ is a jump instruction which jumps to the label specified by the register $r_d$ and executes the instructions of the label.

### 3.3.2 Coerce Instructions

There are 9 coerce instructions for manipulating type information when type-checking. The instructions incur no runtime overhead because they are interpreted only by type checkers and not executed at runtime.

`apply` $r\,[c_1, \ldots, c_n/\Delta]$ is a type application instruction which substitutes $c_1, \ldots, c_n$ for the type variables $\Delta$ that are bound by the type of the label specified by the register $r$. A type variable ($\delta$) is an integer type variable ($\alpha$), a stack type variable ($\gamma$) or a memory type variable ($\epsilon$).

`roll`$_{\mu\eta[\Delta].\tau(c_1,\ldots,c_n)}$ $i$ and `unroll` $i$ are instructions for recursive types which unroll a recursive type once (`unroll`) and vice versa (`roll`).

`pack`$_{[c_1,\ldots,c_n|\Sigma]as\ \tau}$ $i$ and `unpack` $i$ `with` $\Delta$ are instructions for existential types which pack the type of the tuple that resides in the address $i$ into an existential type (`pack`) and vice versa (`unpack`). As in the alias type system [25], we can hide part of memory in existential types. The encapsulated memory cannot be accessed unless the existential type is unpacked.

`split` $i_1, i_2$ and `concat` $i_1, i_2, i_3$ are instructions for the array types. `split` splits an array type into two adjacent array types and `concat` concatenates two adjacent array types into one array type. These instructions are used to access an element of an array (see Sec. 3.4.4 for details). In addition, they are useful for implementing memory management facilities (see Sec. 4 for details).

`tuple_split` $i_1, n_2$ and `tuple_concat` $i_1, i_2$ resemble `split` and `concat`, but for tuple types. `tuple_concat` is used for creating a tuple type from adjacent arrays of size 1, and `tuple_split` is vice versa. In our language, allocation of a tuple can be represented as follows. First, an array is obtained by `split` from one of the arrays that represent the free memory. Then, the obtained array is further split into adjacent arrays of size 1. Then the arrays are concatenated into a tuple by `tuple_concat` (see the example of Sec. 4 for details)

## 3.4 Typing Rules

Typing rules are shown in Fig. 11, Fig. 12 and Fig. 13. $\vdash S$ states that the abstract machine state $S$ is well-formed. If $(P, M, R, I)$ is well-formed, there exists a certain state $(P, M', R', I')$ which is also well-formed and $(P, M, R, I) \mapsto_S (P, M', R', I')$, that is, no runtime errors occur. Strictly speaking, the above statement holds true if all the stacks in memory never conflict with other heap values. This is because the language presented in this paper assumes that the stacks can grow infinitely, but this is impossible in practice. Thus, we can prove the above statement except for the rules for the stacks. This is only a minor matter, because the stacks are not necessary as mentioned in Sec. 3.1

Our type system is based on the one proposed in [25]. Although Walker and Morrisett proved type soundness in [25], their language falls short of a practical language. This is why, for the purpose of writing practical memory management code, we essentially added a variable-length array type to their type system. We claim that those changes are unlikely to break type soundness, because, in TALK, the strong updates that may change types of memory regions are only for tuples (the arrays of size 1) as described in Sec. 3.4.4. This is almost the same as in the type system of [25]. The only concern is the effect of split/concatenation of variable-length arrays, but the typing rules for them are quite straightforward.

The judgement of abstract machine states consists of the judgement of program, memory, registers and instructions.

### 3.4.1 Well-formedness of program

$\vdash P : \Phi$ states that the program $P$ is well-formed (PROGRAM). The rule checks whether all the labels in the program are typed in the program type $\Phi$. It also checks whether each block of instructions in the program is well-formed according to its label type specified in $\Phi$. The well-formedness of instructions are described in Sec. 3.4.4.

### 3.4.2 Well-formedness of registers

$\vdash R : \Gamma$ states that the registers $R$ is well-formed (REGISTERS). The rule checks whether the value of each register has the small value type specified in the registers type $\Gamma$.

### 3.4.3 Well-formedness of memory

$\vdash M : \Sigma$ states that the memory $M$ has the memory type $\Sigma$ (MEMORY). The judgement rule checks whether

$$
\begin{array}{lcl}
(P, M\{R(r_s) \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}, R, \texttt{ld}\ [r_s + n'], r_d; I) & \mapsto_S & (P, M\{R(r_s) \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}, R\{r_d \mapsto v_{n'}\}, I) \\
(P, M\{R(r_d) \mapsto \langle\langle \ldots, v_{n'}, \ldots\rangle\rangle\}, R, \texttt{st}\ r_s, [r_d + n']; I) & \mapsto_S & (P, M\{R(r_d) \mapsto \langle\langle \ldots, R(r_s), \ldots\rangle\rangle\}, R, I) \\
(P, M, R, \texttt{mov}\ r_s, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_s)\}, I) \\
(P, M, R, \texttt{movi}\ v, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto v\}, I) \\
(P, M, R, \texttt{add}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_{s2}) + R(r_{s1})\}, I) \\
(P, M, R, \texttt{sub}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_{s2}) - R(r_{s1})\}, I) \\
(P, M, R, \texttt{mul}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_{s2}) * R(r_{s1})\}, I) \\
(P, M\{R(r_d) \mapsto s\}, R, \texttt{push}\ r_s, [r_d]; I) & \mapsto_S & (P, M\{R(r_d) - 1 \mapsto R(r_s) :: s\}, R\{r_d \mapsto R(r_d) - 1\}, I) \\
(P, M\{R(r_s) \mapsto v :: s\}, R, \texttt{pop}\ [r_s], r_d; I) & \mapsto_S & (P, M\{R(r_s) + 1 \mapsto s\}, R\{r_d \mapsto v\}\{r_s \mapsto R(r_s) + 1\}, I) \\
(P, M, R, \texttt{beq}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & \textit{if } R(r_{s1}) = R(r_{s2}) \textit{ then } (P, M, R, P(l)\,[c_1, \ldots, c_n/\Delta]) \\
 & & \textit{else } (P, M, R, I) \\
 & & \textit{where } R(r_d) = l\,[c_1, \ldots, c_n/\Delta] \\
(P, M, R, \texttt{ble}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & \textit{if } R(r_{s1}) \leq R(r_{s2}) \textit{ then } (P, M, R, P(l)\,[c_1, \ldots, c_n/\Delta]) \\
 & & \textit{else } (P, M, R, I) \\
 & & \textit{where } R(r_d) = l\,[c_1, \ldots, c_n/\Delta] \\
(P, M, R, \texttt{jmp}\ r_d) & \mapsto_S & (P, M, R, P(l)\,[c_1, \ldots, c_n/\Delta]) \\
 & & \textit{where } R(r_d) = l\,[c_1, \ldots, c_n/\Delta]
\end{array}
$$

**Figure 9: Operational Semantics (instructions)**

$$
\begin{array}{lcl}
(P, M, R, \texttt{apply}\ r_d\,[c_1, \ldots, c_n/\Delta]\,; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_d)\,[c_1, \ldots, c_n/\Delta]\}, I) \\
 & & \textit{where } R(r_d) = l\,[c'_1, \ldots, c'_m/\Delta'] \quad (l \in P) \\
(P, M\{n \mapsto \langle t\rangle\}, R, \texttt{roll}_\tau\ n; I) & \mapsto_S & (P, M\{n \mapsto \langle \texttt{roll}\,(t)\rangle\}, R, I) \\
(P, M\{n \mapsto \langle \texttt{roll}\,(t)\rangle\}, R, \texttt{unroll}\ n; I) & \mapsto_S & (P, M\{n \mapsto \langle t\rangle\}, R, I) \\
\Big(P, M\{n \mapsto \langle t\rangle\}M', R, \texttt{pack}_{[c_1,\ldots,c_n|\Sigma] as\ \tau}\ n; I\Big) & \mapsto_S & \Big(P, M\{n \mapsto \Big\langle \texttt{pack}_{[c_1,\ldots,c_n|M']}\,(t)\Big\rangle\}, R, I\Big) \\
 & & \textit{where } Dom(M') \subseteq Dom(\Sigma) \\
(P, M\{n \mapsto \Big\langle \texttt{pack}_{[c_1,\ldots,c_n|M']}\,(t)\Big\rangle\}, R, & & \\
\qquad \texttt{unpack}\ n\ \texttt{with}\ \Delta; I) & \mapsto_S & (P, MM'\{n \mapsto \langle t\rangle\}, R, [c1, \ldots, c_n/\Delta]I) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_n\rangle\}, R, \texttt{split}\ n_1, n_2; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle t_1, \ldots, t_{n_2}\rangle\}\{n'_1 \mapsto \langle t_{n_2+1}, \ldots, t_n\rangle\}, R, I) \\
 & & \textit{where } 0 < n_2 < n \textit{ and } n'_1 = n_1 + \textstyle\sum_{i=1}^{n_2} \textit{sizeof } (t_i) \\
(P, M, R, \texttt{split}\ n_1, 0; I) & \mapsto_S & (P, M, R, I) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_n\rangle\}, R, \texttt{split}\ n_1, n; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle t_1, \ldots, t_n\rangle\}, R, I) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_n\rangle\}\{n_2 \mapsto \langle t'_1, \ldots, t'_m\rangle\}, R, & & \\
\qquad \texttt{concat}\ n_1, n_2, m; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle t_1, \ldots, t_n, t'_1, \ldots, t'_m\rangle\}, R; I) \\
 & & \textit{where } m > 0 \textit{ and } n_2 = n_1 + \textstyle\sum_{i=1}^{n} \textit{sizeof } (t_i) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_n\rangle\}, R, \texttt{concat}\ n_1, n_2, 0; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle t_1, \ldots, t_n\rangle\}, R, I) \\
 & & \textit{where } n_2 = n_1 + \textstyle\sum_{i=1}^{n} \textit{sizeof } (t_i) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_m\rangle\}, R, \texttt{concat}\ n_1, n_1, m; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle t_1, \ldots, t_m\rangle\}, R, I) \\
 & & \textit{where } m > 0 \\
(P, M, R, \texttt{concat}\ n, n, 0; I) & \mapsto_S & (P, M, R, I) \\
(P, M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}, R, \texttt{tuple_split}\ n_1, n_2; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle\langle v_1, \ldots, v_{n_2}\rangle\rangle\}\{n'_1 \mapsto \langle\langle v_{n_2+1}, \ldots, v_n\rangle\rangle\}, R, I) \\
 & & \textit{where } n'_1 = n_1 + n_2 \\
(P, M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}\{n_2 \mapsto \langle\langle v'_1, \ldots, v'_m\rangle\rangle\}, R, & & \\
\qquad \texttt{tuple_concat}\ n_1, n_2; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n, v'_1, \ldots, v'_m\rangle\rangle\}, R; I) \\
 & & \textit{where } n_2 = n_1 + n
\end{array}
$$

**Figure 10: Operational Semantics (coerce)**

the heap value $M(n)$ has the heap type $\Sigma(n)$ for each address $n \in Dom(M)$. In addition, the rule states that all the heap values in the memory (including encapsulated memory regions inside existential packages) do not overlap each other (which is denoted as $\texttt{GU}(M)$) in order to keep track of pointer aliases properly.

$\Delta; C \vdash a : at$ states that, with the type variables $\Delta$ and under the assumption that the integer constraints $C$ are satisfied, the array $a$ has the array type $at$. The typing rule ARRAY checks whether all the elements of the array have the same tuple type $\tau$ and the size of the array equals to the size specified in the array type. For example, $\Delta; C \vdash \langle t_1, t_2 \rangle : \tau\,[i]$ checks whether the tuples $t_1$ and $t_2$ have the type $\tau$. It also checks whether $i = 2$ under the assumption $C$, using a constraint solver. We write this as $\Delta; C \models i = 2$. We do not show the rules for $\Delta; C \models C'$ (read as $C'$ can be deduced from $C$) in this paper. It is well-known that the problem of integer constraints is decidable if the constraints are linear. The only instruction that may introduce a non-linear constraint is the `mul` instruction.

$\Delta; C \vdash s : st$ states that, with the type variables $\Delta$ and under the assumption that the integer constraints $C$ are satisfied, the stack $s$ has the stack type $st$.

$\Delta; C \vdash t : \tau$ states that the element $t$ of an array has the type $\tau$. The typing rule TUPLE checks whether each element $(v_i)$ of an tuple has the type $(\sigma_i)$ specified in the tuple type. The typing rule TUPLE_PACK is for existential types and the typing rule TUPLE_ROLL is for recursive types.

$\Delta; C \vdash v : \sigma$ states that the value $v$ has the type $\sigma$. There are two typing rules for integers (VALUE_INTEGER) and labels (VALUE_LABEL). The VALUE_INTEGER rule checks whether the integer $n$ equals to the type $i$ using a constraint solver ($\Delta; C \models n = i$). The VALUE_LABEL rule checks whether the type of the label $l$ can be instantiated to the specified label type $\sigma$ according to the substitution $[c_1, \ldots, c_n/\Delta'']$.

### 3.4.4 Well-formedness of instructions

$\Delta; \Gamma; C; \Sigma \vdash I$ states that the instructions $I$ is well-formed with the type variables $\Delta$, with the registers that satisfies the registers type $\Gamma$ and under the assumption that the integer constraints $C$ are satisfied and the memory has the memory type $\Sigma$.

The typing rule LOAD is for type-checking the load instruction. First, the rule checks whether the value of the register $r_s$ is a valid memory address in the memory type $\Sigma$ and an array resides in the address. Then, it checks whether the size of the array equals to 1 and the size of the tuple that is only element of the array is larger than $n$. Finally, it checks the rest of instructions $I$ under the new register type that is modified so that the register $r_d$ has the type $\sigma_n$ that represents the loaded value.

The typing rule STORE is for type-checking the store instruction. As with LOAD, it checks whether the value of the register $r_d$ is a valid memory address in the memory type $\Sigma$ and an array resides in the address. Then, it checks whether the size of the array that resides in the address equals to 1 and the size of the tuple that is only element of the array is larger than $n$. Finally, it checks the rest of instructions $I$ under the modified memory type such that the $n$th element of the tuple that resides in the address is replaced with the type of $r_s$.

$$\frac{\vdash P : \Phi \qquad \vdash M : \Sigma \qquad \vdash R : \Gamma \qquad \cdot;\Gamma;\cdot;\Sigma \vdash I}{\vdash (P, M, R, I)} \text{ (STATE)}$$

$$\frac{\begin{array}{c} Dom(P) = Dom(\Phi) \\ \forall l \in Dom(P).\ \Delta; \Gamma; C; \Sigma \vdash P(l) \qquad \Phi(l) \equiv \forall \Delta.\, |C|\,[\Sigma]\,(\Gamma) \end{array}}{\vdash P : \Phi} \text{ (PROGRAM)}$$

$$\frac{\begin{array}{c} \texttt{GU}(M) \qquad M \equiv \{n_1 \mapsto a_1\} \ldots \{n_k \mapsto a_k\} \\ \Sigma' \equiv \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_k \mapsto at_k\} \\ \forall i.\ \cdot;\cdot \vdash a_i : at_i \qquad \cdot;\cdot \vdash \Sigma = \Sigma' \end{array}}{\vdash M : \Sigma} \text{ (MEMORY)}$$

$$\frac{\forall r_i \in Dom(\Gamma).\ \cdot;\cdot \vdash R(r_i) : \Gamma(r_i)}{\vdash R : \Gamma} \text{ (REGISTERS)}$$

$$\frac{\Delta; C \vdash t_j : \tau \qquad \Delta; C \models n = i}{\Delta; C \vdash \langle t_1, \ldots, t_n \rangle : \tau\,[i]} \text{ (ARRAY)}$$

$$\frac{\Delta; C \vdash v_j : \sigma_j}{\Delta; C \vdash v_1 :: \ldots :: v_n : \sigma_1 :: \ldots :: \sigma_n} \text{ (STACK)}$$

$$\frac{\Delta; C \vdash v_j : \sigma_j}{\Delta; C \vdash \langle v_1, \ldots, v_n \rangle : \langle \sigma_1, \ldots, \sigma_n \rangle} \text{ (TUPLE)}$$

$$\frac{\begin{array}{c} \tau \equiv \mu\eta\,[\Delta'].\tau'\,(c_1, \ldots, c_n) \\ \Delta; C \vdash t : \tau'\,[\mu\eta\,[\Delta'].\tau'/\eta]\,[c_1, \ldots, c_n/\Delta'] \end{array}}{\Delta; C \vdash \texttt{roll}\,(t) : \tau} \text{ (TUPLE_ROLL)}$$

$$\frac{\begin{array}{c} \Delta; C \vdash t : \tau'\,[c_1, \ldots, c_n/\Delta'] \qquad \tau \equiv \exists \Delta'.\, |C'|\,[\Sigma']\,\tau' \\ \vdash M : \Sigma'\,[c_1, \ldots, c_n/\Delta'] \qquad \Delta; C \models C'\,[c_1, \ldots, c_n/\Delta'] \end{array}}{\Delta; C \vdash \texttt{pack}_{[c_1, \ldots, c_n | M]}\,(t) : \tau} \text{ (TUPLE_PACK)}$$

$$\frac{\Delta; C \models n = i}{\Delta; C \vdash n : i} \text{ (VALUE_INTEGER)}$$

$$\frac{\begin{array}{c} \forall \Delta'.\, |C'|\,[\Sigma']\,(\Gamma') \equiv \Phi\,(l) \qquad \theta \equiv [c_1, \ldots, c_n/\Delta''] \\ C'' \equiv C'\theta \qquad \Sigma'' \equiv \Sigma'\theta \qquad \Gamma'' \equiv \Gamma'\theta \\ \Delta; C \vdash \sigma = \forall \Delta' \backslash \Delta''.\, |C''|\,[\Sigma'']\,(\Gamma'') \end{array}}{\Delta; C \vdash l\,[c_1, \ldots, c_n/\Delta''] : \forall \Delta. |C_1|[\Sigma_1](\Gamma_1)} \text{ (VALUE_LABEL)}$$

**Figure 11: Typing rules (machine state)**

Because LOAD and STORE only permit load/store operations for arrays whose size is 1, to access an array whose size is greater than 1, it is required to clip out an array of size 1 from the array, with the `split` instruction. At first glance, this limitation seems to be pointless, but it is essential. For example, let us consider the type of an integer array. It can be represented as $\exists\alpha.\langle\alpha\rangle[\beta]$ (The integer constraints and the memory type are omitted). To load a value from the array, we must unpack one of its elements. However, it is difficult to express the type of the array whose all elements have the existential type, except for the one element. The same can be said for storing a value to the array (as mentioned in Sec. 2.4).

The equality of memory types $(\Delta; C \vdash \Sigma = \Sigma')$ is almost the same as the ordinary equality of maps. However, it takes into account the integer constraints between type variables. For example, $\alpha, \beta; \cdot \not\vdash \{\alpha \mapsto \langle\alpha\rangle\} = \{\beta \mapsto \langle\beta\rangle\}$, but $\alpha, \beta; \alpha = \beta \vdash \{\alpha \mapsto \langle\alpha\rangle\} = \{\beta \mapsto \langle\beta\rangle\}$. In addition, arrays whose size is 0 can be ignored in the equality check. For example, $\alpha, \beta; \beta = 0 \vdash \{\alpha \mapsto \langle 0\rangle[\beta]\} = \cdot$.

The typing rule MOVE does almost nothing but checks the rest of the instructions $I$ with the modified registers type that indicates that the register $r_d$ has the same type as the register $r_s$. The typing rule MOVEI type-checks the constant-load instruction. First, it checks the type of the value to be loaded $(\Delta; C \vdash v : \sigma)$. Then, it checks the following instructions $I$ with the modified registers type that indicates that the register $r_d$ has the type $\sigma$.

The typing rule ARITH type-checks the arithmetic instructions. The rule checks whether the operands have the integer types. Then, it type-checks the rest of instructions $I$ with the modified register type that indicates that the register $r_d$ has the result of the arithmetic operations.

The typing rule PUSH type-checks the `push` instruction. First, it checks whether the value of the register $r_d$ is a valid memory address in the memory type $\Sigma$ and there is a stack at the address. Next, it extends the type of the stack by pushing the type of the register $r_s$. It also modifies the address of the stack and the type of the register $r_d$. Then, it type-checks the following instructions $I$.

The typing rule POP type-checks the `pop` instruction. First, it checks whether the value of the register $r_s$ is a valid memory address in the memory type $\Sigma$ and there is a stack at the address. Next, it pops out the top of the stack and overwrites the type of the register $r_d$ with it. It also modifies the address of the stack and the type of the register $r_s$. Then, it type-checks the rest of the instructions $I$.

The typing rule BRANCH is for type-checking the branch instructions. For the taken branch, it first checks whether the value of the register $r_d$ has the label type. Then, it checks whether the condition specified in the label type is satisfied under the current context $(\Delta; C)$ extended with the condition of the taken branch $(\Gamma(r_{s1})(=, \leq)\Gamma(r_{s2}))$. The relation $\Delta; C \vdash \Gamma \leq \Gamma'$ means that the registers type $\Gamma$ indicates a stronger condition than the registers type $\Gamma'$. For example, $\alpha; \cdot \vdash \{\mathtt{r1} \mapsto \alpha\} \leq \{\mathtt{r1} \mapsto \alpha\}$ and $\alpha; \cdot \vdash \{\mathtt{r1} \mapsto \alpha, \mathtt{r2} \mapsto 42\} \leq \{\mathtt{r1} \mapsto \alpha\}$, but $\alpha; \cdot \not\vdash \{\mathtt{r1} \mapsto \alpha\} \leq \{\mathtt{r1} \mapsto \alpha, \mathtt{r2} \mapsto 42\}$. For the non-taken branch, it checks the following instructions $I$ under the extended context with $\Gamma(r_{s1})(\neq, >)\Gamma(r_{s2})$. Moreover, if $C''$ (for the taken branch) or the extended $C$ (for the non-taken branch) contains a contradiction, the corresponding type-check can be omitted without breaking the type soundness, because the contradiction indicates that execution never reaches the branch.

The typing rule JUMP type-checks the jump instruction. The rule checks whether the value of the register $r_d$ has the label type. Then, it checks whether the condition specified in the label type is satisfied under the current context.

Careful readers might notice that nonsense label types can be written in our language. For example, the label type $\forall\alpha, \beta.|\alpha = \beta|[\{\alpha \mapsto \langle 0\rangle\} \otimes \{\beta \mapsto \langle 1\rangle\}].(\Gamma)$ is nonsense because the memory type indicates that the tuple at the address $\alpha(= \beta)$ has the integer value 0 and 1. Even if a block of instructions passes the type check of our language according to the nonsense label type, it may raise a runtime error if it is executed. However, the nonsense label type does not break the type soundness of our language because the block is never executed. For example, let us suppose that there exists a well-formed machine state $(P, M, R, I)$, where the last instruction of $I$ is the jump instruction and its target register $(r_d)$ has a nonsense label type. From the JUMP typing rule, we know that the typing context $(\Delta; \Gamma; C; \Sigma)$ is also nonsense when type-checking the jump instruction. If $I$ does not contain the branch instructions, it contradicts the well-formedness of the machine state because the initial typing context $(\cdot; \Gamma; \cdot; \Sigma)$ is valid (not nonsense) and the only typing rule that may generate a nonsense context from a valid context is the BRANCH typing rule, more specifically, the non-taken branch of the rule. Therefore, there must exists at least one branch instruction which introduces a new integer constraint which conflicts with the typing context of the BRANCH rule. This means that no matter how we instantiate the type variables, the new constraint is never satisfied. That is, the branch is never taken at runtime. Thus, execution never reaches the jump instruction.

The typing rule APPLY type-checks the type application instruction. The rule type-checks the rest of instructions $I$ with the modified registers type that indicates that the register $r$ has the instantiated type $\sigma'_f$.

The typing rule ROLL and UNROLL check whether the instructions for recursive types (`roll` and `unroll`) are well-formed. The rule ROLL checks whether the type of the tuple at the address $i$ can be rolled to the specified recursive type. Then, it checks the following instructions $I$ with the new memory type modified so that the type is rolled. The rule UNROLL is vice versa.

The typing rule PACK and UNPACK type-check the instructions for packing and unpacking the existential types (`pack` and `unpack`). The rule PACK first checks whether the tuple at the address $i$ can be packed into the specified existential type. Next, it modifies the memory type so that a tuple is packed into an existential type, and removes the portion of the memory that is hidden into the existential type. Then, it checks the rest of the instructions $I$. The rule UNPACK is vice versa. The rules ROLL, UNROLL, PACK and UNPACK only allow arrays whose size is 1, as with LOAD and STORE.

The typing rule SPLIT and CONCAT check the well-formedness of the instructions for splitting/concatenating arrays (`split` and `concat`). The rule SPLIT checks whether the size $(j_1)$ of the array to be split is greater than or equal to the required size $(\Delta; C \models i_2 \leq j_1)$. Then, it splits the array into two arrays and extends the memory type with them. The rule CONCAT checks whether the given two arrays are adjacent $(\Delta; C \models j_1 = i_1 + \mathit{sizeof}(\tau) * i_2)$. Then, it concatenates the two arrays into one and extends the memory

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma(r_s) \mapsto \langle \ldots, \sigma_n, \ldots \rangle\} \quad \Delta; \Gamma\{r_d \mapsto \sigma_n\}; C; \Sigma \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{ld}\ [r_s + n], r_d; I} \text{ (LOAD)}$$

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma(r_d) \mapsto \langle \ldots, \sigma_n, \ldots \rangle\} \quad \Delta; \Gamma; C; \Sigma' \otimes \{\Gamma(r_d) \mapsto \langle \ldots, \Gamma(r_s), \ldots \rangle\} \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{st}\ r_s, [r_d + n]; I} \text{ (STORE)}$$

$$\frac{\Delta; \Gamma\{r_d \mapsto \Gamma(r_s)\}; C; \Sigma \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{mov}\ r_s, r_d; I} \text{ (MOVE)}$$

$$\frac{\Delta; C \vdash v : \sigma \qquad \Delta; \Gamma\{r_d \mapsto \sigma\}; C; \Sigma \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{movi}\ v, r_d; I} \text{ (MOVEI)}$$

$$\frac{\Delta; \Gamma\{r_d \mapsto \Gamma(r_{s2})\,(+,-,*)\Gamma(r_{s1})\}; C; \Sigma \vdash I}{\Delta; \Gamma; C; \Sigma \vdash (\texttt{add}, \texttt{sub}, \texttt{mul})\ r_{s1}, r_{s2}, r_d; I} \text{ (ARITH)}$$

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma(r_d) \mapsto st\} \quad \Delta; \Gamma\{r_d \mapsto \Gamma(r_d) - 1\}; C; \Sigma' \otimes \{\Gamma(r_d) - 1 \mapsto \Gamma(r_s) :: st\} \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{push}\ r_s, [r_d]; I} \text{ (PUSH)}$$

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma(r_s) \mapsto \sigma :: st\} \quad \Delta; \Gamma\{r_s \mapsto \Gamma(r_s) + 1\}\{r_d \mapsto \sigma\}; C; \Sigma' \otimes \{\Gamma(r_s) + 1 \mapsto st\} \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{pop}\ [r_s], r_d; I} \text{ (POP)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Gamma(r_d) = \forall.\, |C'|\,[\Sigma'](\Gamma') \\ C'' \equiv C \wedge \Gamma(r_{s1})\,(=,\leq)\Gamma(r_{s2}) \\ \Delta; C'' \models C' \qquad \Delta; C'' \vdash \Sigma = \Sigma' \qquad \Delta; C'' \vdash \Gamma \leq \Gamma' \\ \Delta; \Gamma; C \wedge \Gamma(r_{s1})\,(\neq,>)\Gamma(r_{s2}); \Sigma \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash (\texttt{beq}, \texttt{ble})\ r_{s1}, r_{s2}, r_d; I} \text{ (BRANCH)}$$

$$\frac{\Delta; C \vdash \Gamma(r_d) = \forall.\, |C'|\,[\Sigma'](\Gamma') \quad \Delta; C \models C' \qquad \Delta; C \vdash \Sigma = \Sigma' \qquad \Delta; C \vdash \Gamma \leq \Gamma'}{\Delta; \Gamma; C; \Sigma \vdash \texttt{jmp}\ r_d} \text{ (JUMP)}$$

**Figure 12: Typing rules (instructions)**

type with it. Here $sizeof(\tau)$ is the size of the tuple represented by the type $\tau$. If $\tau$ is a recursive type $(\mu \ldots \tau')$ or an existential type $(\exists \ldots \tau')$, $sizeof(\tau)$ is recursively applied to the inner tuple type $\tau'$. Because $sizeof(\tau)$ is always a constant integer, the rule SPLIT and CONCAT never generate non-linear constraints. Please note that the `split` instruction can create the array of size 0 (if the second argument of the instruction is 0 or equals to the size of the array). This is because, without the array of size 0, special handling is required to access the first and the last element of the variable-length arrays. The arrays of size 0 do not affect the type soundness because they are never accessed and the equality check of the memory types absorbs them.

The typing rule TUPLE_SPLIT and TUPLE_CONCAT are almost the same as SPLIT and CONCAT, but they typecheck the split and concatenation of tuples.

# 4. MEMORY MANAGEMENT WITH TALK

In this section, we show simple memory management code which is written in TALK. Although its algorithm is simple and naive, we believe that it is sufficient to show the flexibility and expressiveness of TALK.

## 4.1 Representation of the free memory

Fig. 14 represents the type of the free memory. It is a list of variable-length arrays. Each element of the list is a variable-length array and a tuple which has two elements. The size of the array is stored in the second element of the tuple. The first element of the tuple is a pointer to the next element of the list. The one argument for the recursive type represents the address of the tuple itself. Therefore, the integer constraint inside the existential type (line 2) indicates that the tuple and the variable-length array are adjacent. In addition, the memory type inside the existential type (line 3) indicates that there surely exists a memory region which satisfies the memory type. Strictly speaking, the definition of the list in Fig. 14 represents an infinite list because the definition does not include any list terminator. Therefore, it might be unrealistic because the free memory is finite. To define finite lists, the type system of TALK supports variant types (or union types), but we do not explain them in this paper for clarity.

## 4.2 Implementation of malloc

Fig. 15 is a simple implementation of malloc. For clarity, the syntax of instructions are slightly extended. In addition, the `apply` instruction and the argument for the `pack`, `unpack` and `roll` instructions are omitted for clarity.

The label type of `malloc` indicates that the function takes a free memory ($FreeMem(\alpha_{free})$ at line 2) as an argument and returns an array of the specified size ($\alpha_{size}$) at line 3). The type of the allocated array is specified at line 61. Please note that the return type of the function is abbreviated as $ret_t$.

The function first checks whether the array of first element of the given free memory list satisfies the requested size (line 11). If so, the function jumps to `malloc_success`. Otherwise, it tries the next element in the free memory list. First, it stores the current element of the list and the return address on the stack (line 13 and 14). Then, it calls itself recursively (from line 16 to 21). After the return from the recursive call (the instructions of the label `malloc_cont`), it concatenates the saved element with the returned free memory

$$\frac{\begin{array}{c}\Delta; C \vdash \Gamma(r) : \sigma_f \qquad \sigma_f \equiv \forall \Delta'.\, |C'|\,[\Sigma']\,(\Gamma') \\ \theta \equiv [c_1, \ldots, c_n/\Delta''] \quad C'' \equiv C'\theta \quad \Sigma'' \equiv \Sigma'\theta \quad \Gamma'' \equiv \Gamma'\theta \\ \sigma'_f \equiv \forall \Delta' \backslash \Delta''.\, |C''|\,[\Sigma'']\,(\Gamma'') \quad \Delta; \Gamma\{r \mapsto \sigma'_f\}; C; \Sigma \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{apply}\ r\,[c_1, \ldots, c_n/\Delta'']\,; I} \text{(APPLY)}$$

$$\frac{\begin{array}{c}\tau \equiv \mu\eta\,[\Delta'].\tau'\,(c_1, \ldots, c_n) \\ \Delta; C \vdash \Sigma = \Sigma' \otimes \{i \mapsto \tau'\,[\mu\eta\,[\Delta'].\tau'/\eta]\,[c_1, \ldots, c_n/\Delta']\} \\ \Delta; \Gamma; C; \Sigma' \otimes \{i \mapsto \tau\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{roll}_\tau\ i; I} \text{(ROLL)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{i \mapsto \mu\eta\,[\Delta'].\tau'\,(c_1, \ldots, c_n)\} \\ \Delta; \Gamma; C; \Sigma' \otimes \{i \mapsto \tau'\,[\mu\eta\,[\Delta'].\tau'/\eta]\,[c_1, \ldots, c_n/\Delta']\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{unroll}\ i; I} \text{(UNROLL)}$$

$$\frac{\begin{array}{c}\theta \equiv [c_1, \ldots, c_n/\Delta'] \quad \Delta; C \vdash \Sigma = \Sigma'' \otimes \{i \mapsto \tau\theta\} \otimes \Sigma'\theta \\ \Delta; C \models C'\theta \qquad \Delta; \Gamma; C; \Sigma'' \otimes \{i \mapsto \exists \Delta'.\, |C'|\,[\Sigma']\,\tau\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{pack}_{[c_1,\ldots,c_n\,|\Sigma'[c_1,\ldots,c_n/\Delta']]as\exists\Delta'.|C'|[\Sigma']\tau}\ i; I} \text{(PACK)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma'' \otimes \{i \mapsto \exists \Delta'.\, |C'|\,[\Sigma']\,\tau\} \quad \theta \equiv [\Delta''/\Delta'] \\ \Delta\Delta''; \Gamma; C \wedge C'\theta; \Sigma'' \otimes \{i \mapsto \tau\theta\} \otimes \Sigma'\theta \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{unpack}\ i\ \texttt{with}\ \Delta''; I} \text{(UNPACK)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \tau\,[j_1]\} \qquad \Delta; C \models 0 \leq i_2 \leq j_1 \\ k_1 \equiv i_1 + \mathit{sizeof}\,(\tau) * i_2 \qquad k_2 \equiv j_1 - i_2 \\ \Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \tau\,[i_2]\} \otimes \{k_1 \mapsto \tau\,[k_2]\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{split}\ i_1, i_2; I} \text{(SPLIT)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \tau\,[i_2]\} \otimes \{j_1 \mapsto \tau\,[j_2]\} \\ \Delta; C \models j_1 = i_1 + \mathit{sizeof}\,(\tau) * i_2 \\ \Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \tau\,[i_2 + j_2]\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{concat}\ i_1, j_1, j_2; I} \text{(CONCAT)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\} \\ \Delta; C \models 0 < n_2 < n \\ \Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_{n_2} \rangle\} \\ \qquad \otimes \{i_1 + n_2 \mapsto \langle \sigma_{n_2+1}, \ldots, \sigma_n \rangle\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{tuple_split}\ i_1, n_2; I} \text{(TUPLE_SPLIT)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\} \otimes \{i_2 \mapsto \langle \sigma'_1, \ldots, \sigma'_m \rangle\} \\ \Delta; C \models i_2 = i_1 + n \\ \Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m \rangle\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{tuple_concat}\ i_1, i_2; I} \text{(TUPLE_CONCAT)}$$

**Figure 13: Typing rules (coerce instructions)**

$$\begin{array}{l} \mathit{FreeMem} \equiv \\ \quad \mu\eta\,[\alpha_{self}].\exists\,\alpha_{next}, \alpha_{size}, \alpha_{mem}.\ |\alpha_{mem} = \alpha_{self} + 2| \\ \quad\quad [\{\alpha_{mem} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size}]\} \otimes \{\alpha_{next} \mapsto \eta\,(\alpha_{next})\}] \\ \quad\quad \langle \alpha_{next}, \alpha_{size} \rangle \end{array}$$

**Figure 14: Type of the free memory (list of variable-length arrays)**

```
∀ α_size, α_free, α_stk, γ, ϵ. | · |
[{α_free ↦ FreeMem(α_free)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
   (r1 : α_size, r2 : α_free, r3 : ret_t, r4 : α_stk)
malloc:
      unroll α_free
      unpack α_free
      ld [r2 + 1], r5
      # try the first element
      movi malloc_success, r10
      apply r10
      ble r1, r5, r10
      # not enough, try next
      push r2, [r4] # save local vars.
      push r3, [r4]
      # recursive call
      ld [r2], r2
      movi malloc_cont, r3
      apply r3
      movi malloc, r10
      apply r10
      jmp r10

∀ α_size, α_tag, α_stk, α_junk, α_mem, α'_size, α, α_free, γ, ϵ.
|α_mem = α_tag + 2|
[{α_tag ↦ ⟨α_junk, α'_size⟩} ⊗ {α_mem ↦ ∃ β. ⟨β⟩ [α'_size]}⊗
  {α ↦ ∃ β. ⟨β⟩ [α_size]} ⊗ {α_free ↦ FreeMem(α_free)}
  {α_stk − 2 ↦ ret_t :: α_tag :: γ} ⊗ ϵ]
   (r1 : α, r2 : α_free, r4 : α_stk − 2)
malloc_cont:
      pop [r4], r3 # restore local vars.
      pop [r4], r5
      # link the previous element
      # to the returned free memory list
      st r2, [r5]
      mov r5, r2
      pack α_tag
      roll α_tag
      apply r3 [α, α_tag/α, α_free]
      jmp r3

∀ α_size, α_tag, α_stk, α_free, α_mem, α'_size, γ, ϵ.
|α_size ≤ α'_size ∧ α_mem = α_tag + 2|
[{α_tag ↦ ⟨α_free, α'_size⟩} ⊗ {α_mem ↦ ∃ β. ⟨β⟩ [α'_size]}⊗
  {α_free ↦ FreeMem(α_free)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
   (r1 : α_size, r2 : α_tag, r3 : ret_t, r4 : α_stk)
malloc_success:
      # allocate memory by split
      # from the end of the array
      split α_mem, (α'_size − α_size)
      # rewrite the tag
      ld [r2 + 1], r5
      sub r1, r5, r6
      st r6, [r2 + 1]
      # set the address of
      # the allocated memory to r1
      mov r2, r5
      add 2, r5, r5
      add r6, r5, r1
      pack α_tag
      roll α_tag
      apply r3[α_mem + α'_size − α_size, α_tag/α, α_free]
      jmp r3

ret_t ≡ ∀ α, α_free. | · |[{α ↦ ∃β. ⟨β⟩ [α_size]}⊗
        {α_free ↦ FreeMem(α_free)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
          (r1 : α, r2 : α_free, r4 : α_stk)
```

**Figure 15: Simple malloc implementation in TALK**

list (line 33) and returns it as a new free memory list (from line 34 to 38) through the register `r2`. Of course, the array allocated by the recursive call is also returned through the register `r1`. Here the stack type $\{\alpha_{stk} - 2 \mapsto ret_t :: \alpha_{tag} :: \gamma\}$ in the memory type (line 26) represents a stack whose top element has the type $ret_t$ and the next element has type $\alpha_{tag}$ and the rest is unknown ($\gamma$).

The code of `malloc_success` first splits the array of the first element of the given free memory list into the array of the requested size and the rest (line 47). The `split` instruction passes the type check of TALK because the type checker knows that the length of the array is greater (or equal) than the requested size from the label type of `malloc_success` (line 40). Then, it rewrites the information about the unused array and its size in the second element (from line 43-45) and returns the allocated array (from line 54 to 60).

## 4.3 Implementation of free

Fig. 16 is a simple implementation of free. First, the code converts the first two elements of the array to be freed into a tuple (from line 9 to 13). Then, the code concatenates the tuple to the given free memory list along with the rest of the array (from line 15 to 19). The label type of `free` indicates that the freed array cannot be used any more because the array is deleted from the memory type after the function return (line 5).

```
1  ∀ α_mem, α_free, α_size, ϵ. |α_size > 2|
2  [{α_mem ↦ ∃ β. ⟨β⟩ [α_size]}⊗
3    {α_free ↦ FreeMem(α_free)} ⊗ ϵ]
4     (r1 : α_mem, r2 : α_free, r4 : α_size,
5      r3 : ∀ α.| · |[{α ↦ FreeMem(α)} ⊗ ϵ](r1 : α))
6  free:
7        # create a tag from
8        # the memory to be freed
9        split α_mem, 2
10       split α_mem, 1
11       unpack α_mem
12       unpack α_mem + 1
13       tuple_concat α_mem, α_mem + 1
14       # initialize the tag
15       sub 2, r4, r4
16       st r4, [r1 + 1]
17       # link the tag to
18       # the free memory list
19       st r2, [r1]
20       pack α_mem
21       roll α_mem
22       apply r3[α_mem/α]
23       jmp r3
```

**Figure 16: Simple free implementation in TALK**

# 5. IMPLEMENTATION

We implemented a TALK assembler and a TALK type checker for the IA-32 [10] architecture. The TALK assembler takes TALK code and emits binary executables annotated with the TALK type information. The format of the binary executables are usual ELF format. Therefore, they can be executed without any special runtime support. The TALK type checker takes the binary executables and type-checks them. Because the type system of TALK includes integer constraints, the type checker must be able to solve the constraints. To this end, we utilized the algorithm of the Omega test [16].

Using the TALK assembler, we implemented a prototype OS kernel in TALK[1]. The kernel provides a memory management facility, a multi-thread management facility and a very basic device control facility. For booting the kernel, the GNU GRUB boot loader [8] is used. In addition, some peculiar boot procedures (e.g., segment preparation) of IA-32 are not typed. Except for them, the kernel is completely written in TALK. The size of the kernel is about 1700 lines of TALK code. It takes about 0.9 seconds to type-check the whole kernel on the Pentium 4 (3GHz) machine.

The TALK assembler, the TALK type checker and the prototype OS kernel are available from our web site [23].

# 6. RELATED WORK

Linear type systems [24] ensure that a memory region is accessed only once. That is, they can prevent pointers from aliasing. Therefore, the memory region can be reused safely. There exist TALs based on the linear types [2, 1]. One big problem of the linear types is that the expressiveness of linearly-typed languages is largely limited because no aliases are allowed.

Alias type systems [20, 25] do not prevent pointers from aliasing, but track the information about aliases for reusing memory regions safely. Thus, the alias type systems are more expressive than the linear type systems. However, it is impossible to implement practical memory management in the original alias type system because it does not support variable-length arrays. As described in Sec. 3, our TALK is based on the alias types and extended to support variable-length arrays and integer constraints. Thus, we are able to implement practical memory management in TALK.

Hawblitzel et al. [9] extends the alias type system for implementing flexible memory management. The similarity between our approach and theirs is that both introduce integer constraints to the alias types. The important difference is that, in their type system, variable-length arrays are realized as a combination of fixed-length tuples and recursive types. However, there are two problems in their approach. One problem is that elements of an array cannot be accessed in $O(1)$ order because the array type must be unrolled ($O(n)$ time at worst) in advance. The other problem is that it requires runtime type checks for managing arrays. To solve these problems, they extended their type system intricately for detecting useless runtime type-checks as precisely as possible. For example, they defined 'split' of arrays as a function, and showed that the function is not needed at runtime, with their complex typing rules. On the other hand, there are no such problems in our type system because it directly supports the variable-length arrays as language primitives. Thus, our type system is simpler than theirs and yet powerful enough to implement memory management code.

DTAL [26] is a typed assembly language extended with the dependent type. As our type system, DTAL also introduced integer constraints to its type system. However, DTAL is not flexible enough to implement memory management because memory reusing is impossible. The goal of DTAL is to type-

[1]'K' of TALK stands for 'Kernel'

check array bound-checking.

In region-based memory management [21, 22, 7], heap values are allocated in one of memory regions. When a memory region is deallocated, all the heap values in the region are deallocated. The region-based memory management does not allow programmers to directly manage memory. Calculus of Capability [3] extends the region-based memory management and allows programmers to explicitly allocate and deallocate memory regions, but memory regions cannot be reused explicitly and the heap values allocated in memory regions cannot be managed directly.

Shape analysis [5, 6, 19] is an analysis which estimates the shape (e.g., tree, DAG or cyclic graph) of the data structure that is accessible from pointers. Although the shape analysis is developed in the research area of compiler optimization, it can be used for detecting pointer aliases because it determines whether two pointers point to the same data structure. However, the approach of the shape analysis cannot be applied directly to memory management because it is a conservative analysis. In addition, the analysis can tell whether a data structure can be deallocated safely, but programmers cannot reuse the data structure explicitly.

As for verifying the correctness of existing memory management programs, Marti et al. [12] proved the correctness of the heap manager of the Topsy operating system [18] using separation logic [17].

# 7. CONCLUSION

We designed and implemented a new strictly and statically typed assembly language (TALK) which is powerful enough to implement practical memory management (e.g., malloc/free). The type system of our TALK supports variable-length arrays as language primitives. Therefore, our TALK is able to efficiently handle free memory of systems whose size is not known until runtime. In addition, The type system of our TALK keep track of aliases of pointers explicitly. Therefore, programmers are able to reuse memory regions safely because the type system allows them to change the types of the regions. We implemented the assembler and the type checker of our TALK for IA-32. We also implemented a prototype OS kernel for the IA-32 architecture in TALK. The kernel provides memory management facilities and a multi-thread management facilities.

# 8. REFERENCES


[1] D. Aspinall and A. Compagnoni. Heap bounded assembly language. *Automated Reasoning*, 31:261–302, 2003.

[2] J. Cheney and G. Morrisett. A linearly typed assembly language. Technical report, Department of Computer Science, Cornell University, 2003.

[3] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, 1999.

[4] C#. http://msdn.microsoft.com/net/ecma.

[5] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 230–241, 1994.

[6] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, 1996.

[7] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293, 2002.

[8] GNU GRUB. http://www.gnu.org/software/grub/.

[9] C. Hawblitzel, E. Wei, H. Huang, E. Krupski, and L. Wittie. Low-level linear memory management. In *SPACE 2004*, 2004.

[10] IA-32 Intel Architecture. http://developer.intel.com.

[11] Java. http://java.sun.com.

[12] N. Marti, R. Affedlt, and A. Yonezawa. Verification of the heap manager of an operating system using separation logic. In *SPACE 2006*, Jan. 2006.

[13] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, 1999.

[14] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.

[15] Objective Caml. http://caml.inria.fr.

[16] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[17] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[18] L. Ruf, C. Jeker, B. Lutz, and B. Plattner. Topsy v3: A nodeos for network processors. In *ANTA 2003*, 2003.

[19] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

[20] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 366–381. Springer-Verlag, 2000.

[21] M. Tofte and J. P. Talpin. Implementation of the typed call-by-value $\lambda$-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 188–201, 1994.

[22] M. Tofte and J. P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[23] TOS project. http://web.yl.is.s.u-tokyo.ac.jp/~tosh/tos/.

[24] D. Turner, P. Wadler, and C. Mossion. Once upon a type. In *ACM International Conference on Functional Programming and Computer Architecture*, 1995.

[25] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Types in Compilation*, 2000.

[26] H. Xi and R. Harper. A dependently typed assembly language. In *ICFP*, 2001.

[27] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, January 1999.

# APPENDIX

## A. TYPING DERIVATION OF MALLOC

The typing derivation of the code in Fig. 15 is as follows. First, the type of the initial memory is

$$\{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

Next, after the `unroll` and `unpack` instructions (line 5 and 6), the memory type becomes

$$\{\alpha_{free} \mapsto \langle \alpha_{next}, \alpha'_{size}\rangle\} \otimes \{\alpha_{mem} \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha'_{size}]\} \otimes \{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon,$$

where $\alpha_{mem} = \alpha_{free} + 2$. Here the argument for the `unpack` instruction is `with` $\alpha_{next}, \alpha'_{size}, \alpha_{mem}$. Then, after the `ld` instruction at line 7, the type of the register `r5` is $\alpha'_{size}$. Thus, the `ble` instruction at line 11 checks whether $\alpha_{size} \leq \alpha'_{size}$ or not. Then, the argument for the `apply` instruction at line 10 is

$$[\alpha_{size}, \alpha_{free}, \alpha_{stk}, \alpha_{next}, \alpha_{mem}, \alpha'_{size}, \gamma, \epsilon / \alpha_{size}, \alpha_{tag}, \alpha_{stk}, \alpha_{free}, \alpha_{mem}, \alpha'_{size}, \gamma, \epsilon].$$

Accordingly, the type of the register `r10` becomes

$$\forall.|\alpha_{size} \leq \alpha'_{size} \wedge \alpha_{mem} = \alpha_{free} + 2| [\{\alpha_{free} \mapsto \langle \alpha_{next}, \alpha'_{size}\rangle\} \otimes \{\alpha_{mem} \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha'_{size}]\} \otimes \{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon] (\mathtt{r1} : \alpha_{size}, \mathtt{r2} : \alpha_{free}, \mathtt{r3} : \mathit{ret_t}, \mathtt{r4} : \alpha_{stk}).$$

Here the memory type, the registers type and the integer constraints satisfy the precondition specified by the above label type, because the `ble` instruction adds a new integer constraint ($\alpha_{size} \leq \alpha'_{size}$). Thus, the `ble` instruction at line 11 is type checked successfully.

Next, after the two `push` instructions (line 12 and 13), the memory type becomes

$$\{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\} \otimes \{\alpha_{stk} - 2 \mapsto \mathit{ret_t} :: \alpha_{free} :: \gamma\} \otimes \epsilon',$$

where

$$\epsilon' \equiv \{\alpha_{free} \mapsto \langle \alpha_{next}, \alpha'_{size}\rangle\} \otimes \{\alpha_{mem} \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha'_{size}]\} \otimes \epsilon.$$

In addition, the type of the register `r4` becomes $\alpha_{stk} - 2$. Then, after the `ld` instruction at line 16, the type of the register `r2` becomes $\alpha_{next}$. Here the argument for the `apply` instruction at line 18 is

$$[\alpha_{size}, \alpha_{free}, \alpha_{stk}, \alpha_{next}, \alpha_{mem}, \alpha'_{size}, \gamma, \epsilon / \alpha_{size}, \alpha_{tag}, \alpha_{stk}, \alpha_{junk}, \alpha_{mem}, \alpha'_{size}, \gamma, \epsilon].$$

Then, the type of the register `r3` becomes

$$\forall\alpha, \alpha'_{free}.|\alpha_{mem} = \alpha_{free} + 2| [\{\alpha \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha_{size}]\} \otimes \{\alpha'_{free} \mapsto \mathit{FreeMem}(\alpha'_{free})\} \otimes \{\alpha_{stk} - 2 \mapsto \mathit{ret_t} :: \alpha_{free} :: \gamma\} \otimes \epsilon'] (\mathtt{r1} : \alpha, \mathtt{r2} : \alpha'_{free}, \mathtt{r4} : \alpha_{stk} - 2).$$

(Here the bound integer variable $\alpha_{free}$ is renamed to $\alpha'_{free}$.) In addition, the argument for the `apply` instruction at line 20 is

$$[\alpha_{size}, \alpha_{next}, (\alpha_{stk} - 2), (\mathit{ret_t} :: \alpha_{free} :: \gamma), \epsilon' / \alpha_{size}, \alpha_{free}, \alpha_{stk}, \gamma, \epsilon].$$

Then, the type of the register `r10` becomes

$$\forall.|\cdot| [\{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\} \otimes \{\alpha_{stk} - 2 \mapsto \mathit{ret_t} :: \alpha_{free} :: \gamma\} \otimes \epsilon'] (\mathtt{r1} : \alpha_{size}, \mathtt{r2} : \alpha_{next}, \mathtt{r3} : \mathit{ret_t'}, \mathtt{r4} : \alpha_{stk} - 2),$$

where

$$\mathit{ret_t'} \equiv \forall\alpha, \alpha'_{free}.|\cdot| [\{\alpha \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha_{size}]\} \otimes \{\alpha'_{free} \mapsto \mathit{FreeMem}(\alpha'_{free})\} \otimes \{\alpha_{stk} - 2 \mapsto \mathit{ret_t} :: \alpha_{free} :: \gamma\} \otimes \epsilon'] (\mathtt{r1} : \alpha, \mathtt{r2} : \alpha'_{free}, \mathtt{r4} : \alpha_{stk} - 2).$$

Now, the `jmp` instruction at line 21 is type checked because the current memory and registers type satisfies the precondition specified in the label type of the register `r10`. Please note that the integer constraint ($\alpha_{mem} = \alpha_{free} + 2$) specified in the label type of the register `r3` is satisfied by the current integer constraints. Thus, the type system ignores the constraint when checking the equality of the label types of the register `r3` and *ret_t'*.

Next, the typing derivation of the instructions of the label `malloc_cont` is as follows. First, the type of the initial memory is

$$\{\alpha_{tag} \mapsto \langle \alpha_{junk}, \alpha'_{size}\rangle\} \otimes \{\alpha_{mem} \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha'_{size}]\} \otimes \{\alpha \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha_{size}]\} \otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \{\alpha_{stk} - 2 \mapsto \mathit{ret_t} :: \alpha_{tag} :: \gamma\} \otimes \epsilon,$$

where $\alpha_{mem} = \alpha_{tag} + 2$. Next, after the two *pop* instructions (line 29 and 30), the memory type becomes

$$\{\alpha_{tag} \mapsto \langle \alpha_{junk}, \alpha'_{size}\rangle\} \otimes \{\alpha_{mem} \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha'_{size}]\} \otimes \{\alpha \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha_{size}]\} \otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

In addition, the type of the register `r4` becomes $\alpha_{stk}$, the type of the register `r3` becomes *ret_t*, and the type of the register `r5` becomes $\alpha_{tag}$. Then, after the `st` instruction at line 33, the memory type becomes

$$\{\alpha_{tag} \mapsto \langle \alpha_{free}, \alpha'_{size}\rangle\} \otimes \{\alpha_{mem} \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha'_{size}]\} \otimes \{\alpha \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha_{size}]\} \otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

In addition, after the `mov` instruction at line 34, the type of the register `r2` becomes $\alpha_{tag}$. Next, after the `pack` instruction at line 35, the memory type becomes

$$\{\alpha_{tag} \mapsto \tau\} \otimes \{\alpha \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha_{size}]\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon,$$

where

$$\tau \equiv \exists\alpha_{next}, \alpha_{size}, \alpha_{mem}.|\alpha_{mem} = \alpha_{tag} + 2| [\{\alpha_{mem} \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha_{size}]\} \otimes \{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\}] \langle \alpha_{next}, \alpha_{size}\rangle.$$

Here the argument for the `pack` instruction is

$$[\alpha_{free}, \alpha'_{size}, \alpha_{mem}| \{\alpha_{mem} \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha'_{size}]\} \otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}] \textit{ as } \tau.$$

Next, after the `roll` instruction at line 36, the memory type becomes

$$\{\alpha_{tag} \mapsto \mathit{FreeMem}(\alpha_{tag})\} \otimes \{\alpha \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha_{size}]\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

Then, after the `apply` instruction at line 37, the type of the register `r3` becomes

$$\forall.|\cdot|[\{\alpha \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size}]\} \otimes \{\alpha_{tag} \mapsto \mathit{FreeMem}(\alpha_{tag})\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon] \; (\mathtt{r1} : \alpha, \mathtt{r2} : \alpha_{tag}, \mathtt{r4} : \alpha_{stk}).$$

Here the current memory and registers type satisfies the precondition specified by the above label type. Thus, the `jmp` instruction at line 38 is type checked successfully.

Last, the typing derivation of the instructions of the label `malloc_success` is as follows. First, the type of the initial memory is

$$\{\alpha_{tag} \mapsto \langle\alpha_{free}, \alpha'_{size}\rangle\} \otimes \{\alpha_{mem} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha'_{size}]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon,$$

where $\alpha_{size} \leq \alpha'_{size} \wedge \alpha_{mem} = \alpha_{tag} + 2$. Then, after the `split` instruction at line 47, the memory type becomes

$$\{\alpha_{tag} \mapsto \langle\alpha_{free}, \alpha'_{size}\rangle\} \otimes \{\alpha_{mem} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha'_{size} - \alpha_{size}]\}\otimes \{\alpha_{mem} + \alpha'_{size} - \alpha_{size} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size}]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

Here the `split` instruction passes the type check because the type checker knows that $\alpha_{size} \leq \alpha'_{size}$. Then, after the `ld` and `sub` instruction at line 49 and 50, the type of the register `r6` becomes $\alpha'_{size} - \alpha_{size}$. Next, after the `st` instruction at line 51, the memory type becomes

$$\{\alpha_{tag} \mapsto \langle\alpha_{free}, \alpha'_{size} - \alpha_{size}\rangle\}\otimes \{\alpha_{mem} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha'_{size} - \alpha_{size}]\}\otimes \{\alpha_{mem} + \alpha'_{size} - \alpha_{size} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size}]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

Then, after the instructions from line 54 to 56, the type of the register `r1` becomes $\alpha_{tag} + 2 + \alpha'_{size} - \alpha_{size}$. Next, after the `pack` instruction at line 57, the memory type becomes

$$\{\alpha_{tag} \mapsto \tau\} \otimes \{\alpha_{mem} + \alpha'_{size} - \alpha_{size} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size}]\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon,$$

where

$$\tau \equiv \exists\alpha_{next}, \alpha_{size}, \alpha_{mem}.|\alpha_{mem} = \alpha_{tag} + 2| \; [\{\alpha_{mem} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size}]\} \otimes \{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\}] \; \langle\alpha_{next}, \alpha_{size}\rangle\,.$$

Here the argument for the `pack` instruction is

$$[\alpha_{free}, \alpha'_{size} - \alpha_{size}, \alpha_{mem}| \; \{\alpha_{mem} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha'_{size} - \alpha_{size}]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}] \; \mathit{as}\ \tau.$$

Next, after the `roll` instruction, the memory type becomes

$$\{\alpha_{tag} \mapsto \mathit{FreeMem}(\alpha_{tag})\}\otimes \{\alpha_{mem} + \alpha'_{size} - \alpha_{size} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size}]\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

Then, after the `apply` instruction at line 59, the type of the register `r3` becomes

$$\forall.|\cdot|[\{\alpha_{mem} + \alpha'_{size} - \alpha_{size} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size}]\}\otimes \{\alpha_{tag} \mapsto \mathit{FreeMem}(\alpha_{tag})\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon] \; (\mathtt{r1} : \alpha_{mem} + \alpha'_{size} - \alpha_{size}, \mathtt{r2} : \alpha_{tag}, \mathtt{r4} : \alpha_{stk}).$$

Now, the current memory and registers type satisfies the precondition specified in the above label type. Thus, the `jmp` instruction at line 60 passes the type check. As for the type of the register `r1`, please note that the type checker knows that $\alpha_{mem} = \alpha_{tag} + 2$.

# B. TYPING DERIVATION OF FREE

The typing derivation of the code in Fig. 16 is as follows. First, the type of the initial memory is

$$\{\alpha_{mem} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size}]\} \otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \epsilon,$$

where $\alpha_{size} > 2$. Next, after the two `split` instructions (line 9 and 10), the memory type becomes

$$\{\alpha_{mem} \mapsto \exists\beta.\langle\beta\rangle\} \otimes \{\alpha_{mem} + 1 \mapsto \exists\beta.\langle\beta\rangle\}\otimes \{\alpha_{mem} + 2 \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size} - 2]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \epsilon.$$

Then, after the two `unpack` instructions (line 11 and 12), the memory type becomes

$$\{\alpha_{mem} \mapsto \langle\beta_1\rangle\} \otimes \{\alpha_{mem} + 1 \mapsto \langle\beta_2\rangle\}\otimes \{\alpha_{mem} + 2 \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size} - 2]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \epsilon,$$

Then, after the `tuple_concat` instruction (line 13), the memory type becomes

$$\{\alpha_{mem} \mapsto \langle\beta_1, \beta_2\rangle\} \otimes \{\alpha_{mem} + 2 \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size} - 2]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \epsilon.$$

Thus, a tuple of size 2 was created at the top of the memory (array) to be freed.

Next, we initialize the created tuple. First, we store the size of the array to be freed in the second element of the tuple (line 15 and 16). Thus, the memory type becomes

$$\{\alpha_{mem} \mapsto \langle\beta_1, \alpha_{size} - 2\rangle\} \otimes \{\alpha_{mem} + 2 \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size} - 2]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \epsilon.$$

Then, we link the tuple to the free memory list (line 19). Now, the memory type becomes

$$\{\alpha_{mem} \mapsto \langle\alpha_{free}, \alpha_{size} - 2\rangle\} \otimes \{\alpha_{mem} + 2 \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size} - 2]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \epsilon.$$

Then, after the `pack` instruction (line 20), the memory types becomes $\{\alpha_{mem} \mapsto \tau\} \otimes \epsilon$, where

$$\tau \equiv \exists\alpha_{next}, \alpha_{size}, \alpha'_{mem}.|\alpha'_{mem} = \alpha_{mem} + 2| \; [\{\alpha'_{mem} \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size}]\} \otimes \{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\}] \; \langle\alpha_{next}, \alpha_{size}\rangle\,.$$

Here the hidden argument of the `pack` instruction is

$$[\alpha_{free}, (\alpha_{size} - 2), (\alpha_{mem} + 2)| \; \{\alpha_{mem} + 2 \mapsto \exists\beta.\langle\beta\rangle\,[\alpha_{size} - 2]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}] \; \mathit{as}\ \tau.$$

Next, after the `roll` instruction (line 21), the memory type becomes $\{\alpha_{mem} \mapsto \mathit{FreeMem}(\alpha_{mem})\} \otimes \epsilon$. The omitted argument of the `roll` instruction is $\mathit{FreeMem}(\alpha_{mem})$. Then, after the `apply` instruction at line 22, the type of the register `r3` becomes

$$\forall.|\cdot|[\{\alpha_{mem} \mapsto \mathit{FreeMem}(\alpha_{mem})\} \otimes \epsilon](\mathtt{r1} : \alpha_{mem}).$$

Last, the `jmp` instruction passes the type check because the precondition specified in the above label type is satisfied by the current memory and registers type.

# Static Types for Scripting

Torben Mogensen

DIKU

**Abstract**

Most scripting languages are dynamically typed, both to save the programmer from writing verbose typing declaratings and to allow heterogeneous collections and other constructions that are troublesome for typical static type systems. But by being dynamically typed, these languages lose the advantages of static typing, most importantly static type-error detection. Type inference, like dynamic types, can save the programmer from writing typing declarations, and still retain the advantages of static typing, but typically restricts the type system so heterogeneous collections (without explicit sum types) and reflection aren't supported. We will look at a static type system that addresses some of these issues: We allow heterogeneous collections by introducing implicit untagged sum types and we allow record/object-like structures through unordered product types.

# Waitomo: Web-Programming with Objects and Interfaces

Peter Thiemann

Universität Freiburg, Germany

DIKU-IST Workshop, Shonan Village, Japan 22.04.2006

Introduction

Waitomo

Language Features

Status

## Web Sites

- Modern web site
  - static contents (pictures, texts, downloads, . . . )
  - dynamic contents depending on daytime, user profiles, database accesses, etc
- Often more than 50% dynamic contents (search engines, news services, blogs, forums, . . . )
- must be programmed

## Web Sites

- Modern web site
  - static contents (pictures, texts, downloads, . . . )
  - dynamic contents depending on daytime, user profiles, database accesses, etc
- Often more than 50% dynamic contents (search engines, news services, blogs, forums, . . . )
- must be programmed

## Web Programming

- Server-side web programming
  - CGI
  - Scripting: Perl, PHP, Python, Ruby, . . .
  - Integrated: Servlets, JSP, ASP, Web Services, . . .
- Client-side web programming
  - JavaScript, VisualBasic
- Mixed-side web programming
  - Avoids mix of programming languages in web applications
  - Avoids impedance mismatch (different data representation)
  - Avoids explicit programming with networking APIs

## Web Programming

- Server-side web programming
  - CGI
  - Scripting: Perl, PHP, Python, Ruby, ...
  - Integrated: Servlets, JSP, ASP, Web Services, ...
- Client-side web programming
  - JavaScript, VisualBasic
- Mixed-side web programming
  - Avoids mix of programming languages in web applications
  - Avoids impedance mismatch (different data representation)
  - Avoids explicit programming with networking APIs

## Waitomo

Waitomo is ...

- a dialect of Java
  - server-side programming
  - interface to Java API
- a typed dialect of JavaScript with interfaces
  - client-side web programming
  - XML integration (E4X), AJAX
  - due to type system: more robust programs, automatic input checking
- a strict Haskell with object-oriented syntax (no casts)
  - goal: make WASH features (session model; typed, first-class entry fields; typed callback; ...) available
  - type classes $\longleftrightarrow$ interfaces
  - algebraic types $\longleftrightarrow$ classes + union types (later: regexp types) $\longrightarrow$ seemless XML processing

## Waitomo

Waitomo is ...

- a dialect of Java
  - server-side programming
  - interface to Java API
- a typed dialect of JavaScript with interfaces
  - client-side web programming
  - XML integration (E4X), AJAX
  - due to type system: more robust programs, automatic input checking
- a strict Haskell with object-oriented syntax (no casts)
  - goal: make WASH features (session model; typed, first-class entry fields; typed callback; ...) available
  - type classes $\longleftrightarrow$ interfaces
  - algebraic types $\longleftrightarrow$ classes + union types (later: regexp types) $\longrightarrow$ seemless XML processing

## Waitomo

Waitomo is ...

- a dialect of Java
  - server-side programming
  - interface to Java API
- a typed dialect of JavaScript with interfaces
  - client-side web programming
  - XML integration (E4X), AJAX
  - due to type system: more robust programs, automatic input checking
- a strict Haskell with object-oriented syntax (no casts)
  - goal: make WASH features (session model; typed, first-class entry fields; typed callback; ...) available
  - type classes $\longleftrightarrow$ interfaces
  - algebraic types $\longleftrightarrow$ classes + union types (later: regexp types) $\longrightarrow$ seemless XML processing

## Compilation of Waitomo

- Waitomo is influenced by Java, JavaScript, and Haskell
- Waitomo compiles to . . .
  - Java-Servlets (server-side)
  - JavaScript (client-side)
- Waitomo partitions code automatically

## A Waitomo Session

```
function mdrill () {
  standardQuery ("Multiplication Drill",
  <#>
   <p>Choose a factor:
     <input type="text" value="2" name="mpy" /></p>
   <p>Number of exercises:
     <input type="text" value="10" name="rpt" /></p>
   <input type="submit" value="START"
          W:callback={firstExercise(mpy,rpt)} />
  </#>)
}

function firstExercise (Integer mpy, Integer rpt) {
  // actual program logic
}
```

## A Taste of Waitomo for the Web
### Automatic Error Handling

- Before the click




- After the click




## Waitomo Language Features
### High Level

- XML-Literals (XML processing via Java-APIs)
- Input widgets **bind variables**
- Actions defined by callbacks
- Automatic, extensible typ checking on data entry

## Waitomo Language Features
Low Level

- Generic classes with inheritance but without subtyping
- Novel concept of interfaces (with Subtyping)
- Union types (later: regular expression types)
- No cast operation, instead: safe type-switch expression

## Waitomo Language Features
Interfaces

- Interfaces are declared

```
interface Comparable {
  int compareTo (This that);
}
```

`This` is the type of the implementing class

- and **implemented outside a class**

```
Integer implements Comparable {
  int compareTo (Integer that) {
    return this.intValue() - that.intValue();
  }
}
```

- Interfaces are **not types**, but state restrictions on parametric types

## Waitomo Language Features
Interfaces

- Interfaces are declared

```
interface Comparable {
  int compareTo (This that);
}
```

`This` is the type of the implementing class

- and **implemented outside a class**

```
Integer implements Comparable {
  int compareTo (Integer that) {
    return this.intValue() - that.intValue();
  }
}
```

- Interfaces are **not types**, but state restrictions on parametric types

## Waitomo Language Features
Interfaces

- Interfaces are declared

```
interface Comparable {
  int compareTo (This that);
}
```

`This` is the type of the implementing class

- and **implemented outside a class**

```
Integer implements Comparable {
  int compareTo (Integer that) {
    return this.intValue() - that.intValue();
  }
}
```

- Interfaces are **not types**, but state restrictions on parametric types

## Waitomo Language Features
### Interfaces — Use

- In comparison to Java 5.0:

```
interface Comparable<A> {
  int compareTo (A that); }
```

- Type parameter instead of `This` complicates the use
- Java's method signature requires reflexive bound (generic maximum of two values)

```
static <T extends Comparable<T>>
       T max (T x1, T x2)
 { ... x1.compareTo (x2) ... }
```

- Waitomo's method signature

```
static <T extends Comparable>
       T max (T x1, T x2)
 {  ... x1.compareTo (x2) ... }
```


## Waitomo Language Features
### Interfaces — Use

- In comparison to Java 5.0:

```
interface Comparable<A> {
  int compareTo (A that); }
```

- Type parameter instead of `This` complicates the use
- Java's method signature requires reflexive bound (generic maximum of two values)

```
static <T extends Comparable<T>>
       T max (T x1, T x2)
 { ... x1.compareTo (x2) ... }
```

- Waitomo's method signature

```
static <T extends Comparable>
       T max (T x1, T x2)
 {  ... x1.compareTo (x2) ... }
```


## Waitomo Language Features
### Static Methods in Interfaces

- Generic reading and validation of form entries

```
interface Parseable {
  static This parse (String s);
}
```

- "Normal" interface does not work, because there is no object available before parsing is complete.
- `This` is type of implementing class
- Implementation

```
Integer implements Parseable {
  static Integer parse (String s) {
    return new Integer (s);
  }
}
```

## Waitomo Language Features
### Static Interface Methods

```
<T extends Parseable>
  T processEntry (String fieldName) {
  String s = getParameter (fieldName);
  return Parseable@T.parse (s);
}
```

- Static method invocation on type parameter
- Avoids explicit construction of a generic factory

## Waitomo Language Features
Workaround in Java 5.0

```
abstract class ParseableInterface<T> {
  abstract T parse (String s);
}

class IntegerImplementsParseable
  extends ParseableInterface<Integer> { ... }

<T>T processEntry (String fieldName,
                   ParseableInterface<T> pf) {
  String s = getParameter (fieldName);
  return pf.parse (s);
}

Integer i = processEntry ("ifield",
              new IntegerImplementsParseable ());
```

## Waitomo Language Features
Generic Collections

```
interface Collection<A> {
  static This empty ();
  This add (A elem);
  <It implements Iterator<A>> It iterator ();
}
```

- Genericity: all elements have the same type
- Sometimes too strong:
  sufficient that all elements understand the same methods
- Solution:
  existential type ∃A `implements ToString` . A
  written as `* implements ToString` or {`ToString`}
- Advantage:
  multiple interfaces possible {`ToString,ToNumber`}

## Waitomo Language Features
Generic Collections

```
interface Collection<A> {
  static This empty ();
  This add (A elem);
  <It implements Iterator<A>> It iterator ();
}
```

- Genericity: all elements have the same type
- Sometimes too strong:
  sufficient that all elements understand the same methods
- Solution:
  existential type ∃A `implements ToString` . A
  written as `* implements ToString` or {`ToString`}
- Advantage:
  multiple interfaces possible {`ToString,ToNumber`}

## Waitomo Language Features
Union Types

- Task: define tree structure with several types of nodes
- Java: Composite Pattern

```
abstract class BTree {}
class Leaf extends BTree {...}
class Branch extends BTree {
  BTree l, r;
}
```

- Waitomo: Union Type

```
type BTree = Leaf|Branch;
class Leaf {   // body as in Java
}
class Branch { // body as in Java
}
```

## Waitomo Language Features
### Type Switch

- Scrutiny of a union type by type switch

```
type BTree = Leaf|Branch;
function count (BTree t) {
  switch (t) {
  case Leaf l:
    return 0;
  case Branch b:
    return b.l.count () + b.r.count () + 1;
  }
}
```

- Avoids castind and the `instanceof` operator
- Type switch must be exhaustive (default)
- No `IllegalClassCast` exception

## Waitomo Language Features
### Simulating the Visitor Pattern

```
interface Count {
  static int count (This x);
}
Leaf implements Count {
  static int count (Leaf l) {
    return 0;
  }
}
Node implements Count {
  static int count (Node n) {
    return Count@BTree.count (n.left) +
           Count@BTree.count (n.right) + 1;
  }
}
```

- no anticipation required
- interfaces guarantee coherent types

## Waitomo Language Features
### Interface Implementation and Union Types

- Question: if `A implements I` and `B implements I`, does `(A|B) implements I` hold? And vice versa?
- Not trivial because `I` may refer to `This` in covariant and contravariant positions!
- Either direction may not hold
- Explicit `implements` declaration for `(A|B)` allowed

## Waitomo Language Features
### Interface Implementation and Union Types

- Question: if `A implements I` and `B implements I`, does `(A|B) implements I` hold? And vice versa?
- Not trivial because `I` may refer to `This` in covariant and contravariant positions!
- Either direction may not hold
- Explicit `implements` declaration for `(A|B)` allowed

## Waitomo Language Features
### Interface Implementation and Union Types

- Question: if `A implements I` and `B implements I`, does `(A|B) implements I` hold? And vice versa?
- Not trivial because `I` may refer to `This` in covariant and contravariant positions!
- Either direction may not hold
- Explicit `implements` declaration for `(A|B)` allowed

## Waitomo Language Features
### Interface Implementation and Union Types

$$\texttt{A} \lhd \texttt{I} \wedge \texttt{B} \lhd \texttt{I} \quad \mathcal{R} \quad \texttt{A|B} \lhd \texttt{I}$$

| occurrences of `This` in `I` | $\mathcal{R}$ |
|---|---|
| $\leq 1$ arguments/method, no results | $\Leftrightarrow$ |
| $> 1$ arguments/method, no results | $\Leftarrow$ |
| $\leq 1$ arguments/method, result | $\Rightarrow$ |
| $> 1$ arguments/method, result | $\emptyset$ |

- implicit `This` parameter counts also

## Waitomo Language Extension
### Subject/Observer Pattern

- Typed rendition of Subject/Observer pattern
- Problem: mutual dependency of subject and observer
  - `Subject.register` takes `Observer` argument
  - `Object.update` takes a `Subject` argument
- Solution: **multi-headed interfaces**
  aka Haskell's multi-parameter type classes
- Example code also makes use of default code in interface declarations

## Waitomo Language Extension
### Subject/Observer Pattern

- Typed rendition of Subject/Observer pattern
- Problem: mutual dependency of subject and observer
  - `Subject.register` takes `Observer` argument
  - `Object.update` takes a `Subject` argument
- Solution: **multi-headed interfaces**
  aka Haskell's multi-parameter type classes
- Example code also makes use of default code in interface declarations

## Waitomo Language Extension
### Subject/Observer Pattern

- Typed rendition of Subject/Observer pattern
- Problem: mutual dependency of subject and observer
  - `Subject.register` takes `Observer` argument
  - `Object.update` takes a `Subject` argument
- Solution: **multi-headed interfaces**
  aka Haskell's multi-parameter type classes
- Example code also makes use of default code in interface declarations

## Waitomo Language Extension
### Subject/Observer Pattern Programmed

```
interface <Subject, Observer>SubjectObserver {
  class Subject {
    private ArrayList<Observer> observers =
      new ArrayList<Observer> ();
    void register (Observer o) {
      observers.add (o);
    }
    void notify () {
      for (Observer o : getObservers ()) {
        o.update (this);
      }
    }
  }
  class Observer {
    void update (Subject);
  }
}
```

## Waitomo Language Extension
### Syntactic Sugar

- Earlier notation of interfaces is syntactic sugar
- Example of `Comparable` in the full notation

```
interface <This>Comparable {
  class This {
    int compareTo (This that);
  }
}
```

## Status

- Untyped implementation of server part
- Compiler Waitomo → Java
- Type soundness
- Splitting client/server

**To Do**

- Implementation of client part
- Extension of type system to regular expression types (for XML-processing)
- Integration

# Towards Deterministic Top-Down Parsing of LR Languages

Morten Fjord-Larsen <fjord@diku.dk>

Department of Computer Science
University of Copenhagen

DIKU-IST 2006



## Background

### LR Parsing

- LR parsing was introduced by Knuth back in 1965 in his classic article: *On the Translation of Languages From Left to Right*.
- LR($k$) parsers are able to handle a large class of context-free grammars.
- But: LR($k$) parsers constructed using Knuth's method become very large for $k \geq 1$.

### LL Parsing

- LL parsers, on the other hand, are appreciated for their simplicty and compactness.
- LL parsers are easily written by hand.
- But: Strictly weaker than LR. There exists LR languages that are not LL.



## Goal

### Our goal

Our goal is to achieve the power of LR, while retaining the simplicity and compactness of LL parsers.

### Idea

1. Extend context-free grammars to what we call *pair-matching* grammars.
2. Extend definition of context-free LL($k$) grammars to pair-matching grammars (PMLL).
3. Transformation of LR($k$) grammars into PMLL($k$) grammars.

## Generative Grammars

**Definition (Generative Grammar)**

A *generative grammar* $\mathcal{G}$ is a quadruple $(\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$, where

- $\mathcal{T}$ and $\mathcal{N}$ are finite disjoints sets of symbols called *terminals* and *non-terminals*, respectively.
- $\mathcal{P}$ is a finite set of *productions* of the form $\alpha \to \beta$, were $\alpha \in (\mathcal{N} \cup \mathcal{T})^+$ and $\beta \in (\mathcal{N} \cup \mathcal{T})^*$.
- $S \in N$ is a designated non-terminal, called the start symbol.

**Definition (Context-Free Grammar)**

A generative grammar $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$ is said to be *context-free* iff all productions in $\mathcal{P}$ are of the form

- $A \to \alpha$,

where $A \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \mathcal{T})^*$.



## Pair-Matching Grammars

**Definition (Pair-Matching Grammar)**

A grammar $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$ is said to be *pair-matching* iff all productions in $\mathcal{P}$ are of the form

- $A \to \alpha$,
  where $A \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \mathcal{T})^*$
- $A!\ A? \to \varepsilon$,
  where $A!, A? \in \mathcal{N}$ does not occur on the left-hand side on any other production.

**Convention**

- Pair-matching productions $A!\ A? \to \varepsilon$ will not be written explicitly.
- Whenever we write $A?$ or $A!$, we will assume that $A!\ A? \to \varepsilon \in \mathcal{P}$.
- $\mathcal{N}_! = \{A! \mid A!\ A? \to \varepsilon \in \mathcal{P}\}$ and $\mathcal{N}_? = \{A? \mid A!\ A? \to \varepsilon \in \mathcal{P}\}$.



## Pair-Matching Grammars

**Example**

Consider the follow pair-matching grammar:

$$\begin{array}{lcl} S & \to & U\ V \\ U & \to & a\ A! \mid b\ B! \\ V & \to & A?\ c \mid B?\ d \end{array}$$

We have the following (left-most) derivations:

$$\underline{S} \Rightarrow \underline{U}\ V \Rightarrow a\ A!\ \underline{V} \Rightarrow a\ \underline{A!\ A?}\ c \Rightarrow a\ c$$
$$\underline{S} \Rightarrow \underline{U}\ V \Rightarrow b\ B!\ \underline{V} \Rightarrow b\ \underline{B!\ B?}\ d \Rightarrow b\ d$$

## Look-Ahead Sets

### Definition (Look-Ahead Set)

For a production $A \to \alpha$, the *look-ahead set* $\mathrm{LA}(A, \alpha)$ is defined by

$$\mathrm{LA}_k(A, \alpha) = \{k : x \mid S \Rightarrow^* zA\beta \Rightarrow z\alpha\beta \Rightarrow^* zx\}$$

where $x \in (\mathcal{T} \cup \mathcal{N}_?)^*, z \in \mathcal{T}^*\mathcal{N}_!^*, \beta \in (\mathcal{N} \cup \mathcal{T})^*$.

### Relation to FIRST and FOLLOW

For a production $A \to \alpha$, the *look-ahead set* $\mathrm{LA}(A, \alpha)$, is the smallest set such that

$$\begin{aligned} &\mathrm{LA}(A, \alpha) \supseteq \mathrm{FIRST}(\alpha) && \\ &\mathrm{LA}(A, \alpha) \supseteq \mathrm{FOLLOW}(A) && \text{if } \alpha \Rightarrow^* \varepsilon \\ &\mathrm{LA}(A, \alpha) \supseteq \mathrm{FOLLOW}(AA_n?) && \text{if } \alpha \Rightarrow^* A_1! \ldots A_n! \end{aligned}$$



## Pair-Matching LL($k$) Grammars

### Definition (Pair-Matching LL($k$) Grammar)

A pair-matching grammar $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$ is said to be PMLL($k$) iff for all production rules $A \to \alpha_1 \mid \ldots \mid \alpha_n$,

$$\mathrm{LA}_k(A, \alpha_i) \cap \mathrm{LA}_k(A, \alpha_j) = \emptyset$$

for each $1 \leq i, j \leq n, i \neq j$.



## A Non-LL($k$) Language

### Example

Consider the following LR(0) grammar $\mathcal{G}$:

$$\begin{aligned} S &\to A \mid B \\ A &\to a\ A\ b \mid 0 \\ B &\to a\ B\ b\ b \mid 1 \end{aligned}$$

- $\mathcal{L}(\mathcal{G}) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$,
- $\mathcal{L}(\mathcal{G})$ is not LL($k$) for any $k$.
- The following PMG generates the same language:

$$\begin{aligned} S &\to M\ A? \mid M\ B? \\ M &\to a\ M\ A?\ b\ A! \mid 0\ A! \\ &\mid a\ M\ B?\ b\ b\ B! \mid 1\ B! \end{aligned}$$

## A Non-LL($k$) Language

### Example

Consider the following LR(0) grammar $\mathcal{G}$:

$$\begin{array}{lcl} S & \rightarrow & A \mid B \\ A & \rightarrow & a\ A\ b \mid 0 \\ B & \rightarrow & a\ B\ b\ b \mid 1 \end{array}$$

- $\mathcal{L}(\mathcal{G}) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$,
- $\mathcal{L}(\mathcal{G})$ is not LL($k$) for any $k$.
- The following PMG generates the same language:

$$\begin{array}{lcl} S & \rightarrow & M\ C \\ C & \rightarrow & A? \mid B? \\ M & \rightarrow & a\ M\ L \mid 0\ A! \mid 1\ B! \\ D & \rightarrow & A?\ b\ A! \mid B?\ b\ b\ B! \end{array}$$



## Merging

### Definition (Merging)

The result of *merging* non-terminals $A$ and $B$ with productions rules

$$\begin{array}{lcl} A & \rightarrow & \alpha_1 \mid \ldots \mid \alpha_m, \\ B & \rightarrow & \beta_1 \mid \ldots \mid \beta_n \end{array}$$

is a non-terminal $[A, B]$ with production rule:

$$\begin{array}{rcl} [A, B] & \rightarrow & \alpha_1\ A! \mid \ldots \mid \alpha_m\ A! \\ & \mid & \beta_1\ B! \mid \ldots \mid \beta_n\ B! \end{array}$$

### Proposition

$\mathcal{L}(A) = \mathcal{L}([A, B]\ A?)$ *and* $\mathcal{L}(B) = \mathcal{L}([A, B]\ B?)$



## Transformation to PMLL(1): Another Example

$$\begin{array}{lcl} S & \rightarrow & L = R \\ S & \rightarrow & R \\ L & \rightarrow & \mathrm{id} \\ L & \rightarrow & * R \\ R & \rightarrow & L \end{array}$$

### Comments

- Since $\mathrm{FIRST}(L) \cap \mathrm{FIRST}(R) \neq \emptyset$, we proceed by merging $L$ and $R$ into $[L, R]$.

## Transformation to PMLL(1): Another Example

$$
\begin{array}{rcl}
S & \to & [L, R]\ C \\
C & \to & L? = R \\
C & \to & R? \\
L & \to & \mathrm{id} \\
L & \to & * R \\
R & \to & L \\
\\
[L, R] & \to & \mathrm{id}\ L! \\
[L, R] & \to & *\ R\ L! \\
[L, R] & \to & L\ R!
\end{array}
$$

### Comments

- Since $\mathrm{FIRST}(L) \cap \mathrm{FIRST}(R) \neq \emptyset$, we proceed by merging $L$ and $R$ into $[L, R]$.
- Since $*, \mathrm{id} \in \mathrm{FIRST}(L)$, unfold $L$.



## Transformation to PMLL(1): Another Example

$$
\begin{array}{rcl}
S & \to & [L, R]\ C \\
C & \to & L? = R \\
C & \to & R? \\
L & \to & \mathrm{id} \\
L & \to & * R \\
R & \to & L \\
\\
[L, R] & \to & \mathrm{id}\ U \\
[L, R] & \to & *\ R\ U \\
U & \to & L! \\
U & \to & R!
\end{array}
$$

### Comments

- Since $\mathrm{FIRST}(L) \cap \mathrm{FIRST}(R) \neq \emptyset$, we proceed by merging $L$ and $R$ into $[L, R]$.
- Since $*, \mathrm{id} \in \mathrm{FIRST}(L)$, unfold $L$.
- Perform left-factoring.



## Transformation to PMLL(1): Algorithm Sketch

### Input: $\mathcal{G}$ — A grammar (free of left-recursion)

Repeat the following steps until no changes are made:

1. Perform full left-factoring.
2. Replace each $A \to B_1\ \alpha_1$ and $A \to B_2\ \alpha_2$ for which $\mathrm{FIRST}(B_1\ \alpha_1) \cap \mathrm{FIRST}(B_2\ \alpha_2) \neq \emptyset$ with

$$
\begin{array}{l}
A \to [B_1, B_2]\ C \\
C \to B_1?\ \alpha_1 \mid B_2?\ \alpha_2
\end{array}
$$

3. For each $A \to b\alpha$ and $A \to B\beta$ for which $b \in \mathrm{FIRST}(B)$, unfold $B$.

## Transformation to PMLL(1)

**Conjecture (Strength)**

*If $\mathcal{G}$ is an LR(1) grammar, then the transformed grammar $\mathcal{G}'$ will be PMLL(1).*



## Top-Down Parsing Algorithm for PMGs

TOP-DOWN-PARSE ($w$)

```
1:  L ← [S]
2:  R ← w⊥
3:  while L ≠ [] do
4:    X ← POP(L)
5:    Y ← TOP(R)
6:    if X = A! then
7:      PUSH(R, A?)
8:    else
9:      if M[X, Y] = A → B_1 ... B_m then
10:       PUSH(L, B_1, ..., B_m)
11:       output A → B_1 ... B_m
12:     else
13:       if X = Y then
14:         Pop(R)
15:       else
16:         ERROR()
```

**Example**

$$\begin{array}{lcl} S & \rightarrow & U\ V \\ U & \rightarrow & a\ A! \mid b\ B! \\ V & \rightarrow & A?\ c \mid B?\ d \end{array}$$

**LL Parse Table: $M$**

| | $a$ | $b$ | $A?$ | $B?$ |
|---|---|---|---|---|
| $S$ | $U\ V$ | $U\ V$ | | |
| $U$ | $a\ A!$ | $b\ B!$ | | |
| $V$ | | | $A?\ c$ | $B?\ d$ |



## Summary

**Conclusion**

- We conjecture that PMLL($k$) grammars have the expressive power of LR($k$) grammars. (Look for a formal proof in my upcoming thesis).
- We believe that PMLL parsing retains much of the simplicity known from LL parsing.
- Through grammar transformations, PMLL parsing provides an alternative to LR parsing.

**Ongoing and Future Work**

- PMLL *covers* for LR grammars.
- Establish size bounds on the transformed grammars.
- Comparison to LR parsing.

# The Density of Words in Formal Languages

Jakob Grue Simonsen

DIKU

**Abstract**

Given a language $L$ over some alphabet, how large a fraction of the total number of available words of a given length $n$ is contained in $L$? And can we say anything about the fraction as $n$ tends to infinity? The answers to these questions have surprising connections to many areas of computer science, *e.g.* the "average" fraction above corresponds to the optimal compression rate obtainable for sufficiently well-behaved languages. In this talk, we will discuss some old and new ways to find such answers; we will show when they can definitely be found fast, when they definitely cannot be found in reasonable time, and when they definitely cannot be found at all.

# Shuffle Expressions for Static XML Type-checking

Tadahiro SUDA (University of Tokyo)
tada@is.s.u-tokyo.ac.jp
Haruo Hosoya (University of Tokyo)
hahosoya@is.s.u-tokyo.ac.jp
Akihiko TOZAWA (IBM Tokyo Research Laboratory)
atozawa@jp.ibm.com



## Static XML Type-checking

**Schemas:** User-descriptions for structural constrains on XML documents.

**Static XML type-checking:** Verifying generated documents to conform to the user-given schema.
(cf. XDuce [Hosoya], CDuce [Frisch], etc.)



## Shuffle Expressions

Usually, a schema uses regular expressions to describe an element sequence.

**ex.)** Schema `name,email*` allows

```
<name>T.Suda</name>
<email>tada@is.s.u-tokyo.ac.jp</email>
<email>tada@triax.jp</email>
```

Some schema languages allow shuffle expressions.
(Relax NG [Murata], XML-Schema [W3C], etc.)

- allow elements to occur in any order.
- useful for describing record-like data.

## Example

BibTeX-like schema

$$\langle \texttt{title} \rangle \& \langle \texttt{author} \rangle \& \langle \texttt{year} \rangle = \{ \langle \texttt{title} \rangle \langle \texttt{author} \rangle \langle \texttt{year} \rangle, \langle \texttt{title} \rangle \langle \texttt{year} \rangle \langle \texttt{author} \rangle, \langle \texttt{author} \rangle \langle \texttt{title} \rangle \langle \texttt{year} \rangle, \langle \texttt{author} \rangle \langle \texttt{year} \rangle \langle \texttt{title} \rangle, \langle \texttt{year} \rangle \langle \texttt{title} \rangle \langle \texttt{author} \rangle, \langle \texttt{year} \rangle \langle \texttt{author} \rangle \langle \texttt{title} \rangle \}$$



## Static XML Type-checking with Shuffle Expressions

Existing static type-checkers for XML processing programs don't handle shuffle expressions.

(Reason)

- Those algorithms are based on regular expressions.
- A regular expression equivalent to a shuffle expression has a factorial size of the shuffle expression.

$\longrightarrow$ A naive algorithm easily blows up.



## Our Work

- Introduce Ternary Decision Diagrams (TDDs) as an extension of BDDs.
- Reduce operations on shuffle expressions to operations on TDDs.
- Construct efficient algorithms based on TDDs, for static XML type-checking with shuffle expressions.

## Outline of the Remaining Presentation

- Shuffle Expressions
- Ternary Decision Diagrams (TDDs)
- Static XML Type-checking with TDDs



## Most General Shuffles

**&-operator (shuffle):** allows any "interleave" of a sequence from the left-side and a sequence from the right-side.

$$T\&T' = \{u \mid u \text{ is any interleave of } v \text{ and } w, v : T, w : T'\}$$

**ex.)** $ab \,\&\, xy = \{abxy, axby, axyb, xaby, xayb, xyab\}$



## Shuffle Expressions in Real Schema Languages

**Relax NG:** allows the most general shuffles (&-operator is usable on any types).

- We failed to find an efficient type-checking algorithm with them.

**XML-Schema:** allows only the form

$$a_1\& \cdots \& a_m \& b_1? \& \cdots \& b_n?$$

where each $a_i$, $b_j$ is a single, distinct label.

- Expressiveness seems not sufficient.
  **ex.)** $\langle \texttt{name} \rangle \& \langle \texttt{email} \rangle *$ is NG.

# Our Shuffle Expressions

We want

- The label occurs any-times anywhere
  **ex.)** $\langle \texttt{name} \rangle \& \langle \texttt{email} \rangle *$
- Choice of shuffle expressions
  **ex.)** $\langle \texttt{student} \rangle \& \langle \texttt{year} \rangle | \langle \texttt{teacher} \rangle \& \langle \texttt{position} \rangle$
- Choice from infinite sets of labels
  **ex.)** $\langle \texttt{title} \rangle \& \overline{\{\langle \texttt{title} \rangle\}} *$ ( $\overline{\{..\}}$ is the complement of $\{..\}$ )

But we don't need (not useful):

- The label occurs "exactly $n$ times"
  **ex.)** $\langle \texttt{name} \rangle \& \langle \texttt{name} \rangle \& \langle \texttt{name} \rangle$



# Our Shuffle Expressions

$\mathrm{Sh} = \mathrm{Base} \mid \mathrm{Sh} \& \mathrm{Sh}$ (shuffle) $\mid \mathrm{Sh} | \mathrm{Sh}$ (choice)
$\mathrm{Base} = \mathrm{L}$ (label set) $\mid \mathrm{L}?$ (optional) $\mid \mathrm{L}*$ (repetition)
$\mid \mathrm{L}+$ (one or more)

where $\mathrm{Sh} \& \mathrm{Sh}'$ requires that

the labels in $\mathrm{Sh}$ and those in $\mathrm{Sh}'$ are disjoint.



# Closer Look at Our Shuffle Expressions

**Permutability:** If $\nu$ in $\mathrm{Sh}$, then permutations of $\nu$ in $\mathrm{Sh}$.
**ex.)** $\langle \texttt{name} \rangle \& \langle \texttt{email} \rangle *$

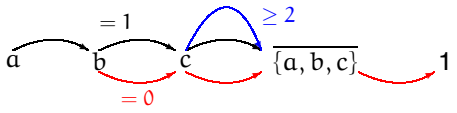
- A permutable shuffle expression represents a set of multi-set of labels.
- But our shuffles classify the times of label occurrences into three:
  – zero
  – one
  – two or more

# Ternary Decision Diagrams

**Ternary Decision Diagrams (TDDs):** Extension of BDDs with "two-or-more" edges in addition to "zero" and "one" edges.

- TDDs can represent all our shuffle expressions !

**ex.)** $a\&b?\&c*$






# Static Type-checking with TDDs

```
type Data = name & tel? & email*
fun trans (val x as Data) : (name|tel|email)* =
  match x with
    y as (name & tel & email+) -> y
  | _  -> ()
```

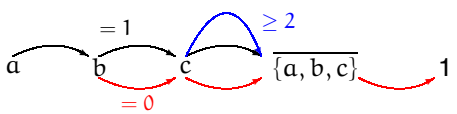
1. Infer the type of documents generated by the function: `name&tel&email+|()`.
2. Check the containment `name&tel&email+|() ⊆ (name|tel|email)*`.

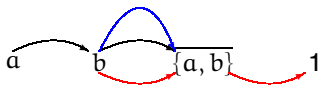
⟶ need to compute
intersections, complements, and emptiness.



# Intersection of TDDs

Let's compute the intersection of $a\&b?\&c*$ and $a\&b*$ !

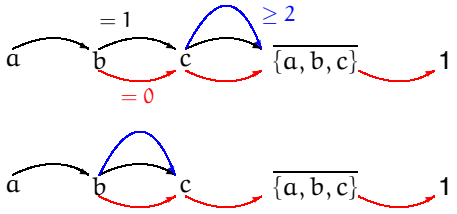



Transform $a\&b*$.

# Intersection of TDDs

Let's compute the intersection of $a\&b?\&c*$ and $a\&b*$ !




Compute the intersection.



# Intersection of TDDs

Let's compute the intersection of $a\&b?\&c*$ and $a\&b*$ !

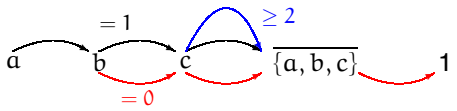



Get $a\&b?$ .



# Complement of TDD

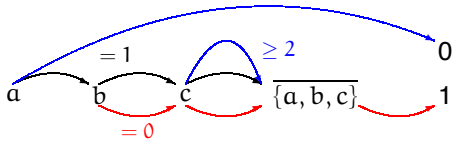
Let's compute the complement of $a\&b?\&c*$ !




Describe edges to "0".

# Complement of TDD

Let's compute the complement of $a\&b?\&c*$ !

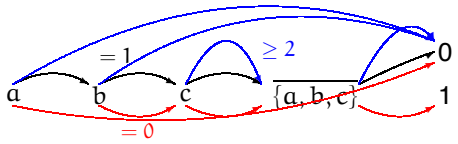



Describe edges to "0".



# Complement of TDD

Let's compute the complement of $a\&b?\&c*$ !

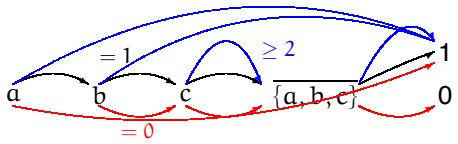



Turn "0" and "1".



# Complement of TDD

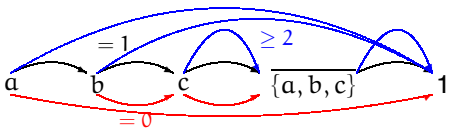
Let's compute the complement of $a\&b?\&c*$ !




Get.

# Complement of TDD

Let's compute the complement of $a\&b?\&c*$ !




Get.



# Mixing Regular Expressions and Shuffle Expressions

```
type Data2 = name,tel?,email*
fun trans2 (val x as Data2) : name&tel&email =
  match x with
    name,tel,email* -> x
  | name,email* -> x,tel
```

1. Infer the type of generated documents:
   `name,tel,email*|name,email*,tel`.
2. Check the containment between this regular expression and the shuffle expression `name&tel&email`.



# Containment check

**Between regular expressions:** Well-known.

**Between shuffle expressions:** Already presented:
$\mathrm{Sh} \subseteq \mathrm{Sh}' \iff \mathrm{Sh} \cap \overline{\mathrm{Sh}'} = \phi$

**Between regular expression and shuffle expression:**
For any regular expression $\mathrm{R}$, we can easily compute $\mathrm{Sh}(\mathrm{R})$: the smallest shuffle expression (TDD) that includes $\mathrm{R}$.

Then, we can prove:

$$\mathrm{R} \subseteq \mathrm{Sh} \iff \mathrm{Sh}(\mathrm{R}) \subseteq \mathrm{Sh}$$

We thus only need to operate on TDDs !

## Containment check

**Between shuffle expression and regular expression:**

$\mathrm{Sh} \subseteq \mathrm{R} \Longleftrightarrow \overline{\mathrm{R}} \subseteq \overline{\mathrm{Sh}} \Longleftrightarrow \mathrm{Sh}(\overline{\mathrm{R}}) \subseteq \overline{\mathrm{Sh}}$

Computing $\overline{\mathrm{R}}$ is exponential in general, but known to be efficient in practice.

snc / 25

## Type Inference with TDDs

We realize the infrence of schemas by introducing new operations of TDDs. (Omit details.)

snc / 26

## Related Work

Sheaves Automata [Dal Zilio, Lugiez.'03]

- By Presburger formula, describe occurrence-times of each label.
- If deterministic, unions, intersections, complements, etc. are computable.
- Presburger seems overly more expressive than necessary e.g., can express "$\mathfrak{a}$ and $\mathfrak{b}$ occur the same time"

snc / 27

## Future Work

- Implementation and experiments.
- “Top-down” strategy for containment-check.



# Shuffle Expressions for Static XML Type-checking

Tadahiro SUDA (University of Tokyo)
tada@is.s.u-tokyo.ac.jp
Haruo Hosoya (University of Tokyo)
hahosoya@is.s.u-tokyo.ac.jp
Akihiko TOZAWA (IBM Tokyo Research Laboratory)
atozawa@jp.ibm.com

# XML Transformation Language Based on Monadic Second Order Logic



Kazuhiro Inaba
Haruo Hosoya

IST, the University of Tokyo

DIKU-IST 2006

## Monadic Second-order Logic (MSO)

- First-order logic extended with “second-order variables” ranging over sets of elements

ex. $\forall A.(A \neq \phi \Rightarrow \exists x.\ (x \text{ in } A\ \&\ \forall y.(y \text{ in } A \Rightarrow x \leqq y)))$

- Long history of study
  - Büchi 1960, Fagin 1974, Thomas 1982, ...

## Monadic Second-order Logic (MSO)

- As a foundation of **XML processing ...**
  - XML Query languages provably MSO-equivalent in expressiveness (Neven 2002, Koch 2003)
  - Theoretical model of XML Transformation with MSO as a sub-language for node selection (Maneth 1999, 2005)

## Goal of This Work

- Bringing MSO formulae themselves into an actual language system for XML processing!

- **MTran**: a language for XML transformation
  - Querying XML by **MSO formulae**
  - Surface language design that leverages the strength of MSO queries
  - **Efficient algorithm** to process MSO Queries

## Outline

- Why MSO queries?
  - Regularity
  - No explicit recursions
  - N-ary queries
- Language design
- Algorithm for MSO queries

## Why MSO? (1) Regularity

Each component is expressible in traditional query languages (e.g., XPath), but **combining them is difficult**.

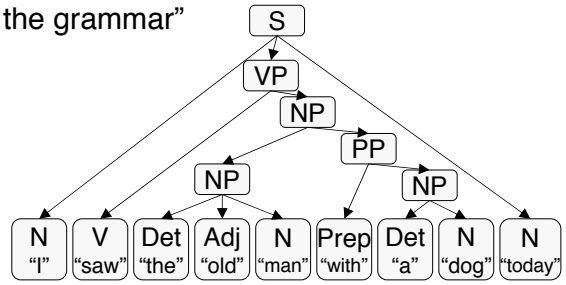
- Generating a Table of Contents
  - "Select all <h2> elements *y* appearing between an (already selected) element *x* and the next <h1> element"
    - "all <h2> elements after *x*"
    - "the <h1> element *z* next to x"
    - "all <h2> elements before *z*"

```
x<y & y in <h2> &
 all1 z.(x<z & z in <h1> => y<z)
```

```
<h1>
... <h2> ....
..... <h2> ..
.... <h3> ...
<h1>
... <h2> ....
.. <h3> .....
<h1>
```

# Why MSO?
# (2) No-recursion

- LPath[Bird et atl., 2005] Linguistic Queries
  - "Select all nodes *y* that immediately follows after *x* in some proper analysis according to the grammar"




# Why MSO?
# (2) No-recursion

- Linguistic Queries
  - LPath extends XPath with a primitive defined by the following algorithm:
    - "recursively go down from *x* to rightmost child until it reaches a leaf, and from *y* to the leftmost leaf respectively, ..."
  - In MSO, **using second-order variables**, we directly say "in some analysis, *y* follows after *x*."

```
pred proper_analys(var2 A) =
 all1 x.(x notin A    A//x | x//A);
pred follow_in(var2 A, var1 x, var1 y) =
 x in A & y in A & all1 z.(z in A => x<z <=> y≦z);

ex2 A. (proper_analysis(A) & follow_in(A,x,y))
```

# Why MSO?
# (3) N-ary Queries

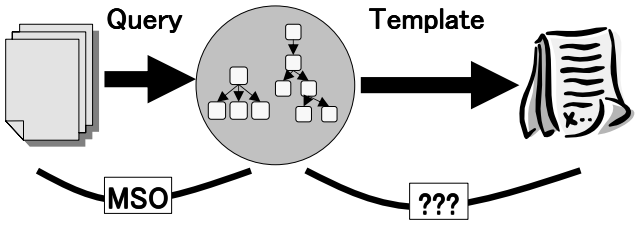
- N-ary Query
  - "Select all 3-tuples of <foo>, <bar>, and <buz> elements with a common parent node."

```
ex1 p. (p/x:<foo> & p/y:<bar> & p/z:<buz>)
```

MSO formulae with N free variables naturally defines N-ary queries

## Question

- "What is a design principle for templates that fully exploits the power of MSO?"




## Design Principle of Transformation Templates

- Corresponding to the strength of MSO...
  - No-recursion
    - ⇒ **"Visit" templates**
  - N-ary queries
    - ⇒ **Nested templates and variable scopes**

## "Visit" Templates

- MTran:

Implicitly applied to all <a>'s

```
{visit x :: x in <a> :: b[x]}
```

- XSLT:

Explicit recursion

```
<xsl:stylesheet ...>
  <xsl:template match="a">
    <b><a><xsl:apply-templates/></a></b>
  </xsl:template>
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates/></xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

## Nested Templates

- MTran:

```
{visit x :: x in <div> ::
   {visit y from x :: y in <selflink> ::
     @href[{gather z :: x/z:@id :: z}]}}
```

Nested templates and variable scopes

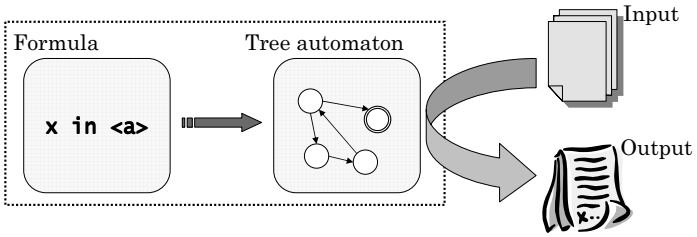
- XSLT:

```
<xsl:stylesheet ...>
  <xsl:template match="div">
    <xsl:copy><xsl:apply-templates>
     <xsl:with-param name="x" select="."/>
    </xsl:apply-templates></xsl:copy>
  </xsl:template>
  <xsl:template match="selflink">
    <xsl:param name="x"/><xsl:attribute ...>
    <xsl:value-of select="$x/@id"/>
    ...
```

Flat templates and parameter passing

## Usual 2-step strategy

① Compile a formula to a tree automaton
② Run queries using the automaton



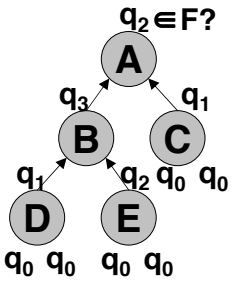

## Our Approach

① Compilation
- Exploit MONA[Klarlund et al.,1999] system

② Querying by Tree Automata
- We developed **O(t+s)** algorithm
  - Previous best known result was **O($t^N$+s)**

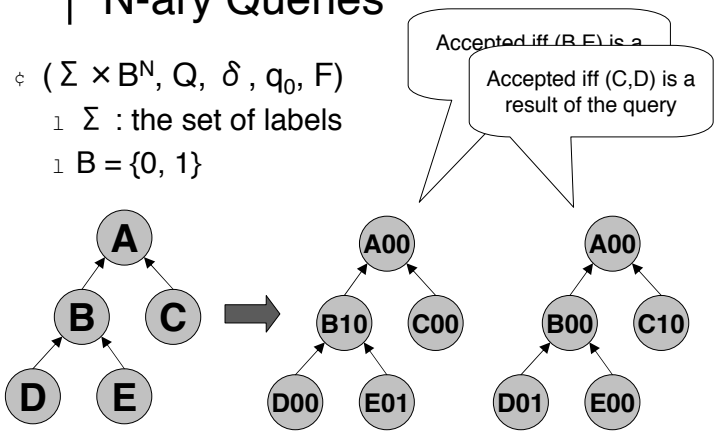
## Definition (Tree Automata)

- Deterministic Bottom-up Tree Automaton (DBTA)
  - $\Sigma$ : alphabet
  - Q : the set of states
  - $\delta$ : $Q \times Q \times \Sigma \rightarrow Q$
  - $q_0$ : initial state
  - F : accepting states



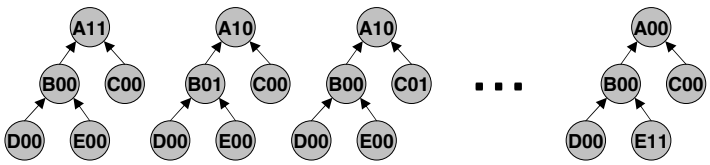

## DBTA Representation of N-ary Queries

- ( $\Sigma \times B^N$, Q, $\delta$ , $q_0$, F)
  - $\Sigma$ : the set of labels
  - B = {0, 1}

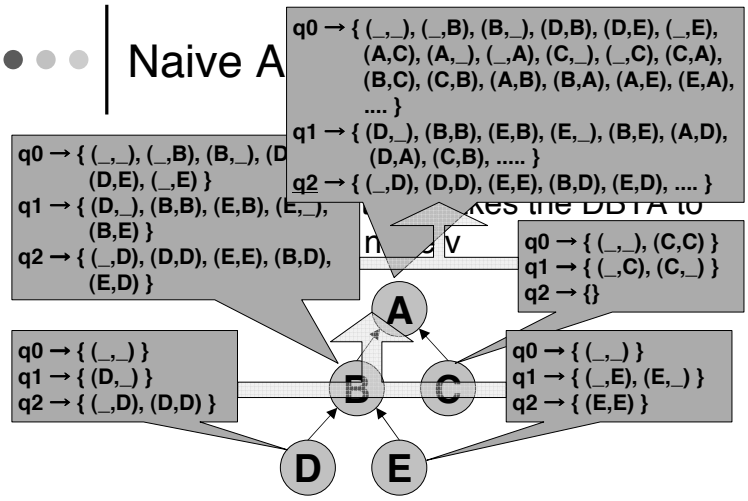



## Naive Algorithm

- Testing all marking patterns → **O( $t^{N+1}$ )**
  - $O(t^N)$ possible markings
  - O(t) run per each marking



## Naive A

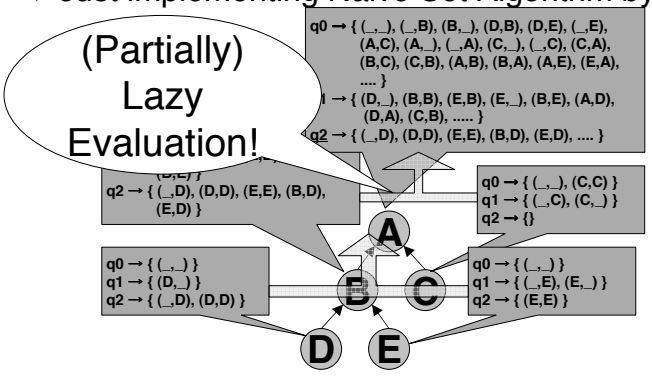



## Naive Algorithm using “sets”

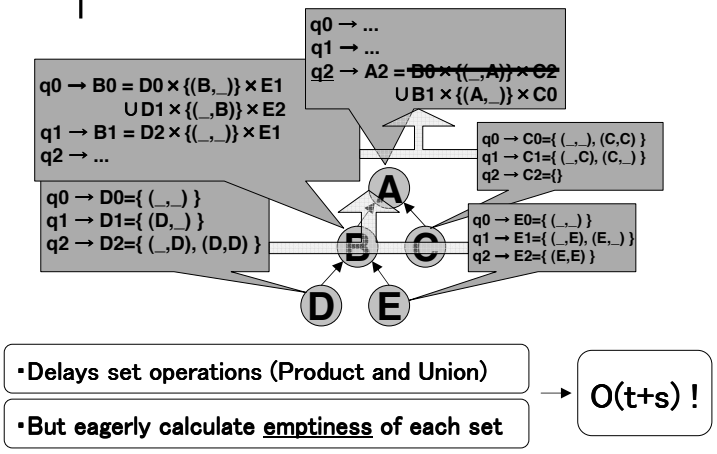
- Still **O( $t^{N+1}$ )**
  - Sets of size O($t^N$)
  - O(t) operations over the sets
- 2-phase approach [Berlea and Sidel, 2001]
  - Splitting the calculation into one Bottom-up run + one Top-down run
  - **O($t^N$)**

## Our Approach

- Just implementing Naive Set Algorithm by...



## Our Approach




- Delays set operations (Product and Union)
- But eagerly calculate emptiness of each set

→ O(t+s) !

## Related Work

- fxgrep, fxt [Berlea et al. 2001]
  - Node selection by path-like language
    - Implemented using tree automata, with 2-phase approach
    - Binary queries
- XSLT 1.0 [Clark et al. 1999]
- XSLT 2.0 (draft) [Kay 2005]
  - First-order “exist” and “forall”

## Related Work

- TL [Maneth 2005]
  - Node selection by MSO formulae
    - For unary “patterns” and binary “selections”
    - No general N-ary queries
  - Templates with explicit recursion
  - Static Type Checking

# Summary

- MTran: a language for XML Transformation
  - Querying XML by MSO formulae
    - Regularity
    - No recusrion
    - N-ary queries
  - Efficient Algorithm
    - Tree automata
    - Partially lazy set operations
- Future Work
  - Static Type Checking
  - More expressive logic?

**Thank you for listening.**

# Author Index


Andersen, Jesper, 96
Artho, Cyrille, 60
Avery, James, 9

Biere, Armin, 60

Fjord-Larsen, Morten, 147

Gade, Johan, 28
Garoche, Pierre-Loïc, 60
Grue, Klaus, 49

Hannemann, Jan, 90
Hauberg, Søren, 36
Henglein, Fritz, 71
Honiden, Shinichi, 60
Hosoya, Haruo, 154, 164
Hu, Zhenjiang, 28, 116

Inaba, Kazuhiro, 164

Jones, Neil D., 1

Kataoka, Satoshi, 48
Kristiansen, Lars, 1

Li, Xin, 65

Maeda, Toshiyuki, 123
Masuhara, Hidehiko, 90
Matsuda, Kazutaka, 28
Matsuzaki, Kiminori, 110
Mogensen, Torben, 137
Morihata, Akimasa, 116

Nakano, Keisuke, 28
Nielsen, Morten Ib, 78

Ogawa, Mizuhito, 65
Ohkawa, Noriyuki, 110

Quenum, Jose Ghislain, 50

Sekizawa, Toshifusa, 20
Simonsen, Jakob Grue, 153
Skelboe, Stig, 35
Sommer, Christian, 60
Suda, Tadahiro, 154

Takahashi, Koichi, 20
Tanabe, Yoshinori, 20
Thiemann, Peter, 138
Tozawa, Akihiko, 154

Yonezawa, Akinori, 123
Yuasa, Yoshifumi, 20

Zweimüller, Boris, 60



After Work
Tokyo Food Theater, Akihabara