Kilim:
Isolation-Typed Actors for Java

秋山 茂樹

Secure Compiler Seminar (2011/1/12)

Kilim:
Isolation-Typed Actors for Java

- Java & 共有メモリマシン上で
  actor ライブラリ Kilim を設計 & 実装

  1) ultra-lightweight threads

  2) a message-passing framework

  3) **isolation-aware messaging**

- 静的な手続き内ヒープ解析により、
  安全な zero copy messaging を実現

Kilim:
Isolation-Typed Actors for Java

- 著者:

  Sriram Srinivasan, Alan Mycroft

  (University of Cambridge Computer Laboratory)

- 発表場所:

  ECOOP 2008

  (22nd European Conference on Object-Oriented Programming)

発表の流れ

1. Kilim の概要
2. Core Language
3. Heap Graph Construction
4. Isolation Capability Checking
5. まとめ

並列分散プログラミングの現状

- multiple cores in one processor,
  multiple NUMA processors in one node,
  many nodes in a data center,
  many data centers
- 問題点:
  - レイヤ間でプログラミングモデルが異なる
  - 共有メモリモデル (shared object, lock) は
    correctness, fairness, efficiency を得るのが難しい

Actor モデル

- independent communicating sequential entities
  - no sharing
  - message passing
- 利点
  - component-oriented testing
  - elimination of data races
  - unification of local and distributed programming
  - better optimization opportunity
  - failure-independence between actors

Motivation

- split-phase workload (CPU, I/O) や service-oriented workflow をうまく扱える言語
  - Actor の並行で疎結合であるという性質を利用
  - すぐに実用化するための要求:
    - Java syntax や JVM に変更を加えない
    - 軽量な actor 生成
    - 高速なメッセージング
    - 言語/処理系非依存な手法

Kilim

- 次の特徴を持つ actor フレームワーク
  - Ultra-lightweight threads
  - **メッセージ型**
  - **メッセージの静的な isolation を保証**
  - ランタイムライブラリ (mailbox, alting, etc...)

複数のスレッドで
メッセージが共有されない

メッセージ

- メッセージ型 (Message interface)
  - フィールドとして許されるのは:
    - Primitive types
    - Message
    - Primitive types と Message の配列
  - **たかだか1つのオブジェクトから参照される**
  - たかだか1つの actor に所有される (**isolation**, interference-freedom)

メッセージはヒープ内で**木構造**をとる

Isolation qualifiers と静的ヒープ解析 (shape analysis) を用いて保証

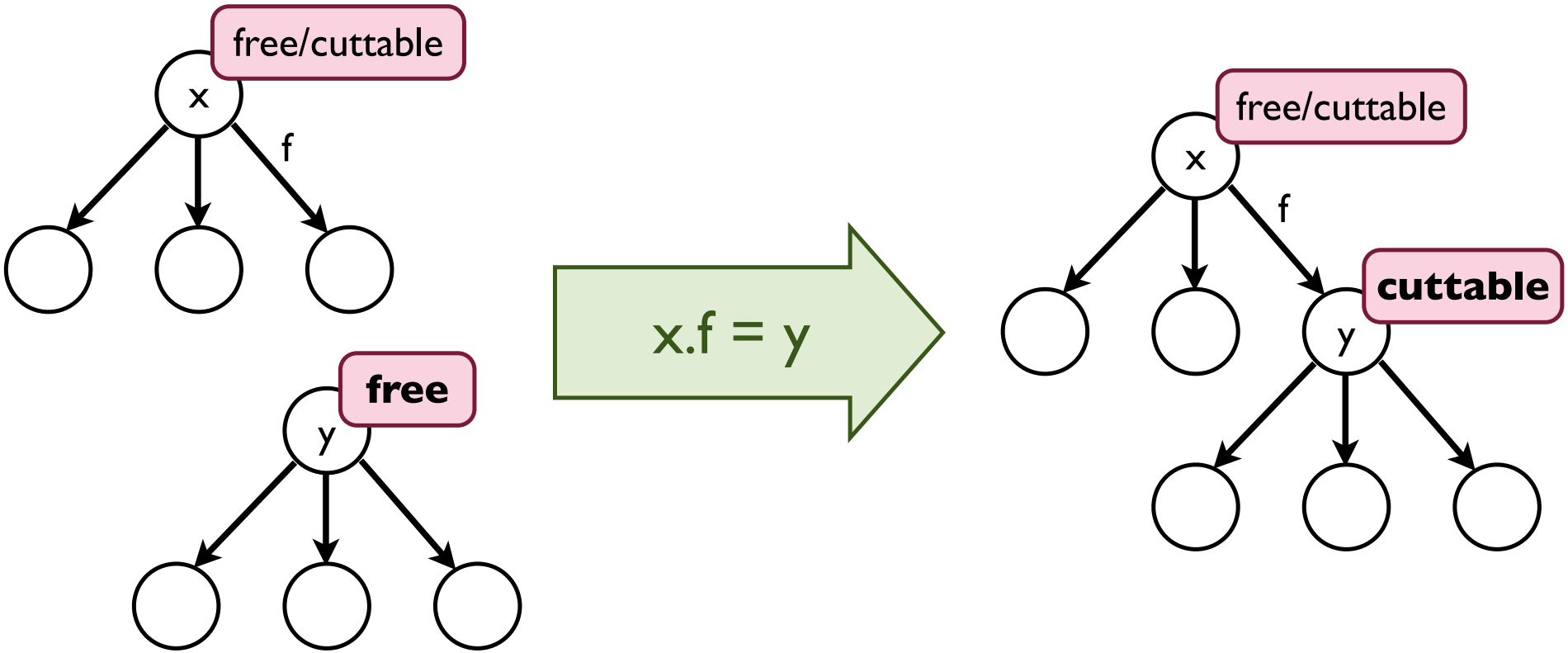
Isolation qualifiers / capabilities

- Message に対する操作の権限 (capability)
  - free:
    - フィールドを変更可能
    - ヒープ内から参照されていない
      (= ヒープ上の木構造を考えたとき root である)
  - cuttable:
    - フィールドを変更可能
  - safe:
    - フィールドを変更不能 (cf. C's const)

(ローカル変数を考えなければ)
安全に send 可能

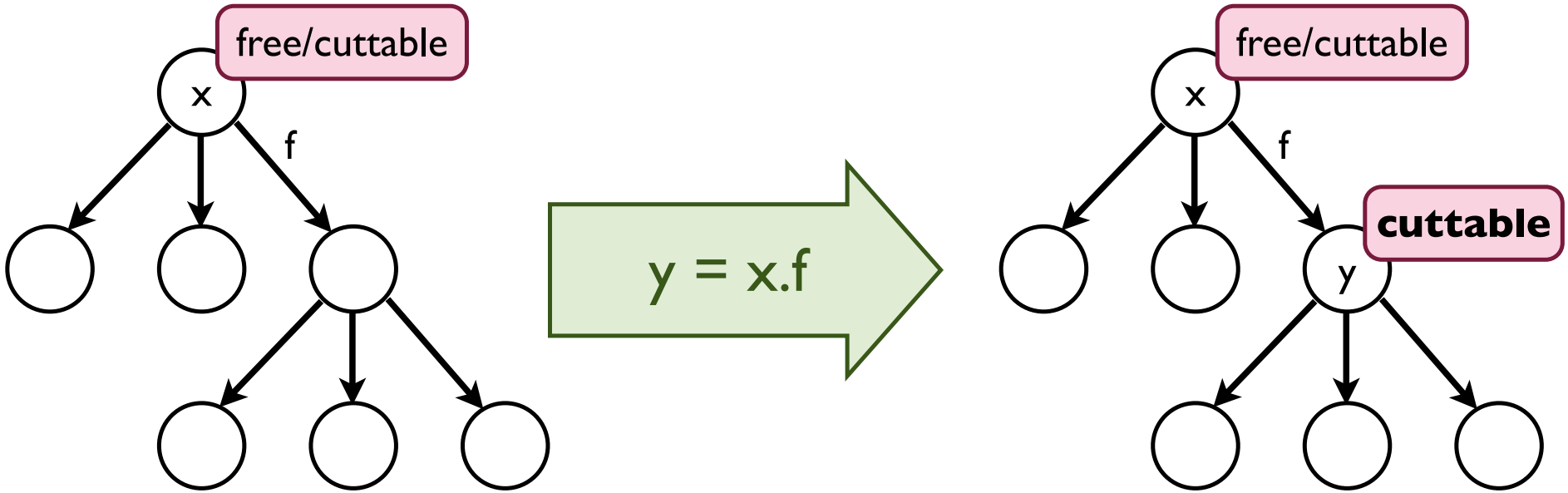
free と cuttable

- フィールドの更新



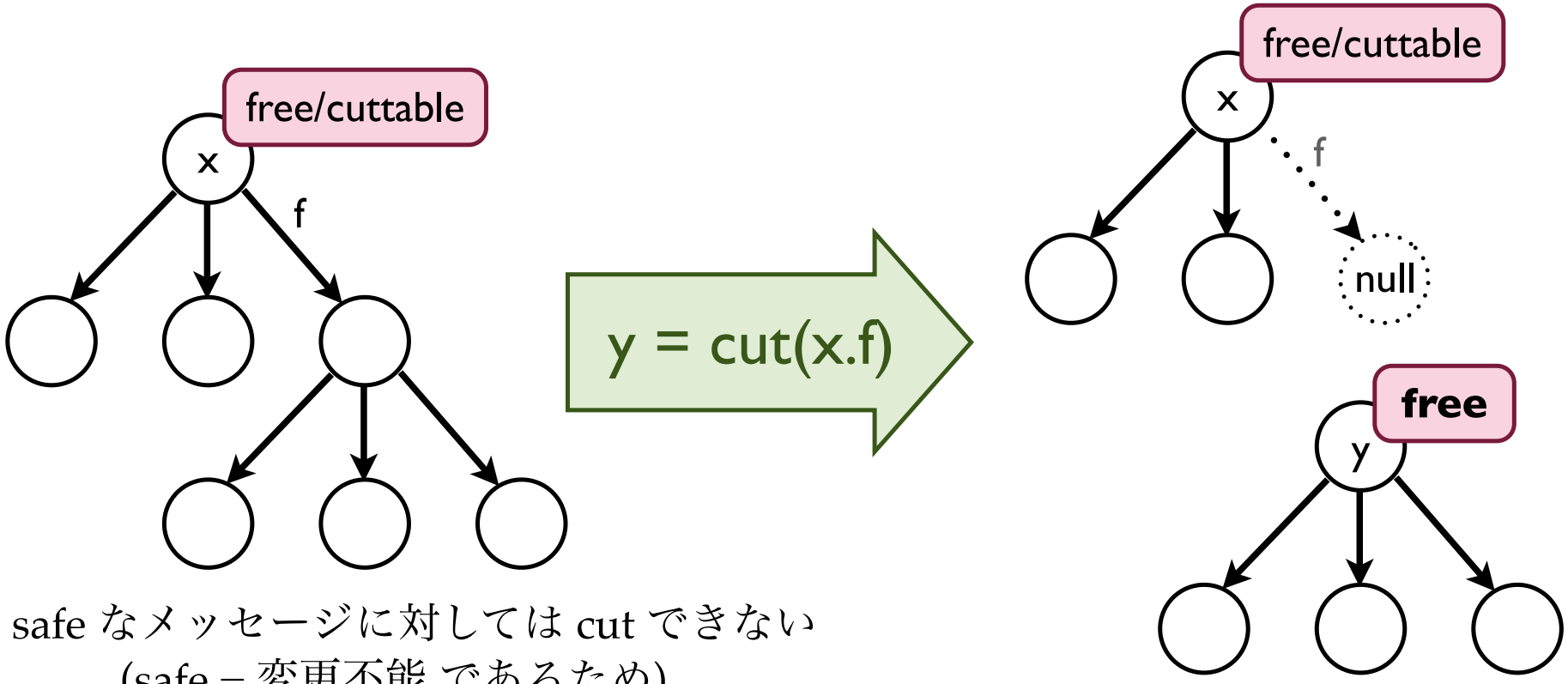
free と cuttable

- フィールドの読み出し



Cut operator

- メッセージのフィールドが参照している
  メッセージを"切り取る"



※ safe なメッセージに対しては cut できない
(safe = 変更不能 であるため)

例: TextServer

MailBox = thread-safe queue
thread 間で共有可能 (**Sharable**)

```
class TextServer extends Actor {
  MailBox mb;
  TextServer(MailBox mb){this.mb = mb;}
  @pausable
  void execute() {
    while (true) {
      TextMsg m = mb.get();
      transform(m);
      reply(m);
    } }
  @pausable
  void reply(@free TextMsg m) {
    m.replymb.put(m);
  }
  void transform(@safe TextMsg m){...}
}
```

1. メッセージを受け取り、
2. 内部状態を変更、
3. メッセージを返す

例: TextServer

コンテキストスイッチし得る
(MailBox.get/put などを呼び出し得る)
関数には **@pausable** をつける

```
class TextServer
  MailBox mb;
  TextServer(MailBox mb){this.mb = mb;}
  @pausable
  void execute() {
    while (true) {
      TextMsg m = mb.get();
      transform(m);
      reply(m);
    } }
  @pausable
  void reply(@free TextMsg m) {
    m.replymb.put(m);
  }
  void transform(@safe TextMsg m){...}
}
```

例: TextServer

```
class TextServer extends Actor {
  MailBox mb;
  TextServer(MailBox mb){this.mb = mb;}
  @pausable
  void execute() {
    while (true) {
      TextMsg m = mb.get
      transform(m);
      reply(m);
    } }
  @pausable
  void reply(@free TextMsg m) {
    m replymb.put(m);
  }
  m(@safe TextMsg m){...}
```

MailBox.put の引数は free でないといけない (安全な messaging のため) ので **@free** をつける

reply 呼び出し後は、 free 引数に渡したメッセージは使用不能になる

例: TextServer

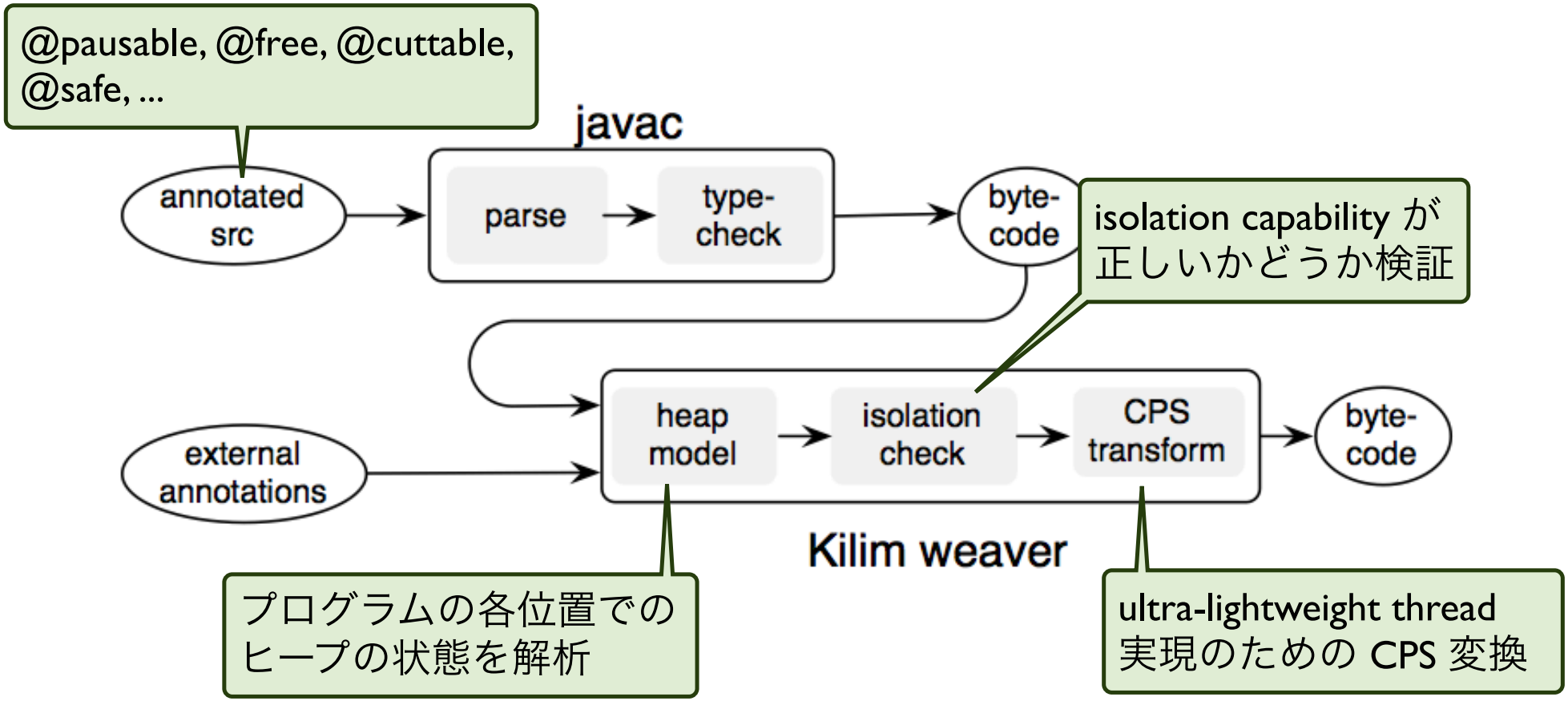
```
class TextServer extends Actor {
  MailBox mb;
  TextServer(MailBox mb){this.mb = mb;}
  @pausable
  void execute() {
    while (true) {
      TextMsg m = mb.get();
      transform(m);
      reply(m);
    } }
  @pausable
                          @free TextMs
                          .put(m);

  void transform(@safe TextMsg m){...}
}
```

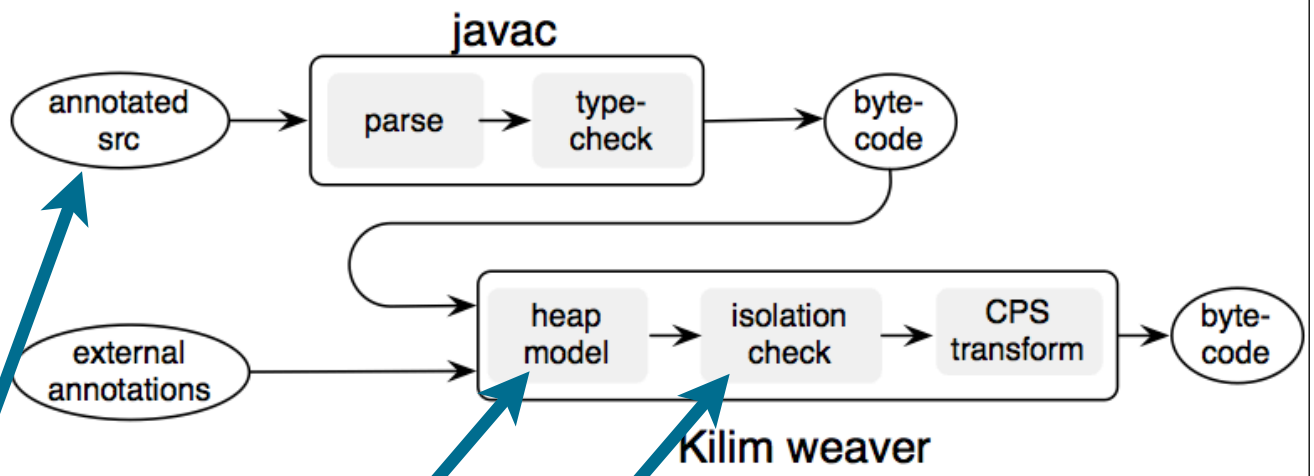
safe 引数に渡したメッセージは
メソッド呼び出し後も使用可能

transform 内で変更する必要が
ないので、変更不能を表す
@safe をつけている

コンパイルプロセス


@pausable, @free, @cuttable, @safe, ...
javac
annotated src
parse
type-check
byte-code
isolation capability が正しいかどうか検証
external annotations
heap model
isolation check
CPS transform
byte-code
Kilim weaver
プログラムの各位置でのヒープの状態を解析
ultra-lightweight thread 実現のための CPS 変換

コンパイルプロセス



1. Kilim の概要
2. Core Language
3. Heap Graph Construction
4. Isolation Capability Checking

Core syntax

qualifier は引数および戻り値に対してのみ記述する

$$
\begin{array}{rl}
\mathit{FuncDcl} ::= & \mathit{free}_{\mathrm{opt}}\ m(\vec{p} : \vec{\alpha})\ \{\ (lb : \mathit{Stmt})^{*;}\ \} \\
\mathit{Stmt} ::= & x := \texttt{new} \quad | \quad x := y \\
& | \ x := y.f \quad | \ x.f := y \quad | \ x := \texttt{cut}(y.f) \\
& | \ x := y[\cdot] \quad | \ x[\cdot] := y \quad | \ x := \texttt{cut}(y[\cdot]) \\
& | \ x := m(\vec{y}) \ | \ \texttt{if/goto}\ \vec{lb} \ | \ \texttt{return}\ x
\end{array}
$$

$x, y, p \in$ variable names　　$f \in$ field names

$lb \in$ label names　　$m \in$ function names

$sel \in$ field names $\cup \{[\cdot]\}$　　$[\cdot]$ pseudo field name for array access

$\alpha, \beta \in$ isolation qualifier $\{\mathit{free},\ \mathit{cuttable},\ \mathit{safe}\}$

null is treated as a special readonly variable

Heap graph construction

- Shape analysis [Wilhelm et al. 00] の一種
- 特徴:
  - Isolation qualifiers
  - Tree-structure
  - Local analysis
  - Cut operator

Heap graph

$G: \langle L, E \rangle$	Heap graph is a pair of local var info L and edges E
$L \in \mathcal{P}(\langle Var, LNode \rangle)$	L = relation between local variable names and *nodes* (*LNode* is logically the nodes of the graph)
$E \in \mathcal{P}(\langle Node, sel, Node \rangle)$	E = a set of Node-Node edges labelled with field names
$LNode \in \mathcal{P}(Var)$	Heap Graph node; in this formalism the name of the node consists of the set of local variable names that may point to it. Well-formedness: $\langle x, N \rangle \in L \Leftrightarrow x \in N$
$Node \in \mathcal{P}(Var) \cup \{\emptyset\}$	Labelled nodes plus summary node.
Convenience: $L(x) \stackrel{def}{=} \{N \mid \langle x, N \rangle \in L\}$	set of *LNodes* to which a local variable might point.

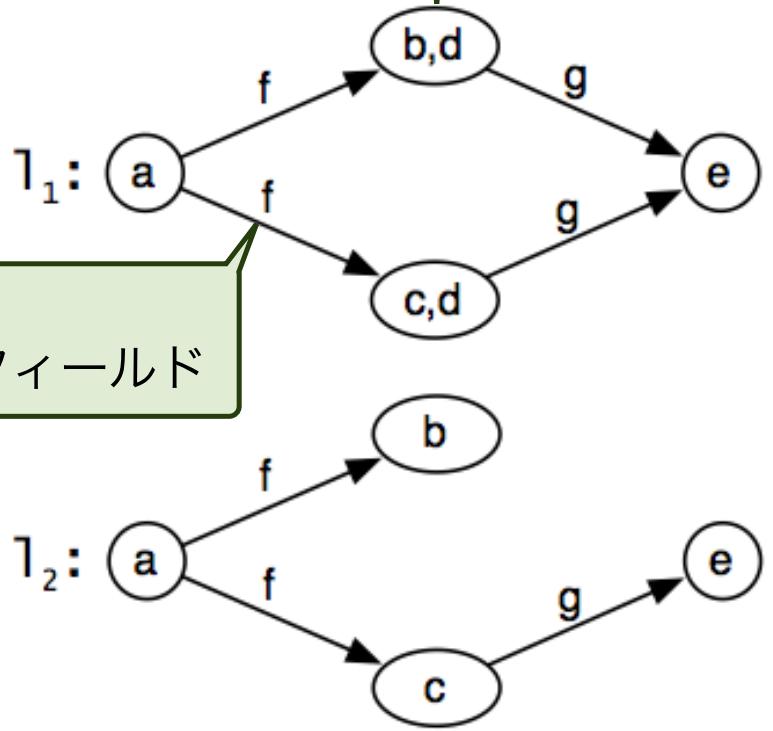
Heap graph

- 変数から参照されている
  オブジェクトにのみ着目

```
    a = new; b = new; c = new
    if ...
      a.f = b
      d = b
    else
      a.f = c
      d = c
    e = d.g
l1:
    d = null
    b.g = null
l2:
```

ノード:
参照している可能性のある変数の集合
= オブジェクト(runtime object の集合)

エッジ:
オブジェクトのフィールド



Heap graph construction

初期状態のヒープグラフ: 空グラフ

$$G^{init}_{out} = \langle \{\,\}, \{\,\} \rangle$$

$$G^{l}_{in} = \bigcup \{G^{l'}_{out} \mid (l', l) \in CFG\}$$

$$G^{l}_{out} = [\![\cdot]\!](G^{l}_{in})$$

ノードの入口におけるヒープグラフ:
predecessors の出口でのヒープグラフの和集合

ノードの出口におけるヒープグラフ:
ノードの入口におけるヒープグラフに対して transfer function を適用

transfer function:
各ステートメントが
ヒープに及ぼす作用を表現した関数

通常の shape analysis との違い:
データ共有を扱わない => シンプル

Transfer functions

$[\![entry(mthd)]\!]\ G$	$\mathbf{L}'' = \bigcup_i \{\ \langle p_i,\ \{p_i\}\rangle\ \}$ where p_i is the i^{th} parameter of $mthd$ $\mathbf{E}'' = \{\}$
$[\![x := \mathbf{new}]\!]\ G$	$G' : \langle L', E'\rangle = kill(G, x)$ $\mathbf{L}'' = L' \cup \langle x,\ \{x\}\rangle, \quad \mathbf{E}'' = E'$
$[\![x := y]\!]\ G$	$G' : \langle L', E'\rangle = kill(G, x)$ $\mathbf{L}'' = \{\ \langle v,\ V_x^y\ \rangle \mid \{v,\ V\} \in L'\}$ $\mathbf{E}'' = \{\ \langle S_x^y,\ sel,\ T_x^y\rangle \mid \langle S,\ sel,\ T\rangle \in E'\}$
$[\![x.f := y]\!]\ G$	$E' = E \setminus \{\ \langle S,\ f,\ *\rangle\ \in E \mid\ x \in S\ \}$ $\mathbf{E}'' = \begin{cases} E' & \text{if } y \equiv null \\ E' \cup \{\ \langle S,\ f\ , T\rangle \mid x \in S \wedge y \in T\} & \text{otherwise} \end{cases}$ $\mathbf{L}'' = L$
$[\![x[\cdot] := y]\!]\ G$	$\mathbf{E}'' = \begin{cases} E & \text{if } y \equiv null \\ E \cup \{\ \langle S,\ \text{`[·]'}\ ,\ T\rangle \mid x \in S \wedge y \in T\} & \text{otherwise} \end{cases}$ $\mathbf{L}'' = L$
$[\![x := y.sel]\!]\ G$	$G' : \langle L', E'\rangle = kill(G, x)$ $\mathbf{L}'' = L'$ $\quad \cup \{\ \langle t,\ T_x\rangle \mid \langle t, T\rangle\ \in\ L'\ \wedge \langle y, S\rangle \in L'\ \wedge\ \langle S, sel, T\rangle \in E'\}$ $\quad \cup \{\ \langle x,\ T_x\rangle \mid \langle y, S\rangle \in L'\ \wedge\ \langle S, sel, T\rangle \in E'\}$ $\mathbf{E}'' = (\ E'\ \setminus\ \bigcup \{\langle y,\ sel,\ *\rangle \in E'\ \})$ $\quad \cup \{\ \langle y, sel, T_x\rangle \mid \langle y, sel, T\rangle \in E'\}$ $\quad \cup \{\ \langle T_x,\ sel,\ U\rangle \mid \langle T, sel, U\rangle \in E'\}$
$[\![x := cut(y.sel)]\!]$	$[\![y.sel := null]\!] \quad \circ \quad [\![x := y.sel]\!]$
$[\![x := m(\vec{v})]\!]\ G$	$G' : \langle L', E'\rangle = kill(G, x)$ $\mathbf{L}'' = L' \cup \{\ \langle x,\ \{x\}\ \rangle\ \}$ $\mathbf{E}'' = E'$

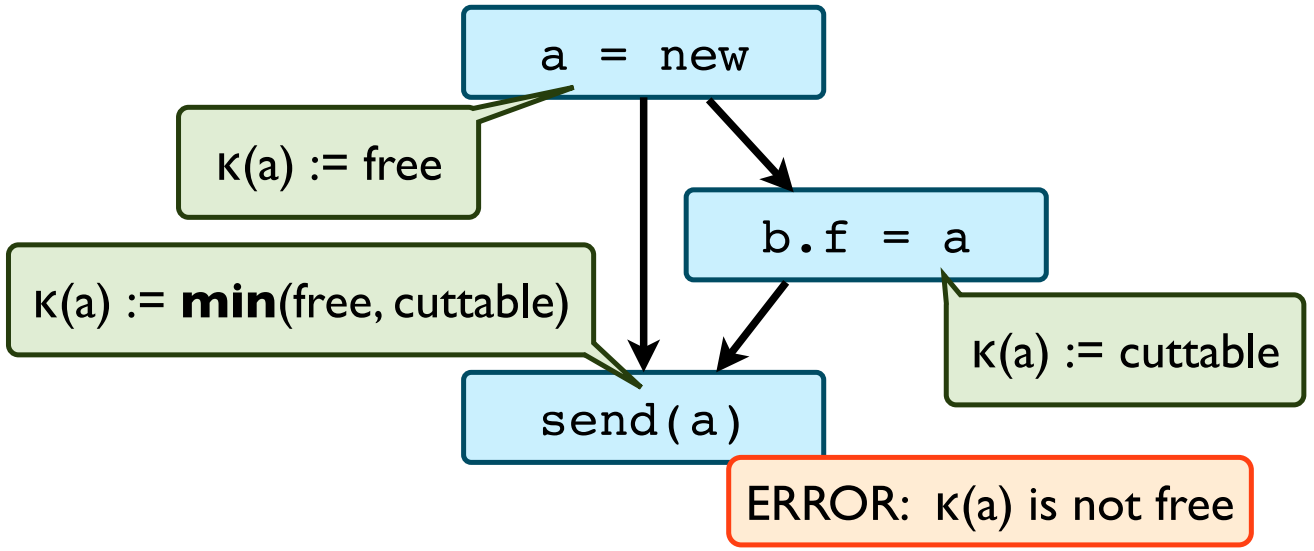
Isolation capability checking

- Capability の推論

  各プログラムポイントにおいて
  ヒープグラフの各ノードとcapability を関連付ける
  (= $\varkappa$: LNode → Capability を求める)

```
a = new
if ...
  b.f = a

send(a)
```



Transfer functions

$[\![entry(mthd)]\!]\ \kappa$	$=$	$[\vec{p} \overset{\kappa}{\mapsto} \vec{\alpha}]$
$[\![x := \mathbf{new}\ T]\!]\ \kappa$	$=$	$\kappa[x \overset{\kappa}{\mapsto} free]$
$[\![x := y]\!]\ \kappa$	$=$	$\kappa[x \overset{\kappa}{\mapsto} \kappa(y)]$
$[\![x.f := y]\!]\ \kappa$	$=$	$precondition : \kappa(y) = free$ $\kappa[y \overset{\kappa}{\mapsto} cuttable]$
$[\![x := y.f]\!]\ \kappa$	$=$	$\kappa[x \overset{\kappa}{\mapsto} s]$ $s = \begin{cases} safe & \text{if } \kappa(y) = safe \\ cuttable & \text{if } \kappa(y) \in \{free, cuttable\} \end{cases}$
$[\![x := m(\vec{y})]\!]\ \kappa$	$=$	$precondition : \beta_i \sqsubseteq \kappa(y_i) \ \wedge\ (\forall i \neq j)(disjoint(y_i, y_j) \vee \beta_i = \beta_j = safe)$ $\kappa \left[\begin{array}{ll} dependants(y_i) \cup \{y_i\} \overset{\kappa}{\mapsto} \bot, & \text{if } (\beta_i = free) \\ dependants(y_i) \overset{\kappa}{\mapsto} \bot, & \text{if } (\beta_i = cuttable) \end{array} \right] \left[x \overset{\kappa}{\mapsto} free \right]$ (assumption: m's signature is $free\ m(\vec{\beta})$. Return value is always $free$)
$[\![x := cut(y.f)]\!]\ \kappa$	$=$	$precondition : \kappa(y) \in \{free, cuttable\}$ $\kappa[x \overset{\kappa}{\mapsto} free]$
$[\![\mathbf{return}\ x]\!]\ \kappa$	$=$	$precondition : \kappa(x) = free \wedge \forall i(\alpha_i = cuttable \implies disjoint(x, p_i))$ κ (no change)

Transfer functions

メソッドのエントリポイント:
仮引数と与えられた capability の対応を追加

$$[\![entry(mthd)]\!]\ \kappa = [\vec{p} \overset{\kappa}{\mapsto} \vec{\alpha}]$$

$$[\![x := \mathbf{new}\ T]\!]\ \kappa = \kappa[x \overset{\kappa}{\mapsto} free]$$

オブジェクト生成:
新たに生成したオブジェクトは free

$$[\![x := y]\!]\ \kappa = \kappa[x \overset{\kappa}{\mapsto} \kappa(y)]$$

$$[\![x.f := y]\!]\ \kappa = \begin{array}{l} precondition : \kappa(y) = free \\ \kappa[y \overset{\kappa}{\mapsto} cuttable] \end{array}$$

フィールドへの代入:
代入できるのは free のみ。
代入後に代入した変数は cuttable になる

Transfer functions

フィールドの読み出し:
オブジェクトが safe なら読んだ値も safe
free/cuttable なら読んだ値は cuttable

$$[\![x := y.f]\!]\ \kappa \quad = \quad \kappa[x \overset{\kappa}{\mapsto} s]$$
$$s = \begin{cases} safe & \text{if } \kappa(y) = safe \\ cuttable & \text{if } \kappa(y) \in \{free, cuttable\} \end{cases}$$

$$precondition : \beta_i \sqsubseteq \kappa(y_i) \ \wedge \ (\forall i \neq j)(disjoint(y_i, y_j) \vee \beta_i = \beta_j = safe)$$
$$[\![x := m(\vec{y})]\!]\ \kappa \quad = \quad \kappa \begin{bmatrix} dependants(y_i) \cup \{y_i\} \overset{\kappa}{\mapsto} \bot, & \text{if } (\beta_i = free) \\ dependants(y_i) \overset{\kappa}{\mapsto} \bot, & \text{if } (\beta_i = cuttable) \end{bmatrix} \left[x \overset{\kappa}{\mapsto} free\right]$$
(assumption: m's signature is $free\ m(\vec{\beta})$. Return value is always $free$)

メソッド呼び出し:
実引数は qualifier に従うもの
かつ safe でないなら領域が重ならない
適用後は free/cuttable 変数は使用不能
戻り値は free

free なら
「オブジェクトが参照しうる
すべてのオブジェクト」を
保持する変数が使用不能になる

Transfer functions

cut operator:
free, cuttable のみ cut 可能
cut した結果は free

$$
\begin{array}{lll}
 & & precondition : \kappa(y) \in \{free, cuttable\} \\
[\![x := cut(y.f)]\!]\ \kappa = & & \kappa[x \overset{\kappa}{\mapsto} free] \\
\hline
 & & precondition : \kappa(x) = free \wedge \forall i(\alpha_i = cuttable \Longrightarrow disjoint(x, p_i)) \\
[\![\texttt{return}\ x]\!]\ \kappa = & & \kappa\ (\text{no change}) \\
\hline
\end{array}
$$

return:
返す変数は free
かつ cuttable な引数がある場合は
その引数が指す領域との共通部分があってはならない

Soundness

※ formal proof は future work

- 基本的には shape analysis から “heap-sharing” を除いたものの正しさに基づく
- 加えて次を保証しなければならない
  - 関数呼び出し後に、 free として渡したデータが使用されないこと
  - 任意の操作がメッセージの木構造を保存し、メッセージの root だけが free になること

プログラムのある点において、free なデータは
1つのメソッドからのみアクセス可能 (= 1つの actor からのみアクセス可能)

まとめ

- Zero copy messaging を実現するための言語と静的ヒープ解析手法を提案
  - メッセージ型
  - メッセージへのアクセス権限 (capability) の付与
  - 静的ヒープ解析による capability 検査