全体ミーティング
(2006/10/17)

M2 佐藤秀明

今回の内容

- Stabilizers: A Modular Checkpointing Abstraction for Concurrent Functional Programs.
  - Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan.
  - ICFP 2006.
- Concurrent ML 上のエラー回復機構
  - Checkpointing 専用の言語プリミティブを導入
  - syntax と semantics を定式化
  - 実行性能を評価

並行プログラムの Checkpointing

- Checkpointing: ある時点の計算機の状態を保存
  - 応用例：エラーが起きる前まで状態を巻き戻して再実行
- マルチスレッドの場合は困難
  - 依存関係にある他スレッドも適切に巻き戻す必要性

Checkpointing 機構を定式化しよう！
（実装も軽いとなおいいなあ）

Stabilizers

- Concurrent ML 上のエラー回復機構
  - Checkpointing 専用の言語プリミティブを導入
  - Stable sections の概念を導入
    - 状態を巻き戻す範囲を表現

言語プリミティブ

- stable: 受け取った関数を巻き戻す最小単位とする
  - 関数実行中の巻き戻しを捕捉
- stabilize: 状態を巻き戻す
- cut: 以前に保存した状態をすべて破棄する

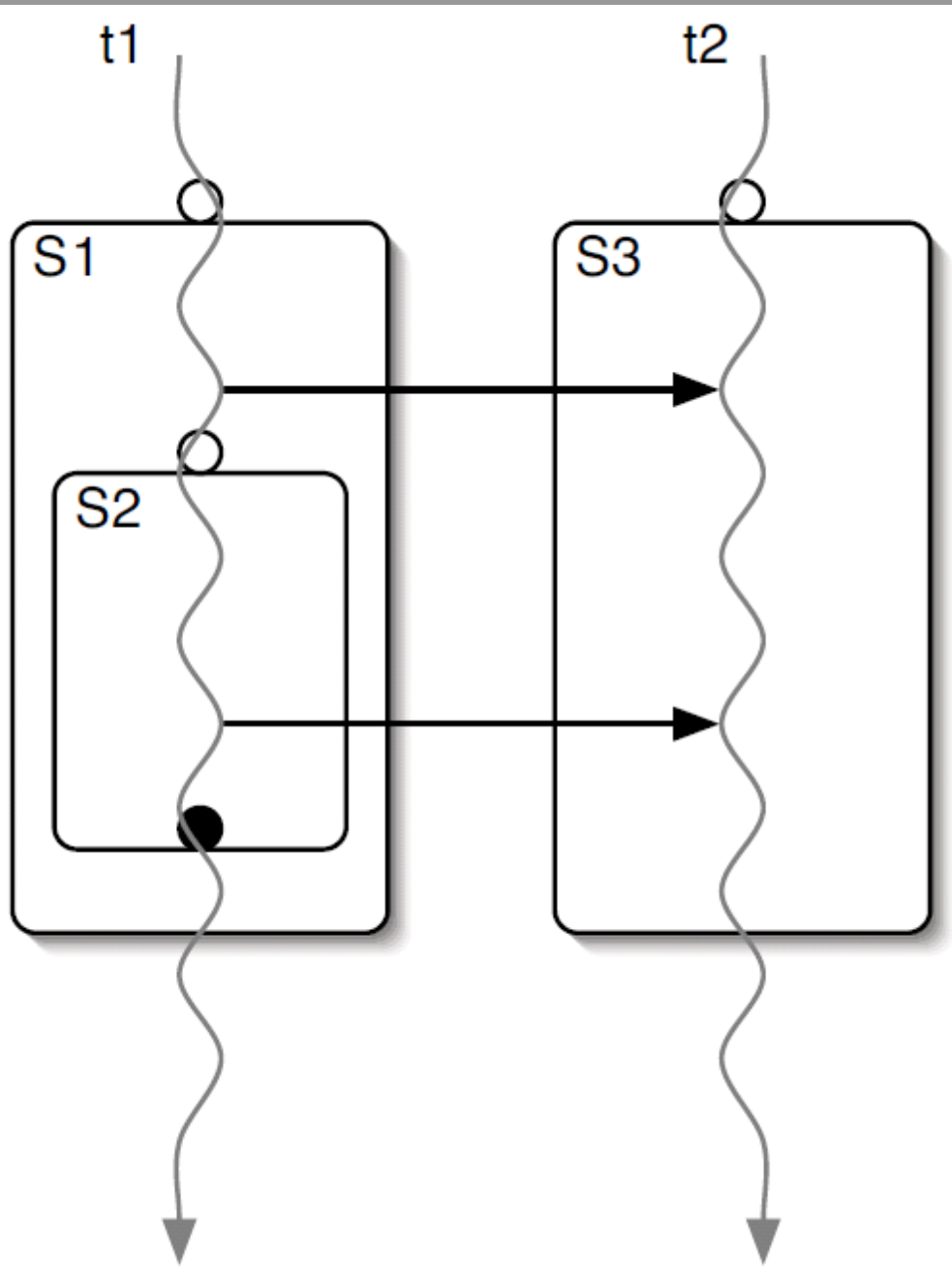
```
stable      : ('a -> 'b)  -> 'a -> 'b
stabilize   : unit -> 'a
cut         : unit -> unit
```

Stable Sections

- 状態を巻き戻す最小単位
  - ネスト可能
    - stable 関数が first-class だから
- 常に最内 section だけ巻き戻されるとは限らない
  - 他スレッドとの依存関係による

状態巻き戻しの例 (1)

- t1 は S1 の最初まで戻される
  - S2→S3→S1 の依存関係

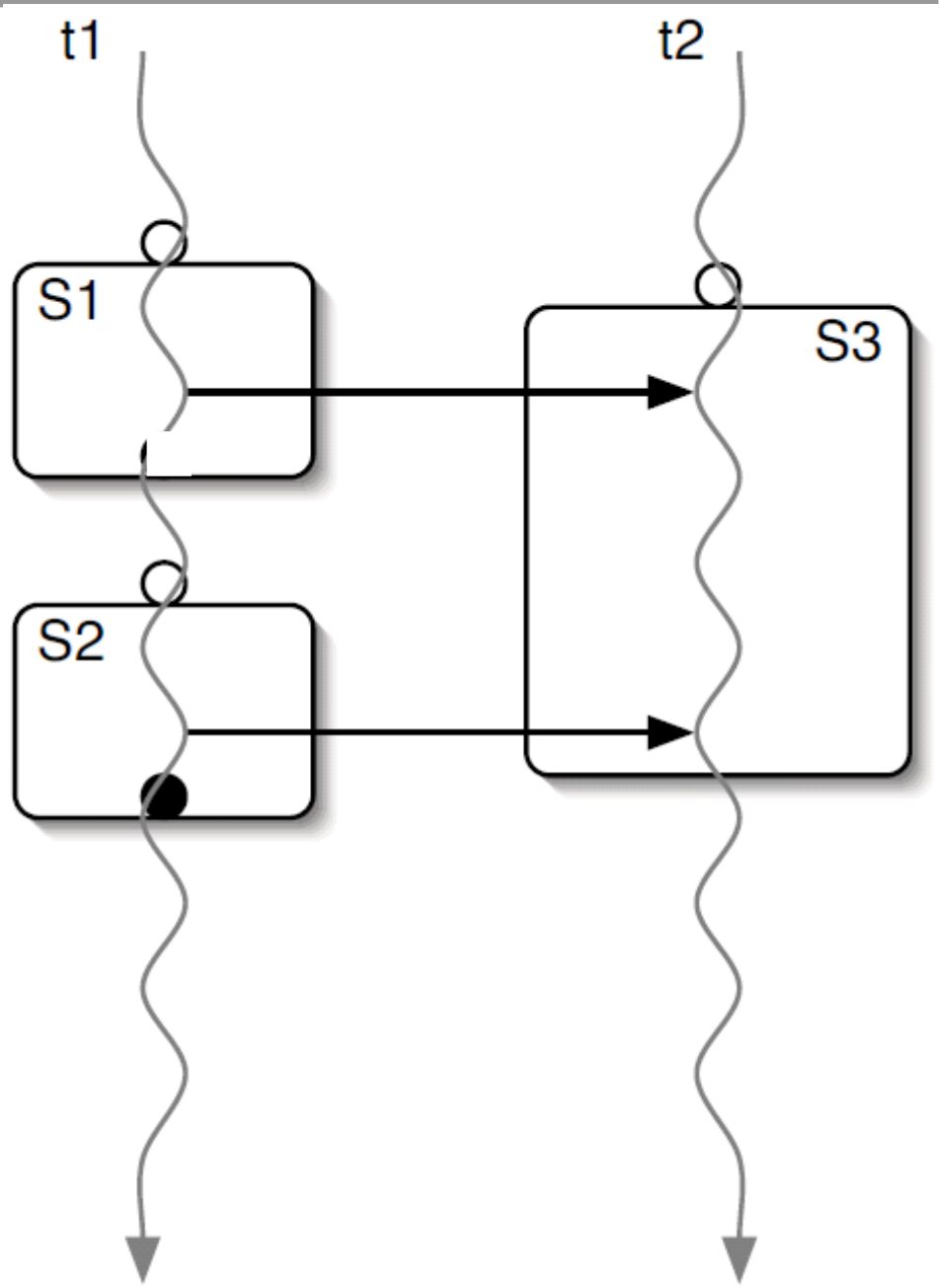


tn: スレッド
Sn: stable section
○: 状態の保存
●: stabilize 呼び出し
→: 通信

状態巻き戻しの例 (2)

- t1 は S1 の最初まで戻される
  - S2→S3→S1 の依存関係

tn: スレッド
Sn: stable section
○: 状態の保存
●: stabilize 呼び出し
→: 通信



Motivating Example: Swerve

- Concurrent ML で書かれた web server
  - 5 つのモジュールからなる
- モジュールごとにスレッドが生成される
  - スレッド間通信で情報をやり取り
- タイムアウト処理の仕様が複雑
  - Modularity の低下

Swerve のスレッド間通信の例

- Figure 2 を参照
  - (1) Listener(L) が Timeout Manager(TM) を生成
  - (2) L が File Processor(FP) を生成、さらに
    タイムアウトを受け取る通信チャンネルを FP に通知
  - (3) FP がタイムアウトをポーリング
  - (4) FP 生成前のタイムアウト通知
  - (5) FP 処理中のタイムアウト通知
  - (6) FP から L へのタイムアウト通知の転送

Stabilizer による Modularity の保護

- 互いの内部仕様に依存した処理を除去可能
  - FP: ポーリング操作を除去 + stable で囲む
    - Figure 3 を参照
  - TM: タイムアウト通知を stabilize に置換 + stable で囲む
    - Figure 4 を参照
  - L: タイムアウト待ち受け処理を除去 + stable で囲む
    - Figure 5 を参照

タイムアウト時に全スレッドを適切に巻き戻せる！

対象言語の構文

- Figure 6 を参照
  - λ 計算 + スレッド間同期通信 + スレッド生成
  - 非同期通信は新しく生成したスレッドに任せる
- プログラム = 式をスレッドの数だけ並べたもの

$$P = t1[e1]_{\delta 1} \;||\; t2[e2]_{\delta 2} \;||\; \ldots \;||\; tn[en]_{\delta n}$$

  - t: スレッドの ID
  - e: 評価する式
  - δ: stable section ID の列
    - stable section のネスト状態を表現

対象言語の評価規則 (1)

- Figure 6 を参照
- プログラムの状態を P, Δ のペアで表現

$$P, \Delta \Rightarrow P', \Delta'$$

  - P: プログラム
  - Δ: checkpoint をとった状態を保持
    - stable section ID から「保存した状態」への写像

対象言語の評価規則 (2)

- 新しいスレッドの spawn

$$t[\mathrm{spawn}(e)]_{\delta}, \Delta \Rightarrow t[\mathrm{unit}]_{\delta} \mid\mid t'[e]_{\varphi}, \Delta$$

  - 新しいスレッドに e を評価させ、自分自身は unit を返す
- send/recv による通信

$$t[\mathrm{send}(l, v)]_{\delta} \mid\mid t'[\mathrm{recv}(l)]_{\delta'}, \Delta \Rightarrow t[\mathrm{unit}]_{\delta} \mid\mid t'[v]_{\delta'}, \Delta$$

  - l: 通信チャンネルの ID
  - 受信側へ値を送る

対象言語の評価規則 (3)

- stable section への進入

$$t[\mathrm{stable}(\lambda x.e)(v)]_{\delta}, \Delta \Rightarrow t[\overline{\mathrm{stable}}(e[v/x])]_{\delta'.\delta}, \Delta[\delta' \rightarrow (\Delta \text{ の中で最も古い state})]$$

  - 新しい stable section ID δ' を導入
  - とりあえず保守的に「Δ の中で最も古い state 」を束縛
  - stable section の評価中であることを $\overline{\mathrm{stable}}$ で表現

- stable section からの退出

$$t[\overline{\mathrm{stable}}(v)]_{\delta'.\delta}, \Delta \Rightarrow t[v]_{\delta}, \Delta - \{\delta'\}$$

  - δ' を削除

対象言語の評価規則 (4)

- stabilize による状態の巻き戻し

$$t[\mathrm{stabilize}()]_{\delta'.\delta},\ \Delta \Rightarrow \Delta(\delta')$$

  - δ' に対応する状態を写像 Δ を用いて取得

- cut による checkpoint の破棄

$$t[\mathrm{cut}()]_{\delta},\ \Delta \Rightarrow t[\mathrm{unit}]_{\varphi},\ \varphi$$

  - stable sections をすべて忘却

対象言語の性質 [Safety]

- stabilize しても結果は変わらない

操作列 α

$P1||t[e]_{\varphi}, \Delta 1 \Rightarrow^{+} P2, \Delta 2$

⇓ stabilize

$P3, \Delta 3 \Rightarrow^{*} P4||t[v], \Delta 4$

操作列 β

ならば、

$P1||t[e]_{\varphi}, \Delta 1$

操作列 α' ⇓
($|\alpha'| \leqq |\alpha|$)

$P3', \Delta 3' \Rightarrow^{*} P4||t[v], \Delta 4$

操作列 β

評価規則の問題

- 利用可能な最も古い状態に巻き戻る
  - 戻りすぎて無駄な再計算を行っている可能性
- プログラム全体で１つの状態を保存
  - 依存関係のないスレッドも一緒に巻き戻ってしまう

↓

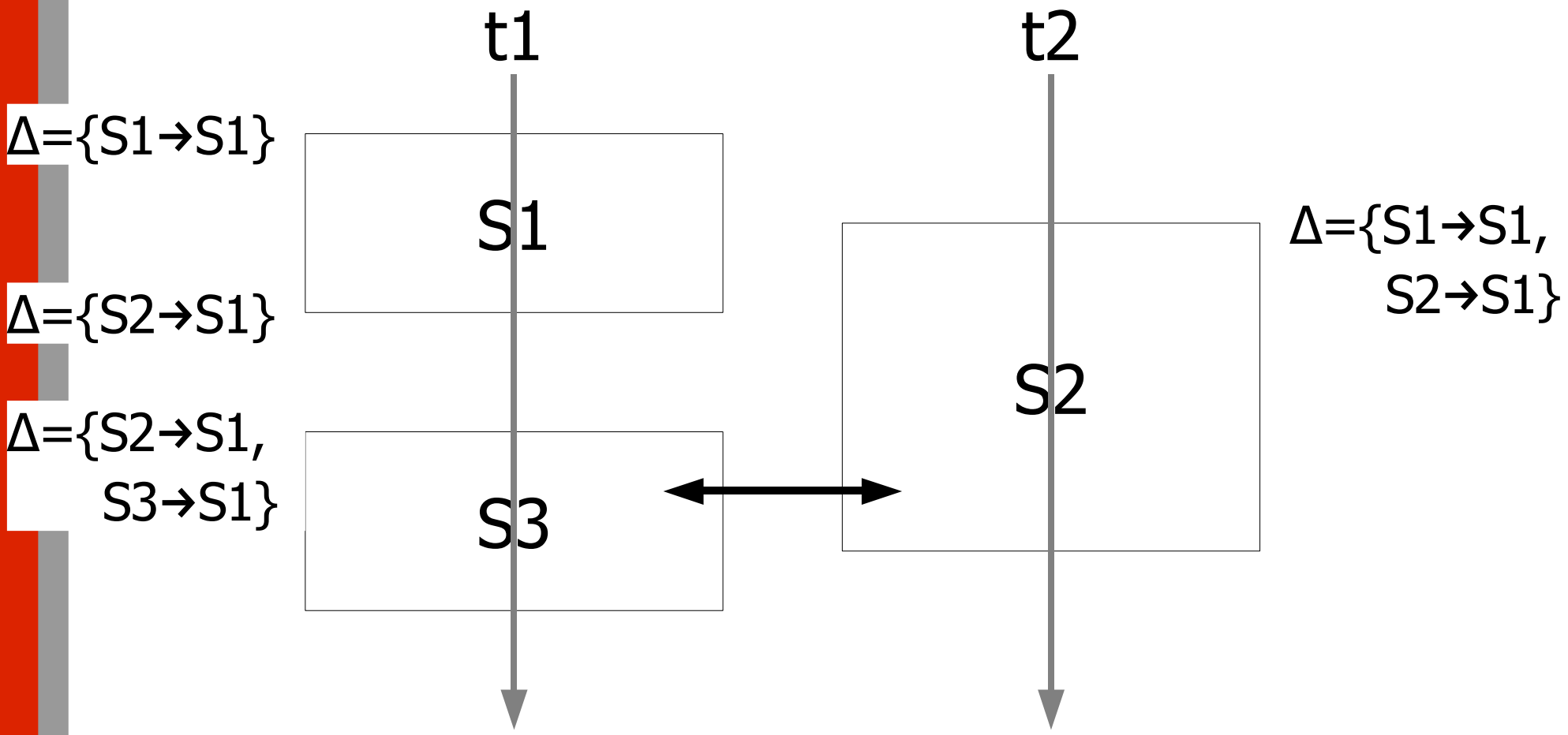
巻き戻しの影響範囲を正確に把握する必要性

依存関係グラフの構築

- ノード：スレッドの各状態
- エッジ：ノード間の依存関係
  - 前状態から現状態へ
  - 現状態から最内 stable section を表すノードへの後退辺
  - スレッドの生成元から生成先へ (spawn)
  - 通信相手のノードと互いに指し合う (send/recv)

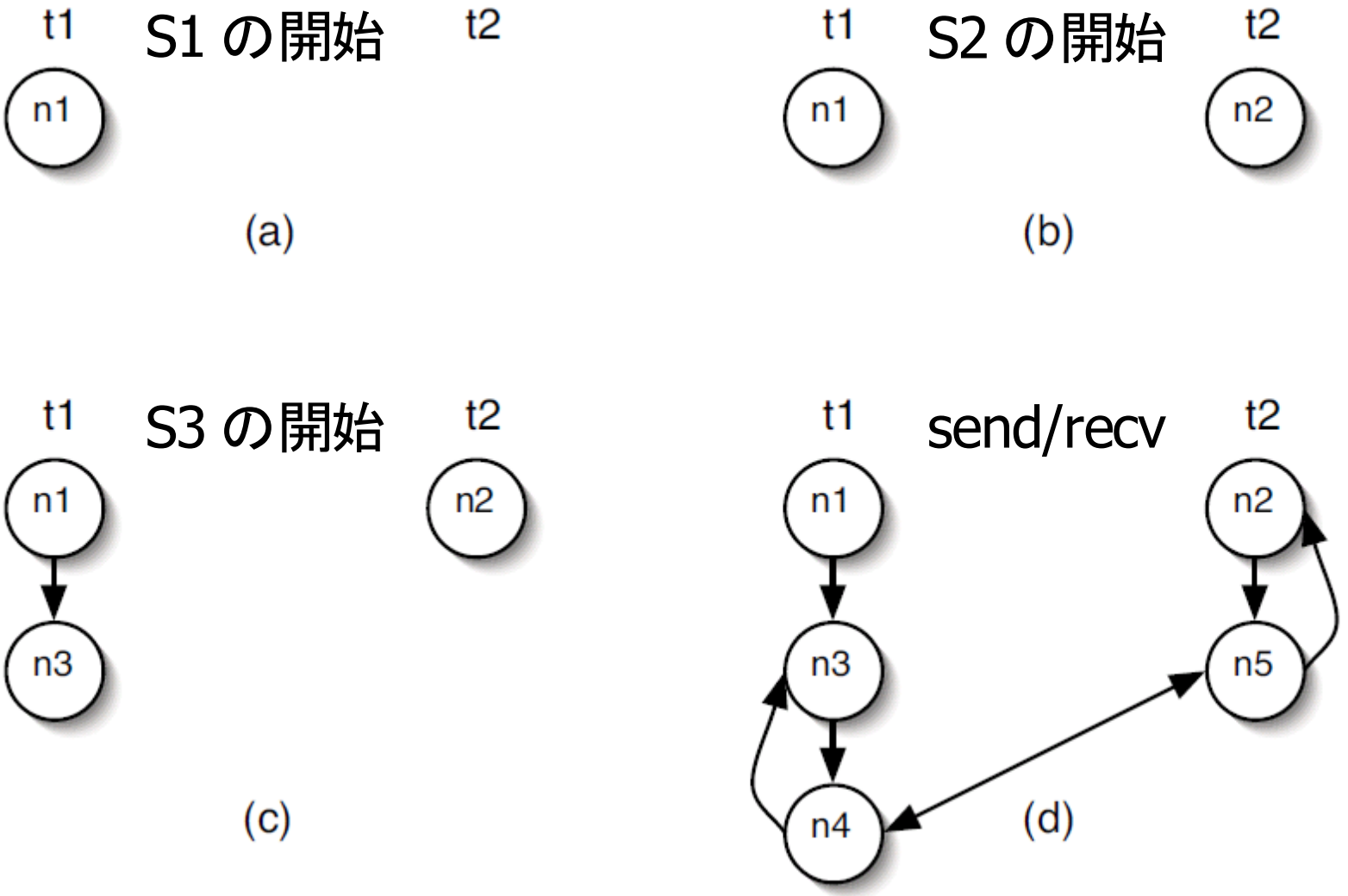
stabilize のノードから到達可能な
最も古いノードまで巻き戻せばよい！

依存関係グラフの例 (1)



S2/S3 での stabilize は
ともに S1 の開始前まで巻き戻ってしまうが…

依存関係グラフの例 (2)



n4/n5 から到達可能な n2/n3 まで戻れば十分

グラフ版評価規則

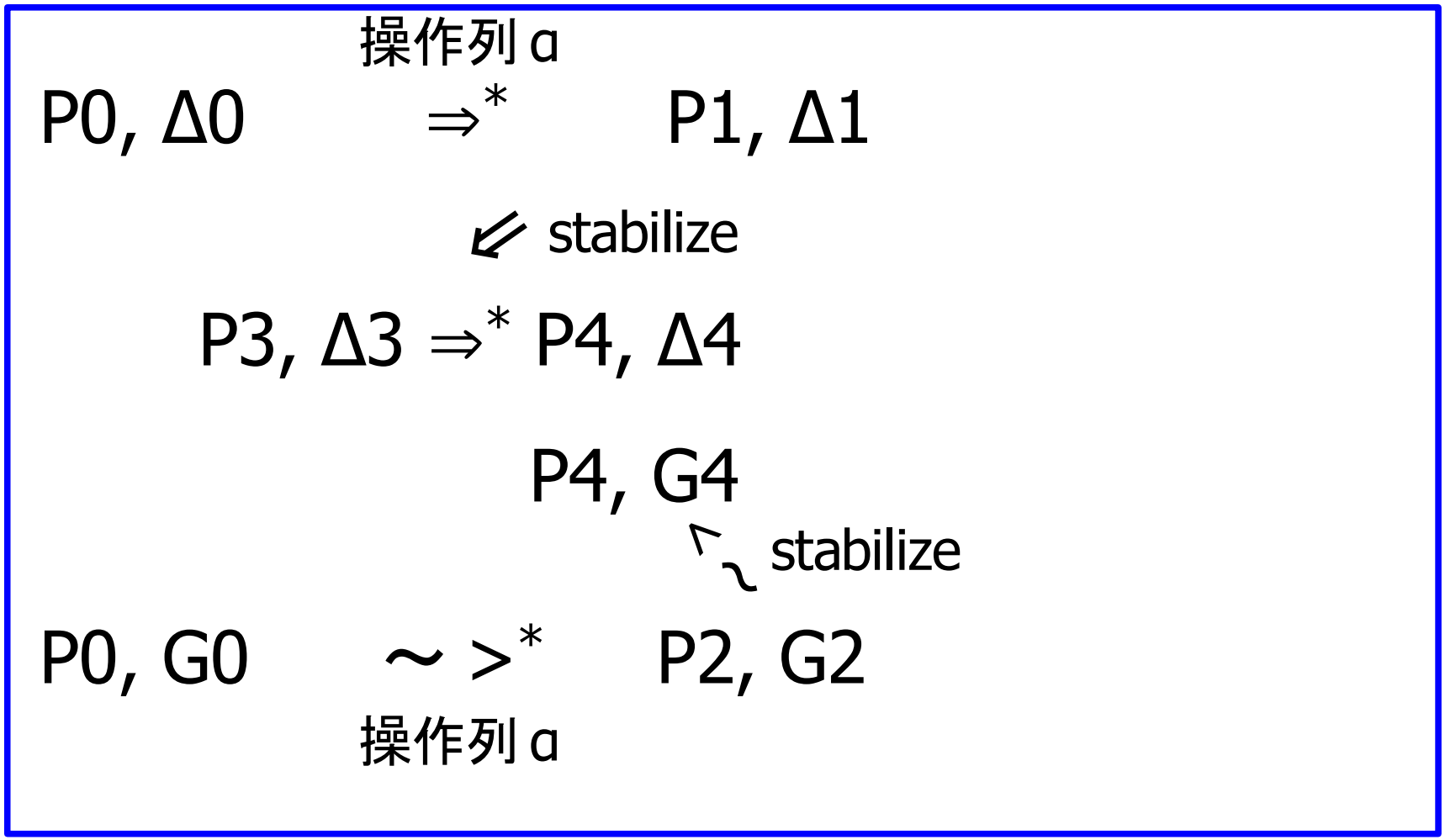
- Figure 9 を参照
  - 詳細は略
- プログラムの状態を P, G のペアで表現

P, G ～> P', G'

  - P: プログラム
  - G: 依存関係グラフ

グラフ版評価規則の性質 [Efficiency]

- グラフ版評価規則のほうが無駄な巻き戻りが少ない

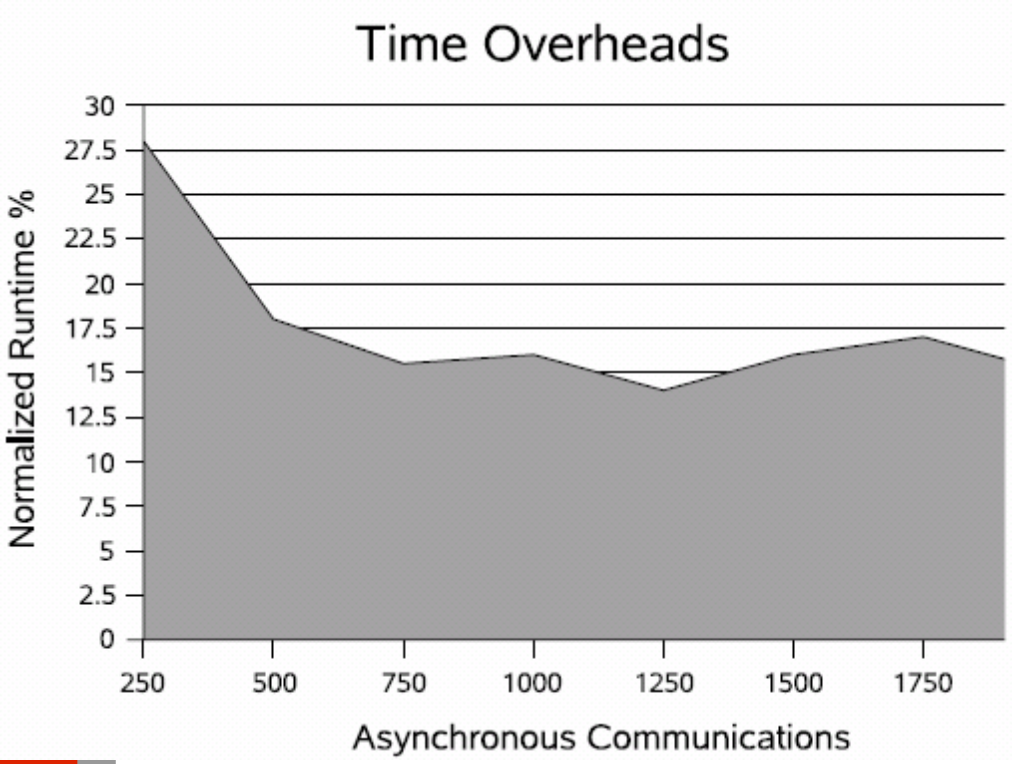


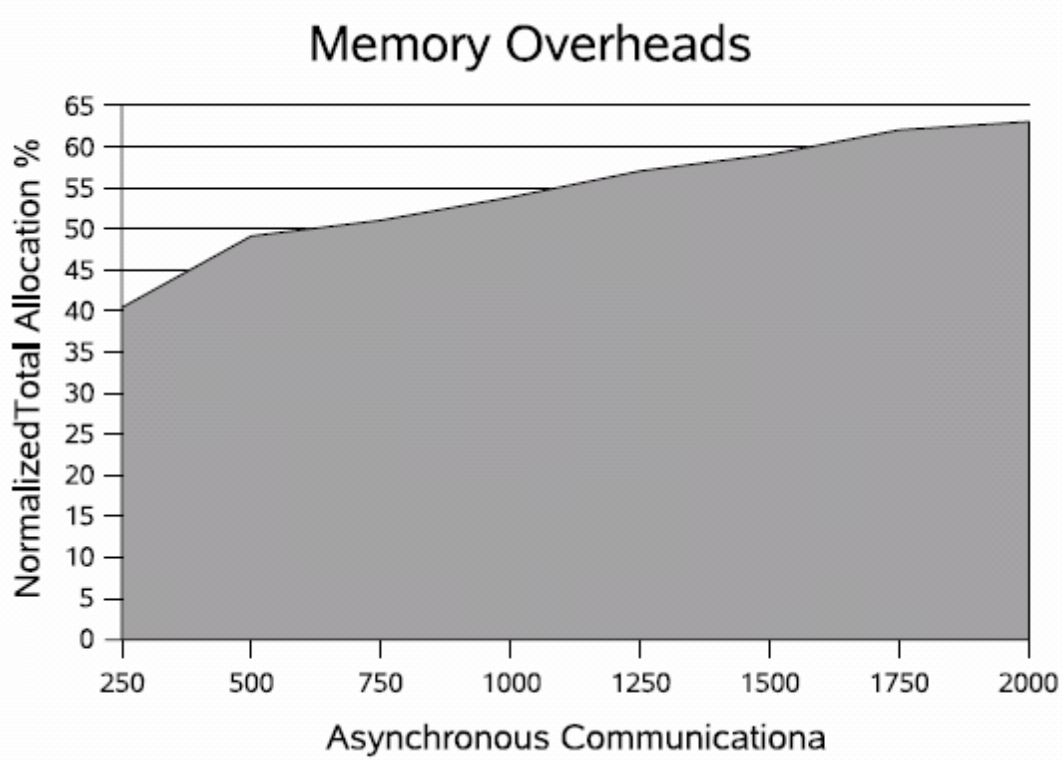
実装

- Shared memory に対する副作用の扱い
  - 最初の write のみ検知して version list に値を backup
  - shared memory に対する lock 取得を監視
- グラフ構造の表現方法
  - ノード / エッジの合併
  - 到達不可能なノードを破棄

実行性能の合成ベンチマーク

- グラフ構成処理によるオーバヘッド増加を評価
- 時間は定数、空間は線形





実在のソフトによる実行性能評価

- 6% ほどの実行時間オーバヘッド増加

Benchmark	LOC incl. eXene	Threads	Channels	Comm. Events
Triangles	16501	205	79	187
N-Body	16326	240	99	224
Pretty	18400	801	340	950
Swerve	9915	10532	231	902

Shared		Graph	Overheads (%)	
Writes	Reads	Size(MB)	Runtime	Memory
88	88	.19	0.59	8.62
224	273	.29	0.81	12.19
602	840	.74	6.23	20.00
9339	80293	5.43	2.85	4.08

Swerve の巻き戻し操作の性能評価

- web リクエスト回数が多くてもほとんど影響なし

Reqs	Graph Size	Channels		Threads Affected	Runtime (seconds)
		Num	Cleared		
20	1130	85	42	470	0.005
40	2193	147	64	928	0.019
60	3231	207	84	1376	0.053
80	4251	256	93	1792	0.094
100	5027	296	95	2194	0.132

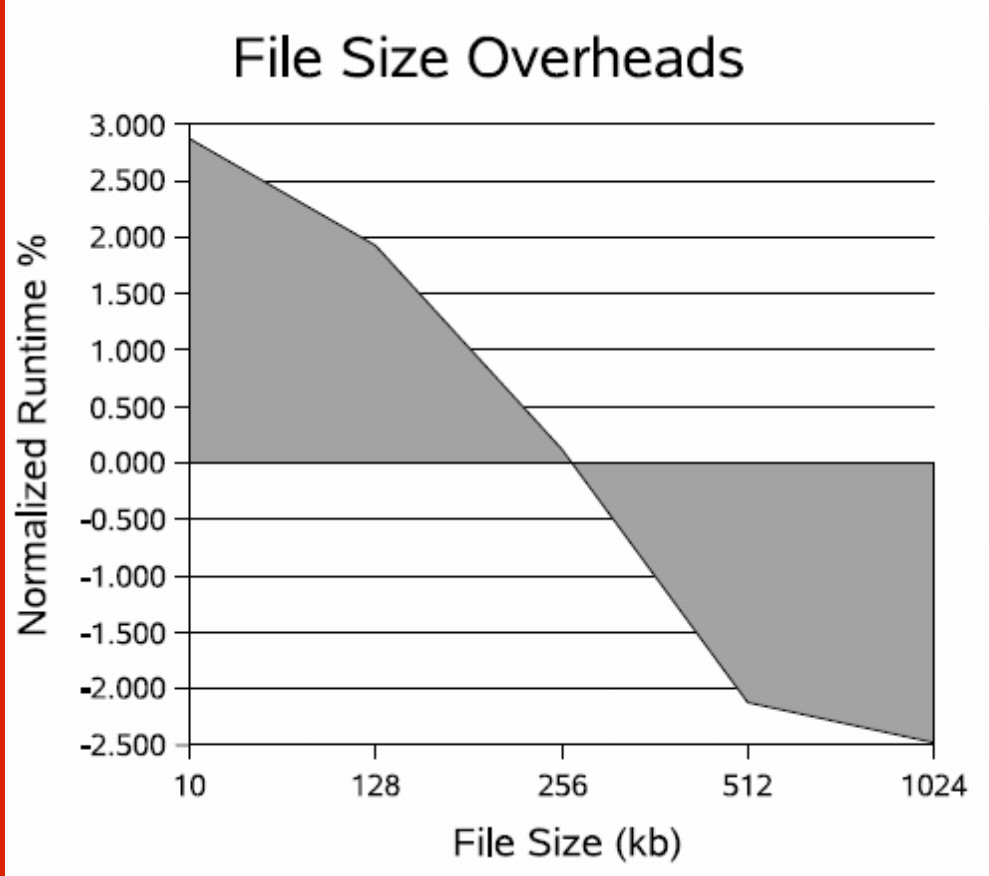
エラー処理の性能評価

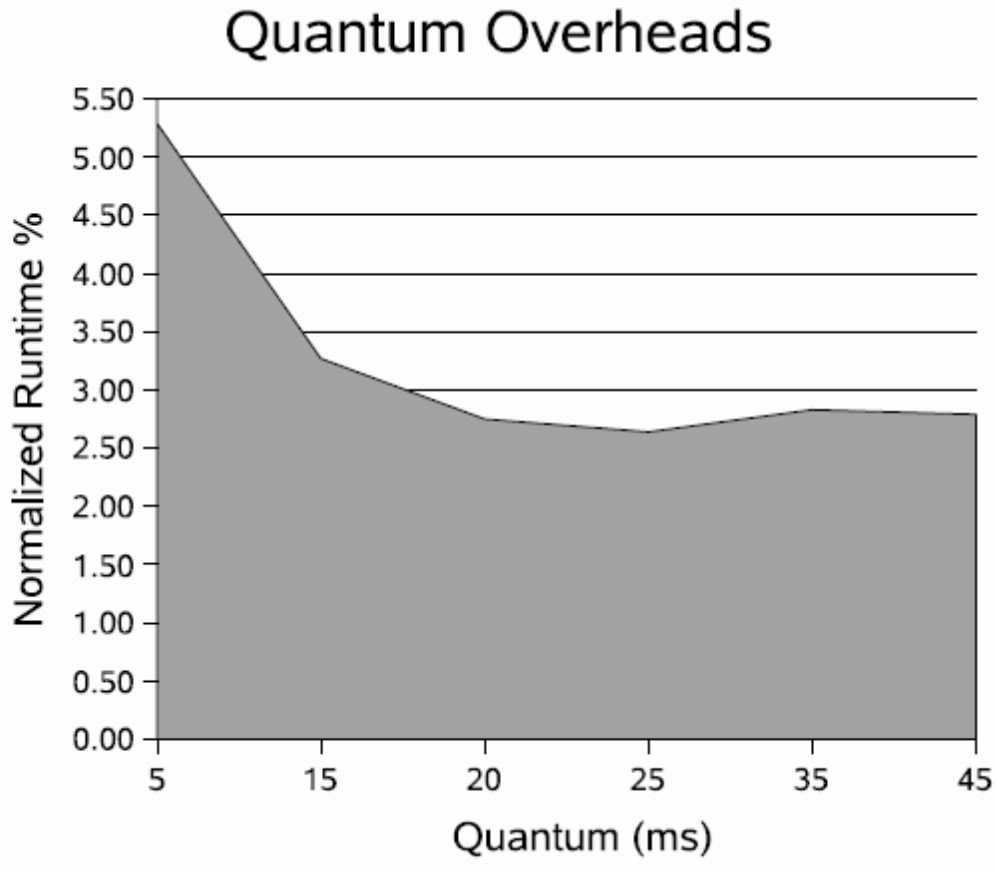
- 10 回に 1 回の割合でタイムアウトさせてみた
  - 巻き戻しの負荷をみてみる
- 実行時間はほぼ不変

Benchmark	Channels		Threads		Runtime
	Num	Cleared	Total	Affected	(seconds)
Swerve	38	4	896	8	.003
eXene	158	27	1023	236	.019

Swerve の詳細な評価

- ファイルサイズが大きいほど速い
  - ポーリングが要らないから
- 1 回の割り当て時間が短いと遅い
  - グラフ処理時間の割合が大きくなる





まとめ

- Concurrent ML をエラー回復支援機能で拡張
  - Checkpointing を行うための言語プリミティブを導入
  - syntax と semantics を定式化し、良い性質を証明
  - 実行性能の悪化はほとんどなし