別紙６

機能と構成」研究領域 領域活動 評価報告書

－平成１５年度終了研究課題－

研究総括 片山 卓也

1. 研究領域の概要

本領域は、"先進情報システムとその構成に向けて"、独創性に富んだ提案を積極的にとりあげようとするものです。具体的には、これからの社会を支える高度な機能をもった情報システムの構築を目指し、そのための構成や構築方法に関して、基本技術や先進応用事例および基礎となる理論の研究を行います。例えば、ソフトウェア、ネットワーク、プロセッサ、分散・実時間・埋め込みシステム、セキュリティ、設計・実装・進化方法論と環境、テスト・検証技術、形式的手法、高信頼性技術、ユーザインタフェースなどの研究を含みます。

2. 研究課題・研究者名

別紙一覧表参照

3. 選考方針

選考の基本的な考え方は下記の通り。

1)選考は、「機能と構成」領域に設けた8名の領域アドバイザーの協力を得て、研究総括が行う。

2)選考方法は、書類選考、面接選考、および総合選考とする。

・書類選考においては、1提案に付き研究総括と2名の領域アドバイザーが査読し、評価する。

・面接選考では、可能な限り多くの提案者から直接説明を受け、質疑応答を行う。

3)個人レベルで行う研究提案であり、萌芽的でかつ独創性に富んでおり、3年間の研究により科学技術への大きな貢献が期待できそうなものを優先する。また、研究テーマや研究者の所属機関が特定のところに集中しないよう配慮する。

4)審査は書類選考結果、面接選考結果および研究実施の条件等を加味し、総合的観点から行う。

4. 選考の経緯

選　考	書類選考	面接選考	採用者
対象者数	40 名	15 名	6 名

5. 研究実施期間

平成12年10月～平成15年9月

6. 領域の活動状況

・領域会議： 7回

・研究報告会：1回

・研究者訪問（研究総括が、研究開始後、全研究者を訪問。その後、研究実施場所の移動時あるいは適宜、研究者を訪問（研究総括および／または技術参事）。

7. 評価の手続き

研究総括が各研究者からの報告および自己評価を基に、アドバイザーの協力を得て行った。また研究終了時に当機構が開催する研究報告会（一般公開）等の参加者の意見を参考とした。

(評価の流れ)

平成15年 9月	研究期間終了
平成15年12月	研究報告会を東京ガーデンパレスにて開催
平成16年 1月	研究報告書および自己評価提出
平成16年 2月	研究総括による評価

8. 評価項目

(1) 外部発表（論文、口頭発表等）、研究を通して得られた新しい知見などの研究成果
(2) 得られた研究成果の科学技術への貢献

9. 研究結果

平成 12 年度は 40 件の提案の中から、ソフトウェア自動生成に関する研究（2 件；小川瑞史、R. Glueck）、ソフトウェアの検証技術の視点（河野真治）、安全で効率的なインターネット用ソフトウェア開発の視点（関口龍郎）からの新しいプログラミング言語の研究、複雑なプログラムの作成・検査・文書化や理解を容易にするための研究（K. Potter）、および超高速ネットワーク時代に対応できる超高速 I/O 指向オペレーティングシステムの研究（河合栄治）を採択した。応募数はやや少な目であったが、採択した提案はいずれも優れたものであった。

3 年間の研究期間を終了し、6 名の研究者全員が当初の期待通りの研究成果を挙げ、我が国の科学技術への発展に大きな貢献ができたものと考える。平成 15 年 12 月 12 日に東京ガーデンパレスにて開催した、広く一般を対象とした研究報告会では、多くの好意的な感想を頂いた。また、このうち4名の研究者が実装したプログラムは既に公開ないしは近々公開を予定しており、ご利用頂きたい。

研究者毎に言えば、小川瑞史研究者は、「効率的で正しいプログラムの自動生成」の研究で、適切な応用領域を切り出して組み合わせ理論を応用するというアプローチにより理論面で大きな成果が得られた。特に、プログラムのフロー解析において関数型プログラムによる仕様記述とプログラム構造表現のための SP 項の概念が有効であることを示した。

河合栄治研究者は、「超高速 I/O 指向オペレーティングシステム」の研究で、ネットワークサーバの I/O 機構に非同期性を導入することにより大幅な性能向上を実現した。この成果は、実際の商用サービスを含む数多くの運用実験で、その有効性が実証されている。

R. Glueck 研究者は、「超計算：ソフトウェア自動生産のための新領域探求」の研究で、関数型プログラムのプログラム変換に関して、多くの科学的に重要な研究結果を得た。特に逆転計算に関しては理論と実験の両面から研究を行い、自動逆転インタプリータなどを構築した。

河野真治研究者は、「理論領域と実用領域を結ぶ新しいプログラミング単位」の研究で、継続を基本

とした新しいプログラミング言語の開発・実装を行い、この言語のためのモデル検査並列検証系の構築を行った。この言語がシステムプログラミングに適していることから、プログラム検証を利用して正確かつ実用的なシステムプログラムを構築する可能性を開いた。

関口龍郎研究者は、「インターコミュニケーション・プログラミング」の研究で、Java 仮想機械や .Net 仮想機械とは質的に異なる全く新しい手法により安全性と実行効率を両立させるプログラミング言語システムを研究し、実装をおこなった。ポインタを許しながら安全性も高く、また、十分に高速で稼働できる実用的なシステムであり、非常に高く評価できるものである。

R. Potter 研究者は、「計算状態パーソナル・スクラップブック」の研究を行い、Lisp 用と Linux 用の計算状態パーソナル・スクラップブック SBDebug および SBUML を実装した。計算状態の保存、復帰、途中状態からの再実行などを行うものであるが、Linux オペレーティングシステム上のプログラム開発を対象にした SBUML は特にそのアイディアの斬新さや適用範囲の広さから大いに期待できるツールである。

10. 評価者

研究総括 片山 卓也 北陸先端科学技術大学院大学情報科学研究科 教授
兼）同附属図書館長

領域アドバイザー氏名（五十音順）

青山 幹雄	南山大学 数理情報学部 教授（平成 13 年 4 月～）
阿草 清滋	名古屋大学 大学院情報科学研究科 教授 (兼) 名古屋大学 情報連携基盤センター長
市川 晴久	日本電信電話(株) NTT 未来ねっと研究所長
岩野 和生	日本アイ・ビー・エム株式会社（～平成 13 年 3 月）
菊野 亨	大阪大学 大学院情報科学研究科 教授
中島 秀之	独立行政法人産業技術総合研究所 サイバーアシスト研究センター長
南谷 崇	東京大学 先端科学技術研究センター長 (兼) 東京大学 大学院情報理工学系研究科 教授
湯淺 太一	京都大学 大学院情報学研究科 教授
米崎 直樹	東京工業大学 大学院情報理工学研究科 教授

参考）

(1) 外部発表件数

	国　内	国　際	計
論 文	12	5	17
口 頭	47	40	87
その他	2	3	5
合 計	61	48	109

(2) 特許出願件数

国 内	国 際	計
0	0	0

(3) 受賞等

・ICFP2002 プログラミングコンテスト優勝（2002 年 10 月）

学位

・河合栄治　奈良先端科学技術大学院大学より工学博士号授与（2001 年 3 月）

・小川瑞史　東京大学より理学博士号授与（2002 年 4 月）

別紙

機能と構成」領域 研究課題名および研究者氏名

研究者氏名 （参加形態）	研 究 課 題 名 （研究実施場所）	現　　職 （応募時所属）	研 究 費 (百万円)
小川 瑞史 （兼任）	効率的で正しいプログラムの自動生成 （北陸先端科学技術大学院大学）	北陸先端科学技術大学院大学 情報科学研究科 特任教授 （NTTコミュニケーション科学基礎研究所 主任研究員）	15
河合 栄治 （兼任）	超高速 I/O 指向オペレーティングシステム （奈良先端科学技術大学院大学）	奈良先端科学技術大学院大学 付属図書館研究開発室 助手 （奈良先端科学技術大学院大学 情報科学研究科 博士後期課程）	40
Robert Glueck （専任）	超計算：ソフトウェア自動生産のための新領域探求 （早稲田大学）	科学技術振興機構 さきがけ研究者 （早稲田大学ソフトウェア生産技術研究所 訪問研究員）	31
河野 真治 （兼任）	理論領域と実用領域を結ぶ新しいプログラミング単位 （琉球大学）	琉球大学工学部 助教授 （同上）	43
関口 龍郎 （専任）	インターコミュニケーション・プログラミング （東京大学）	科学技術振興機構 さきがけ研究者 （東京工業大学大学院情報理工学研究科 リサーチアソシエイト）	32
Richard L. Potter （専任）	計算状態パーソナル・スクラップブック （東京大学）	科学技術振興機構 さきがけ研究者 （科学技術振興機構 さきがけ研究補助者）	20

研究課題別評価

1. 研究課題名：効率的で正しいプログラムの自動生成

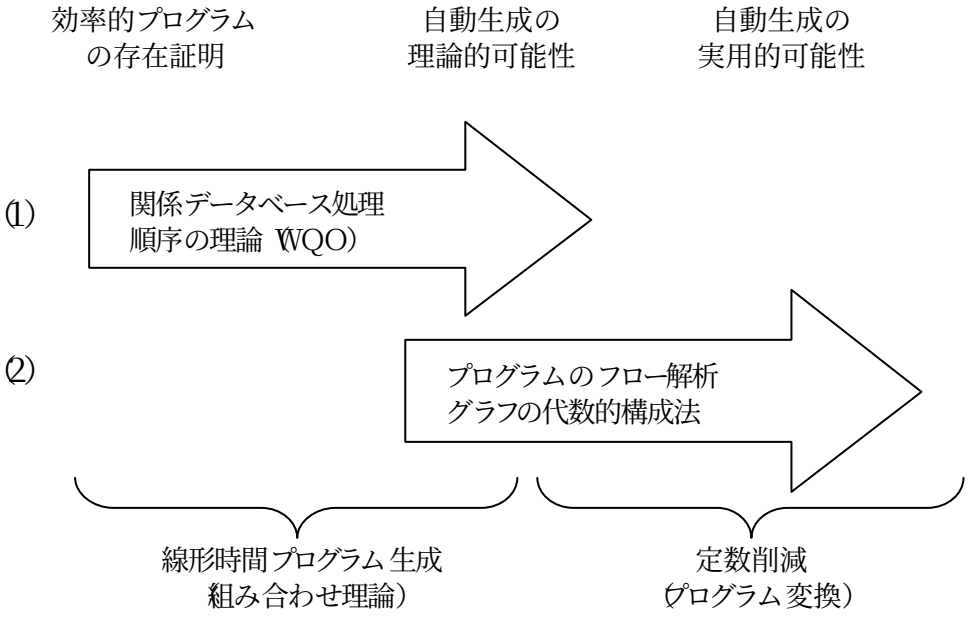
2. 研究者氏名：小川瑞史

3. 研究の狙い：

効率的で正しいプログラムを得るために、計算機が可能なサポートには
(1) 人間の書いたコードの解析に基づく最適化とエラー検出、および
(2) 仕様に基づく自動生成
の二つがある。後者ではプログラムの正しさは仕様と自動生成系が正しければ自動的に保証される。

本研究では、後者の立場をとり、「効率的で正しいプログラムの自動生成」というテーマを設定した。しかし、一般的なプログラムを対象に自動生成を試みれば、構成的証明からのプログラム抽出の研究でも広く知られるように簡単に決定不能性に陥る。したがって、有用な応用領域への適切な制限が必要になる。また、理論計算機科学では論理の枠組みを用いることが多いが、その中にもるべき内容を数学（特に組み合わせ理論）から借りてくることで、対象とする領域では人間のコーディングを凌駕する自動生成を狙いとした。

具体的なアプローチは、応用領域の制限として、
(1) 関係データベースやデータマイニング、
(2) プログラムのフロー解析
を、また組み合わせ理論としてそれぞれ
(1) 順序の理論（Well-Quasi-Order や Kruskal 型定理の構成的証明）
(2) グラフの代数的構成法（グラフの木分割や木幅）
を用いた（図）。これらの組み合わせ理論は、線形時間プログラムの生成、および生成されたプログラムの定数の削減において用いられる。



4. 研究結果：

本研究では、有用な応用領域と計算機の計算可能性の微妙なバランスの上に、上記の図にあげた

(1) 関係データベースにおける順序の理論の応用

(2) グラフの代数的構成法に基づくコントロールフロー解析

の二つの研究課題を設定した。つまり効率的なプログラムの自動生成には、存在証明、自動生成の理論的可能性、自動生成の実用的可能性、の3つの段階がある。効率的アルゴリズムの存在証明はできても実際に生成できない状況は、Kruskal 型の定理を用いた場合にしばしば生じる。これを構成的証明を借りてきて乗り切きろう、というのが第一の課題である。

また自動生成の理論的可能性が示されたとしても、それは実用的可能性をそのまま示している訳ではない。典型的には、自動生成されたプログラムの計算量評価は良くても、実際には定数が爆発して現実的でない状況である。ここでは、実際のコントロールフローグラフは比較的良い構造をもっているという観察を出発点として、グラフの代数的構成法、およびその上の関数型プログラミング＋プログラム変換を用いよう、というのが第二の課題である。

関係データベースにおける順序の理論の応用

当初は課題 1.について進めた。具体的には、関係データベース処理において van der Meyden 現ニューサウスウェールズ大)が 1993 年に提示した未解決問題、

時間概念をもつ不定データベース上の選言 (∨を含む)単項質問処理の線形時間アルゴリズム

を対象とした。この問題は線形時間アルゴリズムの存在は示されていたが、その具体的構成法が知られていなかったものである。

線形時間アルゴリズムの存在証明は、Higman の補題を用いて、充足可能な (∨を含まない)単項質問が多い」という不定データベース間の関係≧が well-quasi-order (WQO) であることから得られる。≧が WQO であることは、任意のデータベースの集合に対し極小元が有限個しかないことを意味しているので、選言単項質問を満たす極小不定データベース $\{D_1, \ldots, D_k\}$ を先に計算しておけば、各 D_i との比較は線形時間でできるので、全体として線形時間で質問処理がなされる。

しかし極小不定データベースが有限個であることがわかっていても、具体的にそれらを決定することは容易ではない。たとえば、単項質問 A<B<C ∨ B<C<A ∨ C<A<B において、その極小不定データベースの集合は以下のようになる。

{A<B<C}, {B<C<A}, {C<A<B}, {A<B, B<C, C<A},

{A<B<A, B<C}, {B<C<B, C<A}, {C<A<C, A<B}, {A<C<A, B<C}, {B<A<B, C<A}, {C<B<C, A<B},

{A<B<A<B, C}, {B<C<B<C, A}, {C<A<C<A, B}, {B<A<B<A, C}, {C<B<C<B, A}, {A<C<A<C, B}.

一番上の行が含まれるのは明らかだが、それ以外についてはパズルを解くような考察を要する。

本研究では、Higman の補題の構成的証明から極小元の計算法を抽出することで、自動的に極小不定データベースの計算、すなわち線形時間質問処理プログラムの自動生成が可能となった。この結果は 2001 年の秋、国際会議 TACS01 で発表し、続いて幸いにも特集号に招待され採録となった。

しかし定数の削減の良いアイデアがみつからなかったこと(fold/unfold 変換は一つの可能性である)、また問題として自然なものがあまり設定できなかったため、中断して課題 2.に力をそそぐことにした。

グラフの代数的構成法に基づくコントロールフロー解析

プログラム解析の自動生成の一般的な手法として、

プログラム解析＝抽象化＋モデル検査

というパラダイムにのっとり、SMVやSPINなどの既存の効率的な実装を適応するアプローチがある。これについては、Java からJimple という3 アドレスコードの中間言語に既存ツールのJ3Cコンパイラとモデル検査系SMVを組み合わせた実装を試みた。既存ツールの活用により、わずか500 行程度の記述でプログラム解析の自動生成系が実際に構成され、素朴な実装であってもかなり実用的な処理時間で実行可能なことを実証した。実際、Java 1.3.1 のjava.math.* クラスの解析を行い、30Kbytes程度のプログラムを数分で解析し、10 数個の不要変数を発見した。しかしプログラムサイズの増大とともに処理時間も急速に増大する傾向も観察され、数万行レベルが限界になりそうな感触がある。

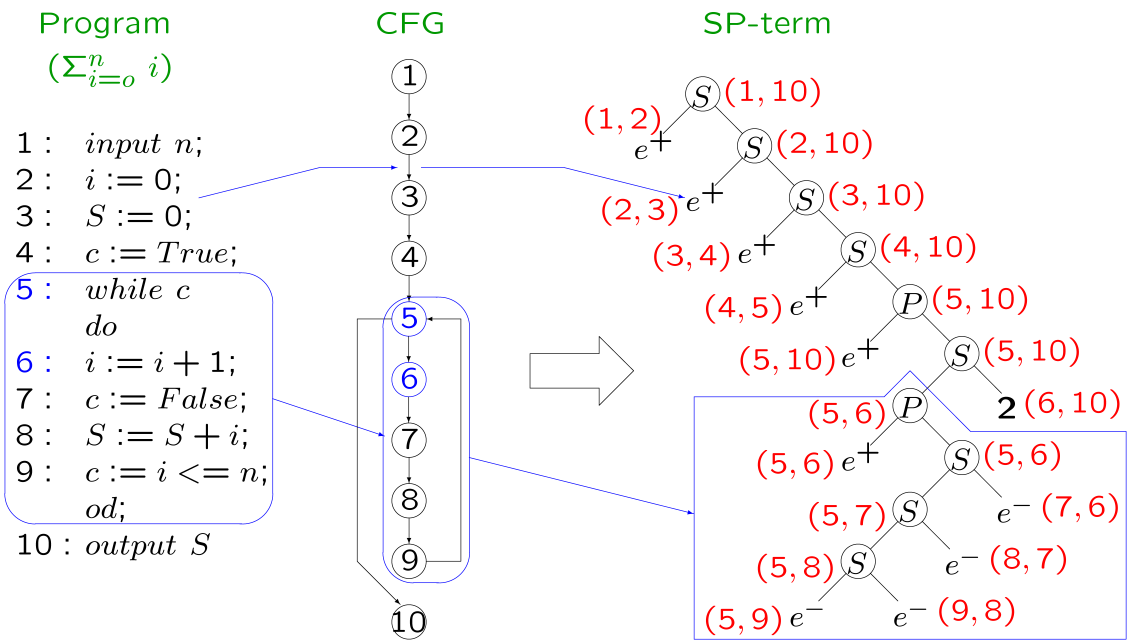
モデル検査は優れた手法であるが、異なるアプローチとして、Thorup (現ATT Labs)が1998年に示した実際のプログラムのコントロールフローグラフは比較的良い構造を持っているという観察（具体的には、GOTO-free C プログラムならば木幅は6以下、Java プログラムではほとんど木幅が3以下などが知られる)に基づき、プログラムのフロー解析のアルゴリズムレベルからの革新を目指した。

もともとグラフの木幅が有界なとき、さまざまな性質の判定が線形時間でできることが知られていた（80年代からのCourcelle、Arnborg や Borie などの研究)。これは論理式による仕様を出発点として、動的プログラミングの手法を適用することで得られている。しかし、論理式を変換する途上で、∀やヨの限量子が現れるごとに指数関数的に状態数（定数係数に対応する)が増加し、線形時間といっても定数が爆発し、理論的な計算量評価にとどまっていた。

本研究では、仕様を論理式のかわりに直接関数型プログラミングで記述し(必要に応じて)融合・組化変換を施すことで劇的な定数の改善が可能となることを示した。関数型プログラムによる仕様記述を可能させるために、多項式型によるグラフ構造の代数的表現が必要となる。木幅の上限 k を制限したグラフの代数的表現として、SP項 SP_k という概念を新たに提案した。SP項は

$$SP_k = e_k(i,j) \mid \mathrm{k} \mid s_k\ SP_k \cdots SP_k \mid p_k\ SP_k\ SP_k$$

として定義される。たとえば、単純なループ構造は k=2 の場合に帰着し、SP項による構成は以下のようになされる。



過去にもいくつかの代数的表現が提案されていたが、SP項の特徴は定数以外の関数記号が s_k（逐次結合）、p_k（並列結合）の二つに限られる点であり、その単純さによりSP項上のプログラミングを容易にしている。具体的なフロー解析の例としては、不要変数解析、ならびに最適レジスタ割当の新しいアルゴリズムを提示し、国際会議 ICFP03において発表した。

5. 自己評価：

当初の理論と実践の溝を埋めたい、という望みは十分にかなえられたとはいえないが、3 年間の研究を経て、ようやくその燭光が見えてきたという状態である。その一方、適切な応用領域を切り出して組み合わせ理論を応用するというアプローチ自体は理論面でかなりの成功であったと考えている。特に、上記の課題 2. において提案したSP項の概念は、試行錯誤の末、最終的に非常に簡潔なものとして提案することができた。このような現実的なグラフの制限を出発点とする研究は、1970 年前後の可約フローグラフ以外にあまり類を見ないものであり、しかもここで提案したSP項は可約フローグラフと独立な概念である。可約フローグラフは深さ優先探索を基本にするのに比べ、SP項ではそれに加え動的プログラミングなどの手法も適用可能とする。そのためより複雑な解析の記述に適している感触を持っている。

SP項に基づくプログラム解析の自動生成は、さきがけの 3 年間の研究期間の最終期に到達したものであり、まだ研究は緒に就いたばかりである。今後、実装も含め、現実的な有効性を実証していきたいと考えている。

6. 研究総括の見解：

正しいプログラムの自動生成はプログラム開発の究極の姿であるが、これまでは生成されたプログラムの実行効率が非常に低いために、特別な領域を除いては一般的には実用化されていない。小川研究者は、この問題にチャレンジし、プログラムのフロー解析を対象に新しい手法「グラフの代数的構成法、その上の関数型プログラミング＋プログラム変換、SP項の導入」という非常に有望な方法を開発した。時間的な制約で研究期間内にこの方法の実用性を示すことは出来なかったが、今後が大いに期待できるものである。

7. 主な論文等：

海外論文誌(4件)

1. Ken Mano, Mizuhito Ogawa. Unique Normal Form Property of Compatible Term Rewriting Systems - A New Proof of Chew's Theorem -, Theoretical Computer Science, 258 (1-2), pp.169-208, 2001.
2. Zurab Khasidashvili, Mizuhito Ogawa, Vincent van Oostrom. Perpetuality and Uniform Normalization in Orthogonal Rewrite Systems. Information and Computation, 164 (1), pp.118-151, 2001.
3. Mizuhito Ogawa. A Linear Time Algorithm for Monadic Querying of Indefinite Data over Linearly
4. Ordered Domains. Information and Computation, 186(2), pp.236-259, 2003, Fourth International Symposium on Theoretical Aspects of Computer Science special issue.
5. Mizuhito Ogawa. Well-Quasi-Orders and Regular ω-languages. Theoretical Computer Science 掲載予定, Third International Colloquium on Words, Languages and Combinatorics special issue.

国内論文誌（4件）

1. 篠埜 功, 胡 振江, 武市 正人, 小川 瑞史. ナップサック問題およびその発展問題の統一的解法, コンピュータソフトウェア 18 (2), pp.59-63, 2001.
2. 篠埜 功, 胡 振江, 武市 正人, 小川 瑞史. 最大重み和問題の線形時間アルゴリズムの導出. コンピュータソフトウェア 18 (5), pp.1-17, 2001.
3. Isao Sasano, Zhenjiang Hu, Masato Takeichi, Mizuhito Ogawa. Derivation of Linear Algorithm for Mining Optimized Gain Association Rules. コンピュータソフトウェア 19 (4), pp.39-44, 2002.
4. 山岡裕司, 胡振江, 武市正人, 小川瑞史. モデル検査技術を利用したプログラム解析器の生成ツール. 情報処理学会論文誌:プログラミング 44 (SIG13/PRO18), pp.25-37, 2003.

査読付国際会議（5件）

1. Zurab Khasidashvili, Mizuhito Ogawa, Vincent van Oostrom. Uniform Normalization beyond Orthogonality. Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA01), Lecture Notes in Computer Science 2051, pp.122-136, May 2001, Springer-Verlag.
2. Mizuhito Ogawa. Generation of a Linear Time Query Processing Algorithm Based on Well-Quasi-Orders. Proceedings of the Fourth International Symposium on Theoretical Aspects
3. of Computer Software (TACS2001), Lecture Notes in Computer Science 2215, pp.283-297, October 2001, Springer-Verlag.
4. Mizuhito Ogawa. Call-by-Need Reductions for Membership Conditional Term Rewriting Systems.
5. The 3rd International Workshop on Rewriting Strategies in Rewriting and Programming (WRS03), June 2003, Electronic Notes in Theoretical Computer Science 86(4), Elsevier (http://www.elsevier.nl/gej-ng/31/29/23/135/49/show/Products/notes/index.htt).
6. Mizuhito Ogawa, Zhenjiang Hu, Isao Sasano. Iterative-Free Program Analysis, Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP03), pp.111-123, August 2003, ACM Press.
7. Mizuhito Ogawa. Complete Axiomatization for an Algebraic Construction of Graphs. The Seventh International Symposium on Functional and Logic Programming (FLOPS04) 採録 (Lecture Notes in Computer Science, Springer-Verlag 掲載予定)

学位論文

1. 小川瑞史. 関数型プログラムの自動解析・検証・生成. 東京大学大学理学系研究科（情報科学）, 2002.4.8 授与.

研究課題別評価

1. 研究課題名：超高速 I/O 指向オペレーティングシステム

2. 研究者氏名：河合　栄治

3. 研究の狙い

インターネットにおける従来の情報配信サービスでは、ボトルネックは低速で高価なネットワークにあるとされ、そのような状況を改善するために配信の最適化技術が数多く開発されてきた。その代表として挙げられるのがキャッシュ技術であり、現在広く普及しつつある CDN (Contents Delivery Network) サービスもその一例である。

しかしながら、ネットワーク技術の発展は当初の予想を遥かに越えて進んでおり、バックボーンネットワークにおける性能（スループット）は 6 ヶ月に 2 倍向上しているという報告があるほどである。そのため、これまでのネットワークサービスインフラストラクチャの構造に多くの歪みが発生してきている。例えば、先に挙げた CDN などは、当初の目的であった分散化によるネットワーク負荷の軽減という意義が薄れる一方で、一時的なリクエスト集中によるシステムダウンの回避や、経路切断など故障からのサービスの保護、さらには DoS 攻撃などからのサービスの防御など、目的の多様化が進みつつある。

本研究では、そうした「歪み」が最も顕著に現れる場所としてエンドノードすなわちサーバに着目し、超高速ネットワークを支えるサーバのシステム技術に焦点を当てて研究を行った。特に、現在のオペレーティングシステムアーキテクチャがその I/O 処理機構において、同期的、すなわちイベント駆動的な構造を用いているために高いオーバヘッドが生じている点に着目し、非同期性を導入することによりこれからの超高速ネットワークに対応したサーバプラットホームを創出することを目的とした。

4. 研究結果

本研究では、高負荷のかかるネットワークサーバにおいて、クライアントからの多数のコネクションを管理する多重化 I/O と呼ばれる機構の高速化を中心に、開発を行った。主要な成果は下記の 2 点である。

(1)　多重化 I/O の実行間隔制御

高速ネットワークサーバでは、数千から数万ものソケットを同時に扱うことができなければならない。特に現在代表的なネットワークサービスの一つである Web においては、HTTP/1.1 永続コネクションが導入されたため、サーバにおける同時ソケット数が増加する傾向にある。

Unix 上のサーバプログラムなどで、このような同時ソケットにおける I/O 処理を多重化するのによく用いられる select() や poll() 多重化 I/O）には、サーバ負荷の増加に対する性能のスケーラビリティに欠けるという問題がある。この問題の原因は、select() や poll() におけるソケットテーブルの走査の処理コストが大きいことにあると広く認識されており、それゆえこれまでに提案されてきた解決手法は、こうしたソケットテーブルの走査を廃止し、特別なイベント通知機構を設けるものが多い。しかし、これらの手法は、オペレーティングシステムの改造が必要であったり、プログラミングモデルの変更を要したりするため、導入コストが高いという別の問題がある。多重化 I/O において真に問題なのは、ソケットテーブルの走査そのものではなく、多重化 I/O がそのイベント駆動的な処理構造により必要以上に頻繁に呼び出されてしまうことにある。

そこで本研究では、多重化 I/O の呼び出し間隔を制御し、サーバの性能を向上させる手法を提案した。本手法により、高頻度の多重化 I/O 呼び出しによって引き起こされていた CPU 処理能力の枯渇が防止され、サーバの処理能力が向上する。また、本手法は従来の select() や poll() を用いたプロ

グラミングモデルを踏襲するため、適用コストが非常に小さいという特長も併せ持つ。

⑵　実時間スケジューリングによる実行間隔制御における確定的なプロセッサ利用の実現

本研究で開発した多重化 I/O における実行間隔制御機構において、同時ソケット数が非常に大きい場合、サービス遅延時間の低減などの効果は確認できるものの、いくつかの特異な現象が二点見られた。

一つは、サービス遅延時間の増加傾向である。実行間隔制御を行う場合、一定間隔でソケットのチェックが行われるため、リクエストレートの増加に対してサービス遅延時間はほぼ一定で推移すると考えるのが妥当である。しかし、実験では同時ソケット数が大きい場合は、提案方式を組み込まない場合と比較して大幅にサービス遅延時間の低減を実現しているものの、サービス遅延時間に増加の傾向が観測された。

もう一つは、プロセッサの利用率がほぼ 100%となってしまう点である。実行間隔制御では、リクエストレートに応じてスレッドをブロックするため、プロセッサの利用率はリクエストレートの増加に対して線形に増加すると考えるのが妥当である。

これらの点を考察した結果、制御を行う間隔がオペレーティングシステムのスケジューリングにおけるタイムスライスを越えてしまい、予期しないコンテキストスイッチ等が含まれてしまうことが判明した。そこで本研究では、実時間スケジューリングを用いることでこれらのコンテキストスイッチを防止する手法を提案した。本方式により、ソケット数が非常に多い場合でも、サービス遅延におけるスケーラビリティが向上した。また、プロセッサ利用率もリクエストレートに対して線形に推移するようになった。後者の利点は、特に近年プロセッサの消費電力が増加しているため、データセンターなどにおける大規模 PC サーバクラスタ等で大きな利点になると考える。

5. 自己評価

3年間のさきがけ研究を振り返って、ネットワークサーバの I/O 機構に非同期性を導入によることによる性能向上という目標は達成されたと考えている。さらに私が提案した手法はサーバソフトウエアおよびオペレーティングシステムへの変更が少なくて済み、導入コストが小さく現実的であるという利点も持っている。また、サーバアーキテクチャの検討という基礎研究的な側面を持ちつつ、実際の商用サービスを含む数多くの運用現場での実証実験を通じ、使える技術の開発ができたことは、本研究の大きな特長である。

一方で、当初掲げていた目標の中で達成できなかったものに、マルチスレッド環境における I/O 処理量に着目したスケジューリングによるサーバ性能向上手法がある。本目標については、実際にスケジューラを構築し、性能評価を行ったが、最終性能的に 5%程度の向上しか観測されなかった。商用サービスに見られる大規模アプリケーションサーバのように、多種多様なスレッドが多数動作する環境で評価ができれば、大きな効果が見られたのではないかと予想している。その意味では、近年の大規模化したサービスプラットホームを対象とした研究において、実験環境構築の難しさを痛感した。

本研究では、その成果物として、Chamomile と名付けた Web Accelerator を開発した。本ソフトウェアは ISP や通信機器ベンダー等からも問い合わせを受けた。今後も開発を進め、ドキュメント等を整備し、フリーソフトウエアとしての公開を目指していきたいと考えている。

ネットワークの高速化の流れは現在も加速中であり、今後はホスト内におけるハードウエア的な分散処理を視野に入れた基盤の構築が必要となると考えている。特にプロセッサ技術では従来の SMP

技術に加え、SMT 技術が広く利用可能になってきている。また、ネットワークプロセッサのような ネットワークインタフェースに近い場所に処理能力を確保する技術も登場してきている。そのため、これらの多様な処理要素の容易かつ効率的な利用を可能にするような機構を構築していく必要があるだろう。これらの技術と、現在盛んに開発が進められているサーバクラスタリング技術を融合し、統一的な真の分散サービスプラットホームを構築していきたい。

6. 研究総括の見解

超高速ネットワークを利用したインターネットによる情報配信サービスが社会インフラとなっている現在、その効率化が大きな問題となっている。河合研究者は、サーバに搭載されているオペレーティングシステムの I/O 処理機構が同期的であることにより高いオーバヘッドが生じている点に着目し、非同期性の導入によりこの問題を解決する方法に挑戦した。その結果、CPU 処理能力の枯渇防止のための多重化 I/O 実行間隔制御、実時間スケジューリングによる実行間隔制御における確定的なプロセッサ利用という、2つの技術を開発することにより目標を達成することが出来た。また、具体的な成果物として、Chamomile と名付けた Web Accelerator を開発したが、これは ISP や通信機器ベンダー等からも高い評価を受けており、河合研究者は目標とした研究課題を現実的な形でも解決した。

7. 主な論文等

招待講演

1.高速ネットワークサービスを実現するオペレーティングシステム, NETWORLD + INTERNET Tokyo 2001, 2001 年 6 月.

2.オペレーティングシステムからみた高速ネットワークサービスの実現, NETWORLD + INTERNET (N+I) Tokyo 2003, 2003 年 6 月

論文

1.Eiji Kawai, Youki Kadobayashi, and Suguru Yamaguchi. Alleviation of Processor Usage on Heavily-Loaded Network Servers with POSIX Real-time Scheduling Control. (Submitted to IEICE Transactions)

2.河合 栄治, 門林 雄基, 山口 英. ネットワークサーバにおける多重化 I/O の実行間隔制御による性能向上手法. 情報処理学会論文誌, 情報処理学会, Vol.45, No.2, 2004 年 2 月.

3.河合 栄治, 門林 雄基, 山口 英. ネットワークプロセッサ技術の研究開発動向. 情報処理学会論文誌 コンピューティングシステム, 情報処理学会, Vol.45, No. SIG1 (ACS4), 2004 年 1 月.

4.河合 栄治, 白波瀬 章, 塚田 清志, 山口 英. 商用 WWW サービスの IPv6 への現実的な移行手法. 情報処理学会論文誌, 情報処理学会, Vol.44, No.3, 2003 年 3 月.

5.西馬 一郎, 河合 栄治, 知念 賢一, 山口 英, 山本 平一. 通知によるコンテンツ一斉公開機構を用いた WWW クラスタシステム. 情報処理学会論文誌, 情報処理学会, Vol.43, No.11, pp.3439-3447, 2002 年 11 月.

口頭発表（国際会議）

1.Eiji Kawai, Youki Kadobayashi, and Suguru Yamaguchi. Improving Scalability of Processor Utilization on Heavily-Loaded Servers with Real-Time Scheduling. International Conference on Parallel and

Distributed Computing and Networks (PDCN 2004), Innsbruck, Austria, February, 2004.

2.Eiji Kawai, Youki Kadobayashi, and Suguru Yamaguchi. Efficient Network I/O Polling with Fine-Grained Interval Control. International Conference on Communication, Internet, and Information Technology (CIIT 2003), Scottsdale, AZ, USA, November, 2003.

3.Eiji Kawai, Akira Shirahase, Kiyoshi Tsukada, and Suguru Yamaguchi. Practical Migration Strategy to IPv6 for Enterprise Web Services. The 11th International World Wide Web Conference, May, 2002.
その他3件 計6件）

口頭発表 研究会等）
1.河合栄治, 門林雄基, 山口英. ネットワークプロセッサ技術に関するサーベイ. 電子情報通信学会 IA 研究会, 2003年 5月.

2.河合栄治, 門林雄基, 山口英. 多重化 I/O の実行間隔制御におけるスケジュール操作による確定的なプロセッサ利用の実現. 情報処理学会 システムソフトウエアとオペレーティングシステム研究会, 2003年 5月.

3.河合 栄治, 門林 雄基, 山口 英. 多重化 I/O の実行間隔制御による効率化手法. 日本ソフトウエア科学会, SPA2003, 2003年 3月.

4.河合 栄治, 白波瀬 章, 塚田清志, 山口英. 商用 WWW サービスの IPv6 環境移行技術の研究. 情報処理学会 マルチメディア通信と分散処理研究会, 2002年 3月.
その他8件 計12件）

研究課題別評価

1. 研究課題名：超計算 :ソフトウェア自動生産のための新領域探求

2. 研究者氏名：Robert GLUECK

3. 研究の狙い：

The research explored new frontiers of automated software production. The goal is to build programs that build program. The scientific approach taken in this project is unique in that I investigate a combination of three fundamental principles: (1) three basic transformation operations on programs (program composition, program inversion, and program specialization), (2) multiple layers of these transformations, and (3) their portability to new languages via interpreters. I study these principles using semantically clean functional languages.

4. 研究結果：

Our goal is to explore the frontiers of automatic software production based on a combination of three fundamental insights.

(1) *Three operations.* Our thesis, based on a structural analysis of formal linguistic modeling as explained in our earlier publication [13], is that three fundamental operations are needed: *program composition, program inversion, and program specialization.* We found that these operations have to be performed efficiently and effectively by tools for software production to be truly powerful. Of these, program specialization, also known as partial evaluation, has been studied intensely and is the best understood method.

(2) *Layers of metasystems* solve a wide spectrum of transformation problems using only the three types of operations listed in (1). A cornerstone in this development are the *Futamura projections* which make use of two metasystem layers of program specialization. We examined novel meta-system structures including the specializer projections, multi-level generating extensions and a new metasystem scheme for program composition and program inversion (cf. [2, 7, 12]).

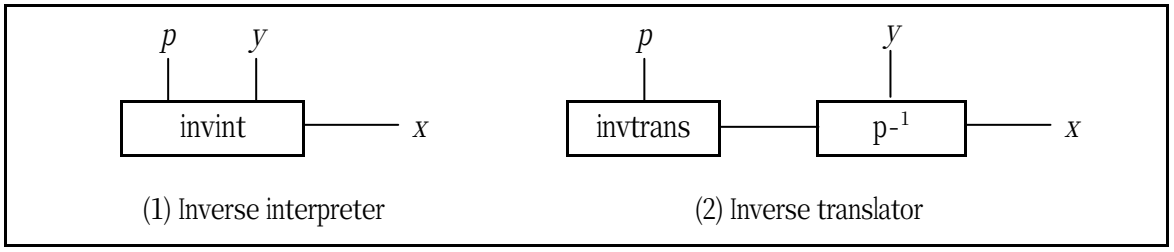


Figure 1: Two tools for solving inversion problems (where [[*p*]] *x* = *y*)

3. *New programming languages* for the construction software will continue to emerge rapidly as

information technology evolves (cf. the recent phenomenal success of Java). There is no evidence that any particular programming language will be the last in this series. Solutions for (1) and (2) must be able to accommodate languages as they are needed to be truly successful. *Semantics modifiers*, a novel concept for robust semantics [2], promise language independence for composition, inversion, and specialization.

We have identified these three principles as important through our research. Existing approaches to automatic program transformation have only considered part of the operations in (1) or used only restricted forms of metasystem schemes (2). Semantics modifiers (3) are original and, thus, have not been investigated before.

Our research goal was to advance the theory and methods for automatic program transformation based on the principles identified above, and to study the computational feasibility of our scientific ideas for theoretically clean, functional languages. We approached these scientific questions partly by theoretical means and partly by experimental work. What follows is a technical overview. References to publications are provided for more detailed information.

A. Inversion of functions is a fundamental concept in mathematics, but the inversion of programs has received little attention in software science (with the exception of logic programming). Programs that are inverse to each are often used. Perhaps the most common example are programs for compressing and decompressing files sent via networks. Today, programs for both transformations need to be written manually, but this is not necessary. One program should be sufficient, and then have a program inverter derive the other program automatically.

Inversion problems can be solved in two ways, either by an *inverse interpreter* or by a *program inverter*. Both software tools are illustrated in Fig. 1. We studied both approaches for first-order functional languages. A difficulty for program inversion is that traditional programming languages do not support computation in both directions and that there is little known about the automatic generation of inverse programs. Logic programming is suited to find multiple solutions and can be regarded as a method for inverse interpretation, but only for relational programs. A detailed description of these notions can be found in our publications [1, 2, 3].

We studied the Universal Resolving Algorithm (URA), a powerful method for inverse computation for first-order functional programs. The algorithm was implemented in Scheme for a typed dialect of S-Graph, and shows some interesting results for the inverse computation [2, 3]. The algorithm is powerful enough to deal with multiple solutions. We also showed that the algorithm is sound and complete, and computes each solution in finite time [4]. Due to the interpretive nature of the algorithm, inverse computation by URA is slower than using an inverse program.

We analyzed the Korf-Eppstein method (short, KEinv) for automatic program inversion of first-order functional programs [10] and formalized the transformation using a structural operational semantics. It is one of only two existing general-purpose automatic program inverters that were ever built. This was the basis for studying the generation of inverse programs.

Recently we proposed [11] a method for automatic program inversion in a first-order functional

programming language that achieves transformations beyond KEinv. One of our key observations is that the duplication of values and testing their equality are two sides of the same coin in program inversion. This led to the design of a new self-inverse primitive function that considerably eases the inversion of programs. We illustrated the method with several examples including the automatic derivation of a program for run-length decoding from a program for run-length encoding. This derivation is not possible with other methods, such as KEinv. Another example, more theoretical in nature, is the inversion of a program *fib* that computes pairs of neighboring Fibonacci numbers; for instance, *fib*(2)=<2, 3>. The automatic inversion is successful and produces an inverse program fib^{-1}; for instance, fib^{-1}(<34, 55>)=8.

B. Composition The construction of complex software by sharing and combining components in order to ease software construction is the main focus of many recent approaches. But abstraction layers do not come for free: they add redundant computations, intermediate data structures, extra run-time error checking. Program composition is a program optimization that can remove such redundancies, and allows the composition of software parts without paying an unacceptably high price in terms of efficiency.

We examines the problem to transform functional programs, which intensively use append functions into programs, which use accumulating parameters instead (like efficient list reversal) [14]. We studied an (automatic) transfomation algorithm for our problem and identify a class of functional programs, namely restricted 2-modular tree transducers, to which it can be applied [15]. We showed how intermediate lists built by a selected class of functional programs, namely "accumulating maps", can be deforested using a single composition rule. For this we introduced a new function 'dmap', a symmetric extension of the familiar function 'map'. While the associated composition rule cannot capture all deforestation problems, it can handle accumulator fusion of functions defined in terms of 'dmap' in a surprisingly simple way. For this research direction we conclude, that automatic, non-trivial composition remains a challenging research problem for the future. Possibly, program composition the most difficult of the three operations to achieve in an automatic and general fashion.

C. Semantics modifiers A key ingredient of our approach are *semantics modifiers* because they allows the design of general and reusable program transformers which make use of results of task A and B, in principle, portable to other programming language.

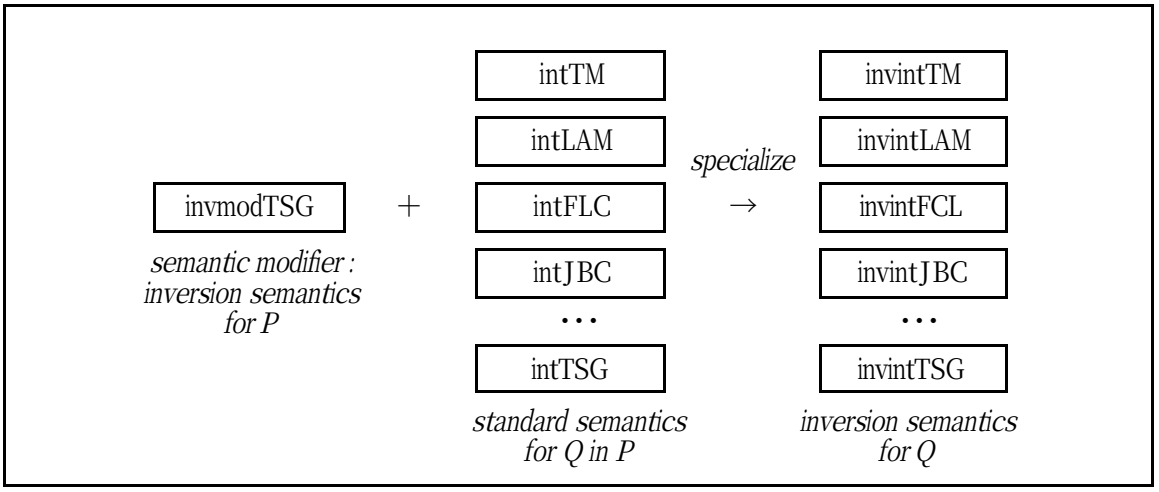


Figure 2: Semantics modifier + standard semantics = non-standard semantics.

We developed a mathematical theory for non-standard semantics and examined the meaning of several non-standard interpreter towers [1]. Our results suggest a technique for the implementation of a certain class of programming language dialects by composing a hierarchy of non-standard interpreters. Based on this theory, we experimented [12] with the Universal Resolving Algorithm (see A above) to prototype programming language tools from robust semantics: we used automatic program specialization to turn interpreters into inverse interpreters for several small languages for which no hand-written tools exist (including interpreters for an applied lambda calculus, an imperative flowchart language, and a subset of Java bytecode). This is illustrated in Fig. 2.

This application of self-applicable program specializers is remarkable since it suggest a new use of program specialization that is different from the familiar Futamura projections. Also, we studied powerful specialization methods [6], loop peeling to increase the accuracy of program analysis [16] and edited a special issue on program transformation and partial evaluation [9].

For our experiments we needed to analyze the power of program specialization and have done so for online and offline partial evaluation [5], for the Futamura projections [8] and binding-time improvements [7].

Despite practical successes with the Futamura projections, it has been an open question whether target programs produced by specializing interpreters can always be as efficient as those produced by a translator. We showed that, given a Jones-optimal program specializer with static expression reduction, there exists for every translator an interpreter which, when specialized, can produce target programs that are at least as fast as those produced by the translator. We call this class *translation universal* specializers. We also showed that a specializer that is not Jones-optimal is not translation universal. In a second step we examined Ershov 's generating extensions and introduced the class of generation universal specializers. We answered these questions on an abstract level, independently of any particular program specializer. We were interested in statements that are valid for all specializers, and have identified such conditions.

In another study about the strength of program specializers, we showed that the accuracy of online

partial evaluation, or polyvariant specialization based on constant propagation, can be simulated by offline partial evaluation using a maximally polyvariant binding-time analysis [5]. We showed [7] that Jones optimality, which was originally aimed at the Futamura projections, plays an important role in binding-time improvements. The main results show that, regardless of the binding-time improvements which we apply to a source program, no matter how extensively, a specializer that is not Jones-optimal is strictly weaker than a specializer which is Jones optimal.

5. 自己評価：

Our research centered around three important principles (three program operations, metasystem layers, adaptability). In particular, we examined inverse computation theoretically and experimentally, and adapted an algorithm to several programming language subsets by automatic program specialization, including a small subset of Java Bytecode. We characterized the accuracy of online and offline specialization [5] and identified the conditions for strong binding-time improvements [7] and the translation universality [8] of Futamura projections. We proposed an automatic method for program inversion that is stronger in some important aspects than other inversion methods and shows some remarkable results. [10]. For program composition, attribute grammars are promising and we have done steps in this direction [14,15], but conclude that the fundamental problem of accumulator fusion remains a challenging research task for future work.

We found that there is no theoretical limit to the translation power of the Futamura projections provided a specializer with static expression reduction is also Jones-optimal and introduced the class of translation universal specializers. We believe that the power to perform universal computations is another property for the theoretical power of a program specializer. Whether the results can be adapted to other non-standard interpreter hierarchies as developed in [1] is a topic for future work. It is quite possible that the results [7,8] can be carried to the next metasystem level. We also want to explore the conditions for generating translators and other program generatorsfrom generation universal specializers.

Our experiments applied the idea of prototyping programming language tools from robust semantics [12]: we produced automatically inverse interpreters for programming languages for which no inverse interpreter existed before. Even though these languages are small, the results demonstrate that it is possible in practice with existing partial evaluators. To the best of our knowledge, these are the first results regarding this use of partial evaluation. Our results show that a speedup of an order of magnitude can be achieved for some interpreters. Limiting factors of offline partial evaluation was the need for binding-time improvements and the lack of generalization.We believe there is still more to be gained by partial evaluation and want to investigate stronger specialization techniques, such as [6].

A main difficulty in the generation of inverse programs are conditionals and recursive functions. We now try to solve some of these difficulties through the application of parsing techniques to program inversion. Tasks for future work also include the refinement of the well-formedness criteria [10]. We have not exploited mathematical properties of operators during the inversion. A possible extension of our techniques may involve the use of constraint systems for which well-established theories have been developed in other areas.

We described an algorithm for inverse computation, studied its organization and structure, and illustrated our implementation with several examples [3,12]. Methods for detecting finite solution sets and cutting infinite branches can make the process of inverse computation terminate more often (while preserving soundness and completeness) and may make the method more practical. Techniques from program transformation and logic programming may prove to be useful in this context, and we are now taking first steps into this direction. We also want to explore further its portability to new languages via semantics modifies [1,2].

6. 研究総括の見解：

仕様からソフトウェアを自動生成する方法に関してはいくつかのアプローチがあるが、Glueck 研究者は、正しいが効率の悪いプログラムをプログラム変換の手法により効率を上げる方法を研究した。この方法は従来から多くの研究が続けられ、多くの成果が報告されているが、未だ実用的な解決が得られていない難問である。Glueck研究者は、プログラム変換における基本変換である、合成、逆転、特殊化について科学的に質の高い研究を行い、この領域の発展に貢献した。また、評価の高い国際会議や論文誌にて多数の論文発表を行った。特に、プログラム逆転や特殊化については非常に優れた結果を出している。もちろん、これらの結果によっても実用的な問題が解ける段階にはなっていないが、そのための基礎となる理論的、科学的貢献は大きい。

7. 主な論文等 〔References (international)〕:

1. S. M. Abramov and R. Gluck. Combining semantics with non-standard interpreter hierarchies. In S. Kapoor and S. Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science. Proceedings*, Lecture Notes in Computer Science, Vol. 1974, pages 201?213. Springer-Verlag, 2000.
2. S. M. Abramov and R. Gluck. From standard to non-standard semantics by semantics modifiers. *International Journal of Foundations of Computer Science*, 12(2):171?211, 2001.
3. S. M. Abramov and R. Gluck. Principles of inverse computation and the universal resolving algorithm. In T. A. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, Lecture Notes in Computer Science, Vol. 2566, pages 269? 295. Springer-Verlag, 2002.
4. S. M. Abramov and R. Gluck.The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193?229, 2002.
5. N. H. Christensen and R. Gluck. Offline partial evaluation can be as accurate as online partial evaluation. *ACM TOPLAS*, to appear, 2003.
6. Y. Futamura, Z. Konishi, and R. Gluck. WSDFU: Program transformation system based on generalized partial computation. In T. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 358?378. Springer-Verlag, 2002.
7. R. Gluck. Jones optimality, binding-time improvements, and the strength of program specializers In *Proceedings of the Asian Symposium on Partial Evaluation and Semantics-Based Program*

Manipulation, pages 9?19. ACM Press, 2002.

8. R. Gluck. The translation power of the Futamura projections. In M. Broy and A. V. Zamulin, editors, *Perspectives of System Informatics. Proceedings*, volume 2890 of Lecture Notes in Computer Science, pages 133-147. Springer-Verlag, 2003.
9. R. Gluck and Y. Futamura. Special issue on partial evaluation and program transformation. *New Generation Computing*, 20(1):1?124, 2002.
10. R. Gluck and M. Kawabe. An automatic program inverter for Lisp: potential and limitations. In Y. Fu and Z. Hu, editors, *Proceedings of the Third Asian Workshop on Programming Languages and Systems*, pages 230?245. Shanghai Jiao Tong University, 2002.
11. R. Gluck and M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori, editor, *Asian Symposium on Programming Languages and Systems. Proceedings*, volume 2895 of Lecture Notes in Computer Science, pages 246-264. Springer-Verlag, 2003.
12. R. Gluck, Y. Kawada, and T. Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 10?19. ACM Press, 2003.
13. R. Gluck and A. V. Klimov. Metacomputation as a tool for formal linguistic modeling. In R. Trappl, editor, *Cybernetics and Systems '94*, volume 2, pages 1563?1570. World Scientific, 1994.
14. K. Kakehi, R. Gluck. and Y. Futamura. On deforesting parameters of accumulating maps. In A. Pettorossi, editor, *Logic Based Program Synthesis and Transformation. Proceedings*, volume 2372 of *Lecture Notes in Computer Science*, pages 46?56. Springer-Verlag, 2002.
15. A. Kuhnemann, R. Gluck. and K. Kakehi. Relating accumulative and non-accumulative functional programs. In A. Middeldorp, editor, *Rewriting Techniques and Applications. Proceedings*, Lecture Notes in Computer Science, Vol. 2051, pages 154?168. Springer-Verlag, 2001.
16. L. Song, R. Gluck and Y. Futamura. Loop peeling based on quasi-invariance/ induction variables. *Wuhan University Journal of Natural Sciences*, 6(1-2):362?367, 2001.

研究課題別評価

1. 研究課題名：理論領域と実用領域を結ぶ新しいプログラミング単位

2. 研究者氏名：河野真治

3. 研究の狙い：

実用領域では、JavaやC++などの複雑なプログラミング言語や、高度なGUIが導入され、ハードウェアを含む組込みシステムなどの複雑な開発を迅速に行なうことが要求されている。一方で、理論領域でも、モデル検査やCPS変換によるプログラム理論の詳細化などが進んでいる。しかし、その双方を接続する方法は少ない。我々は、これを継続を基本とした新しいプログラミング単位を導入することで解決したいと考えている。

従来のプログラミング単位

これまで、主に情報を隠すため、プログラムを分割するために関数呼び出しや制御構造、オブジェクト指向プログラムなどが導入されて来た。一方で理論の方もさまざまな進歩があり、特に、最近のソフトウェアの信頼性に対する要求から、さまざまな手法が開発されている。特に、定理証明や、充足可能性を検査するモデル検査の手法が導入され、実際のプロトコルやプログラムなどに応用されている。

これらの両方を結びつけるプログラミング単位を従来のプログラム言語に対応した形で、提供できれば、理論領域の成果を、既に書かれたソフトを大量に持つ実用領域に結びつけることが容易になると考えられる。

継続を基本とした新しいプログラム単位 (code segment)

ここで導入する単位はステートメントよりも大きくサブルーチンよりも小さい単位である。ループ構造を持たないステートメントの集合であり、コンパイラでは基本ブロックと呼ばれるような部分に相当する。これをプログラム単位としてプログラム言語的に明示的に使用するのが、ここでの新しい提案である。この単位を code segment と呼ぶ。code segment は継続(continuation)で接続される。継続への移動は引数付きのgoto文 (parameterized goto statement)で表現される。code segment 自体は制限されたC のステートメントにより表現される。この言語を CbC (Continuation based C)と呼ぶ。継続を持つCに近い言語としては、C - - が知られているが、CbC は、継続を基本とするところが異なる。

code segment は入力interfaceから条件文によって分岐する複数の出力interfaceを接続する単位となっている。状態遷移系を直接に表現する単位となっている。

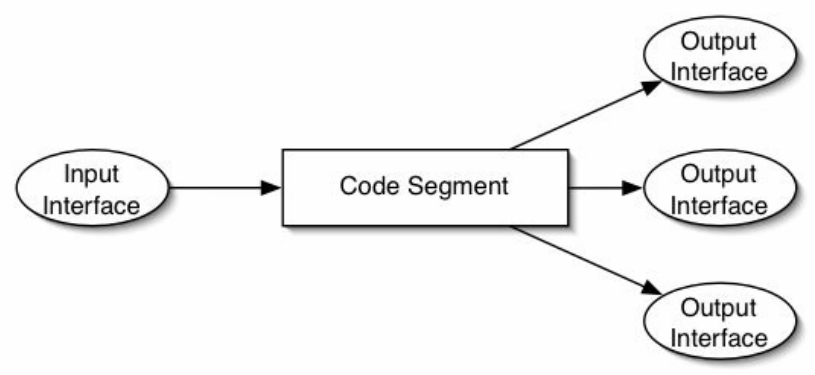


図１ Code Segment

以下の例は、CbC による階乗の計算である。

```
code fact(int n,int result,
          code (*print)()){
    if(n>0){
        result *= n;
        n--;
        goto fact(n,result,print);
    } else
        goto (*print)(result);
}
```

goto fact(n,result,print); は、直接の継続であり、goto (*print)(result);は間接の継続である。その引数は、interface と呼ばれる。属するcodeと同じinterfaceを持つgoto文は、一つのjump命令にコンパイルされる。

CbC は、C の下位言語であるが、C のサブルーチンへ戻るための環境付き継続を導入すると C の上位言語となる。この言語を CwC (C with Continuation)と呼ぶ。CwC ではCbCから通常のCの関数を呼び出すことができる。

CbC の使い方

CbC はプログラム変換の基本単位とし使うことを想定している。CbCのinterfaceを物理的なレジスタなどとの対応を定義することにより最適化そのものをCbCレベルで記述することも可能である。また、CPUの動作そのものを記述することも可能である。また、既存の言語からCbCに変換することで、既存のソフトウェア資産の解析ツールとして使うことが出来る。

C からの変換はコンパイラによって行なわれるが、ループの変換は自明である。難しい部分は、関数呼び出し時のスタックの明示的な記述である。

```
j = g(i+3);
```

のようなCの関数呼び出しは、 struct f_g0_save などの明示的なスタックの中身を表す構造体を用いて、

```
struct f_g0_interface *c =
  (struct f_g0_save *)(sp -=
    sizeof(struct f_g0_save));
c = sp;
c->ret = f_g1;
goto g(i+3,sp);
```

のような形で、明示的なスタック操作に変換される。これは変換の一例であり、他の方法、例えばリンクトリストなどを用いても良い。f_g1 は、関数呼び出しの後の継続であり、g では、

```
code g(int i,stack sp) {
    goto (* ((struct
      f_g0_save *)sp)->ret)
        (i+4,sp);
}
```

のように間接的に呼び出される。スタックの中は、継続と中間変数などを格納する構造体である。スタックそのものは、これらの構造体を格納するメモリである。

仕様記述としての使用法

仕様記述としては一般的には時相論理などが使われることが多いが、□(p→◇q) などの意味は決定的なオートマトンとして定義することが可能である。このオートマトンをCbCによって記述することにより、仕様記述として用いることが出来る。

仕様の検証手法としては、実装記述とともに同時に仕様記述である決定的オートマトンを走らせるassert的な使い方が考えられる。また、CbCのinterface の状態を有限な状態に抽象化することができれば、実装記述とともにタブロー展開などの手法で直接的に実装の正しさを証明することも可能である。

タブロー展開は、仕様とプログラムの大域的な状態をすべて生成する手法である。反例を探す場合は反例が見つかった場合に停止して良いが、証明を行なう場合はすべての大域的な状態を生成する必要がある。大域的な状態の生成は、初期状態から非決定的に生成されるすべての次の状態を生成すること(状態の展開)により行なわれる。証明にはプログラムの抽象化された状態の数に比例し、また、プログラムが含む変数の数の指数乗の計算量がかかる。

CbC と並列検証系

実用的なプログラムではプログラムの持つ抽象化された状態は有限であり、状態遷移に影響を与える変数の数もそれほど多くはないと予想される。このような場合には、並列計算機による力技的な検証系が有効であると考えられる。この研究では、以下のようなリダクション・マシン的な並列タブロー法

を採用している。

状態は時相論理式の部分項を変数として持つ部分項BDD(Ordered Binary Decision Diagram 決定二分木)を用いて表現され、部分項BDDの変数の順序を並列計算機全体で維持する単一の部分項サーバを持つ。状態の展開はワーカーと呼ばれる各計算機ノードで行なわれる。生成された状態は部分項BDDに変換される。変換された部分項 BDDは、ハッシュ関数により各計算機ノードにランダムに分配される。分配はランダムなので十分な状態数があれば適切な台数効果が得られるはずである。

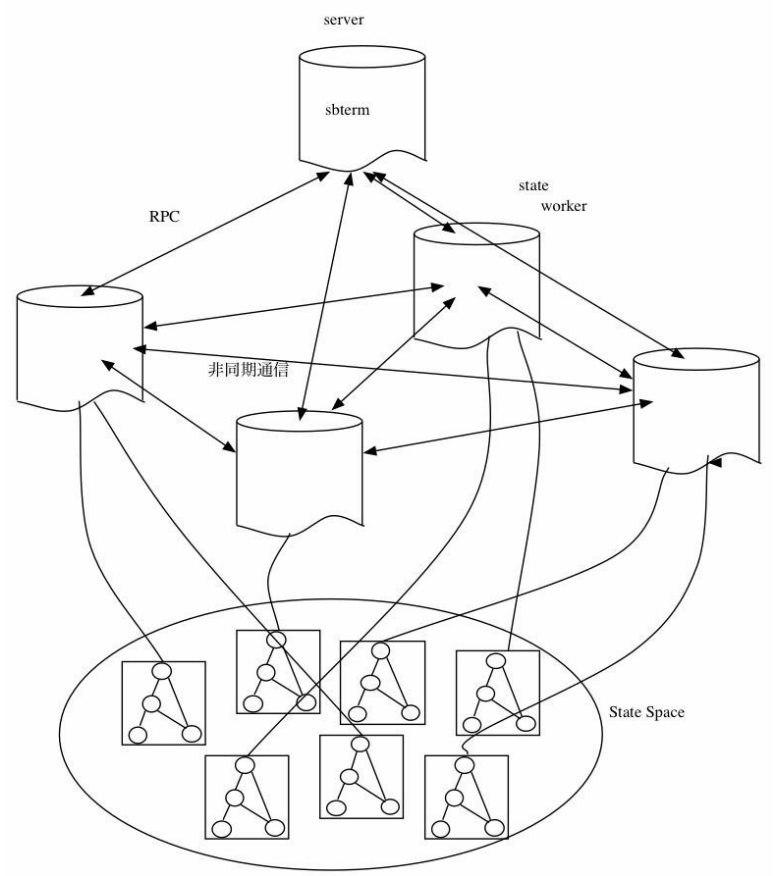


図2 並列検証例

並列検証系のための通信ライブラリ

このような並列計算手法はデータは計算機ノードに分散し共有されないので、PCクラスタなどに向いた実装となる。ただし、この場合の通信は完全にランダムに行なわれる。PCクラスタ用の通信ライブラリはMPIなど、いくつかあるが完全結合的な通信を行なう場合を想定したライブラリではなく、部分的な通信を想定した実装が多いので、このような分散プログラムには不向きである。したがって、目標となる並列検証系に適した通信ライブラリを設計、実装する必要がある。

4. 研究結果：

CbC, CwC について

Continuation based C および C with Continuation の実装を行った。CwC は現在、IA32 用、PowerPC 用のコンパイラが実装されている。

C からの変換は、機械的に行なうことが可能だが、変換の詳細は若干複雑であり変数の種類や寿命などを含めた構文解析をする必要がある。変換の結果は、call などのアセンブラによる関数呼び出しのサポートを使えないために、元の C よりも早くなることはない。しかし、スタック操作が不要になる場合を検出し、それを取り除くことができれば、高速になる場合がある。このような変形を行なった場合と、gcc の最適化を行なった場合の実行時間の比較を行なってみた。以下のような三段の関数呼び出しについて10^7程度の繰り返しを例題として用いた。

```
        int k,j;
        k = 3+i;
        j = g0(i+3);
        return k+4+j;
}
g0(int i) { return h0(i+4)+i; }
h0(int i) { return i+4; }
```

CwC コンパイラでの同程度の最適化を見ると C から CbC への変換後のソースは Pentium Ⅲ で 12.5%、PowerPC で 42.9%程度遅くなるなることがわかる。スタック操作を削除する変換により、元の C の関数呼び出しよりも、50%程度高速になる。これは、gcc で最適化を行なわない場合よりも高速である。gcc は、-O6 による最適化の効果が大きく スタック操作を削除した CwC コンパイラによるものよりも高速な実行となる。特に PowerPC での gcc の最適化の効果が大きい。これは、h0(i) のような関数呼び出しはリンクレジスタのみを使うスタック操作を含まない形にコンパイルされるためである。

Suci Library について

完全結合向きの PC クラスタ通信ライブラリ Suci を作成した。このライブラリは、Unix のデータグラム・ソケットを用いており、ユーザレベルで TCP の信頼性制御、フロー制御に相当する部分を行なう。これにより、スループット優先の通信とレスポンス優先の通信を持つこの並列検証系でも有効な通信を行なうことが可能である。いくつかのベンチマークで TCP ベースの MPI の実装である MPICH よりも良好な結果を得ている。

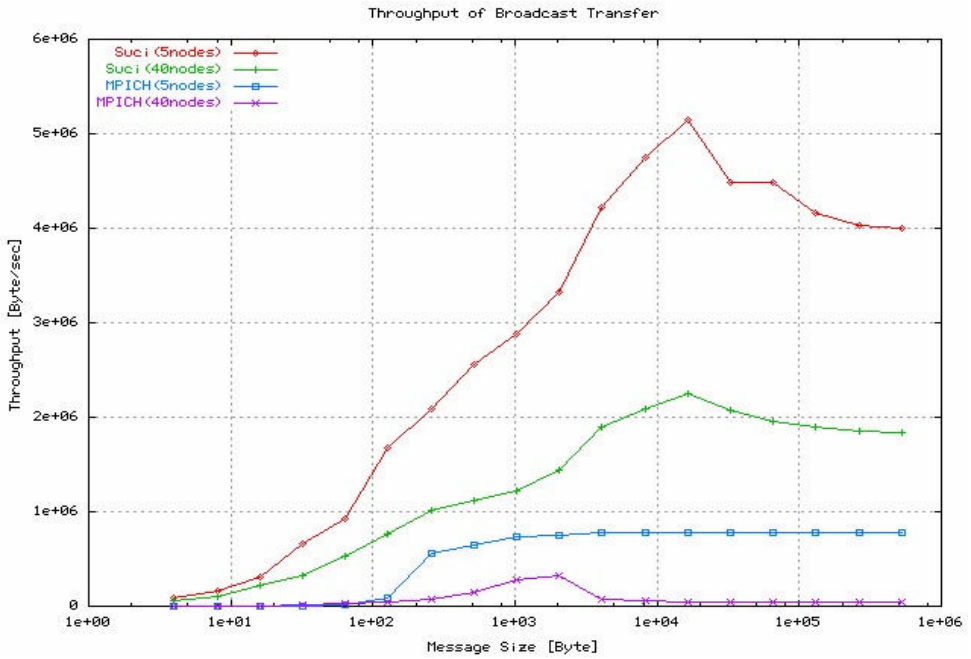


図３ Suci ベンチマーク

並列検証系について

並列検証系自体は Prolog で記述されており、ITL に関する検証を行なうことができる。Prolog は C で記述された Suci ライブラリを呼出して通信を行なう。

50 台程度の PC クラスタで良好な台数効果が得られることを確認している。ここでは例題は Dining Philosopher (6 人)と Unix のマウスドライバを用いている。Interleaving を多く含み状態数の多い Dining Philosopher では台数効果が得られやすい。一方で、変数の数の多いマウスドライバでは 30 台程度で十分な高速化が得られていることがわかる。実時間では 43 台で 30 秒、マウスドライバの場合で 6 分程度で終了している。

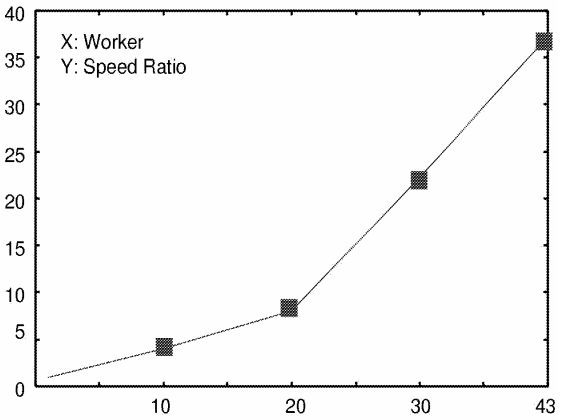


図4 並列検証系 Dining Philosopher 6

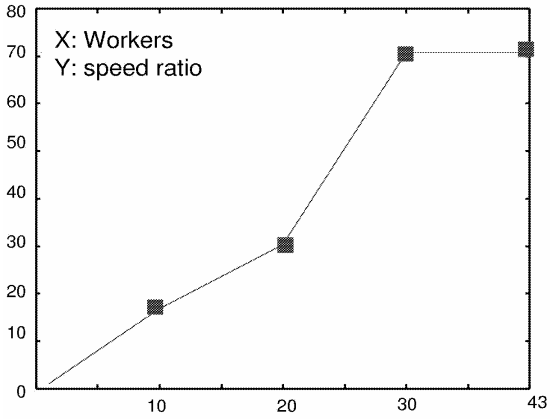


図5 並列検証系 Mouse Driver

5. 自己評価：

この研究は、継続を基本とした新しいプログラム単位を提供することにより、プログラム変換やプログラム検証という理論的な成果を、既存のプログラム資産を多量に含む実用領域に適用することを目的としたものである。

新しいプログラム単位は、CbC/CwC コンパイラという形で提供されるが、その上でプログラムを行なうだけでなく、明確な実行意味を持つ仕様記述としても用いることが出来る。

また、その単位を用いた並列検証系を実装するための並列検証系のアルゴリズムを提案し、それを効率良く実現するための汎用の並列通信ライブラリを作成した。

今後は、CbC/CwC コンパイラを実用的なレベルにまで質を上げるとともに、並列検証系と組み合わせた、新しいプログラム単位を直接的に使った検証システムを提供することを考えている。

6. 研究総括の見解：

河野研究者は、主に組込みソフトウェアなどを対象として実用的なソフトウェアの開発に利用でき、かつ、モデル検査などによる正しいプログラムの構築が行いやすいプログラミング言語、その言語で記述されたプログラムの検証法、検証システム構成法に関する研究を行い、実用的で有望な結果を得た。従来この種のプログラムの開発はC言語で行われているが、河野研究者はC言語のサブセットで検証に適したCbCと呼ばれる言語を新たに設計し、その言語に対するモデル検査による並列検証系の研究とその実装を行い、CbCによる実用的で正しいプログラムの開発の可能性を実証した。ソフトウェアの不具合による携帯電話のリコール問題などが多発する現在、河野研究者の研究成果は非常に重要であり、その実用化に期待したい。

7. 主な論文等：

1. 河野真治, 神里健司. UDP を使った分散計算環境とその応用. 日本ソフトウェア科学会第 16 回大会論文集, September 1999.
2. Shinji Kono. Parallelization of Temporal Logic Verification by Dividing State Set. 1st International Workshop on Specification and Verification of Timed Systems. March 1999.
3. 河野真治, 継続を持つC の下位言語によるシステム記述. 日本ソフトウェア科学会第 17 回大会論文集. Sep 2000.
4. 河野真治, 神里健司. User Level Flow Control API をもつ並列ライブラリの実装. SwoPP 2001. July 2001.
5. 河野真治, 揚挺. C 言語の Continuation based C への変換. SwoPP 2001. July 2001.
6. 佐渡山陽, 河野真治. Continuation Based C による PS2 Vector unit のシミュレーション. 情報処理学会システムソフトウェアとオペレーティング・システム研究会. June 2002.
7. 河野真治, 佐渡山陽. Continuation Based C による Technology Mapping のサポート. FIT 2002. Aug 2002.
8. 屋比久友秀, 河野真治. ユーザレベル通信ライブラリ Suci のスナップショット・アルゴリズムへの応用. FIT 2002. Aug 2002.
9. 河野真治. 継続を基本とした言語 CbC の gcc 上の実装. 日本ソフトウェア科学会第 19 回大会論文集. Sep 2002.
10. 屋比久友秀, 河野真治, 山城潤. トランスポート層を考慮したスナップショット・アルゴリズムの考察. 日本ソフトウェア科学会第 19 回大会論文集. Sep 2002.
11. 山城潤, 河野真治. Java によるユーザトランスポート層の実現と評価. 情報処理学会システムソフトウェアとオペレーティング・システム研究会. May 2003.
12. 上里献一, 河野真治. Suci ライブラリのスナップショット API を利用した並列デバッグツールの設計. 日本ソフトウェア科学会第 20 回大会論文集. Sep 2003.
13. 河野真治. 継続を基本とするプログラム単位を用いた組込みシステム開発. 組み込みソフトウェア工学シンポジウム 2003. Oct 2003.

研究課題別評価

1. 研究課題名：インターコミュニケーション・プログラミング

2. 研究者氏名：関口龍郎

3. 研究の狙い：

この研究ではインターネット上での人と人との双方向コミュニケーションを支援するソフトウェアを安全で効率良く実行できるプログラミング言語システムの構築を目指した。具体的には、Java 仮想機械や.NET ランタイム(CLR)のような、言語非依存、アークテクチャ非依存のコード表現の開発とその実行環境の実装を目標とした。普遍的なコード表現の開発における最大の問題は実行効率と安全性の両立である。Java 仮想機械では安全性を重視するためにプログラムを実行する直前にプログラム検証を行い、合格したものだけが実行される。C 言語などの効率の良い言語で記述されているコードパターンにはポインタ演算等のプログラム検証の制限によって許可されない演算があるため Java 仮想機械では C 言語ほど効率の良い実行ができない。.NET ランタイムではこの制限を回避するためにプログラム検証を行わずに実行するオプションがある。プログラム検証を行わなかった場合にはもちろん実行の安全性は保証されないことになる。インターネットで使われているメールやウェブのサーバやブラウザに対する攻撃方法のほぼ半数はプログラミング言語の仕様の欠陥である不正メモリアクセスを利用しており、言語としての安全性の保証は重要である。この研究では、高い実行効率を実現するために極めて重要な演算であるが Java のプログラム検証では除外されているポインタ演算に着目し、ポインタ演算を含むプログラムの検証を可能にする検証技術を開発し、実行効率と安全性が両立されたプログラミング言語システムの構築を目指した。

4. 研究結果：

本研究の成果は、①ポインタ演算を記述できる言語非依存、アーキテクチャ非依存の低水準言語、②前項の低水準言語の SPARC と IA32 プラットフォームへのコンパイラ、③ポインタ演算を含むプログラムの検証手法の設計と検証器の実装、④標準的なベンチマークである SPECCPU2000 を利用した検証アルゴリズムの性能データ、以上 4 点である。

(1) 我々の設計した低水準言語と Java バイトコードとの大きな違いは 2 点ある。一つはポインタ演算を許可していることであり、もう一つは Java のように例外を言語 primitive として提供するのではなく、「例外を実装できる機構」を提供していることである。ポインタ演算を記述できるので様々な高級言語のプログラムを効率を落さずに我々の低水準言語にコンパイルすることができる。現代的なプログラミング言語では例外機構は不可欠の機能であるため、我々の低水準言語では例外を効率良く実装できる機能を提供している。これは、高級言語から C 言語への変換によって高級言語を実装するときにうまく記述できない箇所である。我々の言語は C 言語に近い syntax と BCPL に近い semantics を持っている。

(2) 我々はこのような低水準言語を単に設計するだけではなく、Solaris と Windows 上で実行可能な SPARC と IA32 の機械語コードを生成するコンパイラの実装を行い、設計の有効性の検証を行った。

コンパイラは Objective Caml で記述されており、コード量はおよそ 18000 行である。同じ意味を持つコードの実行効率をGNU C Compiler と MS C の出力するコードと比較を行った。多くの場合は同程度であり、いくつかのケースでは我々のコンパイラの方が 2? 3 割速いコードを生成した。

(3) 我々は、ポインタ演算を記述できる我々の低水準言語上でのプログラム検証を行う検証アルゴリズムを開発し、検証器の実装を行った。この検証アルゴリズムはいわゆるポインタ解析と呼ばれるプログラムの静的解析手法に基づいている。この解析によって変数の指す可能性のある値の範囲とメモリに格納される可能性のある値の範囲を静的に見積もることができる。変数の取り得る値を静的に決定できるため、実行時にヌルチェック、境界チェックなどの種々のチェックを省くことができ、実行効率を向上させることができる。解析器は Objective Caml で記述されており、コード量はおよそ 2000 行である(ただし予備処理を行うコードが加えて 14000 行ある)。

(4) 我々は標準的なベンチマークである SPEC CPU2000 に対して解析器を適用し、性能を計測した。解析器は数千行の C 言語のプログラムを数秒から十数秒で解析することができ、ほぼ線形時間で解析を行うことができる。この解析時間は充分実用的な水準にある。解析の結果を利用して 2 割から 7 割の実行時チェックを除去することができ、実行時間は 1 割から 4 割向上することが分かった。我々のプログラム検証技術により安全性を損なうことなく実行効率を高めることができた。

5. 自己評価：

Java 仮想機械や .Net 仮想機械とは質的に異なる全く新しい手法により安全性と実行効率を両立させるプログラミング言語システムを作ることができたと思っている。しかし当初想定していたメールサーバなどの現実的なアプリケーションを対象に有効性の検証を行うところまで至らなかったのは残念である。プログラム検証アルゴリズムは従来あるポインタ解析アルゴリズムと比べてかなり複雑なものであり、現実的な時間で実用的なプログラムを解析できるかどうか不安があったが(ポインタ解析には数ギガバイトのメモリと数十時間の解析時間を要するものも多い)結果を見ると充分高速に解析できており、ほっとしている。当初は実用的なシステムを作ることに主眼を置いていたが、実用的な規模のプログラムを解析してみた経験からプログラム検証の新しい方法の着想も得ており、今後理論的に新しい結果も出せそうである。この研究は、一般のユーザが利用できるツールを公開するところをもって完結すると思っており、これからは「きっかけ」と離れることになるが、ツールの公開まで研究を続けるつもりである。コンピュータ科学は1人から数人のグループによる研究結果によって世界に影響を与える仕事を行える分野であり、個人研究型の形態は意味があると思う。

6. 研究総括の見解：

インターネットが社会基盤となっている現在、そこでの安全性の確保は極めて重要であり、安全性を担保可能なインターネットプログラミング言語の研究が活発に行われている。関口研究者はそのような言語の設計と安全性確保のための検証系を組み込んだ言語処理系の設計・構築を行ったが、第 1 級の研究結果を出したと考える。実用性のある言語では安全性上問題のあるポインターを排除することは出来ないことを前提として、ポインター解析に従来より強力な方法を導入することにより、実行速度と安全性の両面において現在開発されている言語としては最高レベルの性能を発揮している。実用的なソフトウェア開発での採用にあともう一歩必要であるが、この分野に非常に大きな貢献をしたと考

える。

7. 主な論文等：

1. Tatsurou Sekiguchi, Takahiro Sakamoto, and Akinori Yonezawa. Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling. Advances in Exception Handling Techniques, LNCS2022, Springer Verlag, May 2001.
2. 関口 龍郎、大岩 寛、米澤 明憲。オブジェクト指向言語によって記述された、携帯電話、PDA のアプリケーションプログラム 圧縮方式. 第 3回プログラミングおよびプログラミング言語ワークショップ. 2001 年 3 月 (コンピュータソフトウェア． Vol.19, No.1． 2002 年 1 月)．
3. Takeo Imai, Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. Dynamic Access Control of Mobile Objects by Switching Namespaces. OOPSLA Workshop of Patterns and Techniques for Designing Object-Oriented Mobile Wireless Systems. October 2001.
4. Tatsurou Sekiguchi, Yutaka Oiwa, and Eijiro Sumii. Software: Rog-O-Matic III , The 2002 International Conference of Functional Programming (ICFP '02) Programming Contest Playoff, October 2002.
5. Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure - Progress Report. International Symposium on Software Security, November 2002 (pp133-153, LNCS2609, Springer Verlag, April 2003).
6. 住井 英二郎、関口 龍郎、細谷 春夫． PLI2002 報告． コンピュータソフトウェア． Vol.20, No.2, pp79-84, 2003 年 3 月．
7. 関口 龍郎． Java のための二つのモバイル技術． オブジェクト指向 2003 シンポジウム． 2003 年 8 月．
8. 関口龍郎． C 言語のための現実的なポインタ解析． 第 6 回プログラミングおよびプログラミング言語ワークショップ PPL2004(予定)． 2004．

研究課題別評価

1. 研究課題名：計算状態パーソナル・スクラップブック

2. 研究者氏名：リチャード ポッター

3. 研究の狙い

The goal of this research is to investigate new programming tools that work by saving and restoring intermediate computation state. The same basic idea has brought important benefits to other types of tools and applications. Scientific and other distributed applications can benefit by saving an application in mid-execution and moving it to a machine with more resources. Other applications gain fault tolerance by saving application state for rolling back to a safe checkpoint in case of hardware or software failures. However, few benefits have been explored for programming tools that can save and restore the computation state of a program under development.

One reason to expect benefits is that quick initialization of programs in mid-execution can make it possible to focus programming tools and programmer attention in new creative ways. People who are learning new programming skills can benefit because it allows a wider range of techniques to be applied to a program that has been subdivided into more manageable pieces. An additional general benefit is that program state can be enumerated more efficiently and methodically, which enables model checking on actual implementations. The questions for this research are what tools can make these potential benefits practical for actual programming activity and what infrastructure will make these tools easy to implement.

Tools based on Computation Scrapbooks have potential to be simple because it is simple and increasingly practical to copy computation state. It is conceptually simple because, like photo copying complex information on a sheet of paper, it is not necessary to understand the information to easily copy it. Copying computation state is practical because hard disk sizes and processor speeds are fast enough to copy realistic computation state quickly.

The challenge of this research is to demonstrate that Computation Scrapbooks can be both useful and practical without adding too much complexity to the simple core functionality that copies the computation state. Thus it is important to distinguish between the *core functionality*, which is well defined and clearly practical, and the *extended functionality*, which may be less well defined and only be possible for certain special cases.

The approach of the research has been to create two systems. The first system is for fast prototyping various Computation Scrapbook based tools. It has limited core functionality, but is flexible for exploring the various types of extended functionality necessary to support the tools. The second system is designed for actual use by real users. It has more rigorous core functionality that is general, however the range of extended functionality and tools is less than the first system.

The name "Computation Scrapbook" represents the core infrastructure itself, and is intended to convey that multiple snapshots of computation state are saved and used in creative ways. It encompasses functionality to save, organize, and restore snapshots of computation state. To support programming tools, a Computation Scrapbook must support thread persistence, because use in programming tools requires checkpointing at a fine granularity and therefore the stack states must be preserved. Also, multiple persistent snapshots will be required for several of the programming tools.

4. 研究結果

In this research, two different Computation Scrapbooks have been implemented using different techniques. SBDebug, the first system implemented, captures the state of Lisp programs that run in the Emacs text editor. It creates snapshots that are typically less than 10KB in size. It works by

instrumenting functions so that they make internal stack frames easy to capture in a top-down manner. Saving and restoring state is quick and the Lisp environment makes it easy to manipulate programs and data. Therefore, SBDebug has been useful for quickly prototyping new Computation Scrapbook tools, although its use is limited to a restricted class of Emacs Lisp programs.

SBUML, the second system implemented, captures the state of the Linux operating system, including all file systems, applications, processes, and kernel state. SBUML is early in development, but it will be useful for creating tools for a wide range of programming languages and will support complex, real-world, multithreaded programs. It works by copying the complete low-level image of all changes to memory and disks of User-Mode Linux, a virtual version of Linux. This makes it less flexible than SBDebug for prototyping innovative tools because the Linux state is more heterogeneous and therefore harder to manipulate than Lisp. Also, the snapshots require more computing resources. Raw snapshots start at about 30MB in size, although snapshots can sometimes be compressed to around 100KB. Saving and restoring raw snapshots takes around 5 seconds on a 2.8gHz dual processor workstation.

Several tools based on the core Computation Scrapbook infrastructure were developed to support various programming activities, such as reading code, writing code, debugging, and testing.

For example, when reading a program, a user can sometimes benefit by watching the program execute using debugging or software visualization tools. However, setting up the debugger so that it shows something useful about a particular section of code can require a lot of skill. A Computation Scrapbook can let a skilled programmer prepare and save such a debugger configuration and share it with other users.

This idea was explored in SBDebug by creating a *snapshot documentation tool* with two features. The first feature lets the experienced programmer paste a hyperlink into source code text. The second quickly takes anybody who selects the hyperlink back to the same debugger configuration. Experimenting with these features has made other uses obvious. For example, hyperlinks could be put in on-line programming language documentation to quickly take readers to live examples of specific language features in action.

While developing a program, sometimes testing has to be delayed until a whole module is completed. Since a snapshot can initialize any part of a larger program, arbitrarily small and partially completed code segments can be tested earlier.

This idea was implemented in SBDebug as a *snapshot test case tool*. To use it, the user sets up the initial computation state for the beginning of the code segment using whatever techniques are most convenient, which might be done by running the program manually, writing a driver, manually editing the state, or some combination of these. This first snapshot is then saved. The user then runs the program to the end of the code segment and saves a second snapshot, possibly editing the computation state manually if the code does not compute the correct outcome. Finally, the user uses the test case tool to create a test case from the two test cases just saved. A second test case tool feature runs the test case. It initializes the code with the first snapshot, runs until the program counter matches that of the target snapshot, and then compares the computation state with the second snapshot to judge if the test passed. A third feature can run a set of test cases with a single command.

Experience with the tool shows that it can be very easy to create a test case in the middle of writing and debugging a program. If later the code is changed, it is easy to rerun the collected set of test cases to quickly check for any new bugs might have been unintentionally introduced.

When writing code, it can sometimes be desirable to give a specific example of what the program does rather than write the abstract code. Although automatic programming techniques are well researched, they only work for very small programs of limited use. Computation Scrapbooks make it possible to apply such techniques to small parts of larger programs, so that they can generate useful code. This can be practical because the snapshot test cases work for code segments that are so short that all possibilities can be enumerated.

This idea can be demonstrated in SBDebug with its *programming by demonstration (PBD) tool*. The user first selects an incomplete or incorrect Lisp expression and also selects a set of test cases. The PBD tool can then enumerate possible expressions until one passes all test cases. In order to make the interface this simple, some plausible defaults were chosen for how to enumerate the expressions. For example, only functions, constants, and other tokens that already appear in the function that contains the original expression are used.
So far this tool is only a proof of concept implementation to show automatic programming is possible for realistic programs when a Computation Scrapbook is used to focus the technique on a small parts of the programs. Some refinements were implemented to delay the inevitable combinatorial explosion. In addition to static type checking, the PBD tool also keeps track of runtime errors to reduce the number of expressions that must be generated and tested.
When verifying the correctness of concurrent programs, model checking techniques can be useful. However, model checking must usually be performed on a separate abstraction that may not match the actual implementation. Computation Scrapbooks can restore the state of an actual implementation repeatedly so that all possible successor states can be generated for model checking. This idea has been demonstrated by enumerating all the possible states of a C implementation of the Dining Philosopher 's algorithm running in SBUML.

5. 自己評価

Overall, the research proceeded steadily with progress in both implementations and high-level ideas. At the beginning of the research period, the high-level ideas were loosely strung together notions about end-user programming, gentle-slope systems, invisible computation state, the increasing practicality of copying computation state, and the idea that interaction with computation state can lead to higher-level understanding. This was enough of a framework to guide the quick implementation of SBDebug. Experience with SBDebug resulted in a clearer understanding of the high-level benefits of Computation Scrapbooks, which mostly come from the ability to quickly initialize programs in mid-execution. From this core benefit, other benefits are easy to explain, including how programs in mid-execution provide context that is useful for gentle-slope systems. Some of the refined high-level ideas were published in a paper that explained how Snapshot Documentation is an appropriate technique for gentle-slope systems. Experience with SBDebug also gave the confidence to start investigating the practicality of Computation Scrapbooks for other systems.
After investigating the practicality of a Computation Scrapbook system for Java, it became apparent that one for Linux might be easier to implement. At first it seemed that checkpointing Linux might be slow and only good for very slow proof-of-concept demonstrations. However, SBUML has turned out to be much faster and useful than expected. It can save and restore snapshots in a few seconds and compress snapshots down to sizes that are practical to download quickly over the Internet.
This research provides a foundation for several useful and challenging research directions. For the short term, most practical benefit will be using SBUML for its Snapshot Documentation potential. SBUML is currently being distributed publicly and has great potential to be useful to researchers and other users. For advanced computer science research, the model checking applications of SBUML show great potential. For ground breaking research, it will be interesting to transfer the extended functionality demonstrated in SBDebug to SBUML and show how the testing and automatic programming tools can work for popular languages like C and Java.

6. 研究総括の見解

ポッター研究者は、プログラムの実行状態の表示、保存、復帰、途中状態からの実行再開などを行うツールがプログラムの理解や開発に有効であるという考えにもとづき、コンピュテーションスクラップブックというツールの研究を行った。この原理自身は特に新しいものではないが、構築されたツールは非常に優れ、有効なものである。ポッター研究者のプログラミングの能力とセンスの良さ、ツール設計

の適切さによるものである。開発された2つのツールのうち、Linuxオペレーティングシステム上のプログラム開発を対象にしたSBUMLは特にそのアイディアの斬新さや適用範囲の広さから期待の大きいツールである。ネットワークプログラミングやシステムプログラミングにとって非常に強力なツールになり得ると考える。

7. 主な論文等

論文

1. O. Sato, R. Potter. M. Yamamoto, and M. Hagiya, UML スクラップブックとスナップショットプログラミング環境の実現, Linux Conference 2003

2. R. Potter and Y. Harada, Additional Context for Gentle-Slope Systems, IEEE Symposia on Human-Centric Computing Languages and Environments (HCC 2003)

3. R. Potter and M. Hagiya, Computation Scrapbooks for Software Evolution, Fifth International Workshop on Principles of Software Evolution (IWPSE 2002)

4. R. Potter, Computation Scrapbooks of Emacs Lisp Runtime State, 第43回プログラミング・シンポジウム

5. R. Potter, Computation Scrapbooks of Emacs Lisp Runtime State, IEEE Symposia on Human-Centric Computing Languages and Environments (HCC 2001)

6. R. Potter, Computation Scrapbooks, 第4回プログラミングおよび応用システムに関するワークショップ (SPA2001)

ソフトウェア

1. SBDebug: A Computation Scrapbook for Emacs Lisp. (Approximately 8500 lines of Lisp)

2. SBUML: A Computation Scrapbook for User-Mode Linux. (Approximately 5500 lines of C and 1800 lines of Shell Script)