性能に関して健全なコンパイラの構築に向けて

南出　靖彦

1. 研究のねらい

コンパイラは、プログラムをコンピュータ上で直接実行できる機械語とよばれる言語に翻訳するプログラムで、ソフトウェアの作成には必要不可欠である。コンパイラがプログラムを正しく翻訳しないと、プログラムは誤った動作をすることになり、大きな問題になる。そのため、これまでのコンパイラの研究では、高性能のコンパイラ(高性能なプログラムを生成するコンパイラ)の研究とともに、コンパイラで行われる翻訳の健全性(正しさ)の研究が行われてきた。プログラムがどのような動作をすべきか(プログラムの意味)は、自然言語や数学的な手法で定義されている。コンパイラは、プログラムをこの意味に従って動作するプログラムに翻訳する必要がある。これまでのコンパイラの研究では、健全性としてプログラムの実行結果がプログラムの意味で定義された結果になることを示していた。

しかし、プログラムが期待したような動作するためには、結果が正しいだけでは不十分である。実行時間や必要な記憶領域の大きさなどプログラムの性能に関してプログラムから期待される性質を保存する必要がある。例えば、一定の大きさの記憶領域で動作し続けるはずのプログラムが、実行時間の経過とともに必要な記憶領域が大きくなる実行プログラムに翻訳されてしまったとすると、意図したようにずっと動作し続けるのではなく、いつかは記憶領域が足りなくなって実行が止まってしまう。このようにコンパイラが性能に関してプログラムを健全に(正しく)翻訳することが非常に重要である。

これまでの研究では、コンパイラの性能に関する性質は、ベンチマーク、すなわち、幾つかのプログラムに関して生成される実行コードの性能を調べることで議論されてきた。ベンチマークでは、すべてのプログラムに関してコンパイラが性能に関して健全であるか示すことはできず、実際に、ベンチマークでは良い結果を示したコンパイルの手法が、数年後に、ある条件を満たすプログラムの性能を非常に悪くしてしまうことがわかった事例もある。

このような状況を考え、本研究では、プログラムの性能を理論的・数学的に扱い、性能に関して健全なコンパイラを構築する基礎をつくることを目指した。これまでのプログラム意味ではプログラムの性能に関する議論を考慮していなかったので、まず、プログラムの性能を数学的に定義する意味論を考える必要がある。実行時間やスタック空間の大きさなどのプログラムの性能は比較的単純にモデル化できるが、ごみ集めでヒープ空間を管理するプログラミング言語における必要なヒープ空間の大きさなど単純にモデル化するのが難しい性能もある。プログラミング言語の意味の定義の仕方により、性能に関する健全性の証明の難しさが大きく変わってくるので、健全性の証明に適した意味論を構築することを目指した。

さらに、性能を定義した意味論の基で健全性を証明する手法を明らかにする。コンパイラは非常に大きなソフトウェアなので、全体の健全性を一度に示すことは非常に困難である。コンパイラは、プログラムをより低レベルなプログラムに翻訳するプログラム変換を幾つか組み合わせてできていると考えられるので、健全性の証明に関しては、プログラム変換の性能に関する健全性を示すことを目指した。性能に関する健全性を証明したプログラム変換を組み合わせてコンパイラを構築することで、全体として性能に関して

健全なコンパイラが構築できる。これまでに、プログラム変換の性能を考慮しない正しさについては、証明法が整理されており、新しいプログラム変換を考えたときにも、既にある証明法に基づき証明が比較的容易にできる。性能に関する健全性の証明についても、幾つかのプログラム変換の健全性を証明するなかで、健全性の証明法を明らかにしたい。

このようなプログラムの性能を扱った理論的・形式的な議論は、これまでの単純に実行結果だけを考慮した議論に比べ非常に複雑なものになる。そこで、本研究では性能を扱う意味論の形式化とプログラム変換の健全性の証明をコンピュータのプログラムとして実現されている定理証明システムで行うことも目指した。ここで、定理証明システムに適したプログラムの意味、プログラム変換の形式化を明らかにすることで、今後、より複雑になっていくプログラミング言語、コンパイラの検証が可能になると考えた。

2. 研究成果

本研究では、コンパイラの性能に関する健全性を議論するためにプログラムの意味論と健全性の基準について、まず、研究を行った。その意味論の上で幾つかのプログラム変換について実行時間や必要な記憶領域について健全性を証明し、健全性の証明法を明らかにした。さらに、プログラムの性能に関する性質を型情報として推論し、その情報に基づきプログラムをコンパイルする手法を提案した。また、プログラム変換の正当性の証明を定理証明システムで行う場合の幾つかの問題点を明らかにし、その解決策を示した。以下にそれぞれの研究の成果を述べる。

2. 1. プログラムの意味論と健全性の基準

プログラミング言語に対しては、その言語のプログラムがどのように動作すべきかを定めた意味が自然言語や数学的な手法で定められている。しかし、プログラムがどのような性能を持つべきかについては、明示的に決められていることは少ない。本研究では、これまでに考えられてきたプログラミング言語の操作的意味論を比較的単純に拡張することで、実行時間や必要なスタック空間の大きさを定める意味を構築することができた。必要なスタック空間を定める意味論を考える場合には、自然な定義として 2 種類の定義が考えられた。プログラムをインタープリタで実行したときに必要なスタック空間の大きさをモデル化した意味論と、プログラムをコンパイラで翻訳してから実行したときに必要なスタック空間の大きさをモデル化した意味論が考えられた。このように複数の意味が考えられる場合があるが、ある意味で対応していることを示すことができた。

次に、関数型プログラミング言語や Java など、ごみ集めによってヒープ領域を管理する言語のプログラムに対して、必要なヒープ領域の大きさを定義する意味について研究を行った。ヒープ領域の大きさを考える場合は、ヒープ上でどのように値が共有されているかを考える必要があり、定義が複雑になり、共有を考えないで議論ができる実行時間やスタック空間などのように単純に意味を定義できない。本研究では、ヒープ空間をアドレスから値への関数と考えることで共有を考慮する幾つかの定義を考えた。しかし、定義の方法によって健全性の証明の難しさが大きく変わり、まだ、決定版と呼べる意味の定義は得られていない。

性能を定義する意味論は、ある程度、抽象的であることが望ましい。CPU などの実行環境が変われば プログラムの実行時間や必要な空間の大きさが変わってくるのは当然である。よって性能を定義する場合

には、定数倍を無視して考えるのが自然である。また、ヒープ領域に関する意味論では、ごみ集めの手法に依存しないように意味論を構築した。

このように性能を数学的に定義した意味が定義できても、どのような基準を満たせばコンパイラが性能について健全であるといえるのかは明らかではない。 例えば、以下の二つの基準が考えられる。

1. プログラムの計算量を保存する。
2. プログラムの性能を定数倍の範囲で保存する。

1の基準が考えた中では最も弱い基準で、コンパイラは必ずこの性質は満たす必要がある。例えば、入力の大きさに比例した時間で実行できるプログラムを大きさの二乗に比例した時間が必要なプログラムに変換してはならない。しかし、この基準は弱すぎると考えられる。例えば、入力の集合が有限であるプログラムに関しては全く制限を与えていない。一方、2 の基準が最も強い基準である。意味でも定数倍を無視していたので、安全性の基準でも定数倍は無視しないと整合性がとれない。コンパイラで用いられるプログラム変換に対して健全性を考えた場合、健全なプログラム変換の合成が、また、健全になることが望ましい。上に挙げた二つの基準は、この合成に関する性質を満たす。

このような基準を考えたがコンパイラで用いられるプログラム変換には、2の基準を満たさないものがある。プログラム変換の中には、性能に与える影響がプログラムの大きさ(実際には関数の大きさ)に依存するものがある。このようなプログラム変換を許す場合には、下の基準を考えることになる。

- プログラムの性能の悪化の割合が、プログラムの大きさに関する多項式で抑えられる。

このように、コンパイラの性能に関する健全性の基準としては、幾つかの性質を考えることができる。その中でどの基準が適当であるかは、プログラミング言語や議論する性能の種類(実行時間や記憶領域)などに依存すると考えられる。

2．2．健全性の証明法

プログラム変換の正当性の証明及びプログラミング言語の型システムの健全性の証明に用いられる証明法が、プログラム変換の性能に関する健全性の証明法に拡張できることを示した。証明法としては、計算の長さに関する帰納法、プログラムの型の導出に関する帰納法を用いることができることを示した。

2．2．1．型の導出に関する帰納法による証明

型付ラムダ計算の強正規化定理の証明ために導入された論理関係と呼ばれる型でインデックスされた関係を用いた証明法を実行時間及びスタック空間に関する健全性の証明に応用できることを示した。論理関係とは以下のような性質を持つ関係である。

- 型 T1 から型 T2 への関数F1 とF2 が関係 $R_{T1\to T2}$を満たす⇔関係R_{T1}を満たす値 V1 と V2 に対して F1(V1)と F2(V2)が関係R_{T2}を満たす。

F1,V1 は変換前の値、F2,V2 は変換後の値とする。このような関係がすべてのプログラムとその変換に対して成り立つことをプログラムの型の導出に関する帰納法で示すことができる。それによってプログラム変換の正しさを証明することができる。この方法を拡張して性能に関する健全性を証明できることを示した。例えば、あるプログラム変換が実行時間を 2 倍以上延ばすことがないことを示すためには、以下のように関係に拡張すればよい。

- 型 T1 から型 T2 への関数 F1 と F2 が性質 $R_{T1\to T2}$を満たす⇔R_{T1}を満たす値 V1 と V2 に対してF1(V1)の計算が n ステップで終わるならば、F2(V2)の計算は、2×nステップ以内で終わり、それぞれの

結果は、R_{T2}で関係している。

この方法で性能に関する健全性を証明したプログラム変換としては、以下の二つのプログラム変換がある。

- 多相型を効率的に実現するために値の表現の変換を挿入する変換。実行時間に関する健全性。
- 部分型の実現のために値の表現の変換を挿入する変換。実行時間及びスタック空間に関する健全性。

2. 2. 2. 計算の長さに関する帰納法による証明

ごみ集めよってヒープ領域を管理するプログラミング言語を考えた場合、必要なヒープ領域の大きさに関するコンパイラの健全性を証明するためには、型の導出に関する帰納法を用いることはできなかった。そこで、本研究では、計算の長さに関する帰納法によってヒープ領域に関する健全性を示した。関数型プログラミング言語のコンパイラで用いられる CPS 変換と呼ばれるプログラム変換が、必要なヒープ領域の大きさを保存することを示した。この研究で参照型を持つプログラミング言語の型システムの健全性の証明法とヒープ領域に関するプログラム変換の健全性の証明法が非常に関連したものであることがわかった。

2. 3. 型情報に基づくコンパイル法

最悪の場合の性能を保証するようにコンパイラを作ると高性能な実行プログラムを生成するのが困難になる。この問題の一つの解決策として、プログラムの性能に関する情報をプログラムの型として推論し、その情報に基づきプログラムをコンパイルする方法を提案した。

関数型プログラミング言語では、関数の末尾呼び出しはスタックを消費しないように実現する必要がある。下のプログラムで、sum の呼び出しは、関数の中で行う最後の処理なので末尾呼び出しと呼ばれる。この関数 sum のように関数型プログラミング言語では、ループを末尾呼び出しを用いて表すことが多いので、末尾呼び出しをスタック空間を消費しないように実現することが重要になる。

```
fun sum (x, a) = if x = 0 then a else sum (x-1, a+x)
```

しかし、関数型プログラミング言語を Java 仮想機械など末尾呼び出しをサポートしていない環境にコンパイルする場合は、末尾呼び出しの実現には特殊な処理が必要となり効率的な実装ができない。そこで、末尾呼び出しの正しい実装が必要な関数を型推論で判定し、必要な場合にだけ特殊な処理をする方法を提案した。そのために、int $\to^3$ int や int $\to^\omega$ int など関数の型に性能の情報が付加されている型の体系を考えた。末尾呼び出しの効率的な実現のためには、この情報でその関数を呼び出したときに何回末尾呼び出しが連続するかを表す。情報 ω は、何回連続するか有限の値で制限できないことを表している。例えば、下のプログラムを考えてみる。

```
fun f x = 0
fun g x = f x
fun h x = if ... then h (x - 1) else g x
```

- 関数 f は、その中で関数呼び出しを全く行っていないので、型は、int $\to^0$ int となる。
- 関数 g は、末尾呼び出しで関数 f を呼び出しているので、型は、int $\to^1$ int となる。
- 関数 h は、再帰的に関数 h を呼び出しているので、末尾呼び出しの回数を制限することはできないので、型 int $\to^\omega$ int となる。

この情報から関数 f,g の末尾呼び出しは、スタックを消費しても安全であることがわかり、効率的な通常の関数呼び出しとして実現できる。一方、関数 h の呼び出しには特殊な処理が必要なる。この方法を関数型プログラミング言語MLからJavaのバイトコードへのコンパイラに実装し、末尾呼び出しを効率的かつ健全に実現できることを示した。

2．4．定理証明システムによる検証

プログラミング言語に関して形式的、理論的な研究を行う場合には、人が紙の上で行う議論にはインフォーマルな部分が残る。また、証明が冗長で複雑なものになれば、証明に誤りが生じることも多い。プログラミング言語とそのコンパイルの手法は、近年、非常に複雑なものになってきており、紙の上での議論は不十分ではないかと考えた。そこで、関数型プログラミング言語のコンパイラで用いられるCPS変換と呼ばれるプログラム変換の検証を Isabelle/HOL と呼ばれる定理証明システムで行った。Isabelle/HOL では、部分的には自動的な証明を行うことができるが、証明の多くの部分は人が書く必要がある。完成した証明に誤りがないことは保障される。

CPS 変換には、幾つかの種類があり、その中で、本研究では、最も単純な Plotkin による変換と、実際にコンパイラで用いられる変換に近い Danvy と Filinski による変換の検証を行った。CPS 変換は、その正当性の証明が比較的難しいと考えられているプログラム変換である。また、Danvy と Filinski の変換はその定義に高階関数が用いられており、定理証明システムによって単純に形式化することができない。これらの検証で、変換によって導入される変数の扱いとプログラムのアルファ同値の扱いが問題になることがわかった。これらについて定理証明システムによる検証に適した形式化を見つけ、検証を行った。

コンパイラで用いられるほとんどのプログラム変換は、プログラムに新しい変数を導入する。例えば、プログラム P がプログラム Q に変換されるときに変数 x が導入されるとする。この変数 x が元のプログラム P で使われていると正しい変換にならない。そこでプログラム変換の正しさを形式化すると下のようになる。

- 変換で導入される変数 x がプログラム P で使われていなければ、プログラム P とプログラム Q は同じ値を計算する。

このように変数に関する条件を性質に明示的に付加する必要があり証明が非常に複雑になる。この問題をプログラムの変数を表す型に関して多相的な以下のようなデータ型でプログラミング言語を形式化することで解決した。

```
datatype 'a term = Var 'a
                 | Abs 'a "'a term"
                 | App "'a term" "'a term"
```

これは、文法 M = x | λx.M | M M で表される言語をデータ型として形式化したものになっており、変数を表す型が型変数'a となっており、多相的になっている。

プログラムのアルファ同値に関しては、通常の形式化とは異なる形式化を採用した。名前を付け替えることによって同じプログラムに変換できるプログラムをアルファ同値と呼ぶ。プログラミング言語のモデルとして広く用いられているラムダ計算では、アルファ同値は以下の公理によって形式化されている。

$$\lambda x.\, M = \lambda y.\, M[y/x]$$

ここで、λx. M は仮引数が変数 x で M で表される計算を行う関数を表しており、 M[y/x]は式の M 中の変数 x を変数 y で置き換えたものを表している。しかし、この公理に基づきアルファ同値を定義すると、 無限個の変数を仮定した推論が必要となり、 定理証明システムで扱いにくく、証明の自動化も行いにくい。

そこで、変数の数が有限の場合も扱うことができるアルファ同値を示すための推論系を定義した。この推論系は、変数の数が無限個の場合には通常のアルファ同値と同等になっており、また、証明の自動化を行うことが容易になっている。この形式化によって、CPS 変換の正当性の証明では、プログラムのアルファ同値を証明する部分をほぼ自動化することができた。

3. 今後の展開

これまでの研究で、コンパイラで用いられる幾つかのプログラム変換の性能に関する健全性を証明することができた。その証明法は、多くのプログラム変換の健全性の証明に応用できるので、今後、証明を整理、一般化し、他のプログラム変換の証明に容易に適用できるように研究を進める。それによって、コンパイルの手法を研究する場合に、性能に関する健全性の証明が広く行われるようになれば、コンパイラの性能に関する健全性が保障できるようになると考えている。

プログラム変換がプログラムの性能に与える影響の議論が難しい場合としては、ごみ集めでヒープ空間を管理する言語のコンパイラのヒープ空間に関する健全性の議論がある。本研究では、関数型プログラミング言語で用いられるプログラム変換のヒープ空間に関する健全性を証明したが、近年、Java やC#などごみ集めによってヒープ空間を管理する言語が多くなってきているので、これらの言語のコンパイラについて、今後、ヒープ空間に関する健全性の議論が必要になってくるのではないかと思っている。

また、定理証明システムを用いた形式化については、最終的には、コンパイラ全体の形式化を目指している。また、定理証明システムによる形式化から実行可能なプログラムを生成する研究が行われているので、さらに、検証した形式化から健全性が保障された実行可能なコンパイラを生成することを考えている。しかし、実用的なコンパイラをこのような方式で構築するには、まだまだ、様々な面で研究が必要になると考えている。

4. 成果リスト

1. 大熊浩示，南出靖彦、定理証明システムを用いた CPS 変換の正当性の検証、ソフトウェア科学会第 17 回大会、2000.
2. Yasuhiko Minamide, A New Criterion for Safe Program Transformations, In Proceedings of the Forth International Workshop on Higher Order Operational Techniques in Semantics (HOOTS), ENTCS 41, 2000.
3. Yasuhiko Minamide, Runtime Behavior of Conversion Interpretation of Subtyping, In Proceedings of the 13th International Workshop on the Implementation of Functional Languages, LNCS 2312, pages 155–167, 2001.
4. Yasuhiko Minamide, Selective Tail Call Elimination, In Proceedings of the 14th International Workshop on Implementation of Functional Languages, 2002.