ICOT Technical Report: TR-0807

TR-0807

論理プログラムの抽象解釈を用いた解析

堀内　謙二

September, 1992

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

論理プログラムの抽象解釈を用いた解析

堀内　謙二

(財) 新世代コンピュータ技術開発機構*

概要

抽象解釈はプログラムの様々な特性を解析するための手法を統一するものとして提案された理論的な枠組である．70年代後半にCousot達によって提案されて以来，様々な言語に対する様々な解析の応用例が研究されている．論理プログラムに限ってみても，そのバックグラウンドにあたる意味論や抽象化される領域などによって，様々な分類をすることができる．しかし，ほとんどすべてのアプローチを「プログラムの意味を不動点で与え，それを抽象化する」という同じ範疇に入るものと見ることができる．

本稿では，論理プログラムを対象とした場合のさまざまな抽象解釈の手法について解説し，その根本的な原理が不動点計算の抽象化であることを示す．

1　はじめに

プログラムの解析がプログラミングやその支援ツールにおいて重要な役割を果たしているのは，いまさら敢えて言うまでもないことである．解析された情報は，デバッガーやタイプチェッカーなどにおいてプログラムのエラーを発見するのに，また，コンパイラーやプログラムの自動変換システムなどにおいて様々な最適化を行なうのに役立つ．しかし，最適化に役立つ情報が解析できたとしてもそれが正しい情報でなければ当然ながら意味のないものである．

1977年にCousot達[10]によって抽象解釈(*abstract interpretation*)と呼ばれる理論が提案された．当時，様々なプログラム解析に対する理論的形式化は各解析応用毎にad hocに研究されていたが，彼らはそれらのプログラム解析を統一的に扱うような理論的枠組を示した．その後，関数型言語を中心に抽象解釈の研究は様々な方向に発展を見るが，論理プログラムの世界に抽象解釈という言葉がが持ち込まれたのは意外に新しく，文献[37]や文献[20]において提案されているのが最初である．

抽象解釈の基本的なアイデアは，実際にプログラムを実行するのではなく，近似的な実行を行い，その近似実行の過程からプログラムの特性を得ることにある．近似実行の一つの利点は実行時間が短縮されることであり，極端な場合には，実際には停止しないかも知れないプログラムの実行でも有限時間で停止させることができる．しかしその反面(当然のことであるが)，実行を近似的に行なうために正確さを欠くことになる．近似によって失われた情報が，実は，解析したい特性に大きく関与する部分だった場合には，必要な特性が解析できないかもしれない．これらの

Analysis of Logic Programs Based on Abstract Interpretation,

Kenji Horiuchi, Institute for New Generation Computer Technology, E-mail: horiuchi@icot.or.jp.

本稿は日本ソフトウェア科学会会誌「コンピュータソフトウェア」に掲載が予定されているものを許可を得て転載したものである．

*平成4年10月より三菱電機中央研究所．E-mail: horiuchi@sys.crl.melco.co.jp.

$$plus(zero, Y, Y). \quad (1)$$
$$plus(suc(X), Y, suc(Z)) \leftarrow plus(X, Y, Z). \quad (2)$$

図 1: プログラム *plus*

トレードオフを考え，近似の度合を決定しなければならない．近似されるものの種類や近似の度合いに依存して，どのような特性が解析されるかは変化するが，近似実行を行なうための基本的な枠組は同じものが使える．この統一性が，抽象解釈の大きな武器である．以下では，この“近似実行 (*approximate execution*)”のことを“抽象解釈 (*abstract interpretation*)”と呼び，“近似 (*approximation*)”を“抽象化 (*abstraction*)”と呼ぶことにする．

以下，まず 2 節において，簡単な実例を用いて論理プログラムの抽象解釈について説明する．3 節では，実際提案されている抽象解釈の枠組をいくつか紹介する．4 節では，抽象解釈と不動点意味論の関係を述べ，抽象解釈の安全性についてもふれる．最後に 5 節で，実際にどのような特性の解析に利用されているかを簡単に紹介する．

2 抽象解釈の直感的な説明

ここでは，文献 [44] で用いられた例と同様の例を用いて論理プログラムの抽象解釈について簡単に説明する．

2.1 論理プログラムの通常の実行

まず，論理プログラムとその動作について簡単に説明する．ここで扱う論理プログラムは，バックトラック探索の一部削除 (カット) や副作用を伴う述語 (assert,retract などのデータベース操作述語) を含まない，一般には，純論理プログラム，または，pure Prolog と呼ばれる言語である．

まず，図 1 にプログラム *plus* を示す．*zero* は定数記号で *suc* は引数を 1 つ持つ関数記号である．プログラムは確定節 (*definite clause*) と呼ばれる“$H \leftarrow B$”の形の式の集合からなり，各節の $\leftarrow$ の左側を頭部と呼び，その右側を本体と呼ぶ．本体は節 (1) のように空の場合もあり，そのような節を単位節 (*unit clause*) と呼ぶ．また，以後，確定節を単に節と呼ぶこともある．

zero を整数の 0 を表す定数，*suc* を“引数 +1”を返す関数と見なすことによって，このプログラム *plus* は自然数上の加算のプログラムと見ることができる．以下では，*zero* に *suc* を n 回に施した項を $suc^n(zero)$ と記述し，整数の n と見なすことにする．また，*zero* を $suc^0(zero)$ と記述することもある．

このプログラムにおいて，例えば，

$$\leftarrow plus(suc(suc(zero)), suc(suc(zero)), A)$$

というゴールは成功し，

$$A = suc(suc(suc(suc(zero))))$$

という結果を返す．これは，“2 + 2”という計算を実行し，その結果として“4”を返している．では次に，以下のようなゴールに対してはどうだろう?

$$\leftarrow plus(A, B, suc(suc(zero)))$$

このゴールもまた成功し，変数 A, B の値に関しては，

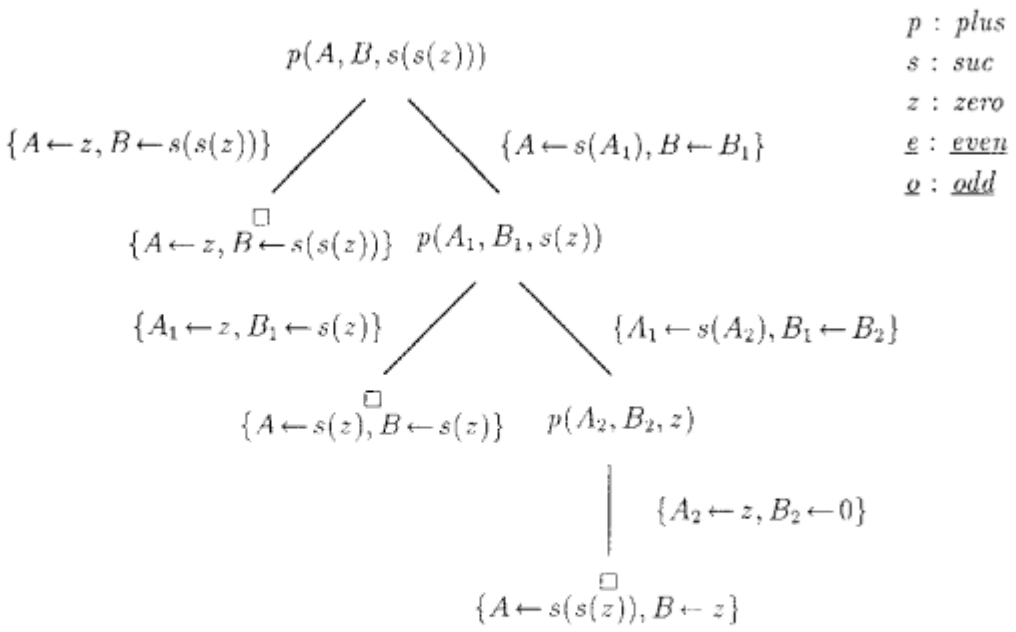

図 2: ゴール $plus(A, B, suc(suc(zero)))$ の実行

$$A = zero, B = suc(suc(zero))$$
$$A = suc(zero), B = suc(zero)$$
$$A = suc(suc(zero)), B = zero$$

という 3 種類の結果を返す．このゴールは，“$A + B = 2$” なる自然数 A, B をを求めるためのゴールである．

通常の論理プログラムの処理系では，ゴールで実行が起動され，そのゴールのアトムと unify 可能な頭部をもつ節がプログラムから選ばれ，そのアトムがその節の本体に置き換えられることによって実行が進む．ゴールにアトムがなくなれば実行は終了する．アトムのないこのようなゴールを空節と呼び，□で表す．また，ゴールが呼び出されてその実行が終った時返される結果を解 (または，解代入) と呼ぶ．A, B の値に関する上記の 3 種類の結果は，すべて，ゴール

$$\leftarrow plus(A, B, suc(suc(zero)))$$

の解である．論理プログラムの実行の最終的な目的はそのゴールが成功するかどうかと成功した場合の解を求めることである．図 2は，上記のゴールに対する実行の様子を木構造で表現している．各枝の横の代入はそのステップの unification の結果を表し，空節の下の代入は実行がその空節に至った時の解代入を表している．

ここでは論理プログラムの実行メカニズムや操作的意味論について詳細に説明することはせず，直感的な説明にとどめた．これらに関して詳しく知りたい読者は論理プログラムに関する教科書を参照されたい (例えば，文献 [30] など)．

2.2 領域の抽象化

何を抽象化するかという問題もあるが，ここでは論理プログラムの抽象解釈においてもっとも一般的である計算領域の抽象化を考えることにする．

2.1 節の実行例からも分かるように論理プログラムの実行はゴールによって起動され，各アトムが引数としてとり得る値は項である．(項を，以下の議論で出てくる抽象項と区別して具体項

表 1: 偶数 / 奇数判定のための抽象項

抽象項	対応する具体項の集合
$\top$	すべての項の集合
$\underline{even}$	$\{ suc^n(zero) \mid n$は偶数$\}$
$\underline{odd}$	$\{ suc^n(zero) \mid n$は奇数$\}$
$\bot$	空集合

表 2: 抽象領域上の unification 例

具体項	抽象項	結果
$suc(X)$	$\underline{even}$	$\{ X \leftarrow \underline{odd} \}$
$suc(X)$	$\underline{odd}$	$\{ X \leftarrow \underline{even} \}$
$suc(X)$	$\top$	$\{ X \leftarrow \top \}$
0	$\underline{even}$	成功
0	$\underline{odd}$	失敗

図 3: 抽象領域の束構造

と呼ぶこともある.) ここで, 項とは定数記号, 関数記号, 変数からなる構造である. 通常は, すべての項の集合を引数のとり得る領域としてプログラムは動作する. この集合を特に具体領域 (*concrete domain*) と呼ぶことにする. 抽象解釈は, その具体領域の部分集合を表現するようなある種の記号を要素とするような有限集合上で行なわれる. この集合を抽象領域 (*abstract domain*) と呼ぶ. この場合, 具体領域の各要素は項であるが, 抽象領域の各要素は, 解析したい特性によって, データタイプであったり, データフロー情報であったする.

ここでは文献 [44] と同様, 偶数 / 奇数判定のための抽象領域を考えてみよう. ただし, その構成要素は若干簡単化, 変更されている. 抽象領域の各要素 (抽象項と呼ぶ) は具体領域の部分集合を表現しており, それぞれの対応を表 1に示す. この抽象領域は対応する具体領域上の集合の包含関係 $\subset$ に関して図 3 のような完備束を形成している.

2.3 論理プログラムの抽象領域上での実行

論理プログラムの実行過程の基本操作は, ゴールのアトムと節の頭部との unification である. 前節で導入された抽象領域上で論理プログラムの実行を行なうためには, まず, 抽象領域上の unification を定義しなければならない. 例えば, ゴール

$$plus(A, B, suc(suc(zero)))$$

に対応する抽象領域上のゴールは

$$plus(\top, \top, \underline{even})$$

である. この抽象領域上のゴールは,

$$\{ plus(t_1, t_2, t_3) \mid t_1 \in \gamma(\top) \wedge t_2 \in \gamma(\top) \wedge t_3 \in \gamma(\underline{even}) \}$$

という具体領域上のゴールの集合を表している. ここで, γ は抽象項を引数にとり対応する具体項の集合を返す関数である (γ に関しては, 4.2節で再びより詳しく述べる). この時, 元のゴール $plus(A, B, suc(suc(zero)))$ はこの集合に含まれている. このような抽象領域上のゴールを抽象化ゴールと呼び, 後の処理の都合上,

$$plus(A{:}\top, B{:}\top, C{:}even)$$

のように，各引数が変数と抽象項の対であるようなゴールで表現する場合もある．また，

$$\{ A \leftarrow \top, B \leftarrow \top, C \leftarrow \underline{even} \}$$

のような抽象領域要素の代入を抽象代入と呼ぶ．

抽象領域上の実行ではこの抽象化ゴールと図 1 で示した *plus* のプログラムの各節の頭部との unification が行なわれる．すなわち，各引数において，具体項と抽象項との unification が行なわれる．表 2 は，この時の基本となるいくつかの具体項と抽象項との unification の結果の例を示している．例えば，表 2 の最初の行の意味は，具体項 $suc(X)$ と抽象項 $\underline{even}$ の unification は成功し，その結果 (抽象領域上の)unifier として $\{ X \leftarrow odd \}$ が得られるということである．これらのことから，図 1の節 (2) の変数を適当に renaming した節

$$plus(suc(A_1), B_1, suc(C_1)) \leftarrow plus(A_1, B_1, C_1)$$

の頭部である $plus(suc(A_1), B_1, suc(C_1))$ と抽象化ゴール $plus(A{:}\top, B{:}\top, C{:}even)$ の unification は成功し，その unifier は以下のような抽象代入である．

$$\{ A_1 \leftarrow \top, B_1 \leftarrow \top, C_1 \leftarrow \underline{odd} \}$$

抽象領域とその上での unification が定まったところで，実際に抽象化ゴールを抽象領域上で実行してみる．通常の論理プログラムの実行と同様，節の頭部との unification によって得られた抽象代入をその節の本体に施す．上記例では，抽象代入 $\{ A_1 \leftarrow \top, B_1 \leftarrow \top, C_1 \leftarrow \underline{odd} \}$ を節の本体 $plus(A_1, B_1, C_1)$ に施すことにより新しい抽象化 (サブ) ゴール

$$plus(A_1{:}\top, B_1{:}\top, C_1{:}\underline{odd})$$

を得る．この抽象化 (サブ) ゴールは，通常の実行時における呼び出し (サブ) ゴールを抽象化したものに対応している．すなわち，第 3 引数が偶数であるような述語 *plus* によるゴールが通常実行された時，そのサブゴールとして必ず第 3 引数が奇数であるようなゴールを呼び出すということが解析できたわけである．これは，プログラムの実行時の呼び出しゴールの引数の解析に相当する．

次に，計算結果すなわち解代入の解析について考える．解代人に関する特性の解析を行なうためには，抽象領域上の実行の終了時にも解代入の抽象化に対応する結果を返す必要がある．

図 4を見てみよう．これは，抽象化ゴール

$$plus(A{:}\top, B{:}\top, C{:}\underline{even})$$

を G_0 とした時，抽象領域上での実行の様子を木構造で表現している．G_0 は，*plus* のプログラムの節 (1) の renaming である

$$plus(zero, B', B')$$

とも unify 可能である．節 (1) は単位節なのでこの実行経路に関して抽象領域上での実行は終了する．この時，$plus(A, B, C)$ と $plus(zero, B', B')$ の unifier である $\{ A \leftarrow zero, B \leftarrow C \}$ から，

$$\{ A \leftarrow even, B \leftarrow \underline{even}, C \leftarrow \underline{even} \}$$

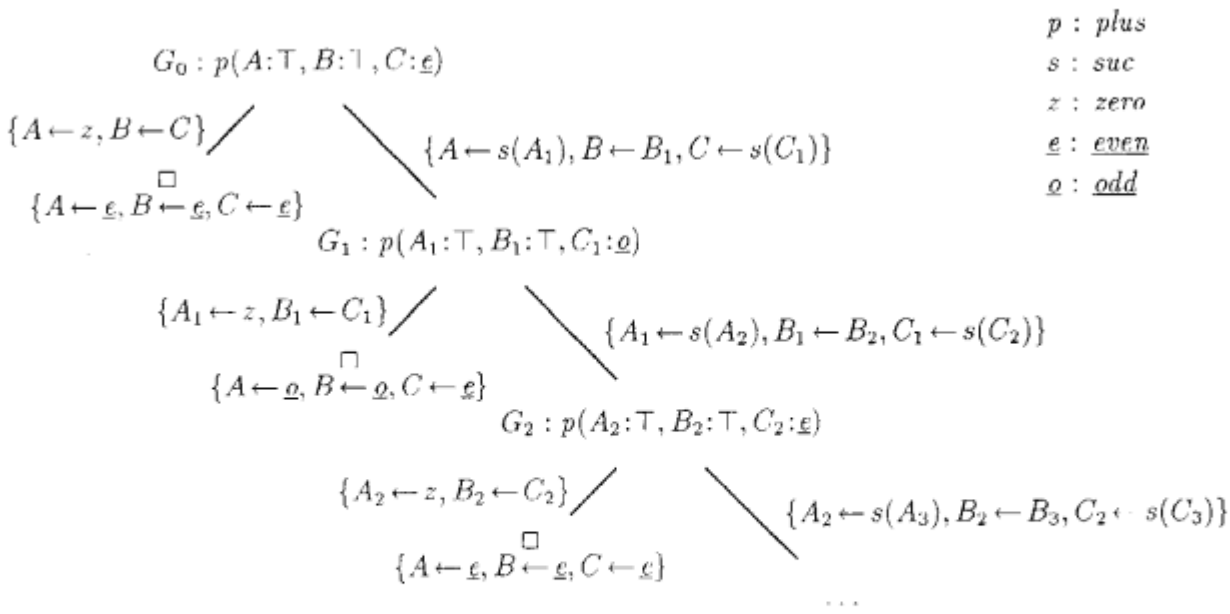

図 4: 抽象化ゴール $plus(A{:}\top, B{:}\top, C{:}\underline{even})$ の抽象解釈

という抽象領域上での実行における一つの解代入が得られる.

図 4 においても，各枝の横の (具体領域上の) 代入はそのステップで行なわれた unification の結果を表し，空節の下の抽象代入は G_0 からそこに至る抽象領域上での実行において得られた解代入を表している．この抽象代入は，通常の実行時の解代入を抽象化したものに対応している．すなわち，第3引数が偶数であるような述語 $plus$ によるゴールが実行され，成功した場合，第1,第2引数の値は，共に奇数同士，または，偶数同士であるということが分かる．このことは，ゴールの実行結果の解析を行なったことになる.

以上，非常に簡単な例を用いて抽象領域上の論理プログラムの実行について説明した．まず最初に，どのように抽象解釈が進むかを直感的に掴んで頂きたかったので，図 4で示した抽象領域上での実行過程については，すこし厳密さを欠いている．例えば，上記の通り通常の実行と同様に抽象領域上でも実行を進めると，図 4の最右の実行経路において無限ループに陥る．しかし，抽象領域が有限であることから (この例では，4 種類の抽象項からなる集合)，抽象領域上での呼び出しゴールや解代入の種類は有限である[1]．そこで，抽象領域上での実行メカニズムにすこし手を加えることによってこの種の無限ループを避けることが必要になる．次節ではそれらの実行メカニズムについて述べる.

3 論理プログラムの抽象インタプリタ

2 節の最後で触れたように，通常の top-down 実行 (OLD 導出[2]) を naïve に抽象化しただけでは無限ループに陥ってしまう場合がある．これは，一般に元々の解析したいプログラムが左再帰的 (*left recursive*) といった特殊なプログラムでなくとも，抽象領域上で通常の top-down 実行を

[1] $plus(suc(A{:}\top), B{:}\top, suc(C{:}\underline{even}))$ のように，抽象化ゴールとして引数の項の部分に変数でない項を許すような場合は，抽象化ゴールの種類は一般には無限に存在する．そのような場合でも，引数の項の部分を，プログラムに現れる項に限定したり，項の複雑さを限定したりすることにより，抽象化ゴールの種類を有限に押えることができる.

[2] top-down 実行に基づいた枠組のほとんどは，通常の論理プログラムの実行戦略を考慮してゴール内のアトムの選択規則が left-to-right に固定されている場合を考えており，その基本となる導出原理は，Selective Linear Definite Clause 導出ではなく，Ordered Linear Definite Clause 導出 [49] である.

行なうとしばしば起こることである．しかし，抽象解釈の目的はプログラム解析にあるので，抽象解釈自体は停止することが望ましい．

ここでは，現在までに研究されている代表的な論理プログラムの抽象解釈の枠組について，その基礎となる実行メカニズムに注目して紹介し，その枠組の特徴について述べる．

3.1 Bottom-up インタプリタを用いるアプローチ

ここでは，無限ループを避けるための実行メカニズムとして最も単純である bottom-up 実行 (いわゆる，Hyper 導出) を用いた枠組について紹介する．まず，通常の bottom-up 実行について説明し，その後，抽象領域上の bottom-up 実行について述べる．

bottom-up 実行はプログラムのすべての単位節を解アトムの集合 ($AnsSet$ とし，解アトム集合と呼ぶ) として見なすことにより，実行が起動される．それらの解アトム集合 $AnsSet$ に含まれるアトムと unify 可能なアトムを本体に持つ節がプログラムから選ばれ，その unification の結果の代入を頭部に適用したアトムを新たに $AnsSet$ に加える．この動作を Hyper 導出と呼ぶ．このようにして Hyper 導出を繰り返し，解アトム集合 $AnsSet$ を順次大きくすることによって実行が進む．ここで，Hyper 導出によって増加する解アトム集合とは，直感的には，それらと unify 可能なゴールを top-down に実行した時にうまくアトムを選択するならば[3]成功するようなアトムの集合である[4]．もし初期ゴールが与えられるなら，成功アトム集合の中にその初期ゴールと unify 可能なアトムが現れた時点で実行を終了すれば良い．

例えば，図 1の $plus$ のプログラムでは，最初は単位節 (1) が成功するアトムの集合 $AnsSet^1_{plus}$ として登録され，次に，順次 Hyper 導出を繰り返すことにより，

$$
\begin{aligned}
AnsSet^1_{plus} &= \{\, plus(zero, Y_1, Y_1)\,\} \\
AnsSet^2_{plus} &= AnsSet^1_{plus} \cup \{plus(suc(zero), Y_2, suc(Y_2))\} \\
AnsSet^3_{plus} &= AnsSet^2_{plus} \cup \{plus(suc(suc(zero)), Y_3, suc(suc(Y_3)))\} \\
AnsSet^4_{plus} &= AnsSet^3_{plus} \cup \{plus(suc(suc(suc(zero))), Y_4, suc(suc(suc(Y_4))))\} \\
&\vdots
\end{aligned}
$$

というように $AnsSet_{plus}$ は増加する．ここで，$AnsSet_{plus}$ の上付の添字は Hyper 導出の回数を表している．

2 節での例のように，初期ゴール

$$\leftarrow plus(A, B, suc(suc(zero)))$$

が与えられたとする．その時，このゴールは $n \geq 3$ のすべて $AnsSet^n_{plus}$ において，3 つのアトムとのみ unify 可能で，その unifer は 2節での通常の (top-down 実行) の結果と同じ，

$$
\begin{aligned}
&A = zero, B = suc(suc(zero)) \\
&A = suc(zero), B = suc(zero) \\
&A = suc(suc(zero)), B = zero
\end{aligned}
$$

[3]常に最左のアトムを選択するような通常の戦略の top-down 実行ではいつも成功終了するとは限らない．

[4]論理プログラムの標準的な意味論に，ここでいう解アトム集合と良く似た概念である成功集合 (*success set*) を用いたものがある [30]．成功集合は有限回の導出で成功するようなすべての変数のないアトム (*ground atom*) の集合であるのに対し，解アトム集合には変数が含まれる場合がある．解アトム集合をより形式的に説明するなら，プログラムを論理式 (Horn 節) の集合 P と見なし，アトム A に含まれるすべての変数が全称記号 (*universal quantifier*) で束縛されているような一階論理式を $\forall.A$ とした時，解アトム集合に含まれる任意のアトム A に対して，P から $\forall.A$ はいつも証明できる，すなわち，$P \vdash \forall.A$ である．

である.

次に，上記の Hyper 導出を 2節の抽象領域と unification の定義を用いてそのまま抽象化してみよう．抽象領域上の実行で成功するアトムの集合を $\underline{AnsSet}_{plus}$ としよう．まず，プログラム *plus* の単位節から求まる抽象化アトムの集合を $\underline{AnsSet}^1_{plus}$ として登録する．ここでは，話しをより簡単にするために抽象化ゴールの各引数を抽象項のみを用いて表現する.

$$\underline{AnsSet}^1_{plus} = \{\, plus(\underline{even}, \top, \top) \,\}$$

次の Hyper 導出では，この抽象化アトムとプログラム *plus* の確定節 (2)

$$plus(suc(X), Y, suc(Z)) \leftarrow plus(X, Y, Z)$$

の本体と unify することにより得られる抽象代入をこの確定節の頭部に適用し得られる抽象化アトムが $\underline{AnsSet}_{plus}$ に新たに加えられる．その結果,

$$\underline{AnsSet}^2_{plus} = \underline{AnsSet}^1_{plus} \cup \{\, plus(\underline{odd}, \top, \top) \,\}$$

が得られる．以下，抽象領域上の Hyper 導出を繰り返すことにより,

$$\begin{aligned} \underline{AnsSet}^3_{plus} &= \underline{AnsSet}^2_{plus} \\ &\vdots \end{aligned}$$

のようになり，$\underline{AnsSet}_{plus}$ は増加するが，Hyper 導出の回数が 2 回で収束するのがわかる．一般に bottom-up 実行 (Hyper 導出) は有限で停止する訳ではないが，抽象領域が有限であるなら，すなわち，抽象領域上のアトムの種類が有限個しかないなら，このように必ず有限回の Hyper 導出で解アトム集合は収束する．そのため停止性という面から見れば，図 4 の top-down 実行をそのまま抽象化した場合と比較して，bottom-up 実行は抽象解釈に適した実行メカニズムであると言える.

また抽象化初期ゴール

$$plus(\top, \top, \underline{even})$$

が与えられたとすると，このゴールと $\underline{AnsSet}^2_{plus}$ の抽象化ゴールを unify することにより,

$$\{\, plus(\underline{even}, \top, \underline{even}), plus(\underline{odd}, \top, \underline{even}) \,\}$$

を得ることができる．ここでは，*plus* のプログラムにおいて成功するアトムは第一引数が偶数かまたは奇数であるとか，または，第三引数が偶数であるようなゴールが成功する時は第一引数は偶数かまたは奇数であるというようなことが判明したわけである．結局ここでは，このように非常に自明な解析結果しか得られなかった訳であるが，実際は，Hyper 導出の各段階で変数の共有情報などをうまく保存し，抽象項の情報を，例えば，第三引数から第二引数へと伝播させるなどにより，より詳細な解析結果を得ることもできる．抽象領域上の bottom-up 実行のより詳しい説明は 4 節で，また，変数の共有情報については 5 節で説明するのでそれまで待って頂きたい.

以上，例を用いて説明したが，Herbrand 解釈などの概念を用いた bottom-up 実行を基礎にした抽象解釈の一般的な枠組が提案されており，そのいくつかを紹介する．Herbrand 領域とはプログラムに現れる定数記号，関数記号からなる変数を含まない項 *(ground term)* の集合である．Herbrand 基底とは述語の引数が Herbrand 領域の要素であるすべてのアトムの集合である．また，Herbrand 基底の部分集合を Herbrand 解釈と呼び，Herbrand 解釈であるようなモデルを Herbrand モデルと呼ぶ．その中で最も小さいものが最小 Herbrand モデルである.

bottom-up の枠組を最初に用いたのは金森達 [23] で， Herbrand 領域をデータタイプとして抽象化し，最小 Herbrand モデルに基づいてタイプ推定を行なった．すなわち，直感的には，上記で説明した bottom-up 実行を最も naïve に抽象化したわけである．

Marriott 達は文献 [33] で文献 [16] の意味論に基づいた bottom-up 型の抽象解釈を提案した．これは，文献 [23] の一般化にあたる．文献 [16] の bottom-up 意味論は ground な成功集合と失敗集合を同時に bottom-up に集めるような意味関数を定義しその最小不動点で本体に負リテラルを含むような論理プログラムに意味を与えるものである．

通常，意味領域は Herbrand 基底であり変数は含まれないが，変数を含むような一般のアトムに Herbrand 基底を拡張した最小不動点意味論 [15] を基にした抽象解釈が提案されている [3, 29]．この拡張によって，この種の bottom-up 型の抽象解釈によって解析できる特性の種類も広がった．

上でも述べたように論理プログラムにおける最も標準的な不動点意味論は最小 Herbrand モデルを最小不動点として特徴付けるものであり，それは bottom-up 実行 (いわゆる， Hyper 導出) に対応している．ここで紹介された枠組はいずれもその意味論 (またはその拡張版) を直接抽象化することにより抽象解釈を実現している．その利点は，標準的な意味論をそのまま抽象化するため実際の意味論と抽象解釈との対応が明らであり，かつ， bottom-up 実行に対応しているので，無限ループに陥るということがないことである．しかし，解析したいプログラムの初期ゴールが与えられたとしても， top-down の場合とは異なりそのゴールと関係のない実行過程に関する特性も解析しなければならないかもしれない．また加えて．初期ゴールに密接に依存する特性，例えば呼び出しゴールのパターンなどの解析が困難である． 3.2 節ではそのような解析に適した top-down 実行に基づいた枠組について述べる．

3.2 Top-down インタプリタを用いるアプローチ

次に，抽象解釈の枠組として通常の top down 実行 (OLD 導出) を用いる場合を考えてみよう．この場合は， 3.1 節の場合とは逆に以下のような長所・短所がある．

1. 初期ゴールで起動される実行経路に関する解析しか行なわない．そのため，初期ゴールの形に直接依存するような解析，例えば，呼び出しゴールのモード解析などに利用できる．
2. 馴染みのある通常の top-down 実行をシミュレートしているので理解し易い．
3. 通常の top-down 実行 (OLD 導出) を naïve に抽象化したものと異なり，無限ループに陥る危険性がない．

ここでは，無限ループの危険性を避けるために工夫された top down 実行を基にしたアプローチの一つとして，金森達の枠組を紹介する．

金森達は文献 [24, 28] で OLDT 導出 [49] を基本とした抽象解釈を提案した． OLDT 導出とは OLD 導出と Hyper 導出をうまく融合した導出機構であり，もともとは通常の実行での冗長計算や無限ループを避けるために考案されたものである． OLDT 導出は，以下の手順でゴールが導出されるテーブル付きの OLD 導出である．

1. 最初は通常の論理プログラムの場合と同様にゴール (G とする) の最左のアトム (A とする) をプログラムの節で展開する．
2. 一度節で展開されたアトム A の instance であるアトム A' がゴール G のサブゴールの最左のアトムとして現れた場合，これ以上節では展開されず，中断される．

3. 他の実行経路によって求まったアトム A の結果はテーブルに蓄えられ，それを用いて中断されていた A' の展開が進む．

OLDT 導出を用いた抽象解釈について，もう一度図 4の例で説明する．ここでは，ゴールは唯一のアトムからなるので，必要ならば“ゴール”を“ゴールの最左アトム”と読み換えて頂きたい．抽象化ゴール G_0 を図 1の再帰節 (2) で 2 回展開したことによって得られる抽象化ゴール G_2 は，G_0 と変数の renaming 以外は同じである．そのため，OLDT 導出に基づいた抽象解釈を用いると図 4の場合とはことなり，G_2 は図 1の各節で展開されないで，中断される．一方，G_0 や G_1 は単位節 (1) を用いて展開され成功終了し，それぞれ，

$$G_0 : [\{ A \leftarrow \underline{even}, B \leftarrow \underline{even}, C \leftarrow \underline{even} \}, \{ A \leftarrow odd, B \leftarrow \underline{odd}, C \leftarrow \underline{even} \}]$$
$$G_1 : [\{ A_1 \leftarrow \underline{even}, B_1 \leftarrow \underline{odd}, C_1 \leftarrow \underline{odd} \}]$$

という解代入が得られる．ここでは，ゴールと対応する解代人のテーブルの対で表している．この時点で G_0 に対する 2 つの解代入がテーブルに蓄えられている．G_0 と G_2 との変数の違いに対応させて renaming を施した G_0 の 2 つの解代入

$$\{ A_2 \leftarrow \underline{even}, B_2 \leftarrow \underline{even}, C_2 \leftarrow \underline{even} \}, \{ A_2 \leftarrow \underline{odd}, B_2 \leftarrow \underline{odd}, C_2 \leftarrow \underline{even} \}$$

を，それぞれ，実行を中断していた抽象化ゴール G_2 の解として用いる．この抽象代人を，G_0, G_1, G_2 の変数間の関係を表す 2 つの代入

$$\{ A \leftarrow suc(A_1), B \leftarrow B_1, C \leftarrow suc(C_1) \}, \{ A_1 \leftarrow suc(A_2), B_1 \leftarrow B_2, C_1 \leftarrow suc(C_2) \}$$

を用いて，木構造のルートである G_0 に向かって伝播させることによって，G_0, G_1 の新たな解が求まる．この解が新しいものであれば，それは再び G_2 の解として適用され，新たな解が求まらなくなるまで繰り返される．この例では，G_1 に対する新しい解代入が求まるだけで，もうそれ以上新しい解は得られない．すなわち，抽象化ゴール G_0 を OLDT 導出で抽象解釈することにより得られる結果は

$$G_0 : [\{ A \leftarrow \underline{even}, B \leftarrow \underline{even}, C \leftarrow \underline{even} \}, \{ A \leftarrow \underline{odd}, B \leftarrow \underline{odd}, C \leftarrow \underline{even} \}]$$
$$G_1 : [\{ A_1 \leftarrow \underline{even}, B_1 \leftarrow \underline{odd}, C_1 \leftarrow \underline{odd} \}, \{ A_1 \leftarrow \underline{odd}, B_1 \leftarrow \underline{even}, C_1 \leftarrow \underline{odd} \}]$$

がすべてである．

この他にも，Bruynooghe[4],Debray[12],Mannila[31] 達によって提案されている枠組が上記の枠組と同じ OLD 導出を基礎とした枠組という範疇に入る．

Bruynooghe 達は文献 [4] で，2節の図 2と同様に初期ゴールによって起動されるすべての OLD 導出の可能性を木構造で (こちらは AND-OR 木であるが) 表現している．無限ループを避けるための工夫は，OLDT 導出の場合とほぼ同様で，一旦展開したゴール G と同じ形のゴール G' は節で展開しないで，G' の結果として，G' のすべての変数が抽象領域の最小要素 ($\bot$) となるような代入を G' の解代入として返すものとして実行を継続させる．その実行が最終的に返す G の解代入を再び G' の結果として適用し，実行を反復する．この繰り返しが G の新しい解代入を生成しなくなれば実行は終了する．

Debray 達は文献 [12] で，呼び出し (サブ) ゴールと成功 (サブ) ゴールの対を初期ゴールから OLD 導出に従って top-down に順次求めテーブルに蓄える方式を提案している．その時点までに求まっているテーブルから節の本体のアトムに対応する成功ゴールを求め，それらを用いて節の頭部に対応する成功ゴールを求めている．節に再帰呼び出しがある場合，その成功ゴールが最初は蓄えられていないので他の (再帰呼び出しを用いない) 実行経路によって得られるまでその頭部の成功ゴールは得られない．結局，無限ループの問題は，OLDT 導出の場合と同様の効果

$$plus(zero, Y, Y)_{(7)}. \quad (1)$$
$$plus(suc(X), Y, suc(Z)) \leftarrow_{(5)} plus(X, Y, Z)_{(6)}. \quad (2)$$
$$\leftarrow_{(4)} plus(X, Y, suc(suc(zero))). \quad (3)$$

$$call\_pl(X, Y, suc(suc(zero))). \quad (4)$$
$$call\ pl(X, Y, Z) \leftarrow call\_pl(suc(X), Y, suc(Z)). \quad (5)$$
$$exit\_pl(suc(X), Y, suc(Z)) \leftarrow call\_pl(suc(X), Y, suc(Z)), exit\_pl(X, Y, Z). \quad (6)$$
$$exit\_pl(zero, Y, Y) \leftarrow call\_pl(zero, Y, Y). \quad (7)$$

図 5: プログラム *plus* と Alexander 法によって変換されたプログラム *plus_Alex*

で避けられている．また，Mannila 達は文献 [31] で，文献 [12] と同様の枠組を提案し，より詳細なアルゴリズムの提案，その計算量の議論を行なっている．

ここで紹介した枠組は，いずれも，不動点意味論を直接利用している訳ではないが，結局，何らかの形で不動点計算を用いて最終的な解析結果を求めている．

3.3 Magic Set 風の変換と Bottom-up インタプリタを用いるアプローチ

最近，文献 [7, 13, 27, 38, 42] において，それぞれ独立に，Magic Set 法 [2] や Alexander 法 [45] を利用した抽象解釈が提案されている．

Magic Set 法や Alexander 法などの手法は主に演繹データベースの分野で盛んに研究された手法である．プログラム P と初期ゴール G とが与えられた時，ある規則にしたがってプログラム P を変換する．その変換されたプログラムを bottom-up に実行すると，その各ステップはプログラム P の下でゴール G を top-down に実行する動作に対応しており，3.2節で紹介した OLDT 導出などのメモ付き top-down 実行と等価な動作をすることがわかっている．詳しくは文献 [40, 43, 46] などを参照されたいが[5]，ここでは *plus* の例を用いて簡単に説明する．

図 1 のプログラム *plus* と初期ゴール

$$\leftarrow plus(X, Y, suc(suc(zero)))$$

が与えられた時に Alexander 法を用いて変換されたプログラムを考えてみよう．まず，述語 *plus* に対応して，新しい述語 *call_pl*, *exit_pl* を導入する．Alexander 法では，それらの新しい述語を用いて，図 5 のプログラム *plus_Alex* に変換される．

例えば，プログラム *plus* の節 (2)

$$plus(suc(X), Y, suc(Z)) \leftarrow plus(X, Y, Z).$$

から，述語 *call_pl*, *exit_pl* を用いた二つの節 (5),(6)

$$call\_pl(X, Y, Z) \leftarrow call\_pl(suc(X), Y, suc(Z)).$$
$$exit\_pl(suc(X), Y, suc(Z)) \leftarrow call\_pl(suc(X), Y, suc(Z)), exit\_pl(X, Y, Z).$$

が生成される．これは，節 (2) を用いた top-down 実行を bottom-up 実行でシミュレートするような節 (5),(6) を生成したことになる．すなわち，節 (2) を用いた top-down 実行は，以下のように進む．

[5] 本格的に Magic Set 法や Alexander 法を知りたい読者はそれらの原著である文献 [2, 45] などを参照されたい．解説書としては文献 [40, 43] を見られたい．

1. 節 (2) の頭部 $plus(suc(X), Y, suc(Z))$ と unify できる呼び出しゴールが現れると，節 (2) の本体の (一般には本体の最左の) アトム $plus(X, Y, Z)$ が呼び出される.

2. 節 (2) の頭部 $plus(suc(X), Y, suc(Z))$ と unify できる呼び出しゴールが現れ，節 (2) の本体の (一般には本体のすべての) アトム $plus(X, Y, Z)$ が成功すると，その解代人を反映させてその頭部と unify できる呼び出しゴールが成功する.

一方，節 (5),(6) を用いた bottom-up 実行は，次のように進む.

1. 節 (5) の本体 $callLpl(suc(X), Y, suc(Z))$ と unify できるアトムが解アトム集合に現れると，その unifier を反映させた節 (5) の頭部 $callLpl(X, Y, Z)$ が新たに解アトム集合に加えられる.

2. 節 (6) の本体である $call\_pl(suc(X), Y, suc(Z))$ や $exit\_pl(X, Y, Z)$ と unify できるアトムが解アトム集合に現れると，その unifier を反映させた節 (6) の頭部 $exit\_pl(suc(X), Y, suc(Z))$ が新たに解アトム集合に加えられる.

この bottom-up 実行は，述語 $callLpl$ のアトムが $plus$ の呼び出しゴールのアトムに，述語 $exitLpl$ のアトムがその成功終了時のアトムに対応していると考えると，上記の top-down 実行をシミュレートしているのが分かる.

また，初期ゴール (3) や単位節 (1) も定義節の特殊な場合と見なすことによって，同様に，変換後の節を得ることができる．すなわち，初期ゴール (3) や単位節 (1) の top-down 実行時の動作は,

3. 無条件で初期ゴール (3) は呼び出される.

4. 節 (1)(の頭部)$plus(zero, Y, Y)$ と unify できる呼び出しゴールが現れると，直ちに，それが成功する.

であり，その動作をシミュレートするような変換された節 (4), (7) の bottom-up 実行時の動作は,

3. 単位節なので (無条件に) 節 (4) は解アトム集合に加えられる.

4. 節 (7) の本体 $call\_pl(zero, Y, Y)$ と unify できるアトムが解アトム集合に現れると，その unifier を反映させた節 (7) の頭部 $exit\_pl(zero, Y, Y)$ が，直ちに，解アトム集合に加えられる.

である.

一般に，変換後の各節はそれぞれ変換前のプログラムを top-down 実行した場合のあるプログラム・ポイントを表しており，実際，(4)–(7) の各節が表しているプログラム pl の top-down 実行時のプログラム・ポイントは，プログラム (1)–(3) の各節に付加されている添字 (4)–(7) に対応している.

このような変換規則で得られたプログラム $plus\_Alex$ を 3.1節で述べた Hyper 導出を用いて bottom-up に実行してみる．プログラム $plus\_Alex$ の解アトム集合を $AnsSet_{plus\_Alex}$ とすると，最初は単位節 (4) が $AnsSet^1_{plus\_Alex}$ として登録される.

$$AnsSet^1_{plus\_Alex} = \{\, callLpl(X_1, Y_1, suc(suc(zero))) \,\}$$

次に，順次 Hyper 導出を繰り返すことにより，以下のように $AnsSet_{plus\_Alex}$ は増加し，やがてこれ以上は増えなくなる．ここでも，$AnsSet_{plus\_Alex}$ の上付の添字は Hyper 導出の回数を表している.

$$
\begin{aligned}
AnsSet^{2}_{plus\_Alex} &= AnsSet^{1}_{plus\_Alex} \cup \\
&\quad \{\, callpl(X_2, Y_2, suc(zero)), exit\_pl(zero, suc(suc(zero)), suc(suc(zero))) \,\} \\
AnsSet^{3}_{plus\_Alex} &= AnsSet^{2}_{plus\_Alex} \cup \\
&\quad \{\, callpl(X_3, Y_3, zero), exit\_pl(zero, suc(zero), suc(zero)) \,\} \\
AnsSet^{4}_{plus\_Alex} &= AnsSet^{3}_{plus\_Alex} \cup \\
&\quad \{\, exit\_pl(zero, zero, zero), exit\ pl(suc(zero), suc(zero), suc(suc(zero))) \,\} \\
AnsSet^{5}_{plus\_Alex} &= AnsSet^{4}_{plus\_Alex} \cup \\
&\quad \{\, exit\_pl(suc(zero), zero, suc(zero)) \,\} \\
AnsSet^{6}_{plus\_Alex} &= AnsSet^{5}_{plus\_Alex} \cup \\
&\quad \{\, exit\_pl(suc(suc(zero)), zero, suc(suc(zero))) \,\}
\end{aligned}
$$

ここで，最初に与えられた初期ゴール

$$\leftarrow plus(X, Y, suc(suc(zero)))$$

と同じ形の引数を持つ述語 $exit\_pl$ によるアトム

$$exit\_pl(X, Y, suc(suc(zero)))$$

と unify 可能なアトムが解アトム集合 $AnsSet^{6}_{plus\_Alex}$ にある．それらとの unifier

$$
\begin{aligned}
&X = zero, Y = suc(suc(zero)) \\
&X = suc(zero), Y = suc(zero) \\
&X = suc(suc(zero)), Y = zero
\end{aligned}
$$

が初期ゴールを top-down 実行した場合の解代入である．

次に，プログラム *plus Alex* を抽象領域上で bottom-up 実行してみる．プログラム *plus Alex* に現れる関数記号や定数記号はプログラム *plus* のそれと同じなので，2 節で導入されたのと同じ偶数 / 奇数判定のための抽象領域

$$\{\, \bot, \underline{even}, \underline{odd}, \top \,\}$$

を用い，3.1 節と同様にそのまま抽象化し，抽象領域上での解アトム集合を $\underline{AnsSet}_{plus\_Alex}$ とする．まず，プログラム *plus_Alex* の単位節から求まる解アトム集合を $\underline{AnsSet}^{1}_{plus\ Alex}$ とし，順次，抽象領域上での Hyper 導出を繰り返すと，

$$
\begin{aligned}
&\underline{AnsSet}^{1}_{plus\_Alex} = \{\, callpl(\top, \top, \underline{even}) \,\} \\
&\underline{AnsSet}^{2}_{plus\_Alex} = \underline{AnsSet}^{1}_{plus\_Alex} \cup \{\, exit\ pl(\underline{even}, \underline{even}, \underline{even}), callpl(\top, \top, \underline{odd}) \,\} \\
&\underline{AnsSet}^{3}_{plus\_Alex} = \underline{AnsSet}^{2}_{plus\_Alex} \cup \{\, exit\_pl(\underline{even}, \underline{odd}, \underline{odd}), exit\_pl(\underline{odd}, \underline{even}, \underline{odd}) \,\} \\
&\underline{AnsSet}^{4}_{plus\_Alex} = \underline{AnsSet}^{3}_{plus\_Alex} \cup \{\, exit\_pl(\underline{odd}, \underline{odd}, \underline{even}) \,\} \\
&\underline{AnsSet}^{5}_{plus\_Alex} = \underline{AnsSet}^{4}_{plus\_Alex}
\end{aligned}
$$

となる．ここで求まった $\underline{AnsSet}_{plus\_Alex}$ と 2 節の図 4 とを対応させると，

$$\{\, callpl(\top, \top, \underline{even}), callpl(\top, \top, \underline{odd}) \,\}$$

は，図 4 の呼び出しゴール G_0 や G_1 に対応しており，

$$
\begin{aligned}
\{\ & exit\ pl(\underline{even}, \underline{even}, \underline{even}),\ exit\_pl(\underline{even}, \underline{odd}, \underline{odd}), \\
& exit\_pl(\underline{odd}, \underline{even}, \underline{odd}),\quad exit\_pl(\underline{odd}, \underline{odd}, \underline{even}) \,\}
\end{aligned}
$$

は，各呼び出しゴール G_0 や G_1 に対応する解代入をゴールに適用した抽象化ゴールである．また，具体領域上の場合と同様，初期ゴールから求まる抽象化アトム $exit\_pl(\top, \top, \underline{even})$ と unify 可能な抽象化アトム

$$\{\, exit\_pl(\underline{even}, \underline{even}, \underline{even}), exit\_pl(\underline{odd}, \underline{odd}, \underline{even}) \,\}$$

が，初期ゴールに対する解アトムである．

このような手法を用いることにより，bottom-up 型の抽象解釈だけでは解析できなかった例えば呼び出しゴールのモード解析などの様に与えられた初期ゴールやそのサブゴールに強く依存した特性の解析を bottom-up 型の抽象解釈で行なうことができる．

以上は Alexander 法を用いた抽象解釈について説明したが，この節の最初に述べたように，他にも Magic Set 法やそれらの改良版を用いた抽象解釈が最近いくつか提案されている．それらはいづれも使用されているプログラム変換法の違いに依存して少し差異はあるが基本的にはプログラム変換と Hyper 導出を基礎とするという点では共通している．

金森は文献 [27] で Alexander 法の改良版である Alexander Templates[46] を用いた抽象解釈を提案している．そこでは，Alexander Templates を用いた抽象解釈と，OLDT 導出を用いた抽象解釈や Mellish の提案した抽象解釈 [37] などとの関係が論じられている．以下では，Mellish の抽象解釈を簡単に説明し，Alexander 法を用いた抽象解釈との対応について述べる．

Mellish は通常の OLD 導出による top-down 実行をトレースするような概念を用いて，論理プログラムの実行を形式化し，それを抽象化することにより，彼がそれ以前に考案してきたデータフローなどの解析アルゴリズムの理論的形式化を行なった．

非常に荒っぽくいうならば，一般に「トレーサー」と呼ばれているデバッギング・ツールを用いて実行した時にユーザが見ることができる実行の各導出ステップでの情報，すなわち，呼び出しゴールとその終了ゴールのパターンをすべて集めたような情報を，以下のようにして二段階で計算する手法を提案した．

1. 初期ゴールとプログラムが与えられ時，そのゴールから派生するすべての呼び出しゴールとその終了ゴールの集合についての再帰定義を定める．

2. その再帰定義を方程式と見立て，その方程式の解の集合を bottom-up 近似を用いて求める．

次に，彼は第一段階の再帰定義を定めるステップを，与えられたプログラムに現れる述語に特化された，すなわち，ある種の部分評価を施された再帰定義を定めるように改良した．この改良された枠組を用いると次のステップでは効率良く解の集合を bottom-up に求めることができる．

Mellish の改良された枠組における再帰定義を与える段階は，そのまま，Alexander 法でのプログラム変換の段階，例えばプログラム *plus* と初期ゴールからプログラム *plus_Alex* を生成する段階に対応している．また，bottom-up 近似を用いて方程式の解を求める段階は，Hyper 導出を用いて解アトム集合を計算する段階に対応している．

また，Alexander Templates を用いた枠組 [27] は，解アトム集合を計算する段階での重複計算を避けるために，(*continuation* を用いて) 中間結果を保存する工夫が施されている．この枠組ではさらに効率的に解アトム集合を計算できる．また，文献 [46] によって，Alexander Templates と OLDT 導出の動作の対応が指摘されており，この結果から，これらに基づく抽象解釈がほぼ等価であることがわかる [27].

4 抽象解釈についての不動点計算を用いた一般論

3 節では，bottom-up 型，top-down 型，Magic Set 型の抽象解釈をそれぞれ紹介してきた．

他にも論理プログラムの抽象解釈には様々なアプローチがあるが，いずれも「新たな解(すなわち解析結果)が発見されなくなるまで，(ある種の)計算を繰り返す」という点において，なんらかの不動点計算を行なっているわけである．ただ，その不動点計算には以下の条件が必要である．

1. 不動点計算がプログラムの実行(または意味論)と正当な関係にある．

2. 有限回の計算で不動点が求まる．

条件 1 は得られた解析結果の信頼性において，条件 2 はプログラム解析の停止性において必要である．例えば，3.2 節の例で用いた OLDT 導出は OLD 導出と全く等価な導出であることが証明されており [49]，それに基づいた抽象解釈による解析の正当性が保証されている [24]．また，解析の停止性は抽象領域の有限性によって保証されている．また例えば，これも 3.2 節において紹介したが，Bruynooghe 達は文献 [4] で OLD 導出を基礎に AND-OR 木構造上の処理で抽象解釈を行なっていたが，Kemp 達は文献 [29] で完備束上の不動点意味論を用いた一般的な抽象解釈の枠組を提案し，その枠組で文献 [4] の枠組を形式化している．

ここでは，適当に計算の(具体)領域を設定し，その領域上での変換(いわゆる T-operator)を定めることにより，不動点計算を用いた一般的な枠組として抽象解釈を定式化できることを示す．以下では，特に意味論を限定せずに，具体領域上の不動点意味論，具体領域の抽象化，その抽象領域上の不動点意味の順で解説し，それらの間の“安全な”関係について述べる．また，それら一般的な解説の実例として，2 節の *plus* のプログラムを再び用い標準的な不動点意味論である Herbrand 解釈上の最小不動点意味論に意味論を特定し，その上での抽象解釈について(3.1 節よりやや形式的に)述べる．

4.1 具体領域上の不動点意味論

解析される結果の正当性を保証するためにはその抽象解釈の背景となる意味論が必要である．その意味論は不動点意味論である必要はなく，2 節の例のように OLD 導出というような操作的意味論を基礎とする抽象解釈も多く存在する．代表的な抽象解釈の枠組については 3 節で述べたが，そのような操作的意味論を基礎とするような枠組も結局なんらかの不動点計算を行なっている．

ここでは，抽象解釈の基礎としてどのような不動点意味論が用いられるかを議論の対象とはせず，プログラムの意味が具体領域上のある意味関数の不動点として与えられているとして話しを進める．

プログラムの通常の実行が具体的な対象(*object*)の集合である具体領域(D とする)上で行なわれるとする．具体領域 D はその上のある順序関係 $\subseteq$ に関して完備束を形成するとし，その最大要素(*top element*)，最小要素(*bottom element*)をそれぞれ $\top_D, \bot_D$ と記述する．プログラム P の意味が，単調で連続な(具体)意味関数 $T_P : D \rightarrow D$ の最小不動点 $lfp(T_P)$ で与えられるとする．$lfp(T_P)$ は，

$$T_P \uparrow 0 \subseteq T_P \uparrow 1 \subset \ldots T_P \uparrow \omega = lfp(T_P)$$

なるたかだか ω 回 T_P を適用することによって求まることが知られている．ここで，$T_P \uparrow n$ は $T_P(\bot_D)$ から n 回 T_P を適用することを意味する．この時，$lfp(T_P)$ をプログラム P の具体意味論と呼び，$Sem(P)$ と記述する．

論理プログラムにおける意味論には様々なものがあるが，不動点計算と直接関係付けられている意味論でもっとも標準的なものは Herbrand 解釈上の最小不動点意味論である．Herbrand 解釈やこの意味論の詳しい説明は文献 [30] に譲り，3.1 節の説明と少し重複するが，ここでは再び

図1のプログラム *plus* を用いて簡単に説明する．Herbrand 解釈では，プログラムの定数記号や変数記号をその記号通りに解釈するので，Herbrand 領域とよばれる解釈の領域はプログラムに現れる定数記号，関数記号から構成される変数を含まない項の集合である．すなわち，プログラム *plus* の Herbrand 領域 ($\mathcal{U}_{plus}$ で表す) は

$$\mathcal{U}_{plus} = \{\, suc^n(zero) \mid n\text{は自然数} \,\}$$

である．また，述語の引数が Herbrand 領域の要素であるすべてのアトムを Herbrand 基底と呼ぶ．プログラム *plus* の Herbrand 基底 ($\mathcal{B}_{plus}$ で表す) は

$$\mathcal{B}_{plus} = \{\, plus(t_1, t_2, t_3) \mid t_1, t_2, t_3 \in \mathcal{U}_{plus} \,\}$$

である．Herbrand 基底の部分集合を Herbrand 解釈と呼び，Herbrand 解釈であるようなモデルを Herbrand モデルと呼ぶ．また，その中で集合の包含関係において最も小さい Herbrand モデルを最小 Herbrand モデルと呼ぶ．プログラム P が与えられた時，Herbrand 基底 $\mathcal{B}_P$ のベキ集合上の意味関数 T_P が定義され，その最小不動点として最小 Herbrand モデルが求まることが知られている．一般のプログラム P に対する意味関数 T_P は

$$T_P(I) = \{\, H \in \mathcal{B}_P \mid\ H \leftarrow B \text{ が } P \text{ の節の ground instance} \wedge B \subseteq I \,\}$$

と定義される．これより，プログラム *plus* の場合の意味関数 T_{plus} は，

$$\begin{aligned} T_{plus}(I) = &\{\, plus(zero, t, t) \mid t \in \mathcal{U}_{plus} \,\} \cup \\ &\{\, plus(suc(t_1), t_2, suc(t_3)) \mid plus(t_1, t_2, t_3) \in I \,\} \end{aligned}$$

と定義でき，この最小不動点を求めると，以下のようになる．

$$\begin{aligned} &T_{plus}\uparrow 0 = \emptyset \\ &T_{plus}\uparrow 1 = \{\, plus(zero, suc^j(zero), suc^j(zero)) \mid j\text{は自然数} \,\} \\ &T_{plus}\uparrow 2 = T_{plus}\uparrow 1 \cup \{\, plus(suc(zero), suc^j(zero), suc^{j+1}(zero)) \mid j\text{は自然数} \,\} \\ &T_{plus}\uparrow 3 = T_{plus}\uparrow 2 \cup \{\, plus(suc^2(zero), suc^j(zero), suc^{j+2}(zero)) \mid j\text{は自然数} \,\} \\ &\qquad\qquad\vdots \\ &T_{plus}\uparrow \omega = \{\, plus(suc^i(zero), suc^j(zero), suc^{i+j}(zero)) \mid i, j\text{は自然数} \,\} \end{aligned}$$

すなわち，*plus* の具体意味論 $Sem(plus)$ (これは *plus* の最小 Herbrand モデルであるが) は

$$Sem(plus) = \{\, plus(suc^i(zero), suc^j(zero), suc^{i+j}(zero)) \mid i, j\text{は自然数} \,\}$$

である．また，これはプログラム *plus* の下で SLD 導出で実行して成功するすべてのアトムを過不足なく満たしている [30, 50].

4.2 抽象領域の設計

抽象解釈は具体領域 D の部分集合を表現するある種の記号を要素とするような有限集合である抽象領域 ($\underline{D}$ とする) 上で行なわれる．ここで，$\underline{D}$ もまた $\underline{D}$ 上に定義された順序関係 $\sqsubseteq$ に関して完備束を形成しなければならない．その最大要素，最小要素をそれぞれ $\top_{\underline{D}}, \bot_{\underline{D}}$ と記述する．

D は対象言語や意味論によって決定されるのに対し，$\underline{D}$ は，データタイプやデータフロー情報などの解析したい特性に応じて，抽象解釈を行なおうとする者が D からうまく設計し選ばなければならない．この時，具体領域 D と抽象領域 $\underline{D}$ 間には次の条件を満足する単調な抽象化関数 $\alpha : 2^D \rightarrow \underline{D}$ と具体化関数 $\gamma : \underline{D} \rightarrow 2^D$ が定義できなければならない．ここで，2^D は集合 D のベキ集合である．

1. 抽象領域 $\underline{D}$ の任意の要素 $\underline{d}$ に対して，$\underline{d} = \alpha(\gamma(\underline{d}))$ である．

2. 具体領域 D の任意の部分集合 d に対して，$d \subseteq \gamma(\alpha(d))$ である．

このような α, γ を定義するための $\underline{D}$ 上の条件など，詳細については，文献 [33, 39, 48] 等を参照されたい．

ここでもまた，4.1 節の *plus* の最小 Herbrand モデルの例で考えてみよう．具体領域は *plus* の Herbrand 基底 $\mathcal{B}_{plus}$ である．2 節の偶数 / 奇数判定のための抽象化を用いて対応する抽象領域を設定する．Herbrand 領域 $\mathcal{U}_{plus}$ に対応する抽象領域を $\underline{\mathcal{U}}_{plus}$ とすると，

$$\underline{\mathcal{U}}_{plus} = \{ \bot, \underline{even}, \underline{odd}, \top \}$$

である．また，意味関数の領域である Herbrand 基底に対応するその抽象化された領域 $\underline{\mathcal{B}}_{plus}$ は

$$\underline{\mathcal{B}}_{plus} = \{ plus(A{:}\underline{t}_1, B{:}\underline{t}_2, C{:}\underline{t}_3) \mid \underline{t}_1, \underline{t}_2, \underline{t}_3 \in \underline{\mathcal{U}}_{plus} \}$$

である．その時，$\mathcal{B}_{plus}, \underline{\mathcal{B}}_{plus}$ 上の具体化関数，抽象化関数は，それぞれ，以下のようにして定義できる $\mathcal{U}_{plus}, \underline{\mathcal{U}}_{plus}$ 上の具体化関数，抽象化関数から自然と定義できる．

$$\gamma(x) = \begin{cases} \emptyset & \text{if } x = \bot \\ \{suc^n(zero) \mid n\text{は偶数}\} & \text{if } x = \underline{even} \\ \{suc^n(zero) \mid n\text{は奇数}\} & \text{if } x = \underline{odd} \\ \mathcal{U}_{plus} & \text{if } x = \top \end{cases}$$

$$\alpha(x) = \begin{cases} \bot & \text{if } x = \emptyset \\ \underline{even} & \text{if } x \subseteq \{suc^n(zero) \mid n\text{は偶数}\} \\ \underline{odd} & \text{if } x \subseteq \{suc^n(zero) \mid n\text{は奇数}\} \\ \top & \text{上のいづれでもない場合} \end{cases}$$

このように定義された α, γ が上記 1，2 の条件を満たしているのは明らかであり，また，抽象領域上の順序関係 $\sqsubseteq$ を γ によって集合の包含関係 $\subseteq$ へと自然に帰着することによって，

$$\underline{d}_1 \sqsubseteq \underline{d}_2 \text{ iff } \gamma(\underline{d}_1) \subseteq \gamma(\underline{d}_2)$$

と定義すれば，この $\underline{\mathcal{U}}_{plus}$ は $\sqsubseteq$ に関して完備束を形成している．これらのことから，ここで設計された $\underline{\mathcal{U}}_{plus}$ が抽象領域のための条件をすべて満たしていることが分かる．

4.3 抽象領域上の不動点意味論

抽象解釈は，抽象領域の設計に加えて具体意味関数 T_P に対応する連続な抽象意味関数と呼ばれる関数 $\underline{T}_P : \underline{D} \to \underline{D}$ を定義することによって実現される．T_P を定義するのに用いられている具体領域 D 上の基本的な操作を抽象領域上の対応するものに修正することによって $\underline{T}_P$ は定義される．

例えば，最小 Herbrand モデルの場合だと，項と項の unification や代入の項への適用などを抽象領域上で再定義することにより，対応する抽象領域上の意味関数を定義できる．2.3 節で述べた $\underline{even}$ や $\underline{odd}$ と $suc(X)$ などの項との unification がその例である．

$\underline{D}$ が有限集合であることから，抽象意味関数 $\underline{T}_P$ は次のような高々有限回 k の変換で $\underline{T}_P$ の最小不動点 $lfp(\underline{T}_P)$ が求まる．この最小不動点 $lfp(\underline{T}_P)$ を抽象意味論と呼び，$\underline{Sem}(P)$ と記述する．

$$\underline{T}_P \uparrow 0 \subseteq \underline{T}_P \uparrow 1 \subseteq \ldots \subseteq \underline{T}_P \uparrow k = lfp(\underline{T}_P)$$

$$
\begin{array}{lccccccccccc}
\text{具体領域上の変換:} & T_P\uparrow 0 & \subseteq & T_P\uparrow 1 & \subseteq & \cdots & \subseteq & T_P\uparrow k & \subseteq & \cdots & \subseteq & T_P\uparrow\omega = Sem(P) \\
& \cap\!| & & \cap\!| & & \cdots & & \cap\!| & & \cdots & & \cap\!| \\
\text{抽象領域上の変換:} & \underline{T}_P\uparrow 0 & \sqsubseteq & \underline{T}_P\uparrow 1 & \sqsubseteq & \cdots & \sqsubseteq & \underline{T}_P\uparrow k & = & \cdots & = & \underline{T}_P\uparrow\omega = \underline{Sem}(P)
\end{array}
$$

図 6: 具体領域上，抽象領域上の変換ステップ

このようにして抽象意味論を定義しても，その意味論が具体的なプログラムの挙動を形式化している具体意味論に関してなんらかの基準において“正当な”ものでないならば，それを基礎とした抽象解釈，ひいてはその抽象解釈を用いて得られたプログラム解析の結果に正当性を見い出すことはできない．正当性の基準として望むべくは，任意のプログラム P に対して $Sem(P)$ と $\underline{Sem}(P)$ が常に同じ集合を表現していることであろう．それは抽象解釈，すなわち近似実行によって得られる近似解が実際の実行による解を常に過不足なく反映していることを意味し，直感的にもこの基準に無理があるのが分かる．より具体的には，前節で述べた α,γ が満たすべき 2 つの条件の両方において常に等号が成立しなければならないことを意味する．それは，有限とは限らない D のすべての部分集合それぞれを過不足なく表現する要素が $\underline{D}$ に必要で，$\underline{D}$ の有限性と矛盾する．そこで，抽象意味論の具体意味論に関する合意のある正当性の基準として以下の条件が用いられる [10]．これを抽象解釈の安全性の条件と呼ぶ．

任意のプログラム P に対して，$Sem(P) \subseteq \gamma(\underline{Sem}(P)))$ である．

Sem と $\underline{Sem}$ の間に上記の関係がある時，$\underline{Sem}$ は Sem を安全に抽象化していると呼ばれる．直感的に言えば，この安全性の条件は，抽象解釈の結果には余分な情報も含まれているかもしれないが，実際の実行に対応する情報は必ず含まれていなければならないということ要求している．

さて，抽象意味論 $\underline{Sem}$ が具体意味論 Sem を安全に抽象化するための条件は示された．次は，この安全性の条件が成り立つような抽象意味関数を定義しなければならない．このためには，$D, T_P, \underline{D}, \underline{T}_P$ の間に次の条件が成り立っていれば十分である．

1. $\bot_D = \gamma(\bot_{\underline{D}})$，かつ，$\bot_{\underline{D}} = \alpha(\bot_D)$ である．

2. $\underline{D}$ の任意の要素 $\underline{d}$ に対して，$T_P(\gamma(\underline{d})) \subseteq \gamma(\underline{T}_P(\underline{d}))$ である．

なぜなら，これらの条件から n に関する帰納法を用いて，次の式が証明できる．

任意の自然数 n に対して，$T_P\uparrow n \subseteq \gamma(\underline{T}_P\uparrow n)$.

この時，最小不動点を求めるための変換の各ステップで上記の式が成立するので，図 6 のようにして安全性の条件が証明できる．ここで，(図 6 では $\cap\!|$ で表わされているが) $T_P\uparrow i \subseteq \underline{T}_P\uparrow i$ は，$T_P\uparrow i \subseteq \gamma(\underline{T}_P\uparrow i)$ を表わしている．

このようにして，具体意味論を安全に抽象化している抽象意味論を求め，抽象解釈と称してプログラムの解析に利用するのである．どのような解析に用いられるかは抽象領域の設計に依存して決まる．

4.2 節で導入した抽象領域と最小 Herbrand モデルを基に，$plus$ のプログラムの偶数 / 奇数解析を行なってみる．

まず，$\underline{T}_{plus}$ を定義しなければならない．4.1 節の T_{plus} から素直に定義すると，

$$
\begin{aligned}
\underline{T}_{plus}(I) = &\{\, plus(A{:}\underline{even}, B{:}\underline{t}, C{:}\underline{t}) \mid \underline{t} \in \underline{\mathcal{U}}_{plus} \,\} \cup \\
&\{\, plus(A{:}\underline{suc}(\underline{t}_1), \underline{t}_2, \underline{suc}(\underline{t}_3)) \mid plus(A{:}\underline{t}_1, B{:}\underline{t}_2, C{:}\underline{t}_3) \in I \,\}
\end{aligned}
$$

と定義できる．ここで，関数 $\underline{suc} : \underline{\mathcal{U}}_{plus} \to \underline{\mathcal{U}}_{plus}$ は

$$\underline{suc}(x) = \begin{cases} \bot & \text{if } x = \bot \\ \underline{even} & \text{if } x = \underline{odd} \\ \underline{odd} & \text{if } x = \underline{even} \\ \top & \text{if } x = \top \end{cases}$$

である．この $\underline{T}_{plus}$ の最小不動点を求めると，

$$\underline{T}_{plus}\uparrow 0 = \emptyset$$
$$\underline{T}_{plus}\uparrow 1 = \{\, plus(A{:}\underline{even}, B{:}\underline{t}, C{:}\underline{t}) \mid \underline{t} \in \underline{\mathcal{U}}_{plus} \,\}$$
$$\underline{T}_{plus}\uparrow 2 = \underline{T}_{plus}\uparrow 1 \cup \{\, plus(A{:}\underline{odd}, B{:}t, C{:}\underline{suc}(\underline{t})) \mid \underline{t} \in \underline{\mathcal{U}}_{plus} \,\}$$
$$\underline{T}_{plus}\uparrow 3 = \underline{T}_{plus}\uparrow 2$$

となり，3回の繰り返しで不動点が得られ，$plus$ の抽象意味論 $\underline{Sem}(plus)$ が求まる．この最小不動点を求めるステップと 4.1 節での $lfp(T_{plus})$ を求めるステップを比較すると，各ステップ i で $T_{plus}\uparrow i \subseteq \gamma(\underline{T}_{plus}\uparrow i)$ が成り立っていることがわかる．また，求まった抽象意味論も

$$Sem(plus) \subseteq \gamma(\underline{Sem}(plus))$$

を満たしており，$\underline{Sem}(plus)$ は $plus$ の Herbrand モデルである $Sem(plus)$ を安全に抽象化していることが分かる．

さて，多くの読者は，$plus$ の偶数 / 奇数解析の結果として，“偶数 + 奇数” や “偶数 + 偶数” が偶数になるか奇数になるかの正しい解析ができていると予想しているであろうし，期待していると思われる．しかし，実はここで求まった $lfp(\underline{T}_{plus})$ は，そのような正しい解析結果に対応する抽象意味論

$$\underline{Sem}(plus)_{precise} = \{\ plus(A{:}\underline{even}, B{:}\underline{even}, C{:}\underline{even}), plus(A{:}\underline{even}, B{:}\underline{odd}, C{:}\underline{odd}),\ plus(A{:}\underline{odd}, B{:}\underline{even}, C{:}\underline{odd}),\ \ plus(A{:}\underline{odd}, B{:}\underline{odd}, C{:}\underline{even})\ \}$$

だけではなく，それに加えて，

$$\underline{Sem}(plus)_{empty} = \{\, plus(A{:}\underline{even}, B{:}\bot, C{:}\bot), plus(A{:}\underline{odd}, B{:}\bot, C{:}\bot)\,\}$$

や

$$\underline{Sem}(plus)_{redundant} = \{\, plus(A{:}\underline{even}, B{:}\top, C{:}\top), plus(A{:}\underline{odd}, B{:}\top, C{:}\top)\,\}$$

が含まれている．$\gamma(\bot) = \emptyset$ より，$\gamma(\underline{Sem}(plus)_{empty}) = \emptyset$ と分かるので，こちらは無視していいが，$\underline{Sem}(plus)_{redundant}$ は，“偶数 + 自然数” が “自然数” であるとか “奇数 + 自然数” が “自然数” であるとかの情報であり，解析結果としては冗長なものである．しかし，抽象意味論を具体化した集合が具体意味論を包含しているという意味において安全な結果が得られているわけである．このような冗長な結果しか得られない理由は，Herbrand 基底やその上での unification 処理において抽象化の度合が少し強いからである．より正確な結果を得るためには，例えば，5.2 節で紹介するような変数の共有情報を保存する抽象化を行えばいい．詳細については，文献 [3, 4, 18, 24] などを見られたい．

5 抽象解釈を用いたプログラム解析の具体例

3 節では，抽象解釈の枠組としてどのようなものがあるかという観点から研究されている代表的な抽象解釈について，分類，紹介した．ここでは，それらの枠組を用いて実際どのようなプログラム解析が可能かについて簡単に紹介する．誌面の関係上，代表的なものを簡単にしか紹介できないことをお詫びしておく．これらの他にも，構造体の再利用などコンパイル技術に密接に関係のある解析 [32] やプログラムの停止性の検出 [26] や安全に SLDNF 導出を行なうための負リテラルの groundness の解析 (*non-floundering*) [36] など抽象解釈を用いた様々なプログラム解析例が提案されている．興味のある読者は各文献を参照されたい [11]．

5.1 モード解析

論理プログラムの抽象解釈において，モード解析は最も基本的な解析応用である．単純なモード解析では，具体項を変数を含まない項 (*ground term*) であるか否か分類し，解析を行なう．モード解析の目的は，節の頭部の各変数が，呼び出し時にどのように具体化 (*instantiate*) されているかを解析するものである．文献 [37] をはじめとして提案されているほとんどすべての枠組において，モード解析が取り上げられているのは，比較的簡単に解析でき，かつ，その結果のコンパイラー最適化への利用技術も確立されているからである．

モード解析のようなデータフロー解析には，二つの変数が実行時に同じ変数として unify されているか (変数の *aliasing* または *sharing* と呼ばれる) どうかが重要な情報となる．この sharing 情報をより正確に解析することが，モード解析の精度を上げることになる [3, 4, 18, 24].

文献 [41] では，モード解析を用いて並列実行可能なゴールの解析を行なっている．節の本体に現れる複数の And ゴールがお互い独立に実行可能であることがコンパイル時に解析できるなら，それらのゴールを異なるプロセッサー上で並列に実行できる．例えば，共有変数を持たない場合などは明らかであるが，共有変数を持つ場合でもそれらのゴールが呼ばれた時その共有変数が必ず ground 項に instantiate されているなら独立実行，すなわち，並列実行可能である．

5.2 データタイプ推定

項のパターンをデータタイプとして抽象化し，呼び出しゴールの引数や計算結果のデータタイプを推定する．項の深さによって抽象化したり項をリストや整数などある特定の集合に含まれるかどうかで分類し，抽象化する [23, 24]．より一般的には，有理木 (*rational tree*) のような木構造で項を表現することによって無限種類ある項を有限の領域に抽象化したりする [19]．また，モード解析で有効であった sharing 情報はタイプ推定の精度を上げるのにも有効である [5].

例えば，4.3 節で解析結果として冗長な結果が得られている例を示した．ここで sharing 情報を用いれば以下に解析結果が改善されるかを簡単に説明する．

抽象領域上のプログラム *plus* に対する変換 $\mathcal{T}$ の最小不動点 $lfp(\mathcal{T}_{plus})$ として $\underline{Sem}(plus)_{precise}$ のみが得られるような抽象解釈は，4.2 節で定義した抽象領域 $\mathcal{B}_{plus}$ や抽象意味関数 $\mathcal{T}_{plus}$ を少し修正すれば良い．例えば，抽象化ゴールにゴールの変数の共有情報 (*sharing*) を付加することにより，より正確な解析を行なうことができる．sharing は例えば (X, Y) などで表現される抽象化ゴールに現れる変数の対からなる集合で，その対の変数は具体領域上では同じ変数として unify されていることを意味している．つまり，X がある項 t に instantiate された場合，もう一つの変数 Y もその項 t に instantiate される．例えば，抽象化ゴール $plus(A{:}\underline{even}, B{:}\top, C{:}\top)$ に対応する具体領域上のゴールは

$$\{plus(t_1, t_2, t_3) \mid t_1 \in \gamma(\underline{even}) \wedge t_2, t_3 \in \gamma(\top)\}$$

である．しかし，それに sharing を付加した抽象化ゴール $plus(A{:}\underline{even}, B{:}\top, C{:}\top)\{(B, C)\}$ は，sharing によって第 2 引数と第 3 引数が共有されているという制限が加わるので，その対応する具体領域上のゴールは

$$\{plus(t_1, t_2, t_2) \mid t_1 \in \gamma(\underline{even}) \wedge t_2 \in \gamma(\top)\}$$

である．例えば，$plus(A{:}\underline{even}, B{:}\top, C{:}\top)\{(B, C)\}$ と抽象化ゴール $plus(\top, \top, \underline{even})$ と unify した場合，第 3 引数が $\underline{even}$ であるという情報が sharing によって第 2 引数に伝播し，その結果の抽象化アトムは，$plus(A{:}\underline{even}, B{:}\underline{even}, C{:}\underline{even})\{(B, C)\}$ となる．

このように情報の欠落が sharing がない場合より少ないので，4.3 節の例で含まれていた冗長な解析結果を取り除くことができる．

5.3 決定性検出，関数性検出

backtrackingによる非決定的な動作を利用したプログラムを書けるというのは，論理プログラムの大きな利点である一方，そのような論理プログラムは一般にメモリー消費的であるという欠点がある．あるプログラムがbacktrackingしないで一意に成功するとか(決定的)，あるプログラムのある引数が決まっているなら(入力として与えられているなら)一意に解が求まる(関数的)とかのような特性をコンパイル時に検出できるなら，そのようなプログラムに対してはメモリー消費的でないコードを生成することができる．

Mellishは文献[37]で提案した抽象解釈の枠組で決定性の検出を定式化できることを示唆している．また，Debray達は文献[12]で関数性の検出方法の概略を述べている．金森達は文献[25]で文献[24]の枠組を用いた関数性の検出について述べている．

5.4 Occur Check解析

unificationの実装においては，いわゆる*occur check*を行なわないものがほとんどである．occur checkを行なわないunificationを用いることは，unificationが無限ループに陥る危険性があるだけでなく，論理的に誤っている結果を導くかもしれない．一方，すべてのunificationに対してoccur check処理を行なうことは，効率面で大きなマイナスである．コンパイル時にどのunifiactionがoccur checkの不要であるかを発見できるなら，より高速に実行でき，かつ，安全な処理系の実現が可能である．文献[47]では，sharing情報をもちいたoccur check解析を提案している．

5.5 プログラムの部分計算への応用

論理プログラムの部分計算における基本的なアイデアは与えられたゴールGを用いてできる限りプログラムPの各節を展開(*unfolding*)しておくことである．文献[17]では，ゴールGの実行において決定的な導出，すなわち，最左のアトムとunify可能な節が唯一しかないような導出を抽象解釈を用いて求めている．この情報を用いて論理プログラムの部分計算の制御に利用している．

6 おわりに

本稿では，対象言語を論理プログラムに限った抽象解釈の基本的な枠組について述べ，簡単な例を用いて解説し，代表的な応用解析についても紹介した．

論理プログラムの抽象解釈の枠組は，top-down実行をそのままうまく抽象化した枠組，bottom-up実行を抽象化した枠組など様々なものが提案されている．しかし，意味論を規定し，その意味領域を抽象化し，不動点計算で結果を求めるという観点で見ると，いづれの枠組も同じ範疇に入る．そのような抽象解釈の(抽象化される実行形態や意味論に依存しない)統一性は，文献[34]で表示的意味論を土台として述べられている．

抽象解釈の各応用分野に関しては，5節で簡単に紹介するにとどめた．より詳細な内容に興味がある読者は各参考文献を参照されたい．また，最近，並行論理型言語や制約論理型言語の抽象解釈に関する研究も行なわれているが，ここでは触れなかった．文献[6, 9, 8, 22, 35]などを参照されたい．また，*The Journal of Logic Programming*の抽象解釈特集[14]が発行されており，そこには抽象解釈の解説論文も含まれている．これら参考にされたい．

最後に，抽象解釈に関する今後の研究について筆者なりの考えを述べる．

論理プログラムに限らず抽象解釈の関する研究は，実際の応用解析を理論的に説明するのを目的として理論的枠組の研究が起こり，それを後追いする形で再びその理論的枠組にのっとり様々

な応用解析の研究がなされてきた．論理プログラムの抽象解釈に関しては，文献 [29, 48] に代表される表示的意味論を用いた形式化で，(純論理プログラムに関しての) 理論的研究にだいたい一段落ついたのではないかと思う．一方，実用的な応用解析はというと，コンパイルコードやスケジューリングの最適化といった分野で盛んに行なわれているが，まだまだ広範囲な応用が残されている．また，副作用などを含む論理プログラムへの拡張や並行，制約論理プログラムなどに関しては，理論的研究もまだまだ不十分であろう．また，一般に抽象解釈による解析は全域的解析，すなわち，すべてのプログラムが解析に必要であり，通常のコンパイラーが部分コンパイル可能なのと異なり大規模なプログラム解析に適さないという批判がある．モジュール化されたプログラムに対する抽象解釈も今後の研究課題である．

なお，以下の参考文献で抽象解釈に関係が深く，本文ではあまり触れなかった文献について簡単な説明と筆者の感想を加える．読者の参考になれば幸いである．

謝辞

最後に，本チュートリアルを書くにあたって貴重なコメントを頂いたNTTソフトウェア研究所の小野 諭氏に感謝します．また，章の構成の段階から数多くのコメントを頂いた三菱電機中央研究所の金森 直氏に感謝します．

参考文献

[1] Abramsky, S. and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

抽象解釈の教科書的な論文集である．論理プログラムに関して2件の文献が掲載されている [37, 20]．全体としては関数型言語に対するものが多い．第1章は編集者達による抽象解釈の一般的な解説とサーベイである．

[2] Bancilhon, F. and R. Ramakrishnan: An Amateur's Introduction to Recursive Query Processing Strategies, *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1986, pp.16-52.

[3] Barbuti, R., R. Giacobazzi and G. Levi: A Declarative Approach to Abstract Interpretation of Logic Programs, Univ. of Pisa, TR-20/89, Italy, 1989.

文献 [15] のモデルを基礎に bottom-up 型の抽象解釈の理論を構築している．変数情報がモデルに含まれるので，変数の sharing 情報などを用いて抽象化することにより，bottom up 型の抽象解釈でモード推定が行なえる．この枠組は文献 [34] などで一般化されるが，文献 [34] が枠組の概説にとどまっているのに比べ，ここでは，基本的な操作の抽象化や具体化の定義などが詳細に述べられている．

[4] Bruynooghe, M., G. Janssens, A. Callebaut and B. Demoen: Abstract Interpretation: Towards the Global Optimization of Prolog Programs, *Proc. of the 4th International Symposium on Logic Programming*, pp.192-204, San Francisco, California, 1987.

抽象解釈の基本となる操作の定義，それらの操作のタイプ推定やモード解析における実際のアルゴリズムを与えている．グラフを用いてデータタイプを表現しており，グラフを操作するアルゴリズムを詳細に述べている．直感的には正しそうであるが，全体の枠組の正当性に関する理論的考察があまり述べられていない．本文でも述べたが，後に，文献 [29] で形式化されている．

[5] Bruynooghe, M. and G. Janssens: An Instance of Abstract Interpretation Integrating Type and Mode Inferencing, *Proc. of the 5th International Conference and Symposium on Logic Programming*, R .A. Kowalski and K. A. Bowen (eds.), pp.669–683, MIT Press, 1988.

データフロー情報を sharing 情報を用いて伝播することにより，より詳細なデータタイプを解析できることを示した論文である．枠組は文献 [4] を用いている．

[6] Codish, M., J. Gallagher: A Semantic Basis for the Abstract Interpretation of Concurrent Logic Programs, *Technical Report CS89–26*, November, 1989.

[7] Codish, M., D. Dams and E. Yardeni: Bottom-up Abstract Interpretation of Logic Programs, Unpublished Draft, August, 1990.

[8] Codish, M., M. Falaschi and K. Marriott: Suspension Analysis for Concurrent Logic Programs, *Proc. of the 8th International Conference on Logic Programming*, Furukawa, K. (ed.), pp.331–345, 1991.

Plotkin style で定義された並行論理型言語の操作的意味論をそのまま抽象化することによって， suspension 解析を行なった．枠組は非常に単純であるが，抽象解釈の停止性を保証するために過度の抽象化を行なう必要があり，データタイプなどの解析には適さない．

[9] Codognet, C., P. Codognet and M. M. Corsini: Abstract Interpretation for Concurrent Logic Languages, *Proc. of the North American Conference on Logic Programming*, S. Debray andM. Hermenegildo (eds.), pp.215–232, 1990.

並行論理型言語の実行アルゴリズムを抽象化する枠組を提案している．文献 [8] の枠組に比べて複雑であるが，停止性に関する問題はない．

[10] Cousot, P. and R. Cousot: Abstract Interpretation: A Unified Framwork for Static Analysis of Programs by Construction or Approximation of Fixpoints, *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*, Los Angeles, pp.238–252, 1977.

抽象解釈のバイブル的な論文である．対象言語は手続き型を想定したフローチャート言語で，フローチャートの各プログラムポイントでの変数の値が具体領域となる．完備束上の不動点理論を用いた枠組で，多くの抽象解釈の基本となっている．

[11] Cousot, P. and R. Cousot: Abstract Interpretation and Application to Logic Programs, *The Journal of Logic Programming, Special Issue: Abstract Interpretation*, S. K. Debray (ed.), Vol.13, No.2&3, pp.103-179, July, 1992.

抽象解釈の解説論文である．前半は Lattice 理論や不動点理論と (論理プログラムに限らない一般の) 抽象解釈との関係を述べ，論理プログラムの抽象解釈についても， bottom-up 型，top down 型，それらの混合型と様々な枠組について形式的にかつ詳しく述べられており，解説論文としての内容は十分備わっているが複雑でかつ読み難い論文である．その最後の章で (約 6 ページで全体の 1/10 であるが)，論理プログラムの抽象解釈に関する文献の "Thematic Survey" と称して，枠組の説明，それらの枠組を用いた解析の具体例が非常に多く，かつ，網羅的に紹介されている．各項目はそれほど詳細ではないが，紹介 (多くは参照のみであるが) されている文献は相当の量であるので，サーベイとしては適当であろう．

[12] Debray, S. K. and D. S. Warren, Automatic Mode Inference for Prolog Programs, *Proc. of 1986 Symposium on Logic Programming*, Salt Lake City, Utah, Sep. 1986, pp.78-88, Revised version in *The Journal of Logic Programming*, Vol.5, No.1, pp.207-229, 1988.

[13] Debray, S. K. and R. Ramakrishnan: Canonical Computation of Logic Programs, Unpublished Draft, July, 1990.

[14] Debray, S. K. (ed.), *The Journal of Logic Programming, Special Issue: Abstract Interpretation*, Vol.13, No.2&3, pp.103-179, July, 1992.

[15] Falaschi, M., G. Levi, M. Martelli and C. Palamidessi, Declarative Modeling of Operational Behaviour of Logic Languages, *Theoretical Computer Science*,69, 1989.

[16] Fitting, M., A Kripke-Kleene Semantics for Logic Programming, *The Journal of Logic Programming*, Vol.2, No.4, pp.295–312, 1985.

[17] Gallagher, J. and M. Bruynooghe: The Derivation of an Algorithm for Program Specialisation, *Proc. of the 7th International Conference on Logic Programming*, D. H. D. Warren and P. Szeredi (eds.), pp.732–746, 1990.

[18] Jacobs, D. and A. Langen: Accurate and Efficient Approximation of Variable Aliasing in Logic Programs, *Proc. of the North American Conference on Logic Programming*, E. L. Lusk and R. A. Overbeek (eds.), pp.154–165, 1989.

[19] Janssens, G. and M. Bruynooghe, An Application of Abstract Interpretation: Integrated Type and Mode Inferencing, Report CW86, Katholieke Universiteit Leuven, April, 1989.

文献 [4] より詳細なレポートである.

[20] Jones, H. D. and Søndergaard: A Semantics-Based Framework for the abstract Interpretation of Prolog, in [1], pp.123–142.

[21] Hecht, M. S., *Flow Analysis of Computer Programs*, North-Holland, 1977.

[22] Horiuchi, K.: Less Abstract Semantics for Abstract Interpretation of FGHC Programs, *Proc. of the International Conference on Fifth Generation Computer Systems*, June, 1992.

[23] Kanamori, T. and K. Horiuchi: Type Inference in Prolog and its Application, *Proc. of 9th International Joint Conference on Artificial Intelligence*, pp.704–707, 1985.

[24] Kanamori, T. and T. Kawamura: Analyzing Success Patterns of Logic Programs by Abstract Hybrid Interpretation, ICOT Technical Report TR-279, Tokyo, 1987.

本文でかなり詳細に説明した OLDT 導出を用いた枠組である. 枠組の理論的正当性が詳細に証明されており, タイプ推定, モード解析などの例も述べられている. 改定版が文献 [28] に掲載される.

[25] Kanamori, T., K. Horiuchi, and T. Kawamura: Detecting Functionality of Logic Programs Based on Abstract Hybrid Interpretation, ICOT Technical Report TR-331, Tokyo, 1987.

[26] Kanamori, T., T. Kawamura and K. Horiuchi: Detecting Termination of Logic Programs Based on Abstract Hybrid Interpretation, ICOT Technical Report TR-398, Tokyo, 1987.

[27] Kanamori, T.: Abstract Interpretation based on Alexander Templates, ICOT Technical Report TR-485, Tokyo, March, 1990, Revised version in *The Journal of Logic Programming*, 1992.

[28] Kanamori, T. and T. Kawamura: Abstract Interpretation Based on OLDT Resolution, *The Journal of Logic Programming*, 1992.

[29] Kemp, R. S. and G. A. Ringwood: An Algebraic Framework for Abstract Interpretation of Definite Programs, *Proc. of the North American Conference on Logic Programming*, S. Debray andM. Hermenegildo (eds.), pp.516 530, 1990.

[30] Lloyd, J. W.: *Foundation of Logic Programming*, Second, Extended Edition, Springer-Verlag, 1987.

非常に有名な論理プログラムの教科書である．文献 [50] で述べられている一連の標準的な意味論にはじまり，否定を含む論理プログラム，演繹データベース， 無限実行を含むプログラムの意味論と論理プログラムの基礎理論に関して広範囲にカバーしている．なお，first edition の翻訳が産業図書から出版されている．佐藤雅彦，森下真一 (訳): 論理プログラミングの基礎.

[31] Mannila, H. and E. Ukkonen, Flow Analysis of Prolog Programs, *Proc. of the 4th Symposium on Logic Programming*, pp.205–214, San Francisco, California, 1987.

[32] Marien, A., G. Janssens, A. Mulkers, andM. Bruynooghe: The Impact of Abstract Interpretation on Code Generation: An Experiment in Efficiency, *Proc. of the Sixth International Conference on Logic Programming*, G. Levi and M. Martelli (eds.), pp. 33-47, 1989.

[33] Marriott, K. and H. Søndergaard: Bottom-up Abstract Interpretation of Logic Programs, *Proc. of the 5th International Conference and Symposium on Logic Programming*, R .A. Kowalski andK. A. Bowen (eds.), pp.733 748, MIT Press, 1988.

[34] Marriott, K. and H. Søndergaard: Semantic-based Dataflow Analysis of Logic Programs, *Information Processing 89*, G. X. Ritter (ed.), pp.601–606, North-Holland, 1989.

表示的意味論を基礎に bottom-up と top-down の双方のタイプの抽象解釈について述べており，具体的な解析例によってそれらを使い分ける必要があると主張している.

[35] Marriott, K. and H. Søndergaard: Analysis of Constraint Logic Programs, *Proc. of the North American Conference on Logic Programming*, S. Debray and M. Hermenegildo (eds.), pp.531 547, 1990.

[36] K. Marriott, H. Søndergaard, and P. Dart: A Characterization of Non-floundering Logic Programs, *Proc. of the North American Conference on Logic Programming*, S. Debray and M. Hermenegildo (eds.), pp. 662–680, 1990.

[37] Mellish, C. S.: Abstract Interpretation of Prolog Programs, *Proc. of the 3rd International Conference on Logic Programming*, E. Shapiro (ed.), pp.463–474, LNCS-225, Springer-Verlag, 1986, Revised version in [1], pp.181–191.

[38] Mellish, C. S.: Using Specialisation to Reconstruct Two Mode Inference, Unpublished Draft, March, 1990.

[39] Melton, A., D. Schmidt and G. Strecker: Galois Connections and Computer Science Applications, *Category Theory and Computer Programming*,D. Pitt *et al* (eds.), pp.299–312, LNCS-240, Springer-Verlag, 1986.

抽象解釈における安全性の条件などの数学的基盤となる理論に関する論文である.

[40] 宮崎 収兄, 世木 博久: 演繹データベースの問合せ処理, 情報処理, Vol.31, No.2 (1990), pp.216-224

[41] Muthukumar, K. and M. Hermenegildo: Determination of Varible Dependence Information Through Abstract Interpretation, *Proc. of the North American Conference on Logic Programming*, E. L. Lusk and R. A. Overbeek (eds.), pp.166–185, 1989.

[42] Nilsson, U.: Abstract Interpretation: A Kind of Magic, Unpublished Draft, Linköping University, Sweden, 1990.

[43] 西尾 章治郎, 楠見 雄規: 演繹データベースにおける再帰的な問い合わせの評価法, 情報処理, Vol.29, No.3 (1988), pp.240-255.

[44] 小野 諭 : 抽象実行 — そのフレームワークと実例 —, JSSST 関数プログラミング 91, 近代科学社, 1992.

[45] Rohmer, J., R. Lescoeur and J. M. Kerisit: The Alexander Method – A Technique for the Processing of Recursive Axioms in Deductive Databases, *New Generation Computing*, 1986, Vol.4, No.3,pp.273-286.

[46] Seki, H.: On the Power of Alexander Templates, *Proc. of ACM Symposium on Principles of Database Systems*, 1989, pp.150-159.

[47] Søndergaard, H.: An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction: *European Symposium on Programming*, B. Robinet and R. Wilhelm (eds.), pp.327–338, LNCS-213, Springer-Verlag, 1986.

[48] Søndergaard, H.: Semantic-Based Analysis andTransformation of Logic Programs, Ph. D. Thesis, University of Copenhagen, Denmark, 1989, Revised version in TR-89/12, Dept. of Computer Science, University of Melbourne, Australia, 1989.

[49] Tamaki, H. and T. Sato: OLD Resolution with Tabulation, *Proc. of the 3rd International Conference on Logic Programming*, pp.88 98, London, 1986.

[50] van Emden, M. H. and R. A. Kowalski: The Semantics os Predicate Logic as a Programm ming Language, *The Journal of the Association for Computing Machinery*, vol.23, No.4, 1976, pp.733 742.