プログラミング-言語・基礎・実践 14－7
（1993．10．29）

SimpleObjectにおけるクラス定義単位の型制約導出と型検査

大久保弘崇　　坂部俊樹　　稲垣康善

名古屋大学

継承と代入を持つ単純化された純粋な型なしオブジェクト指向言語を考え、それに対する型推論について論じる。型はクラスの集合であり、部分型は集合の包含関係によって定められるものとする。このようなオブジェクト指向言語の型推論については、Palsbergらによってプログラム全体を単位とする型推論方法が提案されているが、本稿ではクラス定義単位に型に関する制約関係を抽出し、プログラム全体だけでなく部分的な型検査をも行えるような方法を考察する。

Modular type checking and deriving type restrictions for class definitions in SimpleObject

Hirotaka Ohkubo　　Toshiki Sakabe　　Yasuyoshi Inagaki

Nagoya University

We present a new method of type checking for SimpleObject, a basic model language of untyped object-oriented languages with inheritance and assignment. Palsberg et al. presented a type inference system for a similar language, which works for concrete programs but not for class definitions. Our type inference system works for class definitions to derive type restriction relations and then to check type correctness of class definitions as well as programs composed of class definitions.

1 はじめに

オブジェクト指向言語は記述力の高さとクラスの再利用という形での柔軟性の高さが特徴である。しかしまたその記述力の高さゆえにソフトウエアの信頼性が他の言語に比べ低いという欠点がある。オブジェクト指向言語では、オブジェクトにメッセージを送り、そのメッセージに対する結果としてのオブジェクトの何らかの反応が計算となる。ここで、送ったメッセージを送り先のオブジェクトが受けとることができるかどうかは実際にその計算を行なってみる —つまりメッセージを送ってみる — まで不明である。メッセージが受けとられなければ計算はそこで失敗する。これはオブジェクト指向言語にのみ現れるエラーの形式である。このようなエラーをなくしプログラムの信頼性を向上させるために、プログラムに対して型をつける方法が一般的であるが、オブジェクト指向言語に対しては通常の言語における方法では困難なため、さまざまな方法が提案されている [3, 4]。

Palsberg らは [1, 2] において、継承と代入を持つ単純化された純粋な型なしオブジェクト指向言語を考え、それに対する型推論について論じている。型はクラスの集合であり、部分型は集合の包含関係によって定められるものとする。[1, 2] ではプログラム全体を単位として型推論を行なっているが、本稿ではオブジェクト指向プログラミングにおいてはクラス定義がプログラミングの最小単位であるとする立場から、クラス定義単位に型に関する制約関係を抽出し、プログラム全体だけでなく部分的な型検査をも行えるような方法を考察する。

2 言語 SimpleObject

ここでモデルとする言語 SimpleObject は [1] におけるモデル言語を基にして設計されている。変更点は変数や括弧の扱いを厳密にして式が正しく解釈できるようにしたこと、メタなオペレータ instanceOf を取り除いたこと、継承メソッドの扱いを super から inherited にしたことである。言語の特徴は以下の通りである。

- 計算の過程においてオブジェクトのみが存在する純粋な言語である。
- 継承の機構を持つ。
- 代入文を持つ。

その文法を図 1に示す。

プログラムはクラスの集合である。プログラムを実行した結果とはメインプログラムを表すクラス $C_{MainProgram}$ の メソッド run を評価した結果とする。すなわち式 $C_{MainProgram}$ new ⇐ run の評価値である。クラスはインスタンス変数 $Id_1 \ldots Id_m$ とメソッド $M_1 \ldots M_k$ を宣言する。インスタンス変数はそのクラスのメソッドとそのクラスを継承して定義するクラスのメソッド内がそのスコープである。クラスのインスタンスにメッセージが送られると、そのメソッド名を持つメソッドの定義式 E が実引数を仮引数 $Id_{a1}, \ldots, Id_{an}$ に割り当てたうえで評価される。この結果がメッセージ送信の評価値である。メソッドの仮引数 $Id_{a1} \ldots Id_{an}$ と一時変数 $Id_1 \ldots Id_m$ はメソッド内がそのスコープである。

式の評価について説明する。代入文は、E を評価し、現在そのスコープ内にある一時変数、メソッド引数、もしくはインスタンス変数 Id に結果を代入する。全体の評価値は E の評価値である。メッセージ送信はまず $E, E_1, \ldots, E_n$ を順に評価し、E の結果であるオブジェクトにメソッド名のメッセージを実引数 $E_1, \ldots, E_n$ の評価値で送る。E の結果が nil であった場合にはエラーとなる。親のメソッドの呼び出しはまず $E_1, \ldots, E_n$ を順に評価し、現在実行中のメソッドの定義されているクラスの親クラスのメソッドをメッセージ送信によってそのメソッドが選択されたかのように評価する。評価値についてもメッセージ送信と同様である。ブロックの評価値は E_1 から E_n を順に評価した上で E_n の値である。条件分岐は E を評価し、それが nil でなければ E_t、そうならば E_f を評価し、式の値とする。インスタンス生成は指定されたクラスのインスタンスを作る。インスタンス変数の初期値は全て nil とする。その新たなオブジェクトへの参照を式の値とする。myClass new では self のクラスのインスタンスを作る。self は現在の計算が行われている主体（最も最近に完了していないメッセージ送信を受けとったオブジェクト）への参照である。変数名は変数に代入されている値を参照する。nil はなにも参照していないという特別な評価値である。

あるオブジェクトを他のオブジェクトと共有したり、互いにメッセージを送り合うため、値の代入や引数の割り当ては全てオブジェクトへの参照として扱う。つまり、上の説明において「値」と呼んだのはオブジェクトに対する参照であり、オブジェクト自身は別のところに存在する。

メッセージに対するメソッドは、受け側のオブジェクトのクラスから順に継承関係をたどってメッセージ名と等しいメソッド名を持つメソッドを探索して

(プログラム)	P ::= $C_1 \cdots C_n$ $C_{MainProgram}$		
(クラス)	C ::= **class** クラス名 [**inherits** 親クラス名]	[**var** $Id_1, \ldots, Id_m$]	$M_1 \ldots M_k$
(メソッド)	M ::= **method** メソッド名 [($Id_{a1}, \ldots, Id_{an}$)]	[**var** $Id_1, \ldots, Id_m$]	E
(式)	E ::= Id := E	(代入)	
	| E <= メソッド名[($E_1, \ldots, E_n$)]	(メッセージ送信)	
	| inherited メソッド名[($E_1, \ldots, E_n$)]	(親のメソッドの呼び出し)	
	| {E_1 ; $\cdots$; E_n}	(ブロック化)	
	| if E then E_t else E_f	(条件分岐)	
	| クラス名 new | myClass new	(インスタンス生成)	
	| self	(現在の主体オブジェクト)	
	| Id	(変数)	
	| nil	(空)	

図 1: 文法

決定される。ここで、メソッドが見つからなかった場合や引数の数が合わない場合はエラーとなる。

この言語上で自然数などのクラスが記述できることから、この言語が充分な記述力を持っていることがわかる。また、インタプリタを Standard ML によって記述した (約 500 行)。サンプルプログラムと実行例を Bに挙げる。

3 型

一般に、オブジェクト指向言語のプログラムはオブジェクトがメッセージを受けとれないときエラーを起こす。メッセージが受けとれないとは、つまりデータに対する操作が存在しないということである。型とはある操作の可能なデータの集合を規定するものであるから、オブジェクト指向言語における型とは、あるメッセージを受けとれるオブジェクトの集合と考えることができる。従って、型をオブジェクト指向言語のプログラムに対して判定することで、そのプログラムが「メッセージが受けとれない」というエラーを起こすことがないかどうかを検査することができる。

ここでは、SimpleObject における型をクラス名の集合と定める。SimpleObject では、式はエラーが生じなければ何らかの宣言されたクラスに属するオブジェクトか nil に計算される。ある式 E が型 T を持つとは、その式を評価した結果が nil でない限り結果のオブジェクト O のクラス C が必ず T の要素であるということである。ある型 T′ が型 T の部分集合であるとき ($T' \subseteq T$)、T′ は T のサブタイプであるという。また、T を T′ のスーパータイプという。当然あるサブタイプの値はその型のスーパータイプの値でもある。

nil はどのクラスにも属さないので、型 ϕ を持つ。よって任意の型が nil のスーパータイプとなり、nil は任意の型の値と互換性があり、例えばどのように型付けされた変数にも代入してもよい。

4 クラス定義単位の型制約導出と型検査

前節で定義した型を、SimpleObject のプログラムに対して割り当てる方法を議論する。正しくプログラムが実行されたときに、評価された全ての式は nil かもしくは何らかのオブジェクトとなる。式が評価された結果のオブジェクトのクラスに対応して式に型が付けられる。これを型割当 $[\cdot]$ とよぶ。型割当はクラス定義中のインスタンス変数、メソッド定義中の仮引数、一時変数、全ての部分式を含むメソッド定義式から型への写像である。式の文字列ではなくクラス定義中の式の出現そのものが定義域となることに注意されたい。逆に、プログラムに型割当がないということは、プログラムに型の誤りがあるということである。

クラス定義が与えられたとき、そのメソッドを構成する式から、その型割当に対する制約を考えることができる。例えば、あるクラスのメソッドに「クラス名 new」の式があれば、ここではそのクラス名で何かクラスが定義されていることを仮定している。また、メッセージ送信式があれば、受け手の式のオブジェクトがそのメッセージを受けとれることを期待している。これらのような、式から導ける型に関する仮定を式制約として、クラス定義の各式に対して導出できる。これはクラス中の型に関する情報を与える。この導き方に関しては 4.1 節で議論する。

次に、式制約を使って、そのクラスのメソッドの引数と返り値に関する型の制約と変数や仮引数に関する型の制約を導く。本来全てのクラスについての情報が得られなければ式の型はわからないが、ここでは最低限必要な型のパラメータをとり、これに依

存して型が求まる形式のクラスのメソッドの型の制約を導く。これをクラス制約という。これについては 4.2 節で議論する。

こうして求まったクラス制約を用いて、Simple-Object の プログラムコードに対してさまざまな型検査をすることができる。

一つ目は 4.3 節で述べるクラス間整合性検査である。プログラマはコードを書くとき、インスタンス生成式など他のクラスに対する仮定、メッセージ送信式など式の持つ型についての仮定を既に持っている。それは、この変数には何のクラスのオブジェクトが代入される、などの形で持っていると考えられる。よって、あるクラスのクラス制約の型パラメータに対して、プログラマの想定していたクラスのクラス制約を当てはめてみる。この結果矛盾が生じるようなら、コードに何らかのミスがあると考えられる。

もう一つは 4.4 節で述べるプログラム型検査である。これは、メインプログラムまで実行に必要な全てのクラスが揃った状態では「ある制約を満たすクラス全ての集合」の要素を完全に求めることができるので型の自動推論が可能となりプログラムが型エラーを含むかどうかをチェックすることができる。

本推論系では、「完全に」定義されたクラス定義をその単位とする。継承によって定義されているクラスの場合、クラス制約を求めるためにはその全てのスーパークラスの定義が同時に必要である。(A参照) 式制約は完全でないクラス制約の記述に基づいて行なわれるので、あるクラスを継承することで複数のクラスが定義されているとき、親クラスに対する式制約は再利用できる。よって推論全体のデータフローは図 2の様になる。

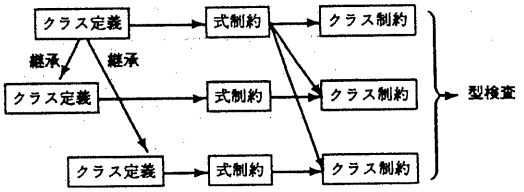

図 2: 型推論のデータフロー

細部を単純化するためにプログラムに対して自明な仮定をする。クラス名は唯一で重複しないものとする。継承しているクラスを含めてインスタンス変数の重複はなく、メソッド内での仮引数と一時変数もインスタンス変数のどれとも異なる唯一のものとする。つまりある変数の出現は即座に定まる唯一な宣言を持つ。

4.1 式制約

クラス定義に対して、その中の各式の型割当が満たすべき制約である式制約を次のように定義する。これは、型割当がそうであったように、式に関するものではなく、式の出現に関するものである点に注意されたい。

まず、インスタンス変数宣言、メソッドの仮引数宣言、一時変数宣言に対して、一意な型変数を割り当てる仮定により、式での変数の各出現に対応する型変数は一意に定まる。

各式に対して、式制約は図 3のように構文的に定まる。ここで、T has メソッド名 は型 T の全ての要素のクラス C について C 自身かそのスーパークラスで メソッド名 のメソッドが定義されていることを示す。needs クラス名 はそのクラスがプログラム中になくてはならないことを表す。$\mathcal{D}$ はメソッドの引数の型に関する型のリストを返す関数である。$\mathcal{R}$ はメソッドの返り値に関する型を返す関数である。この 2 つは次節で定義される。$\mathcal{D}$ に関する関係 $\supseteq$ はリストの長さが等しく、その要素同士が部分集合の関係にあることを示す。□ は継承によって変わる「自分のクラス」を入れるための箱である。

次に、クラス定義中のメソッド定義に対して、その引数と返り値の型の関係を記述する制約をつける。これは、

$$\textbf{method}\ \text{メソッド名}(\mathrm{Id_{a1}},\ldots,\mathrm{Id_{an}})\ \mathrm{E}$$

というメソッド定義に対して

$$\blacksquare\text{has メソッド名}$$
$$\mathcal{D}1(\blacksquare \leftarrow \text{メソッド名}) = \langle \mathrm{t_1},\ldots,\mathrm{t_n}\rangle$$
$$\mathcal{R}1(\blacksquare \leftarrow \text{メソッド名}(\mathrm{t_{a1}},\ldots,\mathrm{t_{an}})) = [\mathrm{E}]$$

という制約をつける。これも式制約の一部である。■ も □ と同様の記号である。

4.2 クラス制約

完全なクラス定義をつくるクラス定義群に対する式制約を用いて、クラス定義に対するクラス制約を求める。これはクラスが外部に求める他のクラスに関する制約と、それに依存して決定できるクラスのメソッドの型の関係を定めるものである。その構成アルゴリズムは以下のようである。

まず $\mathcal{D}1,\mathcal{R}1$ を正しくする。

1. 継承の大元のクラスのメソッド定義に対する $\mathcal{D}1,\mathcal{R}1$ に注目する。

Id := E	$t \supseteq [E]$
	$[Id := E] = [E]$
	ただし t は変数もしくは仮引数 Id に対して割り当てられている型変数
E <= メソッド名 $(E_1, \ldots, E_n)$	
	[E] has メソッド名
	$\mathcal{D}([E] \leftarrow$ メソッド名$) \supseteq \langle [E_1], \ldots, [E_n] \rangle$
	$[E <=$ メソッド名$(E_1, \ldots, E_n)] = \mathcal{R}([E] \leftarrow$ メソッド名$([E_1], \ldots, [E_n]))$
inherited メソッド名 $(E_1, \ldots, E_n)$	
	継承により定義していないクラスならばエラーで終了
	継承元のクラスを $\mathcal{P}$ とすると
	$\mathcal{P}$ has メソッド名
	$\mathcal{D}(\mathcal{P} \leftarrow$ メソッド名$) \supseteq \langle [E_1], \ldots, [E_n] \rangle$
	[inherited メソッド名$(E_1, \ldots, E_n)] = \mathcal{R}(\{\mathcal{P}\} \leftarrow$ メソッド名$([E_1], \ldots, [E_n]))$
$\{E_1; \ldots; E_n\}$	$[\{E_1; \ldots; E_n\}] = [E_n]$
if E then E_t else E_f	[if E then E_t else E_f] $= [E_t] \cup [E_f]$
クラス名 new	needs クラス名
	[クラス名 new] = { クラス名 }
myClass new	[myClass new] = {□}
self	[self] = {□}
Id	[Id] = t
	ただし t は変数もしくは仮引数 Id に対して割り当てられている型変数
nil	$[nil] = \phi$

図 3: 式制約

2. 注目しているクラスの子のクラスでメソッドが override されているならば、override されているクラスの $\mathcal{D}1, \mathcal{R}1$ 中の ■ を注目しているクラスに置き換え、注目から外す。has 制約は捨てる。
3. 子クラスのメソッド全てに対して注目する。
4. さらに次の子のクラスに対して同じ処理を繰り返す。
5. 最後に、注目されている has,$\mathcal{D}1, \mathcal{R}1$ 中の ■ および式制約中の □ をクラス制約を作ろうとしているクラスで置き換える。

この手続きによって、継承における override に対応させてメソッドの型制約を求めることができる。最後に注目されている $\mathcal{D}1, \mathcal{R}1$ はこのクラスのメソッドに関するもの全てである。

次に、$\mathcal{D}1, \mathcal{R}1$ とその他の式制約を全て集め、クラスに対するもっとも簡単な制約関係になるようにこれらを解く。まず has,$\mathcal{D}, \mathcal{R}$ を定義する。

$$T \text{ has メソッド名} = \forall C \in T(C \text{ has メソッド名})$$

$$\mathcal{D}(T \leftarrow \text{メソッド名}) = \bigcap_{C \in T} \mathcal{D}1(C \leftarrow \text{メソッド名})$$

$$\mathcal{R}(T \leftarrow \text{メソッド名}(t1, \ldots, t_n)) = \bigcup_{C \in T} \mathcal{R}1(C \leftarrow \text{メソッド名}(t1, \ldots, t_n))$$

制約が解けるのは次のような場合である。

1. 制約が真であるとき。その制約を取り除く。
2. ある型変数 t に対して、$T \subseteq t, t \subseteq T$ という 2 つの制約があるとき。それらを取り除き t に T を割り当てる。
3. ある型変数 t に対して、$t \subseteq \{C\}$ という制約があるとき。t に型がついたとすると $\{C\}$ しかありえないので、この制約を取り除き t に $\{C\}$ を割り当てる。
4. $\phi \leftarrow$ メソッド名 という形が発生したときはエラーで終了する。

項目 3 において、もし後に $t \subseteq \{C'\} \neq C$ というような制約があり、矛盾する危険があると思われるかもしれないが、この場合どちらにしろ $t = \{C\}$ と矛盾するのでこう仮定して問題はない。

この手続きは後の型検査時の制約を解く手続きと同じものである。そのとき、クラス制約におけるローカルな型変数に対する割当が $\mathcal{D}, \mathcal{R}$ によって起きたときには、それはその回のその変数を持つクラス内の推論についてのみ有効である。次にまたその型変数に対して他の割当が起きた時は、改めて推論を行なう。これによって、実行毎に異なる型を持ち得るメソッドの型についての型割当を柔軟に計算できる。型制約を解く手続きは、互いに矛盾する制約があっ

た場合や、型変数に ϕ が割り当てられた場合は型推論は失敗したとして終了する。

最後に、残った制約をクラス制約として構成する。残った制約式の中に、「t has メソッド名」の形式の制約があれば、この型変数 t は外部のクラスに依存する型なので、クラス制約のパラメータとする。そうでない型変数はクラス定義内にローカルなものである。クラス制約は、こうして求まったパラメータ型変数の集合、ローカルな型変数の集合、それらの間の型制約式、クラスのメソッドに関する(注目されている) $\mathcal{D}1, \mathcal{R}1$ からなる組である。これは、型パラメータに依存して、クラスのメソッドの型に関する関係 $\mathcal{D}1, \mathcal{R}1$ と、式制約によってクラス中の全ての式および変数に型を与えるものである。

4.3 クラス間整合性検査

クラス定義をプログラミングした後、変数についた型制約にプログラマの想定していたクラスを型パラメータに代入してみることによって、「少なくともそのクラスで使う分には型が正しい」という意味での型の整合性の確認ができる。

具体的な手続きは次の通りである。クラス制約で導かれる型注釈を参照して、着目しているクラス制約の型パラメータに想定していた型を割り当てる。これでできた制約式を、式制約からクラス制約を求める方法と同様に解いていく。型推論が失敗せずに停止すれば、クラス間整合性があるといえる。

4.4 プログラム型検査

実行に必要なクラスが全て揃ったとき、それはプログラムとなるが、このとき (そのプログラムの実行については) 完全に型検査をすることができる。

$$\text{t has メソッド名}$$

という形の制約は、

$$\text{t} \subseteq \{\forall \text{C} | \text{C has メソッド名}\}$$

という意味であるが、実行時に存在するクラス全てがあるならばこの右辺の集合の要素が全て求まるので、式制約を後者のものに置き換えることができる。この置き換えを行なうと、型パラメータに対する型制約が充分に求まるので、それらは既にパラメータではない。よって全てのクラス制約の制約式を一つにまとめて、また同様な推論をする。推論が成功すれば、その構成でプログラムを実行したときの全ての式に対する型が求まったことになる。

4.5 例

継承クラス

継承により型が変化する式を扱う例を示す。

```
class A
   method m
      self
class B inherits A
   method n
      myClass new
class C
   method f(o)
      { o <= m; o <= n }
```

このクラス定義に対して、式制約が次のように付けられる。型割当を文字列で書くが、実際には式の出現に対して割り当てられていることに注意されたい。

クラス A について
- [self] = {□}
- ■ has m
- $\mathcal{D}1(\blacksquare \leftarrow \text{m}) = \langle\rangle$
- $\mathcal{R}1(\blacksquare \leftarrow \text{m}) = [\text{self}]$

クラス B について
- [myClass new] = {□}
- ■ has n
- $\mathcal{D}1(\blacksquare \leftarrow \text{n}) = \langle\rangle$
- $\mathcal{R}1(\blacksquare \leftarrow \text{n}) = [\text{self}]$

クラス C について
- o has m
- $\mathcal{D}(o \leftarrow \text{m}) \supseteq \langle\rangle$
- [o <= m] = $\mathcal{R}(o \leftarrow \text{m})$
- o has n
- $\mathcal{D}(o \leftarrow \text{n}) \supseteq \langle\rangle$
- [o <= n] = $\mathcal{R}(o \leftarrow \text{n})$
- ■ has f
- $\mathcal{D}1(\blacksquare \leftarrow \text{f}) = \langle o \rangle$
- $\mathcal{R}1(\blacksquare \leftarrow \text{f}) = [\text{o} <= \text{n}]$

これらを使ってクラス制約が導かれる。

クラス	A
パラメータ	(なし)
ローカル	(なし)
制約	A has m

$\mathcal{D}1(\text{A} \leftarrow \text{m}) = \langle\rangle$
$\mathcal{R}1(\text{A} \leftarrow \text{m}) = \{\text{A}\}$

クラス	B

パラメータ　(なし)
ローカル　(なし)
制約　B has m
　　B has n
$\mathcal{D}1(\mathrm{B} \leftarrow \mathrm{m}) = \langle\rangle$
$\mathcal{R}1(\mathrm{B} \leftarrow \mathrm{m}) = \{\mathrm{B}\}$
$\mathcal{D}1(\mathrm{B} \leftarrow \mathrm{n}) = \langle\rangle$
$\mathcal{R}1(\mathrm{B} \leftarrow \mathrm{n}) = \{\mathrm{B}\}$

クラス　C
パラメータ　o
ローカル　(なし)
制約　o has m
　　$\mathcal{D}(o \leftarrow \mathrm{m}) \supseteq \langle\rangle$
　　$[\mathrm{o} <= \mathrm{m}] = \mathcal{R}(o \leftarrow \mathrm{m})$
　　o has n
　　$\mathcal{D}(o \leftarrow \mathrm{n}) \supseteq \langle\rangle$
　　$[\mathrm{o} <= \mathrm{n}] = \mathcal{R}(o \leftarrow \mathrm{n})$
　　C has f
$\mathcal{D}1(\mathrm{C} \leftarrow \mathrm{f}) = \langle o \rangle$
$\mathcal{R}1(\mathrm{C} \leftarrow \mathrm{f}) = \mathcal{R}(o \leftarrow \mathrm{n})$

メッセージ n を送っているので、クラス C のメソッド f の引数 o には n に対応するメソッドが定義されていなくてはならない。よってこの変数に対する型割当を {B} とするクラス間整合性検査は成功し、A を含めると失敗する。

多相型メソッド

クラス C のメソッド id は多相な型を持つ。

```
class A

class B

class C
   method id(v)
      v

class Main
   method run
      var x
      x := C new;
      x <= id(A new);
      x <= id(B new)
```

式制約は次のようになる。

クラス C について
　C has id
　$\mathcal{D}1(\blacksquare \leftarrow \mathrm{id}) = \langle v \rangle$
　$\mathcal{R}1(\blacksquare \leftarrow \mathrm{id}(v)) = v$
クラス Main について
　[C new] = { C }
　$x \supseteq$ [Cnew]
　needs A
　[A new] = { A }
　[x] = x
　x has id
　$\mathcal{D}(\blacksquare \leftarrow \mathrm{id}) \supseteq \langle[\mathrm{Anew}]\rangle$
　$[\mathrm{x} <= \mathrm{id}(\mathrm{A\ new})] = \mathcal{R}(\blacksquare \leftarrow \mathrm{id}([\mathrm{Anew}]))$
　needs B
　[B new] = { B }
　[x] = x
　x has id
　$\mathcal{D}(\blacksquare \leftarrow \mathrm{id}) \supseteq \langle[\mathrm{Bnew}]\rangle$
　$[\mathrm{x} <= \mathrm{id}(\mathrm{B\ new})] = \mathcal{R}(\blacksquare \leftarrow \mathrm{id}([\mathrm{Bnew}]))$
　Main has run
　$\mathcal{D}1(\blacksquare \leftarrow \mathrm{run}) = \langle\rangle$
　$\mathcal{R}1(\blacksquare \leftarrow \mathrm{run}) = [\mathrm{x} <= \mathrm{id}(\mathrm{B\ new})]$

クラス制約は次のようになる。

クラス　C
パラメータ　(なし)
ローカル　v
制約　C has id
$\mathcal{D}1(\mathrm{C} \leftarrow \mathrm{id}) = \langle v \rangle$
$\mathcal{R}1(\mathrm{C} \leftarrow \mathrm{id}(v)) = \mathcal{R}(x \leftarrow \mathrm{id}(\{\mathrm{B}\}))$

クラス　Main
パラメータ　x
ローカル　(なし)
制約　$x \supseteq \{\mathrm{C}\}$
　　needs A
　　x has id
　　$\mathcal{D}(x \leftarrow \mathrm{id}) \supseteq \langle\{\mathrm{A}\}\rangle$
　　needs B
　　$\mathcal{D}(x \leftarrow \mathrm{id}) \supseteq \langle\{\mathrm{B}\}\rangle$
　　Main has run
$\mathcal{D}1(\mathrm{Main} \leftarrow \mathrm{run}) = \langle\rangle$
$\mathcal{R}1(\mathrm{Main} \leftarrow \mathrm{run}) = \mathcal{R}(x \leftarrow \mathrm{id}(\{\mathrm{B}\}))$

プログラム型検査を行なうと、クラス C のメソッド id の引数 v の型は {A,B} であるが、それを実際に呼び出している Main の run においては

$$[\mathrm{x} <= \mathrm{id}(\mathrm{Anew})] = \{\mathrm{A}\}$$

$$[x <= id(Anew)] = \{B\}$$

という多相な型割当がつく。

5 まとめ

SimpleObject のクラス定義を単位としてクラスの型の制約を抽出し、これを利用してクラスのコーディングの際に型の観点からクラス単位、もしくはプログラム全体でコードをチェックをする方法を提案した。このシステムで求まる型は型割当に対する推定である。これによって、最適化コンパイラなどへの応用が可能であろう。また、型推論機構を拡張して、基本データ型を組み込むことも容易であろう。

今後の課題として、制約の解を求める手続きを確立すること、システムを実装してこのシステムにより得られる型情報が有効なものであることを確かめることがあげられる。また、[2] において、本研究の基となっている Palsberg のシステムがスタック・リストなどの「コンテナ」クラスに対する型付けの不充分さが指摘されている。本システムも同様の弱さを持っていると考えられるので、対応が必要であろう。

参考文献

[1] Jens Palsberg and Michael I. Schwartzbach: "Object-Oriented Type Inference", OOPSLA'91, pp.146-161

[2] Nicholas Oxhøj, Jens Palsberg and Michael I. Schwartzbach: "Making Type Inference Practical", ECOOP'92 European Conference on Object-Oriented Programming, LNCS 615, pp.329-349

[3] John C. Mitchell: "Toward a typed foundation for method specialization and inheritance", Conference Record of the 17th Annual ACM Symposium on Principle of Programming Languages, 1991, pp.109-124

[4] William R. Cook: "Inheritance Is not Subtyping", 同上, pp.125-135

A Discussion: 継承クラスの取り扱いについて

継承を用いたクラス定義は、そのスーパークラス全てが定まらなければ定まらない。これを扱うモデルとして理想的なものは、継承を用いたクラス定義の意味を親クラスの完結した定義から自分のクラスの完結した定義への関数として扱う方法であろうが、本モデルのアプローチからはこれはできない。あくまで完結したクラス定義に対してのみ型推論を行なう。これは、プログラムが実行される段階ではクラス定義は高々静的なものであるという事実に基づく立場である。プログラムの実行時に動的にクラスが定義できる Object Dynamic Language においてはこの立場はとれないであろう。ただ、本モデルにおいても、継承を全くの syntax sugar として textual expansion を用いて型システムの外で解決してしまうのではなく継承を意識した型推論を行なう。ただしこの両者の違いはまだ明らかではない。継承クラスにおける継承した式の型が異なってくることが実際に問題となるかどうかが鍵であろう。

B SimpleObject のプログラム例

B.1 自然数クラス

```
class Natural
  var rep
  method isZero
    if rep then nil else Object new
  method succ
    Natural new <= update(self)
  method update(x)
    { rep := x; self }
  method pred
    if self <= isZero then self else rep
  method less(i)
    if i <= isZero
    then nil
    else if self <= isZero then Object new
    else self <= pred <= less ( i <= pred )
  method plus(i)
    if i <= isZero
    then self
    else self<=succ<=plus(i<=pred)
```

B.2 実行例

B.2.1 自然数クラスを利用したプログラム

```
class Main
  method run
    var one,two
    {
      one := Natural new <=succ;
      two := Natural new <=succ <=succ;
      one <= plus ( two )
    }
```

B.2.2 実行結果

```
Natural 1
rep       Natural 2
          rep       Natural 3
                    rep       Natural 4
                              rep       nil
```