2N－3

データドメインの詳細化に基づくプログラムの段階的構成法*

吉岡信和　鈴木正人　片山卓也†
北陸先端技術大学院大学　情報科学研究科‡

1　はじめに

大規模で複雑な仕様を満たすソフトウェアの開発を行う際、ソフトウェアの段階的構成法が有効である。これは、より簡単なものや重要な部分から段階的にソフトウェアを構築して行く方法である。しかし、従来の手法では、中間段階のソフトウェアは実行することができなかったために、最終段階になってから多くの問題点が発生する場合があった。鶴巻のHAWAII法[1]では関数に注目し詳細化を行う事でこの点に対処している。しかし、これではデータを段階的に詳細化できない。

本研究では、ソフトウェアの詳細化を、データドメインの詳細化という観点から捉える。そして、データドメインの詳細化に伴ってプログラムを構成する方法を提案する。データを抽象化したプログラムに解釈を与える方法として抽象解釈(Abstract Interpretation)[2]の技法がある。これは、データの詳細化段階で現れる中間の値に対するプログラムの解釈を与える時に有効である。プログラムの構成方法は次のようになる。初めに抽象化したデータドメインを考え、その上でのプログラムを定義する。次に、そのデータドメインを段階的に詳細化し、各ドメインに対するプログラムも段階的に詳細化する。プログラムの解釈はこの抽象解釈の技法を利用する。

2　データドメインの詳細化に基づくプログラムの詳細化

2.1　抽象化したドメインとその上での解釈

データドメインを詳細化し、その中間段階のデータに対してもプログラムを定義し、解釈することで複雑なプログラムの開発が容易になる。そこで本研究では、データドメインを詳細化しながらプログラムを構築し、その解釈を与える。

データドメインの要素の部分集合を1つの抽象値と捉え、ドメインを抽象値の集まりとみなす事をデータドメインの抽象化という。この抽象化されたドメイン上でのプログラムの解釈を行うのが抽象解釈(Abstract Interpretation)による技法である。ただし、抽象解釈の技法ではデータの抽象化とともに、プログラム中の演算を抽象値間の演算とみなすことで解釈がすすむが、本研究の手法ではこれを逆に利用し、抽象化されたデータを詳細化するともに、プログラムを構築し、解釈を行うことでプログラムの段階的詳細化をおこなう。

2.2　データドメインの構造と詳細化

データドメインの詳細化は、その構造によって表現することができる。ドメインDは、要素の集合Sと要素の詳細化関係を表す半順序関係"$\preceq$"によって以下のように定義される。

定義1 $D = \langle S, \preceq \rangle$

ドメインの詳細化は次のように定義できる。

定義2 $D_0 = \langle S_0, \preceq_0 \rangle$、$D_1 = \langle S_1, \preceq_1 \rangle$とするとドメイン$D_0$をドメイン$D_1$に詳細化する($D_0 \sqsubseteq D_1$)とは次の関係が成立する時である。

$$S_0 \subseteq S_1,\ \preceq_0 \subseteq \preceq_1$$

例1　唯一の要素から成るドメインD_0を次のように定義する。

$$D_0 = \langle S_0, \preceq_0 \rangle \quad (\text{但し、} S_0 = \{Num\}, \preceq_0 = \{\})$$

これに新しく要素Neg、0、Posを加えたものを、ドメイン$D_1 = \langle S_1, \preceq_1 \rangle$とすると、定義は以下の様になる。

$$S_1 = \{\ Num,\ Neg,\ 0,\ Pos\ \}$$
$$\preceq_1 = \{\ (Num,\ Neg), (Num,\ 0), (Num,\ Pos)\ \}$$

また、ドメインD_1に要素$Even$、Oddを加えることによりドメイン$D_2 = \langle S_2, \preceq_2 \rangle$を定義する。(図1参照)

$$S_2 = \{\ Num,\ Neg,\ 0,\ Pos,\ Even,\ Odd\ \}$$
$$\preceq_2 = \{\ (Num,\ Neg), (Num,\ 0), (Num,\ Pos),$$
$$(Pos,Even), (Pos,Odd)\ \}$$

また、次の関係が成立する。

$$S_0 \subseteq S_1 \subseteq S_2$$
$$\preceq_0 \subseteq \preceq_1 \subseteq \preceq_2$$
すなわち $D_0 \sqsubseteq D_1 \sqsubseteq D_2$　□

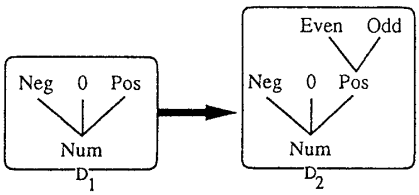

図1: データドメインの構造とその詳細化

2.3　プログラムの詳細化

データドメインを詳細化し、その上でのプログラムを定義するすることで、プログラムは段階的に詳細化できる。ドメインの詳細化に伴い、詳細化する前の関数はそのまま継承され、新しいドメイン上の関数と合成される。ただし、新たに定義した関数が、詳細化する前の関数と同じ定義域をもっていたなら、新たな定義が有効になるように関数は合成される。

例えば、ドメインD_0上で掛け算を計算する関数mul_0は以下のように定義される。

$$mul_0 : D_0 \times D_0 \to D_0$$
$$mul_0 = \{\ (Num,Num) \mapsto Num\ \}$$

ドメインD_1上での関数mul_1は以下のように定義される。

$$mul_1 : D_1 \times D_1 \to D_1$$
$$mul_1 = mul_0 \uplus$$
$$\{\ (Neg,0) \mapsto 0,\ (0,Neg) \mapsto 0,\ (0,0) \mapsto 0,$$
$$(Neg,Neg) \mapsto Pos, (Pos,Pos) \mapsto Pos,$$
$$(Pos,Neg) \mapsto Neg,\ (Neg,Pos) \mapsto Neg\ \}$$

ここで"$\uplus$"は、関数の演算で1項目の関数と2項目の関数との和をとる。ただし、1項目と2項目が同じ定義域を持つ時、2項目の値を関数とする。

さらに、ドメインD_1にEvenとOddを追加し、D_2に詳細化すると、その上での関数mul_2は以下のように定義される。

$$mul_2 : D_2 \times D_2 \to D_2$$
$$mul_2 = mul_1 \uplus$$
$$\{\ (Even,Odd) \mapsto Even,\ (Odd,Even) \mapsto Even,$$
$$(Odd,Odd) \mapsto Odd, (Even,Even) \mapsto Even\}$$

mul_2はドメインD_2上の全ての抽象的な値に対し、関数の詳細度に対応して抽象的な値を返す。

---

*Incremental program creation based on data domain refinements
†YOSHIOKA Nobukazu, SUZUKI Masato, KATAYAMA takuya
‡Japan Advanced Institute of Science and Technology

3 具体例

この節では、今までにで述べた段階的詳細化の原理に基づいてプログラムを定義する例を示す。

例2 整数を引数にとり、その整数に対応する英語の読みを返す関数 *conv* を定義する。関数 *conv* は例えば次のような計算をする。

$$conv(3) = \text{"three"}$$
$$conv(26) = \text{"twenty-six"}$$

ドメイン詳細化の方針 入力のドメイン D を以下のように D_0、D_1、D_2 と詳細化し、それに伴って関数 *conv* を構築する。

$$D_0 = \langle \{ Num \}, \{\} \rangle$$
$$D_1 = \langle \{ Num, One\text{-}digits, Teens, Tens, Three\text{-}up \}, \{(Num, One\text{-}digits), (Num, Teens), (Num, Tens), (Num, Three\text{-}up)\} \rangle$$
$$D_2 = \langle \{ Num, One\text{-}digits, Teens, Tens, Three\text{-}up, 0...9, 20...99 \}, \{(Num, One\text{-}digits), (Num, Teens), (Num, Tens), (Num, Three\text{-}up)\} \rangle$$

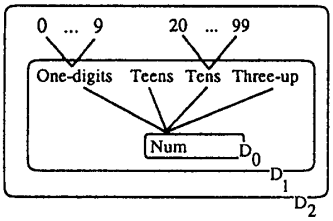

図 2: *conv* のドメインの構造

D_0、D_1、D_2 の構造は図 3 のようになる。最初にドメイン D_0 に対する関数を定義する。関数 $conv_0$ は、全ての入力に対して"*any*"を返す。ここで、S は文字列全体の集合を表す。

$$conv_0 : D_0 \to S$$
$$conv_0 = \{ Num \mapsto \text{"any"} \}$$

次に、要素 *Num* を *One-digits*、*Teens*、*Tens*、*Three-up* に詳細化したドメイン D_1 に対する関数 $conv_1$ を定義する。ここで、*One-digits* は 1 桁の数、*Teens* は 10 代の数、*Tens* は 20 以上の 2 桁の数、*Three-up* は 3 桁の数を表す値である。ここでは、新たに加えた要素のうち、*Teens* に対してのみ関数を定義する。その結果、ドメイン D_1 の要素のうち関数が定義されなかったものはドメイン D_0 上の関数 $conv_0$ の値がそのまま用いられる。結局、関数 $conv_1$ の定義は次のようになる。

$$conv_1 : D_1 \to S$$
$$conv_1 = conv_0 \uplus \{ Teens \mapsto \text{"teens"} \}$$
$$= \{ x \mapsto \text{"any"}, Teens \mapsto \text{"teens"} \mid x \neq Teens \}$$

更に、要素 *Tens* を *20* から *99*、要素 *One-digits* を *0* から *9* までの値に詳細化したドメイン D_2 に対する関数 $conv_2$ を定義する。関数 $conv_2$ は、まず入力と 1 の位の数と 10 の位の数に分け、10 の位の数に対応する文字列と 1 の位に対応する文字列を連結する。

補助関数 *nty*、*ones*、*digits*、*combine* を以下のように定義する。関数 *digits* は数を 10 の位と 1 の位に分けて返し、関数 *nty* と関数 *nty* はそれぞれ 1 の位と 10 の位に対応する文字列を返し、関数 *combine* は 10 の位の数と 1 の位の数からそれぞれに対応する文字列を連結して返す。

$$digits : D_2 \to (D_2, D_2)$$
$$digits = \{x \mapsto (a,b) \mid a = x \text{ div } 10,\ b = x \bmod 10\}$$
$$nty : D_2 \to S$$
$$nty = \{2 \mapsto \text{"twenty"}, \ldots, 9 \mapsto \text{"ninety"}\}$$
$$ones : D_2 \to S$$
$$ones = \{0 \mapsto \text{"zero"}, \ldots, 9 \mapsto \text{"nine"}\}$$
$$combine : (D_2, D_2) \to S$$
$$combine = \{ (x,0) \mapsto s_1, (0,y), \mapsto s_2\ (x,y) \mapsto s_3 \mid s_1 = nty(x), s_2 = ones(x), s_3 = nty(x) \mathbin{++} \text{"-"} \mathbin{++} conv(y)\}$$

ここで、"++" は文字列を連結する演算子である。

これらの関数を用いると関数 $conv_2$ は以下のように定義できる。

$$conv_2 : D_2 \to S$$
$$conv_2 = \begin{cases} combine \circ digits & (x \text{ が } 0...9, 20...99) \\ conv_1 & (\text{その他}) \end{cases}$$

4 不完全なプログラムの解釈

関数を詳細化する時、引数の値に対して関数が定義されていない場合、引数をそのドメインの構造に従ってより抽象的な値と解釈し、その値に対して関数を呼び出しその結果を元の関数の値とする。すなわち、$f(x)$ の値は次のように定義できる。

定義3 f_n を x を含む一番詳細なドメイン D_n 上の関数とし、$x'(\neq x)$ を $x' \preceq x$ となる一番詳細な値とすると、$f(x)$ の値は次のように計算する。

$$f(x) = \begin{cases} f_n(x) & (f_n(x) \text{ が定義されている時}) \\ f(x') & (f_n(x) \text{ が定義されていない時}) \end{cases}$$

例えば、例 2 の関数 *conv(56)* を計算すると、値は"*fifty-six*" となるが、1 桁の数について定義していない場合、つまり *conbine(0,x)* を定義せず、$conv_2$ が、

$$conv_2 = \begin{cases} combine \circ digits & (x \text{ が } 20...99) \\ conv_1 & (\text{その他}) \end{cases}$$

となる不完全なプログラムの場合にも次のように関数の値を決めることができる。

$$\begin{aligned} & conv_2(56) \\ \Rightarrow\ & combine(digits(56)) \\ \Rightarrow\ & combine(5, 6) \\ \Rightarrow\ & \text{"fifty"} \mathbin{++} \text{"-"} \mathbin{++} conv_2(6) \quad (conv_2(6) \text{ が未定義}) \\ \Rightarrow\ & \text{"fifty-"} \mathbin{++} conv_2(One\text{-}digits) \quad (One\text{-}digits \preceq 6) \\ \Rightarrow\ & \text{"fifty-"} \mathbin{++} \text{"any"} \\ \Rightarrow\ & \text{"fifty-any"} \end{aligned}$$

5 おわりに

本論文では、以下の事について述べ、その有効性を例題によって示した。

1. ソフトウェアの詳細化の中間段階において、プログラムの解釈、実行の方法を与えた。
2. ソフトウェアの詳細化をドメインの詳細化という観点から捉えドメインの詳細化を形式的に与えた。
3. ドメイン詳細化に基づいてプログラムを段階的に詳細化する方法を与えた。

これによって、ソフトウェアの問題点やバグが早期に発見できるという利点があることが確認できた。

また、ここではプログラムの詳細化をその質的な側面からしか行わなかったが、量的な側面を捉えた場合、関数の機能を段階的に強化したり、重要な変数からプログラムを構成する方法が考えられる。この場合には、注目する引数毎に関数を定義するというプログラムスライシングの技法を応用できる。

今後は、プログラムの合成規則を確定とそれを計算するアルゴリズムを確立し、それを応用したソフトウェア開発環境を作成する予定である。その環境には、プログラムを支援するために、ドメインの詳細化に伴いどの関数を詳細化したら良いかのガイドラインを示すツールなどが必要であろう。また、その上で複雑で大きな例題を実際に構成してみる事で、この構成法の有効性を実証する。

参考文献

[1] 鶴巻 維男 : インクリメンタルインプリメンテーションにもとづくプログラミング. 修士論文. 1992.

[2] Samson Abramsky, Chris Hankin : Abstract Interpretation of Declarative Languages. ELLIS HORWOOD LIMITED. 1987.