Title	抽象解釈に基づく段階的プログラム構成法のための環境構築に関する研究
Author(s)	石川，俊
Citation	
Issue Date	1998-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1167
Rights	
Description	Supervisor:片山 卓也，情報科学研究科，修士

修 士 論 文

データの段階的具象化に基づく ソフトウェア構成法のための環境構築に関する研究

指導教官　片山卓也 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

石川俊

1998 年 2 月 13 日

目 次

第 1 章

はじめに

近年ソフトウェアの規模が増大している。これにともなって複雑さも大きくなり、設計、作成中にプログラムの誤りを発見することが困難になってきた。

この問題に対して、段階的詳細化によりデータ構造を決定し、そのデータ間の依存関係の定義、解析によって仕様を作成する設計手法がある。この手法は設計段階で適用するためのものであるため、詳細化の途中段階の仕様を抽象実行することがあまり考慮されていなかった。そのため、各段階で発生した誤りを発見することが難しいという問題があった。

ISDR 法はソフトウェアを簡単なものから徐々に複雑なものにしていくことでソフトウェアを段階的に構成する方法である。実際には、まず対象となる問題を抽象化して考える。そして、それに対応した抽象的なデータを処理するソフトウェアをつくる。この最初のソフトウェアを誤りのないように作成し、その後のデータの具体化にともなうソフトウェアの詳細化で新たな誤りが入らないように段階的な誤りの検証を行うことで、信頼性を保ちながらソフトウェアを構成することができる。各段階での誤りはソフトウェアを実行させる事で発見が容易になる。これは抽象解釈の技術を用いてデータが十分に具体化されていない段階においてもプログラムを実行できるようにすることにより実現している。

このような ISDR 法による抽象解釈にもとづいたプログラムを実行するための言語 AL が提案されている。AL は以下の特徴を持っている。

- プログラムの段階的な構成。
- 抽象的なデータの定義。
- 抽象的なデータの段階的な具体化。

- 具体化したデータにあわせた関数の段階的な定義。
- 関数が未定義の値域に対する近似計算による実行。

本研究では言語 AL を使ったプログラムの開発と実行を支援する環境 Alchemy を作成する。Alchemy はユーザーがデータの具体化関係を矛盾させるような操作を行おうとした場合にこれを制限することによって、誤りのないデータの具体化と関数の詳細化を実現している。また、開発環境内からユーザーが作成した AL プログラムを実行することができる。そして実装した開発環境を用いて実際にプログラムを作成し、その結果から ISDR 法による開発手法を評価する。

本論文の構成を以下に述べる。2 章ではソフトウェアの段階的構成法である ISDR 法について述べる。3 章では ISDR 法によってプログラムを構成する言語 AL の概要と AL におけるプログラムの構成について述べる。続いて 4 章で AL によるプログラムの開発、実行を支援する環境 Alchemy について述べ、5 章で Alchemy を用いて与えられた仕様を満たすプログラム開発の実例を示す。6 章で 5 章におけるプログラム開発の実例を通して ISDR 法によるソフトウェアの構成について評価する。

第 2 章

ソフトウェアの段階的構成法 ISDR 法

2.1 概要

ISDR 法ではデータの具体化という観点からプログラムを詳細化する。具体化途中のデータを抽象値として表現することで、抽象解釈の技法に基づいたプログラムの解釈と実行を可能にする。

ISDR 法ではプログラムの入出力データを具体値の集合と考える。この具体値の集合をそれぞれ一つの抽象値として定義し、その抽象値を用いてプログラムを構成することをプログラムの抽象化という。データの具体化とは、この抽象値をより具体的な値の集合に置き換え、それらの値に対応する、より詳細なプログラムを構成することである。この具体化をすべての抽象値が十分具体的になるまで繰り返すことで、目標のプログラムを構成する。

プログラムはまず**抽象化段階**でプログラムを抽象化した後、**詳細化段階**の繰り返しによって段階的に構成するが、これら各段階をバージョンとして管理する。

データの種類は抽象値と具体値に分けられる。元の抽象値があらわす領域を、より具体的な複数の抽象値または 1 つ以上の具体値によって分割するか、またはレコードなどによって置き換えることで抽象値を具体化する。

プログラムは関数によって構成される。関数はまず入出力データの領域を一つの値へ抽象化した値について作成し、以降のバージョンでそれらのデータを具体化した値に対応する関数を作成する。

抽象値上に構成されたプログラムは抽象解釈の技法を用いて実行する。ある値で関数

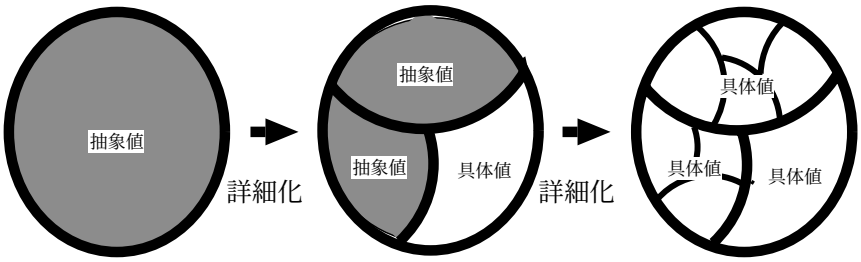


図 2.1: データ領域の具体化

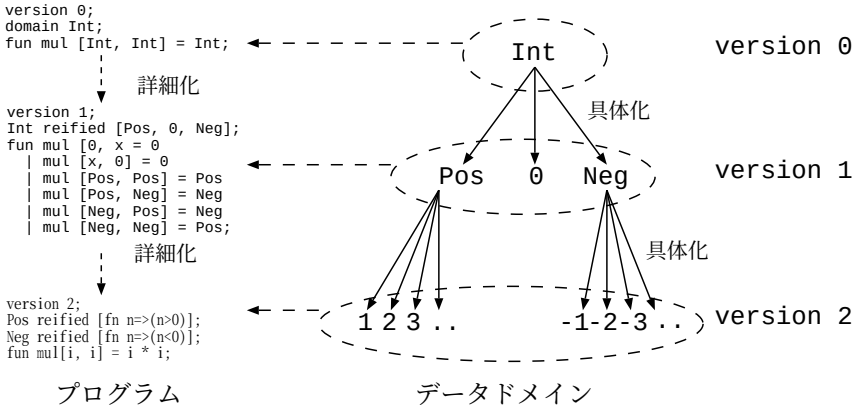


図 2.2: データの具体化と関数の対応の例

を呼び出したときに、まだその値について関数を詳細化していない場合は、代わりに同じデータ領域の高い抽象度を持つ値で作成された関数を呼び出す。これによってプログラムの動作を中断することなく実行を継続することができる。これを近似計算と呼ぶ。

ISDR 法では、まず抽象化したプログラム (原始プログラム) を作成し、これを繰り返し詳細化することで全体のプログラムを構成する。

抽象化段階 以下の手順で原始プログラムをつくる。

1. 入出力データの集合をそれぞれ一つの抽象値に抽象化する。

2. それらの値を使ってプログラムを作成する。

詳細化段階 原始プログラムを、仕様を完全に満たすまで、次の手順にしたがって繰り返し詳細化する。

1. 入出力データドメイン中の具体集合に含まれる値を一段階具体化する。

2. 具体化した値を使ってプログラムを詳細化する。

3. 詳細化したプログラムを実行し、デバッグをおこなう。

この手順のなかで現われる抽象値は仕様中の具体値の集合に対応する。

2.2 データドメイン

データは抽象値と具体値に大きく分けられる。

抽象値は最終的に具体値のみで構成される値の集合に置き換えられるまで、段階的に具体化される。具体化はその抽象値があらわす領域を、より具体的な複数の抽象値または1つ以上の具体値によって、分割することによっておこなう。また、抽象値はそれぞれプログラム中において固有の名前を持つ。

具体値は文字列型、整数型、ブール型等の型を持つ実際の値である。また、十分に具体化された値であるため、さらに具体化することはできない。

データ領域内の値の集合と、その上での値の具体化関係をデータドメインという。データ領域を一つの値に抽象化したときの抽象値名をデータドメイン名とする。値の具体化関係は木構造になる。

定義 1 データドメイン

データドメインは，以下のような値の集合 $\mathbf{S}$ とその上の具体化関係 $\prec$ の組で構成される。

$$\mathbf{D} \equiv \langle \mathbf{S}, \prec \rangle$$

ここで，$\mathbf{S}$ を $data(\mathbf{D})$, $\prec$ を $rel(\mathbf{D})$ と書く。 ∎

この定義中の $\mathbf{S}$ には，具体値または抽象値である**基本値**，**レコード**，または**再帰的なレコード**が含まれる。基本値とは，抽象値または具体値のことであり，再帰的なレコードはリストや木構造などである。

ここで，ドメインに具体化した値を追加する記法を導入する。

定義 2　ドメインへの要素の追加

$\mathbf{D}[x \prec \mathbf{S}]$ は，ドメイン $\mathbf{D}$ 中の $data(\mathbf{D})$ に集合 $\mathbf{S}$ を加え，$rel(\mathbf{D})$ に抽象値 x から $\mathbf{S}$ 中の要素への関係を追加したドメインを表す。

$$\mathbf{D}[x \prec \mathbf{S}] \quad \equiv \quad \langle data(\mathbf{D}) \cup \mathbf{S}, rel(\mathbf{D}) \cup \{x \prec y \mid \forall y \in \mathbf{S}\}\rangle$$

ドメイン $\mathbf{D}$ 中の複数の値を同時に具体化した場合も同様であり，$\mathbf{D}[x_1 \prec \mathbf{S}_1, \ldots, x_n \prec \mathbf{S}_n]$ のように書く。 ∎

2.3　プログラムとバージョン

データの具体化、関数の詳細化を段階に分けて行う。それぞれの段階をバージョンという。

関数とその入出力データは任意のバージョンから構成を始めることができる。最初の定義が ISDR 法の **抽象化段階** にあたり、その後の詳細化が **詳細化段階** にあたる。

つぎに，このドメイン上のプログラムを以下のように定義する。

定義 3　プログラム

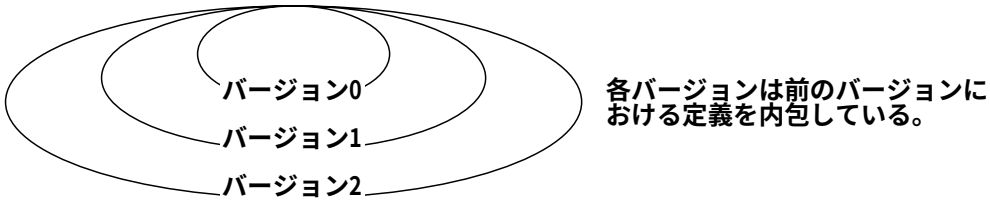
プログラム Progは，主関数 F，補助関数の集合 $\mathbf{AFS}$，ドメインの集合 $\mathbf{DS}$ の *3* つ組で構成される。

$$Prog \quad \equiv \quad \langle F, \mathbf{AFS}, \mathbf{DS}\rangle$$

ここで，F は与えられた仕様を実現する関数であり，$\mathbf{AFS}$ はその関数で使われている関数の集合となる。また，$\mathbf{DS}$ はそれらの関数で用いられているデータに関するドメインの集合である。 ∎

ここで，詳細化の各段階に定義したプログラム Progを区別するために，それぞれの段階で定義したドメイン名や関数名の添字に番号を付加する。原始プログラムには 0 を付加し Prog_0と書き，これを n 回詳細化したプログラムを Prog_nと書き**バージョン** n のプログラムとよぶ。

各バージョンでの記述はそれ以前のバージョンからの差分であるが、以前のバージョンにおける定義をすべて内包している。



各段階で定義したデータや関数を区別する場合は、番号を添字として付加する。番号は 0 から始まる連続した自然数とする。例えば $Prog_2$ のプログラムにはバージョン 0 からバージョン 2 までに定義したすべてのデータと関数が含まれる。

2.4 プログラムの詳細化

2.4.1 データの具体化

抽象データはそのデータが持つ領域を、同一のデータ型からなるデータの集合に分割するかレコードやリストに置き換えることができる。これをデータの具体化という。

各バージョンのデータドメインにおいて、もっとも具体的な値の集合を具体集合という。具体集合に含まれない抽象値について関数を定義することはできない。値を具体化するまえにその関数を定義すべきである。データドメイン中に定義されている具体的な値の集合は以下のようにあらわすことができる。

定義 4 データドメイン中の具体集合 $\uparrow\mathbf{D}$

$$\uparrow\mathbf{D} \equiv \{d \in data(\mathbf{D}) | \forall d' \in data(\mathbf{D}).(d, d') \notin rel(\underline{)}\}$$

データドメインの具体化関係

定義 5 正しく具体化されたデータドメインは以下の条件を満たす。

1. 具体化関係は合流しない。
   $\forall (x, y), (x', y') \in rel(\mathbf{D}).y = y' \Rightarrow x = x'$
2. 具体化関係はループしない。$\forall (x, y) \in rel(\mathbf{D}).(y, x) \notin rel^*(\mathbf{D})$
3. ドメイン中には不必要な値は含まれていない。
   $\forall x, y \in data(\mathbf{D}), \exists z \in data(\mathbf{D}).(z, x) \in rel^*(\mathbf{D}) \wedge (z, y) \in rel^*(\mathbf{D})$

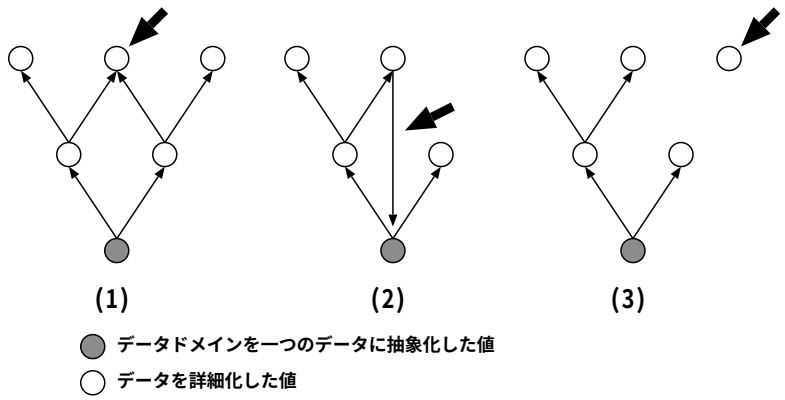


図 2.3: データドメインの具体化関係

ここで $rel^*(\mathbf{D})$ は $rel(\mathbf{D})$ の反射的閉包である。

データドメインの重複

複数のデータドメインが重複したり、包含関係になることがある。

例えば、ドメイン 1 とドメイン 2 の領域が A において部分的に重複している場合にこの領域を抽象値 A とすると、A の具体化にともなって、それぞれのドメインの該当領域も同時に具体化される。(図 2.4)

2.4.2 プログラムの詳細化

プログラムを詳細化するためには，前節のように具体化された値に関してプログラムを定義すればよい。具体化された値は，ドメインを木構造とみたときの葉の内のいずれかである。そこで，次のような記号を導入する。

定義 6 ドメイン中の具体集合

$$\uparrow \mathbf{D} \equiv \{d \in data(\mathbf{D}) | \forall d' \in data(\mathbf{D}).(d, d') \notin rel(\mathbf{D})\}$$

この定義を用いると詳細化した主関数 F の型は，具体化した入出力ドメインをそれぞれ $\mathbf{D}^{\mathrm{I}}$, $\mathbf{D}^{\mathrm{O}}$ とすると，$\uparrow \mathbf{D}^{\mathrm{I}} \rightarrow \uparrow \mathbf{D}^{\mathrm{O}}$ となる。ここで，F はその定義域 $\uparrow \mathbf{D}^{\mathrm{I}}$ 中の値すべてに対して定義する必要はなく部分関数でよい。プログラム Prog は以下の定義をみたすように構成する。

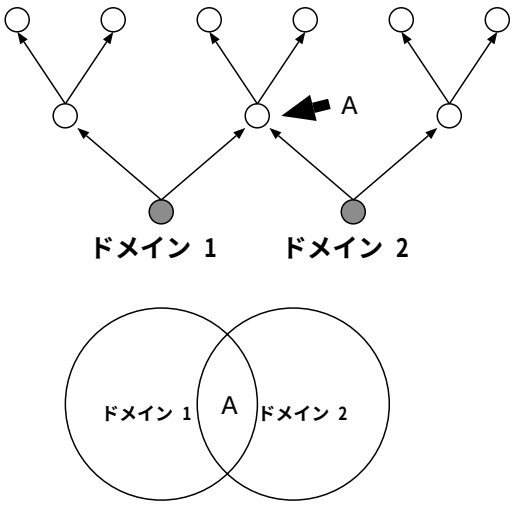


図 2.4: 領域が重複するデータドメイン

定義 7 プログラムの構成

$\mathrm{Prog} \equiv \langle F, \mathbf{AFS}, \mathbf{DS}, \mathbf{RS} \rangle$ とすると，

$$\forall \mathbf{R} \in \mathbf{RS}, \exists \mathbf{D} \in \mathbf{DS}. \quad \mathbf{R} = \uparrow \mathbf{D},$$
$$\forall f \in \{F\} \cup \mathbf{AFS}. \quad f : \mathbf{R} \to \mathbf{R}' \Rightarrow \mathbf{R}, \mathbf{R}' \in \mathbf{RS}$$

∎

ここで定義した関数は，以前のバージョンの関数より具体的な値を処理する関数となっている。そこで，関数の詳細化を形式的に定義すると次のようになる。

定義 8 関数の詳細化

関数 f'が fを詳細化しているとは，fの型が$\uparrow \mathbf{D}1 \times \uparrow \mathbf{D}2 \times \ldots \times \uparrow \mathbf{D}m \to \uparrow \mathbf{D}$, f'の型が$\uparrow \mathbf{D}1' \times \uparrow \mathbf{D}2' \times \ldots \times \uparrow \mathbf{D}m' \to \uparrow \mathbf{D}'$とすると，次の条件が成り立つ時である。

$$\mathbf{D}1 \sqsubseteq \mathbf{D}1' \wedge \ \ldots \ \wedge \mathbf{D}m \sqsubseteq \mathbf{D}m' \wedge \mathbf{D} \sqsubseteq \mathbf{D}' \wedge$$
$$\forall x1 \in \uparrow \mathbf{D}1, \ldots, xm \in \uparrow \mathbf{D}m, \forall x1' \in \uparrow \mathbf{D}1', \ldots, xm' \in \uparrow \mathbf{D}m'.$$
$$f(x1, \ldots, xm) = y \wedge f'(x1', \ldots, xm') = y' \wedge x1 \preceq x1' \wedge \ \ldots \ \wedge xm \preceq xm' \Rightarrow y \preceq y'$$

この時，$f \sqsubseteq f'$と書く。 ∎

この定義の一行目の論理式は f' の入出力ドメインの方が f より具体的であることを表し，それ以降の論理式は，f' の定義は f と矛盾していないことを表している。

これまでの定義を用いて，ISDR 法におけるプログラムの詳細化を次のように定義する。

定義 9　プログラムの詳細化

プログラム *Prog'* が *Prog* を詳細化しているとは，$Prog = \langle F, \mathbf{AFS}, \mathbf{DS} \rangle$，$Prog' = \langle F', \mathbf{AFS}', \mathbf{DS}' \rangle$ とすると，つぎの条件が成り立つ時である。

$$F \sqsubseteq F' \wedge \mathbf{AFS} \subseteq \mathbf{AFS}' \wedge \mathbf{DS} \subseteq \mathbf{DS}'$$

このとき，$Prog \sqsubseteq Prog'$ と書く。 ∎

2.5　プログラムの抽象解釈

この節では，定義されたプログラムをどのように解釈，実行するかをのべる。

プログラム中の関数の実行はドメイン中の値とその構造を利用して行う。ISDR 法で定義する関数は，抽象値を入出力データとしているので，このまま抽象解釈することで実行することができる。ただし，この関数は部分関数であり完全な形で定義されていないので，定義されていない部分に関しては以前のバージョンの関数を利用する。すなわち，入力データがまだ定義されていない領域のデータに対しては，まずその値を以前のバージョンの関数に合せて抽象化し，そしてその値を使って以前の関数を呼び出す。これによって部分関数 F_n は全関数 $\tilde{F}_n$ に変換でき，その関数は必要なだけ抽象化された値を返すことができるようになる。この原理を形式的に定義すると次のようになる。

定義 10　主関数の解釈

主関数 F_n の型を $\uparrow \mathbf{D}1_n \times \ldots \times \uparrow \mathbf{D}m_n \rightarrow \uparrow \mathbf{D}_n$，$F_{n-1}$ の型を $\uparrow \mathbf{D}1_{n-1} \times \ldots \times \uparrow \mathbf{D}m_{n-1} \rightarrow \uparrow \mathbf{D}_{n-1}$ とすると，

1. $\tilde{F}_n$ の型はつぎのようになる。

$$F_n : \uparrow \mathbf{D}1_n \times \ldots \times \uparrow \mathbf{D}m_n \rightarrow data(\mathbf{D}_n)$$

2. 原始プログラムの主関数 F_0 はそのままで全関数 $\tilde{F}_0$ である。($\tilde{F}_0 \equiv F_0$)

3. バージョン n の全関数$\tilde{F}_n$は，次のように F_nで定義されている領域についてはそれを呼び出し，それ以外の領域については前のバージョンの全関数$\tilde{F}_{n-1}$を呼び出す。

$$\tilde{F}_n(x1, x2, \ldots, xm) \equiv \begin{cases} y & (F_n(x1, \ldots, xm) = y\text{の場合}) \\ y' & (\text{それ以外の場合}) \end{cases}$$

ただし，$x1' \in\uparrow \mathbf{D}1_{n-1} \wedge \ldots \wedge xm' \in\uparrow \mathbf{D}m_{n-1} \wedge (x1', x1) \in rel^*(\mathbf{D}1_n) \wedge \ldots \wedge (xm', xm) \in rel^*(\mathbf{D}m_n)$ について，$\tilde{F}_{n-1}(x1', \ldots, xm') = y'$とする。 ∎

この解釈にしたがうとバージョン n の主関数 Fを全域関数$\tilde{F}$に変換するためには，結局つぎのようにそれ以前のバージョンで定義したすべての主関数が必要になる。

主関数以外の **AFS** の中の関数について全域化する必要は無く，全域化しない場合は，上記の定義の 3 のみで計算できる。主関数以外の関数を全域化しなくとも，主関数だけ全域化しておけば，全体のプログラムは必ず全ての入力に対して何らかの値が計算できるようになる。

2.6 プログラムの詳細化方針

この節では，入力や出力が具体化された時，関数をどのように詳細化すればよいかの方針を示す。関数の詳細化は，具体化されたデータによってここで示すテンプレートにしたがって行うことができる。テンプレートはデータが基本値に具体化された場合，レコードに具体化された場合，再帰レコードに具体化された場合に分れている[1]。

一般に任意の 2 つの関数に対し，定義 8の詳細化関係が成り立っていることを調べるのは困難であるが，ここで示すテンプレートのみを適用して得られた関数には必ず詳細化関係が成り立っている。

まず，f_nがつぎのように定義されているとする。

$$\begin{aligned} \mathbf{D}_n &= \langle \{A\}, \emptyset \rangle \\ \mathbf{D}'_n &= \langle \{B\}, \emptyset \rangle \\ \textbf{fun} \quad f_n &:\uparrow \mathbf{D}_n \rightarrow\uparrow \mathbf{D}'_n \\ \textbf{fun} \quad f_n(A) &= B \end{aligned}$$

[1] これだけで全てのデータ構造を表現可能である。

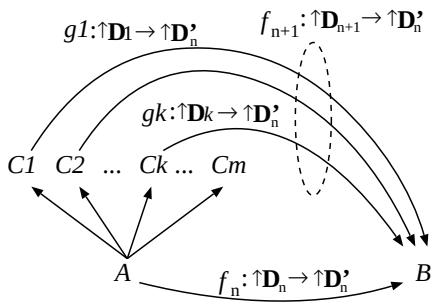


図 2.5: テンプレート 1

説明を簡単にするために，ここでは f_n を唯一の要素に対する一引数関数と仮定する。しかしながら，プログラムを詳細化する場合，ただ一つのドメインのみを具体化し全ての関数の入出力ともに違うドメインで定義できるならば，それらの関数は具体化したドメインに対しここで示すテンプレートのいずれかを適用することが可能である。もし，関数の入出力が同じドメインになるならば，2 つ以上のテンプレートを合成した形でその関数を詳細化すれば詳細化関係を保つことが可能である。

2.6.1 テンプレート 1: 入力を基本値の集合へ具体化した場合

関数 f_n の入力 A を 基本値の集合 $(C1, C2, \ldots, Cm)$ へ具体化した時，f_n は次のように f_{n+1} へ詳細化できる。(図 2.5 [2])

$$
\begin{aligned}
&\mathbf{D}_{n+1} = \mathbf{D}_n[A \prec \{C1, C2, \ldots, Cm\}] \\
&\mathbf{fun}\ f_{n+1}(C1) \ = g1(C1) \\
&\qquad\qquad\qquad\ \vdots \\
&\quad \mid f_{n+1}(Cm) = gm(Cm) \\
&\mathbf{and}\ g1(C1) \quad\ = B \\
&\qquad\qquad\qquad\ \vdots \\
&\mathbf{and}\ gm(Cm) \ = B;
\end{aligned}
$$

ここで $g1, \ldots, gm$ は新しく導入された関数であり，これ以降の段階で詳細化される。また，上記の定義中の g や C の後ろの数字は名前の一部でありバージョン番号ではない。

[2] 図 2.5から図 2.8の中で，$\mathbf{D}1, \ldots, \mathbf{D}m$ $\mathbf{D}1', \ldots, \mathbf{D}m'$ は，それぞれ $C1, \ldots, Cm, C1', \ldots, Cm'$ を唯一要素に持つドメインである。

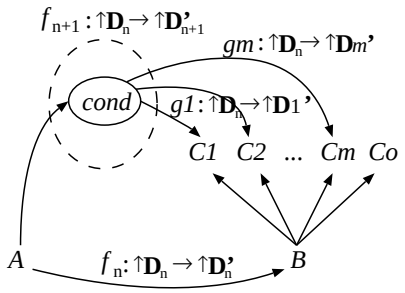


図 2.6: テンプレート 2

2.6.2 テンプレート 2: 出力を基本値の集合へ具体化した場合

関数 f_nの出力 B を 基本値の集合 $(C1, C2, \ldots, Cm)$ へ具体化した時，f_nは次のように f_{n+1}へ詳細化できる。(図 2.6)

$$\mathbf{D}'_{n+1} = \mathbf{D}'_n[B \prec \{C1, C2, \ldots, Cm\}]$$

$$\begin{aligned}
&\textbf{fun } f_{n+1}(A) = \textbf{if } cond1(A) \textbf{ then } g1(A)\\
&\qquad\qquad \textbf{else if } cond2(A) \textbf{ then } g2(A)\\
&\qquad\qquad\qquad \vdots\\
&\qquad\qquad \textbf{else if } condm(A) \textbf{ then } gm(A)\\
&\textbf{and } g1(A) \;= C1\\
&\qquad\qquad \vdots\\
&\textbf{and } gm(A) = Cm\\
&\textbf{and } cond1(A) \;= \text{bool}\\
&\qquad\qquad \vdots\\
&\textbf{and } condm(A) = \text{bool};
\end{aligned}$$

ここで，bool は真偽値のドメイン **Bool** の要素であり，これ以降の詳細化段階で true か false に具体化される。この時点でプログラムを実行するためには実行時に true か false を選択する必要である。

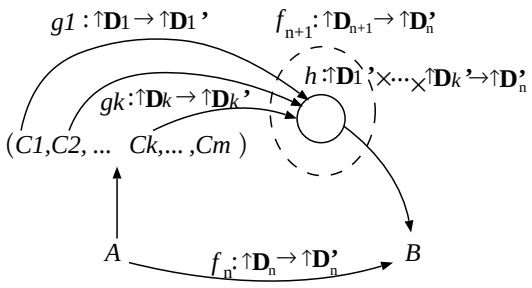


図 2.7: テンプレート 3

2.6.3 テンプレート 3: 入力をレコードへ具体化した場合

関数 f_n の入力 A を m 個のメンバを持つレコード $(C1,\ C2,\ \ldots,\ Cm)$ へ具体化した時，f_n は次のように f_{n+1} へ詳細化できる。(図 2.7)

$\mathbf{D}_{n+1} = \mathbf{D}_n[A \prec (C1, C2, \ldots, Cm)]$

$$\begin{aligned}
&\textbf{fun } f_{n+1}((C1,\ldots,Cm)) = h(g1(A),\ldots,gm(Cm))\\
&\textbf{and } g1(C1) = E1\\
&\qquad\qquad\vdots\\
&\textbf{and } gm(A) = Em\\
&\textbf{and } h(E1,\ldots,Em) = B;
\end{aligned}$$

ここで $E1,\ \ldots,\ Em$ はこのバージョンで新しく導入された中間データであり，これ以降のバージョンで具体化される。

2.6.4 テンプレート 4: 出力をレコードへ具体化した場合

関数 f_n の出力 B を m 個のメンバを持つレコード $(C1,\ C2,\ \ldots,\ Cm)$ へ具体化した時，f_n は次のように f_{n+1} へ詳細化できる。(図 2.8)

$\mathbf{D}'_{n+1} = \mathbf{D}'_n[B \prec (C1, C2, \ldots, Cm)]$

$$\begin{aligned}
&\textbf{fun } f_{n+1}(A) = (g1(A),\ldots,gm(A))\\
&\textbf{and } g1(A) = C1\\
&\qquad\qquad\vdots\\
&\textbf{and } gm(A) = Cm;
\end{aligned}$$

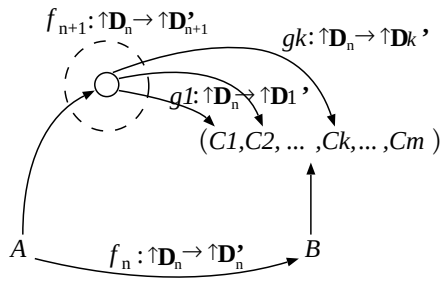


図 2.8: テンプレート 4

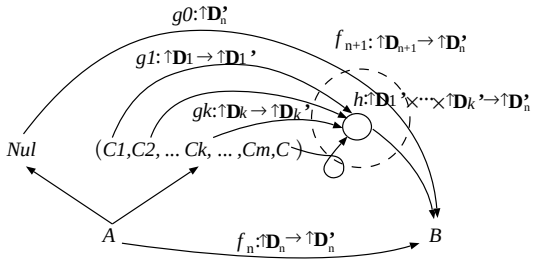


図 2.9: テンプレート 5

2.6.5 テンプレート 5: 入力を再帰的なデータ構造へ具体化した場合

関数 f_n の入力 A を 再帰的に定義されたデータ構造 Cへ具体化した時，f_nは次のように f_{n+1}へ詳細化できる。(図 2.7)

$\mathbf{D}_{n+1} = \mathbf{D}_n[A \prec C] \quad \textbf{where } C = Nul \mid (C1, \ldots, Cm, C)$

$\textbf{fun } f_{n+1}(Nul) = g0()$

$\quad \mid f_{n+1}((C1, \ldots, Cm, C)) \ = \ h(g1(C1), \ldots, gm(Cm), f(C))$

$\textbf{and } g0() = B$

$\textbf{and } g1(C1) \quad = E1$

$\qquad\qquad\qquad \vdots$

$\textbf{and } gm(Cm) = Em$

$\textbf{and } h(E1, \ldots, Em, B) = B;$

この規則では一般性を失うことなくレコードはその中の一カ所で再帰しているとしている。また，$E1, \ldots, Em$ はこのバージョンで新しく定義された中間データであり，関数

f_{n+1}の定義中の fは 2.5 節で示すメカニズムによって解釈される最新バージョンの関数である。

再帰的なレコードは通常リスト構造を表現するために使われる。リストを処理する時，n 個の要素を先読みすると関数の定義が容易になる場合がある。そのような場合，次のように詳細化を行う。

$$
\begin{aligned}
&\textbf{fun}\ f_{n+1}(Nul) = g0() \\
&\quad \mid f_{n+1}((C1,\ldots,Cm,C)) = g(C1,\ldots,Cm,C) \\
&\textbf{and}\ g(C1,\ldots,Cm,Nul) = g0'(C1,\ldots,Cm) \\
&\quad \mid g(C1,\ldots,Cm,(C1',\ldots,Cm',C)) \ = \ h(g1(C1),\ldots,gm(Cm),g(C1',\ldots,Cm',C))
\end{aligned}
$$

この定義では，レコードの次の要素 $(C1',\ldots,Cm')$ を先読みし，gでその要素を使ってレコードの先頭の要素 $(C1,\ldots,Cm)$ の処理の処理を行っている。

また，入力を先頭からグループに分けてそのグループに対して処理う事が既に分かっている場合，このテンプレートの様に関数 h によって先頭から全ての要素を走査するような方法は，グループ分けをするという情報を直接表していない。この様な処理は，グループ毎に小計出す場合など一般的なプログラミングに多々現われる。しかし，この時点では仕様からグループ分けをすることは読み取れても，入力データが具体的でないため，プログラムにそれを表現することができない。

そこで，グループ分けの処理を行うことをこの時点で表現するために，“@” によるパターンマッチを導入する。この記述を使うとテンプレート 5 は次のように書き変わる。

$$
\begin{aligned}
\textbf{fun}\quad & f_{n+1}(Nul) = g0() \\
\mid\quad & f_{n+1}(c1@c2) \ = \ h'(g(c1),f(c2)) \\
\textbf{and}\quad & h_{n+1}(E,B) = B
\end{aligned}
$$

この中で gはグループ毎の計算を行う関数であり，その定義は元のテンプレートの fの定義と全く同じ形とする。そして，Eはグループの結果を表す抽象値とする。また，h'はグループ毎の結果を最終的な値に加味する関数とする。

これによって，関数 fにはグループ分けの処理が含まれていて，これ以降の詳細化によってその処理が現われることが分かるようになる。[3] しかし，“@” を使ったテンプレートは元のテンプレートの特殊形であり，この拡張によって言語の処理能力自体は変化しない。

[3] この時点で抽象実行するためには，bool の場合と同様，ユーザがグループ分けの処理を実行時に行う必要がある。

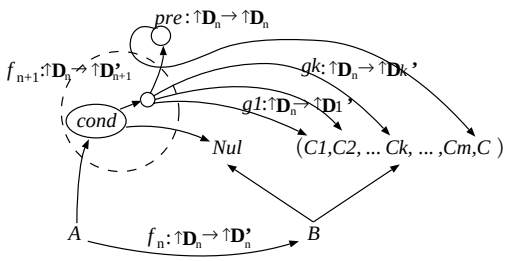


図 2.10: テンプレート 6

2.6.6 テンプレート 6: 出力を再帰的なデータ構造へ具体化した場合

関数 f_nの出力 B を 再帰的に定義されたデータ構造 Cへ具体化した時，f_nは次のように f_{n+1}へ詳細化できる。(図 2.8)

$\mathbf{D}'_{n+1} = \mathbf{D}'_n[B \prec C]$ **where** $C = Nul \mid (C1, \ldots, Cm, C)$

fun $f_{n+1}(A) =$ **if** $Cond(A)$ **then** Nul

else $(g1(A), \ldots, gm(A), f(pre(A)))$

and $g1(A) = C1$

$\vdots$

and $gm(A) = Cm$

and $pre(A) = A$

and $Cond(A) = \mathrm{bool};$

ここで，pre は A から具体化された具体値で構成できるチェイン $(A^*, <)$ 上の単調減少関数である。そして，その関数は $pre^*(x) < x$ が成り立ち，そのチェインのボトム $(\perp)$ に対しては $Cond(\perp) = \mathrm{true}$ が成立する。

第 3 章

言語 AL

3.1 言語の概要

ISDR 法に沿ってプログラムを構成するための言語 AL について述べる。AL は ML を基にいくつかの拡張を加えた言語であり、ML にくらべて以下の特徴を持つ。

- プログラムの段階的な構成が可能。
- 関数の定義域と値域をそれぞれデータドメインとして定義する。
- データドメインを段階的に具体化する。
- データドメインを詳細度に合わせて段階毎に関数を定義する。
- 関数が未定義である値域に対する近似計算による実行が可能。

3.2 段階的な構成

AL ではプログラムを複数の段階に分けて構成することができる。各段階において、以下の定義を記述する。

1. データドメインの定義
2. データの具体化

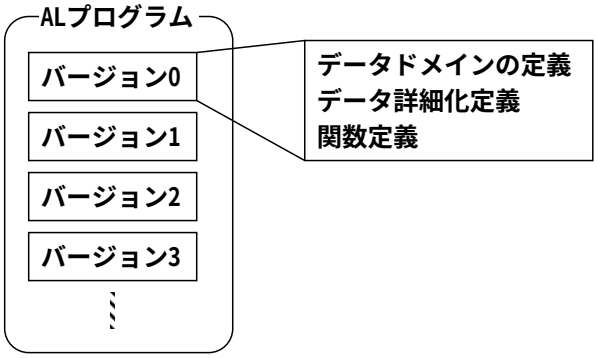


図 3.1: AL プログラムの構造

3. その段階におけるデータドメインに対応した関数の定義

ドメインを定義する予約語には version、domain、reified がある。段階の開始は version で宣言する。各段階にはバージョン番号を指定する。この番号は作成順に 0 1 2 3 ... のように割り振る。バージョン 0 の開始を宣言する場合は以下のように記述する。

```
version 0;
```

これ以降、新たにバージョンの宣言をするまで、バージョン 0 に対しての定義となる。

各段階では前段階から一段階だけ具体化したデータや詳細化した関数を記述する。データの具体化や関数の詳細化が複数段階になる場合はその段階ぶんだけ各バージョンに渡って行う。

3.3 データドメインの定義

データドメインは domain にデータドメイン名を与えることで定義する。

以下の例はデータドメイン Indom を定義する。

```
domain Indom;
```

ここで Indom は抽象値として定義される。後の段階ではこの抽象値を具体化した値に対して関数を定義する。

3.4 データドメインの具体化

前節で述べたようにデータドメインはドメイン名とともに抽象値としても定義される。抽象値はその値があらわす領域を分割する複数の値で具体化することができる。具体化は `reified` 関数を使って、もとの抽象値と具体化した値のリストを与えることで行う。`reified` 関数は infix として定義してある。

以下は抽象値 `Indom` をふたつの文字列 (prefix `s` を持つ) に具体化する例である。

```
Indom reified [s "Foo", s "Bar"];
```

具体値があらわす型には整数型や文字列型などがあるが、それらの型があらわす値の領域は互いに交わらない。また、データドメインはそれぞれいずれかの型があらわす領域に完全に含まれる。よって、一つのデータドメイン中に含まれる具体値の型は一種類とする。

また AL では単一の段階で連続して具体化できないように制限する。バージョン n において、データを具体化するときは具体集合$\uparrow\mathbf{D}_{n-1}$に含まれる抽象値に対して、データの集合を与えることによって元の値が表す領域を分割する。関数はバージョン n での具体集合$\uparrow\mathbf{D}_n$に対して定義するので、バージョン n の具体集合に含まれる値から具体化すると、関数を定義できない値が発生する。このため、直前のバージョンの具体集合に含まれる値について具体化することで、関数を定義できないデータが発生する可能性を排除する。つまり一つのバージョンの中において、ある値を具体化して現れた値を更に具体化することはできない。(図 3.4)

定義 11 *AL* におけるデータドメインの具体化 $\mathbf{D}_n[x \prec \mathbf{S}]$

$$\mathbf{D}_n[x \prec \mathbf{S}] \equiv \langle data(\mathbf{D}_{n-1}) \cup \mathbf{S}, rel(\mathbf{D}_{n-1} \cup \{x \prec y | \exists x \in \uparrow\mathbf{D}_{n-1}, \forall y \in \mathbf{S}\}\rangle$$

3.5 データ型

AL は抽象型、文字列型、整数型、ブール型、レコード型、リスト型、条件型のデータ型を持つ。

データはそれぞれの型に合わせた prefix を持つ。

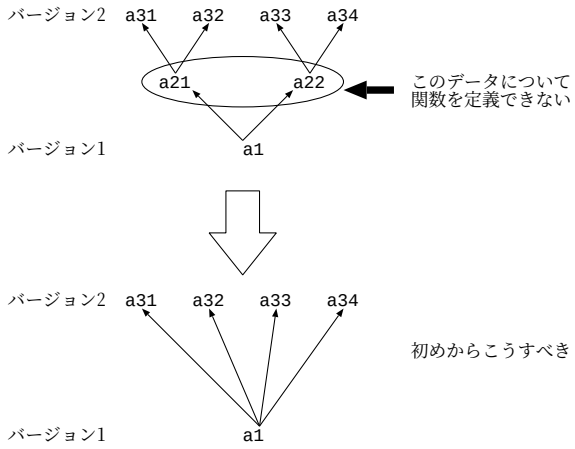


図 3.2: 単一バージョンでの連続した具体化

抽象型

値を抽象化したデータ型で唯一具体化可能な型である。抽象型として定義されたデータは a の prefix を持ち、各々固有のデータ名を持つ。

抽象値 Indom を抽象値 Foo Bar に具体化する場合は、以下のように書く。

```
Indom reified [a Foo, a Bar];
```

文字列型 (s, Strings)

文字列型は prefix として s を持ち、文字列の具体値をあらわす。任意の文字列の具体値をあらわす場合はとくに Strings と記述する。これは prefix s を持つ任意の文字列にマッチする。

ひとつのドメインに s 文字列 と Strings を混在させることができるが、この場合は s 文字列 で定義した文字列以外の文字列が Strings になる。

例えば、抽象値 Indom を "foo" 、"bar" とその他の文字列に具体化する場合は以下のように書く。

```
Indom reified [s "foo", s "bar", Strings];
```

整数型 (i, Integers)

整数型は prefix として i を持ち、整数の具体値をあらわす。任意の整数の具体値をあらわす場合はとくに Integers と記述する。これは prefix i を持つ任意の値にマッチする。

ひとつのドメインに i 整数 と Integers を混在させて具体化ことができるが、この場合は i 整数 で定義した整数以外の整数が Integers になる。

例えば、抽象値 Indom を 3 、4 とその他の整数に具体化する場合は以下のように書く。

```
Indom reified [i 3, i 4, Integers];
```

条件型 (condi)

整数の特定の範囲をデータの領域として定義する場合に使用する。実態は fn : (int -> bool) の型を持つ匿名関数である。抽象値 Indom を 0 より大きい値を持つ整数の領域に具体化する場合は以下のように書く。

```
Indom reified [condi(fn n=>(n > 0))]
```

(なんで fn : (Id -> bool) じゃないのだろう)

ブール型 (Bool, true, false)

ブール型には真または偽をあらわす Bool とそれぞれの真理値 true、false がある。抽象型を true と false の任意の組合せ、または Bool で具体化することができる。AL は Bool を処理する時、真偽のどちらで処理すべきかユーザーに問い合わせる。

レコード型 (r)

レコード型は複数データの組をあらわす。prefix r とデータのリストで構成する。

レコードを定義するときに、それを構成するデータの型は任意である。また、すべて定義する段階において最も具体的な値でなければならない。

例えば、抽象値 Indom を複数の抽象値を持つレコードに具体化する場合は以下のように書く。

```
Indom reified [r[a Foo, a Bar, a Hoge]];
```

レコードを直接具体化することはできないが、レコード内の抽象値を具体化することで、実質的に具体化することができる。

例えば、データ a1 と b1 、レコード (a1,b1) があるとき、a1 を a21 と a22 で具体化すると、レコード (a1,b1) も同時に (a21,b1) と (a22,b1) へ具体化される。b1 を具体化した場合も同様にレコードが具体化される。(図 3.3)

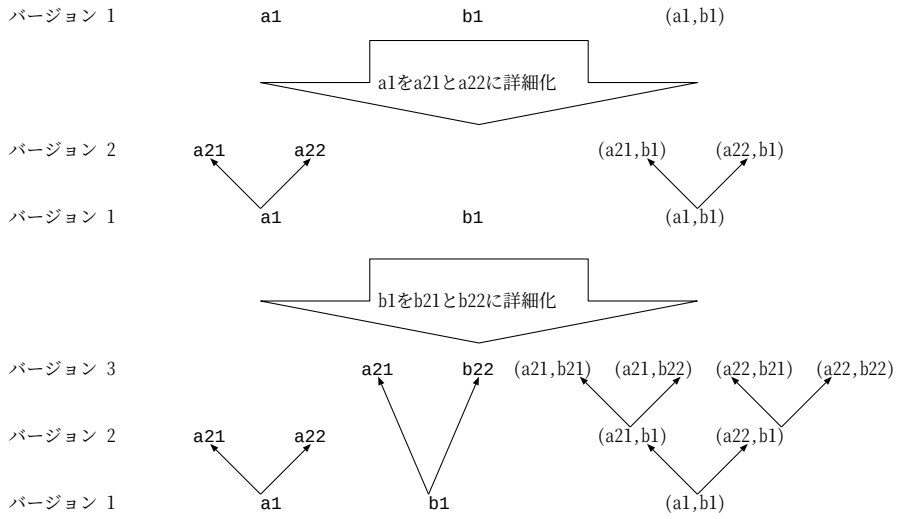


図 3.3: レコードの具体化

リスト型 (l)

ほぼレコード型と同様である。prefix l とデータのリストで構成する。ただし要素はすべて同じ型である。

例えば、抽象値 Indom を抽象値 Foo のリストに具体化する場合は以下のように書く。

```
Indom reified [l[a Foo]];
```

後の段階で a Foo を a Bar と a Hoge に具体化すると、このリストの要素は a Bar または a Hoge を持つようになる。

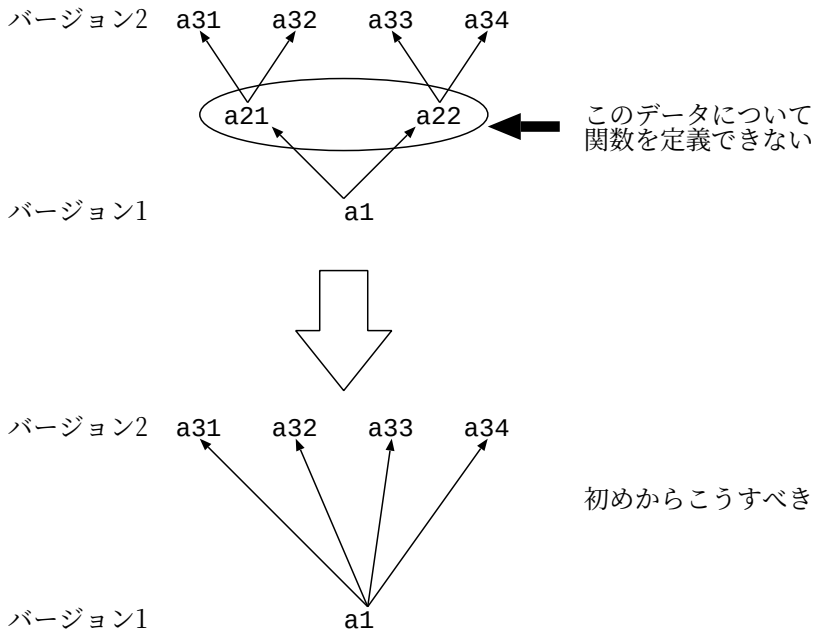


図 3.4: 単一バージョンでの連続した具体化

連続する具体化

具体値があらわす型には整数型や文字列型などがあるが、それらの型があらわす値の領域は互いに交わらない。また、データドメインはそれぞれいずれかの型があらわす領域に完全に含まれる。よって、一つのデータドメイン中に含まれる具体値の型は一種類とする。

バージョン n において、データを具体化するときは具体集合$\uparrow\mathbf{D}_{n-1}$に含まれる抽象値に対して、データの集合を与えることによって元の値が表す領域を分割する。関数はバージョン n での具体集合$\uparrow\mathbf{D}_n$に対して定義するので、バージョン n の具体集合に含まれる値から具体化すると、関数を定義できない値が発生する。このため、直前のバージョンの具体集合に含まれる値について具体化することで、関数を定義できないデータが発生する可能性を排除する。つまり一つのバージョンの中において、ある値を具体化して現れた値を更に具体化することはできない。(図 3.4)

3.6 関数の定義

3.6.1 AL における関数

AL における関数を以下のように定義する。

定義 12 *AL* におけるバージョン n の関数 F_n

$$F_n \equiv \langle \mathbf{D}_n^I, \mathbf{D}_n^O, \mathbf{R}_n \rangle$$

バージョン n までに詳細化した F の入出力データドメインを $\mathbf{D}_n^I$ 、$\mathbf{D}_n^O$ とし、$\mathbf{R}_n$ は以下のようなバージョン n までに詳細化した部分関数の集合とする。

$$\mathbf{R}_n \equiv \{f | \exists di \in \mathbf{D}_n^I, do \in \mathbf{D}_n^O, f : di \rightarrow do\}$$

関数の詳細化

関数の記述は抽象化段階と詳細化段階に分けられる。

抽象化段階において入力データと出力データのあらわす領域を決定し、それらの領域をそれぞれ一つの値に抽象化する。そして、抽象化した入出力データに対応する全関数を作成する。

関数はあるデータを入力したときに対応するデータを出力する。入出力データを段階的に具体化し、その具体化したデータ領域に対応する部分関数を詳細化することで構成する。

関数は入出力データドメインの具体値に対して定義するので、データを具体化したバージョンの中で同時に関数を定義する必要はない。関数を定義する対象となるデータがデータドメインの具体値である限り、任意のバージョンで定義することができる。

AL における関数は入出力データの組に対応する部分関数の集合であるが、それぞれの部分関数の入出力データの組は具体化規則に沿っていなければならない。図 3.5 では部分関数 `f4` の出力データが `f2` と矛盾している。この場合はデータ `B` ではなく `A` を出力データとしなくてはいけない。

定義 13 部分関数の性質

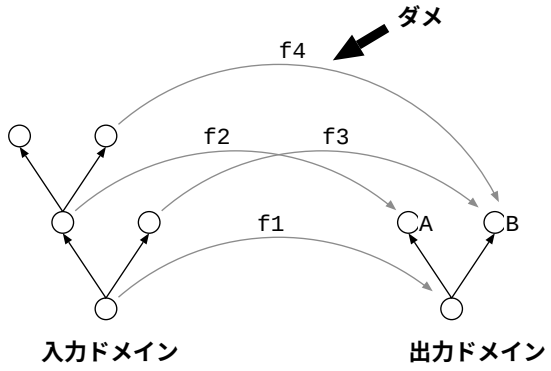


図 3.5: 矛盾した部分関数

- 入出力データドメインの具体集合に対して関数を詳細化する
  関数 $F_n \equiv \langle \mathbf{D}_n^I, \mathbf{D}_n^O, \mathbf{R}_n \rangle$ において、入出力データドメイン $\mathbf{D}_n^I$ 、$\mathbf{D}_n^O$ の具体集合 $\uparrow\mathbf{D}_n^I$ 、$\uparrow\mathbf{D}_n^O$ について部分関数を詳細化する。F を詳細化する部分関数 $f : di \rightarrow do$ は以下の条件を満たす。

$$\exists di \in \uparrow\mathbf{D}_n^I, \exists do \in \uparrow\mathbf{D}_n^O, \forall f' \in \mathbf{R}_{n-1}.(f' : di' \rightarrow do', di' \neq di \vee do' \neq do)$$

- 部分関数の入出力データの組は詳細化規則にそっている。
  関数 $F \equiv \langle \mathbf{D}^I, \mathbf{D}^O, \mathbf{R} \rangle$ において、部分関数の集合 $\mathbf{R}$ には以下の条件が成り立つ。

$$\forall f, g \in \mathbf{R}.(f : fin \rightarrow fout, g : gin \rightarrow gout, fin \preceq gin \Rightarrow fout \preceq gout)$$

3.6.2 AL による関数記述

関数の定義は ML とほぼ同様であるが、関数の入力はデータのリストであり、出力は一つのデータである。ここでいうデータとは 3.5 節で述べたデータ型を持つ値である。

複数のバージョンを通して同じ名前の関数を記述できる。ただし、それらの入出力データドメインは一致していなければならない。各バージョンについて、データドメインの具体集合の値に対応する関数を定義するが、データドメイン中のすべての値について関数を定義する必要は無い。

近似計算による抽象実行

関数がまだ定義されていない値について呼び出されたときは、関数が定義されているデータまで抽象化して呼び出す。関数の定義はまずデータドメインをひとつに抽象化した値についておこなうので、必ず関数は定義されている。よって、プログラムを止めずに実行を続けることができる。

図 ?? の場合、抽象値 `I12` を入力データとして関数 `f` を 呼び出すと、入力データに対して部分関数が定義されていないので、関数 `f` は抽象値 `I12` を抽象化した値 `I0` を入力データとして処理する。このときの関数の戻り値は抽象値 `O0` となる。

3.7 抽象解釈による実行

抽象化段階と詳細化段階を通して関数を定義することで、入出力データとして定義された領域内の任意の値について、一つ以上の関数が定義されることになり、これにより入出力データドメインに定義された任意の値について (値の詳細度はどうあれ) 抽象解釈による関数の実行が可能になる。

関数を呼び出すときは入出力データの抽象度にあわせて最適な部分関数を選択する。入力データに着目し、新しいバージョンから古いバージョンへ向かって、入力データが合致する部分関数を検索する。全ての部分関数について入力データが合致しなかった場合は、そのデータについて関数が詳細化されていないと考えられるので、データを一段抽象化して部分関数を再検索する。

入力データと出力データは独立に具体化することができるため、同一の入力データを持つが出力データの異る部分関数が存在する可能性がある。この場合はそれらの中から出力データが最も詳細化されている部分関数を呼び出す。

関数を定義する手順から、同一の入力データを持つ関数には以下のような関係がある。

n を部分関数 f が定義されているバージョン、m を部分関数 g が定義されているバージョンとする。

$$\forall f_n \exists g_m \in functions(F).((in(f_n) = in(g_m)) \Rightarrow$$
$$((out(f_n) \prec out(g_m)) \wedge (n < m)) \vee (out(g_m) \prec out(f_n)) \wedge (m < n))$$

つまり、入力データが同じ関数の場合、出力データがより具体化された関数はより新しいバージョンに属するので、新しいバージョンから検索すれば最も出力データが具体化され

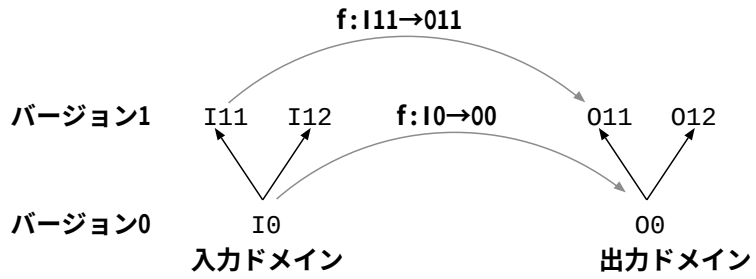


図 3.6: データドメインと対応する関数

ている関数を取り出すことができる。

3.7.1 AL による関数記述例

バージョン 0 において、データドメイン I0 と O0 を定義し、それぞれを関数 f の入出力データドメインとする。

バージョン 1 では I1 を 2 つの抽象値 I11 、I12 に具体化し、O1 を 2 つの抽象値 O11 、O12 に具体化する。ここで f の部分関数として、I11 から O11 への関数を定義する。

この例では I12 を入力とした部分関数が定義されていない。しかし次節で述べるように I12 を入力として関数 f を呼び出すことができる。

```
version 0;
domain I0;
domain I0;

fun f[a I0] = a O0;   (* f0 *)

version 1;
I1 reified[a I11, a I12];
O1 reified[a O11, a O12];

fun f[a I11] = a O11; (* f1 *)
```

3.8 AL によるプログラムの構築

この節では与えられた仕様について実際に AL でプログラムを構築する。

3.8.1 プログラムの仕様

あたえられた 2 つの整数を乗算した値の符号を判定するプログラム。乗算した値の符号によって、正の場合は 1 、負の場合は -1 、0 の場合は 0 を返す。

3.8.2 プログラムの構築

プログラムの構成は以下のように定義する。関数は mul のみで、入力と出力にそれぞれデータドメインを定義する。

バージョン 0

入力データドメインを 抽象値 `Input` 、出力データドメインを 抽象値 `Output` に抽象化し、`Input` から `Output` への関数を定義する。

$$
\begin{aligned}
D_0^I &= \langle \{Input\}, \emptyset \rangle \\
D_0^O &= \langle \{Output\}, \emptyset \rangle
\end{aligned}
$$

バージョン 0 における関数 mul は図?? のように定義する。

```
version 0;
domain Input;
domain Output;
fun mul [a Input] = a Output;
```

図 3.7: バージョン 0

バージョン 1

整数をあらわす抽象値として、`Int` を導入する。仕様から、入力データは 2 つの値をとるので `Int` の 2 項組のレコードに具体化する。まだ、これではなにも符号を決められないので、出力データは具体化しない。

$$
D_1^I = D_0^I[Input \prec (Int, Int)]
$$

バージョン 1 における関数 mul は図?? のように定義する。

```
version 1;
Input reified [r[a Int, a Int]];

fun mul [r[a Int, a Int]] = a Output;
```

図 3.8: バージョン 1

バージョン 2

抽象値 `Int` は整数があらわす値の領域をあらわしている。これを正、負、0 の 3 つの領域に分割する。正の領域を 抽象値 `Pos` 、負の領域を 抽象値 `Neg` 、0 を 具体値 `0` とする。出力データ `Output` は `-1` 、`0` 、`1` に具体化する。これで出力データは完全に具体化された。

$$
\begin{aligned}
D_2^I &= D_1^I[Int \prec \{Pos, 0, Neg\}] \\
D_2^O &= D_0^O[Output \prec \{-1, 0, 1\}]
\end{aligned}
$$

バージョン 2 における関数 mul は図?? のように定義する。関数の入力する 2 つの値

```
version 2;
Int reified [a Pos, a Neg, i 0];
Output reified [i(1),i(0),i(~1)];

fun mul [r[a Pos, a Pos]] = i 1
  | mul [r[a Pos, a Neg]] = i ~1
  | mul [r[a Pos, i 0]] = i 0
  | mul [r[a Neg, a Pos]] = i ~1
  | mul [r[a Neg, a Neg]] = i 1
  | mul [r[a Neg, i 0]] = i 0
  | mul [r[i 0, a Pos]] = i 0
  | mul [r[i 0, a Neg]] = i 0
  | mul [r[i 0, i 0]] = i 0;
```

図 3.9: バージョン 2

を具体化した。ここではすべての組合せについて一度に関数を詳細化したが、複数のバージョンに分けて詳細化してもよい。

バージョン 3

バージョン 2 で分割した正の領域をあらわす抽象値 Pos、負の領域をあらわす抽象値 Neg の領域をさらに具体化する。これらを条件型 condi を使って、Pos を 0 より大きい値に具体化、Neg を 0 より小さい値に具体化する。これで入出力データはすべて具体値となった。

最終的なデータドメインを図?? に示す。

$$D_3^I \;=\; D_2^I[Neg \prec (...-1), Pos \prec (1...), 0]$$

バージョン 3 における関数 mul は図?? のように定義する。

```
version 3;
Pos reified [condi(fn n=>(n > 0))];
Neg reified [condi(fn n=>(n < 0))];

fun mul [r[i a1, i a2]] =
    if(a1 > 0 andalso a2 > 0) then i 1
    else if(a1 > 0 andalso a2 < 0) then i(~1)
    else if(a1 < 0 andalso a2 > 0) then i(~1)
    else if(a1 < 0 andalso a2 < 0) then i 1
    else i 0; (* a1 = 0 orelse a2 = 0 *)
```

図 3.10: バージョン 3

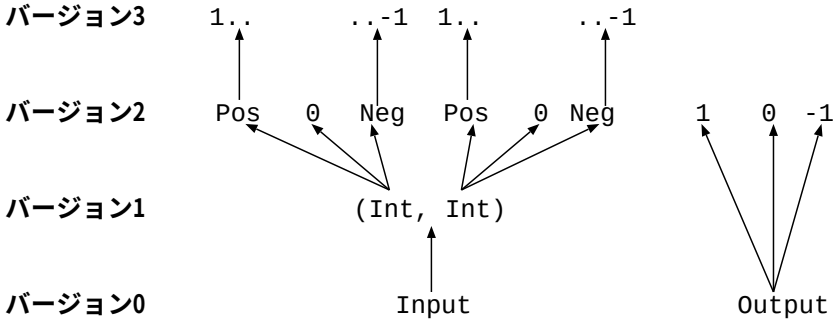


図 3.11: 最終的なデータドメイン

3.8.3 プログラムの実行

作成したプログラムの実行例を以下に示す。

```
- mul [r[i 3, i 0]];    (* 1 *)
val it = i 0 : Id
- mul  [r[i 3, a Neg]]; (* 2 *)
val it = a Neg : Id
- mul [r[i 0, i 3]];    (* 3 *)
val it = i 0 : Id
- mul [r[a Neg, i ~2]]; (* 4 *)
val it = a Pos : Id
- mul [r[i 3, i ~2]];   (* 5 *)
val it = i ~1 : Id
-
```

1、3、5 については整数の具体値の入力に対して具体値の解を求めており、正しく実行されているのがわかる。

2、4 では抽象度のことなる値を同時に入力している。AL では抽象度のことなる値について部分関数を定義できないので、抽象度を高い方に合わせて関数を呼び出す。ここではバージョン 2 の関数を呼び出している。

具体化した値に対応する部分関数を定義しない場合でも、近似計算によって計算を続けることによって解を得ることができる。ただし抽象度の高いバージョンの部分関数を呼び出すため、解もその部分関数と同じ抽象度になる。

例としてバージョン 3 において値の具体化だけを定義し、部分関数をまったく定義しない場合の実行結果を示す。

```
- mul [r[i 3, i ~2]];   (* 6 *)
val it = a ~1 : Id
- mul [r[i ~3, i ~2]];  (* 7 *)
val it = a 1 : Id
- mul [r[i ~3, i 0]];   (* 8 *)
val it = i 0 : Id
-
```

どの場合も入力データはバージョン 3 までに具体化した値となっているが、それらの値に対応する部分関数が存在しないため、近似計算によってバージョン 2 の関数が呼び出されている。正しく正負を判定していることがわかる。

第 4 章

開発環境 Alchemy

4.1 Alchemy の概要

Alchemy は AL プログラムの作成から実行までを支援する統合開発環境である。
この開発環境は以下のような機能を持つ。

- リビジョンの管理。
- データドメインの定義と具体化。
- プログラムの編集。
- 処理系とのユーザーインターフェース。
- 各種ツール。

図?? の斜線部が Alchemy 本体である。

4.1.1 Alchemy におけるプログラムの構築

AL ではプログラムの抽象化段階と詳細化段階をバージョンとして扱い、各バージョンは前バージョンから詳細化した部分のみを記述する。一般に各段階毎にデータドメインの定義、データの具体化、関数の詳細化のプロセスを繰り返す。

これに合わせて、Alchemy 上におけるプログラムの開発ではそれぞれのプロセスを段階毎に区切る。AL のバージョンの概念を拡張して、バージョンを分岐させることできる。

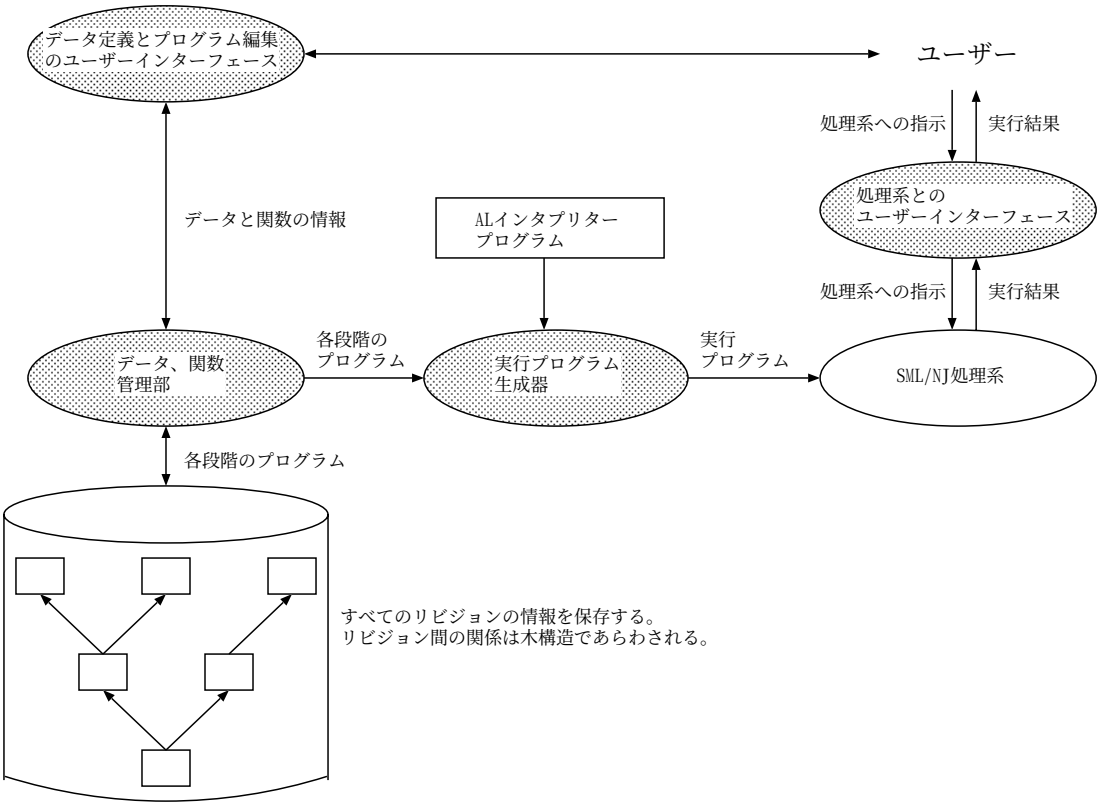


図 4.1: Alchemy の構成

このため Alchemy における各段階の管理をリビジョンと呼んで区別する。また、Revision Control System のように過去のリビジョンの記述を固定することはせず、任意のリビジョンに対してデータドメインと関数の定義を操作できる。これらの操作はあらかじめユーザーが指定したリビジョンのみを対象として行うこととする。前後のリビジョンにおける定義と矛盾しない場合に限り、データドメインや関数の変更や削除が可能である。ただしデータの削除については、他のデータがそのデータに依存していない場合に限り可能とする。

プログラムの構成は以下の操作によって行う。これらの操作はその時点で矛盾しない限り順番は問わない。

- 操作の対象とするリビジョンを指定する。
- データドメインを定義する。
- 前リビジョンのデータドメインの具体集合を具体化する。
- このバージョンにおけるデータドメインの具体集合に対して関数を定義する。

また、任意のリビジョンにおけるデータや関数の定義に対して以下のような操作ができる。

種別	操作
データ	ドメイン定義　データ具体化 データ名変更　データ削除
関数	関数定義　関数名変更 関数削除　入出力データドメイン変更
プログラムコード	編集

また、Alchemy の中ではプログラムを各リビジョン毎に分割して管理するので、そのまま実行することはできない。しかし、いつでも任意のリビジョンまでの実行プログラムを生成することができる。例えば 3 番目のリビジョンまでプログラムを構成してある場合でも 2 番目のリビジョン の実行プログラムを生成できる。この実行プログラムは AL インタプリターを結合したものであり、SML/NJ 上で実行可能なプログラムである。

こうして生成された実行可能プログラムは Alchemy の中から実行することができる。実際には Alchemy から SML/NJ を起動し、その上でプログラムを実行する。実行中に

AL インタプリターからユーザーへの問い合わせがある場合には Alchemy が代理となってユーザーへ問い合わせる。

他には以下のようなツールを提供する。

- データや関数の削除、変更を管理するツール。
- 具体化の方針を示すツール。
- 次に具体化すべきデータや詳細化すべき関数を提示するツール。

4.2 開発支援環境の実装

4.2.1 リビジョン管理

AL と Alchemy におけるバージョンの違い

AL ではバージョンの概念によって、プログラムを詳細化の各段階に対応する部分的なプログラムの集合として扱っているが、各バージョンはその前後のバージョンとだけ繋がっており、リスト構造になっていると考えることができる。。この場合、同時に記述できる詳細化の手順は一種類である。しかし詳細化の手順は一般には一種類だけではない。

Alchemy では、いくつかの手順を実際に作成して比較したり、以前のバージョンに戻ってデータドメインの構成をやりなおすなどの作業を支援するためにバージョンを分岐させる機能を提供する。バージョンの分岐は任意の段階から行うことができるので、各段階間の繋がりは木構造となる。Alchemy では木構造の関係を持つそれぞれの詳細化段階をリビジョンと呼ぶ。

各リビジョンは固有のリビジョン識別子を持つ。

リビジョン間の関連

各バージョンは木構造を構成するために以下のように、他のバージョンとの関連を持つ。

操作の対象となるリビジョンを指定すれば、そのリビジョンを構成する部分の集合は一意に決まる。あるリビジョンについてのすべての情報を得る場合にはリビジョン木中の指定のバージョンから根に向かってリビジョンをたどっていけばよい。

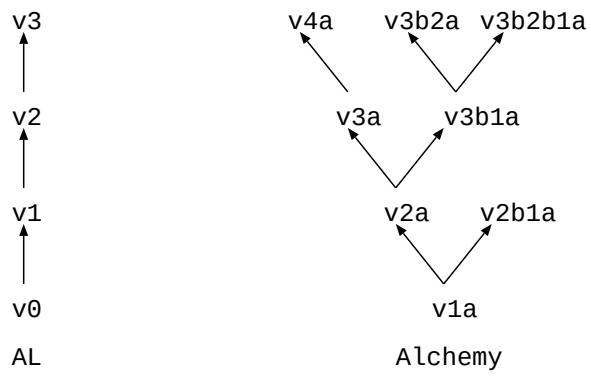


図 4.2: AL と Alchemy のバージョン

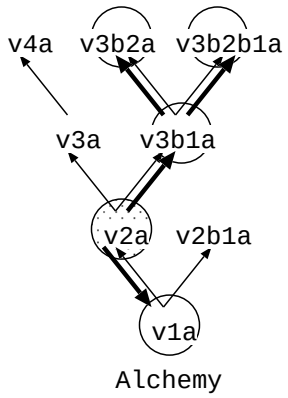


図 4.3: リビジョン v2a に関連するリビジョンの探索

例えば、図 ?? のように構成された Alchemy のリビジョン木において、バージョン v3b2a を構成する全てのバージョンの集合を得る場合は、v3b2a → v3b1a → v2a → v1a の順に木をたどる。

他のリビジョンにおける定義と矛盾する操作を避けるために、操作の対象となるリビジョンと関連するリビジョンが持つ定義を確認する必要がある。関連するリビジョンはそのリビジョンからリビジョン木の根へ向かう経路に存在するすべてのリビジョン。そのリビジョンから分岐しているすべてのリビジョンである。

リビジョン識別子の命名規則

リビジョン識別子 $revisionid$ は数字とアルファベットの列からなり、正規表現で以下のようにあらわすことができる。

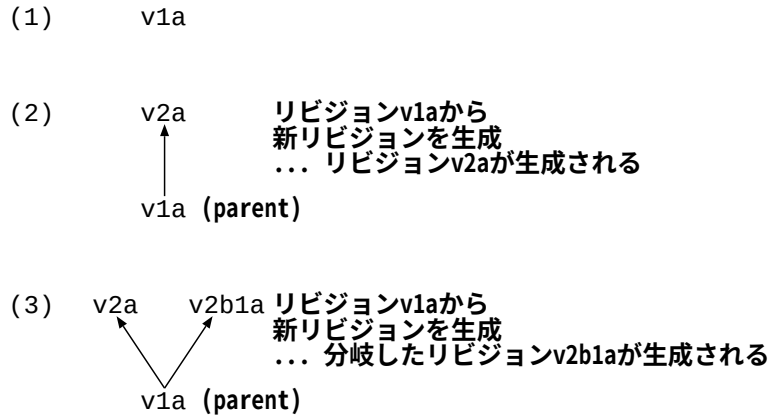
$$revisionId ::= [0-9][0-9]^*[a-z]([0-9][0-9]^*[a-z])^*$$

$versionid$ 以下の規則に従って命名する。

1. 最初のリビジョンは 1a である。
2. 詳細化したときはもとのリビジョン識別子に以下の操作を施して新リビジョンの識別子とする。
   リビジョン識別子の末尾の「数字+アルファベット」の組に対して、
   (a) 数字をインクリメントする。
   (b) リビジョンが分岐するときはアルファベットをアルファベット順の次の文字に置き換えて、末尾に 1a を追加する。

新しいリビジョンの生成

新たにリビジョンを生成する場合は、その基となるリビジョン $parent$ を指定する。$parent$ から生成されたリビジョンがすでに存在する場合は、新しいリビジョンは分岐したリビジョンとして生成される。



Alchemy では任意のリビジョンに対してデータの定義やプログラムの編集が可能である。各リビジョンでは以下の情報を記述する。

- 関数の入出力データドメイン定義。
- データドメインの具体化。
- プログラムコード。

これらは前のリビジョンから詳細化された部分の情報だけを記述する。そのため各リビジョンの情報は直前のリビジョンに依存している。

4.2.2 各リビジョンの情報管理

すでに述べたように Alchemy ではリビジョンの分岐が可能である。また、同時に複数のリビジョンを操作の対象とすることはできない。この操作の対象となっているリビジョンをカレントリビジョンと呼ぶ。

任意のリビジョンについてデータと関数に対する操作が可能であるが、カレントリビジョンに関連のあるリビジョンすべてにおいて一貫性が保たれるように、それらの定義を満たす場合にのみ操作を許可する。

ただしプログラムコードは変更しないので、ユーザー側での修正が必要である。

関数名と抽象値の名前

関数と抽象値はそれぞれ固有の名前を持つが、Alchemy では関連のあるリビジョンの集合の中で固有であればよいので、互いに関連の無いリビジョンでは同じ名前を使用することができる。

新規に関数や抽象値を定義する場合はそれらが既存の名前と一致しないようにしなければならない。

データドメインの定義

データドメインは関数の定義域または値域をあらわす値の集合で構成され、AL ではこの集合を定義すると同時にドメイン名と同じ名前の抽象値が定義される。そのリビジョン n において定義しようとする抽象値 a は以下の条件を満たす。

- リビジョン n におけるデータドメインの集合 $\mathbf{DS}_n$ にデータドメイン a を定義したときに、その名前を持つ抽象値は具体集合に含まれる。(定義 5 - 2 より)

$$\forall \mathbf{D}_n \in \mathbf{DS}_n.(\exists e \in \uparrow \mathbf{D}_n.e = a)$$

つまり、すでに同じ名前の抽象値があってもそれが具体集合に含まれる値であればならば問題無い。

データドメインの具体化

リビジョン n のデータドメインに含まれる抽象値 a を抽象値または具体値の集合 vs に具体化するとき、リビジョン n より詳細であるすべてのリビジョン m における a と vs の間には以下の条件が成り立つ。

- a は リビジョン m において定義されていないか、または具体化されていない。(定義 5 - 2 より)

$$\forall \mathbf{D} \in \mathbf{DS}_m.(\forall d \in data(\mathbf{D}).d \neq a) \vee (\forall d \in \mathbf{D}.d \not\prec a)$$

- vs の要素は a を要素として持つデータドメインに含まれていない。(定義 5-1 より)

$$\forall \mathbf{D} \in \mathbf{DS}_m.(\exists d \in data(\mathbf{D}).d = a) \Rightarrow (\forall d \in \mathbf{D}, \forall e \in vs.d \neq e)$$

データ名変更

抽象値にはそれぞれ固有の名前がつけられる。互いに関連のないリビジョン間に同じ名前の抽象値が存在する場合でも、それらは別の抽象値と見なす。

データ削除

カレントリビジョンに関連するすべてのリビジョン n において、データ d が以下の条件を満たす場合に限り d を削除することができる。

- リビジョン n より詳細化されたリビジョン m において具体化されていない。(定義 5-2 より)

$$\forall \mathbf{D} \in \mathbf{DS}_m.(\forall d \in \mathbf{D}.(d \not\prec a)$$

ただし $\mathbf{DS}_m$ はリビジョン n より詳細であるすべてのリビジョン m におけるデータドメインの集合。

- 関数の入出力データドメインとして使われていない。

$$\forall F \in \mathbf{AFS}_n.(\forall e \in dom(F).e \neq d) \wedge (e \in ran(F).e \neq d)$$

ただし $\mathbf{AFS}_m$ はリビジョン n より詳細であるすべてのリビジョン m における関数の集合。また、$dom(F)$ は関数 F の定義域、$ran(F)$ は関数 F の値域をあらわす。

- レコードやリストに含まれていない。

関数の定義

リビジョン n において定義しようとする関数を F とするとき、F は以下の条件を満たす。

- F の入出力データドメインは、リビジョン n において最も具体化された値である。

$$\exists \mathbf{D} \in \mathbf{DS}_m, \exists x, y \in \uparrow \mathbf{D}.(dom(f) = x) \vee ran(f) = y)$$

- F の関数名がリビジョン n における既存の関数名と一致しない。

関数名変更

リビジョン n において定義しようとする関数を F とするとき、F は以下の条件を満たす。

- F の関数名がリビジョン n における既存の関数名と一致しない。

関数削除

関数の削除に関して、とくに制限は無い。

プログラムコードの編集

ここでは関数の定義を記述する。単なるテキストエディタなので、自由に編集することができる。

Alchemy は、たとえデータドメインと関数の定義を変更したときでもプログラムコードは一切変更しない。そのためユーザーがそれらの変更に対応してプログラムコードを書き換える必要がある。

AL は `version`、`domain`、`reified` によるデータドメインの定義と具体化の記述を必要とするが、これらは Alchemy がデータの定義から自動生成するので、記述する必要はない。

プログラムコードの編集中は任意のリビジョンのプログラムを参照することができる。

4.2.3 実行プログラム

現在の AL 実行環境は SML/NJ 上に実装されており、AL プログラムと AL インタプリターを合成したものを SML/NJ 上で実行する。Alchemy はこの実行環境に沿った実行可能なプログラムを生成する。

AL インタプリター

AL の特徴である近似計算による抽象実行とデータドメインの抽象化とその具体化されたをするプログラム。ML 言語で記述されている。`version`、`domain`、`reified` を提供する。実行環境における実際の関数呼び出しは近似計算をするために AL インタプリターへ各バージョンの関数を登録し、インタプリターを通して呼び出す。

実行プログラムの構造と生成

Alchemy ではプログラムをリビジョン毎に分割して管理しており、これらの任意のリビジョンについて実行プログラムを生成することができる。

各リビジョンの記述は前のリビジョンから詳細化された部分についての記述のみによって構成するので、そのリビジョンが依存しているすべてのリビジョンのプログラムを統合して実行プログラムを生成する。 実行プログラムは AL インタプリターと AL プログラムから成り (図??)、SML/NJ 上で実行可能である。指定のリビジョンからリビジョン木の根へ向かって AL プログラムを構成する各バージョンの定義を集める。実行プログラムは以下の手順で生成する。

1. 各バージョンの記述について抽象度の高い順に以下の作業を繰り返す。

   (a) 定義に従って、AL のデータドメイン定義部を生成する。
   AL にあわせてバージョン識別子が 0 から始まる自然数につけかえる。

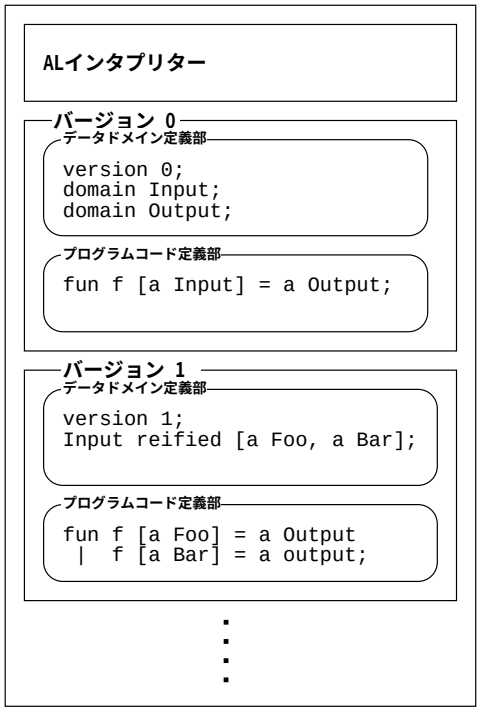


図 4.4: 実行プログラムの構造

(b) AL インタプリターを通して関数呼び出しを行うようにプログラムコード定義部を変更する。

2. AL インタプリターを合成する。

4.3 ツール

4.3.1 詳細化の方針

AL ではデータドメインを続けて詳細化する必要は無く、また、データに対する関数の詳細化も条件を満たす限り、任意の段階で行うことができる。そのため、意図せず極端に詳細化が遅れてしまうデータが発生することがある。このため Alchemy はデータや関数を優先度をつけて挙げる機能を提供する。優先度の付け方には，抽象度の高い順，低い順，あるいは最近頻繁に変更が加えられているもの順などがある。抽象度の高いものと

```
datatype AbstId = Bool | Input | Foo | Bar | Output;

fun printAbstId(Input) = print "Input"
  | printAbstId(Foo) = print "Foo"
  | printAbstId(Bar) = print "Bar"
  | printAbstId(Output) = print "Output";

use "Interpreter.sml";

(* UserProgram *)

initialize();
version 0;
domain Input;
makeDomains();
newFunc("f", [Input]);

fun f_0 [a Input] = a Output;

addFunc("f", f_0);

version 1;
Input reified [a Input, a Output];
makeDomains();

fun f_1 [a Foo] = a Output
  | f_1 [a Bar] = a Output;
addFunc("f", f_1);
```

図 4.5: Alchemy が生成した実行プログラムの例

は，ドメイン中でより古いバージョンで定義された値またはそれに関する関数を指す。また，最近頻繁に変更が加えられている関数の優先度は，次の計算式により決定する。

バージョン $n+1$ で関数 f が持つ優先度

$$priority(f) \equiv \sum_{i=0..n} def(f,i) * w(n-i)$$

ここで，$def(f,i)$ は関数 f がバージョン i で変更されていたら 1 となりそれ以外なら 0 となる関数である。また，w はバージョンに対する重み関数で，バージョンが離れる程小さな値を返す関数 (単調減少関数) となる。

4.3.2 テンプレート

これは，これから定義する関数をどのような形になるか示す機構である。ただし，上記のテンプレート中の再帰的なレコードはリストに特化してあり，関数名やデータ名，分岐の数などはそのプログラムでの定義に従っている。詳細化によって新しく現れた関数には仮の名前が付いており，プログラマは適宜テンプレートの中身を書き換えて，それをプログラム編集画面にコピーできるようになっている。

4.4 Alchemy の実行環境

以上の環境を Java 言語を使って実装した。実行には JDK 1.1 相当の Java VM が必要である。また AL インタプリターが ML 上で動作するために AL プログラムの実行には SML/NJ 0.93 処理系が必要である。Solaris 2.5 上で実行可能であることを確認した。

第 5 章

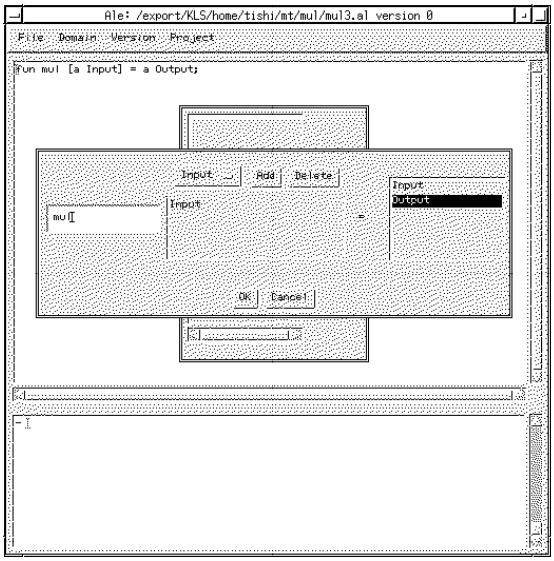
AL と Alchemy によるプログラムの構築

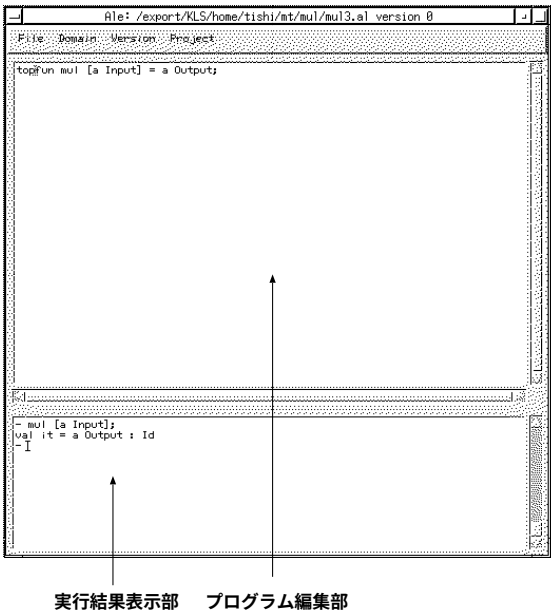
5.1 乗算の符号を求める問題

??節で作成したプログラムを Alchemy 上で作成する。AL のプログラムやデータドメインはすでに示してあるので、ここではプログラム作成中の画面を示す。

5.1.1 バージョン 0

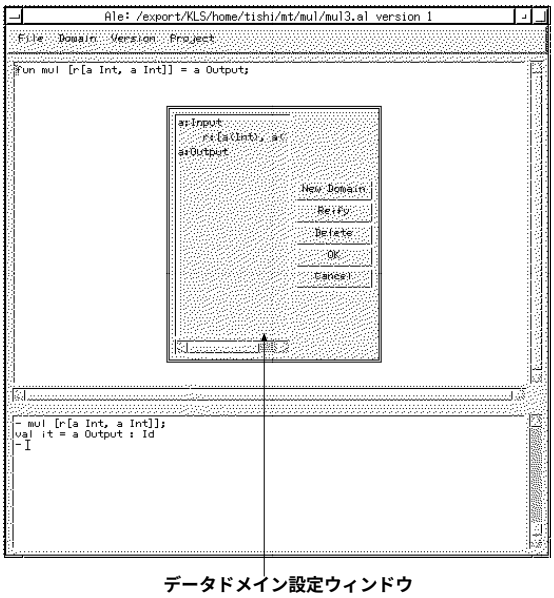
`Input]` から `Output]` への関数 mul を定義する。





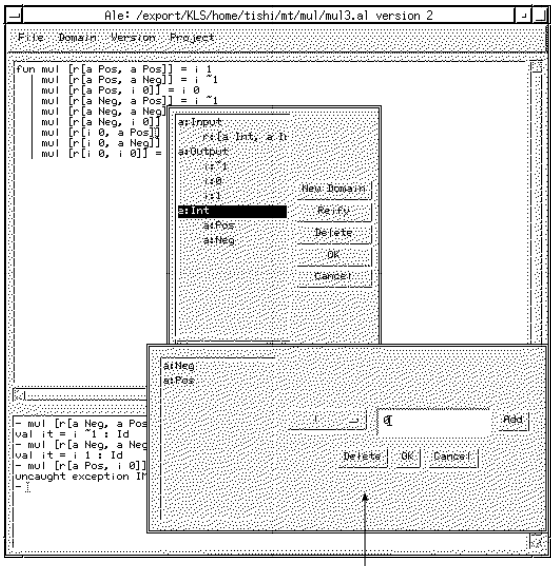
5.1.2 バージョン 1

データドメイン設定ウィンドウで現在のデータドメインを確認できる。また、実行結果表示部で、プログラムが実際に正しく実行されていることが確認できる。



5.1.3 バージョン 2

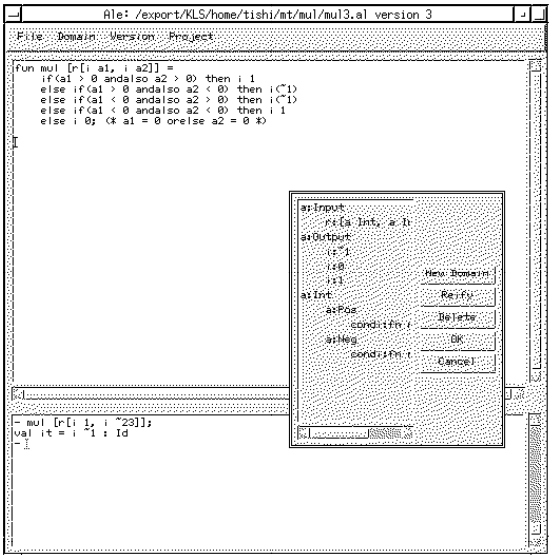
データ詳細化設定ウィンドウで Int を 0 へ詳細化している。すでに Pos と Neg へは詳細化されていることがわかる。



データ詳細化設定ウィンドウ

5.1.4 バージョン 3

ここまでの詳細化でデータドメインはすべて具体値となった。実行結果表示部でいくつかの実行結果が表示されている。ただしく動作していることがわかる。



5.2 FORMATTER 問題

ここでは FORMATTER 問題[1] を取り上げて、Alchemy を使ったプログラムの構築について述べる。

5.2.1 プログラムの仕様

非負整数 MAXPOS と、二つの区切り文字である空白と改行をもつ文字集合を与える。文字列 S において、単語を区切り文字か S の終端にはさまれた、非区切り文字の連続する空でない列と定義する。

このプログラムは文字の有限な列の入力を受理し、以下の条件を満たす文字の列を出力する。

1. 入力列が MAXPOS + 1 個の非区切り文字からなる列を含むならば、出力の並びは空白からなる。
2. 入力列が任意の MAXPOS + 1 個の文字に少なくとも一つの区切り文字を含むならば、
   (a) 入力の全ての単語は同じ順番で出力に現われる。出力は入力に現われない単語は持たない。
   (b) さらに出力は以下の制約を満たさなければならない:
      i. 出力は先頭か末尾に区切りを含まない。また二つ連続した区切りも持たない。
      ii. 任意の MAXPOS + 1 個の連続した文字は改行を持つ。
      iii. わずかに MAXPOS 個の連続した文字から構成され、出力の先頭または左の改行と出力の末尾または右の空白に挟まれた任意の出力部分列は改行を含まない。

5.2.2 プログラムの詳細化

詳細化の方針

まず、機能を以下のように 2 つの関数へ分割する。

[1] FORMATTER Problem (A.mili, "Specification Methodology: An Integrated Relational Approach")

```
version 0;
domain Min;
domain Mout;
domain Gin;
domain Gout;

fun main [a Min] = a Mout;
fun gettoken [a Gin] = a Gout;
```

図 5.1: FORMATTER バージョン 0

- 入力文字列を先頭の単語とそれ以降の文字列に分割する関数 $gettoken$ 。
- 入力文字列を一行あたり `MAXPOS + 1` 文字未満に整形して出力する関数 $main$ 。

これらの関数を組み合わせて仕様を満たすプログラムを作成する。

$$Prog \equiv \langle F, AFS, DS \rangle$$

バージョン 0

$$\begin{aligned} AFS &\equiv \{main, gettoken\} \\ DS &\equiv \{M^I, M^O, G^I, G^O\} \end{aligned}$$

$where$

$$\begin{aligned} main &\equiv \langle M^I, M^O, MF \rangle \\ gettoken &\equiv \langle G^I, G^O, GF \rangle \end{aligned}$$

バージョン `0` では抽象化段階として、関数 $main$ 、$gettoken$ の入出力データをひとつの値に抽象化する。また、それぞれのデータドメインと関数を以下のように定義する。

$$\begin{aligned} M_0^I &\equiv \langle \{Min\}, \emptyset \rangle \\ M_0^O &\equiv \langle \{Mout\}, \emptyset \rangle \\ G_0^I &\equiv \langle \{Gin\}, \emptyset \rangle \\ G_0^O &\equiv \langle \{Gout\}, \emptyset \rangle \end{aligned}$$

この仕様を `AL` によって記述したプログラムを 図?? に示す。

```
version 1;
domain Position;
domain CList;
Min reified [r[a CList, a CList, a Position, a Position]];
Gin reified [r[a CList, a CList]];
Gout reified [r[a CList, a CList]];

fun main [r[src, dest, max, cur]] = src;
fun gettoken [r[a CList, a CList]] = r[a CList, a CList];
```

図 5.2: FORMATTER バージョン 1

バージョン 1

バージョン 1 以降の詳細化段階ではデータドメイン M^I、M^O、G^I、G^O の具体化とそれにともなう関数 $main$ の部分関数の集合 MF と関数 $gettoken$ の部分関数の集合 GF への部分関数の追加をおこなうことで $Prog$ を詳細化する。

関数 $main$ は入力文字列を一行あたり MAXPOS + 1 文字未満に整形して出力する関数として定義するので、入力文字列と MAXPOS を与えなければならない。また、再帰的に処理を繰り返すので、作業中のデータとして、現在行の文字数と最終的に出力する整形済の文字列も同時に入力する。

関数 $gettoken$ は入力文字列を先頭の単語とその後に続く文字列に分割する関数である。ここでは入力文字列と切り出しの途中経過を記憶しておく必要がある。

上に挙げた条件を満たしてデータを入出力するように、それぞれのデータドメインを具体化する。複数のデータを扱うにはレコードを使う。

$$
\begin{aligned}
M_1^I &\equiv M_0^I[\{Min \prec (CList, CList, Position, Position)] \\
M_1^O &\equiv M_0^O[Mout \prec CList] \\
G_1^I &\equiv G_0^I[Gin \prec (CList, CList)] \\
G_1^O &\equiv G_0^O[Gout \prec (CList, CList)]
\end{aligned}
$$

$CList$ は、このプログラムにおいて文字列をあらわすデータドメインとする。

この仕様を AL によって記述したプログラムを 図?? に示す。

```
version 2;
domain Character;
CList reified [l[a Character]];

fun main [r[l [], l dest, maxpos, curpos]] = l dest
  | main [r[l(x::xs), l dest, maxpos, curpos]] =
    main [r[l xs, l(dest @ [x]), maxpos, curpos]];

fun gettoken [r[l token, l[]]] = r[l token, l[]]
  | gettoken [r[l token, l(c::cs)]] = r[l(token @ [c]), l cs];
```

図 5.3: FORMATTER バージョン 2

バージョン 2

$CList$ を文字 $Character$ のリストへ具体化する。$Character$ はすべての文字を含む抽象値である。これで文字のリストを処理することができる。

ここでは入力文字列から出力文字列へ文字列を移す処理を記述する。

$$
\begin{aligned}
M_2^I &\equiv M_1^I[CList \prec [Character]] \\
M_2^O &\equiv M_1^I[CList \prec [Character]] \\
G_2^I &\equiv G_0^I[CList \prec [Character]] \\
G_2^O &\equiv G_0^O[CList \prec [Character]]
\end{aligned}
$$

バージョン 3

仕様から文字には 2 つの区切り文字が含まれていることがわかる。このバージョンでは文字をあらわす抽象値 $Character$ を区切り文字の集合とそれ以外の文字集合に具体化する。実際には $Character$ を $Delimiter$ と $Alphabet$ に具体化する。これにともなってリスト $[Character]$ もそれぞれの要素に $Delimiter$ または $Alphabet$ を持つように具体化する。以降のバージョンでは入力文字列の先頭の文字に着目して関数を詳細化する。

また、行あたりの文字数を MAXPOS にあわせて制限するための数をあらわすデータドメインとして定義した $Position$ を整数の具体値 $Integers$ に具体化する。

ここまでの詳細化で関数 $gettoken$ で単語を取り出すことができるようになった。ここで取り出した単語には区切り文字を含まない。

```
version 3;
domain Alphabet;
domain Delimiter;
Character reified [a Alphabet, a Delimiter];
Position reified [Integers];

fun main [r[l [], l dest, i maxpos, i curpos]] = l dest
  | main [r[l(a Delimiter::xs), l dest, i maxpos, i curpos]] =
    main [r[l xs, l dest, i maxpos, i curpos]] (* eliminate *)
  | main [r[l src, l dest, i maxpos, i curpos]] =
    let
        val r[l head, l tail] = gettoken [r[l [], l src]]
        val tokensize = length(head)
    in
        if(tokensize > maxpos) then
            l []  (* fail, if length of the token larger than maxpos *)
        else (* newpos <= maxpos *)
            if(length(dest) = 0) then
               main [r[l tail, l(head), i maxpos, i 1]]
            else
               main [r[l tail, l(dest @ [a Delimiter] @ head), i maxpos, i 1]]
    end;

fun gettoken [r[l token, l[]]] = r[l token, l[]]
  | gettoken [r[l token, l(a Delimiter::cs)]] = r[l token, l cs]
  | gettoken [r[l token, l(a Alphabet::cs)]] =
    gettoken [r[l(token @ [a Alphabet]), l cs]];
```

図 5.4: FORMATTER バージョン 3

単語が MAXPOS + 1 文字より大きいときは空白を返す。関数 $main$ でこの検査をし、該当する単語を見つけたときは空白を返す。

連続した区切りを削除するために、関数 $main$ において入力文字列の先頭にある区切り文字を削除する。単語毎に区切り文字を付加する。

このプログラムでは文字列をリストとして処理するので、組込み関数 $length$ を使うことで文字数を数える。

$$
\begin{aligned}
M_3^I &\equiv M_2^I[Character \prec \{Delimiter, Alphabet\}, Position \prec Integers] \\
M_3^O &\equiv M_2^I[Character \prec \{Delimiter, Alphabet\}, Position \prec Integers] \\
G_3^I &\equiv G_2^I[Character \prec \{Delimiter, Alphabet\}] \\
G_3^O &\equiv G_2^O[Character \prec \{Delimiter, Alphabet\}]
\end{aligned}
$$

バージョン 4

区切り文字には「空白」と「改行」があるので、$Delimiter$ をそれぞれの具体値に具体化する。それ以外の文字をあらわす抽象値 $Alphabet$ を $Strings$ に具体化する。

これで文字列がすべて具体値に具体化され、実際の文字列を扱うことができるようなった。

関数 $main$ では処理中の行と追加する単語の間に空白文字を挿入する。また、処理中の行に次の単語を追加すると `MAXPOS + 1` 文字を超える場合には、処理中の行と次の単語の間に改行文字を挿入する。

以下の定義では空白文字を $Space$ 、改行文字を CR と表記する。

$$
\begin{aligned}
M_4^I &\equiv M_3^I[Delimter \prec \{Space, CR\}, Alphabets \prec Strings] \\
M_4^O &\equiv M_3^I[Delimter \prec \{Space, CR\}, Alphabets \prec Strings] \\
G_4^I &\equiv G_3^I[Delimter \prec \{Space, CR\}, Alphabets \prec Strings] \\
G_4^O &\equiv G_3^O[Delimter \prec \{Space, CR\}, Alphabets \prec Strings]
\end{aligned}
$$

また、フロントエンドとして `formatter` 関数を用意した。これは文字列と `MAXPOS` を入力とし、整形した文字列を返す関数となる。

最終的なデータドメインは図?? のようになった。

```
version 4;
Delimiter reified [s " ", s "\n"];
Alphabet reified [Strings];
(* Alphabet reified [condi(fn s c => (not((c = " ") orelse (c = "\n"))))]; *)
makeDomains();

(* vv uty vv *)
fun myexplode(ss) =
    let
fun memain(ss, []) = ss
  | memain(ss, x::xs) = memain(ss @ [s x], xs)
    in
l (memain([], explode(ss)))
    end

fun myimplode(l ss) =
    let
fun mimain(lss, []) = lss
  | mimain(lss, (s x)::xs) = mimain(lss @ [x], xs)
    in
implode(mimain([], ss))
    end

fun formatter(s instr, i maxpos) =
    s(myimplode(main [r[myexplode(instr), l [], i maxpos, i 1]]));
(* ^^ uty ^^ *)

fun main [r[l(s("\n")::ss), d, m, c]] =
    main [r[l ss, d, m, c]]      (* eliminate first delimiter *)
  | main [r[l(s(" ")::ss), d, m, c]] =
    main [r[l ss, d, m, c]]      (* eliminate first delimiter *)
  | main [r[l [], l dest, m, c]] = l dest   (* bottom *)
  | main [r[l src, l dest, i maxpos, i curpos]] =
    let
        val r[l head, l tail] = gettoken [r[l [], l src]]
        val token = myimplode(l head)
        val tokensize = (size(token))
        val newpos = curpos + tokensize
    in
        if(tokensize > maxpos) then
            main [r[l [], l [], i maxpos, i curpos]]
        else
            if(newpos >= maxpos) then
                main [r[l tail, l(dest @ [s "\n"] @ head),
                      i maxpos, i tokensize]]
            else (* newpos <= maxpos *)
                if(length(dest) = 0) then
                    (* continue but don't add first Delim *)
                    main [r[l tail, l head, i maxpos, i newpos]]
                else
                    main [r[l tail, l(dest @ [s " "] @ head),
                          i maxpos, i(newpos + 1)]]
    end;

fun gettoken [r[l token, l[]]] = r[l token, l[]]
  | gettoken [r[l token, l(s("\n")::cs)]] = r[l token, l cs]
  | gettoken [r[l token, l(s(" ")::cs)]] = r[l token, l cs]
  | gettoken [r[l token, l(c::cs)]] =
    gettoken [r[l(token @ [c]), l cs]];
```

図 5.5: FORMATTER バージョン 4

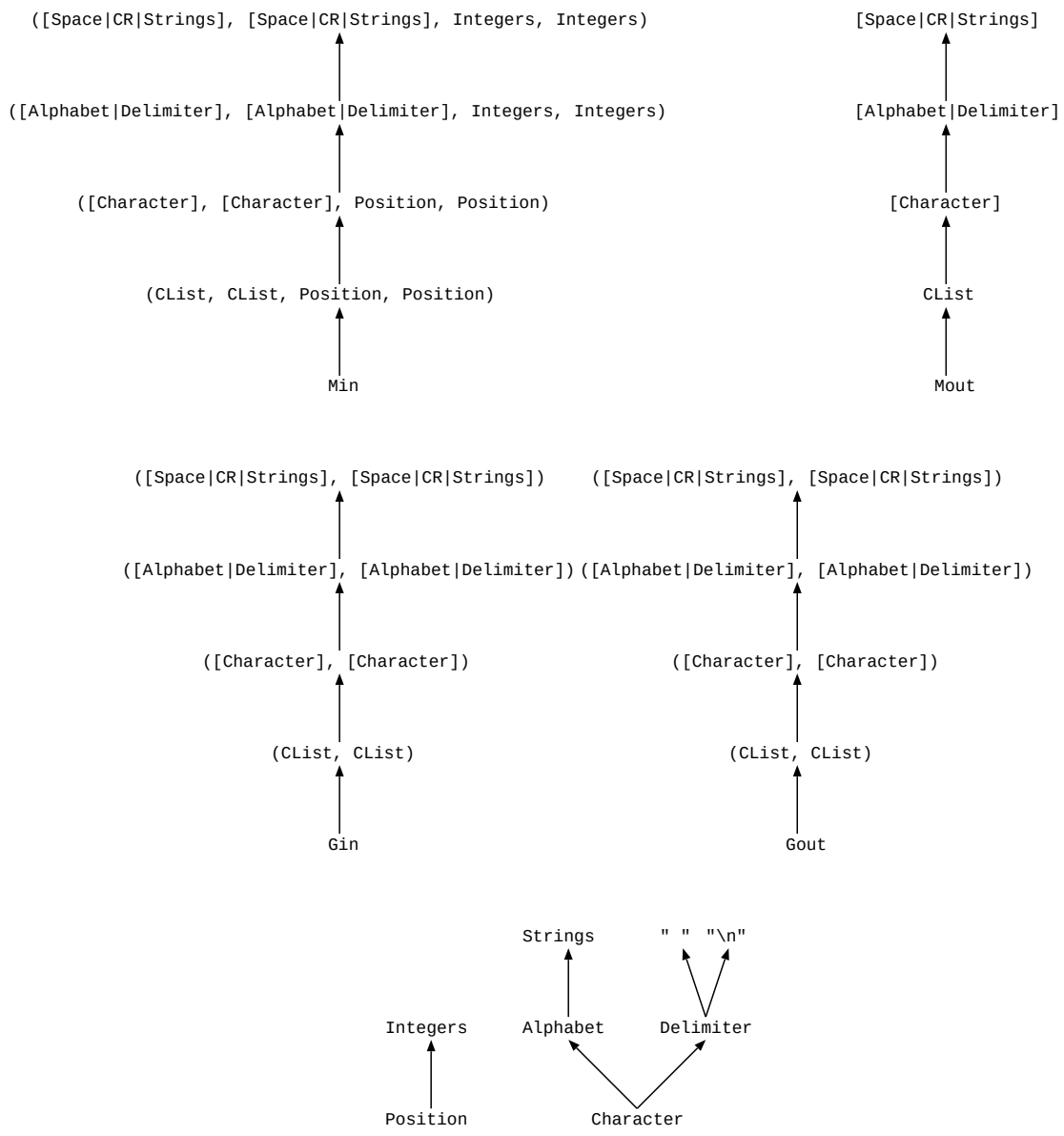


図 5.6: FORMATTER のデータドメイン

5.2.3 実行結果

ここではバージョン 3 とバージョン 4 についての実行結果を示し、これらの結果について考察する。

バージョン 3

関数 $main$ へ入力文字列データは、文字を一般の文字 Alphabet と区切り文字 Delimiter まで具体化した文字データのリストである。このバージョンまでに具体化したデータでは以下の動作を確認できる。

- 関数 $main$ による行の指定文字数をこえる長さを持つ単語の判定。
- 関数 $gettoken$ による文字リストからの単語の切り出し。

```
- main[r[l[a Alphabet, a Alphabet, a Alphabet, a Delimiter, a Alphabet], l [], i 2, i 1]];  (* 1 *)
val it = l [] : Id
- main[r[l[a Alphabet, a Alphabet, a Alphabet, a Delimiter, a Delimiter, a Alphabet], l [], i 3, i 1]];  (* 2 *)
val it = l [a Alphabet,a Alphabet,a Alphabet,a Delimiter,a Alphabet] : Id

-
```

例 1 では行の文字数 MAXPOS として 2 を与えている。ところが入力文字リストは Alphabet が 3 つ連続しているため、MAXPOS + 1 の非区切り文字からなる文字リストとなっている。プログラムはこの入力に対して空白を出力している。これは仕様 1 を満たしている。

例 2 では MAXPOS として 3 を与えている。入力文字リストは例 1 と同じものである。この入力文字リストではすべての非区切り文字の並びが MAXPOS + 1 文字未満であるため、正しい入力文字列であると判定し、そのまま出力している。ただし連続する区切り文字は除去する (仕様 2-b-i)。

また、例 1 と例 2 から関数 $gettoken$ によって単語が正しく切り出されていることがわかる。

バージョン 4

ここまでで文字のドメインはすべて具体値まで具体化されているので、完全に仕様を満たす関数を定義することができる。区切り文字を空白と改行文字 (

n）へ具体化することによって、MAXPOS で指定した行の長さに合わせて改行文字を挿入する。

```
- formatter(s "t1 t2   \n t3 \n t4 ", i 10);   (* 3 *)
val it = s "t1 t2 t3\nt4" : Id
- formatter(s "t1 t2   \n t3 \n t4 ", i 4);    (* 4 *)
val it = s "t1\nt2\nt3\nt4" : Id
-
```

例 3 では連続する区切り文字を除去する（仕様 2-b-i）。

文字列 "t4" の前に改行文字が挿入されているが、これは "t4" を含めると改行までの文字数が MAXPOS + 1 を超えてしまうためである。

例 4 も同様に仕様に合わせて適宜改行を挿入している。

考察

文字列中の文字を処理する場合は、文字列を文字のリストに分解して、それぞれの文字に対する関数を定義する。

文字をグループに分けてそれぞれのグループに対して固有の処理を記述することは、文字ドメインを具体化した値について部分関数を記述することにそのまま対応する。このような処理は容易に AL へ適用できるので、文字列のフィルタ処理のプログラムを作成しやすい。

プログラムへの入力を何らかのリストとしたときに、抽象化したリストの要素を段階的に具体化できるような場合に AL を使った段階的な構築は適用は有効であるといえる。

第 6 章

AL と Alchemy によるプログラム構築の評価

6.1 データドメインの定義

データドメインの定義と具体化については、それらを定義する時点で以下の点について検査する。

- 具体化関係自体が矛盾しないか。
- 前後のバージョンと矛盾しないか。

検査によって矛盾すると判定されたデータドメインの定義と具体化は無効とする。このためデータドメインの詳細化において問題が起きることは少ないと考えられる。

6.2 関数の記述

実際の関数を定義するプログラムコード自体はユーザーによる自由な記述を許しているので、入力中に自動的に検査することは無い。そのため、記述を終えた段階でツールを利用して記述に矛盾が無いか検査する。

AL で作成したプログラムは一般的にかなり冗長である。しかし、データの具体化に対応して詳細化した関数は前段階でも同じ領域のデータについて詳細化されているので、記述も似通ったものになることが多い。実際に、FORMATTER ではバージョン 3 とバージョ

ン 4 の間で顕著である。この傾向を利用して、前バージョンで記述したプログラムコードを修正することで容易に記述することができると考えられる。このため、実際ユーザーが記述するコードはそれほど多くないと考えられる。

Alchemy ではプログラムコード編集中に任意のバージョンのプログラムコードを参照する機能があるので、これを利用する。また、前段階のプログラムは完全に残されているため、新しい段階で記述した関数が期待通りの動作をしなかったときは、いつでもその部分だけを前のバージョンの関数を呼ぶように戻すことができる。

ところがデータドメインの具体化に条件型（condi）を使うとこれは部分関数は ML のパターンマッチング機構を使って別個に記述するが、条件型にはこの機構が使えないために、ひとつの関数中で condi で指定した条件判断を再度行う必要がある。パターンマッチによって部分関数に分けるという構造が崩れてしまう。これは、??節のバージョン 2 からバージョン 3 への詳細化で起きている。ただし、これは AL の問題であって ISDR 法の問題ではない。

パターンマッチング機構を利用して定義する場合の利点と欠点を以下にまとめる。

- 利点
  - 匿名変数を使うことができる。
  - 基本的に部分関数は 1 対 1 対応であるが、同じ処理の部分関数が複数つくられるような場合に n 対 m 対応の部分関数を容易に記述できる。
- 欠点
  - AL インタプリターはデータの抽象度にしたがって正しい抽象度の関数を呼び出すが、部分関数へのディスパッチを ML に依存するために条件型（condi）はパターンマッチで分けることができず、ひとつの関数のなかで条件分岐をさせる必要がある。

6.3 バージョンの扱い

Alchemy では AL におけるバージョンの扱いを拡張して、木構造のかたちにバージョンを分岐させることができ、これをリビジョンと呼んでいる。また、詳細化したリビジョンを作成しても以前のリビジョンが固定されることは無く、任意のリビジョンについて

データドメインやプログラムコードを操作することができる。この場合もデータドメインに関しては操作の対象となるバージョンに関連するリビジョンと矛盾しないように検査するので安全である。しかしプログラムコードについては操作の対象となっているリビジョンから詳細化したリビジョンへ影響が大きい。そのためプログラムコードの操作には細心の注意をはらわなければならない。

6.4 ISDR 法で実装することの利点

??節で実装した FORMATTER 問題はシンプルな仕様であるが、入力文字列のフィルタ処理の雛型となる部分はほぼ完成している。Alchemy のリビジョン管理機能を使ってリビジョンを分岐させ、そのリビジョンでは文字データを違うものに詳細化することで、さらに別のフィルタプログラムを実装することができる。また、文字データの詳細化によっては、規模の大きいプログラムとなることも考えられる。

第 7 章

まとめ

本論文ではソフトウェアの段階的詳細化法として提案されている ISDR 法の概要について述べた。ISDR 法ではいったん抽象化したデータに対して原始プログラムを定義し、データの段階的な具体化に対応したプログラムを詳細化することで構成する。そして具体化したデータについてプログラムを詳細化していない未完成なプログラムでも、近似計算によって以前の段階のデータに対応したプログラムを呼び出すことで、実行することが可能である。

そして ISDR 法によるプログラムの段階的構成を可能にするプログラム言語 AL と AL によるプログラムの構成について述べた。AL は ML に近い仕様を持った言語でバージョンの概念を持ち、プログラムを複数の段階に分けて構成することができる。また抽象値とその段階的な具体化にあわせて各段階で関数を定義する。ISDR 法と同じく具体化したデータについて関数の定義していない場合でも、近似計算によって以前の段階のデータに対応した関数を呼び出すことで、実行することが可能であることを例題を通して示した。

また、AL によるプログラミングを支援する環境 Alchemy についてのべた。Alchemy は木構造となるバージョンの分岐を可能にし、これによって管理される各詳細化の段階をリビジョンと呼ぶ。Revision Control System のように過去のリビジョンの記述を固定することはせず、任意のリビジョンに対してデータや関数の定義を操作することができる。これによって前後のリビジョンとの矛盾が起きないように Alchemy はユーザーの操作を制限し、安全にデータを具体化できるようにする。

最後に AL と Alchemy を使ったプログラムの作成を通して ISDR 法によるソフトウェアの段階的構築の有効性について考察した。

謝辞

最後に，本研究を行うにあたり終始御指導をしていただきました片山卓也教授には心から深く感謝申し上げます．　また，夜も昼もなく私の質問に答えて頂きました鈴木正人助手には深く御礼申し上げます．　夜遅くまで私の論文書きに付き合ってくださった片山研究室の皆様に深く御礼申し上げます．

参考文献

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

[2] 小野諭小川瑞史：抽象実行　そのフレームワークと実例，コンピュータソフトウェア，Vol. 2,4,6, No. 13 (1996).

[3] Milner, R. ed.: *The Definition of Standard ML*, The MIT Press (1990).

[4] 横内寛文（編）：プログラム意味論，共立出版（1994).

[5] 吉岡信和，鈴木正人，片山卓也：抽象解釈に基づく仕様の段階的具体化法，情報処理学会研究報告書，No. 25，pp. 137--144 情報処理学会（1995).

[6] 吉岡信和，鈴木正人，片山卓也：抽象解釈をもちいたプログラムの段階的具象化法，ソフトウェア工学の基礎 III，pp. 142--145 日本ソフトウェア科学会（1996).

[7] 吉岡信和，鈴木正人，片山卓也：抽象解釈にもとづく段階的プログラム構成法（ISDR 法）の記述能力の評価，コンピュータソフトウェア，Vol. 14，No. 2，pp. 85--89 (1997).