JAIST Repository
https://dspace.jaist.ac.jp/

Title	抽象解釈に基づくソフトウェアの段階的構成法とその評価
Author(s)	吉岡，信和
Citation	
Issue Date	1998-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/863
Rights	
Description	Supervisor:片山 卓也，情報科学研究科，博士

JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

博 士 論 文

抽象解釈に基づくソフトウェアの段階的構成法とその評価

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

吉岡 信和

1998 年 3 月

Copyright © 1998 by Nobukazu Yoshioka

要旨

大規模で複雑なソフトウェアの開発には段階的詳細化法が有効である。従来の段階的詳細化法は，主に設計段階に適用されていた。それらの方法では，途中段階のプログラムを実行することが考慮されておらず，各段階で発生した抽象的な誤りが発見しにくい，うまく詳細化できなくなるなどの問題があった。

本論文では，詳細化途中のデータを抽象値として表現することで，その上のプログラムが実行可能なソフトウェアの段階的詳細化法(ISDR 法) を提案した。具体的には，抽象値を段階的に具体化するのに従ってプログラムを段階的に詳細化する。詳細化したプログラムでまだ未定義の部分は，データの具体化関係を利用し，抽象度の高い以前の段階のプログラムを呼び出す。これによって，開発の途中段階の未完全なプログラムを実行可能にした。

また，プログラムの詳細化方針を示すためにその一般的なテンプレートを示した。このテンプレートに従う事で正しくプログラムを詳細化できる。また，詳細化の途中に混入した誤りの一部を自動的に発見するために，プログラム中に成り立つ表明式を言語の一部として記述する方法について述べた。

次に，ISDR の有効性を示すために，複雑な仕様を持つマイクロフィッシュ問題を解いた。そして，同じ問題を JSP 法で解き，この方法と比較，検討を行った。その結果，JSP 法ではこの問題は素直にプログラムを構成できないが，ISDR 法では問題なくプログラムが構成できることが分かった。

計算機環境上で様々な領域のプログラムを構築する事で，ISDR 法の有効範囲や実用性などの評価が可能となる。そこで，ISDR 法に基づく言語と開発環境を設計した。この言語は，関数型言語 ML にバージョンとドメインの定義，データの具体化を行うための構文を追加したものである。また，複数のバージョンに渡って定義された関数に対して，その型情報とドメインに基づき最新のバージョンの関数を呼び出すように意味を与えた。開発環境として，プログラムの詳細化を補助するためのツールや詳細化の誤りをチェックするためのツールの構成法について述べた。

目 次

1 はじめに **2**

1.1 背景 2

1.2 段階的詳細化 6

1.3 抽象解釈 8

1.3.1 プログラム解析のための抽象解釈 9

1.3.2 ISDR 法における抽象解釈 11

1.4 論文の構成 12

2 抽象解釈に基づくプログラムの段階的構成法（ISDR 法） **14**

2.1 準備 15

2.2 抽象化段階 18

2.2.1 データの抽象化 18

2.2.2 原始プログラムの作成 18

2.3 詳細化段階 18

2.3.1 ドメインの具体化 19

2.3.2 プログラムの詳細化 20

2.4 プログラムの詳細化方針 22

2.4.1 テンプレート 1: 入力を基本値の集合へ具体化した場合 23

2.4.2 テンプレート 2: 出力を基本値の集合へ具体化した場合 24

2.4.3 テンプレート 3: 入力をレコードへ具体化した場合 25

2.4.4 テンプレート 4: 出力をレコードへ具体化した場合 26

2.4.5 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 . . . 26

2.4.6 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 . . . 28

2.5 プログラムの抽象解釈 29
2.6 言語の拡張 — 不変表明式の導入 — 30

3 ISDR 法に基づく発展的開発環境 $\mathcal{A}lchEmy$ 32
3.1 抽象解釈に基づく関数型言語 AL 32
3.1.1 シンタックス 33
3.1.2 AL プログラムの解釈の手順 35
3.1.3 ドメインの構築 36
3.1.4 静的型推論 36
3.1.5 操作的意味 39
3.1.6 AL コンパイラの設計 40
3.1.7 AL プログラミング例 44
3.2 $\mathcal{A}lchEmy$の概要 48
3.2.1 開発支援ツール 49

4 ISDR 法による構成例 52
4.1 在庫状況の計算 52
4.2 マイクロフィッシュ問題 56
4.2.1 マイクロフィッシュ問題の概略 56
4.2.2 ISDR 法による解法 58

5 議論 67
5.1 ISDR 法の特徴 67
5.1.1 詳細化の定義 67
5.1.2 抽象実行 68
5.2 ISDR 法の正当性 68
5.3 ソフトウェアプロセスモデル 69
5.3.1 プロトタイピング指向パラダイム 69
5.3.2 探検的パラダイム 74
5.3.3 操作的パラダイム 76
5.3.4 ISDR 法のためのプロセスモデル 77
5.4 柔軟な開発プロセス 78

5.4.1 バックトラック 78
5.4.2 プログラム合成 80
5.5 他の方法論との比較 81
5.5.1 JSP 法との比較 81
5.5.2 複合/構造化設計法 85
5.5.3 実行可能仕様記述言語 85
5.5.4 その他の関連研究 87
5.6 応用分野 89
5.7 他のシステムへの応用 89

6 おわりに **92**
6.1 まとめ 92
6.2 今後の展望 93

謝辞 **94**

A 付録 **98**
A.1 用語と記号の説明 98
A.2 AL のシンタックス 98
A.3 例題 100
A.3.1 在庫状況の計算 100
A.3.2 イオン結合度を求める 101

本研究に関する発表論文 **106**

図 目 次

1.1 The waterfall model of the software life cycle 3
1.2 Spiral model of the software process. 5
1.3 principle of top-down decomposition of black boxes 7
1.4 解析のためのデータドメイン例 10
1.5 ISDR 法のデータドメイン例 12

2.1 ISDR 法の全体像 16
2.2 ドメイン例（自然数のドメイン） 17
2.3 テンプレート 1 23
2.4 テンプレート 2 24
2.5 テンプレート 3 25
2.6 テンプレート 4 26
2.7 テンプレート 5 26
2.8 テンプレート 6 28

3.1 AL プログラムの概要 33
3.2 ドメインの構築例 36
3.3 AL コンパイラの設計 41
3.4 具体化されたドメイン 46
3.5 $A_{L}chEmy$の概念図 48
3.6 抽象度の異なる値への具体化の禁止 50

4.1 入力ドメイン 55
4.2 出力ドメイン 55
4.3 マイクロフィッシュ問題の概略 57

4.4 最終的なドメイン 66

5.1 The prototyping-oriented software life cycle 72
5.2 Software development using the exploratory programming paradigm 75
5.3 Software development using the operational paradigm 77
5.4 Software development using ISDR 78
5.5 ドメインの合成例 81
5.6 JSP の入出力データ構造 83
5.7 入出力データ構造の不一致 84
5.8 中間データの導入 84
5.9 設計，実装プロセスと実行環境 91

A.1 金属データ集合の抽象化 102
A.2 非金属データ集合の抽象化 102
A.3 金属データの具体化 104
A.4 非金属データの具体化 105
A.5 イオン結合度の具体化 105

表 目 次

1.1 抽象値上のかけ算 9
1.2 抽象値上の足し算 10

3.1 AL で扱うことができるデータ型 35
3.2 AL の静的型推論規則 37
3.3 AL の操作的意味規則 39

5.1 ISDR 法と JSP 法の比較 85

第 1 章

はじめに

1.1 背景

近年，ソフトウェア産業の急激な成長によって，ソフトウェアの需要と供給のギャップは無視できないほど広がってきている。産業の成長により，新しいユーザ層が増え，その要求が多様化し，新しいアプリケーションの開発されるより早く，ユーザから次の新たな要求がきてしまうためである。このような状況を打破するために，新しいソフトウェア開発法が求められている。そこで，この節ではまず歴史を振りかえり従来の開発法の問題点を挙げる。

1960 年代には，既にソフトウェア危機が叫ばれていた。これは，もう経験則からだけでソフトウェアを造っていたのでは大きなものが作れないという認識から出たものだった。そこで，ソフトウェアシステムの開発の方法やツールを研究するソフトウェアエンジニアリングという分野が生まれた。この時代の成果は，構造化アルゴリズム (structuring algorithms) を生んだことである。これは，段階的詳細化 (stepwise refinement), 構造化プログラミング (structured programming) やプログラミング言語の支援する理論やツールをもたらした。

1970 年代になるとソフトウェアシステムの構造化が進んだ。様々な抽象マシンが提案され，モジュール，抽象データ型，汎化，継承などの考え方が生まれた。また，ソフトウェアのライフサイクルモデルと共にソフトウェア開発工程のモデルが提案された。その代表例は，ウォーターフォールモデルである。(図 1.1[1]) 開発工程としては，要求 (requirements) の記述，構造設計，実装，テストなどが考えだされた。そして，開発工程を補助するために，構造解析法や設計法，CASE ツールが開発された。ここで，ソフトウェアのシステム

の構造は，データフロー図や ER 図など構造化された設計を使って記述された。

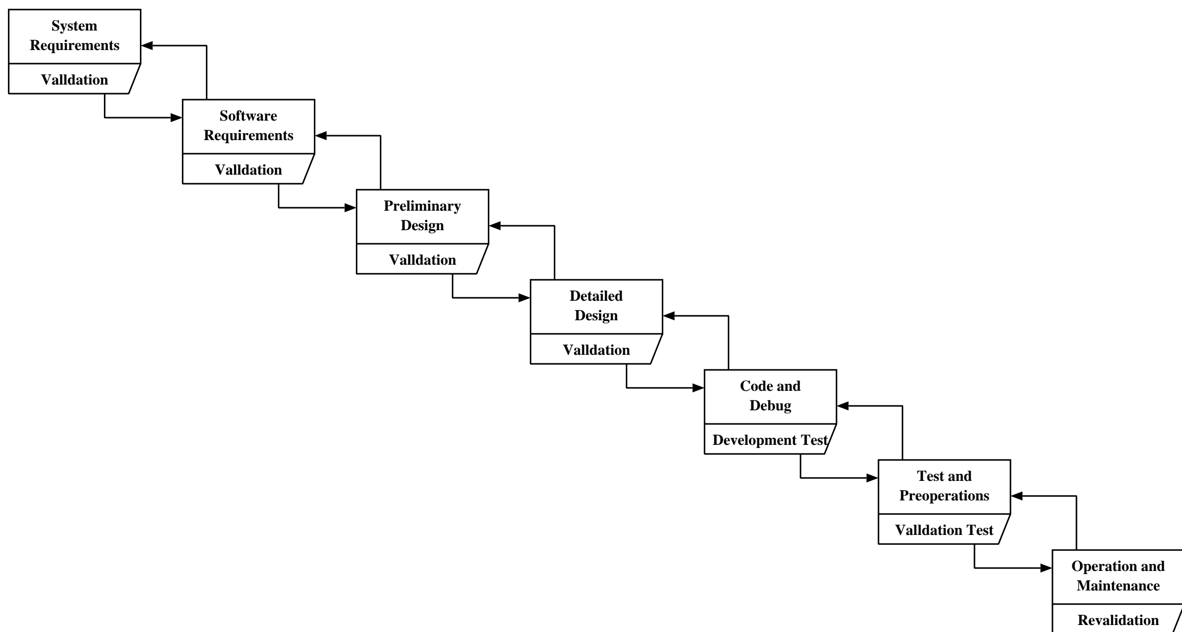

図 1.1: The waterfall model of the software life cycle

ソフトウェアシステムの大きさや複雑さが増すにつれ，開発工程の初めに顧客から正確で完全な要求を得ることはますます困難になってきた。顧客は，自分が必要なことを正確に分かっていたとしても，どうすればうまくいくかや将来の要求がどう変化するかなどは分からないからである。この年代に提案された逐次的な開発工程では，要求の誤りや問題はしばしばそれを実装して顧客でテストされるまで分からなかった。そして，それに対する要求の変更は，再実装のための高いコストを払わなければならなかった。文献 [2] は，逐次的アプローチの欠点として次の点を挙げている。

- 開発の工程が逐次的であり，フェーズ間の行き来がめったにないという仮定は現実とあっていない。
- 工程や段階をきっちり分けることは現実的でなく，実際は工程間がオーバーラップしてしまう。
- 完全な要求仕様や誤りのない設計をしてから次の工程に進めるケースはめったにない。
- 各工程 (特に要求定義) の成果物が正しいかどうかのテストは，実際に実験してみないと確認が困難である。また，要求の変更は実際は顧客からの要望でからくるが，顧

客がその変更が自分の意図を反映しているか確認するのは一番最後の工程を経てからであり，その変更のコストが高くつきすぎる。

1980 年代になり，これらの問題を解決するために初期の要求仕様の改良や逐次的なソフトウェアシステムの進化についての研究が行われた。その研究の成果として，プロトタイピングと探検的プログラミング (exploratory programming) の概念が開発された。これらのどちらかあるいは両方を適用することで，反復的な循環した開発プロセスを考えることができる。プロトタイプを使った代表的な開発プロセスとして，Boehm らによってスパイラルモデル [3] が提案された。(図 1.2)

スパイラルモデルはリスク指向のモデルである。各工程に入る前に，リスクを調査するためのプロトタイプを作成する。これにより，開発者と顧客との間のおおくの相互不理解はその場でとり除かれ，変更のためのコストを最小限に押えている。

プロトタイピングの方法にはそれを補助するためのツールが不可欠である。しかし，これらのツールはプロトタイプの開発時間を劇的に短くしてくれる。今日，プロトタイピング指向の要求定義を補助する様々な方法やツールが提案され，実際に使用されている。

ソフトウェアエンジニアたちとは別に，人工知能の研究者は探検的プログラミングのパラダイムを開発した。これは，人工知能の分野では，よく分かっていない複雑な知識表現でソフトウェアを構成する必要があり，それらに適したプロトタイプのツールを開発することが困難であったためである。探検的プログラミングもプロトタイピングの技法も与えられた問題に対するさまざまな解法を認識し評価するために用いられるが，その大きな違いは，プロトタイプの技法がほとんどの場合要求定義の工程で用いられるのに対して，探検的プログラミングは設計や実装の工程で用いられることである。また，この手法はプロトタイピングの方法と同様，それを補助するためのツールが不可欠となる。現在，非常に高級な言語とそのインタープリタやコード再利用のためのライブラリなどを備えた環境がいくつか作られている。しかし，これらは，素早い開発には対応しているが，システムのアーキテクチャ設計レベルの変更には対応していない。その上，進化的で探検的なプログラミング法を用いる場合，それらの変更は大変困難で，たいていの場合高くついてしまう。

そして，1990 年代になり次のような研究が進められている [2]。

- プロトタイプ指向のソフトウェア開発のための方法の改善とツールの開発
- 探検的 (exploratory) システム設計と実装のための方法の改善とツールの開発

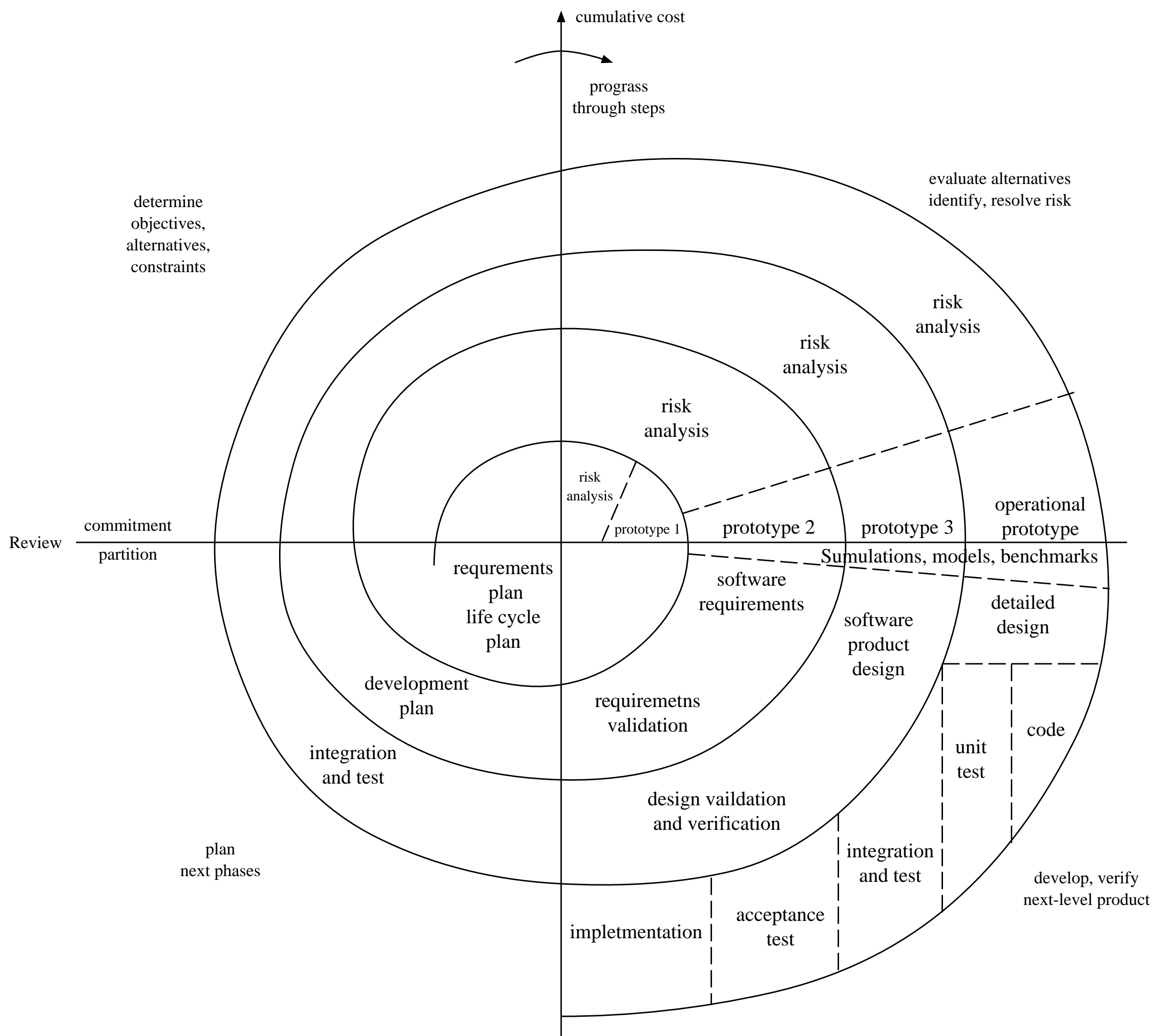

図 1.2: Spiral model of the software process.

- 要求の定義を補助するためのプロトタイプ指向の方法の改善やツールの開発と，システム設計や実装を補助する探検的プログラミングのための方法やツールの統合

これらの方法とは別に，1980 年代から従来のウォーターフォールモデルの前半にある要求仕様を得る工程を改良する方法として，操作的 (operational) な方法が開発された [4]。この方法は，形式的に完全で厳密な要求仕様を得るための方法で，実行可能な仕様記述言語やその仕様の反復的な進化などの方法などもいくつか提案された。現在，そのための環境や仕様から実装への自動化の研究が進められている。

そこで，我々は，要求から実装のまでの開発工程のどこからでもフィードバックできる開発法を探求する。プロトタイピングは初期の段階にある程度解法の方針を定めるために実行できるものを安いコストで作り上げ，探検的プログラミングは段階的に設計，実装を行う。そして，操作的な方法は厳密な要求仕様を作り出す。これらの利点を生かし，全ての工程を自由に行き来しながら段階的に生成物を作り，どの途中段階の生成物も数学的に正しいことがいえる開発を目指す。

1.2 段階的詳細化

ソフトウェアの開発工程の分割は，問題をより小さな制御しやすいサブ問題に分割するいう段階的構成法の考えに基づいている。しかしながら，ソフトウェアが大きくなり複雑化すると，その各開発工程がうまく制御できないほどの大きさになってしまう。そこで，工程をさらに詳細度に基づきトップダウンにより小さな問題を分割する段階的詳細化法が考え出された。

この方法の基本的な考え方は，システムを概略から徐々に説明するところにある。古典的な段階的詳細化による開発工程は次の様になる。まず，最初の段階 (phase) では入力と出力の細部を定義する。そして，次の段階ではそれらの入力から出力がどのように計算されるかを考えながらいくつかの固まりに分解する。それらの固まりは次の段階で明らかにし，これをすべての不明なところがなくなるまで繰り返す。(図 1.3[4])

この工程の初期の段階では，通常文書による定義になりそのためのツールが必要となる。要求定義の段階では，システムの外部の影響のみを正確に定義し，その中身の構造はブラックボックスとして隠しておく。設計段階では，一度だけのトップダウンな分解を行う。システムアーキテクチャはそのシステムを明解なコンポーネントにまで分解する。アーキテク

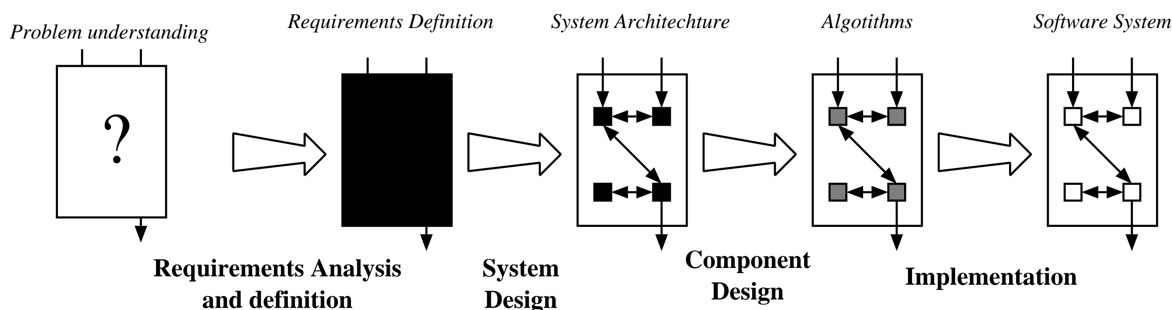

図 1.3: principle of top-down decomposition of black boxes

チャ設計では，システムコンポーネントのインターフェースの記述やそれらの間のやりとりの説明をする。全てのシステムコンポーネントのインターフェースが決定できたら，それらのコンポーネントについてアルゴリズムを設計する。そして，実装段階でそれらのアルゴリズムを他のコンポーネントと依存しないようにプログラム言語に変化し，それぞれのコンポーネント毎にテストを行う。最後に，コンポーネントを統合して，コンポーネント間のやりとりのテストを行う。このアプローチは開発プロセスとシステムそれ自身の両方の複雑さを解決する。

上記のような開発プロセスに代表されるように，従来の段階的詳細化法ソフトウェアを機能の観点からとらえ，その抽象化と詳細化 (分割) を行っていた。機能分割の方法の代表的なものとして，複合/構造化設計法 [5] に従ったプログラムのモジュール分解がある。この設計法は，プログラムを独立性が高いモジュールに分解するための方法である。具体的には，ある指標を用いてモジュール内の関連性を最大限に，かつモジュール間の関連性を最小限にするような分解を行う。これを段階的詳細化と組み合わせて使用する場合，その設計段階にのみ適用されていた。

また，形式的記述は目的のシステムに要求された振る舞いや構造を明解に記述することを可能にするが，大きく複雑なシステムを形式的に一度に記述しようとすると，それ自身大きく複雑になってしまい，仕様の構成や理解，保守が困難になってしまう。そこで，プログラムの詳細化を形式的に公理系で定義したものに Refinement Calculus[6] がある。これは，プログラムの前後に成り立つ条件を論理式で記述し，その事前条件を弱める，または事後条件を強めることに基づいて仕様を段階的にコード化する手法である。この方法では，詳細化途中のプログラムにコードは含まれているが，そのプログラム全体を実行する

事はできない。このように，従来の構成法の多くは開発途中のプログラムの実行を考慮していなかった。そのため，途中段階のプログラムの概念的な誤りの発見が困難となり，開発の最終段階になってから多くの不都合が発生する可能性があった。

そこで，本論文では，ソフトウェアの詳細化をデータの具体化という観点から行うことで，詳細化の途中段階のまだ一部のモジュールが定義されていない未完全なプログラムを解釈，実行を可能にするプログラムの段階的構成法（ISDR 法 [1]）を提案する。ISDR 法では，詳細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 [7] の技法に基づいたプログラムの解釈，実行を可能にする。これによって，各詳細段階ごとにプログラムのふるまいを確認しながら，誤りの修正や詳細化の方針を決定することが可能になる。具体的に，ISDR 法における抽象化とは，プログラムの入出力データ集合をそれぞれを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である。そして，詳細化とは，この抽象値をより具体的な値の集合に置き換え，それらの値を用いてより詳細なプログラムを構成することである。本論文ではこのように抽象値をより具体的な値にすることをデータの**具体化**と呼ぶ。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。

データ中心にソフトウェアを構成する利点は，静的構造は振る舞いを表す動的なものよりもシステムの変化にたいする変更が少ないことが挙げられる。段階手詳細化手法を用いる場合，初期の段階で構成した物を後の段階で変更しようとすると，その変更はシステム全体に及んでしまう。そのため，システムの多少の変更にも耐えられる構造を初期の段階で作成する必要があり，データ中心の構成はこれに適している。

1.3　抽象解釈

抽象解釈とは，抽象的なデータの集合とその上で定義された演算を使ってプログラムを解釈することをいう。一般的に，抽象解釈は，すでに完成したプログラムを抽象化し，それを実行するために利用される。プログラムを抽象化して実行する目的は，プログラム解析である。ISDR 法では，これとは逆に抽象的なプログラムから作り始め，段階的に完成させる過程で抽象解釈を利用する。

[1] Incremental Software development method based on Data Reification

1.3.1 プログラム解析のための抽象解釈

完成したプログラムを抽象的に実行することで部分的な情報を取り出すことができる。例えば，式 $21 \times 53 \times (-13)$ を計算した結果の符号は具体的な計算を行わなくても知ることができる。すなわち，式の中の具体的なデータをそれぞれ正か負に分け，正 $\times$ 正は正，正 $\times$ 負は負という規則を式に当てはめると，その符号は負であると判定できる。このことを形式的に書くと次のようになる。まず，式で使われている実数を正，負，ゼロを表す抽象値 Pos, Neg, $\dot{0}$へ写像し，かけ算を抽象値上で表 1.1のように再定義する。結局，式は，

$$(Pos \times Pos) \times Neg = Pos \times Neg = Neg$$

となり，結局式の符号は負であることが判定できる。

$\dot{\times}$	Pos	Neg	$\dot{0}$
Pos	Pos	Neg	$\dot{0}$
Neg	Neg	Pos	$\dot{0}$
$\dot{0}$	$\dot{0}$	$\dot{0}$	$\dot{0}$

表 1.1: 抽象値上のかけ算

ここで，これらの抽象値の集合を**抽象領域**とよび，抽象領域上の演算を使ってプログラムを実行することを**抽象解釈**という。また，抽象領域に対し，抽象化する前の元の値の集合を**具体領域**とよぶ。

プログラムを解析するために抽象解釈の技法を用いる場合，抽象解釈した結果の正当性の保証のために，以下の条件が成り立つようにプログラム中の任意の演算 $\circ$ を抽象化しなければならない。

$$abs(x \circ y) \quad = \quad abs(x) \mathbin{\dot{\circ}} abs(y) \tag{1.1}$$

ここで，abs は具体領域から抽象領域への関数であり，$\dot{\circ}$ は $\circ$ を抽象領域で再定義された演算である。

どの様な抽象領域を定義するかは，どの様なプログラム解析を行うかによって決まるほか，その上の演算をどのように再定義するかによってきまる。例えば，上の例と同じ抽象領域で足し算を再定義しようとしても，$Pos \dot{+} Neg$ などが条件 (1.1) を満たすように定義で

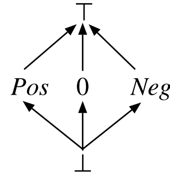

図 1.4: 解析のためのデータドメイン例

きない。すなわち，$Pos\dot{+}Neg$ の結果を $Pos, Neg, \dot{0}$ のいずれにしても正しくない計算をおこなっている可能性がある。そこで，抽象領域にあたらしく実数全体をあらわす抽象値 $\top$ を定義し，$\dot{+}$ の演算を表 1.2と定義することで正しい計算が行えるようになる。これに

$\dot{+}$	Pos	Neg	$\dot{0}$	$\top$
Pos	Pos	$\top$	Pos	$\top$
Neg	$\top$	Neg	Neg	$\top$
$\dot{0}$	Pos	Neg	$\dot{0}$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

表 1.2: 抽象値上の足し算

合わせ，抽象領域に半順序関係 $\subseteq$ を導入し，条件 (1.1) を次のように書き換えることで，より柔軟にプログラムを抽象化することができるようになる。

$$abs(x \circ y) \quad \subseteq \quad abs(x) \mathbin{\dot{\circ}} abs(y) \tag{1.2}$$

ここで，半順序関係 $\subseteq$ は抽象値を集合とみたときの包含関係に一致する。例えば，式 $35 + (34 * (-41))$ は，この計算結果 -1277 を抽象化すると Neg になり，抽象化した式 $Pos\dot{+}(Pos\dot{\times}Neg)$ を計算した結果は $\top$ なので，条件 (1.2) を満たしている。

本論文では，抽象値の集合とその間の半順序関係をあわせて**データドメイン**，あるいは単に**ドメイン**とよぶ。データドメインに空集合や矛盾をあらわす抽象値 $\bot$ を追加することで，関数のストリクト性解析や到達可能解析をおこなうことが可能になる（図 1.4)。

応用範囲は最適化を目的とするストリクトネス解析，インテリジェントなデバッグや仕様の部分的検証など。検証の例として，逆方向の抽象実行により，出力がある性質を満た

すためには入力にどのような性質が要求されるか解析する。

1.3.2 ISDR 法における抽象解釈

ISDR 法ではプログラムを段階的に完成させる過程で抽象解釈を利用する。この節では，ISDR 法のプログラム構成手順を概略した後，簡単な例をもとに抽象解釈の利用法を述べる。

ISDR 法は，抽象化したプログラムを作成する段階と，それを具体化する段階から成る。本論文では抽象化の段階で作成したプログラムを**原始プログラム**とよぶ。原始プログラムを繰り返し具体化することで，段階的にプログラムを構成する。その具体的な手順は，以下の様になる。

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 1 つの抽象値に抽象化する。
2. そ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抽象値を仕様の値を使って一段階具体化する。
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，デバックを行う。

例えば，足し算のプログラムを作りたい場合，まず入出力データ集合を実数を表す一つの抽象値 Num に抽象化し，その上のプログラムを以下のように作成する。

$$plus(Num, Num) \quad = \quad Num$$

次に，このプログラムを具象化する。ここでは，抽象値 Num を正，負，0 を表す 3 つの値 Pos, Neg, 0 に具体化したとする。すると，プログラムは次のように具象化できる。

$$\begin{aligned} plus(x, x) &= x \\ plus(x, 0) &= x \\ plus(0, x) &= x \end{aligned}$$

ただし，x は Pos,0,Neg のうちのいずれかである。

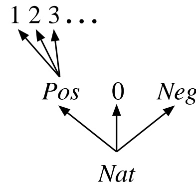

図 1.5: ISDR 法のデータドメイン例

ISDR 法では，抽象値とそれらの間の具体化関係 $\prec$ をもちいてデータドメインを構成することにより，各段階の未完成なプログラムの抽象解釈を可能にする。例えば，この例の場合のデータドメインは，$Num \prec Pos$, $Num \prec Neg$, $Num \prec 0$ という具体化関係を持つ。このように具体化関係をドメインの半順序関係にすると，そのボトムは前節で説明したデータドメインの$\top$と対応し，半順序関係は逆になる (図 1.5)。

また，前節で説明した条件 (1.2) に対応し関数の詳細化関係を数学的に定義している (定義 8)。この定義により詳細化した関数が元の関数と整合性が取れていることを保証している。

未完成のプログラムを実行するために，データドメインの具象化関係を用いてまだ未定義の部分を前の段階で定義したプログラムを呼び出すようにする。例えば上の例の場合，$plus(Pos, 3)$ の値は未定義だが，3 が Pos を具体化した値であることを利用し $plus(Pos, Pos)$ の値 Pos と一致すると解釈することで，実行が可能である。本論文ではこれを**近似計算**とよぶ。近似計算によりプログラムを希望どおり動作しているかどうか確認しながら作成することができるようになる。また，プログラム部品の構成順序に自由度がうまれ重要な部分から作成することが可能になる。

1.4 論文の構成

本論文では前節までに説明したようなソフトウェアの段階的構成法 (ISDR 法) を提案し，複雑な問題をこの方法で構成し他の方法論と比較検討を行う。また，ISDR 法に基づく開発環境 ALchEmy を設計する。

まず 2 章では ISDR 法を例を通して形式的に定義する。つぎに 3 章では，ALchEmy について概要を述べる。ここでは，言語とそのコンパイラを定義し，そのプログラミングを補

助するツールを含む開発環境の概要を述べる。

そして，4 章では，複雑な仕様をもつマイクロフィッシュ問題を ISDR 法で構成する方法を示し，つづく 5 章でこの例を通して JSP 法と比較する。また，同じ章で，ISDR 法の応用分野も検討する。

最後の 6 章では，本論文に関するまとめと今後の課題を述べる。

第 2 章

抽象解釈に基づくプログラムの段階的構成法（ISDR 法）

本論文では，ISDR 法を適用する問題領域として入出力データ間の関係が静的に決定できる領域を想定している。そして，プログラム言語として関数型言語を前提にしている。しかし，これは定義や意味の記述を簡単にするためと，静的に解析しやすいためだけの理由である。すなわち，関数型言語を使うと，詳細化やプログラムの解釈の定義が簡潔になり，詳細化が正しく行われているかのチェックが容易になる。また，プログラム合成などの機能も比較的容易に実現可能になる。しかし，この方法の基本的概念は，抽象的にシステムを記述，解釈し，段階的にそれを完成させることにあり，われわれはこの概念は様々な問題領域で適用可能であると考えている。

また，説明のためにプログラムを構成する前に仕様がすでに完成していると仮定している。しかしながら，ISDR 法は，仕様がまだ確定していない段階から開発を進め，段階的に仕様を明らかにしながらプログラムを構成する場合にも適用可能である。

その他の前提として，ソフトウェアが一連の詳細化で実装できることと，詳細化の前後で仕様の意味が変化しないことを仮定している。後者の仮定によって，この章で説明する詳細化と抽象解釈の完全性がいえる。また，この章の定義では，定義を簡単にするためプログラムの再利用性や効率について考慮しない。

以下は英語のよみを返すプログラムの仕様の例の一部である。これ以後，この仕様を ISDR 法の定義や構成法を説明するために用いる。

仕様例　自然数に対し英語のよみを返すプログラム

- 1 から 12は不規則なよみである。
- 13 から 19は前半のよみの後ろにteenがつく。
- 20 から 99までは二桁のよみと一桁のよみの連結である。
- 13の前半のよみはthirである。

このプログラムは例えば13 を入力とすると出力はthirteen となり，32 に対してはthirty-two となる。

ISDR 法では，この仕様から以下の手順でプログラムを段階的に構成する。(図 2.1)

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 1 つの抽象値に抽象化する。
2. そ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 メイン中の抽象値を仕様の値を使って一段階具体化する。
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，デバックを行う。

この手順の中で現れる抽象値は，仕様中の具体値の集合に対応する。例えば，上記の仕様例中の下線部が具体値の集合であり，ISDR 法では抽象値として定義される。

以下の節から，若干の定義の後，ISDR 法におけるプログラムの抽象化，詳細化そして途中段階のプログラムの解釈方法を形式的に定義する。

2.1 準備

ISDR 法では，具体値やそれを抽象化した値の集合で構成する領域を考え，この上でプログラムを構成する。この節では，この領域とその上のプログラムを定義する。

抽象化されたデータやその具体化のようすを表現するために，以下のようにデータド メイン (以下ド メインと略す) を定義する。

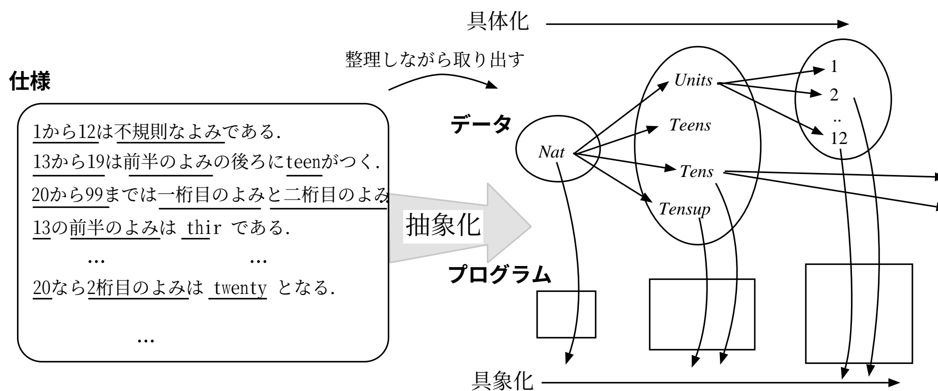

図 2.1: ISDR 法の全体像

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。

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

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。

例えば，図 2.2中の $\mathbf{D}$ の様に，自然数全体を表す抽象値 Nat，1 から 12 までの整数を表す抽象値 $Units$，13 から 19 までの整数を表す抽象値 $Teens$，20 から 99 までの整数を表す抽象値 $Tens$，それ以上の整数を表す抽象値 $Tenup$ を含むドメインは次のように表現する。

$$\mathbf{D} \equiv \langle \{Nat, Units, Teens, Tens, Tenup\}, \{Nat \prec Units, Nat \prec Teens, Nat \prec Tens, Nat \prec Tenup\} \rangle$$

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。

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

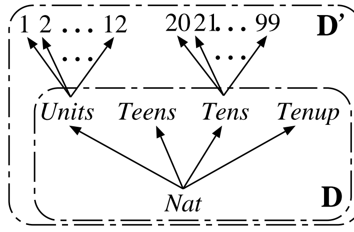

図 2.2: ドメイン例（自然数のドメイン）

ドメイン **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。 ∎

この記法を用いて，図 2.2中の **D** の要素 $Units$ を 1 から 12 までの整数に，$Tens$ を 20 から 99 までの整数に具体化してできたドメイン $\mathbf{D}'$は以下のように表現できる。

$$\begin{aligned}\mathbf{D}' &= \mathbf{D}[Units \prec \{n|1 \leq n \wedge n \leq 12\}, Tens \prec \{n|20 \leq n \wedge n \leq 99\}] \\ &= \langle\{Nat, Units, Teens, Tens, Tenup, 1, \ldots, 12, 20, \ldots, 99\}, \\ &\quad \{Nat \prec Units, Nat \prec Teens, Nat \prec Tens, Nat \prec Tenup, \\ &\quad Units \prec 1, \ldots, Units \prec 12, Tens \prec 20, \ldots, Tens \prec 99\}\rangle\end{aligned}$$

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 **AFS**，ドメインの集合 **DS** の 3 つ組で構成される。

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

ここで，F は与えられた仕様を実現する関数であり，**AFS** はその関数で使われている関数の集合となる。また，**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。

2.2 抽象化段階

2.2.1 データの抽象化

ISDR 法では，最初にもっとも単純なプログラムを構成する。そのために，まず仕様中のすべてのデータ集合をそれぞれ 1 つの抽象値まで抽象化する。ISDR 法における抽象化の意義は，外部とのインターフェースである入出力の数を認識することである。

例えば，英語のよみの仕様例に現れるデータは自然数とその読みなのでそれぞれの集合を Nat と Prn という抽象値に抽象化する。この時，入力ドメイン $\mathbf{D}_0^{\mathrm{I}}$と出力ドメイン $\mathbf{D}_0^{\mathrm{O}}$は以下のようになる。

$$\begin{aligned}\mathbf{D}_0^{\mathrm{I}} &= \langle\{Nat\},\emptyset\rangle \\ \mathbf{D}_0^{\mathrm{O}} &= \langle\{Prn\},\emptyset\rangle\end{aligned}$$

2.2.2 原始プログラムの作成

原始プログラムは，問題があたえられると機械的に定義することができる。すなわち，原始プログラムは入力データ全体を表す抽象値から出力データ全体を表す抽象値への関数である。

例えば，前節で抽象化した値を用いて原始プログラムの主関数 F_0を以下のように定義できる。[1]

$$\textbf{fun}\ F_0(Nat) = Prn;$$

プログラムをもっとも単純なものから作り始める理由は，バグが入る余地のないところから始めるためと，すべての入力に対し何らかの出力を決定できることを保証するためである。つまり，プログラムの中でまだ定義されていない部分があっても，少なくとも原始プログラムを呼び出すことで，プログラム全体の実行が可能となる。

2.3 詳細化段階

目的のプログラムは，前節の手順で作成した原始プログラムを繰り返し詳細化することで構成する。

[1] ここで例示するプログラムは Standard ML の表記法を用いる。

2.3.1 ドメインの具体化

ドメインは，基本値の集合または構造を持った値によって具体化される。前者は，ドメイン中の抽象値をいくつかの別々のより抽象度の低いデータへ具体化する場合である。すなわち，この具体化は元の抽象値が表している集合をお互いに疎な集合に分割し，それぞれの集合を表す値を新しく導入することである。ここで，これらの集合は，最終的に構成される完全なプログラムが扱う具体値で構成されている。ドメインの具体化はその中の抽象値のみに対して行われ，その抽象値をすでにドメインに存在する値に具体化することはない。それゆえ，ドメインは必ず具体化関係に関して木構造になる。よって，正しく具体化されたドメインは以下の条件が成り立つ。

定義 4 ドメインに成り立つ条件

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。

$$\begin{aligned}&\forall x,y \in data(\mathbf{D}), \exists z \in data(\mathbf{D}).\\&\qquad (z,x) \in rel^{*}(\mathbf{D}) \wedge (z,y) \in rel^{*}(\mathbf{D})\end{aligned}$$

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

ドメイン中のどの抽象値を具体化するかは，以下のような 3 つの要因により決定する。

仕様 仕様の中の最も大きな集合を表す抽象値から順に具体化する。

利用者の要求 重要な部分，はやく動かしてみたい部分を先につくる。また，例外処理や利用頻度が高くない部分は後まわしにする。

アルゴリズム プログラムで用いるアルゴリズムによって優先順位を決定する。例えば，英語のよみの仕様例に対して，二桁の処理を一桁の処理を利用するようなプログラムを定義する場合，一桁の値を二桁の値より先に具体化する。

この中で，アルゴリズムによる制約は利用者の要求に対して障害になる場合がある。ISDR法では，2.5 節で示す近似計算のメカニズムによってアルゴリズムの制約を受けずにプログラムを構成することが可能である。

また，抽象値をどの程度具体化すべきか，またどのような値に分割すべきかは仕様によって異なる。例えば，英語のよみの仕様の場合，自然数を表す Nat は英語のよみの規則性によって 1 から 12 を表す $Units$, 13 から 19 を表す $Teens$,20 から 99 を表す $Tens$, それ以上の数 $Tenup$ に分割できる。

ドメインの具体化を形式的に定義するとつぎのようになる。

定義 5 ドメインの具体化

ドメイン $\mathbf{D}_j$がドメイン $\mathbf{D}_i$を具体化しているとは，これらの間につぎの条件が成り立つ時である。

$$data(\mathbf{D}_i) \subset data(\mathbf{D}_j) \wedge rel(\mathbf{D}_i) \subset rel(\mathbf{D}_j)$$

このとき，$\mathbf{D}_i \sqsubset \mathbf{D}_j$と書く。 ∎

例えば，図 2.2中の $\mathbf{D}$ と $\mathbf{D}'$との間には詳細化関係 $\mathbf{D} \sqsubset \mathbf{D}'$が成り立っている。

2.3.2 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。

例えば，図 2.2中の $\mathbf{D}$ の具体集合$\uparrow \mathbf{D}$ はつぎのようになる。

$$\uparrow \mathbf{D} \;=\; \{Units, Teens, Tens, Tenup\}$$

これらに対し，F_1をつぎのように，$Units$, $Teens$, $Tens$ に対してのみ定義できる。

$$\begin{aligned}&\textbf{fun}\ F_1\ (Units)\ = \text{"}unit\text{"}\\ &\quad |\ F_1\ (Teens) = \text{"}xteen\text{"}\\ &\quad |\ F_1\ (Tens)\ \ = \text{"}xten\text{"};\end{aligned}$$

そして，図 2.2のように $\mathbf{D}$ を $\mathbf{D}'$に具体化した場合，$\mathbf{D}'$の具体集合はつぎのようになる。

$$\uparrow \mathbf{D}' \;=\; \{1, \ldots, 12, Teens, 20, \ldots, 99, Tenup\}$$

この具体集合に対し，次のようにアルゴリズムの制約を受けずに 1 から 12 までの値に対する関数を作る前に 20 から 99 までの値に対しての関数を定義することが可能である。

$$\begin{aligned}&\textbf{val}\ nty = [\text{"}twenty\text{"}, \text{"}thirty\text{"}, \text{"}forty\text{"}, \text{"}fifty\text{"}, \text{"}sixty\text{"}, \text{"}seventy\text{"}, \text{"}eighty\text{"}, \text{"}ninety\text{"}];\\ &\textbf{fun}\ F_2(n) = \textbf{if}\ 20 \leq n \wedge n \leq 99\ \textbf{then}\\ &\qquad\qquad combine2(n\ div\ 10, n\ mod\ 10);\\ &\textbf{and}\ combine2(x, 0)\ =\ nth(nty, x-2)\\ &\quad |\ combine2(x, y)\ =\ nth(nty, x-2)\frown\text{"-"}\frown F(y);\end{aligned}$$

そして，このバージョンで定義されなかった 1 から 12 までの部分は前のバージョンの関数を使って近似計算をおこなう。

プログラム Prog は以下の定義をみたすように構成する。

定義 7 プログラムの構成

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

$$\forall f \in \{F\} \cup \mathbf{AFS}, \exists \mathbf{D}, \mathbf{D}' \in \mathbf{DS}. \qquad f : \uparrow \mathbf{D} \rightarrow \uparrow \mathbf{D}'$$

▮

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$\uparrow \mathbf{D}1\times \uparrow \mathbf{D}2 \times \ldots \times \uparrow \mathbf{D}m \rightarrow \uparrow \mathbf{D}$, f'の型が$\uparrow \mathbf{D}1'\times \uparrow \mathbf{D}2' \times \ldots \times \uparrow \mathbf{D}m' \rightarrow \uparrow \mathbf{D}'$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。 ∎

段階的詳細化法では各段階ごとに詳細化したプログラムが正しいかどうかを調べる必要がある。ISDR 法では，各段階で（1）ドメインが正しく具体化されているか,（2）プログラムが正しく詳細化されているか,（3）詳細化した関数が仕様を満たしているかの 3 点について調べる。

2.4 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[2]。

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。

[2] これだけで全てのデータ構造を表現可能である。

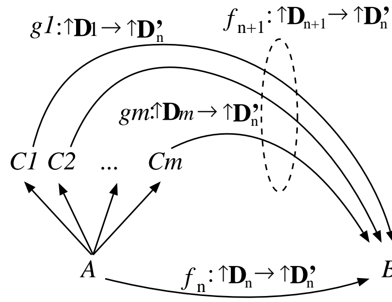

図 2.3: テンプレート 1

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

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。

2.4.1 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 ($C1$, $C2$, ..., 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 2.3 [3])

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

[3] 図 2.3から図 2.6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 2.4: テンプレート 2

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

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。

2.4.2 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 2.4)

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

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

$$
\begin{aligned}
\textbf{and } & g1(A) = C1 \\
& \vdots \\
\textbf{and } & gm(A) = Cm
\end{aligned}
$$

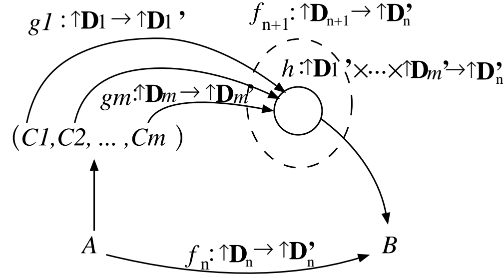

図 2.5: テンプレート 3

$$
\begin{aligned}
&\textbf{and}\ cond1(A) = \text{bool} \\
&\qquad\vdots \\
&\textbf{and}\ condm(A) = \text{bool};
\end{aligned}
$$

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。

2.4.3 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 ($C1$, $C2$, ..., 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)

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

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

ここで $E1$, ..., 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。

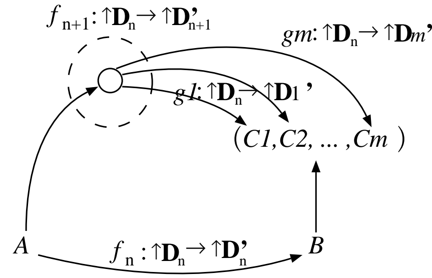

図 2.6: テンプレート 4

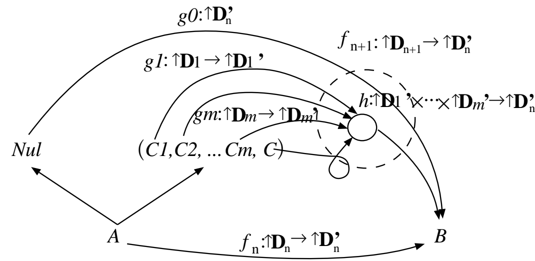

図 2.7: テンプレート 5

2.4.4 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 2.6)

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

$$\textbf{fun } f_{n+1}(A) = (g1(A), \ldots, gm(A))$$

$$\textbf{and } g1(A) = C1$$

$$\vdots$$

$$\textbf{and } gm(A) = Cm;$$

2.4.5 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 2.5)

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

$$
\begin{aligned}
&\textbf{fun } f_{n+1}(Nul) = g0() \\
&\quad \mid f_{n+1}((C1,\ldots,Cm,C)) \;=\; h(g1(C1),\ldots,gm(Cm),f(C)) \\
&\textbf{and } g0() = B \\
&\textbf{and } g1(C1) \quad = E1 \\
&\qquad\qquad\quad \vdots \\
&\textbf{and } gm(Cm) = Em \\
&\textbf{and } h(E1,\ldots,Em,B) = B;
\end{aligned}
$$

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

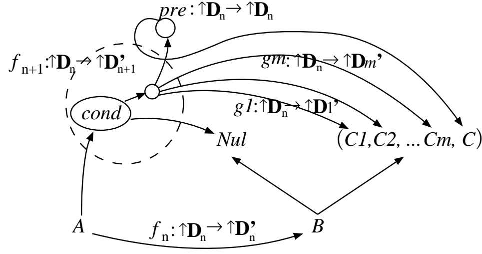

図 2.8: テンプレート 6

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[4]。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。

2.4.6 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 2.6)

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

$$\begin{aligned}
&\textbf{fun } f_{n+1}(A) = \textbf{if } Cond(A) \textbf{ then } Nul \\
&\qquad\qquad\qquad \textbf{else } (g1(A), \ldots, gm(A), f(pre(A))) \\
&\textbf{and } g1(A) \;\; = C1 \\
&\qquad\qquad\quad \vdots \\
&\textbf{and } gm(A) = Cm \\
&\textbf{and } pre(A) = A \\
&\textbf{and } Cond(A) = \text{bool;}
\end{aligned}$$

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 $(\bot)$ に対しては $Cond(\bot) = \text{true}$ が成立する。

[4] 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。

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。

例えば，2.2 節と 2.3 節で作成したプログラム F_0，F_1，F_2を使って$\widetilde{F}_2(32)$ の値は次のように計算できる。

$$\begin{aligned}
\widetilde{F}_2(32) &= F_2(32) \\
&= combine2(3,2) \\
&= nth(nty,1)\frown\text{"-"}\frown\widetilde{F}_2(2) \\
&= nth(nty,1)\frown\text{"-"}\frown\widetilde{F}_1(Units) \\
&= \text{"}thirty\text{-"}\frown F_1(Units) \\
&= \text{"}thirty\text{-"}\frown\text{"}unit\text{"} \\
&= \text{"}thirty\text{-}unit\text{"}
\end{aligned}$$

このようにして，一桁の入力に対する関数を作る前に，二桁の部分が正しく動作しているのが確認できる。

2.6 言語の拡張 — 不変表明式の導入 —

この節では，ISDR 法でプログラムを定義，詳細化する際に，その段階の抽象度で仕様から得られる情報を最大限にプログラムとして記述できるように言語を拡張する。具体的には，プログラム中のデータについて不変的に成り立っている条件 (不変表明式) を直接プログラム中に埋め込めるように言語を拡張する。

この拡張によって，つぎの利点がある。すなわち，bool や “@” のパターンマッチなど，不確定な要素を使ってプログラムを記述した場合も，ある程度関数の振る舞いが制限できるようになるため，デバッグの助けになる。また，詳細化する関数にも元の関数に成り立っている不変表明式が成り立っていないといけないので，それを詳細化の方針やプログラム詳細化の検証に利用可能になる。

不変表明式のシンタックスを BNF 表記で次のように定める。

$$exp \;:=\; exp\ \textbf{where}\ bexp\ \textbf{end}$$

ここで exp は任意の式で，$bexp$ は論理式である。

また，そのセマンティクスは次のように定める。

$$\frac{(exp, \sigma) \rightarrow (exp', \sigma') \quad (bexp, \sigma'[exp'/it]) \rightarrow \mathbf{true}}{(exp\ \mathbf{where}\ bexp\ \mathbf{end}, \sigma) \rightarrow (exp', \sigma')}$$

すなわち，不変表明式が付加された式は，その値を計算した後計算した値 it について不変表明式が成り立っているかチェックされる。

第 3 章

ISDR 法に基づく発展的開発環境 $A_{L}chEmy$

この章では，ISDR 法に基づく発展的開発環境 $A_{L}chEmy$ の概要を述べる。この環境を用いることで，ISDR 法がエディタなどの規模の大きな現実的な問題にも適用可能になる。

まず，ISDR 法のための関数型言語 AL の概要を示す。次に，開発環境の概要を示し，その環境に含まれる支援ツールについて説明する。

3.1　抽象解釈に基づく関数型言語 AL

この節では，ISDR 法のための関数型言語 AL とそのコンパイラを設計する。この言語は，通常の関数型言語と比べ以下の点が異なる。

- 抽象値を通常の値と同様扱うことができる。
- データの具体化を扱うことができる。(ドメインの定義が可能である。)
- プログラムにバージョンやドメインを定義できる。
- バージョンに跨って同じ関数が重複定義できる。
- 全ての関数を定義してなくても任意のバージョンのプログラムを抽象解釈することが可能である。

この様な言語をどのように定義するか，次の節から説明する。まず，AL のシンタックスを述べ，AL プログラムの解釈の手順を概略した後，その手順にしたがってどのように AL

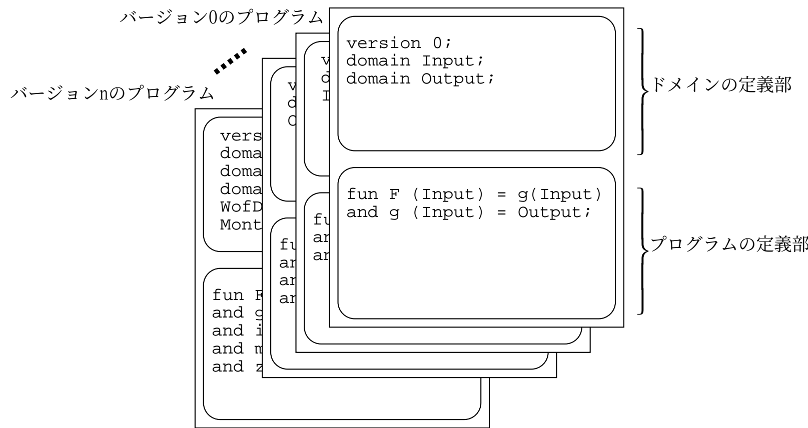

図 3.1: AL プログラムの概要

プログラムを解釈するかを説明する。次に，コンパイラの設計について説明し，AL のプログラミング例を示す。

3.1.1 シンタックス

通常，AL プログラムは図 3.1に示すように，バージョン 0 から最新バージョン n までのプログラムの集合で構成される。そして，個々のバージョン毎のプログラムは，具体化するドメインの定義部とそれに関するプログラムの定義部からなっている。

AL のシンタックスは Standard ML の CORE 言語 [8] とほぼ同じ[1]であるが，プログラムにバージョンをつけるための予約語 `version` とドメインを定義するための予約語が追加されている。予約語 `domain` を用いて新しいドメインを定義し，予約語 `reified` を用いて具体化関係を定義する。

以下に，各予約語の意味を説明する。

- `version` $\langle n\rangle$: n バージョンのプログラムの始まりを表す。
- `domain` $\langle name\rangle$: ドメイン名とドメインのボトムとなる抽象値を同時に定義する。
- $\langle name\rangle$ `reified` $[\langle name1\rangle, \langle name2\rangle \ldots]$: 抽象値 $name$ を，$name1$, $name2$, ... に具体化する。

[1] AL のシンタックスの詳細は付録 A.2を参照のこと。

ここで，`reified`の右辺には，ALが扱うデータの他に集合が記述可能である。集合の定義は，次のような“..”を使って行う。

- i,j .. k: 数iからjおきにkまでの集合を表す。すなわち，次の集合を表している。

  $$\{n|n = i + j * m \wedge m = 0, 1, \ldots \wedge n \leq k\}$$

  ただし，iが整数のとき整数集合を表し，実数のとき実数集合を表す。また，j，kは省略可能である。jを省略した場合，i以上からj以下の集合を表し，kを省略した場合，i以上の無限集合を表す。

- .. i,j: 数jから$-i$おきの無限集合を表す。すなわち，次の集合を表している。

  $$\{n|n = i - j * m \wedge m = 0, 1, \ldots\}$$

  ただし，iが整数のとき整数集合を表し，実数のとき実数集合を表す。また，iは省略可能で，この場合j以下の無限集合を表す。

例えば，“`Tens`”を20から99までの具体値の集合に具体化する場合，次のように記述する。

```
Tens reified [20..99];
```

また，現バージョンのALでは，任意の集合を表現するために値から真偽値への関数に具体化可能である。このような関数による集合表現はALの拡張表現であり，将来変更される可能性がある。例えば上記の定義は拡張表現を用いて次のように書くこともできる。

```
Tens reified [(fn n=>(n>=20) andalso (n<=99))];
```

ISDR法では，具体化する値は，構造を持たない基本値の他に，レコードや再帰的なデータ構造を持つレコードを許している。そこで，ALでは表3.1に示すように，整数，実数，文字列，真偽値などの基本値の他に，レコード[2]およびリストに具体化可能である。この表中の`a`, `b`, `c`はドメイン名である。

例えば，以下はカレンダーを表示するALプログラムの一部である。

[2]Standard MLのレコードは“{”, “}”で記述するが，これはALのレコードとは異なるので注意すること。ALでは，Standard MLでいうレコードには具体化できない。また，リストはStandard MLと全く同様に扱うことが可能である。

言語中での型の表記	説明	例
bool	真偽値	true, false
int	整数	321, ~32
real	実数	35.2, ~12e3
string	文字列	"Hello"
('a * 'b * 'c * ...)	レコード	(Pos,0,Int)
'a list	文字列	[Pos,0,Int]

表 3.1: AL で扱うことができるデータ型

```
version 0;                    (* バージョン 0 のプログラム *)
domain Month;                 (* <-|                       *)
domain Year;                  (* <-+- 新しいドメインの定義 *)
domain Output;                (* <-|                       *)
                 (* ---- 関数の定義 ---- *)
fun F (Month, Year) = Output;

version 1;                    (* バージョン 1 のプログラム *)
domain WanD;                  (* <-|                       *)
domain Wtmp;                  (* <-+- 新しいドメインの定義 *)
domain Day;                   (* <-|                       *)
Output reified [[WanD]];      (* <--- ドメイン具体化の定義 *)
                 (* ---- 関数の定義 ---- *)
fun F (mth, yr) = week(fstday(mth, yr),Day,month(mth, yr))
and week (Wtmp, Day, Day) =
    if daygrt(Day, Day) then []
    else WanD::week(Wtmp, Day, Day)
and fstday (Month, Year) = Wtmp
and month (Month, Year) = Day
and daygrt (Day, Day) = Bool;
```

3.1.2 AL プログラムの解釈の手順

前節のシンタックスで定義された AL プログラムの解釈手順は，おおまかに次のようになる。

1. 各バージョン毎に有効なドメインを構築する。

```
version 0;
domain Output;

version 1;
domain WanD;
Output reified [[WanD]];

version 2;
domain DofW;
domain Day;
WanD reified [(DofW, Day)];

version 3;
DofW reified [Sun, Mon, Tue,
        Wed, Thu, Fri, Sat];
```

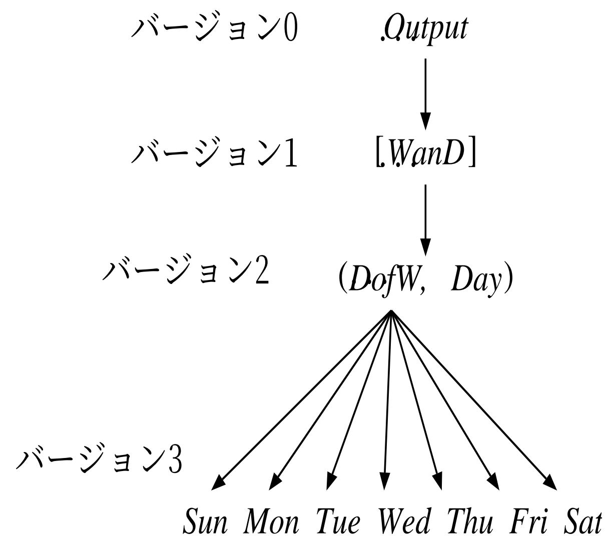

図 3.2: ドメインの構築例

2. 各バージョン毎に静的型推論する。

3. 推論された型情報をもとに次のように抽象解釈する。

    (a) 最新の関数呼び出し，

    (b) 推論された型に合わせて実引数を抽象化する。

    (c) もし解釈ができなかったらそれ以前のバージョンを呼び出す。

以下の節から，上記の手順にしたがって，ドメインの構築法，抽象解釈のための型推論とセマンティクスの定義を説明する。

3.1.3 ドメインの構築

任意のバージョンで有効なドメインの構築は，それまでのバージョンのプログラム中のドメインの定義部のみを用いて行うことができる。例えば，カレンダーを表示する例の場合，図 3.2の様に $Output$ のドメインを構築できる。

3.1.4 静的型推論

AL プログラムの静的型推論は各バージョン毎に独立に行う。しかしながら，具象化関係のある関数の型は，それらの型をドメインとみなした時，全て同じになっている必要があ

$$(\textsc{abst})\frac{\Gamma, x:\sigma \vdash M:\tau}{\Gamma \vdash \lambda x:\sigma.M:\sigma\to\tau} \qquad (\textsc{app1})\frac{\Gamma\vdash M:\sigma\to\tau \quad \Gamma\vdash N:\sigma}{\Gamma\vdash M\ N:\tau}(\sigma\text{が基底型以外})$$

$$(\textsc{if})\frac{\Gamma\vdash L:\mathbf{bool} \quad \Gamma\vdash M:\sigma \quad \Gamma\vdash N:\sigma}{\Gamma\vdash \mathbf{if}\ L:\mathbf{bool}\ \mathbf{then}\ \Gamma\vdash M:\sigma\ \mathbf{else}\ \Gamma\vdash N:\sigma}$$

$$(\textsc{fix})\frac{\Gamma, x:\sigma\vdash M:\sigma}{\Gamma\vdash \mu x:\sigma.M:\sigma} \qquad (\textsc{con})\frac{}{\Gamma\vdash c:\uparrow\mathbf{D}}(c\in\uparrow\mathbf{D}) \qquad (\textsc{bool})\frac{}{\Gamma\vdash \mathrm{bool}:\mathbf{bool}}$$

$$(\textsc{app2})\frac{\Gamma\vdash M:\sigma'\to\tau \quad \Gamma\vdash N:\sigma}{\Gamma\vdash M\ N:\tau}(\sigma\text{と}\sigma'\text{は基底型}\wedge\sigma\cap\sigma'\neq\emptyset)$$

$$(\textsc{total})\frac{\Gamma\vdash M:\sigma}{\Gamma\vdash M:data(\mathbf{D})}(\sigma=\uparrow\mathbf{D})$$

表 3.2: AL の静的型推論規則

るので，これを用いてバージョン間の整合性のチェックを行う事ができる。

AL の静的型推論規則を表 3.2に示す。この表中の規則 (ABST), (APP1), (IF), (FIX) は一般の関数型言語の場合[3] と同じであるが，規則 (CON), (BOOL), (APP2), (TOTAL) は AL 特有の推論規則である。ただし，表中のΓは変数と型の組の列である。また，これらの規則を用いてプログラム中の関数の型が一意に決まらなかったらエラーとする。

規則 (CON) は，AL において型はそのバージョンで有効なドメインの具体値の集合であることを示し，規則 (BOOL) は型 **bool** には ture, false の他にも，抽象値 bool が含まれている事を示す。

規則 (APP2) は，定義域と値域が異なる型になる関数同士を結合させるために必要となる。これによって，関数の再利用性が高められる。例えば，次のように集合 $\{Neg, 0, 1, ..\}$ 上の関数 inc の定義中に整数上の演算 “<=”, “+” を用いる時に必要になる。

```
inc (x, y) = if (x <= 0) then x+1
```

この規則を使うと，式に静的に型を与えても実行時に型エラーを起こす可能性がある。そのため，抽象実行時に引数を抽象化する前に動的検査を行う必要がある。

(TOTAL) は型推論の最後にだけ適用できる特別な規則である。この規則によって，トップレベルの式は必ず何らかの値が返るようになる。

[3]一般的な関数型言語の推論規則については文献 [9] などを参照のこと

以下に，型推論とその情報に基づいた詳細化の検証例を示す。

まず，バージョン 0 のプログラムが次のように定義されていたとする。

$$\begin{aligned}\textbf{Month} &= \langle\{Month\}, \emptyset\rangle\\ \textbf{Year} &= \langle\{Year\}, \emptyset\rangle\\ \textbf{Output} &= \langle\{Output\}, \emptyset\rangle\end{aligned}$$

$$\textbf{fun}\quad F\quad (Month,\ Year) = Output;$$

これを型推論すると，次のようになる。

$$\tilde{F}_0 \ :\ \uparrow \textbf{Month}* \uparrow \textbf{Year} \rightarrow \textbf{Output}$$

次に，バージョン 1 のプログラムが次のように定義されているとする。

$$\begin{aligned}\textbf{Month} &= \langle\{Month\}, \emptyset\rangle\\ \textbf{Wtmp} &= \langle\{Wtmp\}, \emptyset\rangle\\ \textbf{Year} &= \langle\{Year\}, \emptyset\rangle\\ \textbf{Day} &= \langle\{Day\}, \emptyset\rangle\\ \textbf{WanD} &= \langle\{WanD\}, \emptyset\rangle\\ \textbf{Output} &= \langle\{Output \prec [WanD]\}, \{Output \prec [WanD]\}\rangle\end{aligned}$$

$$\begin{array}{lll}\textbf{fun} & F\ (mth, yr) = week(fstday(mth, yr), Day, month(mth, yr)) & \\ \textbf{and} & week\ (Wtmp, Day, Day) = \textbf{if}\ daygrt(Day, Day)\ \textbf{then}\ [] & \\ & \qquad\qquad \textbf{else}\ WanD :: week(Wtmp, Day, Day); & \\ \textbf{and} & fstday\ (Month, Year) = Wtmp & \\ \textbf{and} & month\ (Month, Year) = Day & \\ \textbf{and} & daygrt\ (Day, Day) = \mathrm{bool}; & \end{array}$$

これを，各ドメインの葉の部分を使って型推論すると次のようになる。

$$\begin{aligned}daygrt &: \uparrow \textbf{Day}* \uparrow \textbf{Day} \rightarrow \textbf{bool}\\ month &: \uparrow \textbf{Month}* \uparrow \textbf{Year} \rightarrow \uparrow \textbf{Day}\\ fstday &: \uparrow \textbf{Month}* \uparrow \textbf{Year} \rightarrow \uparrow \textbf{Wtmp}\\ week &: \uparrow \textbf{Wtmp}* \uparrow \textbf{Day}* \uparrow \textbf{Day} \rightarrow \uparrow \textbf{Output}\end{aligned}$$

$$(\textsc{bool1})\frac{}{\text{bool} \Longrightarrow \text{true}} \qquad (\textsc{bool2})\frac{}{\text{bool} \Longrightarrow \text{false}}$$

$$(\textsc{if1})\frac{}{\textbf{if}\ \text{ture}\ \textbf{then}\ M\ \textbf{else}\ N \Longrightarrow M}$$

$$(\textsc{if2})\frac{}{\textbf{if}\ \text{false}\ \textbf{then}\ M\ \textbf{else}\ N \Longrightarrow N}$$

$$(\textsc{fix})\frac{}{\mu\ x{:}\sigma.M \Longrightarrow M[x := \mu\ x{:}\sigma.M]}$$

$$(\textsc{app})\frac{a : \sigma}{(\lambda\ x{:}\sigma.M)\ a \Longrightarrow M[x := a]}$$

$$(\textsc{select})\frac{f{:}\sigma \to \tau \quad f'\ a \Longrightarrow a'' \quad a'{:}\tau}{f\ a \Longrightarrow a'}(a' \preceq a'' \wedge (f' \sqsubseteq f \vee f \sqsubseteq f'))$$

$$(\textsc{abst})\frac{f{:}\sigma \to \tau \quad a'{:}\sigma}{f\ a \Longrightarrow f\ a'}(a' \preceq a)$$

表 3.3: AL の操作的意味規則

結局，Fの型は，次のようになる。

$$\widetilde{F}_1 \quad : \quad \uparrow \textbf{Month} * \uparrow \textbf{Year} \to \textbf{Output}$$

ここまでの型推論結果より，つぎのようにこのプログラムの一貫性の検証を行うことができる。すなわち，バージョン 0 のプログラムの推論結果と合わせると関数 Fは型の観点から一貫性がとれていることが確認できる。

3.1.5 操作的意味

AL プログラムの抽象解釈は，関数の具象化関係に基づき最新の関数を呼び出し，静的に推論した型情報に基づき実引数や返り値を抽象化して行う。

AL の操作的意味を表 3.3に示す。この表中の規則 (IF1), (IF2), (FIX) は一般の関数型言語の場合[4]と同じであるが，その他は AL 特有の規則である。

[4]一般的な関数型言語の操作的規則については文献 [9] などを参照のこと

規則 (BOOL1) および (BOOL2) は抽象値 bool の実際の真偽値は実行時に決定することを示し，規則 (APP) は型の整合性を実行的にチェックすることを示す。これは型推論規則の (APP2) により実行時に型の不整合が起こる可能性があるためである。

3 つの規則 (APP), (SELECT), (ABST) で関数の選択と実引数の抽象化を行っている。その，具体的な関数呼び出しの手順は次のようになる。

1. 規則 (SELECT) によって呼び出されるべき関数を選びだし，
2. 規則 (ABST) によって実引数を選ばれた関数にしたがって抽象化し，
3. 規則 (APP) によって動的に引数の型を調べ，
4. 実際に関数を呼び出す。

その関数が実行できなかった場合，規則 (SELECT) を再びやり直して他の関数を呼び出す。すなわち，規則 (SELECT) は関数の具象化関係を用いて呼び出す関数を選択する機構を表している。もし，このプログラムが定義された時よりも新しいバージョンの関数が選ばれた場合，必要以上に具体的な値 a''が計算される可能性があるので，これを抽象化した値 a' を $f\ a$ の返り値と決定している。また，この規則によって選択された関数にあわせて実引数を抽象化する規則が (ABST) である。

また，規則 (SELECT) だけでは，実際にどのバージョンの関数を選べばよいのか一意に決定できない。しかしながら，各バージョンの関数に定義 8の関係が成り立っていれば，どのバージョンを選んでも計算が可能ならば結果は同じになる。

実際の解釈アルゴリズムは，規則 (SELECT) を適用できる最新バージョンの関数から順に計算を試みるようになっている。

3.1.6 AL コンパイラの設計

一般に，コンパイラとはある言語のソースプログラムをあるマシンの実行コードに変換することをいう。この場合，マシンは実際のハードウエアだけを指すわけではなく，ソフトウェアで実行可能な抽象マシンも含む。AL は Standard ML に，その記法も意味も似ているので，Standard ML インタープリタを抽象マシンとみなす事により，容易に変換できる。その上，Standard ML の意味は厳密に定義されているので，AL の意味を Standard ML 上にマップすることにより，AL の意味の厳密な議論も可能になる。

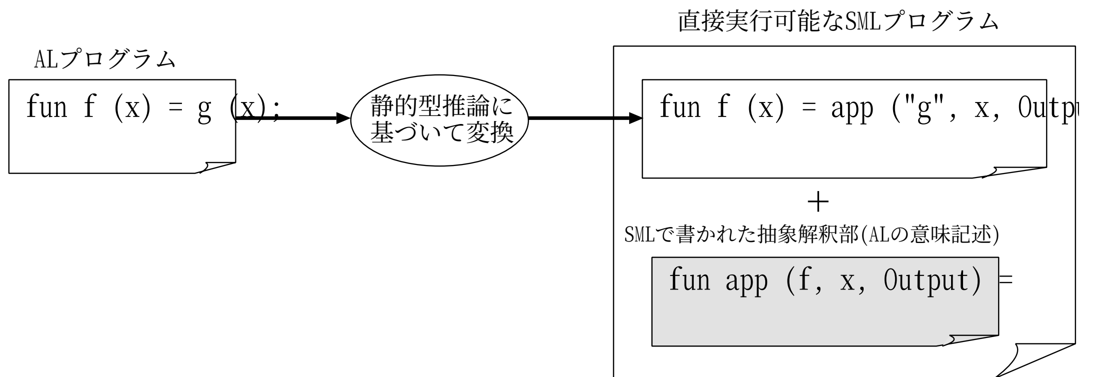

図 3.3: AL コンパイラの設計

そこで，AL プログラムから Standard ML プログラムへ書き換えるようなコンパイラを設計した。その概要を図 3.3に示す。この図にあるように，AL プログラムはまず型推論され，その情報をもとに関数呼び出しの部分[5]が書き換えられ，Standard ML で書かれた抽象解釈部と合成され，直接実行可能な Standard ML に変換される。具体的には，抽象解釈部には関数呼び出しのための関数 app が定義されており，AL プログラムの関数呼び出し部分はすべてその関数が返すべき型の情報とともに，関数 app によって置き換えられる。そして，その抽象解釈部を AL プログラムに挿入することにより，Standard ML プログラムとなる。Standard ML プログラムは SML/NJ などのインタープリタおよびコンパイラにより実行可能である。

次に，どのように表 3.2に従い型推論をするかのアルゴリズムを示し，つぎに表 3.3に従った app の抽象解釈アルゴリズムを示す。

型推論アルゴリズム

型推論のアルゴリズムは次のように定義できる $\mathcal{W}$によって与えることができる。ここで，$\mathcal{W}$は，変数と型の列Γと式 Eを受け取って，その式の型τとその式から得られる型の情報をΓへの代入 Sとして返す。

代入 Sに対して $S\ \Gamma \vdash E : \tau$が成立するとき $\mathcal{W}(\Gamma, E) = (S, \tau)$ になるような $\mathcal{W}$は次のように Eの構造よって再帰的に決定可能である。

[5]ただし，“=” は書き換えず，そのまま ML に解釈させる。

1. Eが定数または変数 a の時,

   (a) Γ 中に a の型が定義されていなければ，τは新しい型変数βとなる。

   (b) Γ 中に a の型がτ_1と定義されていれば，τはτ_1となる。

   いずれの場合も $S = Id$(恒等代入) となる

2. Eが **if** e **then** e_1 **else** e_2の時,

$$\begin{aligned}\tau &= U_1 \circ \tau_2 \\ S &= U_1 \circ S_2 \circ S_1 \circ U_0 \circ S_0\end{aligned}$$

   となる。ただし,

$$\begin{aligned}&\mathcal{W}(\Gamma, e) = (S_0, \tau_0) \\ &U_0 = unify(\tau_0, \mathbf{bool}) \\ &\mathcal{W}((U_0 \circ S_0)\Gamma, e_1) = (S_1, \tau_1) \\ &\mathcal{W}((S_1 \circ U_0 \circ S_0)\Gamma, e_2) = (S_2, \tau_2) \\ &U_1 = unify(S_2\tau_1, \tau_2)\end{aligned}$$

   である。

3. Eが$\mu x.e$ の時,

$$\begin{aligned}\tau &= (U \circ S)\beta \\ S &= U \circ S_1\end{aligned}$$

   となる。ただし,

$$\begin{aligned}&\mathcal{W}(\Gamma + \langle b : \beta\rangle, e) = (S_1, \tau_1) \qquad \text{ただし，}\beta\text{は新しい型変数} \\ &U = unify(S_1\beta, \tau_1)\end{aligned}$$

   である。

4. Eが $e\ e'$の時,

$$\begin{aligned}\tau &= U\beta \\ S &= U \circ S_2 \circ S_1' \circ S_1\end{aligned}$$

   となる。ただし,

$\mathcal{W}(\Gamma, e) = (S_1, \tau_1 \to \sigma_1)$
ここで，τ_1が基底型の時，τ_1からこれと共通要素を持つ型τ_1'への代入を S_1'とする。
τ_1が基底型で無い時，$S_1' = Id$ とする。
$\mathcal{W}(S_1' S_1 \Gamma, e') = (S_2, \tau_2)$
$U = unify(S_2 S_1'(\tau_1 \to \sigma_1), \tau_2 \to \beta)$　ただし，βは新しい型変数

である。

5. Eが$\lambda v.e$ の時,

$$\begin{aligned} \tau &= S_1 \beta \to \tau_1 \\ S &= S_1 \end{aligned}$$

となる。ただし,

$$\mathcal{W}(\Gamma + \langle v : \beta \rangle, e) = (S_1, \tau_1) \qquad \text{ただし，}\beta\text{は新しい型変数}$$

である。

この定義中の $unify$は単一化アルゴリズムである[6]。
また，主関数 Fについては，その型推論アルゴリズムで得られた型情報を元に全域化する。

抽象解釈アルゴリズム

AL の意味記述は，図 3.3中に現れるように，Standard ML によって関数 app によって表現されている。app は，関数と値を受け取り，その値に関数を適用した結果を out の型に抽象化して返す関数である。ここで，app を定義する前に，抽象解釈の下準備として AL 中で定義されたすべての関数を集めてその詳細化関係のリストをつくっておく必要がある。

app のアルゴリズムは次のように Standard ML の例外処理を利用することによって，最新の関数を呼び出す仕組みになっている。

fun $app(f, a, out) =$
　　$app_sub(f, a)$ を呼び出し，その結果を型 out に合わせて抽象化する。

[6]単一化アルゴリズムについての詳細は [10] などを参照のこと

and $app_sub(f, a) =$

  let

    val $newf$ = fの関数リスト中で最新の関数

  in

    引数をこの入力の型に合わせ $newf$ を呼び出す **handle**

    $ABSTRACTION_FAIL$ => (* 引数が型と合わない*)

      関数リストからこの関数を取り除いて再帰する。

    | $Match$ => (* その値について定義されていなかった。*)

      関数リストからこの関数を取り除いて再帰する。

  end

3.1.7 AL プログラミング例

この節では，AL でプログラムを構成する例を示す。具体的には,「月と年を入力しその月のカレンダーを表示する」プログラムを構成する。

例えば，このプログラムは引数として 10 と 1997 を与えると次のリストを返す。

$$[(Wed,1), (Thu,2), (Fri,3), (Sat,4), (Sun,5), (Mon,6), (Tue,7), (Wed,8), (Thu,9), (Fri,10), (Sat,11), (Sun,12), (Mon,13), (Tue,14), (Wed,15), (Thu,16), (Fri,17), (Sat,18), (Sun,19), (Mon,20), (Tue,21), (Wed,22), (Thu,23), (Fri,24), (Sat,25), (Sun,26), (Mon,27), (Tue,28), (Wed,29), (Thu,30), (Fri,31)]$$

抽象化 まず，ISDR 法に基づいて問題を抽象化し，それに対応した AL プログラムを構成する。ここでは，月の集合を `Month` に，年の集合を `Year` に，カレンダーの集合を `Output` に抽象化する。

```
version 0;
domain Month;                   (* 月を表すドメイン *)
domain Year;                    (* 年を表すドメイン *)
domain Output;

fun F (Month, Year) = Output;
```

version 1　次に，出力のフォーマットを定める。出力のフォーマットを定める。ここでは出力を，曜日と日付を含む要素 $WanD$ のリストであると定める。すると，次のようなバージョン 1 の AL プログラムが定義できる。

```
version 1;
domain WanD;                        (* 曜日と日付のペアを表す。*)
domain Wtmp;                        (* 曜日の整数表現。*)
domain Day;                         (* 日付を表す。 *)
Output reified [[WanD]];

fun F (mth, yr) = week(fstday(mth, yr),Day,month(mth, yr))

and week (Wtmp, Day, Day) =          (* 曜日と日付のペアをつくる。*)
    if daygrt(Day, Day) then []
    else WanD::week(Wtmp, Day, Day)
and fstday (Month, Year) = Wtmp      (* mth 月 1 日の曜日の計算 *)
and month (Month, Year) = Day        (* m 月の最終日を計算する。 *)
and daygrt (Day, Day) = Bool;        (* 日付の大小関係 *)
```

ここで用いたアルゴリズムは，yr の元旦の曜日から mth 月 1 日の曜日を計算し，その曜日から一ヶ月分のカレンダーを出力するようなものである。

version 2　そして，$WanD$ を曜日を表す抽象値 $DofW$ と日付を表す抽象値 Day のペアに具体化する。すると，次のように AL プログラムが詳細化できる。

```
version 2;
domain DofW;                          (* 曜日を表すドメイン。*)
WanD reified [(DofW, Day)];

fun F (mth, yr) = week(fstday(mth, yr), Day, month(mth, yr))

and week (Wtmp, Day, Day) =           (* 曜日と日付のペアをつくる。*)
    if daygrt(Day, Day) then []
    else (DofW, Day)::week(Wtmp, Day, Day);
```

version 3　さらに，日付と月を具体値にし，AL プログラムを次のように具体化する。

```
version 3;
Day reified [condi (fn n=> n>0 andalso n <= 32)];
Month reified [condi (fn n=> n>0 andalso n <= 12)];
```

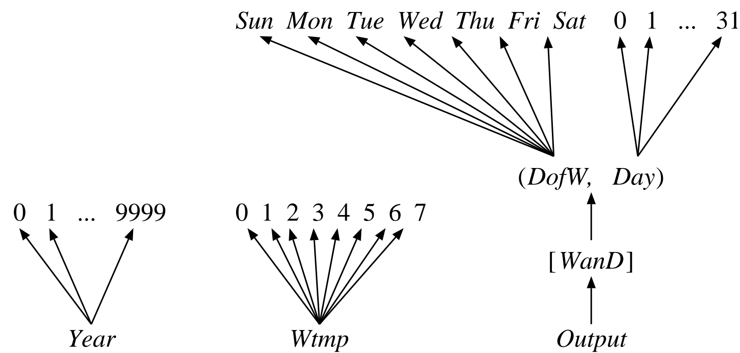

図 3.4: 具体化されたドメイン

```
fun F (mth, yr) = week(fstday(mth, yr), 1, month(mth, yr))

and week (Wtmp, st, en) =    (* 曜日と日付のペアをつくる。*)
    if daygrt(st, en) then []
    else (DofW, st)::week(Wtmp, st+1, en)
and month (m, Year) =        (* m 月の最終日を計算する。 *)
    if (m <> 2) then
        nth([31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31],m-1)
and daygrt (m1, m2) = (m1 > m2); (* 日付の大小関係 *)
```

抽象解釈例 1　ここまでで，2 月以外の月について与えられた月の日にち分の要素を返すプログラムが構成できた。例えば， `F (10, Year)` の計算は次のようになる。

```
- F (10, Year);
val it = [(DofW, 1),(DofW, 2),(DofW, 3),(DofW, 4),..,.(DofW, 31)]
```

version 4　ここで，曜日と年，曜日の正数表現を具体値に具体化し，うるう年以外の処理をすべて記述することにする。ここまでに詳細化されたドメインを図 3.4に示す。

うるう年以外の処理を記述した AL プログラムは次のようになる。

```
version 4;
DofW reified [Sun, Mon, Tue, Wed, Thu, Fri, Sat];
Wtmp reified [condi (fn n=> n>=0 andalso n < 7)];
Year reified [condi (fn n=> n>=0 andalso n < 9999)];

val daynames = [Sun, Mon, Tue, Wed, Thu, Fri, Sat];
fun F (mth, yr) = week(fstday(mth, yr), 1, month(mth, yr))
```

```
and week (w, st, en) =   (* 曜日と日付のペアをつくる。*)
    if daygrt(st, en) then []
    else (nth(daynames, w), st)::week(Wtmp, st+1, en)
and jan1st (yr) =        (* 一月一日の曜日の計算 *)
      (yr+(yr-1) div 4 - (yr -1) div 100 + (yr-1) div 400) mod 7
and months (0, yr) = 0   (* 元旦から n 月までの日数計算 *)
  | months (n, yr) =
    let
        val (i m) = month(n, yr)
    in
        m + months(n-1, yr)
    end
and fstday (mth, yr) =   (* mth 月 1 日の曜日の計算 *)
        (months(mth-1, yr)+jan1st(yr)) mod 7;
```

抽象解釈例 2 ここまでのプログラムを用いると，各月のカレンダーはそれぞれ次のように計算される。

- 1 月のカレンダー：バージョン 4 のプログラムが呼び出される。

$$F_4(1, 1997) \quad = \quad [(\mathit{Wed}, 1), (\mathit{Thu}, 2), (\mathit{Fri}, 3), \ldots]$$

- 2 月のカレンダー：バージョン 2 のプログラムが呼び出される。

$$\begin{aligned} F_4(2, 1997) &= \mathit{week}(\mathit{fstday}(2, 1997), 1, \mathit{month}(2, 1997)) \\ &\quad (\mathit{month}_1(2, 1997) = \mathit{Day} \text{ となるの期待する型と一致しない}) \\ F_3(2, \mathit{Year}) &= \mathit{week}(\mathit{fstday}(2, \mathit{Year}), 1, \mathit{month}(2, \mathit{Year})) \quad \text{(これも失敗)} \\ F_2(\mathit{Day}, \mathit{Year}) &= \mathit{week}(\mathit{fstday}(\mathit{Day}, \mathit{Year}), \mathit{Day}, \mathit{month}(2, \mathit{Year})) \\ &= [(\mathit{DofW}, \mathit{Day}), (\mathit{DofW}, \mathit{Day}), (\mathit{DofW}, \mathit{Day}), \ldots] \end{aligned}$$

- 3 月以降のカレンダー：バージョン 3 のプログラムが呼び出される。

$$\begin{aligned} F_4(3, 1997) &= \mathit{week}(\mathit{fstday}(3, 1997), 1, \mathit{month}(3, 1997)) \\ &\quad (\mathit{months}_4(3, 1997) \text{ が計算できないので失敗。}) \\ F_3(3, \mathit{Year}) &= [(\mathit{Wed}, \mathit{Day}), (\mathit{Thu}, \mathit{Day}), (\mathit{Fri}, \mathit{Day}), \ldots] \end{aligned}$$

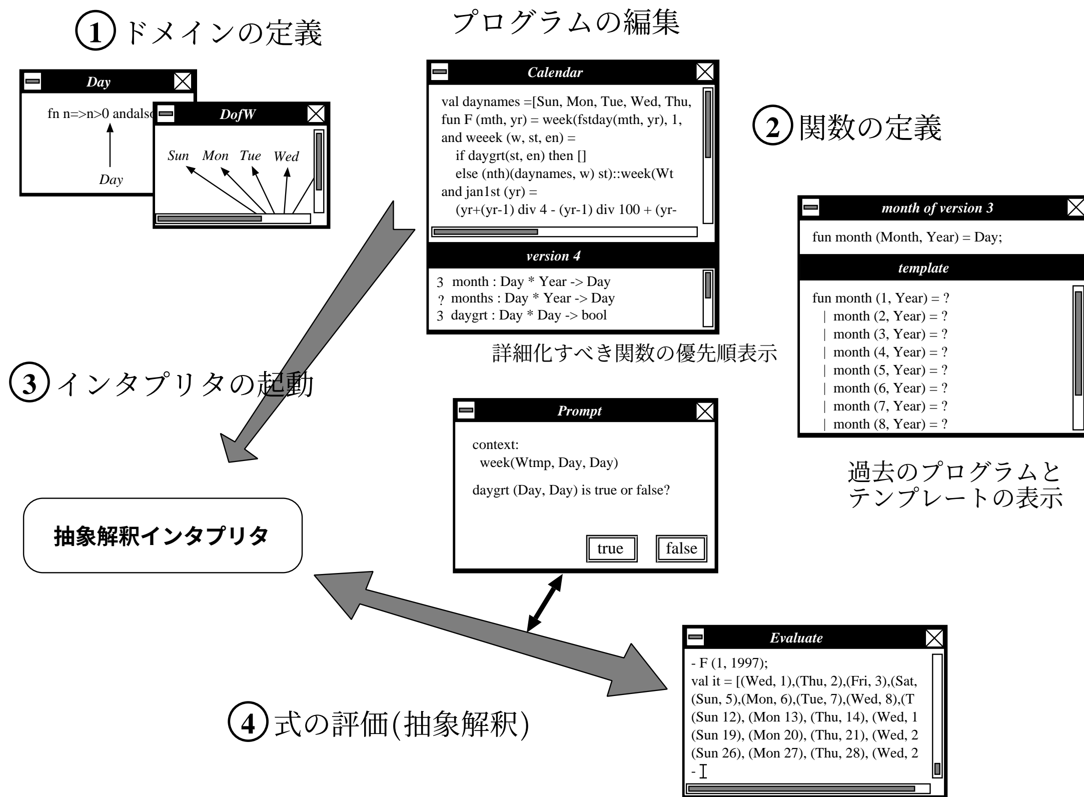

図 3.5: *ALchEmy*の概念図

3.2 *ALchEmy*の概要

開発環境 *ALchEmy* は，Java で実装されたエディタやツールで構成される。この環境では次のように ISDR 法の開発プロセスを支援する。(図 3.5)

1. データの詳細化とドメインの定義　具体化されるべき抽象値は一覧表示され，その中から今のバージョンで具体化されるデータを選択する。GUI で行ったドメインと具体化の定義は，内部では AL の文法に変換される。また，ドメインの具体化の様子をグラフィカルに表示する。

2. プログラムの定義　以前のバージョンのプログラムはどの関数を具象化すべきかの優先順位つきで表示され，具体化したデータに関連したプログラムが容易に判別可能となっている。また，データの具体化にあわせて関数をどのように具象化すべきかのテンプレート

が表示され，過去のバージョンと見比べながらの関数定義が可能となっている。

3.4. プログラムの評価　このバージョンで定義された AL プログラムは型推論されその情報をもとに Standard ML の記法に変換され，ドメインの情報，以前のバージョンのプログラム，Standard ML で書かれた抽象解釈部と共に合成され，最終的には Standard ML インタプリタで実行される。式の評価は対話的に行うことができる。また，式の評価中に未解決な部分が現れたらプロンプトを表示して人間にそれを解決させる。例えば，bool が現れたら，その式が評価された文脈と共にその値を ture にするか false とするか返答を求めるプロンプトが表示される。

3.2.1　開発支援ツール

ALchEmy には，ドメインとプログラムの定義，詳細化を支援するために次のようなツールが含まれている。

- 詳細化が正しいかどうかのチェックするツール
- どのデータや関数を詳細化すべきかのガイド
- どのように関数を詳細化すべきかのガイド

以下では，このような開発支援ツールをどのように設計したか順に述べる。

詳細化が正しいかどうかのチェックするツール

これは，ISDR 法の詳細化の定義にあわせて各バージョン毎に詳細化が正しいかどうか検査するツールである。ISDR 法では，2.3.2節で述べたように各段階で（1）ドメインが正しく具体化されているか,（2）プログラムが正しく詳細化されているか,（3）詳細化した関数が仕様を満たしているかの 3 点について調べる必要がある。以下に開発環境上で，どのようにしてこれらをチェックするかを述べる。

具体化したドメインには，定義 4が成り立っている必要がある。開発環境では，ドメイン編集中に具体化した値が具体化するドメインに元もと含まれている値でないことをチェックすることにより，任意のドメインについてその合流とループが起こらないことを保証している。また，ドメインへの要素の追加は，元に含まれている値を具体化するしかないの

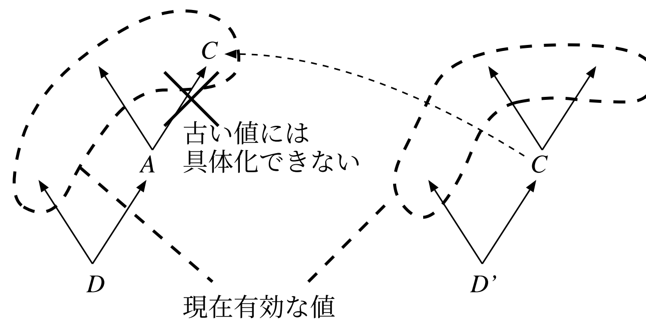

図 3.6: 抽象度の異なる値への具体化の禁止

で，この環境上でドメインを編集している間はその中に余計な要素が含まれることはない。この他にも，この環境では，具体化できる値は新しい値または他のドメインの具体集合のみに限定している (図 3.6)。なぜならば，同じバージョンのプログラムに具体化の度合が違うドメインが存在し，抽象度の異なる値が混在してしまうことを防ぐためである。

プログラムが正し詳細化されてるかチェックするためには，新しく定義したプログラムについて，定義 9が成り立っているか調べなければいけない。ドメインや関数の集合については，この環境上でプログラムを編集する以上，減ることはありえないので，この定義の関係は必ず成り立っている。主関数の詳細化が正しいかどうかを調べるために，新しく定義した主関数が以前のバージョンと定義 8の関係があるかどうかチェックするツールを用意している。このツールは，まず主関数の入力集合をすべて生成し[7]，そのすべての入力に対し主関数の返り値を計算し，前のバージョンと今のバージョンの返り値を比較することで，すべての入力に対して詳細化関係が成り立っていることを確認している。この方法は，扱っている値が少ない初期のバージョンのプログラムに対しては有効であるが，扱う値が増えてくると，その計算量は増大しプログラム中に不確定な部分があれば膨大な数のプロンプトに答えなければ行けないので現実的なチェック方法では無くなってしまう。それに対処するために，このツールではチェックする入力集合を制限することを許している。プログラマはチェックする入力集合が成り立つ条件を式で記述することができ，その条件が成り立つ入力値について上記のチェック機構を働かせることで，現実的な時間内に詳細

[7] このツールでは，遅延評価関数によって入力集合を計算する事により，無限集合も生成可能になっている。しかしながら，その場合，詳細化のチェックは途中で止めない限りシステムがエラーを起こすまで永遠に続くことになる。

化チェックを終らせることができる。これによって，部分的な詳細化しかチェックできなくなるが，通常，バグが入り詳細化が成り立たくなっている部分は一部の入力に対してのみなので，その入力を注意深く見つけ出すことができれば，部分的なチェックだけで十分実用的な場合が多い。

現在，環境上で仕様は自然言語でしか記述することができない。そのため，仕様を利用してプログラムの検証を行うことができない。しかし，将来仕様を論理式などの仕様記述言語で記述できれば，それを使ったプログラムの検証も可能になる。

どのデータや関数を詳細化すべきかのガイド

$\mathcal{A}chEmy$ は，データや関数を，優先度つきで表示する機構を持っている。これは，プログラマがどの値や関数を次に詳細にすべきかガイドを行うためのツールである。その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。

どのように関数を詳細化すべきかのガイド

これは，2.4 節で示した関数詳細化のテンプレートに従って，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。具体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。

第 4 章

ISDR 法による構成例

4.1 在庫状況の計算

この節では 2.4 節で示した詳細化方針に従ってプログラムを構成する例題を示す。ただし，ここでは仕様の細部がまだ確定していない段階からプログラムをつくり始め，段階的に仕様とプログラムを詳細にする。具体的には「製品の入出荷リストから在庫状況を計算する」プログラムを構成する。

抽象化段階　入力集合を抽象値 $Input$ に，出力集合を抽象値 $Output$ に抽象化する。この時，入力ドメイン $\mathbf{D}_0^{\mathrm{I}}$と出力ドメイン $\mathbf{D}_0^{\mathrm{O}}$は以下のようになる。

$$\begin{aligned} \mathbf{D}_0^{\mathrm{I}} &= \langle \{Input\}, \emptyset \rangle \\ \mathbf{D}_0^{\mathrm{O}} &= \langle \{Output\}, \emptyset \rangle \end{aligned}$$

そして，原始プログラムの主関数 F_0は，以下のように $Input$ から $Output$ への関数となる。

$$\textbf{fun}\ F_0(Input) = Output;$$

version 1 — $Output$ の具体化 —　まず，出力のフォーマットの仕様を決定する。ここでは，出力を先頭部分 $Header$，在庫報告 $Stocks$，末尾部分 $Trailer$ からなるとする。つまり，$Output$ をレコード $(Header, Stocks, Trailer)$ に具体化する。

この時，出力ドメインは次のようになり入力ドメインは $\mathbf{D}_0^{\mathrm{I}}$とおなじである。

$$\mathbf{D}_1^{\mathrm{O}} = \mathbf{D}_0^{\mathrm{O}}[Output \prec (Header, Stocks, Trailer)]$$

そして，F_1をテンプレート 4 を使って次のように定義する。

$$\begin{aligned}
\textbf{fun } F_1(x) &= (h(x), rep(x), t(x)) \\
\textbf{and } h_1(x) &= Header \\
\textbf{and } rep_1(x) &= Stocks \\
\textbf{and } t_1(x) &= Trailer
\end{aligned}$$

version 2 — *Stocks* の具体化 —　さらに，出力の構造を明らかにしながら仕様を詳細化しプログラムを定義する。ここでは，*Stocks* は個々の在庫の *Stock* のリストで構成されていることにする。そこで，*Stocks* を再帰的なデータ構造 *Slist* に具体化する。この時，出力ドメインは次のようになり入力ドメインは $\mathbf{D}_1^{\mathrm{I}}$とおなじである。

$$\begin{aligned}
\mathbf{D}_2^{\mathrm{O}} &= \mathbf{D}_1^{\mathrm{O}}[Stocks \prec Slist] \\
&\quad \textbf{where } Slist = Nulstock \mid (Stock, Slist)
\end{aligned}$$

そして，$Prog_2$をテンプレート 6 を使って次のように定義する。

$$\begin{aligned}
&\textbf{fun } rep_2 \text{ (x)} = \\
&\qquad \textbf{if } cond(x) \textbf{ then } Nulstock \\
&\qquad \textbf{else } (calc(take(x)), rep(drop(x))) \\
&\textbf{and } take_2(Input) = Input \\
&\textbf{and } drop_2(Input) = Input \\
&\textbf{and } calc_2(Input) = Stock \\
&\textbf{and } cond_2(Input) = \text{bool}
\end{aligned}$$

ただし，関数 F, h, t は $Prog_1$のものと同じ形をしている。ここで，*calc* は *take* で取り出された入力を使って 1 つの *Stock* を計算する関数である。また，*take* と *drop* の間にはつぎの関係が成り立っている。

$$take(x)@drop(x) = x$$

ただし，ここの “=” はリストを集合とみなした場合の等価性を表す。これら 2 つの関数は入出力ともにリストになる時に現れる。

version 3 — *Input* の具体化 —　つぎに入力の構造を明らかにする。ここでは，*Input* は領収書 *Rept* のリストで構成されていることにする。そこで，*Input* を再帰的なデータ構造 *Rlist* に具体化する。この時，入力ドメインは次のようになり出力ドメインは $\mathbf{D}_2^{\mathrm{O}}$ と同じである。

$$\begin{aligned} \mathbf{D}_3^{\mathrm{I}} &= \mathbf{D}_2^{\mathrm{I}}[Input \prec Rlist] \\ &\quad \textbf{where } Rlist = Nulrept \mid (Rept, Rlist) \end{aligned}$$

そして，$Prog_3$をテンプレート 5 と 6 を使って次のように定義する。

$\textbf{fun } take_3(Nulrept) = Nulrept$

$\quad \mid take_3(x, y) \quad = takesub(x, (x, y))$

$\textbf{and } takesub_3(x, Nulrept) = Nulrept$

$\quad \mid takesub_3(x, (y, ys)) \quad =$

$\quad\quad \textbf{if } samegrp(x, y) \textbf{ then } (y, takesub(x, ys))$

$\quad\quad \textbf{else } takesub(x, ys)$

$\textbf{and } drop_3(Nulrept) = Nulrept$

$\quad \mid drop_3(x, y) \quad = dropsub(x, (x, y))$

$\textbf{and } dropsub_3(x, Nulrept) = Nulrept$

$\quad \mid dropsub_3(x, (y, ys)) \quad =$

$\quad\quad \textbf{if } samegrp(x, y) \textbf{ then } dropsub(x, ys)$

$\quad\quad \textbf{else } (y, dropsub(x, ys))$

$\textbf{and } samegrp_3(x, y) \ = \mathrm{bool}$

$\textbf{and } cond_3(Nulrept) = \mathrm{true}$

$\quad \mid cond_3((x, y)) \quad = \mathrm{false}$

$\textbf{and } calc_3(x, Nulrept) = initstock(\mathrm{x})$

$\quad \mid calc_3(x, y) \quad\quad = upd(cal(x), calc(y))$

$\textbf{and } initstock_3(x) = Stock$

$\textbf{and } cal_3(Rept) \quad = Res$

$\textbf{and } upd_3(Res, Stock) = Stock;$

関数 F, h, t, rep は $Prog_2$のものと同じ形をしている。この定義で，*cal* は一つの領収書から得られるデータを中間データ *Res* に変化する関数で，*upd* はこれまで作成した在庫報告にこの中間データを反映させる関数である。また，関数 *take* と *drop* は，それぞれ入力

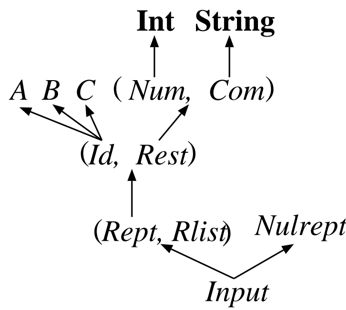

図 4.1: 入力ドメイン

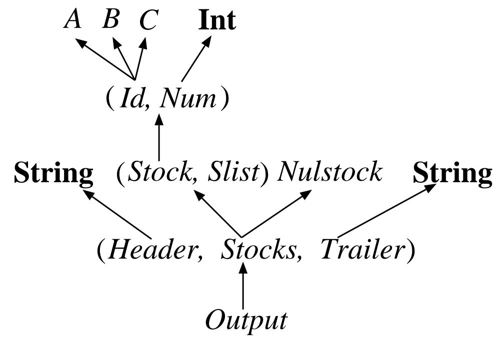

図 4.2: 出力ドメイン

のリストの中でその先頭要素と同じグループに属している要素と属していない要素を取り出す。その実際の処理をしているのが $takesub$ と $dropsub$ である。これらの関数は入出力が同時にリストに具体化されたため，テンプレート 5 と 6 をあわせた形となっている。例えば，$takesub$ の場合，テンプレート 5 中の h が再帰データを構成する部分に相当し，入力データの末尾をとる部分がテンプレート 6 中の pre に相当する。

version 4 — *Rept* の具体化 —　さらに入力の構造を明らかにする。ここでは，$Rept$ には製品名 Id が含まれることにする。そこで，$Rept$ をレコード $(Id, Rest)$ に具体化する。この時，入力ドメインは次のようになり出力ドメインは $\mathbf{D}_3^{\mathrm{O}}$ とおなじである。

$$\mathbf{D}_4^{\mathrm{I}} \;=\; \mathbf{D}_3^{\mathrm{I}}[Rept \prec (Id, Rest)]$$

そして，$Prog_3$ をテンプレート 3 を使って次のように $Prog_4$ に詳細化する。

$$
\begin{aligned}
&\textbf{fun}\ cal_4((Id, Rest)) = make1(getqty(Rest))\\
&\textbf{and}\ make1_4(Num)\ =\ Res\\
&\textbf{and}\ getqty_4(Rest)\ =\ Num\\
&\textbf{and}\ samegrp_4((i1, r1), (i2, r2)) =\\
&\qquad \textbf{if}\ sameid(i1, i2)\ \ \textbf{then}\ \text{true}\\
&\qquad \textbf{else}\ \text{false}\\
&\textbf{and}\ sameid_4(i1, i2) = \text{bool}
\end{aligned}
$$

他の関数は $Prog_3$ のものと同じ形をしている。

さらに，製品名 Id には A, B, C の 3 種類あると具体化する事で，$sameid$ が次のように詳細になり，入力に含まれる製品の種類だけ出庫状況が計算できるプログラムが完成する。

$$\textbf{fun}\ sameid_4(i1, i2) = \textbf{if}\ (i1 = i2)\ \ \textbf{then}\ \text{true}$$
$$\textbf{else}\ \text{false}$$

ここまでのプログラムを用いると，次のように入出力が完全に明らかにならなくても，入出庫数などの処理を記述する前に抽象実行によって，出力の在庫リストの数が正しいかどうか調べる事が可能となる。

$$\begin{aligned} & F((A, Rest), ((B, Rest), ((A, Rest), Nulrept))) \\ = \quad & (Header, (Stock, (Stock, Nulstock)), Triler) \end{aligned}$$

結局，入出力ドメインは最終的には，図 4.1,4.2のようになった。完成したプログラムは付録 A.3.1に示す。

4.2　マイクロフィッシュ問題

マイクロフィッシュ問題は，その仕様が複雑なので形式的な方法なしではプログラムの構成が困難，かつ入出力の対応がつかなく入出力関係から直接プログラムが構成できないという特徴を持つ。この節では，その問題をどのように ISDR 法で構成するかを述べる。

まず，マイクロフィッシュ問題の概要を述べる。そして，次の節で ISDR 法による解法を述べる。

4.2.1　マイクロフィッシュ問題の概略

マイクロフィッシュ問題の実際の仕様は文献 [11] に英語の文章で 2 ページ半に渡って記述してあるが，ここではその概要を述べる。

この問題の主な目的は，図 4.3にあるように,「商品の入出庫情報が書かれたレコードの列を，そのマイクロフィッシュを作るために，ヘッダ，トレイラ，サマリなどを付加したレコードの列に変換する」プログラムを構成することである。

また，出力するマイクロフィッシュは次の規則に従う。

- T1 は、会社名、品目番号、“WAREHOSE”、倉庫番号、“#”、倉庫内のマイクロフィッシュ通し番号からなる。その品目番号はマイクロフィッシュの最初のフレームの品目番号となる。

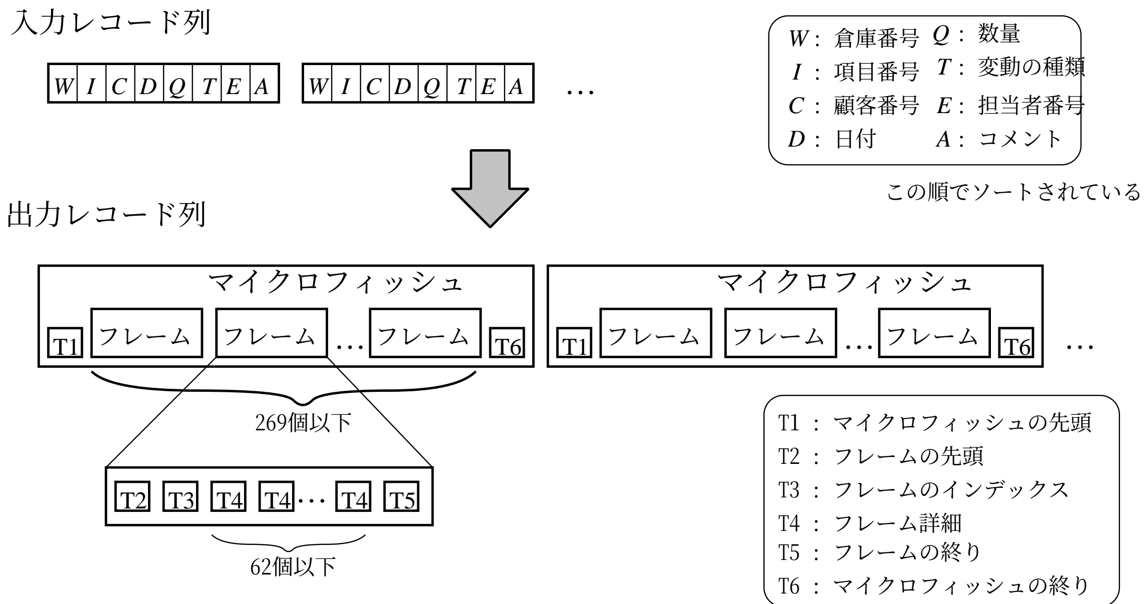

図 4.3: マイクロフィッシュ問題の概略

- マイクロフィッシュの始めの 30 フレーム目までに品目が変われば、そのフレームの前にもう 1 つ T1 を出力する。その T1 の品目番号は 2 番目の品目番号と一致する。
- 異なる倉庫番号に対するフレームは別のマイクロフィッシュに出力する。
- T2 には、品目番号とマイクロフィッシュ内のフレーム通し番号が含まれている。
- 1 つの入力レコードにつき、その備考が空欄なら 1 つの T4 レコードを出力する。このレコードには、C, E, D, Q, T の情報が含まれる。備考が空欄でなければ、これに加えて A の情報が書かれた T4 レコードが出力する。
- 品目ごとに 2 つの空欄 T4 レコードとその倉庫に関する品目の在庫数 (TOTAL NET MOVEMENT) の T4 レコードを出力する。
- 顧客ごとにその人に関する品目の総合変動数 (TOTAL) の T4 レコード 1 つと空欄の T4 レコード 1 つを出力する。
- これら 2 つのサマリーと、そのサマリーを出す直前に処理したレコードに対する T4 列はフレームに跨ってはいけない。

- 1つのフレームの中にはただ1つの品目に関するT4レコードを含める。

4.2.2 ISDR法による解法

この問題のポイントの一つは，入力レコードのグループ分けとマイクロフィッシュを構成するためのフレームの数とは直接関係しない事である。そのため，JSP法を用いてこの問題を解く場合クラッシュが発生してしまう。しかしながら，ISDR法では入力や出力の内部を段階的に具体化すれば良く，ごく自然にプログラムの構成が可能である。[1]

抽象化

まず，目的のプログラムの入力列を *INPUTS*，出力のマイクロフィッシュ列を *FICHES* に抽象化する。この時，入力ドメイン $\mathbf{D}_0^{\mathrm{I}}$と出力ドメイン $\mathbf{D}_0^{\mathrm{O}}$は以下のようになる。

$$\begin{aligned}\mathbf{D}_0^{\mathrm{I}} &= \{\langle INPUTS\rangle, \emptyset\}\\ \mathbf{D}_0^{\mathrm{O}} &= \{\langle FICHES\rangle, \emptyset\}\end{aligned}$$

そして，原始プログラムの主関数 F_0は，以下のように *Input* から *FICHES* への関数となる。

$$\textbf{fun}\quad F_0(INPUTS) \quad = \quad FICHES;$$

version 1

まず，入出力の構造に着目してデータを具体化する。ここでは，*INPUTS* を *INPUT* のリストに具体化し，*FICHES* を *FICHE* のリストに具体化する。

この時，入出力ドメインは次のようになる。

$$\begin{aligned}\mathbf{D}_1^{\mathrm{I}} &= \mathbf{D}_0^{\mathrm{I}}[INPUTS \prec [INPUT]]\\ \mathbf{D}_1^{\mathrm{O}} &= \mathbf{D}_0^{\mathrm{O}}[FICHES \prec [FICHE]]\end{aligned}$$

そして，F_1を次のように定義する。

$$\begin{array}{llcl}\textbf{fun} & F_1([]) & = & []\\ \mid & F_1(x :: xs) & = & \textbf{if}\ \mathrm{bool}\ \textbf{then}\ FICHE :: F(xs)\\ & & & \textbf{else}\ F(xs)\end{array}$$

[1] これについての詳しい議論は5.5.1節を参照のこと

version 2

一種類の倉庫から複数のマイクロフィッシュ($FICHE$) ができるという仕様を用いてプログラムを詳細化する。ここでは，入力ドメイン中の $INPUT$ を次のように，倉庫番号 W を含むレコードに具体化する。

$$\mathbf{D}^{\mathrm{I}}_2 \quad = \quad \mathbf{D}^{\mathrm{I}}_1[INPUT \prec (W, OW)]$$

すると，プログラムは次のように詳細化できる。

$$
\begin{aligned}
&\textbf{fun } F_2(xs \textbf{ as } ((w, _) :: xss) = \textbf{let}\\
&\qquad\qquad \textbf{val } (ws, wrest) = div_whouse(w, xs, [])\\
&\qquad \textbf{in}\\
&\qquad\qquad makeFiches(ws) :: F(wrest)\\
&\qquad \textbf{end}\\
&\textbf{and } makeFiches_2([]) \qquad\qquad = []\\
&\quad |\ makeFiches_2(w :: wss) = \textbf{if } \mathrm{bool} \textbf{ then } FICHE :: makeFiches(wss)\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } makeFiches(wss)\\
&\textbf{and } div_whouse_2(w, [], h) = (rev\ h, [])\\
&\quad |\ div_whouse_2((w, xsasx :: xss, h)) =\\
&\qquad\quad \textbf{if } belong_w(w, x) \textbf{ then } div_whouse(w, xss, x :: h)\\
&\qquad\quad \textbf{else } (rev\ h, xs)\\
&\textbf{and } belong_w_2(w, x) = \mathrm{bool};
\end{aligned}
$$

このプログラム中で，$makeFiches$ は，一種類の倉庫のレコード列から複数のマイクロフィッシュ列をつくる関数で，div_whouse は，第 2 引数のレコード列を第 1 引数の倉庫に属するものとそれ以外のレコード列を分ける関数である。また，$belong_w$ は，第 2 引数の入力レコードが第 1 引数の倉庫に属するかどうか調べる関数である。

version 3

つぎに，倉庫番号を整数に具体化する。

$$\mathbf{D}^{\mathrm{I}}_3 \quad = \quad \mathbf{D}^{\mathrm{I}}_2[W \prec \mathbf{int}]$$

すると，関数 $belong_w$ が次のように詳細化できる。

$$
\begin{array}{llcl}
\textbf{fun} & belong_w_3(w,(wid,_)) & = & \textbf{if}\ (w = wid)\ \textbf{then}\ \mathrm{true} \\
 & & & \textbf{else}\ \mathrm{false};
\end{array}
$$

version 4

次に，一つのマイクロフィッシュはフレーム *FIBODY* の列からなり一種類の品目から複数の *FIBODY* ができ，マイクロフィッシュの先頭や末尾や途中には何らかの情報 (*FIBODY*) がつくという仕様を利用して詳細化を行う。具体的には，*OW* を品目番号 *I* を含むレコードに具体化し，*FICHE* はフレーム *FIBODY* を含むレコードに具体化する。

$$
\begin{array}{rcl}
\mathbf{D}_4^{\mathrm{I}} & = & \mathbf{D}_3^{\mathrm{I}}[OW \prec (I, OI)] \\
\mathbf{D}_4^{\mathrm{O}} & = & \mathbf{D}_3^{\mathrm{O}}[FICHE \prec [FIBODY]]
\end{array}
$$

すると，プログラムは次のように詳細化できる。

$$
\begin{array}{llcl}
\textbf{fun} & makeFiches_4([]) & = & [] \\
\mid & makeFiches_4(xs) & = & fillF(makeFbody(xs))
\end{array}
$$

$$
\begin{array}{ll}
\textbf{and} & makeFbody_4([]) = [] \\
\mid & makeFbody_4(xsas(w,(i,_)) :: xss) = \\
 & \textbf{let} \\
 & \quad \textbf{val}\ (is, irest) = div_item(i, xs, []) \\
 & \textbf{in} \\
 & \quad makeFbodyI(is) :: makeFbody(irest) \\
 & \textbf{end}
\end{array}
$$

$$
\begin{array}{llcl}
\textbf{and} & div_item_4(id, [], h) & = & (rev\ h, []) \\
\mid & div_item_4(id, x :: xs, h) & = & \textbf{if}\ belong_i(id, x)\ \textbf{then}\ div_item(id, xs, x :: h) \\
 & & & \textbf{else}\ (rev\ h, x :: xs)
\end{array}
$$

$$
\begin{array}{llcl}
\textbf{and} & belong_i_4(i, x) & = & \mathrm{bool}
\end{array}
$$

$$
\begin{array}{llcl}
\textbf{and} & makeFbodyI_4(x :: xs) & = & \textbf{if}\ \mathrm{bool}\ \textbf{then}\ FIBODY :: makeFbodyI(xs) \\
 & & & \textbf{else}\ makeFbodyI(xs)
\end{array}
$$

$$
\begin{array}{llcl}
\textbf{and} & fillF_4(xss) & = & fillFin(xss, [])
\end{array}
$$

$$
\begin{array}{lll}
\textbf{and} \quad \mathit{fillFin}_4([[]],[]) & = & [[]] \\
\mid \quad \mathit{fillFin}_4([[]],zs) & = & [rev\ (\mathit{FIBODY} :: zs)] \\
\mid \quad \mathit{fillFin}_4([] :: xs, zs) & = & \textbf{if}\ \text{bool}\ \textbf{then}\ \mathit{fillFin}(xss, [\mathit{FIBODY}] :: zs) \\
 & & \textbf{else}\ \mathit{fillFin}(xs, zs) \\
\mid \quad \mathit{fillFin}_4(xss, []) & = & \mathit{fillFin}(xss, [\mathit{FIBODY}]) \\
\mid \quad \mathit{fillFin}_4(xssas(x :: xs) :: ys, zs) & = & \\
\qquad \textbf{if}\ \text{bool}\ \textbf{then}\ (rev\ (\mathit{FIBODY} :: zs)) :: \mathit{fillFin}(xss, []) & & \\
\qquad \textbf{else}\ \mathit{fillFin}(xs :: ys, x :: zs) & &
\end{array}
$$

このプログラムでは，*makeFbody* によって *FIBODY* の列をつくり、その後に *fillF* によってマイクロフィッシュのヘッダ、フッタなどを追加しながら、マイクロフィッシュを形成する。そして，*makeFbody* で作られた 1 つの *FIBODY* の列から複数のマイクロフィッシュが形成される。

version 5

つぎに，品目番号を具体値に具体化する。

$$\mathbf{D}_5^{\mathrm{I}} \quad = \quad \mathbf{D}_4^{\mathrm{I}}[I \prec \mathbf{int}]$$

すると，次のように *belong_i* が詳細化できる。

$$
\begin{array}{lll}
\textbf{fun} \quad belong_i_5(i, (_, (id, _))) & = & \textbf{if}\ (i = id)\ \textbf{then}\ \text{true} \\
 & & \textbf{else}\ \text{false}
\end{array}
$$

version 6

つぎの仕様を使ってプログラムを詳細化する。すなわち，一種類の品目から複数のフレームができ，296 個 (*FRAMEMAX*) のフレームで 1 つのマイクロフィッシュができ，*FICHE* の最初にはヘッダ T1 が最後にはトレイラー T6 がつく。また，*FICHE* の始め 30 個フレーム以内に品目が変わればその前にもヘッダがつく。具体的には，*FIBODY* をフレームのヘッダ *T*1，フッタ *T*6 と中身 *FRAME* に具体化する。

$$\mathbf{D}_6^{\mathrm{O}} \quad = \quad \mathbf{D}_6^{\mathrm{O}}[\mathit{FIBODY} \prec \{T1, \mathit{FRAME}, T6\}]$$

すると，プログラムは次のように詳細になる。

$$
\begin{array}{llcl}
\textbf{fun} & \mathit{makeFbodyI}_6(x :: xs) & = & \textbf{if}\ \text{bool}\ \textbf{then}\ \mathit{FRAME} :: \mathit{makeFbodyI}(xs) \\
 & & & \textbf{else}\ \mathit{makeFbodyI}(xs)
\end{array}
$$

$$
\textbf{fun}\quad \mathit{fillF}_6(xs) \quad = \quad \mathit{fillFin}(xs, [], 0, 1, \text{false})
$$

$$
\begin{array}{lll}
\textbf{and} & \mathit{fillFin}_6([[]], [], 0, _, _) & = \quad [[]] \\
\mid & \mathit{fillFin}_6([[]], zs, _, _, _) & = \quad [rev\ (\mathit{MFtail}() :: zs)] \\
\mid & \mathit{fillFin}_6([] :: (xssas(x :: _) :: _), zs, n, m, b) = & \\
 & \quad \textbf{if}\ (n < 30)\ \textbf{andalso}\ (b = false)\ \textbf{then} & \\
 & \quad\quad \mathit{fillFin}(xss, \mathit{MFhead}(x, m) :: zs, n, m, \text{true}) & \\
 & \quad \textbf{else}\ \mathit{fillFin}(xss, zs, n, m, b) & \\
\mid & \mathit{fillFin}_6(xssas((x :: _) :: _), [], _, m, b) & = \quad \mathit{fillFin}(xss, [\mathit{MFhead}(x, m)], 0, m, b) \\
\mid & \mathit{fillFin}_6(xssas(x :: xs) :: ys, zs, n, m, b) & = \\
 & \quad \textbf{if}\ (n > FRAMEMAX)\ \textbf{then} & \\
 & \quad\quad (rev\ (\mathit{MFtail}() :: zs)) :: \mathit{fillFin}(xss, [], 0, m + 1, \text{false}) & \\
 & \quad \textbf{else}\ \mathit{fillFin}(xs :: ys, x :: zs, n + 1, m, b) &
\end{array}
$$

$$
\textbf{and}\quad \mathit{MFhead}_6(_, _) \quad = \quad T1
$$

$$
\textbf{and}\quad \mathit{MFtail}_6() \quad = \quad T6;
$$

version 7

次に，フレーム (FRAME) はいくつかのフレームの中身 FRBODY の列からなり，フレームの先頭は 2 つのヘッダがつき、末尾には 1 つのトレイラがつき，一つのレコードに対し、複数の FRBODY が出力されるという仕様を用いてプログラムを詳細化する。

データ及び，プログラムは次のように詳細化される。

$$
\mathbf{D}_7^{\mathrm{O}} \quad = \quad \mathbf{D}_6^{\mathrm{O}}[\mathit{FRAME} \prec [\mathit{FRBODY}]]
$$

$$
\textbf{fun}\quad \mathit{makeFbodyI}_7(xs) \quad = \quad \mathit{makeFrameI}(xs, [])
$$

$$
\begin{array}{lll}
\textbf{and} & makeFrameI_7([],[]) & = [[]] \\
| & makeFrameI_7([],zs) & = [rev\ (FRBODY :: zs)] \\
| & makeFrameI_7(xs,[]) & = makeFrameI(xss,[FRBODY]) \\
| & makeFrameI_7(x :: xs, zs) & = \\
& \quad \textbf{if}\ \mathrm{bool}\ \textbf{then}\ (rev\ (FRBODY :: zs)) :: makeFrameI(x :: xs, []) & \\
& \quad \textbf{else if}\ \mathrm{bool}\ \textbf{then}\ makeFrameI(x :: xs, FRBODY :: zs) & \\
& \quad \textbf{else}\ makeFrameI(x, FRBODY :: zs) &
\end{array}
$$

version 8

さらに，つぎの仕様を用いてプログラムを詳細化する。すなわち，複数のレコードから一つのフレームが出来上がり，1 つのレコードから複数の $T4$ が出来上がる。また，品目ごとに 3 つのサマリー $T4$ を出力し，$T4$ が 62 個以下で 1 つのフレームが出来上がる。ここで，サマリーとその前に出力したいくつかの $T4$ はフレームに跨らないようにする。

データ及び，プログラムは次のように詳細化される。

$$
\mathbf{D}_8^{\mathrm{O}} = \mathbf{D}_7^{\mathrm{O}}[FRBODY \prec \{T2, T3, T4, T5\}]
$$

$$
\begin{array}{lll}
\textbf{fun} & makeFbodyI_8(xs) & = fillFR(makeT4s(xs, smyIinit())) \\
\textbf{and} & makeT4s_8([], s) & = s \\
| & makeT4s_8(x :: xs, s) & = \textbf{if}\ \mathrm{bool}\ \textbf{then}\ T4 :: makeT4s(x :: xs, s) \\
& & \quad \textbf{else}\ T4 :: makeT4s(xs, addsmyI(s, x)) \\
\textbf{and} & fillFR_8(xs as((_,(i,_)) :: _)) & = fillFRin(xs, [], 0, i, 1) \\
\textbf{and} & FRheads_8(_,_) & = [T2, T3] \\
\textbf{and} & FRtail_8() & = T5 \\
\textbf{and} & addsmyI_8(_,_) & = [T4, T4, T4] \\
\textbf{and} & smyIinit_8() & = [T4, T4, T4] \\
\textbf{and} & div_T4s_8([], zs) & = (zs, []) \\
| & div_T4s_8(x :: xs, zs) & = \textbf{if}\ \mathrm{bool}\ \textbf{then}\ div_T4s(xs, x :: zs) \\
& & \quad \textbf{else}\ (zs, x :: xs);
\end{array}
$$

and $\mathit{fillFRin}_8([], [], 0, _, _) \quad = \quad [[]]$

$\mid \quad \mathit{fillFRin}_8([], zs, _, _, _) \quad = \quad [rev\ (\mathit{FRtail}() :: zs)]$

$\mid \quad \mathit{fillFRin}_8(xs, [], 0, i, m) \quad = \quad \mathit{fillFRin}(xs, \mathit{FRheads}(i, m), 0, i, m)$

$\mid \quad \mathit{fillFRin}_8(xs, zs, n, i, m) \quad =$

let

  val $(t4s, rest) = div_T4s(xs, [])$

in

  if $(n + length(t4s) > T4MAX)$ **then**

   $(rev\ (\mathit{FRtail}() :: zs)) :: \mathit{fillFRin}(xs, [], 0, i, m + 1)$

  else

   $\mathit{fillFRin}(rest, t4s@zs, 0, i, m + 1)$

end

このプログラムでは，一種類の品目から $FRAME$ の列を ($FRBODY$ の列の列) つくっている。実際は、$makeT4s$ で $T4$ の列を作ったのち、fillFR でそれらを 62 個以下にまとめてフレームの列を構成している。

version 9

つぎに，顧客ごとに 2 行のサマリーを出力する。そこで，OI を顧客 C のレコードに具体化する。

$$\mathbf{D}_9^{\mathrm{I}} \quad = \quad \mathbf{D}_8^{\mathrm{I}}[OI \prec (C, OC)]$$

すると次のように $makeT4s$ が詳細化できる。

fun $makeT4s_9([], s) \quad = \quad s$

$\mid \quad makeT4s_9(xsas((_,(_,(c,_)\,)) ::_), s) \quad =$

let

  val $(cs, rest) = div_cust(c, xs, [])$

  and $(t4c, ss) = makeT4c(cs, smyCinit())$

in

  $t4c@makeT4s(rest, addsmyI2(s, ss))$

end

$$
\begin{array}{llcl}
\textbf{and} & div_cust_9(c, [], ys) & = & (rev\ ys, []) \\
\mid & div_cust_9(c, x :: xs, ys) & = & \textbf{if}\ belong_c\ \textbf{then}\ div_item(c, xs, x :: ys) \\
 & & & \textbf{else}\ (rev\ ys, x :: xs)
\end{array}
$$

$$
\textbf{and} \quad belong_c_9(c, x) \quad = \quad \mathrm{bool}
$$

$$
\begin{array}{llcl}
\textbf{and} & makeT4c_9([], sc) & = & sc \\
\mid & makeT4c_9(x :: xs, sc) & = & rec2T4s(x)@makeT4c(xs, addsmyC(sc, x))
\end{array}
$$

$$
\begin{array}{llcl}
\textbf{and} & rec2T4s_9(_) & = & \textbf{if}\ \mathrm{bool}\ \mathrm{then}\ T4 :: rec2T4s(x) \\
 & & & \textbf{else}\ [T4]
\end{array}
$$

$$
\textbf{and} \quad addsmyC_9(_, _) \quad = \quad [T4, T4]
$$

$$
\textbf{and} \quad smyCinit_9() \quad = \quad [T4, T4]
$$

$$
\textbf{and} \quad addsmyI2_9(_, _) \quad = \quad [T4, T4, T4];
$$

version 10

つぎに，品目番号を具体値に具体化する。

$$
\mathbf{D}^{\mathrm{I}}_1 0 \quad = \quad \mathbf{D}^{\mathrm{I}}_9[C \prec \mathbf{int}]
$$

すると，

$$
\begin{array}{llcl}
\textbf{fun} & belong_c_{10}(c, (_, (_, _(ci, _)))) & = & \textbf{if}\ (c = ci)\ \textbf{then}\ \mathrm{true} \\
 & & & \textbf{else}\ \mathrm{false}
\end{array}
$$

version 11

つぎに，各レコードごとに、備考 (A) が空欄であれば 1 つの $T4$ を出力し、そうでなければ 2 つの $T4$ を出力するように詳細化する。

$$
\mathbf{D}^{\mathrm{I}}_1 1 \quad = \quad \mathbf{D}^{\mathrm{I}}_1 0[OC \prec (A, D)]
$$

すると version 9 の $rec2T4s$ が次のように詳細化できる。

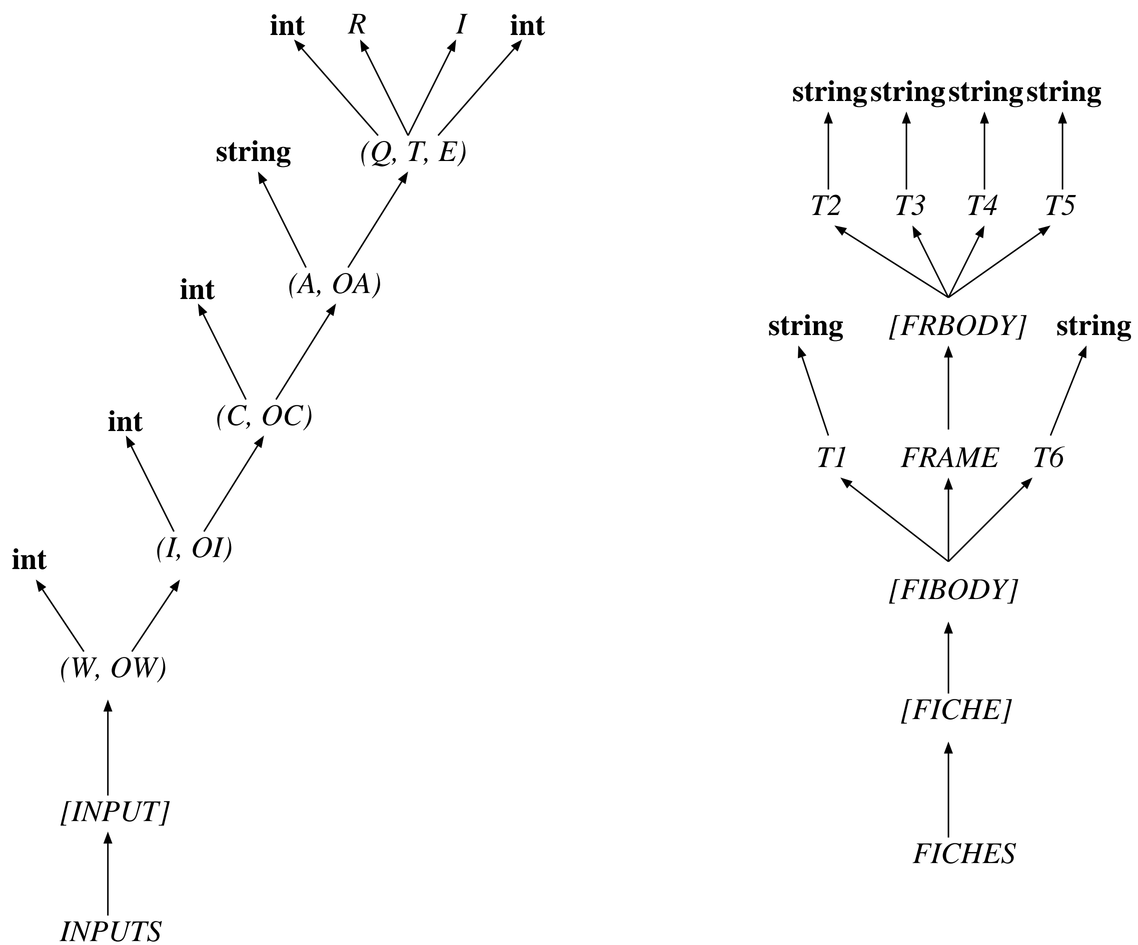

図 4.4: 最終的なドメイン

$$\textbf{fun} \quad rec2T4s_{11}((,(,(,(a,))))) \;=\; \textbf{if}\ nulA(a)\ \textbf{then}\ [T4]\ \textbf{else}\ [T4, T4]$$

$$\textbf{and} \quad nulA(a) \;=\; \text{bool}$$

さらに，備考, $T1$ から $T6$ を文字列に具体化し，品目の変動数を整数に具体化すればプログラムは完成する。

最終的な入出力ドメインは図 4.4の様になった。

第 5 章

議論

5.1 ISDR 法の特徴

この節では ISDR 法が他の方法論にない特徴を述べる。

5.1.1 詳細化の定義

ISDR 法では，他の探検的プログラミングとは異なり数学的に定義された詳細化の観点でしかプログラムを拡張できない。

例えば，探検的プログラミングの一つである Balzer らの方法 [12] では，属性に関する変更として以下のようなものを許している。

- 一つの値が複数の値になる。
- 複数の値が一つの値になる。
- 定義領域や値域が変化する。

しかし，ISDR 法ではこの内の一つの値が複数の値に変化することしか許していない。

また，従来の方法は，進化の前後のプログラムの関係を厳密に付けていない。そのため，進化の前後によってプログラムの振る舞いや意味が変わってしまい，新しく作成したプログラムの把握が難しくなる。それに対して，ISDR 法では進化の前後で詳細化の関係が成り立つことを強制している。

詳細化による制限によって，データの構造と関数の合成との対応をある程度つけられるようになる。それによって，データの抽象度の変化が無秩序にプログラム全体に波及する

ことを防ぎ，開発者が注目すべきスコープが狭くなり，大きなプログラムの詳細化であっても制御可能になる。これは，大規模なソフトウェア開発を段階的詳細化する場合，非常に重要なポイントなる。

もちろん，このシステムの拡張性の制限は，大幅なバックトラックによる再設計を強いる可能性を含んでいる。しかし，そのような場合も，バックトラックのコストを削減することが可能である。

5.1.2 抽象実行

抽象解釈とは，抽象的なデータの集合とその上で定義された演算を使ってプログラムを解釈することをいう。一般的に，この技法は，すでに完成したプログラムを抽象化し実行することで，プログラムの不変的な性質や部分情報をとりだすために用いられる。ISDR 法では，これとは逆に，抽象的なものから作り始め段階的にプログラムを完成させる過程で抽象解釈を利用する。

抽象解釈を利用する最大の利点は，それによって詳細化途中の未完成なプログラムの実行が可能になることである。従来の段階的詳細化法では，詳細化途中の生成物は単に設計のためのドキュメントにすぎず，その実行はまったく考慮されていなかった。しかしながら，途中段階の生成物の実行は，特に複雑なシステムを設計する場合に重要になってくる。なぜならば，そのようなシステムの場合，そのプログラムを作成する前に仕様を完全にすることは困難で，段階的にテストしながら詳細化する必要があるからである。このような場合の設計法として，これまでも発展的なプロトタピイングの手法が提案されてきたが，これまでのものは技術的，原理的な裏付けがなされていなかった。ISDR 法は，理論的に裏付けされた発展的な開発手法の一つである。

5.2 ISDR 法の正当性

ISDR 法によって最終的に仕様を満たすプログラムが構成可能であり，途中段階のプログラムは最終的なプログラムの一部分になっていることを以下のように示す。

さまざまな具体値の集合を用いてかかれた仕様はそれらを表す抽象値間の関数の集合とみなす事ができる。仕様中のすべての関数の入出力データ集合をそれぞれドメインで表現できるならば，仕様を満たすプログラムはドメイン間の関数の合成で表現できる。そして，ISDR

法で構成した最終的なプログラムの解釈がこのプログラムの振舞いと一致すればISDR法で仕様を満たすプログラムが構成可能である事がいえる。

ここで，仕様中の各関数で用いられているすべてのデータからドメインを次のように構成できる。すなわち，任意の関数について，そこで用いられているデータ集合が含意の関係で表現でない場合，例えばあるデータ集合rとsの一部が重なっている場合，$r \cap \bar{s}$, $r \cap s$, $\bar{r} \cap s$の3つの部分集合を用いて仕様を定義し直す。もし，定義し直せない場合は，それぞれのデータに関して関数が存在するとみなす。これを全ての関数に使われてる集合についてそれぞれ含意の関係のみになるまで繰り返す。そうすれば，仕様に現れない集合に対する抽象値を適当に追加してやることですべてのデータについてドメインを構成できる。

この時，ISDR法の各詳細化段階で，このドメイン間の関数の一部を実装する事により最終的に構成したプログラムは必ず仕様を満たす。

ここで，抽象データを具体化していく方法は幾通りもあり，必ずしも一度で最終的に仕様を満たすプログラムが定義できるとは限らない。実際のプログラムの構成は，失敗したら適当なバージョンまで戻り再びやり直すことを繰り返しながら行う。

5.3 ソフトウェアプロセスモデル

ISDR法は，方法論としてプロセスで用いる用語(詳細化など)を定義し，方法としてルール，ガイドライン，テクニックなどを述べている。この節では，実際に方法論を適用する場合そのプロセスとしてどのようなものがふさわしいか議論する。具体的には，現在有効であると認知されているプロトタイプ指向パラダイム，探検的プログラミングパラダイム，操作的パラダイムの3つの代表的なモデルについて述べ，ISDR法にふさわしいプロセスを議論する。

5.3.1 プロトタイピング指向パラダイム

プロトタイピングの研究は現在も盛んに行われており，その用語自身の定義さえ一般的な解釈はまだ存在しないが，[13]では次のように定義している。

”A prototype is an easily modifiable and extensible working model of proposed system, not necessarily representative of a complete system, which provides

later users of the application with a physical representation of key parts of the system before implementation”

しかし，多くのプロトタイプの定義には，早く安く作れるものという概念が含まれている。

[14] では，様々なプロトタイピングを次の 3 つに分類している。

- 予備調査的プロトタイピング (exploratory prototyping)
- 実験的プロトタイピング (experimental prototyping)
- 進化的プロトタイピング (evolutionary prototyping)

また，[2] では，プロトタイプを次の 4 つに分類している。

- 完全なプロトタイプ (complete prototypes)
- 不完全なプロトタイプ (incomplete prototypes)
- 使い捨てプロトタイプ (throwaway prototypes)
- 再利用可能プロトタイプ (reusable prototypes)

それぞれの意味は次のようになる。

予備調査的プロトタイピング　このプロトタイピングは，現実の例をユーザが実際に試してみることで，要求の定義をできるだけ完全にするための目的で行われる。すなわち，エンジニアとユーザ間のギャップを無くすために行われる。また，これは，与えられた問題に対してどのような解法がよいかを開発者が考えることを可能にし，提案されたシステムが与えられた環境で実現可能かどうかを明らかにする。

このようなプロトタイプはできるだけ早い段階で作られ，テストされる。そのためプロトタイプの実装の質は軽視され，実際の例に基づいて素早くテストしたい機能だけが盛り込まれる。

実験的プロトタイピング　このプロトタイピングは，システムアーキテクチャがもつコンポーネントから正確に仕様を作り出すための目的で行われる。このようなプロトタイプはシステムを分割する初期の段階で作られ，設計したコンポーネント間のインタラクションのシュミレーションのために使われる。そして，プロトタイプは，実際の例に基づいて個々のコンポーネントのインターフェースは適当か，システムアーキテクチャの拡張性は十分か実験的に調べられる。この場合もその実装の質は重要でない。

予備調査的プロトタイピングと決定的に違うところはこの作成にユーザが関与しないところである。このプロトタイプはシステムとコンポーネントの設計を補助するために作成される。

進化的プロトタイピング　このプロトタイピングは，段階的にシステムを開発するために行われる。つまり，プロトタイプは開発の最初の段階からユーザの要求を明らかにするために開発さる。そして，ユーザの新しい要求があれば，それは以前のプロトタイプに統合される。新しい要求の統合を繰り返すことでより完全な要求の定義をおこなう。このアプローチではプロトタイプと製品との違いはほとんどない。また，この方法ではシステムの拡張性が必要であり，ソフトウェアアーキテクチャの再設計であっても，シンプルで経済的に実行可能な努力で行えるべきである。しかし，この点を実現した研究はまだ存在しない。実際には，段階的に開発されたシステムの完全な再設計はコストがかかりすぎてできないのが現状である。

進化的プロトタイピングはシュミレーションのためのプロトタイプを必要とせず，作成されたプロトタイプは使い捨てではなく最終的には製品にまで推敲される。

プロトタイプ　完全なプロトタイプは，提案されたシステムの重要な機能は完全であり，最終的にできる製品に組み込まれるようなものである。しかし，このようなプロトタイプを作ることはほとんどない。それに対して，不完全なプロトタイプは提案されたシステムのある側面について有用性や実現可能性の調査のためにつくられる。使い捨てプロトタイプは，目的のシステムの実装に使わないもので，再利用可能プロトタイプは，それが重要な部分で十分な質を持っていて目的のシステムの統合されるものである。

プロトタイピング指向の開発工程は，逐次的なものとさほど変わらない。工程の中に 2 つ以上のプロトタイプの作成と評価のための反復が追加されただけである。これにより，各フェーズ間にオーバーラップが発生する。例えば，要求の解析と定義は完全に重なりあい，設計，実装，テストの工程の多くが融合される。すなわち，このアプローチでは個々の開発工程の境は厳密でない。(図 5.1[2])

このアプローチのもっとも重要な点は仕様や設計の間違いをおかすリスクの削減しすることである。また，プロトタイプによって完成するシステムの質を保証できる。

文献 [2] は，プロトタイプ指向のアプローチの利点に次の項目を挙げている。

- 開発者とクライアント間のギャップを埋める。

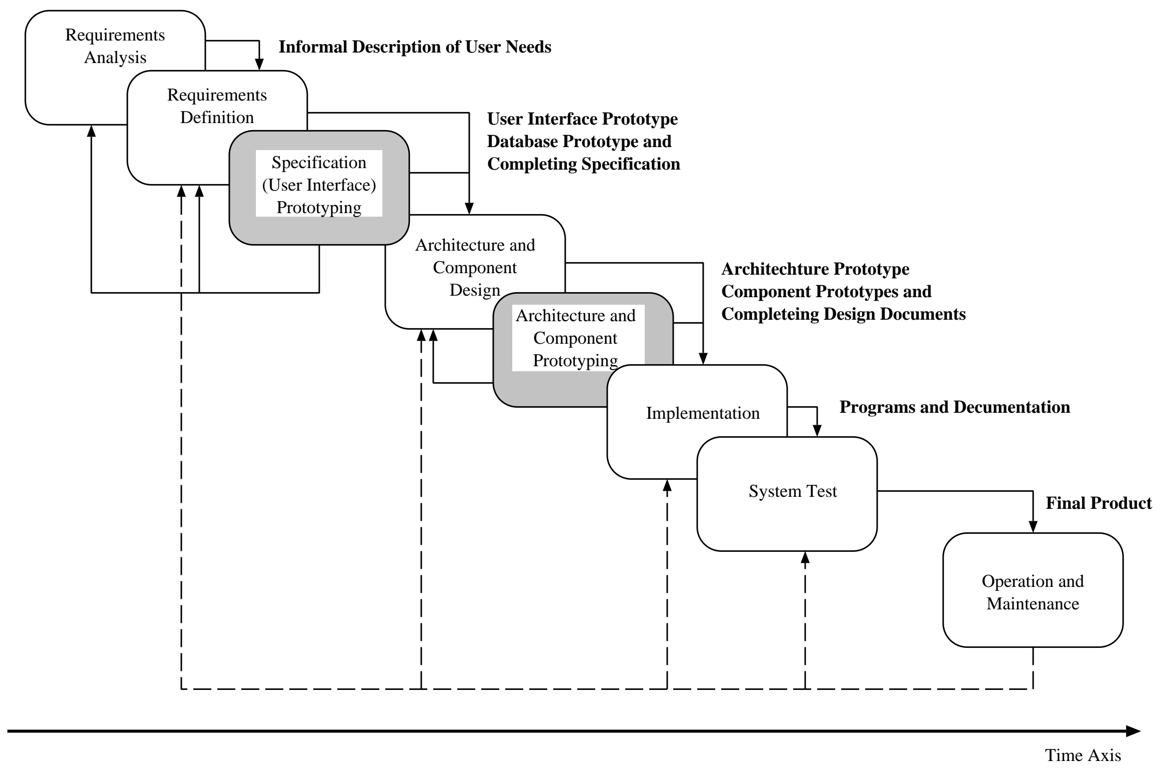

図 5.1: The prototyping-oriented software life cycle

- クライアントは何が実行可能かを開発の早い段階から学ぶことができる。
- 中間段階の実行により間違いを見つける可能性が増える。その結果，再設計のためのコストが減る。
- 反復の工程が組み込まれ，より現実と一致している。
- この開発は，機能の妥当性，ユーザフレンドリー，構造化，変更性，拡張性，完全性や信頼性の質を増加させる。
- このアプローチはテストや保守のコストを削減する。
- プロトタイピングはソフトウェアプロジェクトのスケジューリングを容易にする。なぜなら，改良された要求定義は，その実装やテストの費用をより正確に予想できるようにする。
- 実験的に調べられたモデルは不確定な仮定の数を減らすので，結果としてプロジェクトの失敗のリスクを減らす。

そして，欠点として次の項目を挙げている。

- 開発の工程が発散してしまう。ユーザは必要でないものまで要求してしまう。
- ユーザが要求分析の最後に動いているシステムを見たとき，なぜ設計や実装が必要なのか説明するのが難しくなる。
- 質のよくないプロトタイプをソフトウェアエンジニアリングの原則に反して，そのまま製品として使ってしまう危険がある。
- メインフレームのアプリケーションをパソコン上のツールを使って予備調査的なプロトタイプをつくるのは危険である。なぜなら，そのようなプロトタイプの機能性やユーザインターフェースは簡単に最終的なアプリケーションのそれを越えてしまうからである。

このパラダイムに ISDR 法の原理を応用するなら，進化的プロトタイピングと再利用可能プロトタイプとして応用可能である。ただし，プロトタイプのテストは一般に具体例を用いて行うが，ISDR 法で定義するプログラムは抽象値を扱い必ず値を返すので，実行可能性 (feasible) などの正確なテストができない。すなわち，具体値を用いると計算が不可能な場合も，ISDR 法のプログラムは値を返してしまう。そのような場合，問題となる具体値の計算を行おうとしたところで詳細化ができなくなる。このようなリスクを避けるために，進化のためのプロトタイプとは別に具体値を用いて実験のためのプロトタイプを作り，

それらの間の整合性を取りながら開発を進め，最終的にはそれまでのプロトタイプを合成するプロセスを追加することが考えられる。

進化的プロトタイピングのアプローチではシステムの拡張性が必要となる。しかし，ISDR法で用意している拡張は詳細化のみであり，これに外れた拡張を行うとその解釈の正当性が保証できなくなる。拡張を限定する利点には，解釈を含む原理が明解で分かりやすくなることと，拡張の方針を立てやすくなりそれをサポートするツールや環境を作りやすくなることが挙げられる。また，拡張したプログラムの全体像は任意の拡張を施す場合よりも把握しやすいので，アーキテクチャを再設計する際でも再利用がしやすい。欠点には，一貫性のない要求の変更がある場合，バックトラックが必要になりそのコストが高くつく可能性があることである。すなわち，要求が一貫性なく変化するような領域にはISDR法を進化的プロトタイプとして使用するのには向いていない。

5.3.2 探検的パラダイム

探検的プログラミングは，ほとんど開発の経験がないアプリケーションを開発するために考えられた。この手法によって，開発者は複数の要求やその実現可能性，技術的な問題解決の質などが検討できるからである。そして，この手法では一つ以上の解決案を実装，評価することによって，経験の不足からくるリスクを減らすことができる。しかし，適切な解決案が見出されない場合，要求仕様を変更するかプロジェクトの目的そのものを再検討する必要がある。このような状況はしばしば研究領域でおこり，特に探検的プログラミングの重要性は知識ベースのシステムの開発で重要視されてきた。

探検的プログラミングによるソフトウェア開発の工程は，要求分析まで従来と同じだが，その後の要求定義，設計，実装した後はそれを評価し要求の定義に反映させ進化させるための反復が付け加わっている。(図 5.2[2]) この反復は要求定義，設計，実装のすべてが完全になるまで繰り返される。

このアプローチには進化的なものと実験的なものが存在する。前者は，あるソフトウェアシステムを繰り返し成長，詳細化させ最終的なアプリケーションを開発する方法であり，後者は，要求，設計，実装戦略を練るために途中段階にプログラミングを行う方法である。すなわち，後者においての反復の最後は従来の逐次的な開発工程と一致する。進化的なアプローチの方が，再利用の点から有利であるが実際には再利用させるためのコストが高くついてしまう。それは，段階的に成長させるシステムを設計，実装する場合，単にコード

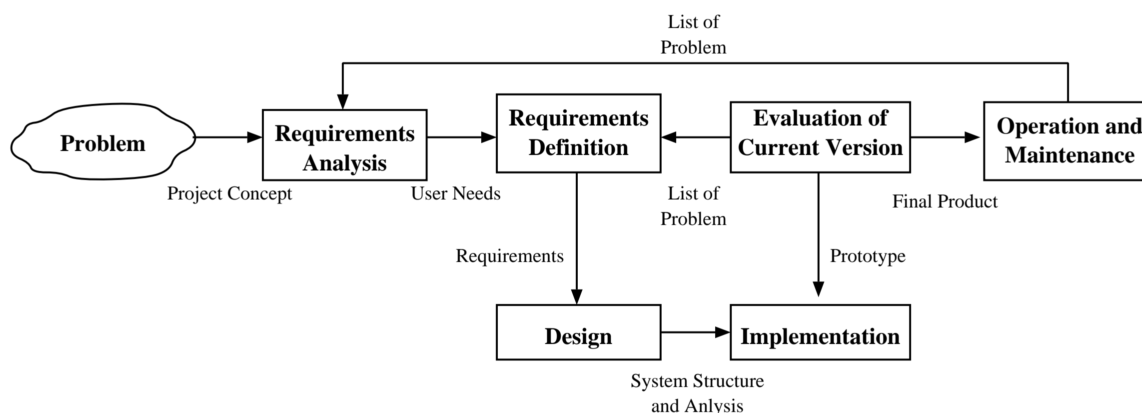

図 5.2: Software development using the exploratory programming paradigm

の一部が正しいどうかをテストする以上の注意を払わなければならないからである。そのため，これらの 2 つのアプローチの選択は，システムを単に実験したいためだけに実装するのかや，あるいは使用するプログラミング言語やツールの質に依存する。

探検的プログラミングはプロトタイピングのパラダイムと非常に似ている。この 2 つの違いは，プロトタイピングで開発するアプリケーションが実行可能性のリスクが低くツールや環境が揃っているのに対して，探検的プログラミングの場合は，技術的に実装可能性のリスクが高くこれまでに設計したことのないような複雑なアプリケーションに適用される。これらのパラダイムは相補的であり，これらの融合によってそれぞれの欠点を補うことが可能である。

文献 [2] は，探検的プログラミングの利点に次の項目を挙げている。

- 要求定義を，妥当で実行可能な解法ができるように改良することができる。
- オブジェクト指向のプログラミングなど他の技術との融合が可能である。
- しばしば，従来の逐次的なアプローチより質がよくなる。なぜなら，いくつかの問題解決案を検討して一番よいものを選択可能であるからである。
- プロトタイピングと相補的である。

そして，欠点として次の項目を挙げている。

- 工程が厳密に分かれていないので，プロジェクトのプランニングや制御が難しい。
- ハッキングに頼った開発に陥りやすい。

- 開発途中のシステムは変更の自由度が高すぎるため，大きなチームでの開発に向いていない。
- アルゴリズムを含む実装を実験のために繰り返し行わなければならないので，大変コストがかかる。

ISDR 法を設計，実装の部分にだけ用いることで，パラダイムとの融合が可能である。しかし，この場合でもプロトタイピングへの応用と同様，拡張性を詳細化のみに制限しているので，経験の不足した不確定要素の多いアプリケーション領域ではうまく開発が進まない可能性がある。その代わり大きなシステムでもその変更は制御可能でチームによる開発も考慮可能である。

また，出来上がったプログラムは抽象解釈のオーバヘッドのために効率や質がよくない。これは，効率が問題となる領域への適用を困難にする。その代償として，実装のコストは低く押えることが可能である。

5.3.3 操作的パラダイム

このパラダイムは要求の定義をそれ以上の工程に進む前に完全にするための方法である。ここで作成する仕様は，完全に設計と切り離される。また，この仕様は実行可能な仕様記述言語を用いることでプロトタイピングの場合と同様実行可能である。

プロトタイピングの場合の仕様は，外部の振る舞いが重要な役割を果たす。これに対して，操作的プログラミングは内部の構造のみが定義され評価される。つまり，操作的な仕様には，通常外部の振る舞いを観察してもその厳密なユーザインターフェースは記述しない。この結果，その仕様は実装に依存しなくなる。そのため，このアプローチの応用領域は主に分散リアルタイムシステムの分野である。

この方法で仕様が完成したらプログラム変換によって実装する。具体的には，完成した仕様から，(段階的に) それを実装するためのアルゴリズム，システムのリソースやその配置などを与えることで，最終的なプログラムに変化する。その変換は自動的に行うことが理想的である。(図 5.3[2]) しかし，そのための理論は現在でも確立していない。

操作的パラダイムの利点はプロトタイピングパラダイム の利点と似ている。それは操作的パラダイムの基本的な考え方は，プロトタイピングパラダイムのそれとよく似ている。どちらも，要求定義の工程に実行できるソフトウェアモデルを造ることである。プロトタイピングと違う点は，実装との切り離しが難しい点，仕様の厳密な記述は教育のためのコ

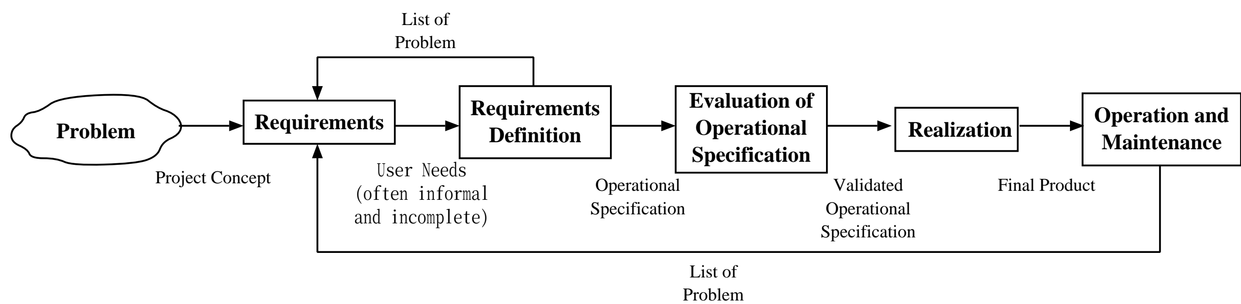

図 5.3: Software development using the operational paradigm

ストが係る点，そしてユーザとのギャップを埋めることが難しい点である。しかし，一度仕様が完成してしまえばシステムはどのアプローチよりも強固になる。操作的なアプローチの応用領域は，要求の変更が少なく，設計や実装に対する誤りのコストが高く付く領域である。これは，プロトタイピングの応用領域と区別できる。

ISDR 法をこのパラダイムに応用することも可能であろう。その場合，外部の仕様を固定してしまい，そのデータの抽象度を用いて内部の仕様を段階的に記述することになる。しかし，この場合いかに要求と設計や実装を切り離すかが問題となる。なぜなら，ISDR 法のプログラムは通常実行するための記述が含まれるからである。設計や実装部分を切り離すには，プログラム中の実装に関連した部分を抽象値を使って抽象化するか，論理式を導入する。しかし，論理式を導入すると詳細化のチェックや解釈が複雑になる。うまく仕様が記述できれば，一般の実行可能な仕様記述言語と同じように実行が可能になる。これら仕様記述言語との違いはそれ以降の工程でも同じ言語が使えるという点である。

5.3.4 ISDR 法のためのプロセスモデル

ISDR 法を応用する開発プロセスは，前節までに説明した 3 つのパラダイムの利点を合わせ持ち，欠点をうまく補ったものが理想である。

我々は，ISDR 法のためのプロセスモデルとして図 5.4のようなものを考案した。この前半の工程では進化的プロトタイピングのアプローチと同様にして要求の外部仕様を固める。この時，実行可能性を調べるために必要に応じて実験的プロトタイプを作成する。要求の外部仕様が固まったら，次にうまく設計や実装と切り離した内部仕様を抽象値を使って記述する。これが終ったら，探検的パラダイムと同様，設計，実装の工程において，段階的

に仕様を実装する。この時点で，要求仕様に誤りが見つかった場合，設計や実装した部分のプログラムを一時切り離して，要求の作成の工程に戻しその定義をやり直す。再定義された要求は前に設計，実装した部分と合成した後，設計，実装の工程を続ける。

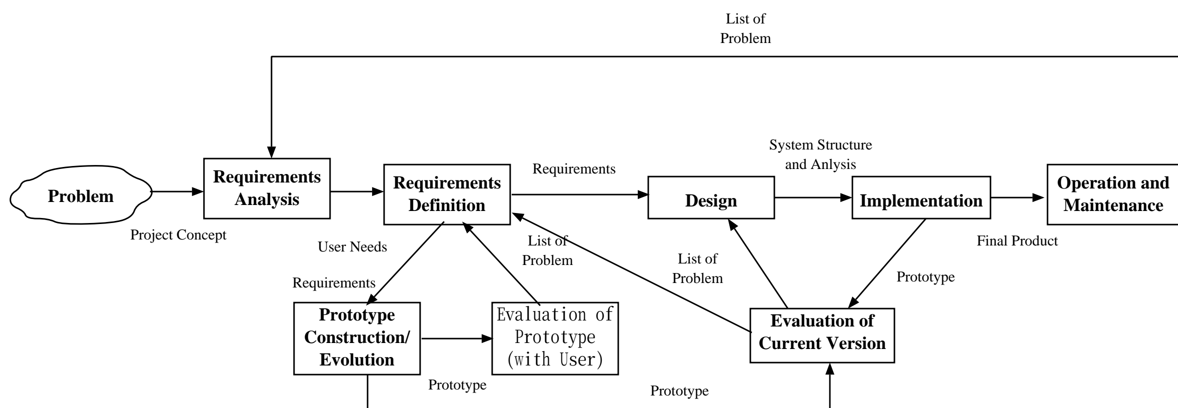

図 5.4: Software development using ISDR

このプロセスモデルを使った場合，ISDR 法をそれぞれをパラダイムに応用した場合と同様の問題がある。すなわち，拡張性の問題，要求と設計，実装とを区別できるかという問題，最終的に実装したプログラムの効率の問題である。これらの問題を解決する方法が必要となる。また，このプロセスの利点は，一貫して同じ言語が使えることである。そのために，各工程間の対応が付きやすく，バックトラックのコストも抑えられる。

5.4 柔軟な開発プロセス

5.4.1 バックトラック

2 章で定義した ISDR 法の開発プロセスでは，途中段階のプログラムのプログラムが仕様を満たしていることがチェック可能であることを前提にしている。しかし，実際の開発では，プログラムが仕様を完全に満たしているかチェックする事は困難で，その上詳細化途中の段階で摘出した抽象的な仕様が正しいかどうかも分からない場合もある。また，段階的に関数が定義できるようなドメインを捜し出せるとは限らない。この節では，途中段階で何らかの不都合が起きた場合，どのようにして対処するかを示す。

詳細化の途中段階の不都合には，過去のバージョンにバグが潜んでいたことが発見された場合と，どのようにデータを具体化しても多くの関数を一度に定義しないといけなくなる場合がある。後者の場合，それまで途中段階のプログラムをほとんど構成していなかったならば，事実上全体のプログラムを段階的に構成したことにはならなくなる。

これらの不都合が起こった場合，その開発プロセスを一時中断し，適当なバージョンまで戻り，そこまでのプログラム (以下ミスプログラムと呼ぶ) を取り除き，再び 2.3 節の詳細化プロセスを再開する必要がある。しかし，ここで行う詳細化では，通常の詳細化と違いそのミスプログラムの再利用が可能である。ミスプログラムの再利用形態には，(1) コードの再利用と (2) プログラムの実行結果の再利用の 2 つある。(1) は，データの具体化関係や関数の定義自体を再利用することである。また，(2) は新しく具体化する値とミスプログラムが扱っている値の具体化関係を利用して，ミスプログラムを**補助計算**に利用する場合や新しいプログラムの検証に利用する場合，詳細化の方針などに役立てる場合である。いずれの場合も再利用するミスプログラム自体には誤りがあってはならず，(1) の場合は，誤っている部分に関するコードは再利用しないように気をつける必要があり，(2) の場合は，誤った部分はあらかじめ完全に取り除いておく必要がある。その取り除きには，プログラムスライスの技法 [15] を用いる。すなわち，誤った入出力値に対してそれに関するスライスを計算しそれを取り除く。

コード の再利用

コードの再利用は，新しく具体化した値がミスプログラムが扱っている値と具体化関係にある場合に行える。

データの場合，ミスプログラム中の具体化関係にしたがって，新しいバージョンでも同様に具体化を行うことによってその関係を再利用する。もちろん，おなじ間違いを起こさないように注意して具体化する必要がある。

関数については，新しく定義する関数の入出力ドメインをそれぞれ $\mathbf{D}^{\mathrm{I}}, \mathbf{D}^{\mathrm{O}}$ とし，ミスプログラムの同じ処理を行う関数の入出力ドメインをそれぞれ $\mathbf{D}^{\mathrm{I}'}, \mathbf{D}^{\mathrm{O}'}$ とすると，以下の条件が成り立つ x', y' に関する部分について再利用可能である。

$$x \in \mathbf{D}^{\mathrm{I}}, y \in \mathbf{D}^{\mathrm{O}}, x' \in \mathbf{D}^{\mathrm{I}'}, y' \in \mathbf{D}^{\mathrm{O}'}. \qquad x' \preceq^{*} x \wedge y \preceq^{*} y'$$

このような場合，x'，y' に関してスライスを計算し，その中の x' を x，y' を y と置換えたコードを再利用する。ただし，このコード中に中間データが含まれていた場合，あらかじめそ

の中間データを扱えるようにしておく必要がある。もし，その中間データが今のバージョンのドメインと不整合を起こすなら，抽象値の名前をすべて置き換えてから再利用を行う。

実行結果の再利用

補助計算とは，新しく定義する関数の解釈を，ミスプログラム中のそれに対応する関数の計算結果を利用して行う方法である。具体的には，新しく定義する関数を fとし，ミスプログラム中のそれに対応する関数を f'とすると，それらの補助計算 ($[f, f']$) はつぎのように定義する。

$$[f, f'](x) \quad \equiv \quad max(f(x), y)$$

ただし，yは，ミスプログラム中の具体化関係上で $x' \preceq^* x$ が成り立つ x'について $f'(x') = y'$ かつ $y \prec y'$が成り立つ最も具体的な新しいプログラム中の値とする。そして，max は新しいプログラム中の具体化関係について具体的な方の実引数を返す関数である。

この補助計算は，直接ミスプログラムを参照しない場合でも次の性質からプログラムの検証に利用可能である。すなわち，ミスプログラム中のすべての対応する関数について上記の計算を行い，もし，その定義中にあるような yがあるにもかかわらず $f(x)$ と yの間に具体化関係が無い場合，新しいプログラムはミスプログラムと矛盾している。

この他にも，ドメインについて新しく具体化したドメインの具体化関係とミスプログラムのドメインの具体化関係と矛盾が無いかどうか検証できる。例えば，新しく定義したドメインにもミスプログラムにも含まれている任意の 2 値について，片方に $(\prec^*)$ 関係があれば他方にも同じ関係がないといけないという性質を利用して検証を行う。

ミスプログラムは補助計算の他にも，新しいプログラムの不変表明式としても再利用可能である。

5.4.2 プログラム合成

複数のプログラムを合成する操作によって，プログラムの再利用性が高まり，分散開発やボトムアップ開発などより柔軟な開発プロセスに対応できるようになる。この節では，ISDR 法で作成したプログラムの合成をどのように定義すればよいか述べる。

ISDR 法で作成したプログラムの合成は，そのデータに着目すれば容易に行えるようになる。具体的には，関数 h と gの合成関数 fはそれらの入出力ドメインをそれぞれ合成し

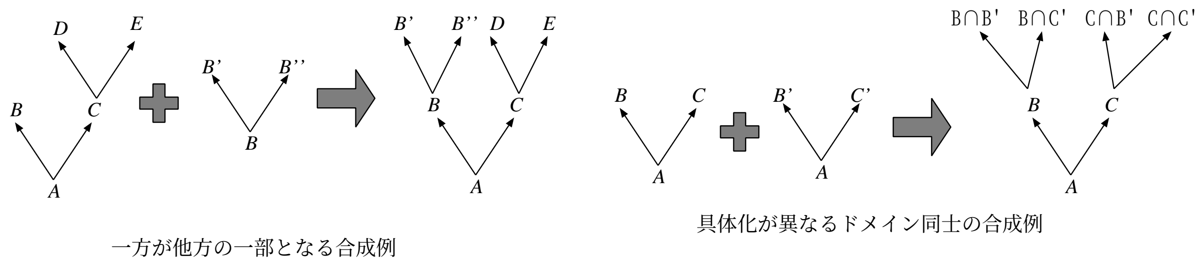

図 5.5: ドメインの合成例

たドメイン上の関数で，その値 $f(x)$ は，つぎのいずれかの条件がなりたつ zの内で最も具体的な値とする。

- x が h の入力値 ∧ $h(x) = y$ ∧ h のドメインに関して $z \preceq^* y$
- x が gの入力値 ∧ $g(x) = y$ ∧ gのドメインに関して $z \preceq^* y$
- x が h の入力値 x'の部分集合 ∧ $h(x') = y$ ∧ h のドメインに関して $z \preceq^* y$
- x が gの入力値 x'の部分集合 ∧ $g(x') = y$ ∧ gのドメインに関して $z \preceq^* y$
- x が gの入力値 x'と h の入力値 x''の共通集合 ∧ $g(x') = y'$ ∧ h のドメインに関して $z \preceq^* y'$ ∧ $h(x'') = y''$ ∧ gのドメインに関して $z \preceq^* y''$

ここでのドメインの合成は，それぞれのドメインの要素と，その和集合または共通集合を表す新しい抽象値をつかって行う。その方法には様々な場合が考えられ，一意には定まらない。例えば，図 5.5がドメインの合成例である。

5.5 他の方法論との比較

この節では，まず広く認知されている方法論である JSP 法 [11] と複合/構造化設計法を ISDR 法と比較検討し，つぎに他の方法論と比較する。

5.5.1 JSP 法との比較

JSP 法は ISDR 法と同様，データ中心のソフトウェアの構成法である。この方法では，まず入出力のデータ構造木を作り，つぎにこれらからプログラムデータ構造木を合成する。

このプログラムデータ構造木にオペレーションを付加した後，入力から出力を計算するプログラムを合成する。

この節では，まず，マイクロフィッシュ問題を JSP 法で構成する概要を述べた後，4.2 節の ISDR 法による解法と比較，検討する。

マイクロフィッシュ問題の JSP での解法

まず，JSP 法で作成する入出力データ構造はそれぞれ図 5.6の INPUT, OF1 上下のようになる。

次に，これらを 1 つのプログラムデータ構造にまとめる必要があるが，この問題の場合，入力データの ITEM-GP と CUST-GP が出力データと対応がとれないため，クラッシュが発生してしまう。(図 5.7)

そこで，図 5.8の様な中間のデータ構造 (IM1,IM2) を導入することで，この問題に対処する。この中間段階のデータは，ページを跨いではいけない行の集まりを区別するため，通常の行か，備考か，集計した結果の行かをあらわす識別子が付いている。そして，次のように 3 つのプログラムを構成し，その合成で全体のプログラムを構築する。すなわち，まずは顧客や項目の合計を数えながら中間データ IM1 を作成し，つぎに各々の中間データがページのどこに割り当てるかを考えながら中間データ IM2 を作成する。そして，最終的にはこのデータを一行ずつ処理することで出力を得る。

JSP 法と ISDR 法

JSP 法では，データの細部を決定し，それを扱うプログラムを組み合わせてより大きなプログラムを構成する。これに対し，ISDR 法ではまずバージョン 4 のプログラムで一種類の項目からフレーム列を生成する関数をつくった後，それ以降のバージョンでフレームの中身を埋める処理を定義した。つまり，まず粗いデータに対応したプログラムをつくり，後にデータの細部を決めながらプログラムを構成する。この方法では，データの細部がわからない段階で，仕様の検証を行いながらプログラムを構成できる。

この比較によって，次の結論を得た。すなわち，JSP 法では，プログラムのデータ構造を設計の最初の段階に決定する必要があるため，入出力の対応関係が直接つかない場合バンダリクラッシュが発生する。これに対し ISDR 法では，入出力データ構造は徐々にプログラムの構造とともに明らかにすればよいので，クラッシュを避ける事が出来る。その上，

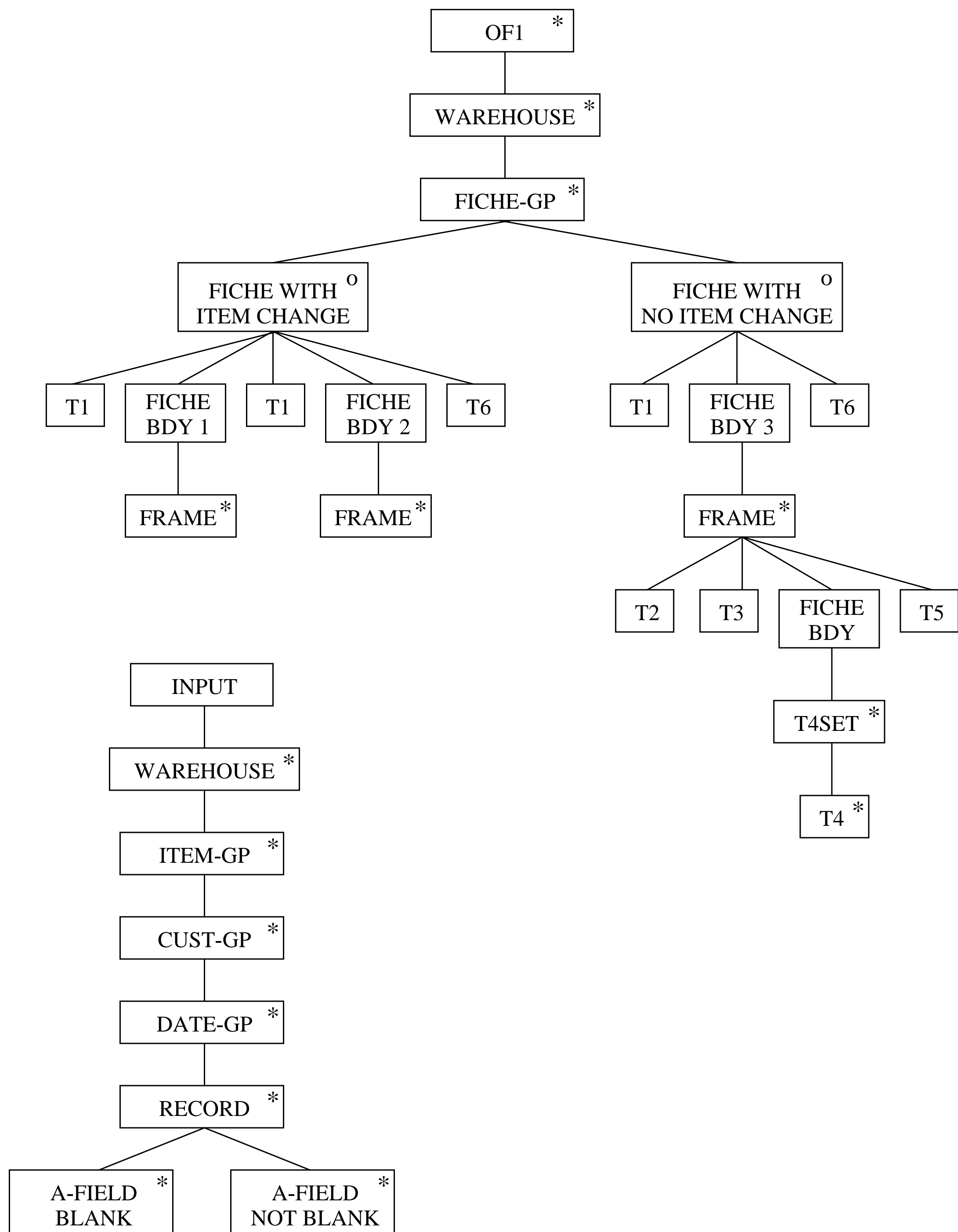

図 5.6: JSP の入出力データ構造

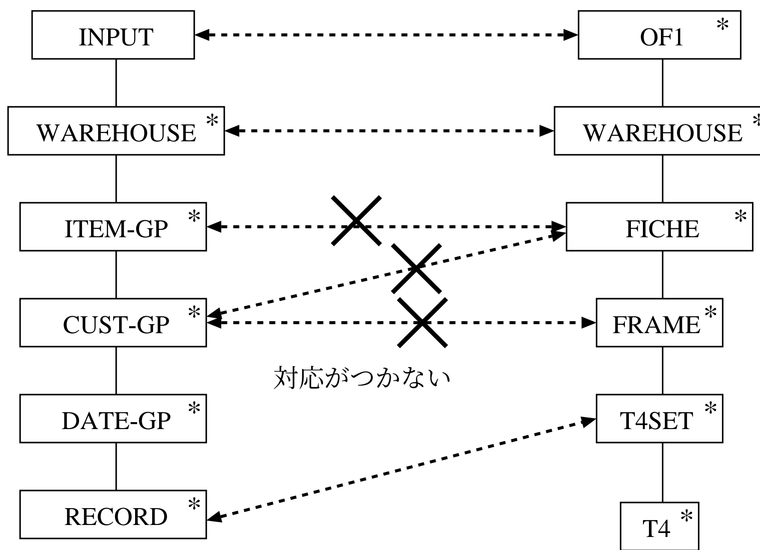

図 5.7: 入出力データ構造の不一致

IM1	W	I	RT	Data in final T4 record

W：倉庫番号
I ：品目番号
RT: IM1のタイプでつぎのいずれか
M： 元の入力レコードそのもの
X ： 備考
C ： 顧客毎の合計
I ： 項目毎の合計
B： 空行

IM2	W	T_j	Data in final T_j record

W：倉庫番号
T： レコードタイプ(2, 3, 4, 5のいず

図 5.8: 中間データの導入

方法	ISDR 法	JSP 法
プログラムの構成手順	概略から細部へ詳細化	データ構造を決めてからプログラムを構成
プログラムの分割	詳細化の任意の途中段階で関数を分割可能	中間データの詳細な構造を決めた後に個々のプログラムを構成
データ構造の枝数	30 個	59 個 (うち中間に 23 個)
関数の数/命令数	25 個 (のべ 37 個)	130 個くらい

表 5.1: ISDR 法と JSP 法の比較

JSP 法ではまったく考慮されていない途中段階のプログラムの実行が可能である。

ISDR 法と JSP 法の特徴と，マイクロフィッシュ問題で作成したプログラムの規模を表 5.1 にまとめた。

5.5.2 複合/構造化設計法

複合/構造化設計によるモジュール分解は，ISDR 法のデータの具体化やプログラムの詳細化と似た点がある。複合/構造化設計ではプログラムを源泉/変換/吸収分割，トランザクション分割，共通機能分割のいずれかに分解する。トランザクション分割は入力データの種類によってモジュールの分割を行う。これは入力データの具体化と直接対応可能である。また，源泉/変換/吸収分割は，モジュールを入力から内部データに変換するモジュール，それから出力データを計算するモジュール，最終的なデータ形式にする変換するモジュールへの分解である。この分解を ISDR 法に適用すると内部データを表現する新しいドメインが現れることになる。これまでも，このモジュール分解を段階的詳細化に取り入れた手法は提案さているが，それらには理論的な裏付けはなく，途中段階でプログラムは実行できない。

5.5.3 実行可能仕様記述言語

実行可能な仕様記述言語として，代数型言語がある。代数的に仕様を記述することで，抽象データ型など抽象度の高い記述が可能となり，その実行モデルとして項書換え型計算モデルを利用することが可能になる [16]。

代数型言語は，多ソート代数の枠組に基づき，抽象データ型を代数とみなして世界を記述する言語である。ここで，ソートとは世界を構成する対象 (またはデータ) の集合を指し，仕様としてそれらのソートに対する関数やソート間の同等関係を記述する。多ソート代数の中で，(1) どのソート中の要素も記述で用いた関数の合成によって構成できる，(2) 記述された等式で導けないような合同関係は存在しない，という性質をもつものを特に始代数と呼び，始代数による記述の意味と項書換えの操作的意味は一致することが知られている。また，多ソート代数に基づく言語で書かれたどんな記述に対しても，それを満たす代数の中に始代数が唯一つ存在することが証明されている。これらにより，多ソート代数に基づく仕様記述言語とその実行モデルの正当性が保証されている。

これらの仕様記述言語は対象の性質を等式として表現するため，ISDR 法のように関数でシステムを記述するのに比べ，データの表現形態に依存しないより抽象度の高い記述が可能であり，書かれた仕様の持つ性質や，正当性の証明がより容易である。そのため，開発プロセスの上流工程でのシステムの記述に適している。

実行可能な仕様記述言語の代表的な物には，Goguen らによる OBJ1[17] がある。OBJ1 は，システムを複数のソートとそれに対する関数や等式で構成されるオブジェクトの集まりで定義する。また，記述の再利用のための仕組みとしてパラメタライズドオブジェクトが用意されている。これによって，抽象的なオブジェクト中のソートをより具体的なソートに書き換え，それに対する関数や等式を追加しながら，より具体的なオブジェクトを定義できる。すなわち，抽象度の高い，あるいは条件が少ないオブジェクトから作り初め，そのオブジェクトを拡張することでシステムを段階的に記述可能である。これは，ISDR 法のデータに基づいた段階的詳細化とは異なる構成法である。

OBJ1 の実行はソートに対する等式を項書換え規則とみなして，与えられた式を正規項に書き換える事で行う。この時，Church-Rosser 性により正規項は一意に定まり，記述が始代数になる場合，その項はソートの要素に対応する。しかしながら，正規項はある一つの記号になるわけではなく，ソートの構成子の組合せで表現される。すなわち，式の値は，ISDR 法の場合その値が属している最小の集合に対応づけているのに対して，代数的仕様の場合はその値の構成のされ方に対応づけている。また，ISDR 法の場合，記述が不完全な場合もそれまでのバージョンを用いて何らかの集合 (抽象値) に対応づけることができるが，代数的仕様の場合は書き換えられなかった項の集まりになり必ずしもその意味を直観的にとらえることができなくなる。

OBJ1 は多ソート代数に基づく言語であるが，これにソートの包含関係を導入して，順序ソート等号論理に基づく言語として Goguen，二木らによって OBJ2[18] が提案されている。この言語では，ソートの包含関係によって，エラーや例外の処理，関数記号の重複，関数の多重継承が可能になっている。ソートの包含関係は，ソートを抽象値と対応させると，ISDR 法の抽象値間の具体化関係とちょうど逆の関係となる。しかしながら，ソートの関係はオブジェクトの詳細化のためだけに導入されたわけではなく，ISDR 法の具体化関係とは直接関係ない。また，OBJ2 ではオブジェクトの定義の他に theory や view の定義が行え，OBJ1 より再利用性が高くなっている。

OBJ2 での実行は，ソートの包含関係を考慮しながら行う。また，書き換え規則を適用する場合，関数に対する交換則，結合則などの性質の利用，どの項から書き換えるべきかの戦略の指定などを行えるようになっている。しかしながら，最終的に正規項を得るというコンセプトは OBJ1 と同様である。

5.5.4 その他の関連研究

データの濃度 (cardinality) に基づいて仕様を変化させる方法として，Feather によって濃度による仕様の進化 (cardinality evalution) が提案されている [19]。この属性の濃度に基づく進化については Balzer ら [12] によって提案されていたが，Feather はその一般的なプログラムのクラスを突き詰めている。Feather は濃度に基づく進化を次のように分類した。

項目 (item) の選択による分類

- *all*: 単一の項目に対する操作から n 個の項目全てに対する操作への進化
- *one*: 単一の項目に対する操作から n 個の項目中の 1 つに対する操作への進化
- *some*: 単一の項目に対する操作から n 個の項目のある部分集合に対する操作への進化

操作の順序による分類

- *sequential*: 新しく出現した項目をシーケンシャルに操作できる場合。
- *parallel*: 新しく出現した項目をそれぞれ非依存に並列に操作できる場合。
- *unbounded parallelism*: 全ての項目に対する操作が並列に操作できる。
- *bounded parallelism*: 項目の一部分に対する操作が並列に操作できる。

また，項目集合の特徴に基づいてその濃度を変化させる方針を示している。

この方法は進化をデータに着目して定義しているが，進化の前後のプログラムの数学的な定義はなく，中間のプログラムの実行も考慮されていない。そのため，データの変更に対する仕様のコンフリクトを機械的に捜し出すことが困難である。また，データの変更に対して仕様全体にその影響が伝播してしまう。

機能の段階的詳細化に基づく方法で，開発途中のプログラムの実行を可能にした構成法として鶴巻らによって HAWAII 法 [20] が提案されている。この構成法は，プログラム中の各関数ごとに詳細化をおこない，詳細化した関数間の依存関係を明示的に記述することで各段階でのプログラムの実行を可能にしている。しかし，データそのものの具体化を考慮しておらずデータ構造と関数の詳細化との対応も明確に示されていない。

データの詳細化をあつかった段階的詳細化法には，織田らによって SDR 法 [21] が提案されている。この方法では，プログラム中の各モジュールごとに，その入力データと出力データの関係を I/O 表として表現し，その I/O 表の詳細化にもとづいてモジュールの分解を決定していく。しかし，この方法は詳細化途中の各モジュールをそのまま実行することができない。

プログラムに対する詳細化や進化の他にも，CSP や Netri Net に対する詳細化や進化についての研究もある。Woodcoock らは CSP を用いてシステムを段階的に構成する方法に例を通して考察している [22]。[22] では，エスカレータの仕様を段階的に CSP を用いて記述している。具体的には，まず，1 つの呼出しボタン，1 つのフロア灯，1 組のドアだけの仕様から初め，これを 2 つのボタン，複数のフロアを持つ仕様に徐々に拡張して，最終的には複数のエレベータに対する仕様を構成している。しかしながら，これは詳細化の一例にすぎず，一般的な原理を示していない。

Netri Net を使って段階的にシステムを構成する方法は，Reisig によって提案された [23]。これは，ペトリネットを階層化し，その上位階層から順にシステムを詳細化する方法である。この論文ではシステムの詳細化を数学的に定義しているが，それは，状態と遷移のサブセットの関係のみからくる定義であり，詳細化前後の状態や遷移の対応関係を考慮にいれていない。その代わり，詳細化と包含の関係を表すスケジューリングを作りその一貫性と完全性を定義している。

Refinement Calculus にデータ構造の詳細化を導入する方法も Morgan らによって開発されている [24]。これは公理系に，データ間の詳細化関係を定義し，仕様中に現われる抽象

データについてより具体的なデータと置き換える規則を追加することで実現する。これによって，bag で表現されていたデータは，より具体的で効率のよいデータである整数などに具体化される。この方法は，データの構造のみに注目し，一般的な抽象値を扱っていない。

5.6 応用分野

ISDR 法はデータが階層的に分類できる場合や構造が複雑な問題に有効であると考えられる。具体的には，データベースを扱うアプリケーションの開発に向いている。このアプリケーションの詳細化は，データベースに追加するデータの抽象度に基づいて行う。まず，将来レコード型になる抽象値データを定義し，そのデータを段階的にその構造とそれに含まれるデータを認識するように具体化することにより，データベース更新関数が詳細化され，新しく現われた値を計算する関数が少しずつ定義できる。データベースアプリケーションで使われるテーブルや関連などはすべてレコードとして表現可能である。

その他の応用分野を考えた時，ISDR 法をどのような開発工程で用いるかによって変わってくる。プロトタイピングとして ISDR 法を使うのなら，ユーザインターフェースなど外部とどのようなやりとりすべきか決定するために適している。ただし，ユーザインターフェースを抽象値で記述する必要があるので，ウインドウの座標など見た目が非常に重要な場合には使えないかも知れない。しかし，ほとんどのユーザインターフェースはリストやレコードなどのデータ構造を用いて表現可能である。

5.7 他のシステムへの応用

本論文では関数型言語を使ってシステムを記述しているが，システムを抽象値を使って記述し抽象解釈によってそれを実行する概念は関数型言語に依存しない。オブジェクト指向の設計，開発にこの概念を応用し，ISDR 法の関数の詳細化や解釈のようにモデルの詳細化や解釈を付けることが可能である。

オブジェクト指向の開発と ISDR 法を融合する方法には以下の方法が考えられる。

- オブジェクトモデルや動的モデル，機能モデルを抽象的に記述し，段階的詳細化法によってモデルを洗練する
- クラス図を抽象値を用いて記述し，そのメソッドを段階的に構築する。

前者の場合，モデルの抽象的な記述は，抽象値を用いてモデルを記述するだけではなく，オブジェクトや状態自体を ISDR 法の抽象値としてみなすこともできる。そうすると，それらの間の関係を詳細化することでモデル自体の詳細化の定義ができ，すべてのバージョンのモデルを用いて抽象解釈を定義可能である。

また，抽象的に記述された各モデルの解釈法の概念は次のようになる。すなわち，バージョン n のモデルまたはそのインスタンスを $\mathbf{M_n}$ としそれに対する環境を環境 $\mathbf{E}$ とすると，その解釈 $\mathcal{I}$ はそれまでのバージョンの $\mathbf{M}$ をそれぞれのバージョンの環境で評価した結果の最大 (max) の環境となる。

$$\tilde{\mathcal{I}}_n\ \mathbf{M}\ \mathbf{E} \;\; = \;\; max(\mathcal{I}\ \mathbf{M}_0\ \mathbf{E}_0,\ \ \mathcal{I}\ \mathbf{M}_1\ \mathbf{E}_1,\ \ \ldots,\ \ \mathcal{I}\ \mathbf{M}_n\ \mathbf{E}_n)$$

ただし，$\mathbf{M}_i \sqsubseteq \mathbf{M}$，$\mathbf{M}_i \sqsubseteq \mathbf{M}$ が成り立つ。

後者の場合の具体的な手順は次のようになる。

1. 属性がとりうる値の集合をそれぞれ 1 つの抽象値として定義し，それらを用いてすべての属性，操作を定義しそれらの型を明らかにする。
2. 以下の手順ですべてのクラスを必要なだけ詳細化する。
   (a) ある操作に対し，それに対する振舞いがより明確になる様な抽象領域を考え，その領域に抽象値を一段階具体化する。
   (b) その値を用いて関連するメソッドを実装するためのアルゴリズムを考え，必要ならそのためのクラスを追加する。
   (c) 関連するメソッドを新しい抽象値を用いて実装する。
   (d) 新しく定義した関数について，具象化したドメインで型を推論する。
   (e) すべてのバージョンのプログラムを用いて，抽象解釈する。

ここで，詳細化のプロセスでは新しく定義した関数は型推論されるので，それに関するメソッドの型が以前のバージョンと一致しているかどうか調べる事によって，詳細化の誤りを調べる事ができる。

このように構成したプログラム中のメソッドの実行はドメイン中の値とその具体化関係を利用して行う。それぞれのメソッド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，これらは部分関数になっているので，定義されていない部分に関してはドメインの構造をもちいてデータを抽象化して，その抽

象度にあわせてインスタンスを生成する事により抽象実行を続ける。すなわち，メソッドの抽象実行はつぎの順序で行う。

1. 呼び出すメソッドにあわせて実引数を抽象化する。
2. その値を用いてメソッドを実行する。
3. もし，その引数についてメソッドの定義してなかった場合，メソッドのバージョンを一つ下げてそれを 1. からの手順にしたがって抽象実行する。

ここで，1. で呼び出すメソッドは，そのインスタンス変数に入っている値に関して定義してるバージョンを呼び出す。図 5.9に設計，実装プロセスと実行環境の概念を示す。

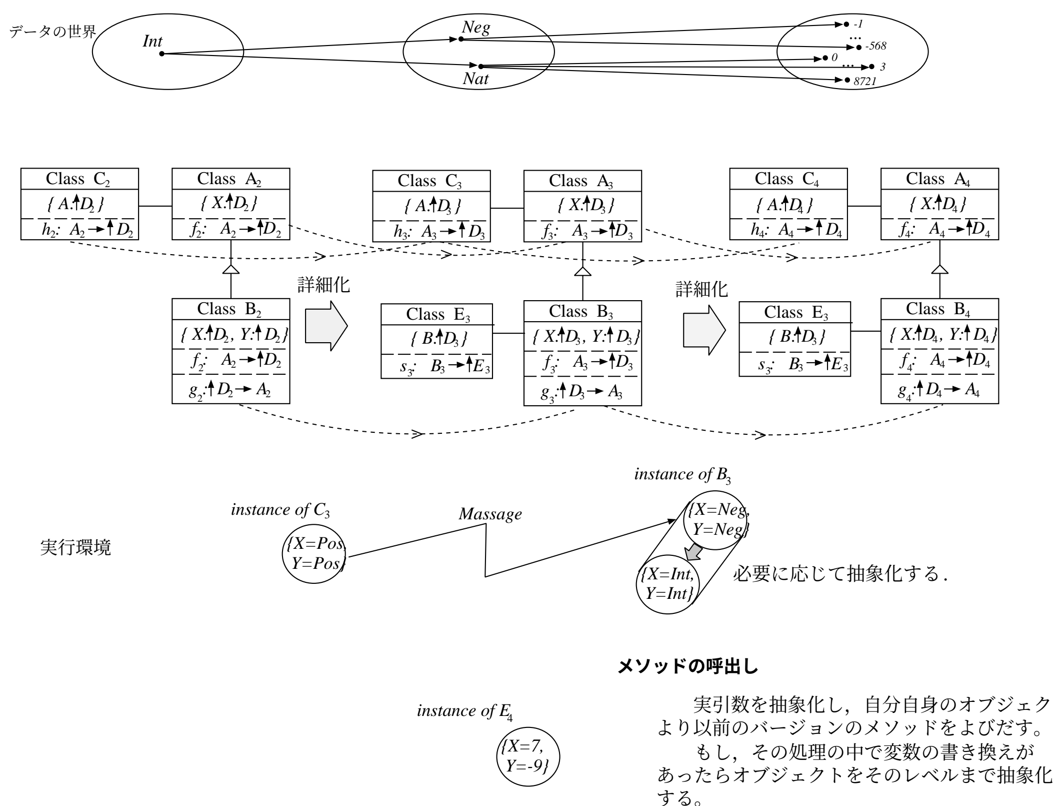

図 5.9: 設計，実装プロセスと実行環境

第 6 章

おわりに

6.1 まとめ

本論文では，途中段階のプログラムが実行可能なソフトウェアの段階的構成法（ISDR 法）を提案した。ISDR 法では，データの具体化に基づいてプログラムの詳細化を形式的に定義している。これによって，開発の途中段階でまだ未完全なプログラムを抽象解釈の技法に基づいて実行可能になった。また，詳細化を形式的に定義した事によって，構成の各段階でプログラムが正しく詳細化されているかどうかチェックする事が可能となった。そして，プログラムの詳細化方針を示すためにその一般的なパターンを示した。このパターンに従う事で自動的に詳細化が正しいことが言える。また，詳細化の途中に混入した誤りを発見するために，プログラム中に成り立つ不変表明式を言語の一部として記述できるようにした。

つぎに，ISDR の有効性を示すために，複雑な仕様を持つマイクロフィッシュ問題を解いた。また，同じ問題を JSP 法で解き，この方法と比較，検討を行った。その結果，JSP 法では簡単に構成できない領域でも，ISDR 法では問題なくプログラムが構成できることが分かった。これによって，ISDR 法が複雑な問題にも有効である事が示された。

そして，ISDR 法に基づく開発環境を実装した。ここで，プログラムの詳細化を補助するためのツールや詳細化の誤りをチェックするためのツールを作成した。この環境上で様々な領域のプログラムを構築する事で，ISDR 法の有効範囲や実用性などの評価が可能となる。

6.2 今後の展望

本論文により，システムを抽象的に記述し，それを抽象解釈しながら段階的にそのシステムを詳細化するための理論はほぼ確立した。今後，この理論をいかに他のシステムに応用し，実用的な方法論を提案，実装し，その有効性を示すかが課題となる。5.3.4 節に ISDR 法に適してるプロセスの概略を述べたが，実際にこのプロセスを利用するには問題点が多い。要求仕様の工程と設計，実装のプログラムの対応関係がうまくつけることがこの問題と鍵となるだろう。

本研究の究極的な課題は，環境や要求の変化に柔軟に対応できるソフトウェアの発展的な開発手法の提案である。そのようなソフトウェア開発では，環境や要求が変化した場合，そのための処理とそれまでのソフトウェアから新しいソフトウェアを合成する手法を規定する必要がある。本論文では，そのための原理の一つになるであろうシステムを構成するプリミティブな要素の抽象度に着目する方法を提案した。将来，この原理に基づいたソフトウェアの発展的な開発手法が提案されることを期待する。

謝辞

本研究を行なうに当たり, 終始御指導を賜わった片山 卓也教授に深謝致します.

また, 日頃から有益な御助言をいただき, 多面に渡って励ましていただいた片山研究室 助手 鈴木 正人博士に感謝致します.

最後に, 本論文をまとめるに当たって御協力いただいた片山研究室の諸兄に厚く御礼申し上げます.

参考文献

[1] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of the 9th International Conference on Software Engineering*, pages 328–339. IEEE Comptuer Society Press, 1987.

[2] W. Bischofberger and G. Pomberger, editors. *PROTOTYPING-ORIENTED SOFTWRE DEVEPLOPMENT CONCEPTS AND TOOLS*. Springer-Verlag, 1992.

[3] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21:61–72, 1988.

[4] Pamela Zave. The Operational Versus the Conventional Approach to Software Sevelopment. *CACM*, 27(2):104–118, 1984.

[5] Glenford J. Myers, 国友義久, 伊藤武夫訳, 編. ソフトウェアの複合/構造化設計. 近代科学社, 1979.

[6] Carroll Morgan, editor. *Programming from Specifications Second Edition*. Prentice Hall, 1994.

[7] Samon Abramsky and Chris Hankin, editors. *ABSTRACT INTERPRETATION OF DECLARATIVE LANGUAGE*. Ellis Horwood Limited, 1987.

[8] Robin Milner, editor. *The Definition of Standard ML*. The MIT Press, 1990.

[9] 横内 寛文, 編. プログラム意味論. 共立出版, 1994.

[10] 米澤明憲, 柴山悦哉, 編. モデルと表現. 岩波書店, 1992.

[11] John R.Cameron, editor. *JSP AND JSD: THE JACKSON APPROACH TO SOFTWARE DEVELOPMENT (SECOND EDITION)* . Michael Jackson Systems,Limited, 1989.

[12] Robert Balzer. Automated Enhancement of Knowledge Representations. In Aravind Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 203–207, Los Angeles, CA, 1985. Morgan Kaufmann.

[13] Bernard H. Boar, editor. *Application prototyping : a requirements definition strategy for the '80s.* Wiley, 1984.

[14] Christiane Floyd. A SYSTEMATIC LOOK AT PROTOTYPING. In R.Budde, K.Kuhlenkamp, L.Mathiassen, and H.Zullighoven, editors, *Approaches to prototyping*, pages 1–17. Springer-Verlag, 1984.

[15] Valdis Berzins, editor. *Software Merging and Slicing.* IEEE Computer Society Press, 1995.

[16] 二木厚吉, 外山芳人. 項書き換え関数型計算モデルとその応用. 情報処理, 24(2):133–146, 1983.

[17] Joseph Goguen and Jose Meseguer. RAPID PROTOTYPING IN THE OBJ EXECUTABLE SPECIFICATION LANGUAGE. In *ACM SIGSOFT SOFTWARE ENGINEERING NOTES*, pages 75–84, 1982.

[18] 井田哲雄, 田中二郎, 編. 続新しいプログラミング・パラダイム, 4 章, pages 77–105. 岩波書店, 1990.

[19] Martin S. Feather. Cardinality Evolution in Specifications. In Il Skokie, editor, *SEKE 2nd International Conference*, pages 575–583. Knowledge Systems Institute, 1990.

[20] 鶴巻 維男, 菊地 豊, 片山 卓也. ソフトウェア進化に基づくフォールトトレラント手法. 第 *11* 回大会論文集, pages 129–132. 日本ソフトウェア科学会, 1994.

[21] 織田 健, 片山 卓也. 出力指向の段階的詳細化に基づく設計法. 情報処理学会論文誌, 34(11):2251–2264, 1993.

[22] J.C.P Woodcock, S. King, and I.H Sørensen. Mathematics for Specification and Design: The Problem with Lifts. In *Fourth International Workshop on Software Specification and Design*, pages 265–268. Computer Society Press of the IEEE, 1987.

[23] W. Reisig. Petri Nets in Software Engineering. *Lecture Notes in Computer Science*, 255:63–96, 1986.

[24] Carroll Morgan and P. H . B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, 27(6):481–503, 1990.

[25] 織田 健. データ構造の段階的詳細化に基づくソフトウェア設計法. 博士論文, 東京工業大学 大学院 博士課程 情報工学専攻, 1993.

[26] 鶴巻 維男, 片山 卓也. 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, 東京工業大学 情報工学専攻, 1993.

[27] P. R. バード, ワドラー共著, and 武市 正人 訳, 編. 関数プログラミング. 近代科学社, 1991.

[28] 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, 2,4,6(13), 1996.

[29] Robin Milner, editor. *Commentary on Standard ML*. The MIT Press, 1991.

[30] Atsushi Ohori and Peter Buneman. Static Type Inference for Parametric Classes. In *ACM OOPSLA Conference*, pages 445–456, 1989.

[31] 所 真理雄, 松岡 聡, 垂水 浩幸, 編. オブジェクト指向コンピューティング, chapter 10. 岩波書店, 1993.

[32] Glynn Winskel, editor. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. The MIT Press, 1993.

第 A 章

付録

A.1 用語と記号の説明

本論文全体を通して以下の記号を使う。

$pow(x)$ – x のパワーセットを表す。

$\prec$ — システムでプリミティブに定義された関係に用いる。実際には人間が定義する関係である。同値を含む関係は$\preceq$で表す。

$\sqsubset$ — $\prec$から計算できる関係に用いる。

クリーネスター (*) — 関係の推移的閉包を表すために使う。

ボールド の変数 — 一般に集合を表現する。

$\sum \mathbf{S}$ — 集合 Sの要素数を表現する。

A.2 AL のシンタックス

この節では，抽象解釈に基づく関す型言語 AL のシンタックスを示す。
以下は関数定義のためのシンタックスである。

```
atexp   ::= scon
         |   <op>var
         |   <op>con
         |   ( exp )
```

```
exp     ::= atexp
        |   exp atexp   (* application *)
        |   exp id exp
        |   exp : ty
        |   fn match

match   ::= mrule < | match >

mrule   ::= pat => exp

dec     ::= val valbind
        |                 (* empty declatation *)
        |   dec <;> dec

valbind ::= pat = exp < and valbind >

atpat   ::= _
        |   scon
        |   <op>var
        |   <op>con
        |   ( pat )

pat     ::= atpat
        |   <op>con atpat
        |   pat con pat
        |   pat : ty
        |   <op>var<: ty> as pat

ty      ::= tyvar
        |   tyseq tycon
        |   ty -> ty
        |   ( ty )
```

以下はドメイン定義のためのシンタックスである。

```
dec     ::= domain tycon
        |   tycon refied dombind
dombind ::= domcon <: ty> < | conbind>
        |   any : ty < | conbind>
domcon  ::= tycon
        |   scon
        |   scon,<scon> ..
        |   scon,<scon> .. scon
        |   .. <scon,>scon
```

A.3 例題

A.3.1 在庫状況の計算

以下に，4.1 節で構成した例題の最後のバージョンで作成したプログラムを示す。

fun $F(x) = (h(x), rep(x), t(x))$

and $rep(x) =$ **if** $cond(x)$ **then** $Nulstock$

  else $(calc(take(x)), rep(drop(x)))$

and $take(Nulrept) = Nulrept$

  $\mid take(x, y) \quad = takesub(x, (x, y))$

and $drop(Nulrept) = Nulrept$

  $\mid drop(x, y) \quad = dropsub(x, (x, y))$

and $takesub(x, Nulrept) = Nulrept$

  $\mid takesub(x, (y, ys)) \quad =$

  if $samegrp(x, y)$ **then** $(y, takesub(x, ys))$

  else $takesub(x, ys)$

and $dropsub(x, Nulrept) = Nulrept$

  $\mid dropsub(x, (y, ys)) \quad =$

  if $samegrp(x, y)$ **then** $dropsub(x, ys)$

  else $(y, dropsub(x, ys))$

and $cond(Nulrept) = \mathrm{true}$

  $\mid cond((x, y)) \quad = \mathrm{false}$

and $calc(x, Nulrept) = initstock(\mathrm{x})$

  $\mid calc(x, y) \quad = upd(cal(x), calc(y))$

and $samegrp((i1, r1), (i2, r2)) =$

  if $sameid(i1, i2)$ **then** true

  else false

and $sameid(i1, i2) =$ **if** $(i1 = i2)$ **then** true

  else false

and $cal((id, (num, rst)) = make1(getqty(rst))$

and $upd(n,(id,m)) = (id, add(n,m))$

and $make1(n) \ = \ n$

and $getqty(n,_) \ = \ n$

and $add(n,m) \ = \ n+m$

and $initstock(id,(n,_)) = (id,n)$

and $h(x)$ = "header of report"

and $t(x)$ = "trailer of report";

A.3.2 イオン結合度を求める

データの分類が複雑な例として，金属元素と非金属元素のイオン結合の度合を調べるプログラムを段階的に構成する。そのプログラムは，入力として金属元素と非金属元素を取り，その結果としてそれらの間のイオン結合の度合を返す。ここで，金属元素と非金属元素は実際には次のような値である。

$$
\begin{aligned}
\text{金属元素の集合} &\equiv \{Li, Be, Na, Mg, K, Ca, Sc, Ti, V, Cr, Mn, \ldots\} \\
\text{非金属元素の集合} &\equiv \{He, B, C, N, O, F, Ne, Si, P, S, Cl, Ar, As \ldots\}
\end{aligned}
$$

原始プログラムの構成

まず，金属元素, 非金属元素の集合を抽象化することで原始ソフトウェアを作成する。ここでは金属元素の集合を表す抽象値として $Metal$, 非金属元素の集合を表す抽象値として $NonMetal$を定義し，イオン結合度はどう分類できるか分からないので抽象値を $Unknown$とする。この時，入力ドメイン $\mathbf{D}_0^{\mathrm{I}}$, $\mathbf{D}_0^{\mathrm{I}}$, 出力ドメイン $\mathbf{D}_0^{\mathrm{O}}$は以下のようになる。

$$
\begin{aligned}
\mathbf{D}_0^{\mathrm{I}} &= \langle \{Metal\}, \emptyset \rangle \\
\mathbf{D}_0^{\mathrm{I}} &= \langle \{NonMetal\}, \emptyset \rangle \\
\mathbf{D}_0^{\mathrm{O}} &= \langle \{Unknown\}, \emptyset \rangle
\end{aligned}
$$

そして，原始プログラムの主関数 F_0は，以下のようになる。

$$
\textbf{fun} \quad F_0(Metal, NonMetal) \quad = \quad Unknown
$$

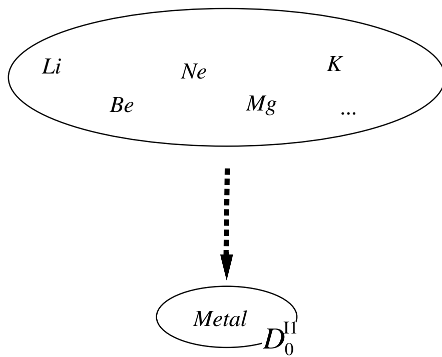

図 A.1: 金属データ集合の抽象化

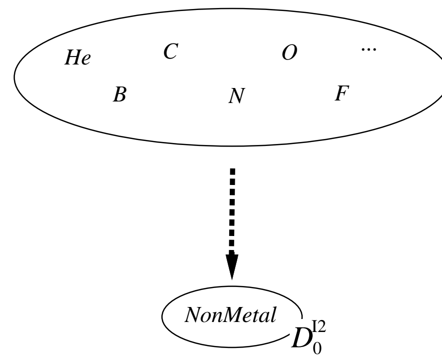

図 A.2: 非金属データ集合の抽象化

バージョン 1

詳細化方針　非金属元素を希ガスとそれ以外に分けて考える。それは，希ガスは他のどの元素とも反応しないからである。それに合わせ，イオンの結合度を結合ありとなしに分けることができる。つまり，非金属元素の $NonMetal$ を希ガスを表す抽象値 $Raregas$ と希ガス以外を表す抽象値 $NonRaregas$ に具体化する。そして，結合度の $Unknown$ をイオン結合度があることを表す値 $Ionize$ とイオン結合度がないことを表す値 $NonIonize$ に具体化する。金属元素は以前の値を用いる。この時，入力ドメイン $\mathbf{D}_1^{\mathrm{I}}$, $\mathbf{D}_1^{\mathrm{I}}$, 出力ドメイン $\mathbf{D}_1^{\mathrm{O}}$ は以下のようになる。

$$\begin{aligned}
\mathbf{D}_1^{\mathrm{I}} &= \mathbf{D}_0^{\mathrm{I}} \\
\mathbf{D}_1^{\mathrm{I}} &= \mathbf{D}_0^{\mathrm{I}}[NonMetal \prec Raregas, NonMetal \prec NonRaregas] \\
\mathbf{D}_1^{\mathrm{O}} &= \mathbf{D}_0^{\mathrm{O}}[Unknown \prec Ionize, Unknown \prec NonIonize]
\end{aligned}$$

そして，主関数 F_0 は次のように詳細化できる。

$$\mathbf{fun} \quad F_1(Metal, Raregas) \quad = \quad NonIonize$$

バージョン 2

詳細化方針　非金属元素の希ガス以外の元素をハロゲンとそれ以外に分ける。そして，金属を最外殻の電子軌道で分ける。これはハロゲン元素は, 相手の金属元素の最外殻の電子の数によってイオン結合度が変化するからである。それに合わせ，イオン結合が強い場合と

弱い場合に分ける。つまり，非金属の $NonRaregas$をハロゲンを表す抽象値 $Halogen$とハロゲン以外を表す抽象値 $NonHalogen$に具体化する。金属の $Metal$は S 軌道の金属を表す抽象値 $OrbitS$, F 軌道の金属を表す抽象値 $OrbitF$, D 軌道の金属を表す抽象値 $OrbitD$, P 軌道の金属を表す抽象値 $OrbitP$ に具体化する。そして，結合度の $Ionize$をイオン結合度が強いことを表す値 $StrongIon$ とイオン結合度が弱いことを表す値 $WeakIon$ に具体化する。この時，入力ドメイン $\mathbf{D}_2^{\mathrm{I}}$, $\mathbf{D}_2^{\mathrm{I}}$, 出力ドメイン $\mathbf{D}_2^{\mathrm{O}}$は以下のようになる。

$$\begin{aligned}
\mathbf{D}_2^{\mathrm{I}} &= \mathbf{D}_1^{\mathrm{I}}[Metal \prec OrbitS, Metal \prec OrbitF, Metal \prec OrbitD, Metal \prec OrbitP] \\
\mathbf{D}_2^{\mathrm{I}} &= \mathbf{D}_1^{\mathrm{I}}[NonRaregas \prec Halogen, NonRaregas \prec NonHalogen] \\
\mathbf{D}_2^{\mathrm{O}} &= \mathbf{D}_1^{\mathrm{O}}[Ionize \prec WeakIon, Ionize \prec StrongIon]
\end{aligned}$$

そして，主関数 F_1は次のように詳細化できる。

$$\begin{aligned}
\mathbf{fun} \quad & F_2(OrbitS, Halogen) &=& \quad StrongIon \\
| \quad & F_2(OrbitD, Halogen) &=& \quad WeakIon \\
| \quad & F_2(OrbitP, Halogen) &=& \quad NonIonize
\end{aligned}$$

バージョン 3

詳細化方針　D 軌道の金属を周期表の 4 行目,5 行目,6 行目に分ける。これは D 軌道の金属は周期表の下の方に近付く程ハロゲンとの結合が弱くなるためである。これに合わせ，弱い結合をその度合のよって 3 つに分ける。つまり，金属の $OrbitD$を 4 行目の元素を表す抽象値 $OrbitD4$, 5 行目の元素を表す抽象値 $OrbitD5$,5 行目の元素を表す抽象値 $OrbitD5$ に具体化する。そして，結合度の $WeakIon$を結合度の弱い順に $WWWIon$, $WWIon$, $WIon$ を表すの抽象値に具体化する。この時，入力ドメイン $\mathbf{D}_3^{\mathrm{I}}$, $\mathbf{D}_3^{\mathrm{I}}$, 出力ドメイン $\mathbf{D}_3^{\mathrm{O}}$は以下のようになる。

$$\begin{aligned}
\mathbf{D}_3^{\mathrm{I}} &= \mathbf{D}_2^{\mathrm{I}}[\{OrbitD \prec OrbitD4, OrbitD \prec OrbitD5, OrbitD \prec OrbitD6] \\
\mathbf{D}_3^{\mathrm{I}} &= \mathbf{D}_2^{\mathrm{I}} \\
\mathbf{D}_3^{\mathrm{O}} &= \mathbf{D}_2^{\mathrm{O}}[\{WeakIon \prec WIon, WeakIon \prec WWIon, WeakIon \prec WWWIon]
\end{aligned}$$

そして，主関数 F_2は次のように詳細化できる。

$$\begin{aligned}
\mathbf{fun} \quad & F_3(OrbitD4, Halogen) &=& \quad WIon \\
| \quad & F_3(OrbitD5, Halogen) &=& \quad WWIon \\
| \quad & F_3(OrbitD6, Halogen) &=& \quad WWWIon
\end{aligned}$$

バージョン 4

方針 S 軌道の金属を周期表のアルカリ金属, アルカリ土類金属に分ける。これは S 軌道の金属は，周期表の左側程ハロゲンとの結合が強くなるためである。これに合わせ，強い結合をその度合のよって 3 つに分ける。つまり，金属の $OrbitS$をアルカリ金属の元素を表す抽象値 $AlkaliM$とアルカリ土類金属の元素を表す抽象値 $AlEarthM$ に具体化する。そして，結合度の $StrongIon$を結合度の強い順に $SSSIon$，$SSIon$, $SIon$ を表すの抽象値に具体化する。この時，入力ドメイン $\mathbf{D}_4^{\mathrm{I}}$, $\mathbf{D}_4^{\mathrm{I}}$, 出力ドメイン $\mathbf{D}_4^{\mathrm{O}}$は以下のようになる。

$$
\begin{aligned}
\mathbf{D}_4^{\mathrm{I}} &= \mathbf{D}_3^{\mathrm{I}}[OrbitS \prec AlkaliM, OrbitS \prec AlEarthM] \\
\mathbf{D}_4^{\mathrm{I}} &= \mathbf{D}_3^{\mathrm{I}} \\
\mathbf{D}_4^{\mathrm{O}} &= \mathbf{D}_3^{\mathrm{O}}[StrongIon \prec SIon, StrongIon \prec SSIon, StrongIon \prec SSSIon]
\end{aligned}
$$

そして，主関数 F_3は次のように詳細化できる。

$$
\begin{aligned}
\mathbf{fun} \quad & F_4(AlkaliM, Halogen) &= \quad & SSIon \\
| \quad & F_4(AlEarthM, Halogen) &= \quad & SIon
\end{aligned}
$$

最後にこれらの抽象値を具体的にデータに具体化する事で目標のプログラムを得る事ができる。

このプログラムを構成するために最終的に構成した入出力ドメインを図 A.3から図 A.5に示す。

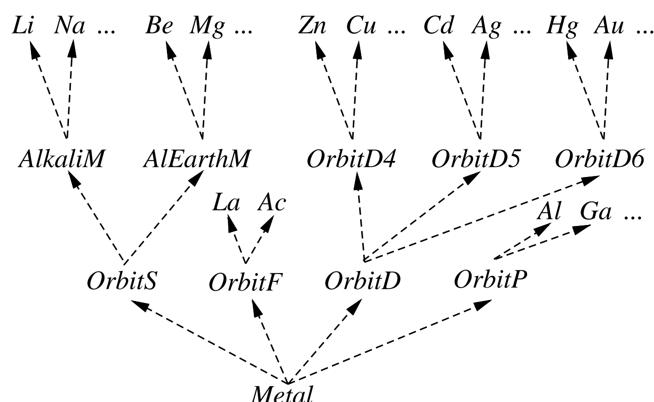

図 A.3: 金属データの具体化

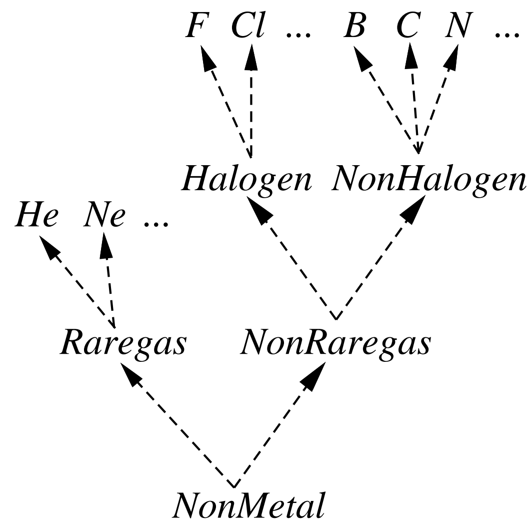

図 A.4: 非金属データの具体化

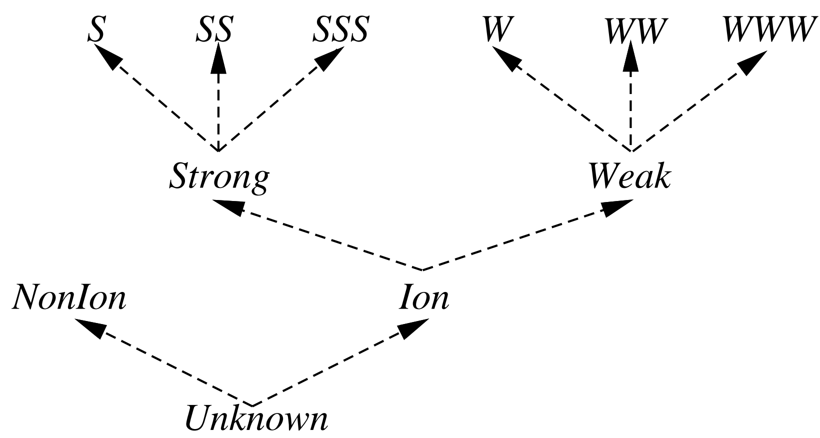

図 A.5: イオン結合度の具体化

本研究に関する発表論文

[1] 吉岡信和，鈴木正人，片山卓也: “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”，第 49 回全国大会講演論文集, 第 5 巻, pp.251–252, 情報処理学会，1994

[2] 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, Vol.95, No.25, pp.137–144, 情報処理学会，1995

[3] 吉岡信和，鈴木正人，片山卓也: “抽象解釈に基づくソフトウェアの段階的具象化法 — データ構造の導入 —”，第 12 回大会論文集, pp.13–16，日本ソフトウェア科学会, 1995

[4] 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”，第 13 回大会論文集, pp.197–200，日本ソフトウェア科学会, 1996

[5] 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，日本ソフトウェア科学会, 1997

[6] 吉岡信和，鈴木正人，片山卓也: “抽象解釈を用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，石川 俊，鈴木正人，片山卓也: “抽象解釈に基づくプログラムの発展的開発環境”，第 14 回大会, 日本ソフトウェア科学会, pp.645–648, 1998

[8] 吉岡信和，石川 俊，鈴木正人，片山卓也: “抽象解釈に基づくプログラムの発展的開発環境”，コンピュータソフトウェア, 日本ソフトウェア科学会, 投稿中

[9] 吉岡信和, 片山卓也: “抽象解釈に基づくプログラムの段階的構成法”, コンピュータソフトウェア, 日本ソフトウェア科学会, 投稿中

[10] Nobukazu Yoshioka, Suzuki Masato, Takuya Katayama: “Incremental Software Development Method based on Abstract Interpretation”, Proceedings of IWSSD-10, IEEE, 1998

索引

ウォーターフォールモデル, 2, 6

関数
　—の解釈, 29
関数の詳細化, 21

具体化, 32–34, 48–51, 69, 79, 80, 85, 87, 90

プログラム
　原始—, 15, 17, 18, 29

構造化アルゴリズム, 2

操作的パラダイム, 6, 69, 76

段階的構成法, 6, 8
段階的詳細化, 2, 6, 7, 22, 68, 85, 86, 88, 89
探検的プログラミング, 4, 67, 69, 74, 75

抽象解釈, 8, 9, 11, 12, 29, 76

データドメイン, 15, 32, 33
　—に成り立つ条件, 19
　—の具体化, 20
　—への要素の追加, 16
　—中の具体集合, 20
データの具体化, 8, 11, 12

不変表明式, 30
プログラム, 17, 32, 33
　—の構成, 21
　—の詳細化, 22
プロトタイピング, 4, 69–71
プロトタイプ, 4, 69–71