VDM의 자료구조인 set, sequence, map의 프로그래밍 언어 자료구조인 linked list로의 변환

유 문 성†

요 약

정형적 개발 방법론은 소프트웨어를 정확하고 체계적으로 개발하기 위하여 사용되며 시스템을 정형 명세 언어를 사용하여 명세하고 이를 구현할 때까지 점진적으로 시스템을 구체화하는 방법으로 개발한다. VDM은 정형 명세 언어의 하나로서 set, sequence, map의 수학적 추상적 자료구조를 사용하여 시스템을 명세하는 데 대부분의 프로그래밍 언어는 이런 자료구조를 가지고 있지 않다. 그러므로 이들 자료구조들의 변환이 필요하며 VDM의 수학적 자료구조들은 프로그래밍 언어의 자료구조인 연결 리스트로 변환 할 수 있다. 본 논문에서는 VDM의 set, sequence, map의 자료구조를 프로그래밍 언어의 자료구조인 연결 리스트로 변환하는 방법과 그 변환의 타당성을 수학적으로 증명하였다.

The Conversion of a Set, a Sequence, and a Map in VDM to a Linked List in a Programming Language

Moonsung Yoo†

ABSTRACT

A formal development method is used to develop software rigorously and systematically. In a formal development method, we specify system by a formal specification language and gradually develop the system more concretely until we can implement the system. VDM is one of formal specification languages. VDM uses mathematical data structures such as sets, sequences, and maps to specify the system, but most programming languages do not have such data structures. Therefore, these data structures should be converted. We can convert mathematical data structures in VDM to a linked list, a data structure in a programming language. In this article, we propose a method to convert a set, a sequence, and a map in VDM to a linked list in a programming language and prove the correctness of this conversion mathematically.

키워드 : 정형(formal development method(s)), 명세(specification, formal specificatio, specification language(s)n), 자료구조(data structure(s)), 자료 정제(data refinement), 자료 구체화(data reification), VDM(Vienna Development Method)

1. 서 론

소프트웨어 개발은 초기의 명세(specification)단계로부터 구현(implementation)단계로 진행된다. 정형 방법론(formal method)은 수학과 논리학에 기반을 둔 소프트웨어 개발 방법론으로 체계적이고 정확한 소프트웨어의 개발을 목표로 한다[1]. 정형 방법론에서는 시스템을 정형 명세 언어(formal specification language)[2-4]를 사용하여 명세하고 이를 구체화하는 방법으로 소프트웨어를 개발하는 방법론이 주로 사용된다. VDM[5-7]은 정형 명세 언어의 하나인데 자료구조로서 수학의 집합론에 기초를 둔 set, sequence, map의 추상 자료구조를 사용하며 일반 프로그래밍 언어의 자료구조인 배열이나 연결 리스트를 가지고 있지 않다. VDM의 수학적 자료구조들은 구체적인 자료구조를 사용하는 것보다 소프트웨어 개발 단계의 전반부에서 시스템을 명세하는 데 훨씬 효율적이며 적절하다. 정형 방법론을 사용한 경우도 구현(implementation)단계에서 프로그래밍 언어를 사용하는데 대부분의 프로그래밍 언어는 set, sequence, map의 자료구조를 가지고 있지 않다. 그러므로 VDM의 수학적 자료구조들은 프로그래밍의 자료구조로 변환이 필요하며 연결 리스트는 그러한 자료구조의 하나다[8]. 또한 정확한 소프트웨어의 개발을 위하여 VDM에서는 자료구조가 변환되었을 때 그 변환의 타당함을 수학적으로 엄밀하게 증명해야 된다[9, 10]. 본 논문에서는 VDM의 set, sequence, map의 자료구조를 프로그래밍 언어의 자료구조인 연결 리스트로 변환하는 방법과 그 변환의 타당성을 Jones[9, 10]가 제안한 검색·함수(retrieve function)를 사용하여 수학적으로 증명하였다.

본 논문의 구성은 다음과 같다. 2장에서는 VDM언어의 개략적인 소개를 하고 3장에서는 VDM과 프로그래밍 언어의

† 정 회 원 : 상지대학교 컴퓨터정보공학부 교수
논문접수 : 2001년 4월 30일, 심사완료 : 2001년 6월 25일

자료구조에 대해서 살펴본다. 4장에서는 자료 구체화(data reification)에 대한 설명을 하며 5장에서는 set, sequence, map을 linked list로 구체화 또는 변환하는 방법을 제시한 후 6장에서는 이 변환이 수학적으로 타당함을 밝힌 후 7장에서 결론을 맺는다.

2. VDM 개요

VDM(Vienna Development Method)[9-11]은 Z[12-15]과 함께 가장 많이 쓰이는 정형 명세 언어(formal specification language)로 비엔나에 있는 IBM 연구소에서 개발되었으며 개발에 참여한 핵심적인 두 인물은 Jones, C. B.와 Bjoner, Dines이다. VDM은 Z과 마찬가지로 집합론과 술어 논리학(predicate logic)에 그 기초를 두고 있다[16]. VDM의 표현법은 외부변수선언, 선조건(precondition), 후조건(postcondition)으로 구성되며. 그 구조는 다음과 같다.

```
함수명(매개 변수들)
ext
pre
post
```

(그림 1) VDM 표현법의 구조

위의 그림에서 **ext**은 **rd**(read only : 읽기 전용)와 **wr** (write-and-read access : 쓰고 읽기 겸용)으로 표현하는 외부 변수들을 선언하는데 사용된다. **pre** 키워드는 함수의 실행 전 만족해야할 술어들 즉, 선조건(pre-condition)들을 **post** 키워드는 함수의 실행 후 만족해야할 술어들 즉, 후조건(post_condition)들을 기술하는데 사용된다. 후조건에서는 이전상태의 변수들을 "<---"을 사용하여 나타낸다. 예를 들어 **post** position = position + v라고 표현하면 position은 연산이 이루어지기 전의 상태인 position을 의미한다. VDM을 사용하여 학번과 학생 이름으로 구성된 간단한 자료를 가지고 있으며 두 가지 연산인 find와 add를 가지고 있는 학적부 시스템을 나타내면 다음과 같다.

```
find(i : Id) n : Name
ext rd s : map Id to Name
pre i∈dom s
post n = s(i)

add (i : Id, n : Name)
ext wr s : map Id to Name
pre i∉dom s
post s = sU{i↦n}
```

(그림 2) VDM으로 표기한 학적부 시스템

3. VDM과 프로그래밍 언어에 있어서의 자료구조

VDM에서의 자료구조(또는 자료형)는 기본 구조와 복잡한 구조로 나뉜다[5-7]. 기본 구조는 숫자형(자연수, 정수, 실수 등)과 boolean형 그리고 문자형 등이 있다. 복잡한 구조에는 set, sequence, map등이 있으며 일반적인 프로그래밍 언어의 자료구조인 배열이나 연결리스트를 가지고 있지 않다. 간단한 시스템을 제외하고 시스템을 명세하는데 set, sequence, map의 사용은 필수적이다. VDM은 시스템 분석 단계에서부터 시스템을 명세하기 위하여 사용된다. 시스템 분석 단계에서 주요 초점은 "how(어떻게)"가 아니라 "what(무엇을)"이다. 따라서 구체적인 자료구조를 사용하는 것보다 추상적인 자료구조를 사용하는 것이 훨씬 효율적이며 적절하다. VDM의 기본 자료구조는 프로그래밍 언어에도 같은 형태로 존재하여 이들 사이의 변환에는 아무 문제가 없다. 그러나 대부분의 프로그래밍 언어에서는 set, sequence, map등 수학적 자료구조를 가지고 있지 못하기 때문에 이들 자료구조의 변환이 필요하다. 이들 자료구조들은 프로그래밍 언어의 배열이나 연결 리스트 자료구조로 변환할 수 있는 데 연결 리스트가 기억 장소를 유연하게 사용할 수 있으므로 본 논문에서는 연결 리스트로 변환하였다. VDM의 자료구조인 set, sequence, map 과 프로그래밍 언어의 자료구조인 연결 리스트의 정의와 관계된 연산은 다음과 같다.

3.1 set

set(집합)은 원소 또는 객체들의 모임으로, set에는 다음과 같은 연산들이 사용된다.

∈(속함), ∉(속하지 않음), ⊂(부분집합)
∪(합집합), ∩(교집합), \(차집합), ∩(분산 교집합),
∪(분산 합집합)

3.2 sequence

sequence(수열)은 요소들을 순서적으로 나열한 것으로, sequence에는 다음과 같은 연산들이 사용된다.

hd(head : 머리), **tl**(tail : 꼬리), **len**(length : 길이),
elms(elements : 요소), **inds**(indices : 색인), ⌢(연결)

3.3 map

map(함수)은 (키, 값) 쌍을 모은 것으로 키로 검색된다. 하나의 키에 대응될 수 있는 값은 하나 이하이며, map에는 다음과 같은 연산들이 사용된다.

dom(domain : 정의역), **rng**(range : 치역), expression^{-1}(역),
◁(domain restriction : 정의역 제한), ⋪(domain exclusion : 정의역 배제),
▷(range restriction : 치역 제한), ⋫(range exclusion : 치역 배제),
◦(결합), ↑(반복), †(수정), U(병합), =(동등), ≠(부등)

3.4 연결 리스트

연결 리스트(연결 리스트)는 각 노드가 다음노드를 가리키는 포인터를 갖는 자료구조로 연결 리스트와 관계된 중요 연산과 이 논문에서 사용될 연산은 다음과 같다.

insert : 연결 리스트에 노드를 추가
delete : 연결 리스트에서 노드를 삭제
count : 연결 리스트의 노드의 수
at(i) : 연결 리스트가 한 개의 자료(data) 필드를 갖고 있을 때 i번째 노드의 자료 값
first_at(i) : 연결 리스트가 두 개의 자료(data) 필드를 갖고 있을 때 i번째 노드의 첫 번째 자료 값
second_at(i) : 연결 리스트가 두 개의 자료(data) 필드를 갖고 있을 때 i번째 노드의 두 번째 자료 값

4. 자료 구체화(data reification)

자료 구체화(data reification)[9, 10, 17, 18] 또는 자료 정제(data refinement)[19-21]는 명세에 나타난 추상적인 자료구조를 구현할 수 있는 형태인 구체적인 자료구조로 변환하는 과정이며 그 변환의 타당함을 수학적으로 엄밀하게 증명해야 된다[9, 10, 22].

추상적 자료구조와 구체적 자료구조 사이의 관계는 **일대 다(one-to-many)** 인 관계가 있다. 왜냐하면 추상 값은 한 개 이상의 구체적인 값을 가질 수 있기 때문이다. 추상 자료구조와 구현 자료구조 사이의 관계는 후자에서 전자로 가는 함수로 표현될 수 있는데 이 함수를 검색(retrieve) 함수라고 부른다. 올바른 변환이 되기 위해서는 검색 함수가 두 가지 성질 즉 전사성(全寫性 : totality)과 적합성(adequacy)를 만족해야 한다.

함수 $f : X \rightarrow Y$가 $dom(f) = X$를 만족 할 경우 전사적(**total**)이라고 한다. 함수가 전사(total)가 되기 위해서는 구현상의 불변 값(invariant)을 강화할 필요가 있다. 그래야 검색 함수가 모든 값에 대해서 정의될 수 있기 때문이다. 검색 함수에서 어떠한 추상적인 값이라도 최소한 하나의 구현되는 값이 있을 때 이 함수를 적합(**adequate**)하다고 하며 다음과 같이 표현된다.

$$\forall a \in Abstract \bullet \exists r \in Representation \bullet retr(r) = a \text{ for } retr : Representation \rightarrow Abstract$$

자료구조가 변환된 후에는 선택된 구현 표현에 맞게 각 연산 및 함수가 다시 쓰여져야 한다. 이것을 operation modeling(연산 모형)이라고 한다. 구체화된 구현은 연산 표현이 더 복잡하게 되고 알고리듬이 정교해진다. 자료구조와 관련된 연산에 필요한 proof obligations(필수 증명)에는 domain rule(정의역 규칙)과 result rule(결과 규칙)이 있는데 다음과 같다.

domain rule -
$\forall r \in R \bullet pre\text{-}A(retr(r)) \Rightarrow pre\text{-}R(r)$
result rule -
$\forall \overline{r}, r \in R \bullet pre\text{-}A(retr(r)) \wedge post\text{-}R(\overline{r}, r) \Rightarrow post\text{-}A(retr(\overline{r}), retr(r))$

여기서 pre-A와 post-A는 추상 상태에서의 선조건과 후조건이고 pre-R과 post-R은 구현상태에서의 선조건과 후조건이다.

5. Set, Sequence, Map에서 연결 리스트로의 자료 구체화

SET $\{a_1, a_2, \cdots, a_i\}$과 SEQUENCE $[a_1, a_2, \cdots, a_i]$를 연결 리스트로 변환하는 방법은 다음과 같다.

> a_k를 데이터 필드로 갖는 k 번째 노드($1 \leq k \leq i$)를 만들고 k 번째 노드가 (k + 1)번째 노드를 가리키도록 링크 필드를 만든다. 단 마지막 즉 i번째 노드의 링크필드는 NIL이 되게 한다. 이렇게 연결 리스트로 구현한 SET와 SEQUENCE를 SET_L과 SEQUENCE_L이라고 설정한다.

MAP $\{a_1 \mapsto b_1, a_2 \mapsto b_2, \cdots, a_i \mapsto b_i\}$을 연결 리스트로 변환하는 방법은 다음과 같다.

> a_k를 첫 번째 데이터 필드로 갖고 b_k를 두 번째 데이터 필드로 갖는 k 번째 노드($1 \leq k \leq i$)를 만들고 k 번째 노드가 (k+1)번째 노드를 가리키도록 링크필드를 만든다. 단 마지막 즉 i번째 노드의 링크 필드는 NIL이 되게 한다. 이렇게 연결 리스트로 구현한 MAP을 MAP_L이라고 설정한다.

다음에 명세 단계와 구현 단계에서 사용한 자료구조와 연관된 연산들을 나타내야 한다. 각 연산에 대한 명세 표현은 유사하기 때문에 각 자료구조에서 대표적인 연산 하나에 대해서만 살펴보기로 한다. 즉, SET에서는 union연산, SEQUENCE에서는 head연산, 그리고 MAP에서는 DomRestr (Domain Restriction)연산에 대해 그 연산들의 명세 단계와 구현단계에서의 선조건과 후조건을 VDM표현을 사용하면 다음과 같다.

명세 단계	구현 단계
union(B : SET) R : SET	union (B : SET_L) R : SET_L
ext rd A : SET	ext rd A : SET_L
pre	pre post ($\forall$i : 1..R.count
post R = A ∪ B	($\exists$j : 1..A.count : R.at(i) = A.at (j))
	∨ ($\exists$k : 1..B.count : R.at(i) = B.at (k)))
head() R : T	head() R : T
ext rd A : SEQUENCE	ext rd A : SEQUENCE_L
pre	pre
post R = hd A	post R = A.at (1)

여기서 T는 SEQUENCE 요소의 자료형(정수, 문자 등)이다.

DomRestr (S : SET) R : MAP	DomRestr (S : SET_L) R : MAP_L
ext rd M : MAP	ext rd M : MAP_L
pre	pre
post R = S◁M	post (∀i : 1..S.count (∃j : 1..R.count : (∃k : 1..M.count : M.firat_at (k) = S.at (i) ∧ R.first_at (j) = M.first_at (k) ∧ R.second_at (j) = M.second_at (k))))

6. 자료 구체화(data reification)에 관한 증명

추상적인 자료구조를 구현상의 자료구조로 변환하면 그 타당함을 엄밀하게 논리적으로 보여야 한다[9, 10, 22]. 그러기 위해서는 우리는 구현 자료구조에서 추상(또는 명세) 자료구조로의 접근(retrieve) 함수가 전사적(total)이고 적합(adequate)함을 보여야한다. 추상 자료구조 set, sequence와 map의 구현 자료구조는 연결 리스트인데 여기서는 SET, SEQUENCE, MAP과 LINKED_LIST로 표현한다.

• SET에서의 접근 함수 retr1은 다음과 같이 정의된다.

retr1(dr) = {dr.at(1), dr.at(2), ···, dr.at (count)}. 여기서 count는 연결 리스트의 노드 수이다. retr1이 전사적(total)이며 적합(adequate)함은 (정리 1)과 (정리 2)와 같다.

(정리 1) retr1은 전사적(total)이다.

(증명) LINKED_LIST의 모든 노드 dr에 대하여 dr.at(i) (1≦i≦count)가 존재하므로 {dr.at(1), dr.at(2),···, dr.at(count)}가 존재한다. 그러므로 retr1은 전사적(total)이다.

(정리 2) retr1은 적합(adequate)하다.

(증명) 구조적 귀납법(structural induction)을 사용하여 증명한다.

from d∈ SET
1. ∃dr ∈ LINKED_LIST · dr.count = 0
2. retr1(dr) = {}
3. ∃dr ∈ LINKED_LIST · retr1(dr) = {}
4. from d∈ SET(T), w ∉ d
   ∃dr ∈ LINKED_LIST · retr1(dr) = d
4.1 from dr ∈ LINKED_LIST, retr1(dr) = d
4.1.1 {dr.at (1), dr.at (2),···, dr.at (count)} = d
4.1.2 w ∉ {dr.at (1), dr.at (2),···, dr.at (count)}
4.1.3 ∃dr1 ∈ LINKED_LIST ∀i · 1≦i≦count ·
dr1.at(i) = dr.at (i), dr1.at (count+1) = w
4.1.4 retr1(dr1)=d ∪ { w }
infer ∃dr1 ∈ LINKED_LIST · retr1(dr1) = d ∪ { w }
infer ∃dr1 ∈ LINKED_LIST · retr1(dr1) = d ∪ { w }
infer ∃dr ∈ LINKED_LIST · retr1(dr) = d

• SEQUENCE에서의 접근 함수 retr2는 다음과 같이 정의된다.

retr2(dr) = [dr.at(1), dr. at(2),···, dr.at(count)]. retr2가 전사적(total)이며 적합(adequate)함은 (정리 3)과 (정리 4)와 같다.

(정리 3) retr2는 전사적(total)이다.

(증명) LINKED_LIST의 모든 노드 dr에 대하여 dr.at(i) (1≦i≦count)가 존재하므로 [dr.at(1), dr.at(2),···, dr.at(count)]가 존재한다. 그러므로 retr2는 전사적(total)이다.

(정리 4) retr2는 적합(adequate)하다.

(증명) 구조적 귀납법을 사용하여 증명한다.

from d ∈ SEQUENCE
1. ∃dr ∈ LINKED_LIST · dr.count = 0
2. retr2(dr) = []
3. ∃dr ∈ LINKED_LIST · retr2 (dr) = []
4. from d ∈ SEQUENCE, w ∉ elems d
   ∃dr ∈ LINKED_LIST · retr2 (dr) = d
4.1 from dr ∈ LINKED_LIST, retr2 (dr) = d
4.1.1 [dr.at (1), dr.at (2), ···, dr.at (count)] = d
4.1.2 w ∉ [dr.at (1), dr.at (2), ···, dr.at (count)]
4.1.3 ∃dr1 ∈ LINKED_LIST∀i · 1≦i≦count ·
dr1.at(i) = dr.at (i), dr1.at(dr.count+1) = w
4.1.4 retr2(dr) = d ⌢ [w]
infer ∃dr1 ∈ LINKED_LIST · retr2 (dr1) = d ⌢ [w]
infer ∃dr1 ∈ LINKED_LIST · retr2 (dr1) = d ⌢ [w]
infer ∃dr ∈ LINKED_LIST · retr2 (dr1) = d

• MAP의 구현은 두 개의 자료 필드가 있는 LINKED_LIST로 구현된다. MAP에서의 접근 함수 retr3은 다음과 같이 정의된다.

retr3 (dr) = {dr.first_at (1)↦dr.second_at (1),dr.first_at (2)↦dr.second_at (2) ,···, dr.first_at (count)↦dr.second_at (count)}. retr3이 전사적(total)이며 적합(adequate)함은 (정리 5)와 (정리 6)과 같다.

(정리 5) retr3은 전사적(total)이다.

(증명) LINKED_LIST의 모든 노드 dr에 대하여 dr.first_ at (i)과 dr.second_at (i) (1≦i≦count)가 존재하므로 {dr.first_at (1)↦dr.second_at (1), dr.first_at (2)↦dr.second_at (2), ···, dr.first_at (count)↦dr.second_at (count)}가 존재한다. 그러므로 retr3은 전사적(total)이다.

(정리 6) retr3은 적합(adequate)하다.

(증명) 구조적 귀납법을 사용하여 증명한다.

1. ∃dr ∈ LINKED_LIST · dr.count = 0
2. retr3 (dr) = {}
3. ∃dr ∈ LINKED_LIST · retr3 (dr) = {}

4. from d ∈ MAP, w = {f↦s} · f ∉ dom d
∃dr ∈ LINKED_LIST · retr3 (dr) = d
4.1 from dr ∈ LINKED_LIST, retr3 (dr) = d
4.1.1 {dr.first_at (1)↦dr.second_at (1),
dr.first_at (2)↦dr.second_at (2),⋯,
dr.first_at (count)↦dr.second_at (count)} = d
4.1.2 f ∉ {dr.first_at (1),dr.first_at (2),⋯,dr.first_at (count)}
4.1.3 ∃dr1 ∈ LINKED_LIST, ∀i · 1≦i≦count ·
dr1.at (i) = dr.at (i),
dr1.first_at (count+1) = f, dr1.second_at (count+1)} = s,
{dr1.first_at (count+1) ↦dr1.second_at (count+1)} = w
4.1.4 retr3 (dr) = dU{w}
infer ∃dr1 ∈ LINKED_LIST · retr3 (dr1) = dU{w}
infer ∃dr1 ∈ LINKED_LIST · retr3 (dr1) = dU{w}
infer ∃dr ∈ LINKED_LIST · retr3 (dr) = d

다음은 SET, SEQUENCE, MAP과 관련된 연산이 LINKED_LIST로 변환하였을 때의 연산과 같은 기능을 한다는 것을 보여야한다. 이에 관련된 proof obligations(필수 증명)은 domain rule(정의역 규칙)과 result rule(결과 규칙)이다. 대부분의 연산에 있어서 명세와 구현의 선조건(pre-condition)이 TRUE이기 때문에 domain rule에 관한 증명은 할 필요가 없다. 따라서 result rule에 대한 증명만 필요하다. 각 자료구조의 대표적인 연산에 대하여 5장에서 기술한 표현을 사용하여 result rule을 (정리 7),(정리 8),(정리 9)에서 기술하고 이에 대하여 증명하였다.

- SET에서의 UNION 연산이 연산에 있어 result rule은 (정리 7)과 같다.

(정리 7)

∀i : 1..R.count,
(∃j : 1..A.count,
R.at (i) = A.at (j)) or
(∃k : 1..B.count,
R.at (i) = B.at (k))
⇒
retr1(R) = retr1(A) ∪ retr1(B)

(증명)

from A, B, R ∈ SET
where retr1(A) = {A.at (1), A.at (2),⋯, A.at (A.count)},
retr1(B) = {B.at (1), B.at (2),⋯,B.at (B.count)} ,
retr1(R) = {R.at (1), R.at (2),⋯,R.at (R.count)}

1 from ∀i : 1..R.count,
(∃j : 1..A.count,
R.at (i) = A.at (j)) or
(∃k : 1..B.count,
R.at (i) = B.at (k))
1.1 retr1(R) = {R.at (1), R.at (2),⋯,R.at (R.count)}
= {A.at (1), A.at (2),⋯,A.at (A.count),
B.at (1),B.at (2),⋯,B.at (B.count)}
= {A.at (1), A.at (2),⋯,A.at (A.count)}∪
{B.at (1), B.at (2),⋯,B.at (B.count)}
infer retr1(R) = retr1(A) ∪ retr1(B)
2 δ(∀i : 1..R.count,
(∃j : 1..A.count,
R.at (i) = A.at (j)) or
(∃k : 1..B.count,
R.at (i) = B.at (k)))
infer
(∀i : 1..R.count,
(∃j : 1..A.count,
R.at(i) = A.at (j)) or
(∃k : 1..B.count,
R.at(i) = B.at (k)))
⇒
retr1(R) = retr1(A) ∪ retr1(B)

- SEQUENCE에서의 HEAD 연산이 연산에 있어 result rule은 (정리 8)과 같다.

(정리 8)

R = A.at (1)
⇒
R = hd(retr2 (A))

(증명)

from A ∈ SEQUENCE
1. from R = A.at (1)
1.1 hd(retr2 (A)) = hd[A.at (1), A.at (2),⋯, A.at (A.count)] = A.at (1)
infer retr2 (R) = hd(retr2 (A))
2. δ(R = A.at (1))
infer R = A.at (1)
⇒
retr2 (R)=hd(retr2 (A))

- MAP에서의 DomRestr(Domain Restriction : 정의역 제한) 연산이 연산에 있어 result rule은 (정리 9)와 같다.

(정리 9)

(∀i : 1..S.count,
(∃j : 1..R.count,
(∃k : 1..M.count,
M.first_at (k) = S.at (i)
R.first_at (j) = M.first_at (k)
R.second_at (j) = M.second_at (k)))
⇒
R = S ◁ M

(증명)

```
from M ∈ MAP
1.from  (∀i : 1..S.count,
          (∃j : 1..R.count,
           (∃k : 1..M.count,
                 M.first_at (k) = S.at (i)
                 R.first_at (j) = M.first_at (k)
                 R.second_at (j) = M.second_at (k))))
1.1 retr1(S) ◁ retr3(M) = {S.at (1), S.at(2),···,S.at (S.count)} ◁
   {M.first_at (1)↦M.second_at (1).,···,
   M.first_at (M.count)↦M.second_at (M.count)}
     = {R.first_at (1)↦R.at (1).second,···,
       R.first_at (R.count)↦R.second_at (R.count)}
infer = retr3 (R)
2. δ(∀i : 1..S.count,
      (∃j : 1..R.count,
       (∃k : 1..M.count,
             M.first_at (k) = S.at (i)
             R.first_at (j) = M.first_at (k)
             R.second_at (j) = M.second_at (k))))
  infer (∀i : 1..S.count,
        (∃j : 1..R.count,
         (∃k : 1..M.count,
               M.first_at (k) = S.at (i)
               R.first_at (j) = M.first_at (k)
               R.second_at (j) = M.second_at (k))))
       ⇒
      R = S ◁ M
```

7. 결 론

VDM을 사용하여 명세한 소프트웨어를 일반적인 프로그래밍 언어로 구현할 때 해결하여야 할 문제가 VDM에서 사용한 수학적 자료구조인 set, sequence, map의 프로그래밍 언어의 자료구조로의 변환이다. 이 수학적 자료구조들은 프로그래밍 언어에서 사용되는 자료구조인 연결 리스트로의 변환이 가능한데 이 논문에서는 그 변환 방법과 그 변환의 타당성을 수학적으로 증명하였다. 이 논문에서는 비록 VDM에서 사용하는 표기법을 사용했지만 일반적으로 명세에서 사용한 수학적 자료인 set, sequence, map은 프로그래밍 언어의 자료구조인 연결 리스트로의 변환이 가능함을 보였다.

참 고 문 헌

[1] Pressman, R. S., "Software Engineering : A Practitioner's Approach," 5th edition, McGraw-Hill, 2000.

[2] Hinchey, M. G. and Bowen, J. P., "High-Integrity System Specification and Design," FACIT series, Springer-Verlag, London, 1999.

[3] Wing, J. M. "A specifier's introduction to formal methods," IEEE Computer, Sept. 1990.

[4] Moller, B., Partsch, H., and Schman, S.(eds.) "Formal Program Development," Springer-Verlag, 1993.

[5] Andrews, D. J. and others, "Information technology programming language-VDM-SL," First Committee Draft Standard : CD13817-1 Document ISO/IEC JTC1/SC22/ WG19 N-20, November, 1993.

[6] Andrews, D., and Ince, D. "Practical formal method with VDM," McGraw Hill, 1991.

[7] Parkin, G. I. "Vienna Development Method Specification Language (VDM-SL)," Computer Standard and Interfaces, 16 : pp.527-530, 1994.

[8] Pratt, T. W., Zelkowitz, M. V., "Programming Languages : Design and Implementation," 4th edition, Prentice Hall, 2001.

[9] Jones, C. B. "Software Development : A Rigorous approach," Prentice Hall Int'l, 1980.

[10] Jones, C. B. "Systematic Software Development using VDM," Prentice Hall Int'l, 2nd Ed., 1990.

[11] Woodman, M. and Heal, B. "Introduction to VDM," McGraw-Hill, 1993.

[12] Sheppard, D. "An introduction to formal specification with Z and VDM," McGraw-Hill, 1995.

[13] Spivey, J. "Understanding Z : A Specification Language and its Formal Semantics," Cambridge Univ. Press, 1988.

[14] Woodcock, J. C. P. & Davies, J., "Using Z : Specification, proof and refinement," Prentice Hall International, 1996.

[15] Wordsworth, J. B. "Software Development with Z : A Practical Approach to Formal Methods in Software Engineering," Addison-Wesley, 1992.

[16] 유문성, "정형적 명세언어인 Z과 VDM의 비교 연구", 평택대학교논문집 제10집 제2호, pp.247-256, 1998.

[17] Clement, T. "Comparing Approaches to Data Reification," in "FME'94 : Industrial Benefit of Formal Methods," pp. 118-133, Springer-Verlag, 1994.

[18] Jones, C. B. and Shaw, R., "Case studies in Systematic System Development," Prentice-Hall, 1990.

[19] Andrews, D., and Ince, D., "Transformational data refinement and VDM," Information and Software Technology, 37(11), pp.637-651, 1995.

[20] Morgan, C. "Programming From Specifications," Prentice-Hall, 1990.

[21] Morgan, C., Vickers, T. (eds.) "On the refinement calculus," Springer-Verlag, 1992.

[22] Bicarregui, J. C. and others "Proof in VDM; A practitioner's guide", Springer-Verlag, 1994.

유 문 성

e-mail : msyoo@mail.sangji.ac.kr

1978년 서울대학교 자연대학 수학과 (이학사)

1991년 미국인디아나대학 대학원 전산학과 (이학석사)

1996년 미국루이지애나주립대학 전산학과 (이학박사)

2000년~현재 상지대학교 컴퓨터정보공학부 전임강사

관심분야 : 소프트웨어 공학, 객체지향 시스템, 인터넷 소프트웨어