ROPAS
Research On Program Analysis System
National Creative Research Initiative Center
Korea Advanced Institute of Science and Technology

ROPAS MEMO
2003-20
2003년 4월 2일

트랜잭션 기반 데이터베이스 응용프로그램의 안전성 자동 검증 및 자동 튜닝 *

강현구 이광근
한국과학기술원 전산과
프로그램 분석 시스템 연구단
{hgkang; kwang}@ropas.kaist.ac.kr

2003년 4월 2일

요 약

본 논문에서는 프로그램 분석 기술에 기반하여, 주어진 프로그램 내에서 트랜잭션 처리 관련 오류를 자동으로 검출해주고, 성능저하 요소가 발견되면 자동으로 개선된 코드로 변환하여 주는 시스템을 제안한다.

트랜잭션 처리 오류란 트랜잭션을 열고서 닫지 않는 경우나, 잘못된 잠금수준(Locking-Level)을 설정하는 경우를 말한다. 전자의 경우, 원하는 대로 데이터가 저장되지 않거나 장시간 데이터베이스 테이블을 잠금(Locking)으로써 시스템 전체의 성능을 떨어뜨릴 수 있다. 후자의 경우, 시스템에 따라 예외상황이나 프로그램의 파행적 실행 중단을 야기한다.

비효율적인 트랜잭션 처리란, 트랜잭션 영역(Boundary) 또는 잠금수준을 비효율적으로 설계하여서 다른 프로세스들의 트랜잭션을 지연시키는 경우를 말한다.

1 문제 제기

트랜잭션을 기반으로 하는 데이터베이스 응용프로그램의 안전성 및 효율성(성능)은 매우 중요하다[1]. 이는 데이터베이스 트랜잭션 처리를 기반으로 하는 응용프로그램들이 은행/증권 시스템, 전사적 자원관리 시스템(ERP), 회사간 전산업무 처리 시스템(B2B), 통신망 시스템과 같은 주요 기간 정보시스템들의 핵심 부품을 이루고 있기 때문이다. 이러한 시스템들에서 버그나 성능 저하는 직접적이고 심각한 비용 손실을 초래할 수 있다.

잘 알려진 트랜잭션 처리 오류에는 다음과 같은 것들이 있다. 첫째, 닫히지 않는 트랜잭션은 문제가 될 수 있다. 아래의 예제 프로그램 1은 if문의 조건이 참이 아닌 경우에 대해 트랜잭션을 닫지 않는다.

[프로그램 1]

```
begin_transaction(Serializable);
balance = read();
b = balance - x ;
if (b > 0) {
```

*This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

```
    write(b);
    commit();
  }
  return;
```

이 경우 현재 프로세스가 종료할 때 까지 테이블에 대한 잠금을 유지하게 되어 시스템의 성능 저하를 유발한다. 또한, 종료시 트랜잭션 취소(rollback)가 자동으로 일어남으로서, 개발자의 의도와는 다르게 데이터의 갱신이 제대로 되지 않을 수 있는 문제가 있다. 따라서, 개발자는 모든 가능한 실행경로에 대하여 트랜잭션을 닫았는지 주의하며 프로그래밍 하여야 한다. 둘째, 읽기만 가능한 잠금수준의 트랜잭션 내에서 쓰기(write)를 사용하면 문제가 될 수 있다. 예를들어, 아래의 예제 프로그램 2는 트랜잭션을 시작할때, 다른 트랜잭션의 쓰기는 허용 안하고, 읽기만 허용하는 잠금수준인 읽기모드(Read_Only)를 설정하였다.

[**프로그램 2**]

```
begin_transaction(Read_Only);
balance = read();
if (balance > 0) {
  read();
}
else {
  write(1);
}
commit();
```

읽기모드에서는 데이터베이스에 쓰기를 수행하면 오류이다. 하지만, 예제에서는 if문의 조건이 참이 아닌 경우 쓰기를 수행 한다. 이는 트랜잭션 관리기(TP Monitor 또는 Middleware)와 프로그래밍 환경(프로그래밍 언어 또는 API)에 따라, 예외상황이나 프로그램의 파행적 실행 중단과 같은 문제를 야기한다.

잘 알려진 트랜잭션 기반 응용프로그램의 성능 관리 방법에는 다음과 같은 것들이 있다. 첫째, 트랜잭션은 가능한 짧게 끝내는 것이 좋다. 즉, 트랜잭션을 너무 오래 유지하면 다른 프로세스들의 트랜잭션이 해당 테이블에 접근할 수 없기 때문에 전체 시스템의 성능이 저하된다. 예를들어, 아래의 예제 프로그램 3은 트랜잭션과 아무 상관없는 show_msg라는 함수를 트랜잭션을 닫기 전에 호출함으로써 트랜잭션이 사용자의 입력 후에서야 종료된다.

[**프로그램 3**]

```
begin_transaction(Serializable);
read();
show_msg();
commit();
```

사용자가 오랫동안 응답하지 않으면 그동안 다른 프로세스들이 모두 관련 테이블에 접근할 수 없으므로, 전체시스템의 성능에 큰 장애가 될 수 있다. 따라서, 이 경우에는 트랜잭션종료(commit) 함수 호출을 show_msg 함수 호출 이전으로 옮겨 주면 좋다. 둘째, 트랜잭션 잠금수준은 가능한 낮게 설정해 주는것이 좋다. 즉, 현재의 트랜잭션이 실제로 쓰기를 하지 않는 경우 읽기모드로 설정해 주면 성능이 개선된다. 예를들

어, 프로그램 3은 잠금수준을 가장 높게(다른 트랜잭션의 읽기/쓰기 금지) 설정하였다. 하지만, 실제 트랜잭션은 읽기만 하고 있으므로 읽기모드로 설정할 수 있다. 예제에서 읽기가 시간이 많이 걸리는 질의(Query)이면, 상대적으로 성능이 많이 개선된다. 실제로 IBM에서는 자사의 WebSphere라는 미들웨어 제품를 사용하는 응용프로그램(EJB Component)들에 이와 같은 잠금수준 최적화를 적용한 결과, 성능(Request per Second)을 15% 이상 향상시킬 수 있었다[2].

한편, 이와 같은 트랜잭션 기반 응용프로그램에 대한 기존의 안전성 및 성능 관리는 테스팅과 모니터링 방법론 등에 기반하고 있다. 그런데, 일반적으로 테스팅 방법은 프로그램의 모든 실행 가능성을 완전히 검증 해주지 못하며, 모니터링 방법은 사전에 발생 가능한 문제를 예측할 수 없다. 이는 결국, 개발자 혹은 전문가의 능력에 전적으로 의존하는 방식이 되어버려서 방법론 자체의 신뢰성에 문제가 있게 된다. 반면, 정보시스템의 규모는 점점 방대해져서 개발자 혹은 전문가의 능력에 의존하는 기존의 안전성 및 성능관리는 점점 더 어려워지고 있다.

본 논문에서는 이와 같은 문제에 착안하여 프로그램 분석 기술에 기반한 자동화된 안전성 검증 및 성능 관리 시스템을 제안한다. 제안된 시스템은 주어진 프로그램을 분석하여 위의 예제에서 제시된 트랜잭션 처리 관련 오류와 성능저하 요소들을 자동으로 검출해주고, 개선된 코드로 변환(튜닝[3])하여 준다. 제안된 방법은 프로그램을 실행하기 전에 분석하여 오류를 예측하고 튜닝을 수행하기 때문에 장애가 발생하기 전에 미리 대처가 가능한 장점이 있다. 또한, 안전성이 증명된 방법으로 이러한 분석을 수행하기 때문에, 자동으로 수행된 오류 검증 및 튜닝결과를 완벽하게 신뢰할 수 있다는 장점이 있다.

논문의 순서는 다음과 같다. 2장에서는 분석 대상 언어를 소개한다. 3장에서는 제안된 언어에서 앞서 소개된 트랜잭션 오류를 검증해주는 타입시스템을 제안하고, 제안된 타입시스템이 올바름(Correctness)을 증명한다. 4장에서는 3장의 타입시스템에 대한 타입 유추 알고리즘을 제안하고, 제안된 타입 유추 알고리즘이 올바름을 증명한다. 5장에서는 각 트랜잭션을 분석해서 최적화된 잠금수준을 결정해주는 확장된 타입시스템을 보인다. 6장에서는 트랜잭션 영역을 최적화 하기 위하여 트랜잭션종료(commit 또는 rollback) 함수 호출을 안전한 수준에서 최대한 앞 당겨주는 튜닝 알고리즘을 제시한다. 그리고 7장에서는 관련 연구에 대하여 논의하고, 8장에서 결론을 맺겠다.

2 대상 언어와 그 의미

본 장에서는 분석 대상 언어인 TL(A Core Transaction Language)을 소개한다. TL은 트랜잭션 기반 응용프로그램 작성시 많이 쓰이는 명령형 언어의 핵심 부품을 추려내고, 프로그램의 트랜잭션 처리(함수호출 등)를 언어의 간단한 구문구조으로 요약한 언어이다. TL은 트랜잭션 기반 응용프로그램의 동작(Behavior)을 정형적으로 정의(Formal Definition)하고 추론(Reasoning) 할 수 있도록 고안되었다.

2.1 TL의 구문 구조

TL의 구문구조는 그림 1과 같다.

프로그램은 함수들의 선언에 이어 `main` 함수를 호출하는 형태를 가진다. 핵심 구문구조는 인자가 하나인 함수호출, 지역변수 선언(`let`), 조건문(`if`) 이다. 제시된 핵심 구문구조는 대부분의 명령형 언어의 구문구조를 표현 가능하다. 예를들어, 반복

$p \in Program$	::=	`fun` $f\ (x)$=e ;p	
	|	e	
$e \in Expression$	::=	v	value
	|	$f\ v$	function call
	|	`let` $x = e$ `in` e `end`	let
	|	`if` v `then` e `else` e	branch
	|	`open_ro`	open read only tr
	|	`open_rw`	open read write tr
	|	`close`	close transaction
	|	`read`	read data from DB
	|	`write` e	update DB
$v \in Value$	::=	c	constant
	|	x	variable
const	::=	`true` | `false` | `()` | *num*	
x			variable identifier
f			function identifier

그림 1: TL의 구문구조

문(Loop)은 재귀호출로, 연속문(Sequencing) 및 할당문(Assignment)은 `let`으로 표현된다.

트랜잭션 관련 명령은 읽기모드 트랜잭션 시작은 `open_ro`로, 읽기-쓰기 모드 트랜잭션 시작은 `open_rw`로 표현한다. 트랜잭션 종료는 `close` 명령을 사용한다. 데이터베이스 접근 명령어중 읽기는 `read`로 쓰기는 `write`로 표현한다. 실제의 프로그래밍 환경에서 트랜잭션 제어는 API 함수 호출 또는 포함된 질의언어(Embeded SQL)등을 통하여 표현된다. 트랜잭션 명령어는 이를 요약된 형태로 표현한 것이며 구체적인 프로그래밍 환경에 따라 해당 트랜잭션 제어 형태를 제시된 트랜잭션 명령어로 변환하여 분석한다.

본 논문에서는 차후 편의를 위해 선언되는 함수 이름은 겹치지 않음을 가정한다.

2.2 TL의 실행

TL 프로그램의 실행(Dynamic Semantics)은 그림 2와 같이 주어진 실행상태 (Configuration)에서 다음 실행상태로의 상태전이(Transition)로 정의된다. 실행상태는 전역함수환경(*Env*)과 현재의 트랜잭션 상태(*TC*), 현재 상태에서 실행할 프로그램(*Program*) 혹은 프로그램식(*Expresssion*), 그리고 남은 계산(*Continuation*)으로 이루어진다. 실행상태가 상태전이를 거치는 동안, 현재 실행할 식이 값(*Value*)이고, 더이상 남은 계산이 없으면 실행이 정상적으로 종료한다. 상태전이가 정의되지 않는 실행상태는 실행 오류(오류상태)를 뜻한다.

함수의 선언, 호출, `let`, `if` 등의 의미는 일반적인 명령형 언어의 실행과 같다. `open_ro`, `open_rw`는 현재의 트랜잭션 상태가 *closed*이면, 각각 *opened_ro*,*opened_rw*으로 트랜잭션 상태를 바꾼다. `close`는 현재의 트랜잭션 상태가 *opened_ro*이나 *opened_rw*일 때 유효하며 트랜잭션 상태를 *closed*로 바꾼다. `read`는 현재의 트랜잭션 상태가 *opened_ro*이거나, *opened_rw*일 때 유효하고, `write`는 현재의 트랜잭션 상태가 *opened_rw*일 때 유효하다.

$$
\begin{array}{rcll}
clos & \in & Closure & = & Id \times Expression & \text{closure} \\
\sigma & \in & Env & = & Id \xrightarrow{fin} Closure & \text{environment} \\
tc & \in & TC & ::= & opened\_ro \mid opened\_rw \mid closed & \text{transaction context} \\
K & \in & Continuation & ::= & \epsilon \mid (x, e) :: K &
\end{array}
$$

$$
\begin{array}{ll}
\text{[prog]} & (\sigma, tc, \texttt{fun}\ f\ (x)\texttt{=}e\ \texttt{;}p, K) \ \rightarrow\ (\sigma + \{f \mapsto \langle x, e\rangle\}, tc, p, K) \\
\text{[val]} & (\sigma, tc, v, (x, e) :: K) \ \rightarrow\ (\sigma, tc, e[v/x], K) \\
\text{[app]} & (\sigma, tc, f\ v, K) \ \rightarrow\ (\sigma, e[v/x], tc, K) \quad \text{if } \sigma(f) = \langle x, e\rangle \\
\text{[let]} & (\sigma, tc, \texttt{let}\ x = e_1\ \texttt{in}\ e_2, K) \ \rightarrow\ (\sigma, tc, e_1, (x, e_2) :: K) \\
\text{[ifT]} & (\sigma, tc, \texttt{if true then}\ e_1\ \texttt{else}\ e_2, K) \ \rightarrow\ (\sigma, tc, e_1, K) \\
\text{[ifF]} & (\sigma, tc, \texttt{if false then}\ e_1\ \texttt{else}\ e_2, K) \ \rightarrow\ (\sigma, tc, e_2, K) \\
\text{[open\_ro]} & (\sigma, tc, \texttt{open\_ro}, K) \ \rightarrow\ (\sigma, opened\_ro, (), K) \quad \text{if } tc = closed \\
\text{[open\_rw]} & (\sigma, tc, \texttt{open\_rw}, K) \ \rightarrow\ (\sigma, opened\_rw, (), K) \quad \text{if } tc = closed \\
\text{[close]} & (\sigma, tc, \texttt{close}, K) \ \rightarrow\ (\sigma, closed, (), K) \quad \text{if } tc = opened\_rw \text{ or } opened\_ro \\
\text{[read]} & (\sigma, tc, \texttt{read}, K) \ \rightarrow\ (\sigma, tc, (), K) \quad \text{if } tc = opened\_rw \text{ or } opened\_ro \\
\text{[write]} & (\sigma, tc, \texttt{write}, K) \ \rightarrow\ (\sigma, tc, (), K) \quad \text{if } tc = opened\_rw
\end{array}
$$

그림 2: TL의 실행

3 트랜잭션 타입시스템

본 장에서는 1장에서의 프로그램 1, 프로그램 2와 같이 주어진 프로그램에 트랜잭션 처리 오류가 있는지 검증해주는 분석 방법을 제안한다. 그리고 제안된 트랜잭션 타입 시스템의 올바름(안전성)을 증명한다.

3.1 타입시스템

본 절에서는 간단한 계층구조타입시스템(Subtype System)[4]을 사용하여 주어진 프로그램에 트랜잭션 처리 오류가 있는지 검증해 주는 분석 방법을 제안한다. 고안된 분석 방법은 일반적인 타입시스템의 특성에 따라 함수별로 따로 분석 가능(Modular)하다. 또한, 분석의 정확도를 위해 함수의 다형성(Polymorphism)과, 재귀호출의 다형성(Polymorphic Recursion)을 지원하도록 설계되었다.

트랜잭션 타입시스템[1]은 주어진 프로그램의 실행시 다음의 두가지 오류가 발생하는지 예측한다.

- 트랜잭션을 열었으면 이후 모든 실행 가능 경로에 대하여 닫아 주는지 검증한다.
- 읽기모드로 설정한 트랜잭션은 실제로 읽기만 하는지 검증한다.

타입 시스템에서 트랜잭션 상태(*TransactionStatus*) 및 트랜잭션 문맥 (*TransactionContext*)은 다음과 같이 정의된다.

[1] 차후 줄여서 타입시스템으로 표현한다

$s \in \mathit{TransactionStatus}$	::=	$\bot$	dead or non-termination
	|	Closed	closed
	|	OR	read possible
	|	ORW	read/write possible
	|	$\top$	can be all
$c \in \mathit{TransactionContext}$	::=	α	variable
	|	$\alpha \sqcup s$	joined TC(delayed join)
	|	s	atomic TC

트랜잭션 상태의 의미는 다음과 같다. *Closed*는 트랜잭션이 닫혔음을 뜻하고, OR은 `read`만 가능한 트랜잭션이 열렸음을 뜻한다. ORW는 `read`, `write`가 가능한 트랜잭션이 열렸음을 의미한다. $\bot$은 죽은 코드(Dead Code) 혹은 무한루프를 의미한다. $\top$은 트랜잭션이 열렸을수도 닫혔을 수도 있음을 뜻한다. 트랜잭션 문맥은 함수의 호출시 함수의 입력으로 주어지게 되는 트랜잭션 상태를 뜻하는 변수 α를 트랜잭션 상태에 도입한 것이다.

제약조건(*Constraints*) B는 다음과 같이 정의된다.

$$\begin{array}{rcl} cs \in \mathit{Constraint} & ::= & c \sqsubseteq c \\ B \in \mathit{Constraints} & = & \wp(\mathit{Constraint}) \end{array}$$

제약조건 B는 트랜잭션 문맥간의 순서 관계(*Constraint*)의 집합이다. 트랜잭션 문맥간의 순서관계는 작은것이(왼쪽) 큰것(오른쪽) 자리를 대치할 수 있음을 뜻한다. 예를 들어, `read`만 가능한 트랜잭션 문맥(OR)의 자리는 `read`, `write`가 모두 가능한 트랜잭션 문맥(ORW)으로 대치할 수 있다. 따라서, $ORW \sqsubseteq OR$과 같은 순서 관계가 성립한다. 이와 같은 의미에 따라, 트랜잭션 상태간의 기본 순서관계 Bas는 다음과 같이 정의된다.

$$Bas = \left\{ \begin{array}{l} \bot \sqsubseteq \mathit{Closed}, \bot \sqsubseteq OR, \bot \sqsubseteq ORW, \bot \sqsubseteq \top, \\ \mathit{Closed} \sqsubseteq \top, ORW \sqsubseteq OR, ORW \sqsubseteq \top, OR \sqsubseteq \top \end{array} \right\}$$

주어진 제약조건 가정 B에서 제약조건 B'이 만족하는지 결정하는 판별식 (Judgement)은 다음과 같은 형태를 가진다. [2]

$$B \vdash_B B'$$

그리고 "주어진 제약조건 가정 B에서 제약조건 B'는 성립한다." 로 읽는다. $B \vdash B'$ 결정 규칙은 그림 3과 같다.

함수타입(*TypeScheme*)및 타입환경(*TypeEnvironment*)은 다음과 같이 정의된다.

$$\begin{array}{rcl} ts \in \mathit{TypeScheme} & ::= & \forall \alpha \text{ with } B_f.\alpha \rightarrow c_f \\ \Gamma \in \mathit{TypeEnvironment} & = & Id \rightarrow \mathit{TypeScheme} \end{array}$$

함수타입의 직관적 의미는 해당 함수 호출시 함수의 입력 트랜잭션 문맥에 대한 제약조건 B_f가 만족되면, c_f가 함수 호출의 결과 트랜잭션 문맥이 된다는 뜻이다. 이를 다시 자세히 살펴보면 다음과 같다. 우선 호출되는 함수의 제약조건 B_f의 내의 변

[2] 판별식의 타입은 차후 생략한다.

$$(hypoth) \quad B \vdash c_1 \sqsubseteq c_2 \quad \text{if } c_1 \sqsubseteq c_2 \in B \text{ or } c_1 \sqsubseteq c_2 \in Bas$$

$$(reflex) \quad B \vdash c \sqsubseteq c$$

$$(trans) \quad \frac{B \vdash c_1 \sqsubseteq c_2 \quad B \vdash c_2 \sqsubseteq c_3}{B \vdash c_1 \sqsubseteq c_3}$$

$$(seq) \quad \frac{\forall cs \in B'.B \vdash cs}{B \vdash B'}$$

그림 3: 제약조건성립 결정규칙(Constraint Rule)

수 α를 입력 트랜잭션 문맥(c_{in})으로 치환(Substitution)한 결과는 호출시점의 제약조건 가정에서 성립해야 한다. 여기서 α를 c_{in}으로의 치환 S는 $[c_{in}/\alpha]$과 같이 표기한다. 예를들어, 함수의 제약조건이 $B_f = \{\alpha \sqsubseteq OR\}$이고 호출시 입력 트랜잭션 문맥 c_{in}이 ORW이면, $Bas \vdash ORW \sqsubseteq OR$이 되므로 성립한다. 또한, 함수 호출후에는 $c_f[c_{in}/\alpha]$가 결과 트랜잭션 문맥이 된다.

만약 제약조건 B'에 대해 어떤 치환 S를 적용하였을 때 제약조건 가정 B에서 성립하면, "제약조건 B'이 B에서 만족가능(Satisfiable)하다"라고 하고 $B \models B'$으로 표기한다.

$$\frac{\exists S.\ B \vdash B'S}{B \models B'}$$

또한, 제약조건 가정 B'이 $\emptyset$이면 아래와 같이 줄여서 표현한다.

$$\vdash B \ \equiv \ \emptyset \vdash B$$
$$\models B \ \equiv \ \emptyset \models B$$

이제 우리의 최종 트랜잭션 타입 결정 판별식(Judgement)[3]은 다음과 같은 형태로 정의된다.

$$\Gamma, B, c \vdash_p \ p : c'$$
$$\Gamma, B, c \vdash_e \ e : c'$$

그리고 "주어진 타입환경 Γ, 제약조건 가정 B, 입력 트랜잭션 문맥 c에서, 프로그램 p 혹은 프로그램식 e를 유추한 결과 트랜잭션 문맥은 c'이다" 또는 "주어진 타입환경 Γ, 제약조건 가정 B, 입력 트랜잭션 문맥 c에서, 프로그램 p 혹은 프로그램식 e는 c'으로 타이핑 된다" 라고 읽는다. 여기서 우리는 초기 제약조건 가정 B는 항상 만족가능하게 주어짐을 가정한다.

주어진 판별식에 따른 트랜잭션 타입 결정 규칙은 그림 4와 같다.

함수가 타이핑 되려면 함수 자신에 대한 타이핑을 추가한 타입환경과, B_f라는 제약조건 가정, 그리고 α라는 입력 트랜잭션 문맥하에서 함수의 몸통(Body)을 유추하여 c_f로 타이핑 되어야 한다. 여기서 함수 자신에 대한 타이핑을 일반화(Generalize)하고 함수의 몸통을 유추하기 때문에 다형(Polymorphic) 재귀 호출이 가능해 진다. 또한, 제약조건 가정 B_f는 만족가능해야 하며, 함수의 몸통을 타이핑하기 위해 필요한 가정의 모음이다.

[3]차후 판별식의 타입은 생략하여 중복표현(overloaded)한다.

[Program] $$\frac{\models B_f \quad \Gamma \cup \{f : \forall\alpha \text{ with } B_f.\alpha \to c_f\}, B_f, \alpha \vdash e : c_f \quad \Gamma \cup \{f : \forall\alpha \text{ with } B_f.\alpha \to c_f\}, B, c \vdash p : c'}{\Gamma, B, c \vdash \texttt{fun}\, f\ (x){=}e\ ; p : c'}$$

[Value] $$\frac{}{\Gamma, B, c \vdash v : c}$$

[App] $$\frac{\Gamma(f) = \forall\alpha \text{ with } B_f.\alpha \to c_f \quad B \vdash\ B_f[c/\alpha]}{\Gamma, B, c \vdash f\ v : c_f[c/\alpha]}$$

[Let] $$\frac{\Gamma, B, c \vdash e_1 : c_1 \qquad \Gamma, B, c_1 \vdash e_2 : c_2}{\Gamma, B, c \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : c_2}$$

[If] $$\frac{\Gamma, B, c \vdash e_1 : c_1 \qquad \Gamma, B, c \vdash e_2 : c_2}{\Gamma, B, c \vdash \texttt{if}\ \ v\ \ \texttt{then}\ \ e_1\ \ \texttt{else}\ \ e_2 : c_1 \sqcup c_2}$$

[Open_RO] $$\frac{B\ \vdash\ c \sqsubseteq \mathit{Closed}}{\Gamma, B, c \vdash \texttt{open\_ro} : \mathit{OR}}$$

[Open_RW] $$\frac{B\ \vdash\ c \sqsubseteq \mathit{Closed}}{\Gamma, B, c \vdash \texttt{open\_rw} : \mathit{ORW}}$$

[Close] $$\frac{B\ \vdash\ c \sqsubseteq \mathit{OR}}{\Gamma, B, c \vdash \texttt{close} : \mathit{Closed}}$$

[Read] $$\frac{B\ \vdash\ c \sqsubseteq \mathit{OR}}{\Gamma, B, c \vdash \texttt{read} : c}$$

[Write] $$\frac{B\ \vdash\ c \sqsubseteq \mathit{ORW}}{\Gamma, B, c \vdash \texttt{write} : c}$$

그림 4: 타입결정규칙(Typping Rules)

함수 호출에서는 호출되는 함수의 입력 제약조건 B_f가 주어진 입력 트랜잭션 문맥에서 성립하는지 검사하고, 입력 트랜잭션 문맥에 따라 결과 트랜잭션 문맥을 결정한다.

값은 트랜잭션 문맥을 변경시키지 않는다. `let`은 실행흐름에 따라 트랜잭션 문맥을 전달한다. `if`는 양쪽 분기에서 나올수 있는 두가지 트랜잭션 문맥을 합치기(Join : $\sqcup$)을 통해 요약한다.

`open_ro`와 `open_rw`은 입력 트랜잭션이 $Closed$인지 검사하고 각각 OR, ORW를 결과로 준다. `close`는 입력 트랜잭션 문맥이 열려있는지 검사하고 $Closed$를 결과로 준다. `read`와 `write`는 각각 입력 트랜잭션 문맥 c가 OR, ORW에 적용가능 한지 검사한 후, c를 결과로 준다. 트랜잭션 관련 명령들은 각각 다음과 같은 타입을 가지는 함수의 호출로 생각할 수 있다.

$$\begin{array}{l}\text{open\_ro} : \forall\alpha \text{ with } \{\alpha \sqsubseteq Closed\}.\alpha \to OR\\ \text{open\_rw} : \forall\alpha \text{ with } \{\alpha \sqsubseteq Closed\}.\alpha \to ORW\\ \text{close} : \forall\alpha \text{ with } \{\alpha \sqsubseteq OR\}.\alpha \to Closed\\ \text{read} : \forall\alpha \text{ with } \{\alpha \sqsubseteq OR\}.\alpha \to \alpha\\ \text{write} : \forall\alpha \text{ with } \{\alpha \sqsubseteq ORW\}.\alpha \to \alpha\end{array}$$

3.2 타입시스템의 올바름 증명

본 절에서는 타입시스템의 안전성을 증명한다. 타입시스템의 안전성이란 타입시스템을 통과한 프로그램을 실제로 실행하면 트랜잭션 처리 오류가 발생하지 않음을 말한다. [4]

타입시스템의 안전성 증명은 아래의 두가지의 보조정리를 증명하는 것으로 충분[5]하다.

- 타이핑보존(Subject Reduction) : 타입시스템을 통과한 프로그램(실행상태)이 한 단계 실행되면(상태전이), 그 결과는 타입시스템을 통과한다.
- 진행(Progress) : 타입시스템을 통과한 프로그램은 문제없이 실행 될 수 있다.

위의 두가지 증명에 앞서 몇가지 정의와 보조정리들의 증명이 필요하다.

정의 1 [실행환경 타이핑]

1. $\vdash \emptyset : \emptyset$
2. $\vdash \sigma + \{f \mapsto \langle x, e\rangle\} : \Gamma \cup \{f : \forall\alpha \text{ with } B_f.\alpha \to c_f\}$
   iff $\vdash \sigma : \Gamma$ and $\models B_f$ and $\Gamma \cup \{f : \forall\alpha \text{ with } B_f.\alpha \to c_f\}, B_f, \alpha \vdash e : c_f$

실행환경 타이핑은 실행환경과 타입환경 사이의 타입주기 관계이다. 실행환경은 실행환경에 바인딩 되어 있는 각 함수들을, 타입시스템으로 타이핑한 결과로 정의한다.

정의 2 [남은 계산 타이핑]

1. $\Gamma, B, c \vdash \epsilon$

[4] 개발자 입장에서는 활용한 검증도구의 안전성이 보장되면, 개발한 프로그램의 트랜잭션 처리 신뢰성에 대해 고객에게 품질 보장을 해줄 수 있게 된다.

2. $$\frac{\Gamma, B, c \vdash\ e\ :\ c' \quad \Gamma, B, c' \vdash\ K}{\Gamma, B, c \vdash\ (x, e) :: K}$$

남은 계산 타이핑은 현재 실행상태에서 남은 계산에 대한 타입주기 관계이다. 남은 계산은 차후 수행되어야할 일(프로그램식)들의 리스트를 타입시스템을 통해 차례로 타이핑한 결과로 정의한다.

정의 3 [트랜잭션 상태 타이핑 : $B \vdash tc \leq c$**]**

1. $B \vdash open\_ro \leq OR$
2. $B \vdash open\_rw \leq ORW$
3. $B \vdash closed \leq Closed$
4. If $B \vdash tc \leq c$ and $B \vdash c \sqsubseteq c'$, then $B \vdash tc \leq c'$

트랜잭션 상태 타이핑은 실행 트랜잭션 상태와 타입시스템의 트랜잭션 문맥 간의 타입 주기 관계이다.

정의 4 [실행상태 타이핑]

$$\frac{\vdash\ \sigma\ :\ \Gamma \quad B \vdash tc \leq c \quad \Gamma, B, c \vdash\ p\ :\ c' \quad \Gamma, B, c' \vdash\ K}{\vdash\ (\sigma, tc, p, K)}$$

$$\frac{\vdash\ \sigma\ :\ \Gamma \quad B \vdash tc \leq c \quad \Gamma, B, c \vdash\ e\ :\ c' \quad \Gamma, B, c' \vdash\ K}{\vdash\ (\sigma, tc, e, K)}$$

실행상태 타이핑은 임의의 제약조건 가정 B, 실행환경의 타입 Γ, 트랜잰션 문맥 c에서 해당 프로그램식의 타입을 결정(c')할 수 있고, 결과 타입 c' 으로 남은 계산을 타이핑할 수 있음을 의미한다. 다시말해, 주어진 프로그램이 타입시스템을 통과할 수 있음을 의미한다.

보조정리 1 [제약조건 판별식 치환1] *$B \vdash B'$이면 $BS \vdash B'S$이다.*

(증명) $B \vdash B'$에 대한 귀납증명(induction)

- (hypoth)인 경우 : 여기서 B'이 $c_1 \sqsubseteq c_2$이라 가정하자. 그러면, $c_1 \sqsubseteq c_2 \in B$ 또는 $c_1 \sqsubseteq c_2 \in Bas$ 이다. 그러므로, $BS \vdash B'S$의 성립은 자명하다.
- (reflex)인 경우 : 치환후에도 계혹 (reflex)를 만족함은 자명하다.
- (trans)또는 (seq)인 경우 : 간단한 귀납증명으로 증명된다.

보조정리 2 [트랜잭션 문맥 치환] *임의의 트랜잭션 문맥 c_0, c와 임의의 치환 S에 대해, $c_0[cS/\alpha] = c_0[c/\alpha]S$ 가 성립한다.*

(증명) c_0에 대한 귀납증명

- $c_0 = s$인 경우 : 좌변 $= c_0$, 우변 $= c_0$

- $c_0 = \alpha$인 경우: 좌변 $= cS$, 우변 $= cS$
- $c_0 = \alpha \sqcup s$인 경우 : 좌변 $= cS \sqcup s$, 우변 $= (c \sqcup s)S = cS \sqcup s$

보조정리 3 [제약조건결정식 치환2] *$B \vdash B'[c/\alpha]$이면, $BS \vdash B'[cS/\alpha]$이다.*

(증명) 보조정리 1, 2의 스타일로 간단히 증명되므로 자세한 과정은 생략한다.

보조정리 4 [판별식 치환] *$\Gamma, B, c \vdash e : c'$이고 $\models BS$이면, $\Gamma, BS, cS \vdash e : c'S$이다.*

(증명) $\Gamma, B, c \vdash e : c$의 증명트리에 대한 귀납증명

- $e = v$인 경우 : [Value] 규칙에 의해 명백함
- $e = f\ v$인 경우 :
  가정과 [App] 규칙에 따라, (1) $\Gamma(f) = \forall\alpha$ with $B_f.\alpha \rightarrow c_f$, (2) $B \vdash B_f[c/\alpha]$, (3) $c' = c_f[c/\alpha]$가 성립한다. (2)와 보조정리 3에 의해, (4) $BS \vdash B_f[cS/\alpha]$가 성립한다. (3)과 보조정리 2에 의해 (5) $c'S = c_f[c/\alpha]S = c_f[cS/\alpha]$가 성립한다. (1), (4), (5)와 [App]규칙에 따라 $\Gamma, BS, cS \vdash f\ v : c'S$가 성립한다.
- $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$인 경우 :
  가정과 [Let] 규칙에 따라 (1) $\Gamma, B, c \vdash e_1 : c_1$, (2) $\Gamma, B, c_1 \vdash e_2 : c'$가 성립한다. (1), (2)와 귀납증명법에 따라 (3) $\Gamma, BS, cS \vdash e_1 : c_1S$, (4) $\Gamma, BS, c_1S \vdash e_2 : c'S$가 성립한다. 그러므로, (3), (4)와 [Let] 규칙에 따라, $\Gamma, BS, cS \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : c'S$가 성립한다.
- $e = \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$인 경우 : `let`과 마찬가지의 귀납증명으로 간단히 증명된다.
- $e =$ `open_ro`인 경우 :
  가정과 [Open_RO] 규칙에 따라 (1) $B \vdash c \sqsubseteq Closed$가 성립한다. (1)과 보조정리 1에 따라 (2) $BS \vdash cS \sqsubseteq Closed$가 성립한다. 그러므로, (2)와 [Open_RO] 규칙에 따라 $\Gamma, BS, cS \vdash$ `open_ro` $: OR$ 가 성립한다.
- 다른 트랜잭션 명령의 경우들도 `open_ro`와 비슷한 귀납증명으로 간단히 증명된다.

보조정리 5 [제약조건 확장] *$\Gamma, B', c_1 \vdash e : c_2$이고 $B \vdash B'$이면, $\Gamma, B, c_1 \vdash e : c_2$이다.*

(증명) $B \vdash B'$이고 $B' \vdash B''$이면 $B \vdash B''$라는것을 보임으로써 충분하다. 왜냐하면, 타입규칙에서 우리가 $B' \vdash B''$를 결정할 때마다 $B \vdash B''$를 결정할 수 있기 때문이다. 그리고, 이는 $B' \vdash B''$의 증명트리에 대한 귀납증명으로 간단하게 증명된다.

보조정리 6 [바꿔치기] *$\vdash (\sigma, tc, e, K)$이면 $\vdash (\sigma, tc, e[v/x], K)$이다.*

(증명) 변수와 값의 타입결정규칙은 같다.

보조정리 7 [판별식 완화] *$\Gamma, B, c_1 \vdash e : c_2$이고 $B \vdash c_1' \sqsubseteq c_1$이면, $B \vdash c_2' \sqsubseteq c_2$를 만족하는 임의의 트랜잭션 문맥 c_2'로 $\Gamma, B, c_1' \vdash e : c_2'$ 형태의 타이핑이 존재한다.*

(증명) $\Gamma, B, c_1 \vdash e : c_2$의 증명트리에 대한 귀납증명

- [Value], [Let], [If]인 경우 : 간단한 귀납증명으로 증명된다.
- [App]인 경우 : B_f에서 변수는 하나 뿐이기 때문에, B_f는 항상 $\alpha \sqsubseteq c$ 형태로 정리될 수 있다. 그러므로, [App] 규칙에 의해 $\Gamma, B, c_1 \vdash e : c_2$이면 $\Gamma, B, c_1' \vdash e : c_2'$가 성립한다.
- [Open_RO], [Open_RW], [Close], [Read], [Write] : 타입결정규칙에 의하여 자명하다.

보조정리 8 [타이핑보존] $\vdash C$ *이고* $C \rightarrow C'$ *이면* $\vdash C'$ *이다.*

(증명) 실행상태 C의 구조에 대한 귀납증명

- C $= (\sigma, tc, \texttt{fun}\, f\ (x)\texttt{=}e\ \texttt{;}p, K)$인 경우 :
  가정에 의해 (1) $(\sigma, tc, \texttt{fun}\, f\ (x)\texttt{=}e\ \texttt{;}p, K) \rightarrow (\sigma + \{f \mapsto \langle x, e\rangle\}, tc, p, K)$, (2) $\vdash (\sigma, tc, \texttt{fun}\ \ f\ (x)\texttt{=}e\ \texttt{;}p, K)$ 임을 알 수 있다. (2)와 정의 4에 따라 (3) $\vdash \sigma : \Gamma$, (4) $B \vdash tc \leq c$, (5) $\Gamma, B, c \vdash \texttt{fun}\, f\ (x)\texttt{=}e\ \texttt{;}p : c'$, (6) $\Gamma, B, c' \vdash K$ 이다. (5)와 [Program] 규칙에 의해 (7) $\models B_f$, (8) $\Gamma \cup \{f : \forall\alpha \text{ with } B_f.\alpha \rightarrow c_f\}, B_f, \alpha \vdash e : c_f$ (9) $\Gamma \cup \{f : \forall\alpha \text{ with } B_f.\alpha \rightarrow c_f\}, B, c \vdash p : c'$ 이다. (3), (7), (8)과 정의 1에 의해서 (10) $\vdash \sigma + \{f \mapsto \langle x, e\rangle\} : \Gamma + \{f : \forall\alpha \text{ with } B_f.\alpha \rightarrow c_f\}$ 가 성립한다. 그러므로, (10), (4), (9), (6)과 정의 4에 따라 $\vdash (\sigma + \{f \mapsto \langle x, e\rangle\}, tc, p, K)$가 성립한다.
- C $= (\sigma, tc, v, K)$인 경우 :
  가정과 정의 4에 의해 (1) $(\sigma, tc, v, (x, e) :: K') \rightarrow (\sigma, tc, e[v/x], K')$, (2) $\vdash \sigma : \Gamma$, (3) $B \vdash tc \leq c$, (4) $\Gamma, B, c \vdash v : c$, (5) $\Gamma, B, c \vdash (x, e) :: K'$ 이다. (5)와 정의 2에 의해 (6) $\Gamma, B, c \vdash e : c'$, (7) $\Gamma, B, c' \vdash K'$ 이다. (2), (3), (6), (7)과 정의 4에 의해 (8) $\vdash (\sigma, tc, e, K)$ 이다. 그러므로, 보조정리 6에 의해 $\vdash (\sigma, tc, e[v/x], K)$가 성립한다.
- C $= (\sigma, tc, f\ v, K)$인 경우 :
  가정에 의해 (1) $(\sigma, tc, f\ v, K) \rightarrow (\sigma, tc, e[v/x], K)$, (2) $\vdash \sigma : \Gamma$, (3) $B \vdash tc \leq c$, (4) $\Gamma, B, c \vdash f\ v : c'$, (5) $\Gamma, B, c' \vdash K$ 이다. (4)와 [App] 규칙에 의해 (6) $\Gamma(f) = \forall\alpha \text{ with } B_f.\alpha \rightarrow c_f$, (7) $B \vdash B_f[c/\alpha]$, (8) $c' = c_f[c/\alpha]$ 이다. (2)와 정의 1에 의해 (9) $\Gamma, B_f, \alpha \vdash e : c_f$ 이다. (7)에 의해 (10) $\models B_f[c/\alpha]$이다. (8), (9), (10)과 보조정리 4에 의해 (11) $\Gamma, B_f[c/\alpha], c \vdash e : c'$ 가 성립한다. (11), (7)과 보조정리 5에 의해 (12) $\Gamma, B, c \vdash e : c'$ 가 성립한다. (2), (3), (12), (5)와 정의 4에 의해 (13) $\vdash (\sigma, tc, e, K)$이 성립한다. 그러므로, (13)과 보조정리 6에 의해 $\vdash (\sigma, tc, e[v/x], K)$ 가 성립한다.
- C $= (\sigma, tc, \texttt{let}\ x = e_1\ \texttt{in}\ e_2, K)$인 경우 :
  가정에 의해 (1) $(\sigma, tc, \texttt{let}\ x = e_1\ \texttt{in}\ e_2, K) \rightarrow (\sigma, tc, e_1, (x, e_2) :: K)$, (2) $\vdash \sigma : \Gamma$, (3) $B \vdash tc \leq c$, (4) $\Gamma, B, c \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : c'$, (5) $\Gamma, B, c' \vdash K$ 이다. (4)와 [let] 규칙에 의해 (6) $\Gamma, B, c \vdash e_1 : c_1$, (7) $\Gamma, B, c_1 \vdash e_2 : c'$ 이다. (7), (5)과 정의 2에 의해 (8) $\Gamma, B, c_1 \vdash (x, e) :: K$ 가 성립한다. 그러므로, (2), (3), (6), (8)과 정의 4에 의해 $\vdash (\sigma, tc, e_1, (x, e_2) :: K)$ 가 성립한다.
- C $= (\sigma, tc, \texttt{if true then}\, e_1\, \texttt{else}\, e_2, K)$인 경우 :
  가정에 의해 (1) $(\sigma, tc, \texttt{if true then}\, e_1\, \texttt{else}\, e_2, K) \rightarrow (\sigma, tc, e_1, K)$, (2) $\vdash \sigma : \Gamma$, (3) $B \vdash tc \leq c$, (4) $\Gamma, B, c \vdash \texttt{if true then}\, e_1\, \texttt{else}\, e_2 : c'$, (5) $\Gamma, B, c' \vdash K$ 이다.

(4)와 [If] 규칙에 의해 (6) $\Gamma, B, c \vdash e_1 : c_1$, (7) $\Gamma, B, c \vdash e_2 : c_2$, (8) $c' = c_1 \sqcup c_2$ 이다. (5), (8)과 보조정리 7, 그리고 정의 2에 따라 (8) $\Gamma, B, c_1 \vdash K$ 가 성립한다. 그러므로, (2), (3), (6), (8)과 정의 4에 따라 $\vdash (\sigma, tc, e_1, K)$ 가 성립한다.

- C $= (\sigma, tc, \texttt{if false then}\, e_1\, \texttt{else}\, e_2, K)$인 경우 :
  $(\sigma, tc, \texttt{if true then}\, e_1\, \texttt{else}\, e_2, K)$의 경우와 비슷한 귀납증명으로 증명된다.

- C $= (\sigma, tc, \texttt{open_ro}, K)$인 경우 :
  가정에 의해 (1) $(\sigma, tc, \texttt{open\_ro}, K) \rightarrow (\sigma, \mathit{opened\_ro}, (), K)$, (2) $\vdash \sigma : \Gamma$, (3) $B \vdash tc \leq c$, (4) $\Gamma, B, c \vdash \texttt{open\_ro} : c'$, (5) $\Gamma, B, c' \vdash K$ 이다. (4)와 [Open_RO] 규칙에 의해 (6) $B \vdash c \sqsubseteq \mathit{Closed}$, (7) $c' = \mathit{OR}$ 가 성립한다. 정의 3에 의해 (8) $\mathit{opened\_ro} \leq \mathit{OR}$ 이다. [Value] 규칙에 의해 (9) $\Gamma, B, \mathit{OR} \vdash () : \mathit{OR}$ 가 성립한다. 그러므로, (2), (8), (9), (7), (5)와 정의 3에 의해 $\vdash (\sigma, \mathit{opened\_ro}, (), K)$ 가 성립한다.

- C $= (\sigma, tc, \texttt{open_rw}, K)$인 경우 :
  $(\sigma, tc, \texttt{open_ro}, K)$의 경우와 비슷하게 증명된다.

- C $= (\sigma, tc, \texttt{close}, K)$인 경우 :
  가정에 의해 (1) $(\sigma, tc, \texttt{close}, K) \rightarrow (\sigma, \mathit{closed}, (), K)$, (2) $\vdash \sigma : \Gamma$, (3) $B \vdash tc \leq c$, (4) $\Gamma, B, c \vdash \texttt{close} : c'$, (5) $\Gamma, B, c' \vdash K$ 이다. (4)와 [Close] 규칙에 의해 (6) $B \vdash c \sqsubseteq \mathit{OR}$, (7) $c' = \mathit{Closed}$ 가 성립한다. 정의 3에 의해 (8) $\mathit{close} \leq \mathit{Closed}$ 이다. [Value] 규칙에 의해 (9) $\Gamma, B, \mathit{Closed} \vdash () : \mathit{Closed}$ 가 성립한다. 그러므로, (2), (8), (9), (7), (5)과 정의 4에 의해 $\vdash (\sigma, \mathit{closed}, (), K)$ 가 성립한다.

- C $= (\sigma, tc, \texttt{read}, K)$인 경우 :
  가정에 의해 (1) $(\sigma, tc, \texttt{read}, K) \rightarrow (\sigma, tc, (), K)$, (2) $\vdash \sigma : \Gamma$, (3) $B \vdash tc \leq c$, (4) $\Gamma, B, c \vdash \texttt{read} : c'$, (5) $\Gamma, B, c' \vdash K$ 이다. (4)와 [Read] 규칙에 의해 (6) $B \vdash c \sqsubseteq \mathit{OR}$, (7) $c' = c$ 이다. [Vaue] 규칙에 의해 (8) $\Gamma, B, c \vdash () : c$ 가 성립한다. 그러므로 (2), (3), (8), (7), (5)와 정의 4에 의해 $\vdash (\sigma, tc, (), K)$ 가 성립한다.

- C $= (\sigma, tc, \texttt{write}, K)$인 경우 :
  $(\sigma, tc, \texttt{read}, K)$의 경우와 비슷하게 증명된다.

보조정리 9 [진행] $\vdash C$*이면* $C \rightarrow C'$*인 상태전이가 존재한다.*

(증명) 우리는 증명의 편의를 위해 본 보조정리의 대우(Contraposition)인 "실행상태 C가 오류가 있으면 $\vdash C$가 되는 Γ, B, c, c' 가 존재하지 않음"을 증명한다. 또한, 전통적인 타입검사는 이미 통과한 프로그램을 가정한다. 따라서, 다음 5가지 오류 상태의 가능성만 존재한다.

- $C = (\sigma, tc, \texttt{open_ro}, K)$이고 tc가 *opened_ro* 또는 *opened_rw*인 경우 :
  먼저 $\vdash (\sigma, tc, \texttt{open_ro}, K)$인 타이핑이 존재한다고 가정하자. 그러면, 정의 4에 의해 $\vdash \sigma : \Gamma$, $B \vdash tc \leq c$, $\Gamma, B, c \vdash \texttt{open\_ro} : c'$, $\Gamma, B, c' \vdash K$ 가 되는 타이핑이 존재한다. 그러면, [Open_RO] 규칙에 의해 $c \sqsubseteq \mathit{Closed}$가 성립한다. 이는 $B \vdash tc \leq c$라는 가정에 모순이다. 따라서, 이와 같은 경우는 타이핑 되지 않는다.

- $C = (\sigma, tc, \texttt{open_rw}, K)$이고 tc가 *opened_ro* 또는 *opened_rw*인 경우 :
  open_ro의 경우와 비슷하게 증명된다.

- $C = (\sigma, tc, \texttt{close}, K)$이고, tc가 *closed*인 경우 :
  먼저 $\vdash (\sigma, tc, \texttt{close}, K)$인 타이핑이 존재한다고 가정하자. 그러면, 정의 4에 의해 $\vdash \sigma : \Gamma$, $B \vdash tc \leq c$, $\Gamma, B, c \vdash \texttt{close} : c'$, $\Gamma, B, c' \vdash K$ 가 되는 타이핑이 존재한다. 그러면, [Close] 규칙에 의해 $c \sqsubseteq OR$가 성립한다. 이는 $B \vdash tc \leq c$라는 가정에 모순이다.

- $C = (\sigma, tc, \texttt{read}, K)$이고, tc는 *closed*인 경우 :
  먼저 $\vdash (\sigma, tc, \texttt{read}, K)$인 타이핑이 존재한다고 가정하자. 그러면, 정의 4에 의해 $\vdash \sigma : \Gamma$, $B \vdash tc \leq c$, $\Gamma, B, c \vdash \texttt{read} : c'$, $\Gamma, B, c' \vdash K$ 가 되는 타이핑이 존재한다. 그러면, [Read] 규칙에 의해 $c \sqsubseteq OR$가 성립한다. 이는 $B \vdash tc \leq c$라는 가정에 모순이다.

- $C = (\sigma, tc, \texttt{write}, K)$이고, tc는 *opened_ro* 또는 *closed* 인 경우 :
  먼저 $\vdash (\sigma, tc, \texttt{write}, K)$인 타이핑이 존재한다고 가정하자. 그러면, 정의 4에 의해 $\vdash \sigma : \Gamma$, $B \vdash tc \leq c$, $\Gamma, B, c \vdash \texttt{write} : c'$, $\Gamma, c' \vdash K$인 타이핑이 존재한다. 그러면, [Write] 규칙에 의해 $c \sqsubseteq ORW$가 성립한다. 이는 $B \vdash tc \leq c$라는 가정에 모순이다.

결론적으로 보조정리 8, 9에 의해 다음의 정리를 얻는다.

정리 1 [타입시스템의 안전성] $\vdash C$ *이면* C*를 실행하여* C'*으로 끝나면* $\vdash C$ *이다*

4 트랜잭션 타입 유추 알고리즘

본 장에서는 3장에서 제시된 증명시스템(타입시스템)에 맞추어, 트랜잭션 타입을 유추해 주는 알고리즘 [5]을 제시한다 . 그리고, 제시된 알고리즘이 올바름을 보인다.

4.1 타입 유추 알고리즘 : I

타입 유추 알고리즘 I [6]는 다음과 형태를 가진다.

$$\begin{aligned} I_p \ : \ & \mathit{TypeBinding} \times \mathit{Constraints} \times \mathit{TransactionContext} \times \mathit{Program} \\ & \rightarrow \mathit{TransactionContext} \times \mathit{Constraints} \\ I_e \ : \ & \mathit{TypeBinding} \times \mathit{Constraints} \times \mathit{TransactionContext} \times \mathit{Expression} \\ & \rightarrow \mathit{TransactionContext} \times \mathit{Constraints} \end{aligned}$$

타입 유추 알고리즘은 프로그램 p 또는 프로그램식 e가 주어진 타입환경 Γ, 제약조건 가정 B, 입력 트랜잭션 문맥 c에서 타이핑 가능하면, 이를 위해 필요한 제약조건 B'과 결과 트랜잭션 문맥 c'을 찾아준다. 타입 유추 알고리즘은 그림 6,그림 7과 같다.

함수의 타입은 먼저 함수 자신의 타입을 가장 작은 함수타입인 $\forall\alpha \,\text{with}\, \alpha \sqsubseteq \top.\alpha \rightarrow \bot$ 로 가정하고 함수의 몸통을 유추하여, 함수의 몸통을 타이핑 하기 위해 필요한 제약조건 B'과 결과 트랜잭션 문맥 c_o 를 찾는다. 이 결과를 바탕으로 다시 함수의 타입을 $\forall\alpha$ with $B'.\alpha \rightarrow c_o$로 놓고 함수의 몸통의 타입을 유추한다. 이 과정에서 고정점(Fixpoint)에 도달하면 함수의 타입을 결정한다. 이때, 매 과정(Iteration)마다 B'에 그림 5의 간단화(Simplify)를 적용하여 알고리즘의 효율을 높인다. 값은 입력으로 받

[5]유추알고리즘의 필요성은 참고문헌 [6]을 참고하기 바란다.
[6]차후 I의 타입은 편의상 생략하고 중복 표현한다

for each constraint $c_1 \sqsubseteq c_2$ of B' do {

case (c_1,c_2) of

- (α, s) : do nothing
- $(\alpha \sqcup s_1, s_2)$: if $\vdash s_1 \sqsubseteq s_2$ then change it to $\alpha \sqsubseteq s_2$ else fail
- otherwise : fail

}
for each constraint pair $\alpha \sqsubseteq s_1$, $\alpha \sqsubseteq s_2$ of modified B' in previous step do
{

combine it to $\alpha \sqsubseteq s_1 \sqcap s_2$

}

그림 5: 제약조건 간단화(Simplify)

은 트랜잭션 문맥을 그대로 결과로 준다. 함수 호출은 먼저 호출되는 함수의 입력 제약조건 B_f의 α에 주어진 입력 트랜잭션 c을 적용(치환)한 결과가 만족가능한지 검사한다. 타입 유추 알고리즘에서 만족가능성(Satisfiability)은 그림 5에서의 제약조건 간단화와 비슷한 구조로 정의되어 여기서 fail이 나는 경우가 있는지 검사한다. 또한 위 결과내에 변수 α가 없으면(Atomic) 제약조건을 생성하지 않는다. 함수 호출의 결과는 함수의 결과 트랜잭션 문맥에 치환 $[c/\alpha]$를 적용한 결과가 된다. `let`은 실행흐름에 따라 트랜잭션 문맥을 전달하고, 필요한 제약조건을 모은다. `if`는, 양쪽 분기에서 나올수 있는 두가지 트랜잭션 문맥을 합치기(Join : $\sqcup$)을 통해 요약하고, 필요한 제약조건을 모은다. `open_ro`, `open_rw`, `close`, `read`, `write`은 각각 $\forall\alpha$ with $\{\alpha \sqsubseteq Closed\}.\alpha \rightarrow OR$, $\forall\alpha$ with $\{\alpha \sqsubseteq Closed\}.\alpha \rightarrow ORW$, $\forall\alpha$ with $\{\alpha \sqsubseteq OR\}.\alpha \rightarrow Closed$, $\forall\alpha$ with $\{\alpha \sqsubseteq OR\}.\alpha \rightarrow \alpha$, $\forall\alpha$ with $\{\alpha \sqsubseteq ORW\}.\alpha \rightarrow \alpha$ 타입의 함수호출과 같은 의미로 유추한다.

이제 우리의 타입 유추 알고리즘으로 1장에서 문제로 제기된 프로그램 1와 프로그램 2를 검증해 보자. 프로그램 1에서 `if`문의 `then`문은 $Closed$로, 그리고 `else`문은 ORW로 결과 트랜잭션 문맥이 유추 된다. 이에 따라, 프로그램 1의 최종 트랜잭션 문맥은 $\top$이 되게 된다. 위의 트랜잭션 관련 명령의 타입에서 볼 수 있듯, 우리의 타입 유추 알고리즘에서는 $\top$이 유추되면 차후 어떤 트랜잭션 명령도 수행할 수 없게 된다. 따라서, 이 경우 즉시 트랜잭션 처리 오류 메시지를 발생시킨다. 프로그램 2에서는 `if`문의 `else`문에서 `write`의 입력 트랜잭션 문맥이 OR로 주어지기 때문에, [I_Write] 규칙에 의거 트랜잭션 처리 오류 메시지를 발생시킨다.

가장 일반적인 계층구조타입시스템의 유추 알고리즘은 지수 시간(Exponential Time) 복잡도를 가지는 것[7]으로 알려져 있다. 하지만, 본 논문에서 다루는 언어는 일차함수(First-Order) 만을 다루고, 트랜잭션 문맥간의 순서관계로부터 얻어지는 래티스(Lattice)가 유한한 간단한 문제이다. 또한, 트랜잭션 문맥 변수가 하나만 존재하기 때문에 간단화 및 만족가능성 검사는 프로그램의 크기 n에 비례하는 선형시간(Linear-Time : $O(n)$)의 복잡도를 가지게 된다. 따라서, 제안된 타입 유추 알고리즘의 복잡도는 프로그램의 크기에 비례하는 선형시간 복잡도를 가진다.

[I_Program] $I(\Gamma, B, c, \texttt{fun}\ f(x)\ =\ e;\ p)\ =$
$\quad \text{let}\quad (B_f, c_f)\ =\ \text{fix}_{(\alpha \sqsubseteq \top, \bot)} \lambda(B_i, c_i).$
$\qquad\qquad \text{let}\quad (c_o, B')\ =\ I(\Gamma \cup \{f : \forall\alpha \text{ with } B_i.\alpha \to c_i\}, B_i, \alpha, e)$
$\qquad\qquad\qquad B_o\ =\ \text{simplify}(B')$
$\qquad\qquad \text{in}\quad (B_o, c_o)\quad \text{end}$
$\qquad (c_p, B_p)\ =\ I(\Gamma \cup \{f : \forall\alpha \text{ with } B_f.\alpha \to c_f\}, B, c, p)$
$\text{in}\ (c_p, B_p)\ \text{end}$

[I_Value] $I(\Gamma, B, c, v)\ =\ (c, \emptyset)$

[I_App] $I(\Gamma, B, c, f\ v)\ =$
$\text{let}\quad \forall\alpha \text{ with } B_f.\alpha \to c_f\ =\ \Gamma(f)$
$\qquad S\ =\ [c/\alpha]$
in
$\quad \text{if satisfiable}(B,\ B_f S) \text{ then}$
$\qquad \text{if atomic}(B_f S) \text{ then } (c_f S,\ \emptyset) \text{ else } (c_f S,\ B_f S)$
$\quad \text{else } fail$
end

[I_Let] $I(\Gamma, B, c, \texttt{let}\ x = e_1\ \texttt{in}\ e_2)\ =$
$\text{let}\quad (c_1, B_1)\ =\ I(\Gamma, B, c, e_1)$
$\qquad (c_2, B_2)\ =\ I(\Gamma, B, c_1, e_2)$
in
$\quad (c_2, B_1 \cup B_2)$
end

[I_If] $I(\Gamma, B, c, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2)\ =$
$\text{let}\quad (c_1, B_1)\ =\ I(\Gamma, B, c, e_1)$
$\qquad (c_2, B_2)\ =\ I(\Gamma, B, c, e_2)$
in
$\quad (c_1 \sqcup c_2, B_1 \cup B_2)$
end

그림 6: 타입 유추 알고리즘(일반 명령)

[I_Open_RO] $I(\Gamma, B, c, \texttt{open\_ro}) =$
let $B' = c \sqsubseteq Closed$
in
if satisfiable($B,\ B'$) then
if atomic(B') then $(OR, \emptyset)$ else (OR, B')
else $fail$
end

[I_Open_RW] $I(\Gamma, B, c, \texttt{open\_rw}) =$
let $B' = c \sqsubseteq Closed$
in
if satisfiable($B,\ B'$) then
if atomic(B') then $(ORW, \emptyset)$ else (ORW, B')
else $fail$
end

[I_Close] $I(\Gamma, B, c, \texttt{close}) =$
let $B' = c \sqsubseteq OR$
in
if satisfiable($B,\ B'$) then
if atomic(B') then $(Closed, \emptyset)$ else $(Closed, B')$
else $fail$
end

[I_Read] $I(\Gamma, B, c, \texttt{read}) =$
let $B' = c \sqsubseteq OR$
in
if satisfiable($B,\ B'$) then
if atomic(B') then $(c, \emptyset)$ else (c, B')
else $fail$
end

[I_Write] $I(\Gamma, B, c, \texttt{write}) =$
let $B' = c \sqsubseteq ORW$
in
if satisfiable($B,\ B'$) then
if atomic(B') then $(c, \emptyset)$ else (c, B')
else $fail$
end

그림 7: 타입 유추 알고리즘(트랜잭션 명령)

4.2 타입 유추 알고리즘의 올바름 증명

본 절에서는 3장에서 제시된 타입시스템에 대하여 타입 유추 알고리즘이 안전함을 보인다. 즉, 주어진 프로그램이 타입 유추 알고리즘을 통과하면, 타입시스템도 항상 통과함을 보인다.

보조정리 10 [만족가능성] *$I(\Gamma, B, c, e) = (c, B')$이면 $B \models B'$이다.*

(증명) 타입 유추 알고리즘 I가 주어진 제약조건 가정에서 항상 만족가능한 제약조건만을 생성함을 보인다. 이는 e에 대한 귀납증명을 통해 간단히 증명된다.

보조정리 11 [간단화] *$B = \text{simplify}(B')$이다.*

(증명) 그림 11의 간단화(Simplify)의 정의에 의해 자명하다.

정리 2 [타입 유추 알고리즘의 안전성] *$I(\Gamma, B, c, p) = (c_p, B_p)$이면 $\Gamma, B \cup B_p, c \vdash p : c_p$이다. 그리고, $I(\Gamma, B, c, e) = (c_e, B_e)$이면 $\Gamma, B \cup B_e, c \vdash p : c_e$이다.*

(증명) p와 e의 구조에 대한 귀납증명을 통해 증명한다.

- $p = \texttt{fun}\, f\ (x)\texttt{=}e\ \texttt{;}p'$인 경우 :
  [I_Program]과 귀납가정(Inductive Hypothesis)에 의해 (1) $\Gamma \cup \{f : \forall\alpha \text{with} B_i.\alpha \rightarrow c_i\}$, $B_i\ \cup B', \alpha \vdash e : c_o$, (2) $\Gamma \cup \{f : \forall\alpha \text{ with } B_f.\alpha \rightarrow c_f\}, B \cup B_p, c \vdash p' : c_p$ 가 성립한다. [I_Program] 에서 (B_f, c_f)가 고정점(Fixpoint)이기 때문에, (3) $(B_f, c_f) = (B_o, c_o) = (B_i, c_i)$ 가 성립한다. (3)과 보조정리 11에 의해 (4) $(B_o, c_o) = (B', c_o)$ 가 성립한다. (1), (3), (4)와 귀납가정에 의해 (5) $\Gamma \cup \{f : \forall\alpha \text{ with } B_f.\alpha \rightarrow c_f\}$, $B_f \cup B_f, \alpha \vdash e : c_f$ 가 성립한다. 그러므로, (5), (2), 보조정리 10, [Program] 규칙에 의해 $\Gamma, B \cup B_p, c \vdash \texttt{fun}\, f\ (x)\texttt{=}e\ \texttt{;}p' : c_p$ 가 성립한다.

- $e = v$인 경우 :
  [I_Value]와 귀납가정에 의해 (1) $c_e = c$ and $B_e = \emptyset$ 가 성립한다. [Value] 규칙에 의해 (2) $\Gamma, B, c \vdash v : c$ 가 성립한다. 그러므로, (1), (2)와 [Value] 규칙에 의해 $\Gamma, B \cup B_e, c \vdash v : c_e$ 가 성립한다.

- $e = f\ v$인 경우 :
  [I_App]와 귀납가정에 의해 (1) $\Gamma(f) = \forall\alpha \text{with} B_f.\alpha \rightarrow c_f$, (2) satisfiable$(B, B_f[c/\alpha])$, (3) $c_e = c_f[c/\alpha]$ 가 성립한다. 만약, $B_f[c/\alpha]$가 변수를 갖지 않으면(Atomic) (4-1) $B_e = \emptyset$ 이다. (4-1)과 만족가능성의 정의에 따라 (4-2) $B \cup B_e \vdash B_f[c/\alpha]$ 이다. (1), (3), (4-2)와 [App] 규칙에 의해 (4-3) $\Gamma, B \cup B_e, c \vdash f\ v : c_e$ 가 성립한다. 만약, $B_f[c/\alpha]$가 변수를 가지면(Not Atomic) (5-1) $B_e = B_f[c/\alpha]$ 이다. (5-1)과 그림 3의 규칙에 의해 (5-2)$B \cup B_e \vdash B_f[c/\alpha]$ 이다. (1), (3), (5-2)와 [App] 규칙에 의해 (5-3) $\Gamma, B \cup B_e, c \vdash f\ v : c_e$ 가 성립한다. 그러므로, (4-3), (5-3)에 의해 $\Gamma, B \cup B_e, c \vdash f\ v : c_e$ 가 성립한다.

- $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$ 인 경우 :
  [I_Let]와 귀납가정에 의해 (1) $\Gamma, B \cup B_1, c \vdash e_1 : c_1$, (2) $\Gamma, B \cup B_2, c_1 \vdash e_2 : c_2$, (3) $c_e = c_2$, (4) $B_e = B_1 \cup B_2$ 가 성립한다. (4), (1), (2)와 보조정리 5에 의해 (5) $\Gamma, B \cup B_e, c \vdash e_1 : c_1$ (6) $\Gamma, B \cup B_e, c_1 \vdash e_2 : c_2$ 가 성립한다. 그러므로, (5),(6),(3)과 [Let] 규칙에 의해 $\Gamma, B \cup B_e, c \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : c_e$ 가 성립한다.

- $e = \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$인 경우 :
  [I_If]와 귀납가정에 의해 (1) $\Gamma, B \cup B_1, c \vdash e_1 : c_1$, (2) $\Gamma, B \cup B_2, c \vdash e_2 : c_2$, (3) $c_e = c_1 \sqcup c_2$, (4) $B_e = B_1 \cup B_2$ 가 성립한다. (4), (1), (2)와 보조정리 5에 의해 (5)$\Gamma, B \cup B_e, c \vdash e_1 : c_1$ (6)$\Gamma, B \cup B_e, c \vdash e_2 : c_2$ 가 정립한다. 그러므로, (5), (6), (3)과 [If] 규칙에 의해 $\Gamma, B \cup B_e, c \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : c_e$ 가 성립한다.

- $e = \texttt{open_ro}$인 경우 :
  [I_Open_RO]와 귀납가정에 의해 (1) satisfiable$(B, c \sqsubseteq \mathit{Closed})$, (2) $c_e = \mathit{OR}$ 가 성립한다. 만약, 제약조건 $c \sqsubseteq \mathit{Closed}$에 변수가 없으면 (3-1) $B_e = \emptyset$ 이다. (3-1)과 만족가능성의 정의에 의해 (3-2) $B \cup B_e \vdash c \sqsubseteq \mathit{Closed}$ 이다. (1), (2), (3-2)와 [Open_RO] 규칙에 의해 (3-3) $\Gamma, B \cup B_e, c \vdash \texttt{open\_ro} : c_e$ 가 성립한다. 만약, $c \sqsubseteq \mathit{Closed}$에 변수가 있으면, (4-1) $B_e = c \sqsubseteq \mathit{Closed}$ 이다. (4-1)과 그림 3의 규칙에 의해 (4-2) $B \cup B_e \vdash c \sqsubseteq \mathit{Closed}$ 이다. (1),(3),(4-2)와 [Open_RO] 규칙에 의해 (4-3)$\Gamma, B \cup B_e, c \vdash \texttt{open\_ro} : c_e$ 이 성립한다. 그러므로, (3-3), (4-3)에 의해 $\Gamma, B \cup B_e, c \vdash \texttt{open\_ro} : c_e$ 가 성립한다.

- $e = \texttt{open\_rw}$, $\texttt{close}$, $\texttt{read}$, $\texttt{write}$인 경우 : $\texttt{open_ro}$와 같은 방식의 귀납증명을 통하여 증명된다.

5 트랜잭션 잠금수준 튜닝

본 장에서는 3장에서 소개된 트랜잭션 타입 시스템을 확장하여 open_rw으로 열린 트랜잭션 중 실제로 read 만하는 트랜잭션을 찾아내는 방법을 제시한다. 이를 바탕으로, 비 효율적으로 설정된 트랜잭션 잠금 명령(open_rw)을 open_ro로 고쳐서 성능을 개선하는 방법을 제시한다.

5.1 타입시스템 확장

$t \in \mathit{TId}$	::=	ρ	open_label variable
	|	$\{l\}$	label
	|	$t \cup t$	union
$\epsilon \in \mathit{Access}$	::=	$\mathit{read}(t)$	read
	|	$\mathit{write}(t)$	write
$\varphi \in \mathit{Accesses}$	|	$\wp(\mathit{Access})$	

그림 8: 트랜잭션 식별자 및 접근

먼저 타입시스템에 그림 8과 같은 도메인(Domain)을 추가한다. 트랜잭션 식별자(TId)는 각 open 명령에 대한 레이블의 집합이다. 이때, 각 open 명령에 고유 레이블이 주어진다고 가정한다. 트랜잭션 접근 정보($\mathit{Accesses}$)는 각 트랜잭션에 대한 접근(Access)을 모은 집합이다.

[E_Program]
$$\frac{\models B_f \quad \Gamma \cup \{f : \forall \boxed{\rho}, \alpha \text{ with } B_f.\alpha \overset{\boxed{\varphi_f}}{\rightarrow} c_f\}, B_f, \alpha \vdash e : c_f, \boxed{\varphi_f} \quad \Gamma \cup \{f : \forall \boxed{\rho}, \alpha \text{ with } B_f.\alpha \overset{\boxed{\varphi_f}}{\rightarrow} c_f\}, B, c \vdash p : c', \boxed{\varphi}}{\Gamma, B, c \vdash \texttt{fun}\, f\ (x)\texttt{=}e\ \texttt{;}p : c', \boxed{\varphi}}$$

[E_Value]
$$\frac{}{\Gamma, B, c \vdash v : c, \boxed{\emptyset}}$$

[E_App]
$$\frac{\Gamma(f) = \forall \boxed{\rho}, \alpha \text{ with } B_f.\alpha \overset{\boxed{\varphi_f}}{\rightarrow} c_f \quad \boxed{\text{label}(c) = t} \qquad B \vdash B_f\boxed{[t/\rho]}[c/\alpha]}{\Gamma, B, c \vdash f\ v : c_f[c/\alpha], \boxed{\varphi_f[t/\rho]}}$$

[E_Let]
$$\frac{\Gamma, B, c \vdash e_1 : c_1, \boxed{\varphi_1} \qquad \Gamma, B, c_1 \vdash e_2 : c_2, \boxed{\varphi_2}}{\Gamma, B, c \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : c_2, \boxed{\varphi_1 \cup \varphi_2}}$$

[E_If]
$$\frac{\Gamma, B, c \vdash e_1 : c_1, \boxed{\varphi_1} \qquad \Gamma, B, c \vdash e_2 : c_2, \boxed{\varphi_2}}{\Gamma, B, c \vdash \texttt{if}\ \ v\ \ \texttt{then}\ \ e_1\ \ \texttt{else}\ \ e_2 : c_1 \sqcup c_2, \boxed{\varphi_1 \cup \varphi_2}}$$

[E_Open_RO]
$$\frac{B \vdash c \sqsubseteq \mathit{Closed}}{\Gamma, B, c \vdash \texttt{open\_ro}_l : \mathit{OR}(\boxed{\{l\}}), \boxed{\emptyset}}$$

[E_Open_RW]
$$\frac{B \vdash c \sqsubseteq \mathit{Closed}}{\Gamma, B, c \vdash \texttt{open\_rw}_l : \mathit{ORW}(\boxed{\{l\}}), \boxed{\emptyset}}$$

[E_Close]
$$\frac{B \vdash c \sqsubseteq \mathit{OR}(\boxed{\text{label}(c)})}{\Gamma, B, c \vdash \texttt{close} : \mathit{Closed}, \boxed{\emptyset}}$$

[E_Read]
$$\frac{\boxed{\text{label}(c) = t} \qquad B \vdash c \sqsubseteq \mathit{OR}(\boxed{t})}{\Gamma, B, c \vdash \texttt{read} : c, \boxed{\{\mathit{read}(t)\}}}$$

[E_Write]
$$\frac{\boxed{\text{label}(c) = t} \qquad B \vdash c \sqsubseteq \mathit{ORW}(\boxed{t})}{\Gamma, B, c \vdash \texttt{write} : c, \boxed{\{\mathit{write}(t)\}}}$$

그림 9: 확장된 타입결정규칙

트랜잭션 상태는 트랜잭션이 열렸음을 뜻하는 OR 또는 ORW가 관련 있는 `open` 명령의 레이블을 가지도록 확장한다.

$$\begin{array}{rcll} s \in \mathit{TransactionStatus} & ::= & \bot & \text{dead or non-termination} \\ & | & \mathit{Closed} & \text{closed} \\ & | & \mathit{OR}(\boxed{t}) & \text{read possible} \\ & | & \mathit{ORW}(\boxed{t}) & \text{read/write possible} \\ & | & \top & \text{can be all} \end{array}$$

함수의 타입은 해당 함수가 호출되면 발생하는 잠재(latent) 트랜잭션 접근 정보를 가지도록 확장한다.

$$ts \in \mathit{TypeScheme} \quad ::= \quad \forall \boxed{\rho}, \alpha \, with \, B.\alpha \xrightarrow{\boxed{\varphi_f}} c$$

트랜잭션 타입 결정 판별식은 트랜잭션 접근 정보를 모을수 있도록 다음과 같은 형태로 확장한다.

$$\Gamma, B, c \vdash_p p : c', \boxed{\varphi}$$
$$\Gamma, B, c \vdash_e e : c', \boxed{\varphi}$$

그리고 "주어진 타입환경 Γ, 제약조건 가정 B, 입력 트랜잭션 문맥 c에서, 프로그램 p 혹은 프로그램식 e를 유추하면, 결과 트랜잭션 문맥은 c', 트랜잭션 접근 정보는 φ이다" 로 읽는다.

이제 확장된 타입결정규칙은 그림 9와 같이 정의된다.

함수의 타입은 함수의 몸통을 수행 했을 때 발생할 수 있는 트랜잭션 접근 정보를 모아서 함수타입의 잠재 접근 정보로 정한다. 값은 트랜잭션에 관여하지 않으므로 접근 정보가 없다. 함수 호출에서는 호출 되는 시점에서 트랜잭션이 열려 있으면 트랜잭션을 열때 트랜잭션 문맥에 설정해 둔 트랜잭션 식별자를 함수의 잠재 접근 정보의 변수 ρ에 적용(치환)하여 트랜잭션 접근정보를 결정한다. `let`과 `if`는 각 부분 프로그램식(Sub-Expression) 을 수행하여 발생할 수 있는 트랜잭션 접근정보를 모은다. $\texttt{open_ro}_l$와 $\texttt{open_rw}_l$은 해당 레이블을 트랜잭션 문맥에 설정한다. `close`는 접근 정보를 생성하지 않는다. `read`와 `write`는 현재의 트랜잭션 식별자 t에 대한 $read(t)$, $write(t)$ 접근정보를 생성한다.

확장된 타입시스템에 의해 분석된 결과 접근 정보 φ는 주어진 프로그램을 수행하면 발생하는 트랜잭션 접근을 뜻한다. 즉, 확장된 타입시스템으로 `read` 접근만 하는 모든 `open_rw`대한 레이블을 결정한 후, 이 정보를 바탕으로 주어진 프로그램의 트랜잭션 잠금수준을 튜닝한다.

본장에서 제시된 잠금수준 튜닝 방법을 통해 1장에서 문제로 제기된 프로그램 3을 튜닝하면 다음과 같이 된다. 우선 첫번째 줄의 `begin_transaction(Serializable)`은 $\texttt{open_rw}_l$의 [E_Open_RW] 규칙에 따라, 결과 트랜잭션 문맥으로 $ORW(l)$을 생성한다. `read`에서는 [E_Read] 규칙에 의해 $read(l)$ 트랜잭션 접근 정보가 발생한다. `show_msg` 함수는 잠재 트랜잭션 접근이 없기 때문에 트랜잭션 접근 정보가 발생하지 않는다. 따라서, $\texttt{open_rw}_l$에 대한 트랜잭션 접근정보는 $\{read(l)\}$ 가 된다. 이에 따라, 주어진 알고리즘은 $\texttt{open_rw}_l$을 $\texttt{open\_ro}_l$ 로 변환한다.

확장된 타입시스템을 위한 확장 타입 유추 알고리즘 및 안전성 증명은 3장, 4장에서 논의 및 증명된 방식으로 막힘없이 자연스럽게 진행되므로 본 논문에서는 자세한 내용을 생략한다.

6 트랜잭션 잠금시간 튜닝

본 장에서는 불필요하게 트랜잭션 종료가 늦추어 지도록 프로그래밍해서, 해당 프로그램이 전체 시스템의 성능에 장애가 되는 경우를 찾아 주고, 이러한 트랜잭션 종료 명령을 안전한 수준에서 최대한 앞 당긴 프로그램으로 자동 튜닝해주는 알고리즘을 제시한다.

1. 주어진 프로그램에 대한 실행흐름그래프를 생성한다.
2. 실행흐름그래프의 각 노드 b에 대해 다음과 같이 고정점을 계산한다.
   $\mathrm{fix}_{(\bot,\bot)}\ \lambda(out[b], in[b])$.
   `let` $joined = \sqcup_{\forall s \in succ(b)} in[s]$
   `in`
   $(joined,$ if $(b =$ "`close`") then $Move$
   else
   if $(joined = Move$ and $type(b) = \forall\rho.\forall\alpha$ with $\alpha \sqsubseteq \top.\ \alpha \xrightarrow{\emptyset} \alpha)$
   then $Move$
   else $Stop$
   `end`
3. 실행흐름그래프에서 모든 "close" 노드를 "skip"으로 바꾼다.
4. 분석결과를 바탕으로 다음과 같이 실행흐름그래프에 "close"노드를 추가한다.
   case a edge's DFI $(d1, d2)$ of
   $|\ (Stop, Move)\ \Rightarrow$ insert
   $|\ (Stop, Stop)\ \Rightarrow$ do nothing
   $|\ (Move, Move)\ \Rightarrow$ do nothing
   $|\ (Move, Stop)\ \Rightarrow$ error

그림 10: 잠금시간 튜닝 알고리즘

트랜잭션 잠금시간 튜닝 알고리즘은 컴파일러의 최적화 기술로서는 잘 알려진 데이터흐름분석(Data Flow Analysis) 방법[8]을 응용하여 정의하였다. 일반적으로 데이터흐름분석은 데이터흐름정보(DFI) D의 정의, 데이터흐름정보 사이의 합치기 연산자 $\sqcup$의 정의, 각 블럭에 대한 요약함수 $F[b]$의 정의의 튜플(Tuple)로 표현된다. 데이터 흐름정보는 분석하고자 하는 대상 성질을 요약한 것으로서, 주어진 프로그램의 실행흐름그래프(Control Flow Graph)의 각 노드 b의 입력부 $in[b]$와 출력부 $out[b]$에 설정된다. 합치기 연산은 데이터흐름정보 사이의 순서관계에 따라 정의 된다. 요약함수 $F[b]$는 각 노드의 입력부를 통해 들어온 데이터흐름정보에 취해지는 함수이다. 분석시 함수의 적용 결과는 해당 노드의 출력부에 설정된다.

본 분석에서는 트랜잭션 종료 명령을 실행흐름의 역방향으로 이동하길 원한다. 따라서, 데이터흐름정보는 트랜잭션 종료 명령을 실행흐름의 역방향으로 이동($Move$), 이동안함($Stop$)으로 정의한다. 데이터흐름정보 간의 순서관계는 $\bot \sqsubseteq Move$, $\bot \sqsubseteq Stop$, $Move \sqsubseteq Stop$ 으로 간단히 정의 된다. 이에 따라, 합치기 연산자 $\sqcup$은 $Move \sqcup Stop = Stop$ 식으로 정의된다. 제안된 분석은 실행흐름의 역방향으로 데이터를 전달하므로 $F[b]$는 $out[b]$를 받아서 $in[b]$를 결정하는 식으로 역방향으로 정의된다. $F[b]$의

자세한 정의는 그림10에서 고정점 계산 대상 함수의 몸통에서 확인할 수 있다. 여기서 해당 노드가 함수 호출일 때는, 호출되는 함수가 트랜잭션과 관련이 있는지 여부를 지금까지 제안된 트랜잭션 함수 타입을 바탕으로 결정한다. 따라서, 함수호출이 있어도 트랜잭션 함수타입을 바탕으로 해당 함수 호출 이전으로 안전하게 트랜잭션 종료 명령을 앞 당길수 있다.

제시된 잠금시간 튜닝 알고리즘을 사용하여 1장에서 문제로 제기된 프로그램 3를 튜닝하면 다음과 같이 된다. 우선 `commit` 함수 호출(`close`)에서 발생한 $Move$가 실행 흐름의 역방향으로 이동되기 시작한다. `show_msg` 함수는 트랜잭션 접근정보가 없기 때문에 $Move$는 실행흐름의 역방향으로 계속 이동하게 된다. `read` 에서는 트랜잭션 접근을 하기 때문에 `read` 노드의 데이터흐름정보는 $Stop$으로 설정 된다. 알고리즘에 따라 `commit`은 `read`와 `show_msg` 사이에 삽입된다.

7 관련 연구와의 비교

Vault 프로그래밍 언어[9]는 본 논문의 트랜잭션 안전성 분석과 같이 함수 호출간에 지켜져야 하는 규칙(protocol)을 검증할 수 있다. 우리의 타입시스템과의 차이는 Vault에서는 상태정보와 그 변화를 프로그래머가 직접 프로그램속에 표기하여야 하고, 우리의 타입시스템은 이를 자동으로 유추한다는 점이다. Vault의 타입시스템은 각 자원(Resouce)에 대한 상태정보를 나타내는 키(Key)라는 개념을 프로그래밍 언어에 도입하여 그 상태정보의 변화를 추적할 수 있도록 하였다. 자원의 상태는 어떤 일(함수호출)을 할 수 있는지를 결정한다. 그리고 가능한 일을 수행함에 따라 상태전이가 일어나게 된다.

Igarashi와 Kobayashi의 자원 사용 분석(Resouce Usage Analysis) [10][11]을 응용하면 자원의 사용이 어떻게 되는지 추적함으로써 본 논문의 트랜잭션 처리 안전성 검증 문제를 풀 수 있다. 우리의 타입시스템과의 차이는 자원 사용 분석은 분석의 정확도에 있어 재귀 호출의 다형성을 지원하지 않으며, 4장에서 제시된바와 같이 구현을 위한 구체적 유추 알고리즘이 정확히 제시되지 않았다는 점이다. 또한, 5장과 6장의 튜닝을 위한 정보를 추출할 수 있도록 직접적으로 확장하여 적용하기에는 제약이 있었다.

Cosimo Laneve는 Java 바이트 코드 검증기를 확장하여 잠금명령(`monitorenter`)과 잠금해제 명령(`monitorexit`)이 제대로 짝지어 졌는지 검증하는 시스템[12]을 제안하였다. Engler등의 Metal 프로젝트[13]에서는 리눅스 운영체제에 들어가는 프로그램들에 대해 잠금(Lock)명령과 잠금해제(Unlock) 명령이 제대로 짝지어 졌는지 구분적으로 간단하게 검증하는 시스템을 제안하였다. 이들 연구는 본 연구의 트랜잭션이 항상 닫히는지 검증하는 첫번째 안전성 분석 문제와 유사하다. 하지만 이들 연구는 함수 호출간의(Interprocedural) 분석을 제대로 지원하지 않는다. 또한, 읽기모드 트랜잭션 처리 검증 및 차후 튜닝을 위한 정보추출 문제등으로 인해, 직접적으로 확장하여 적용하기에는 제약이 있다.

보다 일반화된 안전성 검증 연구로는 마이크로소프트 연구소의 SLAM 프로젝트[14]와 켄사스대학의 Bandera 프로젝트[15]가 있다. 이들은 본 연구 문제중 트랜잭션 처리의 안전성을 검증 가능하게 하는 도구들을 만들고 있다. SLAM프로젝트는 프로그램 분석 기술과 모델검증(Model Checking) 기술에 기반하여 마이크로소프트 윈도우즈 운영체제에 들어가는 디바이스 드라이버(Device Driver)들의 안전성 검증 도구를 개발하고 있다. 구체적으로 SLAM은 C언어로 작성된 프로그램들의 API 함수들이 주어진 스펙(Specification)에 따라 사용되는지 검증한다. Bandera 프로젝트는 모델검증 방법을 사용하여 Java 프로그램의 수행 중 특정 시점 또는 기간에 일어나는

작동 양상(Temporal Property)을 미리 예측 및 검증해주는 일반화된 도구를 만들고 있다. 그러나 매우 일반화된 검증 방법을 고안하는데 집중함으로써 검증의 성능이 본 연구에 비하여 비효율적이고 [7], 튜닝을 위한 분석 정보를 추출 할 수 없어서, 본 문제에 직접적으로 활용하기에는 적당하지 않다.

결론적으로, 아직 본 논문에서 제시된 것처럼 트랜잭션 기반 응용프로그램의 신뢰성 및 성능 관리를 위해 프로그램 분석 기술을 직접적으로 적용하여 개발된 시스템은 없는 것으로 보인다. 이는 프로그램 분석 연구가 프로그래밍 언어 자체와 컴파일러 수준에서 적용되는 기술을 주로 대상으로 삼아 왔고, 데이터베이스 연구는 프로그램과 분리하여 데이터베이스 자체를 대상으로 삼아 왔기 때문으로 추측된다.

8 결론

정보시스템의 핵심부품인 트랜잭션 기반 응용프로그램에 대한 기존의 안전성 및 성능 관리는 전적으로 개발자 혹은 전문가의 능력에 의존해 왔다. 이에 따라, 안전성 및 성능 관리 자체의 신뢰성 및 효율성이 문제가 되어 왔다.

본 논문에서는 이와 같은 문제에 착안하여 프로그램 분석 기술에 기반한 안전하고 자동화된 안전성 검증 및 성능 관리 시스템을 제안하였다. 제안된 안전성 검증 시스템은 주어진 프로그램에서 트랜잭션을 열고서 닫지 않는 오류와, 트랜잭션의 잠금수준을 잘못 설정하는 오류를 자동으로 검출하여 준다. 제안된 튜닝 시스템은 비효율적인 잠금수준 설정과 불필요하게 지연된 트랜잭션을 찾아내서 자동으로 튜닝해 준다.

제안된 방식은 프로그램을 실행하기 전에 분석하여 오류를 예측하고 튜닝을 수행하기 때문에, 실제 장애가 발생하기 전에 미리 대처가 가능한 장점이 있다. 또한, 안전성이 증명된 방법으로 이러한 분석을 수행하기 때문에, 주어진 오류 검증 및 튜닝 방법, 그리고 그 결과를 완벽하게 신뢰할 수 있다는 장점이 있다.

결론적으로, 본 연구의 의의는 데이터베이스 응용프로그래머가 프로그래밍 할 때 믿고 활용할 수 있는 자동화된 트랜잭션 처리 안전성 검증 및 튜닝 도구를 제시한데 있다. 또한, 이미 구축된 정보시스템의 트랜잭션 처리 안전성 검증 및 튜닝 도구로도 활용할 수 있다.

참고 문헌

[1] Philip A. Bernstein and Eric Newcomer, Principles of Transaction Processing, Morgan Kaufmann Publishers, 1997.

[2] Harvey W. Gunther, “WebSphere Application Server Development Best Practices for Performance and Scalability”, IBM White Paper, 2000.

[3] Philippe Bonnet and Denis E. Shasha, Database Tuning: Principles, Experiments, and Troubleshooting Techniques, Morgan Kaufmann Publishers, 2002.

[4] Geoffrey S. Smith, “Polymorphic Type Inference for Language with Overloading and Subtyping”, PhD thesis, Cornell University, August 1991.

[5] Andrew K. Wright and Matthias Felleisen, “A syntactic approach to type soundness”, Technical report TR91-160, Rice University, 1992.

[7] 본 연구의 검증 알고리즘의 효율성은 4장에서 논의하였다.

[6] Benjamin C. Pierce, Types and Programming Languages, Kluwer Acdemic Publisher, The MIT Press, 2002.

[7] Patrick Lincoln and John C.Mitchell, "Algorithmic aspects of type inference with subtypes", In *19th ACM Symposium on Principles of Programming Languages*, pages 193-304, 1992.

[8] Steven S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers, 1997

[9] Robert DeLine and Manuel Fändrich, "Enforcing High-Level Protocols in Low-Level Software", In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI 2001)*, 2001.

[10] Atsushi Igarashi and Naoki Kobayashi, "Resouce Usage Analysis", In *Proceedings of Symposium on Principles of Programming Languages*, pages 331-342, 2002.

[11] Naoki Kobayashi, "Time Regions and Effects for Resouce Usage Analysis", In *Proceedings of ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2003.

[12] Cosimo Laneve. A Type System for JVM Threads. In *Proc. of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000)*, 2000.

[13] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions", In *Synposium on Operating Systems Design and Implementation(OSDI2000)*, Oct. 2000.

[14] Thomas Ball and Sriram K. Rajamani, "The SLAM Project: Debugging System Software via Static Analysis", In *Symposium on Principles of Programming Languages 2002(POPL2002)*, 2002.

[15] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach and Hongjun Zheng, "Bandera: Extracting Finite-state Models from Java Source Code", In *in Proceedings of the 22nd International Conference on Software Engineering*, 2000.