임베디드 시스템을 위한 실시간 함수형 반응적 프로그램 변환기※

이 동 주† · 우 균††

요 약

함수형 반응적 프로그래밍(FRP: Functional Reactive Programming)은 하스켈(Haskell)에 내장된 언어로서 두 핵심 고차 타입인 행위(Behavior)와 이벤트(Event)를 기반으로 반응적 시스템을 선언적으로 프로그래밍 한다. 이 논문은 다양한 제약사항을 가진 임베디드 시스템에 FRP를 이용하기 위해 실시간 함수형 반응적 프로그래밍(RT-FRP: Real-time Functional Reactive Programming) 변환기를 설계하고 구현한다. RT-FRP 변환기는 RT-FRP 언어의 기능적 의미론을 기반으로 RT-FRP 프로그램에서 C 프로그램을 생성한다.

RT-FRP 변환기의 효과적인 변환을 입증하기 위해, 변환기에서 생성된 테스트 프로그램을 실제 임베디드 시스템, 레고 마인드스톰(LEGO Mindstorm)에 탑재하고 수행하였다. 실험 결과, RT-FRP를 이용할 경우 목적파일의 크기가 조금 증가하였으나 명령형 언어를 이용한 프로그램보다 간결하게 반응적 시스템 소프트웨어를 프로그래밍 할 수 있었다.

키워드 : 반응적 언어, 함수형 반응적 언어, 하스켈, 임베디드 시스템

Real-Time Functional Reactive Program Translator for Embedded Systems※

Dong-Ju Lee† · Gyun Woo††

ABSTRACT

FRP(Functional Reactive Programming) is a kind of embedded language in Haskell, it declaratively program reactive system based on two essential high-order types named behavior and events. This paper design and implementation RT-FRP(Real-time Functional Reactive Programming) translator for using FRP in embedded systems with many constraints. The RT-FRP translator generates a C Program from an RT-FRP program according to the operational semantics of the RT-FRP language.

To show the effectiveness of the RT-FRP translator, we loaded and executed the test program generated by the translator onto a real embedded system, LEGO Mindstorm. According to the experimental result, the reactive system software can be programmed more concisely using RT-FRP than using an imperative counter part although the size of the binary code is rather increased.

Key Words : Reactive Language, Functional Reactive Language, Haskell, Embedded Systems

1. 서 론

임베디드 시스템(embedded system)은 다른 시스템의 일부로 내장된 디지털 시스템으로서 주로 특정 기기에 내장된 컴퓨팅 시스템을 말한다. 임베디드 시스템은 외부의 연속적인 입력에 대해 반응하는 특성을 보이는데, 이를 반응적 특성(reactive characteristic)이라고 한다. 전통적인 반응적 프로그래밍 언어는 동기식 데이터 흐름(synchronous data-flow)에 기반하고 있다. 이러한 언어들은 기존의 명령형 언어를 기반으로 하고 있기 때문에 프로그램을 작성할 때는 데이터의 흐름을 프로그래머가 구체적으로 명시해야 한다. 전통적인 반응적 프로그래밍 언어로는 SIGNAL[1], LUSTRE[2], ESTEREL [3, 4]등이 있다.

2000년 함수형 패러다임을 기반으로 하는 반응적 프로그래밍 언어[5]가 등장하였다. 함수형 반응적 프로그래밍(FRP: Functional Reactive Programming) 언어는 Elliott와 Hudak이 제안한 컴퓨터 애니메이션 언어인 Fran[6]에서 기본적인 아이디어를 얻어 발전하였다. FRP의 아이디어는 컴퓨터로 표현하는 모든 사물을, 시간에 연속적이며 외부 자극이나 사건에 반응하는 대상으로 모델링하는 것이다. 실제 FRP는

※ 이 논문은 2단계 두뇌한국21사업 지원비와 2004년도 부산대학교 교내 학술연구비(신임교수연구정착금) 지원으로 이루어졌음.
† 준 회 원 : 부산대학교 컴퓨터공학과 석사과정
†† 종신회원 : 부산대학교 정보컴퓨터공학부 조교수(교신저자)
논문접수 : 2006년 8월 3일, 심사완료 : 2006년 9월 20일

하스켈로 구현된 내장 언어(embedded language)이다. FRP 언어의 구문은 하스켈(syntax)의 구문을 기본적으로 이용하며, FRP 언어를 위한 연산자는 하스켈의 함수로 구현되어 있다. FRP는 하스켈에서 구현되기 위해 행위(Behavior)와 이벤트(Event)라는 두 개념의 고차타입(High-Order Type)을 제공한다. FRP에서는 두 핵심 타입의 값을 일등급값(First-Class Value)으로 다루고 있으며, 두 타입을 위한 다양한 연산자를 제공하고 있다. 따라서 FRP 프로그램은 행위와 이벤트가 재귀적인 구조로 합성된 집합으로 볼 수 있다.

두 가지 개념을 기반으로 하는 FRP는 컴퓨터 애니메이션(Fran)[6], 그래픽 유저 인터페이스(Fruit)[7], 컴퓨터 비전(FVision)[8], 로봇제어(Frob)[9]와 같은 분야에 대하여 응용분야 특화 언어(domain specific language)로 발전하였다. 이러한 발전의 원동력은 FRP 프로그램이 응용분야의 전문가가 기술한 문제에 대한 요구사항 명세와 매우 유사하다는 점에서 찾을 수 있다[10]. 또한 FRP는 하스켈의 내장언어(embedded language)로 구현되어 있으므로 호스트 언어(host language)의 특징과 개발 도구를 그대로 이용할 수 있다는 장점이 있다[11].

하지만 현재 FRP는 다양한 플랫폼의 임베디드 시스템 프로그래밍에는 활용할 수 없는 상황이다. 그 첫 번째 이유는 하스켈 컴파일러가 지원하는 플랫폼의 한계 때문이다. 현재 하스켈 컴파일러가 이식된 플랫폼은 약 9개에 지나지 않는다[12]. 두 번째 이유는 임베디드 시스템의 메모리 한계 때문이다. 예를 들어 로봇 임베디드 시스템인 레고 마인드스톰(LEGO mindstorm)의 경우, 운영체제와 응용프로그램을 위해 단지 32Kbytes의 메모리를 제공한다[13]. 만약 임베디드 프로세서를 지원하는 하스켈 컴파일러가 있어 목적파일로 변역하였더라도 생성된 목적파일의 크기가 시스템의 가용 메모리의 크기보다 클 경우 프로그램을 탑재할 수 없는 문제가 있다. 이 문제의 근본적인 원인은 FRP가 호스트 언어인 하스켈로 구현되어 있기 때문이다.

이 문제를 해결하기 위해서는 FRP가 독립적인 언어로 정의되어야 되며, 수행 플랫폼을 위한 컴파일러가 제공되어야 한다. FRP 관련 연구결과 중 실시간 시스템 도메인을 위해 정의된 실시간 함수형 반응적 프로그래밍(RT-FRP: Real-Time Functional Reactive Programming) 언어가 있다[14]. RT-FRP는 하스켈에 내장된 언어가 아닌 독립적인 구문과 의미가 정의된 언어이다. 하지만 RT-FRP의 구현에 대해서는 하스켈로 작성된 인터프리터(interpreter)만 발표되었을 뿐, 실제 하스켈이 탑재되지 않는 임베디드 플랫폼에서 수행 가능한 코드를 내는 컴파일러에 대한 개발 사례는 발표된 바 없다.

본 논문은 RT-FRP 변환기를 설계하고 구현하여 지원함으로써 FRP 프로그램을 실제 임베디드 시스템에 탑재하고 구동하는 방법을 제안한다. 하스켈에 내장된 FRP 대신 독립적으로 정의된 RT-FRP 언어를 이용하였으며, 실제 수행 가능한 코드를 생성하는 RT-FRP 변환기를 작성하였다. RT-FRP 변환기는 RT-FRP 프로그램을 입력으로 받아 RT-FRP 언어의 기능적 의미(operational semantics)를 기반으로 이와 동등하게 수행하는 C 프로그램을 생성한다. RT-FRP 프로그램의 변환과정 중 C를 중간 코드로 이용한 이유는 대부분의 임베디드 시스템 프로세서에서 C 컴파일러를 기본적으로 지원하며 실제 다양한 플랫폼에 적용할 수 있기 때문이다. 본 논문에서는 RT-FRP 변환기가 생성한 프로그램을 실제 임베디드 시스템(레고 마인드스톰)에 탑재해 봄으로써 RT-FRP 변환기가 효과적인 변환을 수행함을 보이고자 한다.

본 논문의 구성은 다음과 같다. 2장에서는 RT-FRP 언어의 구문 및 의미에 대해 개략적으로 소개한다. 3장에서는 RT-FRP 변환기에 대한 설계, 구현과정에 대해서 살펴본다. 4장에서는 마인드스톰을 이용한 실험 결과를 기술하고 분석한다. 5장에서는 연구 결과를 바탕으로 RT-FRP 변환기의 효율성을 점검한다. 6장에서는 FRP 관련 연구들에 대해서 소개하며, 마지막으로 7장에서 결론을 맺고 향후 연구 방향을 제시한다.

2. 실시간 함수형 반응적 프로그래밍

하스켈을 기반으로 한 FRP는 2000년도에 Wan과 Hudak에 의해 발표된바 있다[5]. 2001년 Wan은 하스켈에 독립적인 RT-FRP[10, 14]를 발표하였는데, 이 장에서는 Wan이 제시한 RT-FRP를 요약하여 설명한다.

2.1 RT-FRP 소개

RT-FRP는 FRP를 실시간 시스템 도메인에서 이용하기 위해 언어의 구문과 의미가 독립적으로 정의된 호스트 언어이다[10]. 실시간 시스템의 가장 큰 특징은 프로그램의 시간과 공간에 대한 제약사항이다. 하지만 하스켈에 내장된 FRP는 표기적 의미론(denotational semantics)으로 의미를 정의하고 있으며, 하스켈의 함수로 구현되어 있기 때문에 프로그램 수행 시 시간 및 공간에 대한 비용을 예측하기 어렵다. 반면 RT-FRP는 기능적 의미론(operational semantics)으로 각 구문을 정의하고 있다. 기능적 의미론은 동작 전, 후 상태의 변화를 기술하므로 프로그램을 수행할 때 드는 시간적 공간적 비용을 비교적 쉽게 측정할 수 있다. RT-FRP는 언어의 구문과 의미를 FRP에 기반하기 때문에 FRP로 작성한 프로그램은 RT-FRP 프로그램으로 일대일 전환이 가능하다.

RT-FRP는 FRP의 핵심적인 두 타입인 행위와 이벤트를 하나의 시그널(Signal) 타입으로 정의하고 있다. (그림 1)은 각 타입 사이의 관계이다. ≈는 좌우 타입의 의미가 동일함을 나타낸다. RT-FRP는 Maybe 타입을 포함하는 행위 타입으로 이벤트를 나타낸다. 이벤트는 특정시각의 값이 Just x일 경우 이벤트가 발생한 것이며 이 때, x는 이벤트에 포함된 값이다. 반면 Nothing은 이벤트가 발생하지 않은 경우이다.

```
Behavior a ≈ Time -> a
Signal a ≈ Behavior a
Event a ≈ Signal (Maybe a)
    where Maybe a = Nothing | Just a
```

(그림 1) RT-FRP의 시그널 타입

2.2 RT-FRP의 구문 및 의미

RT-FRP는 크게 반응적 언어(reactive language)와 기본 언어(base language), 두 부분으로 구성된다. 반응적 언어는 시그널 타입의 데이터를 표현하고 기본 언어는 그 외 일반적인 타입의 데이터를 표현한다. RT-FRP 언어에서 사용하는 타입은 총 4가지이다. 유닛(unit)값 하나만 포함하고 있는 (), 실수를 나타내는 Real, 순서쌍을 나타내는 튜플(tuple), 값을 가지거나 가지지 않는 Maybe 타입이 있다.

기본 언어는 표현(expression)으로 구성되는데 각 표현은 앞서 설명한 4가지 타입 중 하나이다. f e는 함수적용 구문이다. let x = e in e′ 구문은 지역 변수를 정의하는 구문이며, case 구문은 Maybe 타입과 튜플 타입을 분해하기 위해 제공되는 구문이다. 기본 언어에서 Boolean 타입은 Maybe 타입으로 나타낸다(참 = Just (), 거짓 = Nothing). 기본 언어의 모든 표현은 4가지 타입 중 하나이다.

반응적 언어는 시그널(signal)로 구성된다. Wan의 논문[10]에서는 input과 time을 키워드로 정의하며 이를 시스템에서 제공하는 시그널(primitive signal)로 이용하고 있다. 하지만 실제 임베디드 시스템에서는 외부 입력과 시간을 다루는 시스템 콜(system call)이 시스템마다 다르므로 본 변환기에서는 input과 time을 언어의 구문에서 제외하고 대신 라이브러리 함수로 이들 기능을 지원한다.

let snapshot x <- s_1 in s_2 구문은 시그널 s1에 포함된 특정 시점의 값을 x에 연관시킨 후 시그널 s_2를 반환하는 구문이다. 이 구문은 기존에 정의된 시그널을 이용하여 새로운 시그널을 정의할 때 주로 사용하는 구문이다. ext e는 일반적인 타입(시간에 비종속적임)의 값을 나타내는 구문 e를 시그널 형태로 전환하는 구문이다. let snapshot 구문이 시그널 타입의 값을 일반적인 계산에 사용하기 위한 구문이라고 한다면, ext는 그 반대로 일반적인 값을 시그널 타입으로 전환하는 구문이다. 이 두 구문은 시그널 타입과 일반 타입의 가교 역할을 하며 FRP에서 제공되는 lift 연산자[6]와 동일한 역할을 하는 구문으로 볼 수 있다. delay e s는 초기 값 e와 시그널 s를 받아서 시그널 s의 한 시점 이전의 값을 유지하는 시그널을 반환한다. 이 구문은 상태를 포함하는 시그널을 구성하기 위해 사용된다.

let signal $\{z_j(x_j) = u_j\}$ in s 구문은 이벤트에 의해 전환될 시그널을 정의하는 구문이다. z_j는 정의한 시그널을 나타내는 연속변수(continuous variable)이며, x_j는 이벤트가 일어났을 때의 값을 나타낸다. 끝으로, 이벤트에 의해 시그널이 전환되는 것을 나타내기 위한 s until $<s_j => z_j>$ 구문이 있다. 이 구문은 이벤트 시그널 sj의 값이 Just일 때, 시그널 s가 시그널 z_j로 전환한다.

		-- 타입 언어
타입	Type∋t ::=	() | **Real** | (t_1, ... , t_n) | **Maybe** t
		-- 기본 언어
표현	Expression∋e ::=	() | r | (e_1, ... ,e_n) | **Nothing** | **Just** e | $f\ e$ | **let** x = e **in** e' | x | **case** e **of Just** x => e_1 **else** e_2 | **case** e **of** (x_1, ... ,x_n) => e'
		-- 반응적 언어
시그널	Signal∋s ::=	**input** | **time** | **let snapshot** x <- s_1 **in** s_2 | **ext** e | **delay** e s | **let signal** $\{z_j(x_j) = u_j\}$ **in** s | u
스위춰	u ::=	s **until** $<s_j => z_j>$

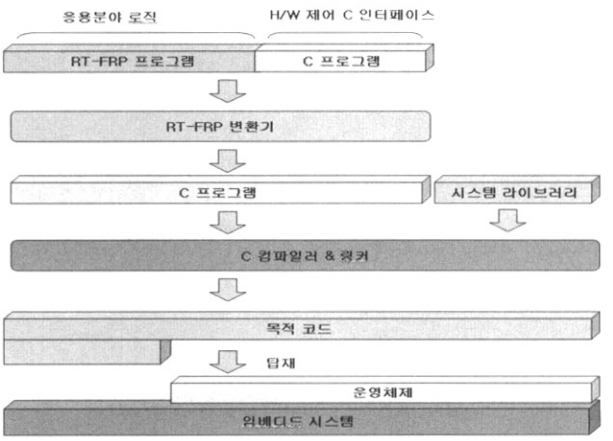
(그림 2) RT-FRP 언어 구문

3. 임베디드 시스템을 위한 RT-FRP 변환기

3.1 RT-FRP 프로그램의 변환과정

RT-FRP 언어로 작성한 프로그램은 (그림 3)과 같이 RT-FRP 변환기와 C 컴파일러를 거쳐 탑재 가능한 프로그램으로 변환된다. 최상위 단계에서는 RT-FRP 프로그램을 읽어 변환하는 변환기가 필요하다. 본 논문에서 구현한 RT-FRP 변환기는 RT-FRP 프로그램을 읽어 C 프로그램으로 변환한다. 중간 코드로 C 언어를 이용한 이유는 대부분의 임베디드 시스템 프로세서에서 C 컴파일러를 지원하기 때문이다.

RT-FRP 소스 내에는 C 프로그램이 내장될 수 있는데, 내장된 C 코드는 하드웨어 인터페이스를 기술하기 위해 이용된다. 변환기에 의해 생성된 C 프로그램은 C 크로스 컴파일러 및 링커를 거쳐 수행 코드로 변환된다. 임베디드 시스템은 운영체제가 탑재될 수도 있고 운영체제 없이 바로 응용프로그램만 탑재될 수도 있다. 현재 RT-FRP 변환기는 위 두 경우를 모두 지원한다. 운영체제에 탑재된 응용프로그램을 생성할 경우, 하드웨어와 인터페이스를 위해 시스템 콜을 이용한다.



(그림 3) RT-FRP 프로그램의 변환과정

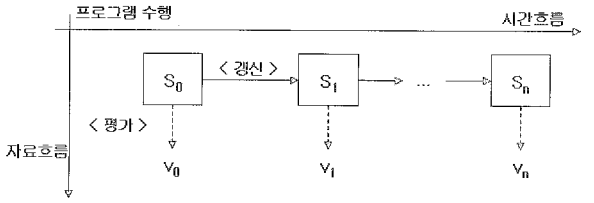
3.2 RT-FRP 프로그램에서 C 프로그램으로 변환

컴파일러를 작성할 때 매우 중요한 것은 의미를 보존하는 것이다. 이를 위해 이 논문의 변환기는 RT-FRP의 기능적 의미론과 수행 모델[10]을 기반으로 구성되었다. RT-FRP 각 구문의 기능적 의미론은 평가 규칙(evaluation rule)과 갱신 규칙(update rule)으로 정의된다. 평가 규칙은 특정 시점에서 시그널 값을 계산하기 위해 이용되며, 갱신 규칙은 다음 단계의 시그널을 결정하기 위해 이용된다. 시간의 영향을 받지 않는 기본 언어에서는 평가 규칙만 정의되어 있다.

또한 각 규칙에서는 변수 환경(variable environment)과 연속적 변수 환경(continuous variable environment)이 사용되는데, 이는 let snapshot이나 let signal 같이 변수를 사용하는 구문에서 이용된다. (그림 4)는 기능적 의미론의 두 가지 규칙을 기반으로 하는 RT-FRP 프로그램의 수행 모델이다.

메인 시그널을 s라고 할 때, RT-FRP 프로그램의 수행은 s에 대해 평가 규칙과 갱신 규칙을 프로그램이 중단될 때까지 반복하여 적용하는 것이다. 메인 시그널의 처음 상태를 s_0라고 하자. 시그널의 현재 상태 s_0의 값 v_0는 평가 규칙을 통해 얻을 수 있다. 시그널의 다음 상태 s_1은 현재 상태 s_0에 갱신 규칙을 적용함으로써 얻을 수 있다. 따라서 RT-FRP 프로그램은 환경변수를 기반으로 평가규칙 코드와 갱신규칙 코드를 무한히 반복하는 C 프로그램과 동등하다고 볼 수 있다. (그림 5)는 변환된 C 프로그램의 전체 구조이다. 메인 함수의 초기화 코드와 종료 코드는 RT-FRP 소스 내에 내장된 C 프로그램이 삽입되는 부분이다. 실제 구동되는 타겟 시스템에 대한 초기화 코드와 종료 코드가 여기에 삽입된다.

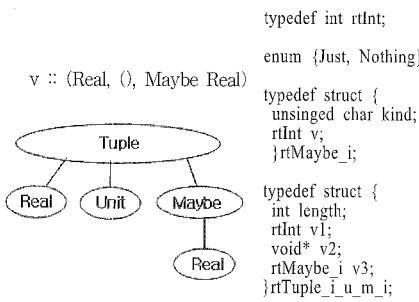
RT-FRP의 Real 타입은 C의 int 타입으로, 유닛은 void 타입으로 변환한다. Maybe와 튜플 타입은 바로 변환이 되지



(그림 4) RT-FRP 프로그램 수행 모델

```
타입 정의 코드
int main() {
   환경변수 선언 코드
   초기화 코드
   while (1) {
      평가규칙 코드
      갱신규칙 코드
   }
   종료 코드
}
```

(그림 5) 변환된 C 프로그램의 전체구조




```
typedef int rtInt;

enum {Just, Nothing};

typedef struct {
 unsinged char kind;
 rtInt v;
 }rtMaybe_i;

typedef struct {
 int length;
 rtInt v1;
 void* v2;
 rtMaybe_i v3;
}rtTuple_i_u_m_i;
```

(a) 타입 v의 구조 (b) 타입 v를 C 프로그램으로 변환한 결과

(그림 6) RT-FRP 타입 변환 예

```
TRs     :: Signal   → CStm
TRS2DS  :: Signal → (Name, TEnv, VEnv) → CStm
TRS2EE  :: Signal → (Name, VEnv) → CExp
TRS2ES  :: Signal → (Name, VEnv) → CStm
TRS2US  :: Signal → (Name, VEnv) → CStm
TRE2DS  :: Expression → (Name, TEnv, VEnv) → CStm
TRE2EE  :: Expression → (Name, TEnv) → CExp
TRT2TE  :: Type → CExp
sType   :: Signal → TEnv → Type
eType   :: Expression → TEnv → Type
```

(그림 7) 변환기에서 사용되는 변환함수

않기 때문에 C의 구조체 타입으로 정의하여 사용한다. RT-FRP는 컴파일 단계에서 모든 구문에 대한 타입이 결정되기 때문에 프로그램에서 사용하는 타입만 선택적으로 미리 정의할 수 있다.

(그림 6)은 Real, (), Maybe Real 타입으로 구성된 튜플 타입에 대한 변환 예를 나타낸 것이다. RT-FRP의 의미 분석 후 결정된 타입은 (그림 6-(a))과 같은 트리 구조를 가진다. C 프로그램의 타입 정의 부분은 트리의 리프(leaf) 노드에서부터 부모 노드로 변환된다. 트리의 가장 하위 노드에 있는 Real은 rtInt 타입으로 정의되며, Real을 포함하는 Maybe는 maybe_i 이름의 구조체 타입으로 정의된다. maybe_i는 Nothing과 Just를 구별하기 위해 kind가 포함되며 rtInt타입의 변수를 가진다. 튜플을 구현한 구조체는 원소의 수를 저장하기 위해 length가 포함되며, 튜플의 원소는 v1,v1,...,vn 형식의 변수로 구성된다.

RT-FRP 변환기는 각 구문에 대해서 환경변수, 평가규칙, 갱신규칙에 해당하는 코드를 생성하기 위해 변환함수(translation scheme)를 이용한다. 변환함수는 RT-FRP 표현 또는 시그널을 받아, C 표현 또는 문장으로 변환한다. 추가적으로 각 기호에 대한 타입정보 또는 변수이름을 갖는 환경이 매개변수로 이용된다. (그림 7)은 변환기에서 사용되는 변환함수이다.

각각의 변환 함수는 RT-FRP의 18가지(기본 언어 구문 13가지, 반응적 언어 구문 5가지) 구문마다 다르게 정의되어 있다. TRs는 최상위 단계의 변환 함수로서 다른 변환 함수를 이용하여 환경 변수 선언 코드, 평가 규칙 코드, 갱신 규칙 코드를 생성한다. 각 TR 함수는 X2YZ 형태의 첨자로 구별하고 있는데, 여기서 X는 함수의 입력 구문 종류, Y는 변환 종류, Z는 출력 구문 종류를 나타낸다. X가 S일 경우 Signal, E일 경우 Expression, T일 경우 Type에 해당한다. Y가 D일 경우 변수 선언 부분, E일 경우 평가 규칙 코드, U일 경우 갱신 규칙 코드이다. Z가 S일 경우 C의 문장, E일 경우 C의 표현에 해당한다. 예컨대, TRS_{2ES}는 RT-FRP의 Signal 구문에 대해 평가 규칙에 해당하는 C 문장 코드를 생성하는 변환 함수이다.

3.3 구현 환경

RT-FRP 변환기는 3.2 절에서 정의한 변환함수를 기반으로 구현하였다. 변환기는 크게 어휘분석기(lexical analyzer),

구문분석기(syntax analyzer), 의미분석기(semantic analyzer), 코드생성기(code generator)로 구성된다. 변환기를 작성하는데 이용된 프로그래밍 언어로는 하스켈이며, GHC 6.4.1 버전의 컴파일러를 이용했다.

구문분석기는 구문분석기 생성도구(parser generator)인 Happy 1.5 버전을 이용하여 작성하였다. 어휘 또는 구문분석기에서 효율적인 오류처리와 오류 메시지를 전달하기 위해 모나딕 파서(monadic parser)을 기반으로 하였다. 의미분석기와 코드생성기는 애로우(Arrows)[15] 기반의 모듈화 방식을 이용하여 작성하였다. 주로 애로우의 do 구문[16]을 이용하여 기술하였으며, 효율적인 오류처리를 위해 애로우 기반의 오류처리 기법[17]을 이용하였다.

4. 실 험

4.1 실험방법(RT-FRP를 이용한 레고 마인드스톰 제어)

본 실험에서는 컴퓨터를 내장한 조립용 로봇 키트인 레고 마인드스톰을 임베디드 시스템으로 선정하였다. 마인드스톰의 두뇌에 해당하는 RCX(Robotics Command System)는 8-bit Hitachi H8/3292 16MHz 프로세서를 내장하고 있으며, 3개의 입력 단자와 3개의 출력 단자, 액정 디스플레이, 4개의 버튼, 적외선 단자로 구성되어 있다[13]. 입출력 단자에는 입력센서와 모터가 연결되는데, 마인드스톰 기본 패키지(Robotics invension system 2.0)는 누름을 감지하는 터치센서, 빛의 세기를 감지하는 광학센서, DC모터를 제공한다. 적외선 단자는 PC 또는 다른 RCX와의 통신에 이용되며, 응용 프로그램을 다운로드 하거나 PC와 메시지를 주고받을 때 사용된다.

RCX는 BrickOS[18]와 GCC(GNU Compiler Collection) 크로스 컴파일러를 통한 개발환경을 지원한다. RCX 응용프로그램은 BrickOS의 시스템 콜[18]을 이용할 경우 쉽게 하드웨어를 제어할 수 있다. 때문에 BrickOS을 이용하면 RCX 하드웨어 제어를 위해 특별히 장치 관리자를 구현할 필요가 없다. 변환기가 생성한 C프로그램은 GCC 크로스 컴파일러와 링커를 통해 탑재 가능한 목적파일로 변환된다.

이 실험에서는 소형 임베디드 시스템인 마인드 스톰을 대상으로 RT-FRP 변환기가 올바르게 C 프로그램으로 변환을 수행하는지, 임베디드 시스템에 FRP를 적용하기 위해 RT-FRP 변환기를 이용하는 것이 적합한지를 알아본다. 구체적으로 두 가지 프로그램을 C, FRP, RT-FRP 언어로 작성하였다. RT-FRP로 작성한 프로그램이 FRP와 동등한 표현력을 제공하는지를 평가하기 위해 소스 코드의 라인수를 비교하였으며, 가용 메모리의 제약이 있는 소형 시스템에 탑재 가능한지를 평가하기 위해 목적파일의 크기를 비교한다. 본 논문의 주요 목적은 FRP의 선언적 프로그래밍 방법을 임베디드 시스템에 적용할 수 있도록 변환기를 작성하는 것이다. 따라서 변환된 RT-FRP 프로그램에 대한 수행 시간 성능에 대한 평가는 제외하였다.

실험에서는 하스켈 기반의 FRP, RT-FRP, C 각각의 언어를 이용하여 RCX 제어 프로그램을 작성하였다. RCX에

```
#include "lego.frp"
touch_time : Int
touch_time = let snapshot t <- inputTouch() in
             let snapshot ti <- timer () in
             let signal { stop (_) = (ext 0) until [ext t => go],
                          go (_) = (ext 1) until [ext t => stop] }
             in (ext 1) until [ext (gt (ti,10)) => stop, ext t => stop]
main : ()
main = let snapshot m <- touch_time ()
       in motorOut (m, m)
```

(그림 8) RT-FRP 이벤트제어 프로그램

```
#include "lego.frp"
detection : Int -> Int -> (Int, Int)
detection l r = if (l=BLACK & r=WHITE) then (BREAK,FWD) else
                if (l=WHITE & r=BLACK) then (FWD,BREAK) else
                (FWD,FWD)
motor : (Int, Int)
motor = let snapshot s1 <- inputLight1() in
        let snapshot s2 <- inputLight2() in
        ext (detection (s1, s2))
main : ()
main = let snapshot m <- motor() in
       let snapshot l <- ext (case m of (le1,re1) => le1) in
       let snapshot r <- ext (case m of (le2,re2) => re2) in
       motorOut(l,r)
```

(그림 9) RT-FRP 라인트레이서 프로그램

탑재하기 위해 중간 코드로 C 코드를 생성해야 되는데, 하스켈 기반의 FRP는 하스켈 컴파일러와 FRP 라이브러리의 문제로 인하여 C 프로그램을 생성하지 못하였다. 따라서 하스켈 FRP의 경우에는 목적코드를 생성할 수 없었다. 이로 인해서 하스켈 FRP의 경우에는 목적코드 크기를 비교하지 못하였다((그림 11) 참고). RT-FRP의 경우 구현한 변환기를 이용하여 C 프로그램을 생성하였다. RT-FRP는 RCX의 입력센서나 모터에 접근하기 위해서는 C 인터페이스로 제공되는 BrickOS의 시스템 콜을 직접 이용하였다. 실험을 위해 두 가지 응용프로그램을 작성하였다. (그림 8)은 RT-FRP 단순한 응용프로그램으로 외부 장치의 이벤트와 내부 상태에 대한 이벤트에 반응하여 동작하는 프로그램이다.

(그림 8) 프로그램은 로봇이 앞으로 10초 동안 진행하다가 멈추거나, 터치센서가 동작하면 멈추도록 하는 프로그램이다. 멈춘 상태에서 터치센서가 동작하면 앞으로 전진한다. 이 프로그램의 특징은 터치센서에 의한 외부 이벤트와 타이머에 대한 내부 이벤트에 반응하여 시스템의 상태가 전환된다는 것이다. RT-FRP 프로그램에서는 let signal 구문을 사용하여 정지(stop) 또는 진행(go) 두 상태를 정의하였으며, timer 시그널을 이용하여 현재 진행 시간을 검사하였다.

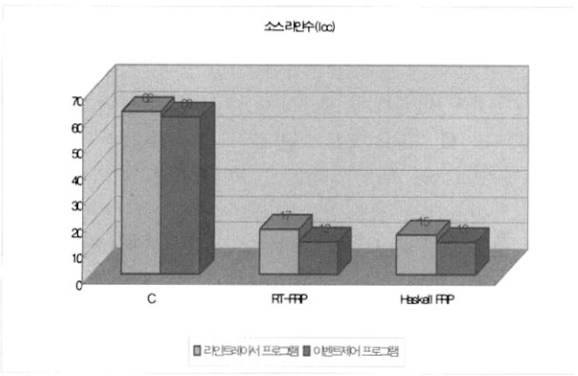
(그림 9)는 RT-FRP로 작성한 라인트레이서 프로그램이다. 라인트레이서는 검정 선을 따라가는 로봇으로 두 개의 입력 광학센서의 값에 따라 모터의 좌우 방향 동작한다. (그림 9)의 detection은 광학센서의 입력 값에 따라 출력 모터의 값을 결정하는 함수이다. 즉 양쪽의 광학센서의 값이 흰색 또는 검정일 경우 앞으로 전진하며, 왼쪽 광학센서의 값이 검정일 경우 오른쪽 바퀴만 동작, 오른쪽이 검정일 경우

왼쪽 바퀴만 동작한다. motor는 출력모터의 시그널로 입력 광학센서와 detection 함수로 인하여 상태가 결정된다. (그림 9)의 main은 이미 정의한 motor 시그널과 시스템의 모터 인터페이스와 연결하는 역할만 한다.

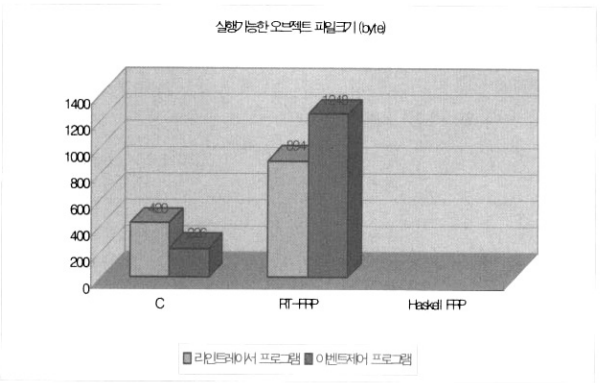
4.2 실험결과

4.1절에서 언급하였듯이 본 실험은 RT-FRP 변환기를 이용하여 FRP를 임베디드 시스템에 이용하는 방법이 적합한지를 평가한다. 실험에서는 소스 프로그램의 라인수를 간결한 표현력의 기준으로 정하였으며, 목적파일의 크기를 탑재 가능한 메모리를 가진 시스템의 기준으로 정하였다. RT-FRP 변환기의 목표는 FRP와 동등한 표현력으로 작성된 프로그램이 명령형 스타일로 작성된 C 프로그램과 동등한 크기의 목적파일 크기를 가지는 것이다. 4.1절에서 기술한 프로그램의 소스코드 라인수와 목적 파일의 크기를 비교하였다. (그림 10)은 각 언어별로 소스 프로그램의 라인수를 비교한 것이며, (그림 11)은 컴파일러와 링커를 통해 생성된 목적 파일의 크기를 비교한 것이다.

(그림 10)에서 볼 수 있는 바와 같이 RT-FRP와 FRP의 경우 이벤트제어 프로그램을 12줄 이내로 구성할 수 있었다. 반면 C에서는 약 60줄 정도로 표현되었다. 라인트레이서 프로그램의 경우에도 RT-FRP에서는 17줄 이내로 표현되었으며, C에서는 약 62줄 정도로 구성되었다. 실험결과 RT-FRP도 FRP와 동등한 수준의 표현력을 가지는 것으로 볼 수 있다. 반면 FRP나 RT-FRP 비해 C는 라인수가 많은데, 그 이유는 다음과 같다. C언어와 같은 명령형 언어는 문장(statement)의 조합으로 프로그램이 구성된다. 문장은 프로그램이 컴퓨터에 내려지는 명령으로 볼 수 있으며, 일반적으로 라인 단위로 구분된다. 따라서 시간에 따른 로봇의 구체적인 동작 문장으로 일일이 기술한 C는 자연스럽게 라인수가 FRP 보다 많아진 것으로 사료된다. 이에 반해 RT-FRP는 언어의 수행모델에서 구체적인 동작이 수행되며, 실제 프로그램에서는 시간에 따른 로봇의 동작을 시그널 타입으로 표현하기 때문에 C보다 라인수가를 1/5 수준 이상으로 줄일 수 있다.



(그림 10) 각 언어별 프로그램 코드 라인 수 비교



(그림 11) 각 언어별 실행 가능한 오브젝트 파일 크기 비교

RT-FRP 변환기와 C 컴파일러에 의해 생성된 목적 파일의 크기는 RT-FRP의 경우 C에 비해 다소 증가한 것으로 나타났다. (그림 11)에서 확인할 수 있는 바와 같이 라인트레이서의 C 코드의 경우 420byte이며, RT-FRP의 경우 894byte이다. 이벤트제어 프로그램의 C코드의 경우에는 226byte이며, RT-FRP의 경우에는 1248byte이다. 순수 C로 작성한 프로그램보다 RT-FRP 변환기에 의해 생성된 C 프로그램의 목적파일이 더 크다. 그 이유는 변환 규칙에 의해 RT-FRP 프로그램은 RT-FRP 수행모델과 동등한 코드를 생성하며, 자동으로 생성된 코드 중 불필요한 코드에 대한 최적화기 이루어지지 않았기 때문이다. 예를 들어 변환 규칙에 의해 생성된 변수 중에 값을 전달(propagation)만 하는 변수가 생길 수 있는데, 이에 대한 최적화가 없으며, 튜플과 같은 C에 없는 타입을 구현하기 위해 사용된 코드 때문에 목적코드 크기가 커진 것으로 본다. RT-FRP 변환기의 목표는 순수하게 작성된 C와 동등한 수준의 목적파일의 크기를 가지는 것인데, 실험결과 목표에는 미치지 못하였다. 하지만 일반적으로 C에 비해 약 10배 이상의 목적파일 크기를 갖는 하스켈 프로그램의 목적파일 보다는 작은 크기이며, 소형 메모리를 가진 RCX에는 충분히 탑재될 수 있는 크기이다.

5. 결 론

본 논문에서는 고 수준의 프로그래밍 언어인 FRP를 가용 메모리가 작은 임베디드 시스템에 적용시키는 방법을 제시하였다. 구체적으로 하스켈 기반의 FRP 대신 실시간 반응적 시스템을 위해 정의된 RT-FRP를 이용하였으며, RT-FRP 프로그램을 실제 시스템에 탑재하기 위해 RT-FRP 변환기를 작성하였다. 또한 실제 테스트 프로그램을 각 언어별로 작성하여 언어의 표현력과 목적코드의 크기 두 부분을 비교하였는데, 다음과 같은 특징을 살펴볼 수 있었다.

먼저 각 언어별로 작성된 소스 프로그램의 라인수를 비교함으로서 응용분야에 대한 언어의 간결한 표현력에 대해서 살펴보았다. 실험 결과에서 나타나듯이 FRP와 RT-FRP는 C 프로그램에 비해 약 5배 정도 간결하게 표현되었다. RT-FRP는 추상화된 수행모델에 의해서 반복적인 동작 상

황이 숨겨지므로 구체적인 수행 코드를 기술하지 않는다. 반면 C 언어는 반복문과 같은 구문을 이용하여 각 상태에 따른 동작 상황을 일일이 기술하기 때문에 소스 프로그램의 라인수가 증가하게 된다. FRP와 RT-FRP은 언어의 구문과 표현방법이 유사하기 때문에 실제 프로그램에서도 동등한 라인수로 프로그램이 구성되었다.

선언적인 프로그래밍은 프로그램에 대한 특징(what)만 기술하지 구현(how)는 기술하지 않는다. 함수형 언어의 대표적인 특징으로 구현부를 직접 기술하는 명령형 언어에 비해 간결하게 프로그램을 작성할 수 있다. 이러한 관점에서 라인수의 감소는 프로그램의 가독성(readability), 유지보수(maintenance), 신뢰성(reliability), 빠른 프로토타입 개발(fast prototype)과 같은 장점을 포함한다.

컴파일러에 의해 생성된 목적파일의 크기는 임베디드 시스템에 프로그램 탑재 여부와 직접적으로 관련이 있다. 하스켈에 내장된 FRP는 FRP 라이브러리와 하스켈 컴파일러의 문제로 인하여 C 프로그램으로 변환을 못하였고 RCX 프로세서에 대한 하스켈 크로스 컴파일러 또한 제공되지 않기 때문에 RCX에 탑재 가능한 목적 파일을 생성하지 못하였다. 그리하여 C와 RT-FRP 프로그램 목적 파일만 생성하여 비교하였다.

동일하게 수행하는 두 프로그램에 대해서 RT-FRP 프로그램의 목적파일의 크기가 C 프로그램의 목적파일에 비해 크게 생성되었다. 이는 RT-FRP 변환기의 초기 목표인 C 프로그램과 대등한 수준의 목적파일을 생성하는 것에는 미치지 못하였다. 변환된 RT-FRP 프로그램이 최적화가 되지 않았기 때문으로 사료되며, 향후 최적화 단계를 거치면 RT-FRP 프로그램의 목적파일도 C 프로그램과 대등한 수준이 될 것으로 판단된다. 실험에서 C 프로그램과 RT-FRP 프로그램 목적파일의 크기는 RCX에 탑재 가능할 정도로 크기가 작았으며, 실제 RCX에 탑재하여 시스템을 구동하였다. 따라서 현재 하스켈 컴파일러가 지원되지 않고 시스템의 가용 메모리가 매우 작은 RCX와 같은 임베디드 시스템에 FRP를 적용하기 위해서는 본 논문에서 구현한 RT-FRP 변환기를 이용하는 방법이 적합하다고 볼 수 있다.

최근 저비용, 고품질의 프로그램을 작성하기 위한 방법으로 응용분야 특화 언어에 관한 연구가 많이 이루어지고 있다. 응용분야 특화 언어는 응용분야에 적합한 표현력을 제공함으로 프로그램을 간결하게 표현할 수 있다. 또한 제공된 구문만 이용하기 때문에 오류발생 가능성이 적어 저비용으로 고품질의 프로그램을 작성할 수 있다는 특징이 있다. 본 연구는 이러한 관점에서 임베디드 시스템에 탑재되는 고품질의 프로그램을 작성하기 위한 방법으로 볼 수 있다.

6. 관련 연구

현재 FRP의 연구동향은 크게 두 가지로 분류된다. 하나는 하스켈 내장언어로 구현된 FRP를 기반으로 하여 다양한 응용영역에 적용시키는 연구가 있고, 또 다른 하나는 RT-FRP 언어처럼 FRP의 특징을 가지며 완전히 독립된 언어를 기반으로 하는 연구가 있다.

하스켈 기반의 FRP 프로그램은 응용분야의 명세와 유사한 것이 가장 큰 특징이었으며, 이로 인하여 하스켈 기반의 FRP는 많은 응용분야에 적용할 수 있었다[6,7,8,9]. 하지만 FRP는 이벤트 처리 시 불 필요한 시간적 공간적 비용에 대한 낭비가 있었다. 이 문제로 인하여 FRP는 새롭게 구현되었으며, 에로우[15,16]라는 강력한 모듈화 도구와 결합하여 AFRP[19,20]라는 이름으로 발표되었다. AFRP는 FRP의 시간적 공간적 낭비에 대한 문제점을 보완하였으며, 언어의 일등급값을 입출력의 개념이 포함된 시그널 함수(signal function)로 정하여 기존의 FRP 보다 강력한 모듈화와 직관적인 표현력을 제공한다. 하지만 AFRP 또한 하스켈에 내장된 언어이므로 하스켈 프로그램의 수행환경과 동일한 제약사항을 가지게 된다.

반응적 시스템에 RT-FRP를 적용하기 위해 RT-FRP를 오토마타 형태로 변역하는 연구가 있었다[21]. 이 연구에서는 부분계산을 이용하여 RT-FRP 오토마타 최적화를 시도하였다. 하지만 이 연구에서도 RT-FRP 번역 방법의 방향만 제시하였을 뿐, RT-FRP 프로그램이 실제 동작하는 구현물을 제시하지는 못했다. 본 연구에서는 실제 임베디드 시스템에서 RT-FRP 프로그램을 동작시킬 수 있도록 변환기를 작성하였다.

7. 결론 및 향후연구과제

본 논문에서는 FRP를 소형 임베디드 시스템에 적용할 수 있는 방법으로 RT-FRP 변환기를 설계하고 구현하였다. 변환기는 RT-FRP 프로그램을 입력으로 받으며, 다양한 임베디드 시스템을 지원하기 위해 C 프로그램을 생성한다. 일반적으로 널리 알려진 레고 마인드스톰의 RCX를 실험 대상으로 선정하였으며, 각 언어별로 동일한 프로그램을 작성하여 실험을 진행하였다. 실험 결과 RT-FRP 변환기는 소스 프로그램의 간결한 표현력과 RCX에 탑재 가능할 정도의 크기를 가지는 목적파일을 생성하였다. 또한 RT-FRP 변환기에 의해 생성된 프로그램은 실제 RCX에 탑재되어 정상적으로 수행되었다.

고수준의 추상화를 제공하는 RT-FRP를 임베디드 시스템에 적용함으로서 기존의 명령형 언어인 C 보다 약 5배 이상 간결하게 시스템을 구성할 수 있었으며, 반응적 시스템에 특화된 RT-FRP 구문을 이용하기 때문에 보다 직관적이며 선언적으로 시스템을 표현할 수 있다. 비록 RT-FRP 목적파일의 크기가 C 목적파일 만큼 작은 크기에는 미치지 못하였지만, 실험 결과 RCX와 같은 작은 가용공간의 시스템에도 충분히 이용될 수 있었다. 따라서 본 연구는 RT-FRP 변환기와 C 크로스 컴파일러를 통하여 저수준의 임베디드 시스템에서 고 수준의 언어를 이용하여 프로그래밍을 할 수 있는 것을 보였다.

본 논문에서 구현한 RT-FRP 변환기는 RT-FRP 프로그램과 동일한 기능적 의미론을 수행하는 C 프로그램을 생성하였다. 하지만 RT-FRP 프로그램 자체를 분석하여 예측할 수 있

는 시간적, 공간적 비용과 변환된 C 프로그램의 수행 시 실제로 필요로 하는 시간적, 공간적 비용이 동일한지는 증명하지 못하였다. 이를 위해 향후 RT-FRP 변환기의 의미 분석 단계에서 수행 시간과 공간을 측정할 수 있어야 할 것이다. 또한 튜플과 같이 구조체 형태로 구현되는 타입의 경우 함수 호출 후 반환 값을 스택을 이용하여 복사하기 때문에 비용이 많이 든다. 따라서 구조체를 반환 값으로 갖는 함수 호출 시 포인터를 이용하는 방법을 고려하면 공간 성능을 더 개선할 수 있을 것이다.

향후 RT-FRP 언어 및 변환기에서도 AFRP와 같은 추상화 기법을 적용하기 위한 방법을 고려해 보아야 할 것이다. 또한 RT-FRP 언어가 보다 실용적으로 사용되기 위해서 기존의 언어에서 유용하게 이용되는 병행성(concurrency), 예외처리(exception)와 같은 고급 기능이 언어 차원에서 지원되어야 할 것이다.

참 고 문 헌

[1] Thierry Gautier, Paul Le Guernic, and Loic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, Functional Programming Languages and Computer Architecture, volume 274 of Lect Notes in Computer Science, edited by G. Goos and J. Hartmanis, pp.257-277. Springer-Verlag, 1987.

[2] P. Caspi , D. Pilaud , N. Halbwachs, J. A. Plaice, LUSTRE: a declarative language for real-time programming, Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, p.178-188, Jan. 1987.

[3] Gérard Berry. The Esterel v5 Language Primer Version 5.21 release 2.0. Centre de Mathématiques Appliquées, Ecole des Mines and INRIA, 2004 Route des Lucioles, 06565 Sophia-Antipolis, Apr. 1999.

[4] 강인혜, 양진석, "초보자를 위한 Esterel 프로그래밍" 홍릉과학출판사, pp3-101, 2005.

[5] Z. Wan and P. Hudak, "Functional Reactive Programming from first principles," in Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI'00), pp.242-252, 2000.

[6] C. Elliott and P. Hudak, "Functional reactive animation," in International Conference on Functional Programming, pp. 163-173, June 1997.

[7] A. Courtney and C. Elliott, "Genuinely functional user interfaces," in 2001 Haskell Workshop, Sep. 2001.

[8] J. Peterson, P. Hudak, A. Reid, and G. Hager, "FVision: A declarative language for visual tracking," in Proceedings of PADL'01: 3rd International Workshop on Practical Aspects of Declarative Languages, pp.304-321, Jan. 2001.

[9] J. Peterson, P. Hudak, and C. Elliott, "Lambda in motion: Controlling robots with haskell," in First International Workshop on Practical Aspects of' Declarative Languages (PADL), Lecture Notes in Computer Science, Vol. 1551, pp. 91-105, Springer, Jan. 1999.

[10] Z. Wan, "Functional Reactive Programming for Real-time Reactive System," Ph.D Dissertation, Computer Science Department, Yale University, Oct. 2002.

[11] P. Hudak, "Modular domain specific languages and tools," in Proceedings of Fifth International Conference on Software Reuse, pp. 134-142, Jun. 1998.

[12] GHC Team, "The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4.2" (http://www.haskell.org/ghc/docs/latest/html/building/sec-port-info.html)

[13] RCX Internal, http://graphics.stanford.edu/~kekoa/rcx

[14] Z. Wan, W. Taha, and P. Hudak, "Real-time FRP," in International Conference on Functional Programming (ICFP'01), pp146-156, 2001.

[15] J. Hughes, "Generalising monads to arrows," Science of Computer Programming, vol. 37, pp. 67-11, May.2000.

[16] R. Paterson, "A new notation for arrows," in International Conference on Functional Programming, pp.229-40, ACM Press, Sep. 2001.

[17] 이동주, 지정훈, 장한일, 우균, "애로우를 이용한 오류 처리 기법" , 한국컴퓨터종합학술대회논문집, Vol.33, No.1(B), pp 397-399, 8월. 2006.

[18] BrickOS Homepage, http://brickos.sourceforge.net

[19] H. Nilsson, A. Courtney, and J. Peterson, "Functional reactive programming, continued," in Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02), (Pittsburgh, Pennsylvania, USA), pp.51-64, ACM Press, Oct. 2002.

[20] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, robots, and functional reactive programming," in Summer School on Advanced Functional Programming 2002, Oxford University, vol. 2638 of Lecture Notes in Computer Science, pp.159-187, Springer-Verlag, 2003.

[21] D.N.Xu and S.C.Khoo, "Compiling real time functional reactive programming," in ASIA-PEPM 2002: Preceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation, (New York, NY, USA), pp.83-93, ACM Press, 2002.

이 동 주

e-mail : mrlee@pusan.ac.kr
2005년 경성대학교 컴퓨터과학과(학사)
현 재 부산대학교 컴퓨터공학과 석사과정
관심분야 : 프로그래밍언어 및 컴파일러, 함수형 언어

우 균

e-mail : woogyun@pusan.ac.kr
1991년 한국과학기술원 전산학과(학사)
1993년 한국과학기술원 전산학과(석사)
2000년 한국과학기술원 전산학과(박사)
현 재 부산대학교 정보컴퓨터공학부 조교수
관심분야 : 프로그래밍언어 및 컴파일러, 함수형 언어, 그리드컴퓨팅, 소프트웨어 메트릭 등