JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 26 No. 2, pp. 1-9, February 2021
https://doi.org/10.9708/jksci.2021.26.02.001

# Deterministic Parallelism for Symbolic Execution Programs based on a Name-Freshness Monad Library

Ki Yung Ahn*

*Assistant Professor, Dept. of Computer Engineering, Hannam University, Daejeon, Korea

## [Abstract]

In this paper, we extend a generic library framework based on the state monad to exploit deterministic parallelism in a purely functional language Haskell and provide benchmarks for the extended features on a multicore machine. Although purely functional programs are known to be well-suited to exploit parallelism, unintended squential data dependencies could prohibit effective parallelism. Symbolic execution programs usually implement fresh name generation in order to prevent confusion between variables in different scope with the same name. Such implementations are often based on squential state management, working against parallelism. We provide reusable primitives to help developing parallel symbolic execution programs with unbound-genercis, a generic name-binding library for Haskell, avoiding sequential dependencies in fresh name generation. Our parallel extension does not modify the internal implementation of the unbound-generics library, having zero possibility of degrading existing serial implementations of symbolic execution based on unbound-genecrics. Therefore, our extension can be applied only to the parts of source code that need parallel speedup.

▸ **Key words**: Multicore, Deterministic Parallelism, Symbolic Execution, Name Binding, Haskell

## [요 약]

본문에서는 순수 함수형 언어인 하스켈로 작성된 심볼릭 실행의 병렬화를 위한 상태 모나드 기반의 라이브러리에 결정적 병렬화를 적용하기 편리한 API를 설계/구현하고 멀티코어 컴퓨터에서 벤치마크를 통해 실제 성능을 향상을 확인해 본다. 일반적으로 순수 함수형 프로그램은 병렬화가 쉽다고 알려져 있으나 실제 구현에서 핵심 알고리듬 외적인 부분에서 의도치 않은 순차적 데이터 의존성의 발생으로 병렬화가 어려워질 수 있다. 심볼릭 실행 구현에서는 지금껏 사용했던 변수와 겹치지 않는 새 이름을 생성함으로써 서로 다른 범위의 이름이 같은 변수끼리 혼동하는 착오를 피하는 방식을 종종 활용한다. 그런데 이를 순차적 상태 관리로 구현한 경우가 많아 병렬화에 걸림돌이 된다. 이 논문에서는 하스켈의 범용적 이름 관리 라이브러리인 unbound-generics의 새 이름 생성 기능에 순차적 의존성을 회피할 수 있는 확장 기능을 제공함으로써 병렬적 심볼릭 실행 구현을 간소화하는 데 기여하였다. 우리가 구현한 병렬화 확장의 특징은 기존 unbound-generics 라이브러리의 내부 구현을 그대로 유지한 상태의 확장이라는 점으로, 기존에 unbound-generics로 작성된 순차적 심볼릭 실행기의 성능 저하 우려가 전혀 없다는 점이다. 따라서 병렬화가 필요한 부분에만 확장 기능을 적용하는 방식으로 활용하여 성능을 개선할 수 있다.

▸ **주제어**: 멀티코어, 결정적 병렬성, 심볼릭 실행, 이름 바인딩, 하스켈

• First Author: Ki Yung Ahn, Corresponding Author: Ki Yung Ahn
*Ki Yung Ahn (kya@hnu.kr), Dept. of Computer Engineering, Hannam University
• Received: 2021. 01. 07, Revised: 2021. 02. 03, Accepted: 2021. 02. 03.


http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

## I. Introduction

함수형 프로그래밍은 같은 입력에 대해 항상 같은 결과값이 보장된 순수 함수를 활용함으로써 프로그램의 어느 부분부터 실행하든 결과가 달라지지 않는 특징이 있어 병렬화에 유리하다고 알려져 있다. 특히 멀티코어 CPU 등 병렬 HW가 일반화된 시대에 기존 프로그램 소스코드를 전면 수정하지 않고도 병렬화로 성능을 높일 가능성을 주목받는 프로그래밍 패러다임이다. 그런데 이론적으로 병렬화에 적합한 알고리듬이라도 실제 구현에서 부수적 혹은 부속적으로 사용된 순차적 구조 때문에 효과적 병렬화가 어려워질 수 있다. 예컨대 로그 고유번호를 부여하기 위해 로그를 남길 때마다 고유번호를 증가시키는 기능을 전역 상태를 순차적으로 1씩 증가시키는 방식으로 구현했다면 알고리듬의 핵심적 구조와 무관한 부가 기능 때문에 효과적인 병렬화가 곤란하다. 이를 해결하기 위해서는 기존의 로그 생성 모듈 혹은 라이브러리를 병렬성이 고려된 다른 것으로 대체하거나 하는 등의 다른 방법을 찾아야 한다.

심볼릭 실행에서는 다른 범위에 있는 같은 이름의 변수들을 혼동하는 착오를 막기 위해 지금껏 사용된 적 없는 새로운 이름을 생성하는 경우가 많다. 새로운 이름을 생성하는 손쉬운 방법은 앞서 언급한 로그 고유번호와 마찬가지로 번호를 저장하는 전역 상태를 1씩 순차적으로 증가시키며 $x_0, x_1, x_2, \ldots$ 처럼 생성하는 것이다. 기존에 이런 방식의 구현이 많으며 그런 코드를 포함하는 모듈이나 라이브러리로 이름을 관리하는 심볼릭 실행 프로그램은 순차적 번호 관리 방식을 대체하지 않고서는 제대로 병렬화되지 않는다. 하스켈 프로그래밍 언어에서 변수 등 이름을 포함한 심볼릭 데이터를 다루기 편하게 돕는 범용적 라이브러리인 unbound-generics[1]도 이렇게 정수 상태값을 1씩 증가시키는 방식을 사용하고 있어 이를 그대로 사용하면 병렬화하기 곤란하다.

본분에서는 unbound-generics 라이브러리의 내부를 수정하지 않고 병렬화를 돕는 추가 API 함수를 제공함으로써 기존 라이브러리의 장점을 그대로 활용하며 필요한 부분에만 비교적 손쉽게 프로그램을 병렬화를 적용할 수 있도록 하였다. 병렬화 API 구현이 의도대로 동작하는지 시험하기 위해 unbound-generics로 간단한 예시용 프로그래밍 언어 문법 및 실행기를 정의하고 (III절), 그에 대한 병렬화 코드에 성능 개선 효과가 있는지 알아보는 실험을 진행하여 순차적으로 작성된 부분의 성능을 저해하지 않으면서도 병렬화로 성능이 개선됨을 확인하였다 (IV절).

## II. Preliminaries

### 1. Styles of parallelism supported in Haskell

하스켈 컴파일러(GHC)는 병렬화를 위한 다양한 프로그래밍 방식을 지원하고 있다. 우선 가벼운 사용자 스레드를 생성하고 상호배제적 잠금장치를 통해 스레드를 조율하는 전통적인 방식도 지원한다. 이런 방식은 함수형 언어가 아닌 언어에서도 활용되던 방식으로 실행할 때마다 스레드 스케줄링에 영향을 받아 따라 다른 결과가 나올 수 있어 *비결정적 병렬성*(nondeterministic parallelism)이라 부른다. 이에 대비되는 개념인 *결정적 병렬성*(deterministic parallelism)도 지원하는데 이는 부수효과(side-effect)가 제한되는 함수형 언어의 특성을 잘 활용하는 방식이라 볼 수 있다. GHC에서는 결정적 병렬성도 조금 다른 두 가지 방식을 제공하고 있다. 첫째는 어느 부분의 계산식에 대한 병렬화를 시도할지 힌트만 주고 나머지는 GHC 런타임이 알아서 자동으로 병렬화하도록 맡기는 방식이다. 둘째는 계산이 진행되는 동안 데이터의 흐름이 몇 갈래로 나뉘고 어디서 다시 합쳐지는지 분기점과 병합점을 코드로 명시하여 분기되어 진행되는 갈래에서 병렬화가 적용되도록 더 세밀하게 프로그래머가 지정하는 방식이다.

이후로 본문에서는 상기 언급된 여러 병렬화 방식 중에서도 결정적 병렬화의 첫째 방식을 다룬다. 대부분의 병렬화 과정을 GHC 런타임에 자동으로 위임하는 방식이므로, 기존 코드의 변화를 최소화하며 병렬성을 활용하기에 적합하다. 즉, 기존 라이브러리로 작성된 심볼릭 실행 코드에 간단히 병렬화 기능을 추가하고자 하는 논문의 취지에 부합하는 병렬화 방식이다.

### 2. Monadic code and deterministic parallelism

이 절에서는 간단한 하스켈 코드를 통해 GHC에서 제공하는 병렬화 기능을 소개하고, 그에 대응되는 모나드 활용 코드에서는 왜 병렬화가 어려워지는지 설명한다.

순수한 계산식 e1 + e2의 병렬화를 위해 다음과 같이 GHC에서 제공하는 par 연산자를 활용할 수 있다.

e1 `par` e2 `par` (e1 + e2)

위 식에서 par 연산자가 하는 역할은 (e1 + e2)의 계산에 앞서 e1과 e2의 계산을 병렬적으로 시작해도 된다고 GHC에게 알려주는 것이다. 위의 par를 사용한 식의 계산 결과도 e1 + e2의 계산 결과와 항상 같다, `par`의 왼쪽 식의 계산을 병렬적으로 진행하되 전체의 결과는 오른쪽 식의 계산 결과를 취하는 것이다. 병렬화 효과로 par를 활용한 식의 수행 시간이 원래 계산식보다 단축될 수 있다. 이를

테면 e1과 e2를 단독으로 계산하는 데 각각 10초씩 걸린다면 병렬화되지 않은 e1 + e2의 계산에는 20초가 걸리겠지만 e1 `par` e2 `par` (e1 + e2)가 실행되는 시점에서 GHC 런타임에 2개 이상 병렬 작업 처리가 가능한 상황이라면 e1과 e2가 동시에 계산되고 그 둘의 결과에 대해 덧셈만 한번 해 주면 되므로 10초 정도의 시간 안에 계산을 두 배 정도 빨리 끝마칠 수 있다.

문제는 현실적인 구현에서는 처음부터 끝까지 최종 결과에 필요한 순수한 계산만으로 이루어지는 프로그램은 많지 않다는 점이다. 핵심 계산의 전후로 사용자 입력을 받고 출력된 결과를 확인하는 등의 전처리 후처리 기능도 있으며, 핵심적 계산 진행 중에도 로그 기록, 에러 처리 등 프로그램의 원활한 실행을 돕는 부가적 기능이 핵심 알고리듬의 흐름과 나란히 진행되는 경우가 많다. 이러한 부수효과(side-effect)를 핵심 알고리듬 진행과 함께 나열하면 프로그램의 핵심적인 흐름을 알아보기 어려워지며 핵심 알고리듬의 구조상에서는 독립적이던 부분이 부수효과로 인해 상호 영향을 미치는 추가적인 의존성이 발생하기도 하여 프로그램 유지보수의 비용이 커진다.

이런 유지보수 문제의 해결을 위해 컴파일러 의미론의 이론적 도구로써 쓰이던 모나드(monad)란 개념을 하스켈에서 언어 기능으로 도입하여 표준라이브러리에서 지원함으로써 부수효과를 동반한 계산을 잘 정돈된 코드로 작성하는 것을 돕고 있다. 하스켈을 시초로 다른 프로그래밍 언어에도 이런 방법이 전파되고 있으며 비동기 병렬 프로그래밍이 이슈로 대두되며 최근 모나드의 프로그래밍 활용에 관한 관심이 확대되고 있다. 비유하자면 핵심 계산 진행 과정에 나타나는 값을 부수효과와 함께 상자에 넣어 정돈해 두되, 계산 시작과 마지막에서만 상자 전체를 열어 초기 및 최종 핵심 결과와 부수적 상태를 확인하자는 것이다. 또, 계산 진행 중에는 핵심 계산에 필요한 값을 확인하기 위해 상자를 드나드는 흐름만을 위주로 표현하고 그 이외의 상자 내용은 드러내 보이지 않는 것이 모나드를 활용한 코드의 특징이다. 특정 지점에서 부수효과를 발동시킬 필요가 있을 때는 상자를 직접 열어보지 않고 각 부수효과의 특성에 맞게 상자 종류마다 정해진 API로 접근하도록 하여 부가적인 의존성을 최소화한다.

다음은 모나드를 활용해 e1 + e2에 대응되는 코드를 작성한 예시로 do로 시작하는 프로그램 블록에서 <- 표기와 return 연산을 활용하고 있다. 참고로 각 줄의 오른쪽에 --로 시작하는 부분은 한 줄 주석이다.

```
do    e1 <- g1              -- (e1, s1) = g1 s0
      e2 <- g2              -- (e2, s2) = g2 s1
      return $ e1 + e2      -- (e1+e2, s2)
```

위 코드는 g1과 g2라는 상자로 비유한 부수효과를 포함할 수도 있는 묶음으로부터 <- 표기법으로 e1과 e2를 꺼내 e1 + e2라는 식의 계산을 통해 핵심 결과값을 계산한다. 마지막 줄의 return은 부수효과 없이 순수하게 핵심 값만으로 상자를 구성하는 연산이다. 위 do 블록 전체를 g3라 정의한다면 다른 do 블록에서 <- 연산으로 g3에서 값을 얻어옴으로써 e1 + e2의 결과에 해당하는 값을 얻게 된다.

모나드는 상자를 사용하는 방식에 대한 일종의 규약(interface)에 해당하고 실제 활용되는 구체적인 모나드의 종류는 다양하다. 여기서는 이 논문의 관심사가 되는 라이브러리 구현 내부에서 활용하는 *상태 모나드*를 기준으로 계속 설명을 진행하겠다. 위의 do 블록을 상자로 포장할 때 s0라는 초기 상태를 넣어두고 계산을 시작했다고 하자. 이 때, 첫째 줄의 g1을 함수로 해석하면 s0를 인자로 받아 e1과 다음 상태 s1의 두 값을 순서쌍으로 계산해 낸다고 볼 수 있다. 마찬가지로 둘째 줄의 g2를 인자 s1에 적용해 e2와 s2를 계산한다. 마지막 줄의 return은 순수한 핵심 값으로만 상자를 구성하므로 이전 줄의 상태 s2를 그대로 가져와 순서쌍을 구성한다.

이제 위의 상태 모나드 활용 코드의 병렬화를 시도하자. 일단 가장 간단히는 앞서 이 절의 초반에 par를 활용해 e1 + e2를 병렬화한 것과 똑같이 마지막 do 블록 마지막 줄에 아래와 같이 par 연산자 활용을 시도해 볼 수 있다.

```
do    e1 <- g1              -- (e1, s1) = g1 s0
      e2 <- g2              -- (e2, s2) = g2 s1
      return $ e1 `par` e2 `par` (e1 + e2)
```

안타깝지만 일반적으로 이런 상태 모나드 코드는 제대로 병렬화되지 않는다. 왜냐하면, 위 do 블록 각 줄의 코드는 이전 줄의 상태에 의존하기 때문이다. (e1,s1)은 g1의 인자인 s0에 그리고 (e2,s2)는 g2의 인자인 s1에 의존한다. 바로 함수의 인자와 결과라는 형식적 의존성 때문에 e2를 포함한 묶음 (e2,s2)의 계산이 완료되기 전에 s1을 포함한 묶음 (e1,s1)의 계산을 GHC 런타임에서 진행하지 못한다. 하지만 상당수의 경우 e1과 e2는 상태 없이 순수한 계산만으로도 정의 가능한데 주변의 다른 연결되는 계산과의 일관성이나 조합 일관성을 위해 모나드가 적용된 상태로

작성하기도 한다. 이럴 때 프로그램을 작성하는 입장에서 e1과 e2가 상태 독립적임을 알기에 논리적으로는 병렬화 가능함을 알지만 모나드라는 프레임워크의 형식적인 제약 때문에 병렬화에 불편함이 발생하는 문제가 있다.

### 3. Unbound-generics Library

Unbound-generics 라이브러리는 하스켈에서 나무(tree) 구조를 정의하는 재귀적 데이터 타입을 선언할 때, 프로그래밍 언어 문법의 변수 범위 개념을 다루기 적합한 *이름 바인딩*(name binding) 구조를 유연하게 정의할 수 있으며, 정의된 문법 데이터 타입으로부터 관련 편의 기능까지 자동으로 유도해 주는 편리한 범용적(generic) 라이브러리다. 예컨대, 람다계산법(λ-calculus)의 문법을 아래와 같이 정의할 수 있다.

```
type Nm = Name Expr  -- Expr을 나타내는 이름

data Expr
      = Var Nm                  -- x
       | Lam (Bind Nm Expr)     -- (λx. e)
       | App Expr Expr          -- (e1 e2)
      deriving (Show, Generic, Typeable)
```

위 데이터 타입 선언에 몇 줄의 기본적 명세만 추가하면 Expr의 이름 바인딩 구조에 맞게 이름의 범위를 따지며 치환하는 subst 함수의 정의를 자동으로 유도한다. 예컨대 (App (Lam (bind x (Var x)) x)는 ((λx. x) x)를 나타낸다. ((λx. x) x)에서 자유로운 이름 x를 y로 치환하라는 subst x (Var y) (App (Lam (bind x (Var x)) x)를 실행한 결과는 람다계산법의 이름 범위를 따르는 규칙에 맞게 ((λx. x) y)를 나타내는 (App (Lam (bind x (Var x)) y)가 된다. (λx. x) 안에 있는 x가 치환되지 않는 이유는 여기서 x는 함수의 형식인자로 그 범위가 함수 안에서만 유효하게 묶인 이름이기 때문이다. 따라서 ((λx. x) x)의 마지막에 있는 x만 치환의 대상이 된다.

그뿐만 아니라 FIg. 1처럼 새로운 문법 요소를 추가하더라도 다른 부분을 변경 없이 subst와 같은 유틸리티 함수가 변경된 문법 정의에 맞게 자동으로 유도된다. 즉, subst 함수를 직접 변경하지 않아도 Fig. 1의 새로운 Expr 문법 정의를 컴파일하기만 하면 x + ((λx. x) x)에서 x를 y로 치환해 y + ((λx. x) y)에 해당하는 식이 나오도록 subst 함수의 정의가 변경된 Expr에 맞게 자동으로 유도된다. subst 함수처럼 기본적이고 기계적인 정의지만 매

```
type Nm = Name Expr  -- Expr을 나타내는 이름

data Expr
  = Var Nm                  -      -- x
   | Lit Integer                   -- n
   | Lam (Bind Nm Expr)            -- (λx. e)
   | App Expr Expr                 -- (e1 e2)
   | Add Expr Expr                 -- e1 + e2
   | AddP Expr Expr                -- e1 |+| e2
   | If Expr Expr Expr  -- if e then e1 else e0
  deriving (Show, Generic, Typeable)
```

Fig. 1. Syntax of the λ-calculus extended with addition (+), parallel addition (|+|), and conditional expressions, defined as a Haskell datatype using the unbound-generics library.

번 직접 작성하기 번거로운 이름 관리와 관련된 유틸리티 함수를 문법 정의가 수정될 때마다 그에 맞는 함수 정의로 자동으로 유도해 주는 것이 unbound-genereics의 편리한 점이다. 따라서 unbound-genereics로 이름의 범위를 명세하며 Fig. 1처럼 문법을 정의하면 subst와 같은 유틸리티 함수를 활용해 심볼릭 실행기를 작성하고 유지보수하는 데 좋다.

## III. Supporting parallelism in a name-freshness monad

람다계산법에 덧셈식과 조건문을 추가한 언어의 문법을 Fig 1.에 정의하였다. 두 종류의 덧셈 연산자가 있는데 Add는 양쪽 인자를 순차적으로 계산해 더하고 AddP는 양쪽 인자를 병렬적으로 계산해 더하라는 의미를 부여하고자 한다. 본문에서 글로 설명할 때는 하스켈 코드 형식의 Add e1 e2나 AddP e1 e2보다는 수식에 가까운 e1 + e2나 e1 |+| e2와 같은 형태의 유사코드도 활용하며 설명을 진행하겠다. 조건문 **if** e **then** e1 **else** e0는 조건식 e의 계산 결과가 정수일 때만 계산이 진행된다. e의 계산 결과가 0이면 e0를 계산하고 그 외의 값이면 e1을 계산한다.

이런 의미대로 작성한 심볼릭 실행기의 (라이브러리 import를 제외한) 전체 소스코드가 Fig 2.에 나타나 있다. unbound-generics에서 지원하는 unbind와 subst 등을 활용하여 함수에 인자를 넘기며 계산하는 람다계산법에서 핵심 계산 규칙인 β줄임(줄번호 2-3)을 beta 함수로 구현하였다. 이 beta 함수는 심볼릭 실행기에서 람다식으로 작성된 함수를 인자에 적용해 호출하는 부분 (줄번호 15)에 활용되고 있다.

```
1. beta (App (Lam b) e2) = do
2.                         (x,e) <- unbind b
3.                         return $ subst x e2 e
4. beta _              = empty
5.
6. redN :: Bool -> Expr -> FreshMT Maybe Expr
7. redN _          e@(Var _) = pure e
8. redN _          e@(Lit _)  = pure e
9. redN intoLam e@(Lam b)
10.   | intoLam   = do (x,e1) <- unbind b
11.                      lam x <$> red e1
12.   | otherwise = pure e
13.   where red = redN intoLam
14. redN intoLam e@(App e1 e2)
15.   | isLam  e1   = beta e >>= red
16.   | hasRed e1   = App <$> eval e1 <*> pure e2
17.                         >>= red
18.   | hasRed e2   = App <$> pure e1 <*> red  e2
19.   | otherwise    = pure e
20.   where eval = redN False
21.           red  = redN intoLam
22. redN intoLam e@(Add e1 e2) = do
23.     e1' <- red e1
24.     e2' <- red e2
25.     case (e1',e2') of
26.         (Lit n1, Lit n2) -> pure $ Lit (n1 + n2)
27.         _                -> pure $ Add e1' e2'
28.     where red  = redN intoLam
29. redN intoLam e@(AddP e1 e2) = do
30.     [e1',e2'] <- forkFreshMT rpar fromJust
31.                                  [red e1, red e2]
32.     case (e1',e2') of
33.         (Lit n1, Lit n2) -> pure $ Lit (n1 + n2)
34.         _                -> pure $ Add e1' e2'
35.     where red  = redN intoLam
36. redN intoLam e@(If eb e1 e0) = do
37.     eb' <- red eb
38.     case eb' of Lit 0 -> red e0
39.                 Lit _ -> red e1
40.                 _     -> pure $ If eb' e1 e0
41.     where eval = redN False
42.           red  = redN intoLam
43.
44. reduceN = redN True
```

Fig. 2. Symbolic evaluator/reducer for the λ-calculus with addition, parallel addition, and conditional expressions.

Fig 2.의 소스코드 중에서 여기서 자세히 살펴볼 부분은 순차 덧셈(줄번호 22부터)과 병렬 덧셈(줄번호 29부터)을 처리하는 부분이다. 두 연산 모두 e1과 e2를 간소화한 e1'과 e2'가 모두 정수 상수의 형태(Lit n1, Lit n2)인지 확인하여 n1+n2의 덧셈을 수행난다. 심볼릭 실행기라 함은 상수로 계산되는 값만이 아닌 자유 변수를 포함하는 식도 처리하기 때문에 x + (3 + 4)와 같은 식이 있다면 x + 7로 간소화하고 x + (y + 4)와 같은 식은 더이상 간소화되지 않고 그 자체로 계산이 완료되므로 Add e1' e2'와 같은 상수 형태가 아닌 결과도 가능하므로 이를 처리하는 부분이 27줄 34줄 등의 코드이다.

지금 살펴보는 순차 덧셈 및 병렬 덧셈 관련 소스코드 중에서도 특히 주목해야 할 부분은 23-24줄과 30-31줄의 차이점이다. 23-24줄은 II-2절에서 소개한 do 블록에서 모나드 활용 방식으로 여기에 단순히 par 연산자만 첨가해서는 병렬화가 곤란하다. 따라서 병렬 덧셈을 구현하려 30-31줄에서처럼 이름을 관리하는 Fresh 모나드 안에서 병렬화 지원을 위해 새롭게 정의한 forkFreshMT 함수를 활용하였다.

```
forkFreshMT :: (Monad m1, Monad m2) =>
    Strategy b ->
    (m1 a -> b) ->
    [FreshMT m1 a] -> FreshMT m2 [b]
forkFreshMT strat runm ms =
  splitFreshMT strat runm $ zip (repeat 0) ms

splitFreshMT :: (Monad m1, Monad m2) =>
    Strategy b ->
    (m1 a -> b) ->
    [(Integer, FreshMT m1 a)] -> FreshMT m2 [b]
splitFreshMT strat runm ps = do
  s <- FreshMT $ get
  let (ks,ms) = unzip ps
      ss = scaln (+) s ks
      ps' = zip ss ms
  FreshMT $ modify (+ last ss)
  return . runEval $ parList strat
       [runm $ contFreshMT m s' | (s',m)<-ps']
```

Fig. 3. Definitions of primitives for deterministic parallelism in the name-freshness monad from unbound-generics library.

Fig. 3은 unbound-generics에서 이름 관리를 위해 제공하는 Fresh 모나드 안에서 결정적 병렬성을 활용하기

위해 설계한 API 함수를 직접 구현한 내용이다. Fig. 2의 redN에 쓰인 forkFreshMT는 splitFreshMT의 특별한 경우로서 이를테면 forkFreshMT strat runm [m1,m2,m3]는 splitFreshMT strat runm [(0,m1),(0,m2),(0,m3)]를 해당한다. 여기서 (0,m1)이 뜻하는 바는 m1의 실행 결과로 외부에 공개되는 새로운 이름이 전혀 없다는 뜻이다. 람다계산법 실행에 핵심적으로 쓰이는 beta 함수(Fig. 2 줄번호 1-4)를 살펴보자면 unbind b로 새로운 이름 x가 생성되더라도 곧이어 e에 나타나는 x를 e2로 치환하는 subst x e2 e로 인해 return되는 결과에는 새 이름 x가 포함되지 않는다. 이런 람다계산법의 특성을 알고 있기에 forkFreshMT를 FIg. 2에서 활용한 것이다, 일반적으로는 각각의 병렬 계산이 최대 몇 개의 새로운 이름을 생성할지 지정해 splitFreshMT를 활용하도록 설계했다. 이를테면, splitFreshMT strat runm [(4,m1),(3,m2),(5,m3)] 실행 직전에 새로 생성될 이름의 인덱스로 활용할 상태값 s=0 이었다면 m1은 4개($x_0, x_1, x_2, x_3$), m2는 3개($x_4, x_5, x_6$), m3는 5개($x_7, x_8, x_9, x_{10}, x_{11}$) 이내의 새로운 이름을 만들어낼 수 있으며 splitFreshMT 이후의 코드에서는 s=11인 상태로 이어서 Fresh 모나드 계산을 계속한다. 대개 실행 효율 향상을 기대하고 병렬화한다는 점을 고려해 실제로 splitFreshMT로 수행할 각 병렬 계산에서 지정한 개수를 초과하여 이름을 생성했는지를 splitFreshMT 함수에서 알아서 검사해 주지는 않는다. 개발자가 필요시 각 병렬 계산(m1, m2 ,...,)의 결과에 최종 인덱스 상태를 포함하여 (계산 끝에 FreshMT $ get으로 얻어와서) 실제 몇 개의 새로운 이름이 각 병렬 계산마다 생성되었는지 원래 설정 개수와 비교하여 검사하는 코드를 추가로 작성할 수 있다.

## IV. Benchmark

이번 절에서는 우리가 설계한 Fresh 모나드 병렬화 API 함수(Fig. 3)가 Expr 심볼릭 실행기(Fig. 2)에서 병렬 덧셈(AddP 또는 |+|로 표기)을 제대로 병렬화했는지 확인하는 벤치마크 결과를 보고한다. 벤치마크는 최적화되지 않은 재귀적 피보나치 함수를 Expr 문법으로 작성해 그 계산 과정 일부분에 병렬 덧셈을 사용했을 때 실행 시간이 단축되는지 측정하는 방식으로 수행했다. 벤치마크에 사용된 피보나치 함수 정의는 Fig. 4에 제시하였다. 참고로 Expr 문법에는 뺄셈이 없으므로 x-1이나 x-2 형태의 의사코드는 사실 음수를 더하는 x+(-1)나 x+(-2)의 형태로 작성하는 것이 Expr 문법 정의에 부합하나 어차피 의사코드인 만큼 가독성을 위해 뺄셈 형태로 표기했을 뿐이다.

피보나치 수열의 일정 구간에 대해, 순차 덧셈만 사용해 병렬화하지 않은 피보나치 함수(fibo)의 심볼릭 실행 시간과 비교해, 최상위 함수 호출 즉 재귀호출 깊이 1까지만 병렬 덧셈(|+|)하는 함수(fibo')에 GHC 런타임 옵션 -N2로 최대 2개까지의 병렬 작업을 허용한 심볼릭 실행한 시간 및 재귀호출 깊이 2까지 병렬 덧셈(|+|)을 사용한 함수(fibo'')에 GHC 런타임 옵션 -N4로 최대 4개까지의 병렬 작업을 허용한 심볼릭 실행 시간을 측정해 정리하였다.

```
Y    = (λf.((λx. f (x x)) (λx. f (x x))))

fibo = Y (λf.λx. if x then ( if x -1 then
                    f (x -1) + f (x -2)
                    else 1 ) else 1 )

fibo' = λx. if x then ( if x -1 then
                    fibo (x -1) |+| fibo (x -2)
                    else 1 ) else 1 )

fibo'' = λx. if x then ( if x -1 then
                    fibo' (x -1) |+| fibo' (x -2)
                    else 1 ) else 1 )
```

Fig. 4. Fibonacci function (in Expr pseudo-code format) parallelized up to recursion depth 0, 1, 2.

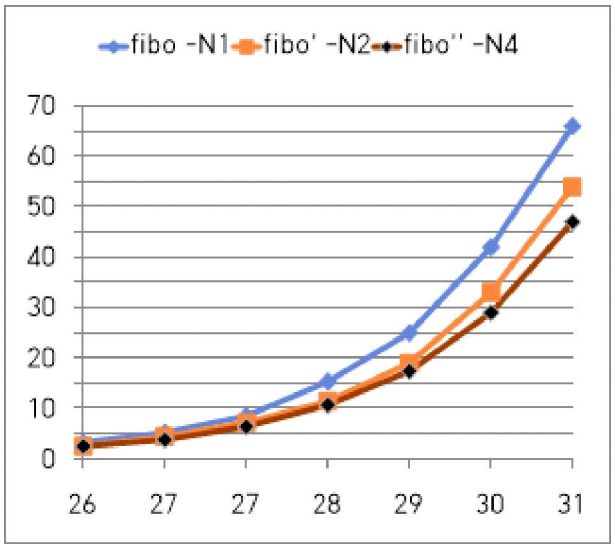


Fig. 5. Benchmark result demonstrating the effect on symbolic execution times of the Fibonacci functions from Fig. 4 due to their varying degrees of parallelism.

프로그램 실행 시간의 측정은 GHC 런타임에서 제공하는 -s 옵션을 활용하여 프로그램 종료 직후 표준 출력으로 보고되는 내용을 기반으로 기록하였다. 참고로 벤치마크는 4개 코어에서 8개 하이퍼스레드를 지원하는 인텔 코어 i7-8565U CPU 및 16GB 주메모리를 갖춘 컴퓨터의 WSL2 가상화 환경에 데비안 배포판 및 GHC 8.8.4가 설치된 환경에서 진행되었다.

Fig. 5의 가로축은 피보나치 함수에 넘기는 인자로 0번째와 1번째 값이 1로 시작하는 피보나치 수열(1, 1, 2, 3 ,5, 8, 13, ...)의 몇 번째를 계산할지 나타내며 세로축은 해당하는 피보나치 수를 계산하는 심볼릭 실행 시간을 초 단위로 나타낸다. 그래프에서 시각적으로 확인할 수 있듯이 재귀함수 호출 깊이의 1과 2까지 병렬화 정도가 높아짐에 따라 그래프가 아래로 떨어지며 실행시간이 단축됨을 관찰할 수 있다. 이로써 우리가 제공하는 Fig. 3의 Fresh 모나드 병렬화를 위한 API 함수들이 설계 의도대로 작동함이 확인되었다. 그래프에서 가로축 왼쪽으로 갈수록 간격이 좁아지지만, 계산 시간이 큰 순서는 -N1, -N2, -N4으로 유지된다는 것은 병렬화 기능 적용으로 인해 직렬 계산 부분에는 부담을 주지 않는다고 분석할 수 있는 근거가 된다. 낮은 순번의 피보나치 수열에는 병렬화로 인한 성능 개선의 효과를 볼 만큼의 계산량이 많지 않으므로 만일에 순차적 계산을 방해하는 요소가 강하게 작용한다면 그래프가 역전되는 현상이 일어날 수 있기 때문이다.

참고로 이번 벤치마크에서 사용한 피보나치 수열 알고리듬은 두 개의 재귀함수 호출이 처리하는 작업량이 대칭적이지 않기 때문에 (하나는 n-1에 대해 다른 하나는 n-2에 대해) 단순한 병렬화로는 실행 시간을 절반에 가깝게 줄이기는 어렵다. 또한, 심볼릭 실행기를 통해 계산하고 있으므로 값이 정해지지 않은 자유 변수를 고려하지 않아도 되는 일반적 실행기(인터프리터)보다 성능이 낮다. 이 벤치마크는 병렬화 API의 정상적인 작동을 확인하기 위해 이해하기 쉬운 간단한 예제를 택했을 뿐이며 심볼릭 실행기를 활용하는 분야는 수치 계산보다는 컴파일러 최적화 과정에서 결과값을 유지한 채로 소스코드를 자동으로 변형한다거나 형식화된 시스템의 논리적 검증을 위해 모든 파라메터에 특정 값을 지정하지 않고도 간소화시킬 수 있는 부분만을 간소화하는 용도 등으로 활용된다.

## V. Related work

GHC에서 제공하는 결정적 병렬화 연산 par는 1992년에 ML 계열의 함수형 언어에 도입[2]한 것을 그 기원으로 볼 수 있다, 이후 당시의 하스켈 언어를 확장한 실험적인 병렬 하스켈 구현[3]을 통해 관련 연구가 발전하다가 2000년대 후반에 이르러 가장 대중적인 하스켈 컴파일러인 GHC의 런타임에서 멀티코어를 CPU를 활용한 병렬 컴퓨팅을 지원하기 시작[4]했다. 하스켈과 같은 함수형 언어 구현에서 내장된 병렬화 기능을 지원하기 이전에도 프로그래밍 언어 차원에서 병렬 계산을 지원하려는 시도들은 다양하게 존재했다. 대표적으로 80년대 일본의 5세대 컴퓨터 프로젝트[5]라는 병렬 하드웨어를 설계하고 논리 프로그래밍 언어를 병렬 처리함으로써 고성능 인공지능을 구현하려 시도한 연구 프로젝트가 있었다. 비록 당시 HW나 인공지능 관련 기술 발전의 한계 등의 시대적 한계로 인해 업계에까지 영향을 미칠만한 성과까지 이루지는 못했으나 이후에 병렬화를 고려한 프로그래밍 언어 설계 등에 큰 영감을 주었다.

최근 함수형 프로그래밍 관련 병렬화 동향을 알 수 있는 사례도 몇 가지 제시하겠다. 우선 결정적 병렬 프로그래밍 모델에서 메모리 효율성을 엄밀하게 검증하려는 연구[6]가 올해 저명 국제학회인 POPL 2021 우수 논문으로 선정되며 관심을 모았다. 하스켈 관련 연구로는 비정형 중첩 데이터 구조의 병렬화를 통해 성능을 개선하는 연구가 꾸준히 진행되고 있으며 그 최근 연구 결과로는 Euro-Par 2020에서 발표된 논문[7]을 들 수 있다. 이 연구는 비정형 중첩 데이터를 다루는 병렬 하스켈 소스코드를 분석하여 성능이 개선된 목적코드로 컴파일되도록 돕는 정적분석 기법 두 가지를 제시하고 이에 대한 벤치마크를 보고하고 있다. 이런 연구는 단지 순차적 알고리듬 병렬화로 성능을 개선을 넘어 이미 병렬화된 프로그램의 성능을 더욱 끌어올려 고성능 컴퓨팅 응용을 염두에 두는 연구 주제이다. 한편, 본문 주제로는 다루지 않는 비결정적 병렬성을 고수준으로 추상화하여 활용하는 소프트웨어 트랜잭셔널 메모리(software transactional memory, STM)를 GHC에서 지원하는데, 액터 모델 기반 병렬 프로그래밍 프레임워크 설계/구현에 GHC에서 제공하는 STM을 활용한 연구들[8,9]도 있다.

국내에서도 하스켈 병렬 프로그래밍 관련 연구를 여러 곳에서 진행하고 있다. 매니코어(수십 개 혹은 백 개 이상의 많은 멀티코어) 환경에서 일반적인 병렬 하스켈 프로그램의 성능 개선의 정도가 증가된 코어의 계산 성능만큼 충분치 못한 점을 해결하기 위해 메모리 튜닝 도구를 활용한 성능 개선 연구[10]나 하스켈로 구현한 쌍방시뮬레이션 검사 알고리듬에 병렬화와 메모이제이션을 함께 적용했을 때 성능 개선에 관한 연구[11] 등이 최근의 사례다.

이 논문에서 활용하고 있는 unbound-generics 라이브러리[1]는 2011년 발표된 논문[12]에서 구현한 이름 관리

를 위한 unbound 라이브러리를, 그 이후 GHC의 기본(base) 라이브러리에서 지원하기로 채택한 범용적 프로그래밍(generic programing) 프레임워크인 GHC.Generics를 기반으로 포팅한 것이다. 최근에는 GHC.Genreics가 자동으로 처리할 수 있는 데이터 타입(ADT)의 범위를 벗어난 더 일반적인 데이터 타입(GADT)까지도 자동으로 처리할 수 있는 kind-generics라는 최근에 개발된 범용적 프로그래밍 프레임워크를 기반으로 unbound-generics의 기능을 확장한 unbound-kind-generics 라이브러리[13]도 출시되었다. 참고로 데이터 타입 구조에 따라 그에 알맞은 동작을 지원하는 방식을 데이터 타입 범용적 프로그래밍(datatype generic programming)이라고 일컫는데, 이를 돕는 자동화 기능을 제공하는 범용적 프로그래밍 라이브러리(혹은 프레임워크)들이 개발되고 연구되어 왔다. 특히 하스켈 진영에서 이러한 움직임이 활발하여 하스켈의 서로 다른 범용적 프로그래밍 프레임워크의 장단점을 분석하는 논문[14]까지 발표되었을 정도이며 그 이후로도 최근의 kind-genercis같은 새로운 범용적 프로그래밍 프레임워크들도 개발되고 있다.

이 논문의 연구 동기인 상태 모나드 기반 라이브러리에서의 병렬화의 문제는 unbound-generics의 이름 관리 모나드에만 국한된 문제는 아니다. 순수 함수형 언어에서 상태 처리가 포함된 프로그램의 병렬화 문제에 대한 좀더 근본적인 해결을 위해 *상태 스레드 조합*(State Thread Composition)이라는 개념을 바탕으로 STCLang이라는 하스켈 라이브러리를 개발한 연구[15]가 2019년 하스켈 심포지움에 발표된 바 있다. 간단히 말하자면 기존에 하스켈에서 주로 활용하는 상태 모나드(StateT)를 STCLang에서 제공하는 상태 스레드 조합이 가능한 모나드로 대체하면 STCLang을 통해 제공하는 GHC의 내장 병렬화 기능을 손쉽게 활용할 수 있다는 것이다.

## VI. Conclusions

이 논문의 본론에서는 Fresh 모나드 안에서 결정적 병렬성을 활용하기 위해 설계한 API 함수를 직접 구현한 내용을 소개하였으며, unbound-generics를 활용해 작성한 심볼릭 실행의 성능을 병렬화를 통해 개선하는 프로그램을 작성하고 이를 벤치마킹하였다. unbound-generics에서 제공하는 이름 관리 모나드가 순차적 의존성에 따라 진행되는 상태 모나드 기반으로 구현되어 있어 단순히 par 연산자를 기존 프로그램 코드에 추가하는 것만으로는 병렬화가 어렵다는 문제를 해결하려는 것이 연구 동기였다. 연구의 결과로 기존 라이브러리의 내부 구현 수정 없이도 unbound-generics를 활용한 심볼릭 실행기와 같은 프로그램을 병렬화로 성능을 개선하려는 한정된 범위의 해결책을 제시하였다. 제시된 해결책으로 병렬화를 적용했을 때 성능 개선이 이루어짐을 확인하기 위해, 병렬 덧셈과 조건문을 포함한 람다식 문법(Expr)으로 작성한 피보나치 함수를 심볼릭 실행기로 처리하는 시간을 측정하는 벤치마크를 진행하였다. 벤치마크 결과 병렬 덧셈을 활용하는 피보나치 함수의 병렬 실행 시간이 순차 덧셈만으로 병렬화되지 않은 피보나치 함수의 실행 시간보다 단축된다는 것을 확인하였다.

장기적으로는 앞서 IV절의 관련 연구 마지막에 언급한 STCLang처럼 상태 처리와 함께 병렬화가 잘 적용되도록 처음부터 설계된 근본적인 해결책으로 기존 상태 모나드를 대체하는 것이 이상적이다. 하지만 단시일 내에 기존 상태 모나드 기반 라이브러리를 STCLang 기반으로 모두 대체하기는 현실적으로 불가능하다. 또한, 상태 모나드(StateT)가 많은 하스켈 프로그램에서 광범위하게 쓰이고 있어 기존의 상태 모나드를 GHC에서 특별히 통해 공을 들여 최적화하고 있다. 따라서 병렬화가 필요없고 순차적 상태만으로도 충분한 프로그램의 경우에는 STCLang으로 대체하면 오히려 성능에 손해를 볼 가능성도 있다. 그러므로 당분간은 기존의 상태 모나드 기반 프로그램의 병렬화가 필요할 경우 이 논문의 내용과 같이 각 라이브러리에 특화된 병렬화 해법을 적용하는 것이 단기간에 병렬화를 적용할 수 있는 접근방법이라 생각된다. 앞으로 STCLang과 같이 근본적 해결책을 향한 시도의 안정성과 범용성이 검증되고 개선되어 기존 상태 모나드의 순차 실행 성능이 유지되면서 병렬화가 필요한 경우 편리하게 활용할 수 있는 범용적 방식을 도입하는 방향으로 하스켈 생태계가 미래에는 발전하기를 기대해 본다.

## ACKNOWLEDGEMENT

This work was supported by the NRF grant 2018R1C1B5046826, funded by Korea government (MSIT).

## REFERENCES


[1] Unbound-generics: Support for programming with names and binders using GHC Generics. https://hackage.haskell.org/package/unbound-generics

[2] K. Hammond and S. L. Peyton-Jones. "Profiling Scheduling Strategies on the GRIP Multiprocessor". In Intl. Workshop on the Parallel Implementation of Functional Languages, pages 73-98, Aachen, Germany, Sept. 1992.

[3] P. W. Trinder, E. Barry Jr., M. K. Davis, K. Hammond, S. B. Junaidu, U. Klusik, H.-W. Loidl, and S. L. Peyton-Jones. "Low level Architecture-independence of Glasgow Parallel Haskell (GpH)". In Glasgow Workshop on Functional Programming, Pitlochry, Scotland, Sept. 1998.

[4] S. Marlow, S. L. Peyton-Jones, and S. Singh. *"Runtime Support for Multicore Haskell,"* In ICFP 2009, pp. 65-78, Aug. 2009. ACM Press. DOI: 10.1145/1596550.1596563

[5] T. Moto-oka. *"Overview of the Fifth Generation Computer System Project,"* Proceedings of 10th Annual International Symposium on Computer Architecture, (10th ISCA'83), SIGARCH Newsletter, pp. 417-422, June 1983. DOI: 10.1145/800046.801682

[6] J. Arora, S. Westrick, U. A. Acar. *"Provably Space-efficient Parallel Functional Programming,"* Proceedings of the ACM on Programming Languages, Volume 5, POPL, Article No.: 18, pp 1–33, ACM, January 2021. DOI: 10.1145/3434299

[7] L. B. van den Haak, T. L. McDonell, G. K. Keller, and I. G. de Wolff. *"Accelerating Nested Data Parallelism: Preserving Regularity,"* Euro-Par 2020: Parallel Processing - 26th International Conference on Parallel and Distributed Computing, LNCS 12247, pp. 426-442, Springer, August 2020. DOI: 10.1007/978-3-030-57675-2_27

[8] Y. Kim, J. Cheon1, T. Hur, S, Byun, and G. Woo. "SSAM: A Haskell Parallel Programming STM Based Simple Actor Model," Journal of Physics: Conference Series 1566 (ICCAI 2019), IOP Publishing, 2020. DOI: 10.1088/1742-6596/1566/1/0 12040

[9] E. Albert, N. Bezirgiannis, F. D. Boer, and E. Martin-Martin. "A Formal, Resource Consumption-Preserving Translation from Actors with Cooperative Scheduling to Haskell," Fundamenta Informaticae, 177(3-4), pp. 203-234. DOI: 10.3233/FI-2020-1988

[10] H. Kim, H. An, S. Byun, and G. Woo. *"Tuning the Performance of Haskell Parallel Programs Using GC-Tune,"* KIISE Transactions on Computing Practices, 23(8), pp. 459-465, August 2017. DOI: 10.5626/KTCP.2017.23.8.459

[11] K.Y. Ahn. "Parallelization of a Purely Functional Bisimulation Algorithm," Journal of KSCI, 26(1), January 2021. DOI: 10.9708/jksci.2021.26.01.011

[12] S. Weirich, B. A. Yorgey, and Tim Sheard. "Binders unbound," Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011, pp. 333-345, ACM, 2011. DOI: 10.1145/2034574.2034818

[13] Unbound-kind-generics: Support for programming with names and binders using kind-generics. https://hackage.haskell.org/package/unbound-kind-generics

[14] A. Rodriguez, Alexey and J. Jeuring, P. Jansson, A Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. *"Comparing Libraries for Generic Programming in Haskell,"* ACM SIGPLAN Notices 44(2), pp. 111-122, January 2009. DOI: 10.1145/1411286.1411301

[15] E. Sebastian, A. Justus, N. A Rink, A. Goens, and J. Castrillon. *"STCLang: state thread composition as a foundation for monadic dataflow parallelism,"* Haskell 2019: Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, pp 146–161, August 2019. DOI: 10.1145/3331545.3342600


## Authors

Ki Yung Ahn joined the faculty of the Department of Computer Engineering at Hannam University, Daejeon, Korea, in 2018. His research interests are parallelism, functional programming, programming languages, type systems, and cyroptographic protocols.