О представлении модельного времени при помощи механизмов функционального программирования[1]

[1] *Д.В.Буздалов <buzdalov@ispras.ru>*
[1, 3, 4] *А.К.Петренко <petrenko@ispras.ru>*
[1, 2, 3, 4] *А.В.Хорошилов <khoroshilov@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9*
[3] *Московский государственный университет имени М. В. Ломоносова Москва, 119991, ГСП-1, Ленинские горы, д. 1*
[4] *НИУ "Высшая школа экономики", 101000, Россия, г. Москва, ул. Мясницкая, д. 20*

Аннотация. Функциональное программирования играет все большую роль в современном компьютеризированном мире. Такой подход позволяет создавать более надежный, абстрактный и автоматически проверяемый код. Однако при этом эти техники мало используются при создании систем проектирования и моделирования ответственных систем. Данная работа является попыткой применения удачных техник функционального программирования для создания системы моделирования на примере системы динамического моделирования поведения систем для проверки характеристик, связанных с временем.

Ключевые слова: моделирование архитектуры; ответственные системы; моделирование поведения; модельное время; функциональное программирование; монады

DOI: 10.15514/ISPRAS-2018-30(6)-20

Для цитирования: Буздалов Д.В., Петренко А.К., Хорошилов А.В. О представлении модельного времени при помощи механизмов функционального программирования. Труды ИСП РАН, том 30, вып. 6, 2018 г., стр. 341-366. DOI: 10.15514/ISPRAS-2018-30(6)-20

[1] Исследования проводились при финансовой поддержке РФФИ в рамках проекта 17-01-00504.

1. Введение

Контекстом работы является создание моделей архитектуры ответственных систем с целью дальнейшей проверки этих моделей на соответствие функциональным и нефункциональным требованиям (требованиям отказоустойчивости, потребления ресурсов памяти, задержек при передаче информации и др.).

Архитектурные модели программно-аппаратных систем управления представляют структуру системы и набор спецификаций поведения отдельных компонентов и связей между ними, в том числе описывающих темпоральные свойства поведения. Соответственно, рассматриваются такие характеристики моделируемой системы, как длительность выполнения тех или иных задач моделируемой системой и размер задержек по передаче информации внутри системы.

В проводившихся ранее работах [1][2] авторы использовали для динамического анализа акторную модель, рассматривая отдельные компоненты системы, наделенные поведением, как отдельные независимые общающиеся акторы, работающие в модельном времени. При этом были выявлены следующие недостатки системы, основывающейся на акторной модели, при использовании в динамическом моделировании поведения моделей систем:

- нелокальность акторной системы
  - акторы рассчитывают на наличие определенных акторов (например, которые умеют обрабатывать определенные сообщения), при этом не могут гарантировать или проверить это;
  - нельзя автоматизированно статически (не тестированием) оценить корректность замены одного актора, на другой в случае, если третий актор, который пользуется заменяемым, рассчитывает на какие-нибудь свойства исходного (например, рассчитывает на возможность принять и полностью обработать определенный тип сообщения);
- нетипизированность акторной системы
  - принятие или непринятие сообщения решается только во время выполнения и не проверяется при компиляции;
- слабость средств для поддержки множественных, недетерминированных, вероятностных или интерактивных параметров на входе:
  - множественные параметры требуют множественных запусков с комбинаторным количеством запусков и повторением общей работы;
  - вероятностные и интерактивные параметры требуют специальной ad hoc обработки в каждом отельном работающим с этим акторе; это приводит к смешению собственной логики актора с реализацией получения этих параметров и приводит к заселению акторов посторонней логикой;

- недетерминизм требует или перезапусков (подобно множественным параметрам), или слишком раннего недетерминированного выбора отдельных параметров.

Акторная модель вычислений широко применяется в традиционном программировании, хотя при этом там не учитывается специальным образом модельное время, с которым необходимо работать при моделировании поведения систем реального времени (для проверки свойств модели, связанных со временем).

Одновременно с этим в программировании наблюдается тенденция противопоставления акторной модели вычисления функциональному программированию, которое позволяет решать (помимо других) те же проблемы, для которых была создана акторная модель. При этом функциональное программирование обладает рядом преимуществ. Аналогично тому, как взамен акторной модели применяются техники функционального программирования в областях, не связанных с моделированием времени, возникает идея попробовать применить аналогичные техники взамен акторной модели в контексте модельного времени, необходимого для поведенческого моделирования систем реального времени.

В данной работе предлагается способ организации поведенческих спецификаций для динамического моделирования систем реального времени, который учитывает модельное время и использует технику функционального программирования для решения проблем, сопутствующих моделированию при помощи акторов.

Само понятие функционального программирования не является четко определенным и порой интерпретируется несколько по-разному разными специалистами. В данной работе мы будем пользоваться понятиями, связанными с чистым типизированным функциональным программированием. Более детальное описание используемых терминов и понятий можно найти на странице проекта РФФИ[2]. Там читатель найдет краткое описание и примеры для таких понятий как *чистая функция*, *параметрический полиморфизм*, *функция высших порядков*, *функтор*, *монада*, *эффект (в чистом функциональном программировании)*, которые важны для понимания основного материала.

2. Эффекты в функциональном программировании

Так как данная работа использует технику моделирования эффектов в *чистом* функциональном программировании, следует остановиться на том, что мы понимаем под *эффектом* и как проходило развитие этого понятия в истории функционального программирования.

[2] Страница проекта гранта РФФИ 17-01-00504: https://forge.ispras.ru/projects/ductilejur

2.1 Побочные эффекты

Основным понятием в чистом функциональном программировании является понятие *чистой тотальной функции* [3][4], т.е. функции без *побочных эффектов*. Однако, в конечном итоге, программы должны производить побочные эффекты – например, читать и писать в базы данных, выводить информацию пользователю, читать и писать файлы, взаимодействовать с другими программами по сети и т.п. Таким образом, перед функциональным стилем программирования встает вопрос совмещения *чистоты* используемых функций и *побочных эффектов* для выполнения полезных действий.

Для того, чтобы совместить эти две, казалось бы, несовместимые вещи, в функциональном программировании используют тот факт, что *чистое* вычисление может возвращать достаточно произвольную структуру данных, в том числе структуру данных, которая внутри содержит описание вычисления, не являющегося *чистым* (т.е., *вычисление с побочным эффектом*).

При этом, если собственно запуск *вычисления с побочным эффектом* недоступен *чистому* коду, то не нарушается никаких предположений о чистом коде.

Таком образом, результатом работы чистого кода может быть описание *вычислений с побочным эффектом*, которое может быть далее выполнено внешним (по отношению к самой программе) исполнителем. Тем самым, производится разделение выполнения функциональной программы на две части:

- в первой части исходная программа, написанная на функциональном языке, вычисляет некую структуру данных, которая содержит помимо других вычислений инструкции, содержащие побочные эффекты, предназначенные для выполнения специальным исполнителем; на этом выполнение собственно программы заканчивается;
- во второй части специальный исполнитель исполняет инструкции с побочным эффектом из вычисленной специальной структуры данных.

В общем случае, может существовать несколько специальных исполнителей инструкций с побочным эффектом и, соответственно, структур данных, с которыми эти исполнители работают. Некоторые из них предназначены для выполнения произвольных внешних побочных эффектов, некоторые только для определенного ограниченного множества (что позволяет гранулировано контролировать их использование). Существуют также библиотеки, позволяющие абстрагироваться от конкретного исполнителя побочных эффектов, позволяя писать полиморфный код с указанием используемых эффектов.

- **Произвольные эффекты**

  Чаще всего, структуры данных, которые инкапсулируют в себе *вычисления с побочным эффектом*, называются `IO` или `Task` [5][6][7][8].

Соответственно, могут быть чистые функции, которые возвращают значения этих типов, например, функции строкового ввода и вывода:

```
def printLine(s: String): IO[Unit]
def readLine: IO[String]
```

В данном случае, функция `printLine` чистая и она лишь возвращает вычисление, которое распечатает строку на экран (в возвращаемом значении типа `IO[Unit]`). Аналогично, функция `readLine` возвращает не строку, введенную пользователем, а вычисление, которое может вернуть строку, введенную пользователем.

- **Гранулярные эффекты**

  Существуют способы описания вычислений, которые *полиморфны* по конкретному типу возвращаемого значения, но имеют гранулярный контроль за производимыми эффектами.

  В этом случае функция возвращает тип произвольного эффекта `F[_]`. При этом в сигнатуре функции указываются ограничения на возможные варианты эффектов, допустимые для осуществления этой функцией.

2.2 Композиция вычислений с эффектами

Промоделировав функции с побочным эффектом как чистые функции вида `A => IO[B]` или более гранулярно как `A => F[B]` для определенных `F`, мы получаем некие возможности обработки информации, полученной при помощи побочного эффекта, но их пока недостаточно.

Например, при помощи функций `readLine` и `printLine`, описанных выше, не получается распечатать строку, которая содержала бы информацию из считанной строки, потому что `IO[A]` – это лишь описание *вычисления с побочным эффектом*, возвращающего `A`, поэтому невозможно его получить вне `IO` в чистой функции.

Для решения проблем *композиции* функций вида `A => IO[B]` и `B => IO[C]` была применена техника, которая называется *монадами* [5] и которая имеет свои истоки в теории категорий [9][10].

Если какой-то тип данных `M[_]` является монадой, это означает, что, имея `M[A]`, мы можем применить к нему функцию `A => M[B]` и получим значение `M[B]`. Эта операция исторически имеет названия `bind`, `flatMap` или оператора >>=.

Для примера с `IO[A]`, описывающей вычисление с побочным эффектом, это означает, что, если мы применим функцию, которая обрабатывает значение типа `A`, полученное в результате побочного эффекта, к функции, которая его обработает и вернет `IO[B]`, мы сможем произвести их композицию с тем, чтобы получить `IO[B]`.

```
def f[A, B](a: IO[A], f: A => IO[B]): IO[B] = a >>= f
```


Другой важной чертой монад является то, что всегда есть возможность по чистому значению получить монадическое, то есть получить значение как бы с *нулевым эффектом*. Обычно такого рода функции называются `pure` или `point` и имеют вид `A => F[A]`.

Монадическая композиция по сути является последовательной композицией зависимых вычислений, где каждое вычисление может производить эффект. Для монад типа `IO` монадическая композиция является своего рода аналогом «точки с запятой» в императивных языках программирования, где точка с запятой часто разделяет отдельные операторы, каждый из которых может обладать побочным эффектом.

Как мы увидим далее, в функциональных языках монады позволяют более гибко управлять этой композицией. При этом такого рода композиция гарантирует упорядоченность возникновения побочных эффектов.

Однако не всегда вычисления с эффектами должны быть строго упорядоченными. Причиной могут быть как соображения *эффективности* (не всегда вычисления стоит делать последовательно), так и соображения *семантики* (не всегда нужна или определена именно последовательная композиция эффектов).

Для такого рода композиций была предложена абстракция *аппликативных функторов* [11]. Аппликативные функторы позволяют слить пару эффектов в эффект получения пары объектов:

```
def product[A, B](a: F[A], b: F[B]): F[(A, B)]
```

Так как аппликативный функтор является функтором, к значению вида `F[(A, B)]` можно применить функцию вида `(A, B) => C` с тем, чтобы получить значение типа `F[C]`.

Для эффектов, которые можно выполнять параллельно, реализация аппликативного функтора может позволить выполнять эти вычисления параллельно. В общем случае, абстракция аппликативного функтора позволяет осуществлять параллельную композицию вычислений с эффектами.

2.3 Эффекты в широком смысле

На понятие *эффекта* можно посмотреть шире: под эффектом можно понимать не только побочные эффекты в виде изменения состояния внешнего по отношению к программе мира или ее переменных.

В общем случае (в зависимости от потребностей точного моделирования семантики выполнения в чистом функциональном языке) под эффектом можно понимать даже обращение к значению переменной находящейся, быть может, на другом узле вычислительной сети при распределенном выполнении [12]. В более частных случаях под эффектом могут пониматься любые действия функции помимо вычисления и возврата простого единственного значения, примеры которых будут рассмотрены далее.

Важным свойством функций с побочным эффектом является то, что они могут делать еще что-то, кроме вычисления возвращаемого значения, а также пользоваться еще чем-либо, кроме своих аргументов для своей работы. При функциональном программировании вместо *функций с побочным эффектом* вида `A => B` рассматривают *чистые* функции вида `A => F[B]`, где `F` инкапсулирует сам эффект (например, произвольный побочный эффект, когда это тип `IO`). При более широком взгляде на понятие *эффекта* можно рассматривать функции вида `A => F[B]` как *чистые* аналоги *функций с эффектом* вида `A => B` для более широкого класса типов `F`, чем просто побочный эффект.

Рассмотрим некоторые примеры монад, которые моделируют те или иные *эффекты* в чистом функциональном программировании.

- **Монада произвольного побочного эффекта.** Упоминавшиеся ранее монады `IO` и `Task` и другие подобные позволяют описывать вычисления с произвольным побочным эффектом.
- **Тривиальная монада.** Монада `Id` описывает "нулевой" эффект. Функция обладающая таким эффектом (т.е., функция вида `A => Id[B]`) на самом деле является чистой (фактически, `A => B`). Последовательная композиция функций с тривиальным эффектом совпадает с простой композицией функций.

  На практике такую монаду используют в для полиморфного по эффектам кода, где в конкретном случае никакие эффекты не нужны или действие определенных эффектов моделируется чистым кодом (например, при тестировании).
- **Монада частичных вычислений.** В качестве эффекта можно рассматривать ситуацию, когда функция в некоторых случаях не может возвратить значение, то есть функция не является *тотальной* (является *частичной*). В таких случаях используют монаду `Option` (она же `Maybe`).

  Последовательная композиция функций с эффектом частичных вычислений обладает семантикой *раннего падения* (fail fast). Это означает, что стоит любой функции в последовательности функций с этим эффектом вернуть значение типа `None` (т.е., отсутствие результата), вычисление прекращается с тем же результатом.
- **Монада множественных вычислений.** Некоторые функции вычисляют несколько значений (в том числе и отсутствие значений). Это тоже можно рассматривать как эффект и композировать такие многозначные функции друг с другом. Композицией многозначных вычислений является аналог декартового произведения (при необходимости с учетом повторений). Обыкновенный список `List` часто является подходящей монадой для многозначных вычислений.
- **Монада вычислений с контекстом.** Монада `Reader` представляет собой вычисление, которое имеет доступ до дополнительной информации: контекст или конфигурация вычисления. Возможность такого доступа (явно не отраженного напрямую в сигнатуре функции) также является эффектом (в широком смысле).
- **Монада вычислений с метаинформацией.** В качестве эффекта рассматривают формирование дополнительной информации при вычислении основного значения. Эта информация является своего рода описанием (возможно пустым), приложенным к вычисленному значению. За это отвечает монада `Writer`.

  Для корректной работы этой монады, в частности, для обеспечения композируемости монадических вычислений, нужно обеспечить, чтобы соответствующие используемые описания, которые идут вместе с вычисленным значением, также композировались.

  Характерным примером использования этой монады является описание вычислений, которые параллельно с какими-то вычислениями, формируют лог событий. В этом случае, этот лог событий и будет являться описанием этих вычислений.
- **Вероятностная монада.** Еще одним примером эффекта можно рассмотреть возврат не единичного значения, а вероятностного распределения значений. При рассмотрении конечных дискретных распределений вероятности, это является обобщением эффекта множественных значений (где каждому значению приписана вероятность возникновения этого значения).

  Можно рассматривать и распределения, не являющиеся дискретными. В этом случае результирующее вероятностное распределение должно быть далее проинтерпретировано (например, при помощи генератора случайных чисел).

2.4 Наслаивание монад

2.4.1 Проблема

Очень часто возникает ситуация, когда либо одновременно используются две или более монад для одного и того же значения (то есть код производит несколько эффектов), либо имеется потребность в композировании функций, использующих различные монады (то есть функции, имеющие разные эффекты). Уже через небольшое время после использования монад для моделирования эффектов (в широком смысле) было показано, что в общем случае разные монады не композируются [13].

Также было показано, что в общем случае порядок наслоения монад друг на друга существенно влияет на результат, что оказалось ограничением для написания полиморфного кода.

Например, композиция множественных и частичных вычислений может продуцировать (в зависимости от порядка применения эффектов

множественности List и частичности Option) либо множественность частичных результатов (List из Option'ов), либо частичный множественный результат (Option List'ов).

2.4.2 Трансформация монад

В качестве попытки разрешения этих и связанных с этим проблем, был предложен механизм *трансформаций монад* (*monad transformers*) [14].

Однако, этот подход не позволял полноценно описывать код, который продуцирует отдельный эффект или несколько эффектов, при этом композируемый с другими эффектами. Код, использующий этот принцип, должен был всегда четко определять, в какой последовательности применяются используемые эффекты, и не допускал произвольной вставки других эффектов между используемыми эффектами (если только не был написан в специальной и очень громоздкой манере).

Таким образом, раньше времени принималось решение о последовательности применяемых эффектов. Из-за этого невозможно было описать единственную функцию, которая могла бы композироваться с другими функциями тех же эффектов в другом порядке. Также возникали проблемы с возникновением несколько раз одного и того же эффекта в этой последовательности эффектов.

2.4.3 Расширяемые эффекты

Позже был предложен подход, описывающий специальную монаду, названную `Eff`, которая позволяла описывать ленивое вычисление с неупорядоченным множеством эффектов [15][16]. Это неупорядоченное множество эффектов (по историческим причинам иногда называемое *стеком эффектов*) является параметром типа `Eff` и тем самым присутствует в сигнатуре всех функций, возвращающих значение типа `Eff`.

При этом были сформулированы правила монадической композиции таких вычислений. Результирующее вычисление имеет объединение множеств эффектов композируемых вычислений. Легкость объединения (и дальнейшей композиции) таких вычислений обеспечивается путем использования *ограниченного полиморфизма* (*bounded polymorphism*) по этому множеству. В таком случае объединение множеств сводится к объединению ограничений на полиморфный параметр вычислений, что естественным образом поддерживается в языках, поддерживающих ограниченный полиморфизм.

Соответственно, для того, чтобы выполнить вычисление со всеми эффектами, требуется применить к значению типа `Eff` (имеющего множество эффектов как параметр типа) специальные интерпретаторы эффектов (в требуемом порядке), где каждый из интерпретаторов будет либо убирать из множества эффектов как минимум один эффект, либо преобразовывать один эффект в другие. Это должно продолжаться до тех пор, пока множество эффектов не станет пустым. Полученное значение будет чистым.

Этот подход обеспечивает гибкие возможности по реализации полиморфного по эффектам кода, который не требует раннего упорядочивания эффектов и не имеет проблем с повторением одного и того же эффекта в наслаивании несколько раз (из-за рассмотрения множества эффектов взамен последовательности). Ограничением подхода является то, что все функции, имеющие какой-либо эффект, необходимо оформлять в виде функций, возвращающих объект специального типа `Eff` (с определенными параметрами типа), а также необходимость использования примитивов, специфичных для типа данных `Eff`.

Рассмотрим пример вычислений, которые оформлены для использования с монадой `Eff`.

```
def multipleIntegers[R: _list]: Eff[R, Int] = ???
def sqrt[R: _option](x: Int): Eff[R, Double] = ???

def composition[R: _list:_option]: Eff[R, Double] =
for {
i <- multipleIntegers
x <- sqrt(i)
} yield x
```

В этом примере мы имеем два вычисления (`multipleIntegers` и `sqrt`), которые полиморфны по эффекту с использованием монады `Eff` (параметр типа `R`), но при этом первая функция требует, чтобы `R` поддерживал эффект множественности, а вторая – эффект частичных вычислений. Соответственно, функция композиции требует оба этих эффекта.

Далее, мы можем проинтерпретировать функцию композиции как минимум в двух упорядочиваниях эффектов: сначала эффект множественности, затем эффект частичных вычислений или наоборот.

```
type LO = Fx.fx2[List, Option]
// множество из двух эффектов

composition[LO].runList.runOption.run
// результат типа Option[List[Double]]

composition[LO].runOption.runList.run
// результат типа List[Option[Double]]
```

Мы можем использовать эту функцию и в более обширном контексте, например, с эффектом ввода-вывода. Таким образом, мы можем использовать одни и те же функции в произвольных контекстах, которые предоставляют обработку требуемых конкретному вычислению эффектов.

```
type Ext = Fx.fx3[List, Option, IO]

composition[Ext].runList.runOption
// результат типа
//   Eff[Fx.fx1[IO], Option[List[Double]]]
```

2.4.4 Полиморфизм по эффекту

Своего рода обобщением предыдущих двух решений является вместо использования определенной структуры данных и полиморфизма по ее параметру использование *полиморфизма высокого порядка* (*higher-kinded polymorphism*) по самому типу эффекта вкупе с *ограниченным полиморфизмом* в каждой отдельной функции. По историческим причинам (и из-за истоков в теории категорий) этот подход получил название *tagless-final* стиль [17][18][19]. В одном частном, но широко распространенном случае подход называется «`F` с дыркой», что подчеркивает использование полиморфизма высокого порядка по эффекту.

В этом подходе функции, обладающие тем или иным эффектов и предназначаемые для композируемости, объявляются полиморфными по типу эффекта (то есть по типу высшего порядка, обычно обозначаемого `F`), возвращающими значение `F` с конкретным типом. При этом для описания того, какие именно эффекты могут быть произведены этой функцией, используется *ограниченный полиморфизм*, то есть полиморфизм, который требует конструктивных доказательств того, что `F`, по которому осуществляется полиморфизм, обладает достаточными возможностями обеспечивать заданный эффект.

Этот подход имеет лишь то принципиальное отличие от подхода с монадой `Eff`, что в общем случае для типа `F` не обязательно требуется, чтобы он являлся монадой. Если функции с эффектом `F` не требуется именно монадическая сущность эффекта, она не обязана требовать это. С другой стороны, если функции требуется, чтобы эффект `F` был монадой (например, сама функция является монадической композицией других функций), она должна потребовать монадичность `F` явно (в отличие от монады `Eff`, которая является монадой сама по себе). Это может иметь свои преимущества и позволяет функции четче декларировать, что именно ей требуется (ограниченный полиморфизм эксплуатируется не только для типов эффекта, но и для способа композиции).

2.4.5 Другие альтернативы

Стоит упомянуть язык Unison, функциональный язык в разработке, который использует наслоение функторов и монад на уровне языка, а не на уровне библиотек (как это делают Haskell и Scala) [20]. Фактически, в языке переопределена композиция функций, которая может быть реализована как композиция аппликативных функторов или монад (в зависимости от типа выражения). Из-за этого улучшается читаемость кода и, возможно, скорость компиляции. Однако, по всей видимости, принципиальные возможности остаются на том же уровне. Поэтому в дальнейшем мы будем рассматривать реализацию наслоения монад при помощи библиотек к широко используемым языкам (в частности, Scala).

Альтернативным вариантом решения проблемы композируемости монад является использование *вместо* монад для композиции вычислений с эффектами абстракции, являющейся чуть менее мощной, чем монада, но при этом все еще применимой и при этом более легко композируемой. Эта абстракция получила название *стрелки* (*arrow*) [21][22]; она является более выразительно мощной, чем аппликативный функтор, но менее мощной, чем монада [23]. Чисто функциональное программирование с эффектами не в монадическом, а в стрелочном стиле требует несколько необычного (и порой непривычного) способа выражения программы, однако позволяет комбинировать различные эффекты легко (для тех эффектов, к которым применимо понятие стрелки) [24][25][26].

3. Монадический подход к моделированию

3.1 Базовая идея

Начнем с представления модельного времени при помощи использования монад. Основной идеей “монадического” подхода к работе с вычислениями в модельном времени состоит в рассмотрении использования модельного времени как эффекта (в широком смысле).

В простейшем случае мы можем представлять функцию, моделирующую вычисление, которое требует определенное модельное время, как функцию, возвращающую вычисленное значение вместе с величиной требуемого времени.

Для *композируемости* такого рода вычислений, представим эффект вычисления вместе с тратой модельного времени как монаду; будем называть ее `TimedValue`.

Например, рассмотрим простейшую функцию сложения двух чисел.

```
def sumTimed(x: Int, y: Int): TimedValue[Int] =
  (x + y) |: 2.microseconds
```

Данная функция описывает вычисление, состоящее из сложения двух чисел, которое (исходя из определения этой функции) занимает две микросекунды модельного времени на вычисление. Здесь оператор `|:` – синтаксически удобный способ создания значений монады `TimedValue`.

Эта монада похожа на монаду `Writer`, но она не имеет типового параметра накапливаемого значения (тут он зафиксирован, это тип `Time`). Как следствие, эта монада не требует дополнительных ограничений на способ композиции для времени. Для последовательной композиции вычислений времена вычислений складываются.

Таким образом, имея, например, две функции

```
val inc: Int => TimedValue[Long]
val sqrt: Long => TimedValue[Double]
```

мы можем скомпозировать их в одну функцию, занимающую сумму модельного времени обеих функций:

```
inc >=> sqrt: Int => TimedValue[Double]
```

3.2 Последовательная и параллельная композиция

Функция сложения выбрана именно для *последовательной композиции* вычислений в модельном времени, подходящая для использования в монаде, то есть для описания зависимых вычислений.

В общем случае, может использоваться произвольная *ассоциативная бинарная* функция композиции двух времен. Выбор этой функции в большой степени определяет семантику комбинации монадических значений `TimedValue`.

Например, эта функция могла бы добавлять определенную величину времени для моделирования задержек на собственно вызов функций (при необходимости и очень точном моделировании затрачиваемого времени). Однако, в данной работе мы не будем рассматривать такой случай.

В качестве другого (весьма важного) примера альтернативной функции можно выбрать функцию максимума. При таком выборе, получится параллельная (с точки зрения модельного времени) комбинация вычислений `TimedValue`. Однако, с такой композицией может возникнуть проблема *физического смысла* результата. В частности, не всегда доступна параллельная композиция вычислений, занимающих (модельное) время, из-за нехватки (модельных) ресурсов даже в случае независимости этих вычислений по данным.

Для гранулированного подхода используются различные абстракции (например, `Parallel` [27]), которые позволяют не давать композировать функции, тратящие модельное время параллельно в случае нехватки модельных ресурсов для этого. Это достигается за счет сложного параметрического полиморфизма, фактически эквивалентного по выразительной мощности системе Хорновских дизъюнктов [28].

3.3 Монады времени в стеке монад

Как и другие монады, монаду вычислений с модельным временем можно использовать в стеке с другими монадами.

Это позволяет использовать один и тот же монадический код в различных ситуациях без переписывания и без изменения семантики с сохранением типобезопасности. При этом, различная семантика итоговой работы функций достигается за счет различных комбинаций функций-интерпретаторов при использовании этого монадического кода.

Правда, такой подход накладывает определенные требования на оформление монадического кода. Так, если до этого функции `A => B`, которые моделируют вычисление, затрачивающее время, мы представляли как `A => TimedValue[B]`, для того, чтобы использовать такие функции в наслоении с

другими монадами и эффектами, нам придется представлять из как полиморфные функции по типу эффекта с ограничениями на этот тип эффекта.

Например, функция вычисления квадратного корня вместо того, чтобы быть объявленной как

```
def sqrt(x: Int): TimedValue[Double] =
  math.sqrt(x) |: 1.millisecond
```

должна быть объявлена *полиморфной по типу эффекта*. В стиле *F с дыркой* это будет выглядеть как

```
def sqrt[F[_]: TimedValueAlgebra](x: Int): F[Double] =
  math.sqrt(x) ||: 1.millisecond
```

В стиле монады `Eff`, эта же функция выглядела бы как

```
def sqrt[R: _timedValue](x: Int): Eff[R, Double] =
  math.sqrt(x) ||: 1.millisecond
```

или, используя «стрелочный» синтаксис вместо указания типа `Eff`,

```
def sqrt[R: _timedValue](x: Int): R ||> Double =
  math.sqrt(x) ||: 1.millisecond
```

Принципиально, в рамках данной работы, оба способа полиморфного задания эквивалентны. Поэтому, для простоты, в этом разделе будем рассматривать только второй способ изложения (при помощи монады `Eff`).

В данном примере типовой параметр `R` отвечает за произвольный эффект, который ограничен тем, что в нем обязан присутствовать эффект `_timedValue`. При этом, `_timedValue` технически объявлен как `TimedValue |= ?`, то есть означает, что эффект, моделируемый монадой `TimedValue` присутствует в стеке монад. Синтаксис оператора `||:` позволяет удобно создавать значения соответствующего типа (то есть, `Eff` с произвольным эффектом `R`, который, в свою очередь, ограничен наличием `TimedValue` в стеке монад).

Перечислим виды значений, с которыми приходится работать в задачах моделирования.

- **Единичное значение**

  Положим, у нас есть простое целочисленное значение, запакованное в произвольный стек монад (то есть, нет никаких ограничений на R):

  ```
  def simpleVal[R]: R ||> Int
  ```

  Мы можем применить функцию вычисления квадратного корня к этому значению:

  ```
  simpleVal >>= sqrt
  ```

  Результатом будет значение типа `R ||> Double`, где на `R` будет наложено ограничение наличия `TimedValue` в стеке монад. Мы можем запустить вычисление этого выражения и получим значение типа `Double` на выходе:

  ```
  type S1 = Fx.fx1[TimedValue]
  (simpleVal[S1] >>= sqrt[S1]).runTimed.run
  // результат типа TimedValue[Double]
  ```

- **Множественное значение**

  Представим себе, что на входе есть множественное значение, для которого мы хотим выполнить наше timed-вычисление. Такое множественное значение может быть промоделировано следующим образом:

```
def severalVals[R: _list]: R ||> Int
```

  Разница с рассмотренным выше `simpleVal` состоит в том, что эффект `R` содержит требование множественности `_list` (которое раскрывается в `List |= ?`, наподобие тому, как `_timedValue` раскрывается в `TimedValue |= ?`).

  Мы можем таким же образом скомпозировать это множественное значение с функцией вычисления квадратного корня (напомним, что эта функция тратит модельное время):

```
severalVals >>= sqrt
```

  Мы так же, как и в предыдущий раз, можем запустить вычисление этого выражения. Однако, до момента запуска результирующее значение имеет неоднозначность порядка запуска. Компилятор не знает хотим ли мы получить результат вычисления списка, занимающий модельное время, или же список результатов, каждый занимающий свое модельное время на вычисление. Это определяется типом результата: в первом случае от `TimedValue[List[Double]]`, во втором он `List[TimedValue[Double]]`. Мощность подхода с использованием стека монад состоит в том, что мы можем получить оба варианта результатов, и при этом отдельные участки кода (в данном случае, функции `severalVals` и `sqrt`) будут одинаковыми в обоих случаях, то есть собственно описание алгоритма вычислений полностью переиспользуется:

```
type S2 = Fx.fx2[TimedValue, List]

(severalVals[S2]>>= sqrt[S2]).runList.runTimed.run
// результат типа TimedValue[List[Double]]

(severalVals[S2] >>= sqrt[S2]).runTimed.runList.run
// результат типа List[TimedValue[Double]]
```

  Разница есть только в порядке вызова интерпретаторов runList и runTimed.

- **Вероятностное значение**

  Аналогичным образом мы можем переиспользовать функцию sqrt для работы с вероятностным значением. Имея variadicVal, возвращающее конечное дискретное распределение целых чисел, мы можем запустить следующий код и получить дискретное распределение чисел Double, занимающих модельное время:

```
type S3 =
  Fx.fx2[TimedValue, DiscreteFiniteDistribution]
```



```
  (variadicVal[S3] >>= sqrt[S3]).runDFD.runTimed.run
  // результат типа
  //   DiscreteFiniteDistribution[TimedValue[Double]]
```

- **Более сложные смешения**

  В общем случае мы можем использовать стек произвольной глубины. В качестве примера, мы можем рассмотреть ситуацию, когда timed-вычисление зависит от источников разных монадических функций. Для простоты, мы будем использовать описанные выше функции `severalVals` и `variadicVal`, а в timed-функцию `sqrt` будем передавать сумму от тех двух функций.

```
def f[R: _timedValue:_list:_dFD]: R ||> Double = for {
  v1 <- severalVals
  v2 <- variadicVal
  s <- sqrt(v1 + v2)
} yield s
```

  В общем случае существует шесть вариантов выполнения этой функции и получения конкретного значения. Все эти варианты будут давать результат различных типов (и, как следствие, вычисление будет иметь различную семантику). При этом для получения этих значений будет использоваться один и тот же код функции `f`.

  Рассмотрим несколько примеров таких вычислений. В первом случае мы будем получать список вероятностных распределений timed-величин типа `Double`. Во втором случае мы будем получать вероятностное распределение timed-величин типа `List[Double]`.

```
type S4 =
  Fx.fx3[TimedValue, List, DiscreteFiniteDistribution]

f[S4].runTimed.runDFD.runList.run
// результат типа List[
//   DiscreteFiniteDistribution[TimedValue[Double]]]

f[S4].runList.runTimed.runDFD.run
// результат типа DiscreteFiniteDistribution[
//   TimedValue[List[Double]]]
```

Таким образом, мы можем использовать вычисления, описывающие вычисления, занимающие модельное время, вместе с другого рода эффектами. При этом легко осуществляется разделение описания действий и выполнения этих действий. Вместе с этим, способ выполнения определяет конечную семантику всей операции, которая задается упорядочиванием эффектов из стека монад. Это позволяет переиспользовать один и тот же код поведений в различных ситуациях и дает возможность не принимать определенных проектных решений (в частности, по упорядочиванию эффектов) раньше времени.

3.4 Обобщения затрачиваемого времени

Значение типа `TimedValue` содержит единственное значение времени, обозначающее длительность – модельное время, затрачиваемое для вычисления хранимого значения. В общем же случае может потребоваться иметь не простое значение длительности, а более сложные варианты. В частности, рассмотрим следующие примеры таких потребностей.

- **Косвенное время**

  При описании вычисления не всегда известно собственно затрачиваемое время на выполнение. Время может зависеть от используемого процессора, затрат на коммуникацию, работы кэша и пр. При детальном моделировании времени, это может существенно сказаться на результате вычислений.

  В частности, вместо указания значений затрачиваемого модельного времени, может быть потребность в указании затрачиваемых модельных тиков процессора или в указании числа различных инструкций определенного вида, количества обращений к памяти и подобном.

  То есть, в конечном итоге, время указывается *косвенно*, в терминах процессора или аппаратуры.

  Можно вводить специализированные аналоги монады `TimedValue` для такого рода косвенного задания времени. Однако могут возникнуть проблемы, связанные с возможной потребностью в разнообразии такого рода монад.

  Также возникают проблемы с композицией этих монад с монадой `TimedValue`, так как для корректной композиции они должны знать конкретные параметры используемых процессоров. Это может приводить к необходимости изменения кода, описывающего вычисления при внешних по отношению к нему изменениях (смене типа процессора).

- **Не единственное значение времени**

  Не всегда может быть известно точное (пусть даже и модельное) время выполнения тех или иных функций. Единственное значение времени, использующееся в монаде `TimedValue` можно, конечно, интерпретировать как ограничение сверху. Правда, в таком случае мы лишаемся возможности оценивать затрачиваемое модельное время снизу, что может быть нужным иногда [29].

  Проблемы с нижней границей можно пробовать решать введением интервала времен вместо отдельного времени в `TimedValue`. Соответственным образом нужно будет изменить правила композиции монады (со сложением интервалов вместо сложения отдельных величин и соответствующей композицией интервалов при параллельной композиции).

  Но интервал может стать лишь грубым приближением действительно необходимого значения времени. Аналогично различным входным параметрам вычислений при построении стека монадических вычислений, может потребоваться выражать, например, конкретный список отдельных времен (а не непрерывный интервал) или даже вероятностное распределение затрачиваемых времен.

Если обобщить эти мысли, получится, что монада `TimedValue` должна содержать не *отдельное единичное значение времени*, а *обобщенное значение, содержащее время*.

```
case class TimedValue[A, F[_]](a: A, time: F[Time])
```

При этом, например, для композиции может требоваться, чтобы данный `F[_]` был, например, функтором. Для параллельной композиции, соответственно, может требоваться, чтобы `F[_]` был бы аппликативным функтором. Аналогично, для последовательной композиции может быть аналогичное требование для монадичности этого параметра типа. Могут существовать и другие дополнительные ограничения на `F[_]`.

Подобные ограничения на `F[_]` могут быть отнесены на как можно более поздний момент по коду (то есть, в функциях, которые действительно используют эти свойства (функторности, аппликативности, монадичности или другие)).

Можно рассматривать также обобщение (с одной стороны усиливающее, с другой ограничивающее), когда хранимое время – это не обернутое значение, а произвольный стек монад, в конечном счете продуцирующий значение времени (в соответствии с семантикой стека).

```
case class TimedValue[A, R](a: A, time: R ||> Time)
```

Более того, это обобщение позволяет естественным образом описывать случай, когда время задано косвенно (например, через затрачиваемые тики процессора и пр.). В таком случае эти данные должны позволять получить конечное время через промежуточные монады, которые в конечном итоге показываются в стеке монад.

```
for {
  a <- timedVal ||: (1 to 5).milliseconds
  b <- tickedVal(a) ||: (100 to 200).processorTicks
  c <- instVal ||: (500.arithOps + 20.controlOps)
} yield b + c
```

Стоит отметить, что даже при использовании непрямого представления времени стеки монад позволяют задавать интервалы, множественности или даже вероятностные распределения затрачиваемого времени через тики процессора и другие способы. При этом при смене типа процессора и других характеристик (и, как следствие, возможного изменения затрачиваемого модельного времени), код, описывающий вычисления, не будет меняться.

3.5 Контроль времени на уровне типов

Рассмотрим другой вариант изменения базовой идеи. Мощная система типов позволяет более точно контролировать затрачиваемое модельное время и соответствие имеющегося и требуемого модельного времени.

В частности, есть возможность указать, что определенный исполнитель timed-функций может принять как аргумент только функции, затрачивающие не более чем заданный объем времени. Это может быть полезно для статической проверки в случае использования со строго-периодическими операционными системами, которые находят широчайшее применение в ответственных системах реального времени.

Само затрачиваемое модельное время можно присоединить к выражению при помощи литеральных типов. Так, например, представим себе функцию, возвращающую значение, которое будет иметь тип, соответствующий результату, вычисляемому за 10 миллисекунд модельного времени. Если мы попытаемся передать это выражение в функцию, которая может выполнить выражения, занимающее не более, например, восьми миллисекунд модельного времени, мы получим ошибку времени компиляции.

Для того, чтобы это сделать, нужно, чтобы тип, соответствующий затрачиваемому модельному времени, был приписан к timed-выражению. Это можно сделать как минимум двумя способами.

- **Типовой параметр**

  Одним из способов является добавление типово́го параметра в монадический тип, который описывает выражение, вычисляемое за модельное время, то есть в тип `TimedValue`.

  Таким образом, этот тип начинает иметь сигнатуру не `TimedValue[A]`, а `TimedValue[A, TimeBound]`. Соответственно, все операции над этим типом должны в своих сигнатурах учитывать этот типовой параметр.

  Это, на самом деле, представляет собой некую проблему для некоторых методов, потому что они перестают отвечать своим абстракциям. Например, таким является функция `flatMap`, определяющая монадическую композицию. Это ограничивает применимость такого метода.

- **Тип-поле**

  Другой способ состоит в том, чтобы привязать typelevel-значение к выражению со временем – это *зависимые типы*. Имеется несколько ограниченный вариант, который называется *path-dependent type* [30]. Подход представляет собой добавление поля в структуру данных `TimedValue`, которое является *типом*, а не значением.

Реализовывать само ограничение по времени можно тоже двумя способами. В обоих случаях типовым параметром является *конструктивное доказательство*

(или *свидетельство*, *evidence*) того, что содержимое значения времени является ограниченным (например, не превышает заданную величину).

4. Заключение

Монадический подход к моделированию времени фактически является применением хорошо известных техник функционального программирования к области архитектурного моделирования, в частности, к области моделирования поведения сложных компьютерных систем.

В частности, строгая типизированность и широкие возможности мощной системы типов позволяют выражать многие аспекты модели системы напрямую в коде, которые будут проверены на стадии компиляции кода, сгенерированного на основе модели системы. Это позволяет обеспечивать ранние проверки композируемости и соответствия отдельных поведений при изменениях модели и отдельных спецификаций поведения компонентов модели. Эти проверки производятся статически, до запуска моделирования всей системы.

Появляются также широкие возможности по композиции разнородных поведений или их аспектов за счет использования ограниченного параметрического полиморфизма функций. За счет использования полиморфизма высокого порядка по эффектам появляется возможность наслаивания дополнительных эффектов к имеющемуся коду без его переписывания (и даже без перекомпиляции). Это позволяет расширять и уточнять семантику отдельных поведений и их композиций без потери корректности.

Помимо прочего, данный подход обладает другими преимуществами функционального программирования, в том числе, возможностью использования повышенного уровня абстракции в коде (как следствие, увеличения скорости разработки и простоты понимания), широкими возможностями автоматической оптимизации кода во время компиляции и автоматического распараллеливания во время выполнения, а также многими другими возможностями. Конечно, при этом стоит учитывать непривычность такого рода подхода для многих профессионалов в прикладной области, что необходимо иметь в виду в будущем и что не относится напрямую к идеям, изложенным в этой работе.

В рамках проекта РФФИ 17-01-00504 в 2019 году планируется разработать экспериментальный комплекс инструментальных средств для моделирования систем управления. Комплекс будет строиться на платформе Scala, имеющей все необходимые для этого средства функционального программирования.

Совместное использование механизмов функционального программирования дает возможность описывать вычисления с минимальными требованиями на инфраструктуру такие, что они гарантированно корректно композируются с другими вычислениями в произвольных подходящих инфраструктурах (при

этом их невозможно использовать в неподходящих инфраструктурах, потому что проверки осуществляются на уровне компилятора). Это позволяет переиспользовать (даже без перекомпиляции) имеющийся код не только для решения конкретной задачи, для которой этот код был разработан, но и для решения задач расширяющего класса.

Как следствие, наращивание набора инструментальных средств и видов анализа в перспективной среде моделирования будет требовать лишь добавления или изменения кода, ответственного за конкретные частные аспекты (в частности, специфичные для задачи виды входных данных и анализ специфичных результатов) и не будет затрагивать имеющийся (полиморфный) код. При этом использование мощных механизмов функционального программирования требует определенной культуры и следования определенному стилю программирования, что в контексте разработки и аттестации систем ответственного назначения является уместной платой за получение преимуществ в плане повышения доли повторно используемых компонентов (re-use), композируемости и достоверной консистентности системы.

Список литературы

[1] Denis Buzdalov, Alexey Khoroshilov. A Discrete-Event Simulator for Early Validation of Avionics Systems. In Proc. of the First International Workshop on Architecture Centric Virtual Integration co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), 2014, CEUR Workshop Proceedings, vol. 1233,

[2] Denis Buzdalov. Simulation of AADL models with software-in-the-loop execution. ACM SIGAda Ada Letters, vol. 36, issue 2, December 2016, pp. 49-53

[3] John Hughes. Why functional programming matters. PMG-40, Chalmers University of Technology, Goteborg, Sweden, 1984. Режим доступа: http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf, дата обращения 18.12.2018

[4] Simon Peyton Jones. Functional programming languages as a software engineering tool. In Embedded Systems. Lecture Notes in Computer Science, vol 284, 1986, pp 153-173

[5] Simon Peyton Jones, Philip Wadler. Imperative functional programming. In Proc. of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages, 1993, pp. 71-84.

[6] The IO Monad for Scala. Режим доступа: https://typelevel.org/cats-effect/, дата обращения 18.12.2018

[7] Monix library, Task monad. Режим доступа: https://monix.io/docs/3x/eval/task.html https://typelevel.org/cats-effect/, дата обращения 18.12.2018

[8] ZIO, Scala-library for asynchronous and concurrent programming, IO monad. Режим доступа: https://scalaz.github.io/scalaz-zio/datatypes/io.html, дата обращения 18.12.2018

[9] Eugenio Moggi. Notions of computation and monads. Information and Computation, vol. 93, issue 1, July 1991, pp. 55-92

[10] Philip Wadler. Comprehending Monads. Mathematical Structures in Computer Science, vol 2, issue 4, Cambridge University Press, 1992, pp. 461–493

[11] Conor McBride, Ross Paterson. Applicative programming with effects. Journal of Functional Programming, vol. 18, issue 1, 2008, pp. 1-13

[12] Oleg Kiselyov. Having an Effect. Presentation at the Seminar at the Institute of Information Science, Academia Sinica, Taipei, Taiwan, December 2, 2016. Режим доступа: http://okmij.org/ftp/Computation/having-effect.html, дата обращения 18.12.2018

[13] Guy L. Steele Jr. Building interpreters by composing monads. In Proc. of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL ’94), 1994, pp. 472–492

[14] Sheng Liang, Paul Hudak, Mark Jonest. Monad Transformers and Modular Interpreters. In Proc. of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL ’95), 1995, pp. 333-343

[15] Oleg Kiselyov, Amr Sabry, Cameron Swords. Extensible Effects: An Alternative to Monad Transformers. In Proc. of the 2013 ACM SIGPLAN Symposium on Haskell, 2013, pp. 59-70

[16] Oleg Kiselyov, Hiromi Ishii. Freer Monads, More Extensible Effects. In Proc. of the 2015 ACM SIGPLAN symposium on Haskell, 2015, pp. 94-105

[17] Oleg Kiselyov. Typed Tagless Final Interpreters. Lecture Notes in Computer Science, vol 7470, 2012, pp. 130-174

[18] Noel Welsh. Uniting Church and State: FP and OO Together. Scala Days conference, Copenhagen, 2017. Режим доступа: https://www.youtube.com/watch?v=IO5MD62dQbI, дата обращения 18.12.2018

[19] Олег Нижников. Современное ФП с Tagless Final. Конференция Joker 2018, Санкт-Петербург. Режим доступа: https://www.youtube.com/watch?v=sWEtnq0ReZA, дата обращения 18.12.2018

[20] Rúnar Bjarnason. Introduction to the Unison programming language. Lambda World 2018 conference, Living Computer Museum, Seattle, 18 september 2018. Режим доступа: https://www.youtube.com/watch?v=rp_Eild1aq8, дата обращения 18.12.2018

[21] John Hughes.Generalising monads to arrows. Science of Computer Programming, vol. 37, issues 1–3, May 2000, pp. 67–111

[22] John Hughes. Programming with Arrows. In Advanced Functional Programming, 2004, pp. 73-129

[23] Sam Lindley, Philip Wadler, Jeremy Yallop. Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous. Electronic Notes in Theoretical Computer Science, vol. 229, issue 5, 2011, pp. 97–117

[24] Yuriy Polyulya, Functional programming with arrows. Scala Days conference, 2015, Amsterdam. Режим доступа: https://www.youtube.com/watch?v=ZfAgvAIoUEY, дата обращения 18.12.2018

[25] Julien Richard Foy, Do it with (free?) arrows! Typelevel Summit Copenhagen, June 2017. Режим доступа: https://www.youtube.com/watch?v=PWBTOhMemxQ, дата обращения 18.12.2018

[26] John A De Goes. Blazing Fast, Pure Effects without Monads. LambdaConf conference, Boulder, CO, USA, June 2018. Режим доступа: https://www.youtube.com/watch?v=L8AEj6IRNEE, дата обращения 18.12.2018

[27] Parallel typeclass, cats library. Режим доступа: https://typelevel.org/cats/typeclasses/parallel.html, дата обращения 18.12.2018

[28] George Leontiev, There’s a prolog in your scala. ScalaIO conference, Paris, France, October 2014. Режим доступа: https://www.youtube.com/watch?v=iYCR2wzfdUs, дата обращения 18.12.2018

[29] A.M. Troitskiy, D.V. Buzdalov. A static approach to estimation of execution time of components in AADL models. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 157-172, doi: 10.15514/ISPRAS-2016-28(2)-10

[30] Nada Amin, Tiark Rompf, Martin Odersky. Foundations of Path-Dependent Types. In Proc. of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, 2014, pp. 233-249

On representation of simulation time in functional programming style

[1] D.V. Buzdalov <buzdalov@ispras.ru>
[1, 3, 4] A.K. Petrenko <petrenko@ispras.ru>
[1, 2, 3, 4] A.V. Khoroshilov <khoroshilov@ispras.ru>

[1] V.P.Ivannikov Institute for system programming, Russian Academy of Sciences, 109004, Russia, Moscow, Solzhenitsyn, d. 25
[2] Moscow Institute of physics and technology (State University), 141700, Moscow region, Dolgoprudny, Institutskiy per., 9
[3] Moscow State University named after M. V. Lomonosov, Moscow, 119991, GSP-1, Lenin hills, d. 1
[4] National research University "Higher school of economics", 101000, Russia, Moscow, Myasnitskaya, d. 20

Abstract. Functional programming plays the big role in the modern computer science and its importance is growing. This is not accidential: this approach helps to create better and more reliable software that is easy to reason about (both manually and automatically). However, these techniques are hardly used in the field of tools helping designing and modeling mission-critical systems. In this paper, we are trying to apply some nice techniques of functional programming to create a modeling system, in particular a simulation system for analysis of temporal behavioural properties of mission-critical systems. As a first step, we designed a representation of simulation time in terms of abstractions used in functional programming and tried to study its compositionability properties.

Keywords: architecture modeling, mission-critical systems, behavioural modeling, simulation time, functional programming, monads.

DOI: 10.15514/ISPRAS-2018-30(6)-20

For citation: Buzdalov D.V., Petrenko A.K., Khoroshilov A.V. On representation of simulation time for functional programming. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 6, 2018, pp. 341-366 (in Russian). DOI: 10.15514/ISPRAS-2018-30(6)-20

References

Engineering Languages and Systems (MoDELS 2014), 2014, CEUR Workshop Proceedings, vol. 1233,

[2] Denis Buzdalov. Simulation of AADL models with software-in-the-loop execution. ACM SIGAda Ada Letters, vol. 36, issue 2, December 2016, pp. 49-53

[3] John Hughes. Why functional programming matters. PMG-40, Chalmers University of Technology, Goteborg, Sweden, 1984. Available at: http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf, accessed 18.12.2018

[4] Simon Peyton Jones. Functional programming languages as a software engineering tool. In Embedded Systems. Lecture Notes in Computer Science, vol 284, 1986, pp 153-173

[5] Simon Peyton Jones, Philip Wadler. Imperative functional programming. In Proc. of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages, 1993, pp. 71-84.

[6] The IO Monad for Scala. Available at: https://typelevel.org/cats-effect/, accessed 18.12.2018

[7] Monix library, Task monad. Available at: https://monix.io/docs/3x/eval/task.html https://typelevel.org/cats-effect/, accessed 18.12.2018

[8] ZIO, Scala-library for asynchronous and concurrent programming, IO monad. Available at: https://scalaz.github.io/scalaz-zio/datatypes/io.html, accessed 18.12.2018

[9] Eugenio Moggi. Notions of computation and monads. Information and Computation, vol. 93, issue 1, July 1991, pp. 55-92

[10] Philip Wadler. Comprehending Monads. Mathematical Structures in Computer Science, vol 2, issue 4, Cambridge University Press, 1992, pp. 461–493

[11] Conor McBride, Ross Paterson. Applicative programming with effects. Journal of Functional Programming, vol. 18, issue 1, 2008, pp. 1-13

[12] Oleg Kiselyov. Having an Effect. Presentation at the Seminar at the Institute of Information Science, Academia Sinica, Taipei, Taiwan, December 2, 2016. Available at: http://okmij.org/ftp/Computation/having-effect.html, accessed 18.12.2018

[13] Guy L. Steele Jr. Building interpreters by composing monads. In Proc. of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '94), 1994, pp. 472–492

[14] Sheng Liang, Paul Hudak, Mark Jonest. Monad Transformers and Modular Interpreters. In Proc. of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '95), 1995, pp. 333-343

[15] Oleg Kiselyov, Amr Sabry, Cameron Swords. Extensible Effects: An Alternative to Monad Transformers. In Proc. of the 2013 ACM SIGPLAN Symposium on Haskell, 2013, pp. 59-70

[16] Oleg Kiselyov, Hiromi Ishii. Freer Monads, More Extensible Effects. In Proc. of the 2015 ACM SIGPLAN symposium on Haskell, 2015, pp. 94-105

[17] Oleg Kiselyov. Typed Tagless Final Interpreters. Lecture Notes in Computer Science, vol 7470, 2012, pp. 130-174

[18] Noel Welsh. Uniting Church and State: FP and OO Together. Scala Days conference, Copenhagen, 2017. Available at: https://www.youtube.com/watch?v=IO5MD62dQbI, accessed 18.12.2018

[19] Oleg Nizhnikov. Modern functional programming with Tagless Final. Joker 2018 Conference, Saint-Petersburg (in Russian). Available at: https://www.youtube.com/watch?v=sWEtnq0ReZA, accessed 18.12.2018

[20] Rúnar Bjarnason. Introduction to the Unison programming language. Lambda World 2018 conference, Living Computer Museum, Seattle, 18 september 2018. Available at: https://www.youtube.com/watch?v=rp_Eild1aq8, accessed 18.12.2018

[21] John Hughes.Generalising monads to arrows. Science of Computer Programming, vol. 37, issues 1–3, May 2000, pp. 67–111

[22] John Hughes. Programming with Arrows. In Advanced Functional Programming, 2004, pp. 73-129

[23] Sam Lindley, Philip Wadler, Jeremy Yallop. Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous. Electronic Notes in Theoretical Computer Science, vol. 229, issue 5, 2011, pp. 97–117

[24] Yuriy Polyulya, Functional programming with arrows. Scala Days conference, 2015, Amsterdam. Available at: https://www.youtube.com/watch?v=ZfAgvAIoUEY, accessed18.12.2018

[25] Julien Richard Foy, Do it with (free?) arrows! Typelevel Summit Copenhagen, June 2017. Available at: https://www.youtube.com/watch?v=PWBTOhMemxQ, accessed 18.12.2018

[26] John A De Goes. Blazing Fast, Pure Effects without Monads. LambdaConf conference, Boulder, CO, USA, June 2018. Available at: https://www.youtube.com/watch?v=L8AEj6IRNEE, accessed 18.12.2018

[27] Parallel typeclass, cats library. Available at: https://typelevel.org/cats/typeclasses/parallel.html, accessed 18.12.2018

[28] George Leontiev, There's a prolog in your scala. ScalaIO conference, Paris, France, October 2014. Available at: https://www.youtube.com/watch?v=iYCR2wzfdUs, accessed18.12.2018

[29] A.M. Troitskiy, D.V. Buzdalov. A static approach to estimation of execution time of components in AADL models. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 157-172, doi: 10.15514/ISPRAS-2016-28(2)-10

[30] Nada Amin, Tiark Rompf, Martin Odersky. Foundations of Path-Dependent Types. In Proc. of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, 2014, pp. 233-249