**УДК 004.434**

**ОБЗОР ФУНКЦИОНАЛЬНЫХ ЛОГИЧЕСКИХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ ДЛЯ МЕДИЦИНСКИХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ**

**© 2021 г. И.А.Юхновский1, А.Г.Мелузов2**

*Нижегородский Государственный Технический Университет им.Р.Е.Алексеева, Образовательно-научный институт ядерной энергетики и технической физики им. академика Ф.М. Митенкова Кафедра «Биоинженерия и ядерная медицина»*

*(г. Н. Новгород, ул. Минина, д.28Л, 5 корпус, ауд. 5302)*

*E-mail:* [*1juhnowski@gmail.com*](mailto:1juhnowski@gmail.com)*, 2TBD*

**Логические функциональные языки программирования могут играть важную роль в разработке диагностических программных систем. По мере того, как встраиваемые устройства становятся повсеместными и выполняют критически важные задачи в нашей жизни, их надежность приобретает первостепенное значение. Функциональные языки программирования обладают важным для надежности свойством – отсутствие побочных эффектов. Логические языки программирования позволяют в более естественной для диагностики форме записывать алгоритмы постановки диагнозов – в виде логических правил. Поддержка современными функциональными языками многопоточности в режиме реального времени и сконструированные на их базисе логические языки позволяют рассматривать их как перспективные технологии для разработки встраиваемых диагностических систем реального времени для медицинского оборудования. В статье исследуются логические языки программирования с бэкендом на существующих компиляторах функциональных языков программирования, и их применимость к встроенным системам реального времени. Определяя важные характеристики, необходимые для языка программирования реального времени, в статье исследуется, насколько хорошо существующие языки соответствуют этим характеристикам. Также был проведен эмпирический анализ времени выполнения, чтобы оценить влияние динамического управления памятью на предсказуемость и производительность. Наконец, были рассмотрены различные модели логического программирования для постановки диагнозов в реальном времени.**

**ОБРАЗЕЦ ЦИТИРОВАНИЯ**

И.А.Юхновский, А.Г.Мелузов Обзор функциональных логических языков программирования для медицинских систем реального времени.

**Введение**

Первоначально логические языки программирования создавались для искусственного интеллекта, а также для задач дискретной математики, используя парадигму программирования, основанную на автоматическом доказательстве теорем методом логического вывода информации на основе заданных фактов и правил вывода. Первым языком логического программирования был язык Planner - функционально-логический язык программирования, схожий по своему синтаксису с Лиспом. Разработан в Лаборатории искусственного интеллекта Массачусетского технологического института Карлом Хьюттом (англ. Carl Hewitt) в 1967—1971 годы. Planner использовался для того, чтобы понизить требования к вычислительным ресурсам (с помощью бэктрекинга — поиска с возвратом) и обеспечить возможность вывода фактов, без активного использования стека. Затем был разработан язык Prolog, который не требовал плана перебора вариантов и был, в этом смысле, упрощением языка Planner.

Основной преградой, ставшей на пути выхода этих технологий в массовое промышленное производство – было высокое требование к аппаратной платформе, которое на сегодняшний момент потеряло актуальность.

Иерархия влияния логических языков программирования приведена ниже:

* LISP
  + Planner
    - QA-4
    - Popler
    - Conniver
    - QLISP
    - Prolog
      * Mercury
      * Visual Prolog
      * Oz
      * Fril
      * miniKanren

Из них Mercury имеет бэкенд на JVM и Erlang, а miniKanren вошел в clojure/core.logic библиотеку.

Нужно отметить, что Java первоначально создавалась для встроенных систем и существуют JVM реального времени [32-40]. На базе JVM были созданы функциональные языки программирования, такие как Clojure, Scala, Kotlin, в то время как Java остается объектно-ориентированным языком.

Если говорить о доли UI в современном программном обеспечении, то на него приходится большая часть кода, поэтому важно, чтобы технология поддерживала средства визуализации. На текущий момент на рынке есть лидирующая платформа вне всякой конкуренции – это Android. Платформа Android это Linux OS с запущенной JVM с богатыми библиотеками как по UI, так и по средствам связи, однако платформа предлагает Soft Real Time что для некоторых задач может иметь ограничение, но нет принципиальных сложностей построить Linux kernel в режиме RT.

Таким образом, если в качестве медицинской платформы брать Android то возможна реализация диагностических модулей «в нативе» для Hard Real-Time и на JVM при требованиях Soft Real-Time.

Перспектива развития логических функциональных языков программирования видится через расширения бэкенда языка, так, ничего не мешает в Mercury дописать модуль трансляции на новый функциональный язык, а для clojure/core.logic можно использовать склонность самой технологии Clojure «парковаться» к различным платформам, не ограничиваясь JVM, например JavaScript, Dart.

Таким образом задача о возможности использования логических функциональных языков сводится сначала к рассмотрению их возможного бэкенда – функциональных языков, а затем уже к применимости технологии на доменном уровне.

**1. Функциональные языки**

Программное обеспечение реального времени является неотъемлемой частью многих сложных программно-аппаратных систем, которые требуют гарантий времени в дополнение к традиционным гарантиям функциональной корректности. Такие системы различаются по своей природе и охватывают различное медицинское оборудование. Все эти системы предоставляют услуги, полезность которых определяется своевременностью их реакции на внешние (например, показания датчиков) и внутренние (например, программные события, прерывания) стимулы. Написание программного обеспечения для этих систем - трудное дело, потому что для обеспечения правильности системы требуются гарантии предсказуемости. Это усугубляется тем фактом, что большая часть программного обеспечения для систем реального времени написана на императивных языках «низкого уровня» (С/С++).

Функциональные языки известны своей способностью обосновывать правильность, часто с помощью автоматизированных инструментов [2-4]. Учитывая, что медицинские системы реального времени часто развертываются в критических для безопасности и критических ситуациях, важна способность писать программное обеспечение с высокой степенью уверенности в его правильности и предсказуемости. Функциональные языки хорошо подходят для первой задачи (корректность), но не разработаны для второй (предсказуемость). Действительно, функциональные языки программирования общего назначения не используются для разработки систем реального времени, и не проводятся исследования для обеспечения их предсказуемости. Однако существует множество функциональных языков, специфичных для предметной области, которые обеспечивают предсказуемость для нишевых доменов. Состояние основных языков функционального программирования с точки зрения их способности выражать и обеспечивать предсказуемость для медицинского домена — это то, что мы хотели бы исследовать в этом обзоре.

Одна из причин, по которой функциональные языки не рассматривались для написания программного обеспечения реального времени, заключалась в том, что они являются управляемыми языками, а автоматическое управление памятью традиционно считалось непредсказуемым, однако появилась концепция сборщика мусора в реальном времени - RTGC [5-23], которая уже давно изучается в контексте Java реального времени, что привело ко многим достижениям и развертыванию систем реального времени с использованием автоматического управления памятью.

Кроме того, было показано, что память с ограниченным объемом [23-28], схема автоматического управления регионами памятью, а также альтернативные высокоуровневые методы [29,30] управления памятью обеспечивают предсказуемое управление памятью.

Важными особенностями языков функционального программирования в реальном времени (RTFP) являются детерминированное поведение, ограниченность в пространстве / времени, асинхронная отзывчивость (реактивность), параллелизм, доказуемая корректность, периодическое планирование и прерывание/опрос по требованию. [31,32]

Исследуем эти особенности более подробно:

* Модель RT – **Hard** или **Soft** Real-Time
* Реализация многопоточности - потоки, задачи или другие примитивы, используемые для представления вычислений, необходимы для выражения систем реального времени, которые не могут быть закодированы как циклический исполнительный механизм. В сочетании с основным механизмом выражения вычислений как задач есть способность различать вычисления на основе приоритетов, а также выражать характеристики вычислений (например, период и характеристики выпуска). Используется классификация потоков или поддержки задач языка в зависимости от примитивов, которые предоставляет язык:
  + **Зеленые** потоки - если потоки поддерживаются только библиотекой времени выполнения и мультиплексируются в одном потоке ОС.
  + **Собственные** потоки - если язык предоставляет способ создания потоков ОС и методы управления ими.
  + Язык также может поддерживать **оба** типа - зеленые и собственные потоки
  + или **другой** - который инкапсулирует множество других примитивов параллелизма, таких как фьючерсы и асинхронные рабочие процессы.
* Чистота – в контексте встроенных систем реального времени ввод/вывод обязателен, поэтому:
  + **Получистый** - язык является функциональным, сохраняет ссылочную прозрачность, но разрешает ввод-вывод, как заведомо операцию с побочными эффектами.
  + **Нечистый** - если язык, представляет собой смесь функциональных и императивных конструкций (например, Scala)
* Межпоточное взаимодействие - язык может поддерживать:
  + общую память (**ОП**), в которой потоки изменяют общую ячейку памяти,
  + передачу сообщений (**ПС**), при которой потоки взаимодействуют явно, отправляя и получая сообщения.
  + **оба** стиля взаимодействия.

Этот выбор полезен при принятии решения, какой язык использовать для конкретного приложения реального времени. Например, Erlang предпочитает использовать только передачу сообщений, поскольку он обычно используется для построения крупномасштабных распределенных систем (мягкое реальное время), и предоставление контролируемого доступа к общей памяти для такого большого приложения было бы непрактичным.

Большинство языков, которые мы рассматриваем, имеют возможность вызывать функции C через интерфейс внешних функций (FFI). Вызов C без ущерба для свойств, предоставляемых языком, требует осторожного использования, и поэтому мы не рассматриваем его как часть общего стиля взаимодействия.

* Реактивные системы — это системы, которые реагируют на событие. Функциональное реактивное программирование (FRP) [35] — это программный framework, который дает декларативный способ выражения реактивных систем. FRP основано на изменяющихся во времени реактивных значениях, называемых поведением, и упорядоченных по времени последовательностях дискретных событий, называемых событиями. Они смоделированы как параметрические типы данных и изначально были встроены в Haskell [33]. FRP успешно использовалось во многих областях, от интерактивной анимации [34] до компьютерного зрения [37] и робототехники [38]. FRP — это естественный способ моделирования систем, в которых язык должен явно моделировать концепцию непрерывного времени. Амсден [39] рассматривает различные реализации FRP, и эта работа дает нам хороший обзор существующих реализаций FRP. Вычисления FRP не ограничены в терминах время и место их выполнения, хотя это кажется хорошим способом выразить системы реального времени, которые по своей сути являются реактивными. FRP был реализован, например, как в Haskell, однако рассуждения о предсказуемости FRP оказались нетривиальны. Это привело к разработке FRP реальном времени (RT-FRP) [36], который является подмножеством FRP, но с ограничениями затрат на выполнение для реактивной части. RTF-RP не встраивает новый язык в основной язык, а вместо этого разделяет модель программирования, чтобы иметь реактивную часть, которая имеет дело со временем и реакциями, и базовый язык, который имеет дело со статическими вычислениями. Базовым языком может быть любой язык, который также лимитирует затраты на выполнение. Большая часть медицинских систем реального времени - времени — это системы, управляемые событиями. Системы, управляемые событиями, обычно изменяют состояние своей программы и выдают выходной сигнал только при наличии внешнего события и бездействуют, когда событий нет. RT-FRP был построен с глобальными часами, и все части программы обновлялись с одинаковой частотой, то есть с каждым тактом часов. Можно использовать RT-FRP для создания систем, управляемых событиями, но за потери счет производительности, поскольку конструкция RT-FRP распространяет каждое событие по всей программе. Для решения этого недостатка FRP была создана концепция FRP управляемого событиями (E-FRP) [40], а глобальные часы RT-FRP были обобщены для набора событий. Структура E-FRP имеет формально проверяемую стратегию компиляции для преобразования декларативного кода E-FRP высокого уровня в императивную реализацию более низкого уровня, которая состоит из событий и их обработчиков. Хотя это очень хорошая стратегия для выражения управляемой событиями подмножества систем реального времени, E-FRP выполняет события атомарно и одинаково, так что все события имеют одинаковую верхнюю границу времени, необходимого для ответа на это событие. Однако у E-FRP оказался свой недостаток – сложность с событиями, которые имеют более высокий приоритет чем другие, поэтому был предложен вариант E-FRP с приоритетами, названный P-FRP [41]. P-FRP предоставил возможность связывать приоритеты с каждым событием, сохраняя при этом операционную семантику, заданную E-FRP, тем самым обеспечивая те же гарантии, что и последний. Эти модели программирования определили элегантный способ представления значительной части систем реального времени, которые реагируют только на внешние события. Они также дают нам жесткую верхнюю границу времени, необходимого для реагирования на такие события, что помогает гарантировать отсутствие сбоев в заданный интервал времени. Однако и этот метод имел свои слабые стороны – случайная смена приоритета могла приводить к неожиданным результатам, например нотификация завершения отправки файла, до завершения его передачи при повышении приоритета нотификации по сравнению с коммуникационным таском. Учитывая глубину исследований в области реактивного программирования и его видимую значимость для проектирования систем реального времени, каждый язык был оценен с точки зрения того, в какой степени они поддерживают стиль программирования FRP. Хотя на момент описания статьи нет ни одного популярного языка, который поддерживает RT-FRP, E-FRP или P-FRP, есть все основания ожидать модификации существующих языков для поддержки этих концепций. Таким образом:
  + **Поддерживает** - язык поддерживает FRP, даже если он изначально не имеет реализации FRP, но есть сторонние библиотеки, которые его предоставляют.
  + **Готов** поддерживать FRP - язык явно не поддерживает FRP, но имеет такие конструкции, как Actors или передачу сообщений, которые являются строительными блоками модели FRP.
  + **Нет** – соответственно, лишен поддержки FRP.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Язык | Модель RT | Тип потоков | Чистота | Взаимодействие | Реактивность |
| Atom | Hard | Другие | Получистый | ОП | Поддерживает |
| Clojure | Soft | Оба | Получистый | ОП | Готов |
| Copilot | Hard | Другие | Получистый | ОП | Поддерживает |
| Erlang | Soft | Зеленые | Нечистый | ПС | Готов |
| Haskell | Soft | Другие | Получистый | Оба | Готов |
| Hume | Hard | Зеленые | Получистый | ПС | Поддерживает |
| Idris | Soft | Оба | Получистый | Оба | Нет |
| Racket | Soft | Другие | Нечистый | ПС | Поддерживает |
| Scala | Soft | Другие | Нечистый | ПС | Готов |
| SML | Soft | Зеленые | Получистый | Оба | Поддерживает |
| Timber | Hard | Нативные | Получистый | ПС | Готов |

**2. Предсказуемость функциональных языков**

**2.1 Haskell (Atom, Copilot)**

Стандартный сборщик мусора, включенный в состав Glasgow Haskell Compiler (GHC), является сборщиком копирования 42 65 с моделью stop-the-world (STW), генерирующим копирование. Существуют различные настраиваемые параметры, например сжатие на месте для пространства старого поколения. Однако этот сборщик мусора не подходит для программного обеспечения реального времени, поскольку хорошо известно, что сборщики мусора, работающие в режиме остановки, оказывают негативное влияние на соблюдение сроков в реальном времени. Например, Pizlo [20] показывает, что STW GC при высокой нагрузке может пропустить до 95% сроков. Двумя заметными усилиями по реализации подходящего GC в реальном времени являются исследования Microsoft в области многоядерного GC с локальными кучами [43] и проект Google Summer of Code (GSoC) [44] по внедрению IMMIX для Haskell. Усилия GSoC дали результат, который «еще не находится в состоянии, когда прототип может быть включен в GHC, но он работает и дает лучшие результаты, чем GC по умолчанию [45].

Сборщик мусора IMMIX [46] представляет собой гибридный сборщик, который объединяет характеристики маркировки / очистки и маркировки / сжатия в то, что называется «областью маркировки».

Сегментируя память на фрагменты и далее сегментируя фрагменты на строки (небольшие последовательные фрагменты памяти), можно оптимизировать маркировку путем переноса меток с уровня строки на уровень блока. Это позволяет оптимизировать поиск доступных сегментов памяти. Кроме того, выделение на уровне блока позволит распределителю быть зависимым от потока и избежать синхронизации на уровне подблока (строки) для повышения производительности мутатора. IMMIX представляет новую стратегию дефрагментации, которая является гибкой - перемещение строк только тогда, когда это имеет смысл. IMMIX сочетает это с указателями пересылки, чтобы еще больше минимизировать влияние на мутатор. Благодаря тщательной оптимизации и проверке оптимизаций компилятора достигается уровень производительности, который последовательно превосходит три канонические стратегии сборки мусора:

1. метка-развертка (MS),
2. полупространство (SS),
3. метку-компактность (MC).

Эмпирические данные IMMIX охватывают широкий спектр тестов и широкий спектр показателей, включая производительность распределения, метаданные / накладные расходы на маркировку на уровне линии и блока, а также производительность уплотнения.

Еще одной интересной деталью реализации было резервирование блоков (перемычка) для облегчения эвакуации. Усилия IMMIX GSoC показывают большие перспективы, поскольку интеграция IMMIX GC в Haskell, безусловно, будет полезна для будущего GC реальном времени.

Работа Microsoft над многоядерным сборщиком мусора включала разработку независимого сборщика мусора, способного работать на нескольких ядрах. Это позволяет каждому процессору независимо выполнять незначительную сборку мусора. Они фокусируются на пропускной способности, а не на задержке и времени паузы, имея локальные для процессора кучи вместе с глобальной кучей со строго определенными правилами пересылки. Это похоже на подход, принятый MultiMLton [47], и, хотя он не сразу применим к системам реального времени, он перспективен для адаптации изучения кучи уровня приоритета (а не процессора) в связке со сборщиком мусора на уровне кучи и, следовательно, на уровне приоритета.

GHC имеет исчерпывающую поддержку параллелизма [48], что позволяет сделать вывод, что его можно адаптировать для разработки в реальном времени с небольшими усилиями.

Одним из таких достижений является Atom [49], предметно-ориентированный язык, используемый в автомобильной промышленности для реализации функций жесткого реального времени. Atom необычен тем, что это синхронный язык, похожий на HDL, где его функции выполняются на основе времени (прошедшие тактовые циклы), а не на событиях (реактивное программирование) или на планировщике приоритетов. По умолчанию Atom генерирует циклический исполнитель [50], который представляет собой набор элементарных правил перехода между состояниями и работает с заданной частотой. GHC проверяет эту исполнительную программу и оптимизирует ее во время компиляции, а не во время выполнения, так что ОС реального времени (RTOS) не требуется для выполнения этой функции. Компилятор также изменяет правила и пытается минимизировать время выполнения наихудшего случая (WCET) исполнительной системы.

Ключевая функция, необходимая компилятору Atom для правильного вычисления WCET – отсутствие необходимости атомарной блокировки задач. Он выполняет переупорядочение задач и использует атомарное определение задачи, чтобы избежать необходимости учитывать взаимозависимости при расчете частоты исполнительных механизмов. Рекомендуемая реализация - использовать прерывания часов, установленные для запуска списка задач, предназначенных для выполнения с этим интервалом. Одним из ограничений его использования в других доменах является отсутствие динамического распределения памяти (и GC). Atom создает код C, в котором предварительно объявлены переменные с минимальным набором поддерживаемых типов, чтобы облегчить низкоуровневый аппаратный контроль и измерение простых систем.

Еще один многообещающий DSL — это Copilot [51], язык, предназначенный для программирования мониторов времени выполнения для распределенных реактивных систем жесткого реального времени. Copilot использует Atom для генерации кода C в реальном времени и, таким образом, генерирует код, который является постоянным во времени и пространстве.

Есть также исследования в области RT-FRP [52] относительно Haskell. Подобно подходу, принятому в Atom и Copilot, RT-FRP ограничивает затраты как на время выполнения, так и на пространство. Модель FRP сочетает в себе непрерывные вычисления с вычислениями на основе событий, типичными для медицинских систем. Приложения, сочетающие эти два режима вычислений, такие медицинские мониторы, водители ритма хорошо подходят для FRP. Yale Haskell Group [53] имеет множество примеров встроенных доменно-ориентированных языков (DSL), в которых используется модель FRP.

Поскольку эта модель сразу же применима к медицинским системам, работа, проделанная над RT-FRP, является отличной отправной точкой для продолжения исследований в этой области. Однако, как отмечалось ранее, для использования этой модели при разработке основных систем реального времени потребуется, чтобы базовый язык также был ограничен по времени и пространству выполнения, чего в Haskell в настоящее время нет. Существуют и другие DSL Haskell, такие как Dance [54], но мы не рассматриваем их, поскольку они имеют тенденцию быть похожими по реализации.

В целом, GHC можно отнести к категории языков для медицинских систем мягкого реального времени с перспективой применимости к системам жесткого реального времени. Для существующих DSL с жестким режимом реального времени доработки в областях GC, динамических типов и конструкций планировщика в отношении спецификации допустимого джиттера, чтобы можно было реализовать более интуитивное обнаружение пропуска крайнего срока.

В общем, перегрузка операторов и расширяемость Haskell делают его привлекательным для экспериментов с новыми подходами к применению концепций функционального программирования в режиме реального времени.

**2.2 SML**

SML (MLton) демонстрирует тот же образец вмешательства сборки мусора в предсказуемость вычислений [55-72].

**2.3 Scala**

Когда нет давления на память, Scala ведет себя предсказуемо, в отличии от нехватки памяти.

**2.4 Clojure**

Поскольку Scala и Clojure используют один сборщик мусора JVM, то результаты тестов по предсказуемости можно сравнить: если использовать кучу одинакового размера для Scala и Clojure и изменять размер дерева только для заполнения этой кучи, то Clojure будет более предсказуема, чем Scala.

**2.5 Erlang**

Erlang поддерживает асинхронную передачу сообщений, что означает, что поток может отправить сообщение другому потоку и продолжить свою работу, не дожидаясь, пока получатель получит сообщение. Однако поток, получающий сообщение, блокируется до тех пор, пока в очереди сообщений не появится соответствующее сообщение, что вносит фактор непредсказуемости во время выполнения.

Erlang поддерживает асинхронную передачу сообщений, что означает, что поток может отправить сообщение другому потоку и продолжить свою работу, не дожидаясь, пока получатель получит сообщение. Однако поток, получающий сообщение, блокируется до тех пор, пока в очереди сообщений не появится соответствующее сообщение, что вносит фактор непредсказуемости во время выполнения.

Среднее время выполнения простого цикла намного выше с передачей сообщений, чем без. Стандартное отклонение также выше, поскольку накладные расходы включают затраты на приоритетное обслуживание и синхронизацию.

Еще одним накладным расходом на передачу сообщений в Erlang является стоимость копирования сообщения из кучи одного процесса в другой. В языках с общей кучей таких накладных расходов нет, но Erlang пытается снизить эти затраты, используя двоичную кучу. Что означает, если сообщение содержит только двоичные данные, например битовой последовательности или строки, преобразованной в битовую последовательность, можно уменьшить накладные расходы на копирование. Если ваше сообщение содержит что-то еще, например, большой набор цифр, процесс будет намного медленнее.

Erlang в целом более предсказуем, чем другие языки, рассмотренные выше. Анализ возможности планирования в Erlang проще по сравнению с другими языками, так как нет необходимости анализировать шаблоны распределения каждого потока изолированно, но возможно быстрое усложнение в случае, когда потоки обмениваются данными посредством передачи сообщений, поскольку теперь мы должны рассуждать о затратах на помехи, связанных с передачей сообщений.

**3. Пример реализаций гидролокатора на различных функциональных языках**

Рассмотрим пример сонара из введения Timber [73], который в дальнейшем можно адаптировать, например, для УЗИ.

Цель прототипа состоит в том, чтобы оценить, как приложение может быть написано на нескольких выбранных функциональных языках. Само приложение не предназначено для демонстрации всех возможных функций, которые можно было бы ожидать от современного языка реального времени, но для демонстрации базового набора функций языка с поддержкой в реальном времени этого примера будет достаточно.

Прототип позволит нам сравнивать языки с синтаксической точки зрения и оценивать, насколько доступны программисту функции реального времени.

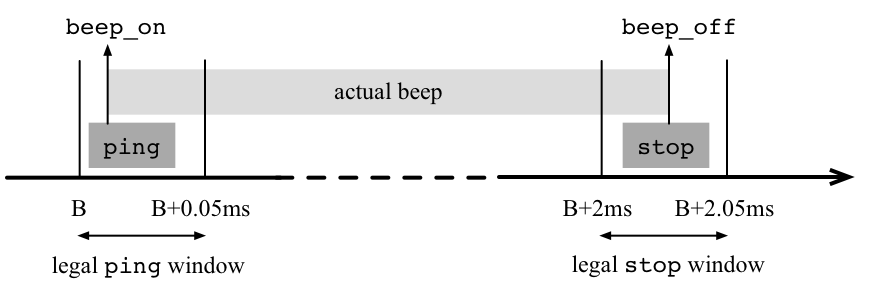


Рис.1: Сонар – работа приложения реального времени

Звуковой сигнал гидролокатора должен составлять 2 миллисекунды с максимальным джиттером 50 микросекунд, и требуемая точность измерений диктует, что временные метки, связанные с сигналами, также должны иметь точность вплоть до диапазона 50 микросекунд. На рисунке показаны временные окна, ограничивающие задействованные методы ping и stop, и, таким образом, косвенно, фактический звуковой сигнал. Кроме того, гидролокатор в примере должен звучать каждые 3 секунды, а крайний срок реакции на измерения вне пределов составляет 5 миллисекунд.

Timber позволяет нам указывать периодичность задач (и гарантирует, что они выпускаются в надлежащее время), а также позволяет нам указывать джиттер, который помогает в приложениях, в которых допустимо несоблюдение крайнего срока, если это происходит в течение определенного времени после крайнего срока. Другие DSL, такие как Atom, имеют необходимый набор инструментов для выражения этих функций, хотя и не так, как Timber, но будет интересно посмотреть, сможем ли мы выразить этот пример на таком языке, как Erlang, который является функциональным языком общего назначения и широко используется в Soft Real-Time приложениях.

Далее приведены листинги этого примера на четырех языках: Timber, Atom, SML и Erlang, также вы можете скачать последнюю версию с GitHub: <https://github.com/juhnowski/lfp>

**3.1 Timber**

sonar port alarm critical =

**class**

tm = **new** timer

count := 0

ping = **before** (microsec 50) **action**

port.write beep\_on

tm.reset

**after** (millisec 2) stop

**after** (sec 3) ping

stop = **before** (microsec 50) **action**

port.write beep\_off

echo = **before** (millisec 5) **action**

diff <- tm.sample

**if** critical diff **then**

count := count + 1

alarm count

**result** {interrupt = echo, start = ping }

**3.2 Atom (Haskell)**

checkSensor :: Word16 -> Atom () -> Atom ()

checkSensor threshold overThresholdAction = atom "check\_sensor" $ **do**

ready <- **return** $ bool’ "sensor\_ready"

sensorValue <- **return** $ word16’ "sensor\_value"

warmup <- timer "warmup"

beepOn <- bool "beep\_on" **False**

period 3000 $ **phase** 000 $ atom "beepOn" $ **do**

call "beep\_on"

beepOn <== true

startTimer warmup $ Const 10

period 3000 $ **phase** 050 $ atom " beepOff " $ **do**

call "beep\_off"

beepOn <== false

**3.3 Erlang**

-**module**(sonar).

-**export** ([start/0, beep/2, period/1]).

period(PID)->

**spawn**(sonar, beep, [**self**(), done]),

**receive** done -> PID ! done

**end**.

beep(PID, Signal) ->

io:format("Beep On ~n", []),

**receive**

**after** 2 -> io:format("Beep Off ~n", []),

PID ! Signal

**end**.

start() ->

**spawn**(sonar, period, [**self**()]),

**receive** done-> timer:sleep(3000),

start()

**end**.

**3.4 SML**

**open** Posix.Signal

**open** MLton

**open** Itimer Signal Thread

**fun** beepOff () = print "Beep Off \n"

**fun** beepOn () = (print "Beep On\n";

loop 5000)

**fun** beep () = (Thread.spawn beepOn;

Thread.spawn beepOff;

Thread.runAfter (50))

**fun** period () = (beep ();

period ())

**val** \_ = (Thread.spawn (period ());

Thread.runAfter (3000000))

**3.5 Пример медицинского прибора**

Данные алгоритмы могут быть реализованы на современной аппаратной базе и послужить базой для модернизации устаревшего медицинского оборудования.



Рисунок – устаревший аппарат УЗИ

**4. Пример реализации прибора диагностики на логических языках программирования**

В [74] вводится понятие факта:

plus(0,0,0). plus(0,1,1). plus(0,2,2). plus(0,3,3).

plus(0,0,0). plus(0,1,1). plus(0,2,2). plus(0,3,3).

plus(0,0,0). plus(0,1,1). plus(0,2,2). plus(0,3,3).

В [75] вводится понятие реляционного программирования. Рассмотрим функция pluso, которая выполняет сложение натуральных чисел.

(pluso1 2 3) успешно, поскольку 1 + 2 = 3, то есть тройка (1,2,3) находится в тройном отношении сложения. Можно использовать плюс, чтобы сложить два числа: (pluso1 2 z) что свяжет логическую переменную z с 3. Также можно вычитать числа, используя pluso: (pluso 1 y 3) связывает y с 2, поскольку 3 − 1 = 2. Можно даже вызвать плюс только с логическими переменными: (pluso x y z) что даст бесконечное количество ответов, в которых натуральные числа, связанные с x, y и z, удовлетворяют условию x + y = z. Например, один такой ответ связывает x с 3, y с 4 и z с 7.

По данному шаблону можно легко реализовать систему поиска диагноза по симптомам:

search (

symptom 1,

symptom 2,

symptom 3,

… ,

diagnosis)

Аппаратное решение, выполненное для медицинского прибора для работников скорой помощи приведен на рисунке:

|  |  |
| --- | --- |
|  |  |

Рисунок – прибор для диагностики. Слева – аппаратное решение логики поиска диагноза на диодной матрице, справа – внешний вид прибора.

**Заключение**

В этой статье мы рассмотрели различные варианты языков функционального программирования и построенных поверх логических фреймвороков с рядом атрибутов исполнения в системах реального времени. Некоторые из этих атрибутов, такие как моделирование ввода-вывода и задач, являются требованиями для создания полезных приложений реального времени, в то время как другие атрибуты, такие как управление динамической памятью, желательны, но не обязательны. Языки с обширной поддержкой параллелизма больше подходят для моделирования системы реального времени как систем, основанных на задачах. Обычно такие языки мультиплексируют примитивы параллелизма в потоке ОС и, следовательно, могут легко использовать RTOS для использования планировщика RTOS для планирования задач. Языки, в которых отсутствует контроль параллелизма, могут по-прежнему моделировать системы как циклические управляющие, но будут ограничены типом систем реального времени, которые могут быть написаны с использованием этого языка. Есть языки, такие как Haskell, которые благодаря гибкости своего компилятора являются хостом для широкой экосистемы DSL в реальном времени. Другие языки более специализированы, например Hume, и, как следствие, имеют большую глубину с точки зрения инфраструктуры доказательства и анализа. Наконец, есть такие языки, как Scala, в которые еще не были вложены значительные средства в реальном времени на лингвистическом уровне, но благодаря использованию JVM они имеют существенные перспективы в области исследований в реальном времени.

Логические языки программирования хорошо приспособлены для решения медицинских задач по нахождения диагноза по симптомам, принятия решений, что вместе с функциональной поддержкой реального времени предоставляет разработчику универсальный инструмент для разработки встроенных медицинским программ.

Данные технологии будут интересны как разработчикам ПО для вновь проектируемых медицинских приборов, так и для обслуживающих и ремонтных организация для замены вышедших из строя или устаревших аналоговых модулей цифровыми решениями. Простота кода делает такие альтернативные решения инспектируемыми и предсказуемыми, что важно для надежности модернизируемых устройств.

**ЛИТЕРАТУРА**

1. Иван Братко. Алгоритмы искусственного интеллекта на языке PROLOG. — М.: «Вильямс», 2004. — С. 640. — ISBN 0-201-40375-7.

2. Audebaud Philippe, Paulin-Mohring Christine. Proofs of randomized algorithms in Coq. Science of Computer Programming. 2009;74(8):568 - 589. Special Issue on Mathematics of Program Construction (MPC 2006).

3. Conchon Sylvain, Contejean Evelyne, Kanig Johannes, Lescuyer Stéphane. Lightweight Integration of the Ergo Theorem Prover Inside a Proof Assistant. In: AFM ’07:55–59ACM; 2007; New York, NY, USA.

4. Contejean Évelyne,Paskevich Andrei, Urbain Xavier, Courtieu Pierre,Pons Olivier, Forest Julien.A3PAT, an Approach for Certified Automated Termination Proofs. In: PEPM ’10:63–72ACM; 2010; New York, NY, USA.

Bacon David F., Cheng Perry, RajanV. T.. Controlling Fragmentation and Space Consumption in the Metronome, a Real-time Garbage Collector for Java. In: LCTES ’03:81–92ACM; 2003; New York, NY, USA.

5. Baker Henry G.. The treadmill: real-time garbage collection without motion sickness. SIGPLAN Not.. 1992;27(3):66–70.

6. Kalibera Tomas. Replicating real-time garbage collector for Java. In: JTRES ’09:100–109ACM; 2009; New York, NY, USA.

7. Cheng Perry, Blelloch Guy E.. A Parallel, Real-time Garbage Collector. In: PLDI ’01:125–136ACM; 2001; New York, NY, USA.

8. Pizlo Filip, Vitek Jan. Memory Management for Real-Time Java: State of the Art. In: ISORC ’08:248–254IEEE Computer Society; 2008; Washington, DC, USA.

9. Kato Shinpei, IshikawaYutaka, Rajkumar Ragunathan (Raj).CPUScheduling and MemoryManagement for Interactive Real-time Applications. Real-Time Syst.. 2011;47(5):454–488.

10. Pizlo Filip, Ziarek Lukasz, Maj Petr, Hosking Antony L., Blanton Ethan, Vitek Jan. Schism: fragmentation-tolerant real-time garbage collection. In: PLDI ’10:146–159ACM; 2010; New York, NY, USA.

11. Cho Hyeonjoong, Na Chewoo, Ravindran Binoy, Jensen E. Douglas. On scheduling garbage collector in dynamic real-time systems with statistical timing assurances. Real-Time Syst.. 2007;36(1-2):23–46.

12. Lim Tian F., Pardyak Przemyslaw, Bershad Brian N.. A memory-efficient real-time non-copying garbage collector. In: ISMM ’98:118–129ACM; 1998; New York, NY, USA.

13. Bacon David F., Cheng Perry, Rajan V. T.. A real-time garbage collector with low overhead and consistent utilization. In: POPL ’03:285– 298ACM; 2003; New York, NY, USA.

14. Higuera-Toledano M. Teresa, Issarny Valérie. Improving the memory management performance of RTSJ: Research Articles. Concurr. Comput. : Pract. Exper.. 2005;17(5-6):715–737.

15. Kalibera Tomas, Pizlo Filip, Hosking Antony L., Vitek Jan. Scheduling real-time garbage collection on uniprocessors. ACM Trans. Comput. Syst.. 2011;29(3):8:1–8:29.

16. Nettles Scott,O’Toole James. Real-time Replication Garbage Collection. In: PLDI ’93:217–226ACM; 1993; New York, NY, USA.

17. Nilsen Kelvin. High-level dynamic memory management for object-oriented real-time systems. SIGPLAN OOPS Mess.. 1996;7(1):86–93.

18. Siebert Fridtjof. Realtime Garbage Collection in the JamaicaVM 3.0. In: JTRES ’07:94–103ACM; 2007; New York, NY, USA.

19. Pizlo Filip, Petrank Erez, Steensgaard Bjarne. A Study of Concurrent Real-time Garbage Collectors. In: PLDI ’08:33–44ACM; 2008; New York, NY, USA.

20. Pizlo Filip, Frampton Daniel, Petrank Erez, Steensgaard Bjarne. Stopless: A Real-time Garbage Collector for Multiprocessors. In: ISMM’07:159–172ACM; 2007; New York, NY, USA.

21. Auerbach Joshua, Bacon David F., Cheng Perry, et al. Tax-and-spend: Democratic Scheduling for Real-time Garbage Collection. In: EMSOFT’08:245–254ACM; 2008; New York, NY, USA.

22. Pizlo Filip, Hosking Antony L., Vitek Jan. Hierarchical Real-time Garbage Collection. In: LCTES ’07:123–133ACM; 2007; New York, NY, USA.

23. oSCJ . Computer Science Department Annual Report, Purdue University . http://www.ovmj.net/oscj/publications/Documents/oSCJ-Report.pdf. Accessed May 2014; .

24. Beebee William S., Rinard Martin C.. An Implementation of Scoped Memory for Real-Time Java. In: EMSOFT ’01:289–305Springer-Verlag;2001; London, UK, UK.

25. Hamza H., Counsell S.. Region-Based RTSJ Memory Management: State of the art. Sci. Comput. Program.. 2012;77(5):644–659.

26. Deters Morgan, Cytron Ron K.. Automated discovery of scoped memory regions for real-time Java. In: ISMM ’02:132–142ACM; 2002; New York, NY, USA.

27. Zhao Tian, Noble James, Vitek Jan. Scoped Types for Real-Time Java. In: RTSS ’04:241–251IEEE Computer Society; 2004;Washington, DC,USA.

28. Parain Frédéric. Region-Based Memory Management for Real-Time Java. In: ISORC ’01:387–IEEE Computer Society; 2001;Washington, DC,USA.

29. Andreae Chris, Coady Yvonne, Gibbs Celina, Noble James, Vitek Jan, Zhao Tian. Scoped types and aspects for real-time Java memory management. Real-Time Syst.. 2007;37(1):1–44.

30. Auerbach Joshua, Bacon David F., Guerraoui Rachid, Spring Jesper Honig, Vitek Jan. Flexible task graphs: a unified restricted thread programming model for Java. In: LCTES ’08:1–11ACM; 2008; New York, NY, USA.

31. Hammond Kevin. Is it time for real-time functional programming?. In: Gilmore Stephen, ed. Revised Selected Papers from the Fourth Symposium on Trends in Functional Programming, TFP 2003, Edinburgh, United Kingdom, 11-12 September 2003., Trends in Functional Programming, vol. 4::1–18Intellect; 2003.

32. McDermid John A.. Computing Tomorrow. In: New York, NY, USA: Cambridge University Press 1996 (pp. 217–245).

33. Wan Zhanyong, Hudak Paul. Functional Reactive Programming from First Principles. In: PLDI ’00:242–252ACM; 2000; New York, NY, USA.

34. Elliott Conal, Hudak Paul. Functional Reactive Animation. In: ICFP ’97:263–273ACM; 1997; New York, NY, USA.

35. Hudak Paul. The Haskell school of expression: learning functional programming through multimedia. Cambridge University Press; 2000.

36. Wan Zhanyong, TahaWalid, Hudak Paul. Real-time FRP. In: ICFP ’01:146–156ACM; 2001; New York, NY, USA.

37. Reid Alastair, Peterson John, Hager Greg, Hudak Paul. Prototyping Real-time Vision Systems: An Experiment in DSL Design. In: ICSE ’99:484–493ACM; 1999; New York, NY, USA.

38. Peterson J., Hager G. D., Hudak P.. A language for declarative robotic programming. In: Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C):1144-1151 vol.2; 1999.

39. Amsden Edward. A survey of functional reactive programming. Rochester Institute of Technology. 2011;

40. Wan Zhanyong, TahaWalid, Hudak Paul. Event-Driven FRP. In: Krishnamurthi Shriram, Ramakrishnan C. R., eds. Practical Aspects of Declarative Languages, :155–172Springer Berlin Heidelberg; 2002; Berlin, Heidelberg.

41. Kaiabachev Roumen, TahaWalid, Zhu Angela. E-FRP with Priorities. In: EMSOFT ’07:221–230ACM; 2007; New York, NY, USA.

42. Marlow Simon, Harris Tim, James Roshan P., Peyton Jones Simon. Parallel Generational-copying Garbage Collection with a Block-structured Heap. In: ISMM ’08:11–20ACM; 2008; New York, NY, USA.

43. Marlow Simon, Peyton Jones Simon. Multicore Garbage Collection with Local Heaps. In: ISMM ’11:21–32ACM; 2011; New York, NY, USA.

44. Silva Marco TÃžlio Gontijo. Implementing the Immix Garbage Collection Algorithm onGHC. <https://www.google-melange.com/gsoc/project/details/google/gsoc2010/marcot/5639274879778816>

45. Silva Marco TÃžlio Gontijo. Porting IMMIX To Haskell as a GSOC Project . [https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/GC/Immix. Accessed January 2018](https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/GC/Immix.%20Accessed%20January%202018).

46. Blackburn Stephen M., McKinley Kathryn S.. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In: PLDI ’08:22–32ACM; 2008; New York, NY, USA.

47. Sivaramakrishnan K. C., Ziarek Lukasz, Jagannathan Suresh. MultiMLton: A multicore-aware runtime for standard ML. Journal of Functional Programming. 2014;24:613–674.

48. Concurrency in Haskell . https://wiki.haskell.org/Concurrency. Accessed June 2018;

49. Hawkins Tom. Atom: A Synchronous Hard Real-Time EDSL for GHC . https://github.com/tomahawkins/atom. Accessed January 2018;

50. Locke C. Douglass. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. Real-Time Systems.;4(1):37–53.

51. Pike Lee, Goodloe Alwyn, Morisset Robin, Niller Sebastian. Copilot: A Hard Real-Time Runtime Monitor. In: LNCSSpringer; 2010. <http://www.cs.indiana.edu/~lepike/pub_pages/rv2010.html>.

52. Wan Zhanyong, TahaWalid, Hudak Paul. Real-time FRP. In: ICFP ’01:146–156ACM; 2001; New York, NY, USA.

53. Yale Haskell Group . http://haskell.cs.yale.edu/. Accessed October 2017;

54. Huang Liwen, Hudak Paul. Dance: A Declarative Language for the Control of Humanoid Robots. YALEU/DCS/RR-1253: Yale University; 2003.

55. Milner Robin, Tofte Mads, Macqueen David. The Definition of Standard ML. Cambridge, MA, USA: MIT Press; 1997.

56. Kumar Ramana, Myreen Magnus O., Norrish Michael, Owens Scott. CakeML: A Verified Implementation of ML. In: POPL ’14:179–191ACM; 2014; New York, NY, USA.

57. Reppy John H.. Concurrent Programming in ML. New York, NY, USA: Cambridge University Press; 1999.

58. Reppy John, Russo Claudio V., Xiao Yingqi. Parallel Concurrent ML. In: ICFP ’09:257–268ACM; 2009; New York, NY, USA.

59. Fluet Matthew, Rainey Mike, Reppy John, Shaw Adam. Implicitly-threaded Parallelism in Manticore. In: ICFP ’08:119–130ACM; 2008; New York, NY, USA.

60. Donnelly Kevin, Fluet Matthew. Transactional Events. J. Funct. Program.. 2008;18(5-6):649–706.

61. Effinger-Dean Laura, Kehrt Matthew, Grossman Dan. Transactional Events for ML. In: ICFP ’08:103–114ACM; 2008; New York, NY, USA.

62. Ziarek Lukasz, Sivaramakrishnan KC, Jagannathan Suresh. Composable Asynchronous Events. In: PLDI ’11:628–639ACM; 2011; New York, NY, USA.

63. Syme Don, Petricek Tomas, Lomov Dmitry. The F# Asynchronous Programming Model. In: Rocha Ricardo, Launchbury John, eds. Practical Aspects of Declarative Languages, :175–189Springer Berlin Heidelberg; 2011; Berlin, Heidelberg.

64. Elsman Martin. MLKit Documentation . http://www.elsman.com/mlkit/. Online. Accessed November 2017;

65. Li Muyuan, McArdle Daniel E.,Murphy Jeffrey C., Shivkumar Bhargav, Ziarek Lukasz. Adding Real-time Capabilities to aSMLCompiler. SIGBED Rev.. 2016;13(2):8–13.

66. Tofte Mads, Birkedal Lars, Elsman Martin, Hallenberg Niels. A Retrospective on Region-Based Memory Management. Higher Order Symbol. Comput.. 2004;17(3):245–265.

67. Tofte Mads, Birkedal Lars. A Region Inference Algorithm. ACM Trans. Program. Lang. Syst.. 1998;20(4):724–767.

68. Tofte Mads, Talpin Jean-Pierre. Region-Based Memory Management. Inf. Comput.. 1997;132(2):109–176.

69. Hallenberg Niels, Elsman Martin, Tofte Mads. Combining Region Inference and Garbage Collection. In: PLDI ’02:141–152ACM; 2002; New York, NY, USA.

70. Tofte Mads, Birkedal Lars, Elsman Martin, Hallenberg Niels, Olesen Tommy Højfeld, Sestoft Peter. Programming with Regions in the MLKit (Revised for Version 4.3.0). : IT University of Copenhagen, Denmark; 2006.

71. Penner Jeremy. Undergraduate Honours Project: Provably Safe Real-Time Programming . www.sporktania.com/Real-time%20SML.doc.

72. MLton . <http://www.mlton.org>

73. Timber: A Gentle Introduction . <http://www.timber-lang.org/index_gentle.html>

74. Leon Sterling, Ehud Shapiro The Art of Prolog Advanced Programming Techniques, Second Edition – The MIT Press, Cambridge, Massachusetts, London, England, 1999

75. Byrd, William E. Indiana University. ProQuest Dissertations Publishing, 2009. 3380156