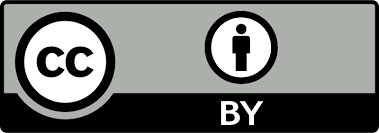
**DOI: 10.15514/ISPRAS-2019-1(2)-1**

**Строго типизированное встраивание реляционного языка программирования в OCaml**

1 Д.C. Косарев, ORCID: 0000-0002-6773-5322 <d.kosarev@spbu.ru>

1Д.Ю. Булычев, ORCID: 0000-0001-8363-7143 <dboulytchev@math.spbu.ru>

1Санкт-Петербургский государственный университет,

Россия, 198504, Санкт-Петербург, Университетский пр., д. 28.

Аннотация. В работе представлена реализация реляционного языка программирования miniKanRen, встроенного в функциональный язык OCaml. Отличительная особенность описываемой реализации — использование *полиморфной унификации*, которая может использоваться для унификации произвольных типов данных. Также представлен систематический подход к разработке реляционных программ и интеграции их с функциональным кодом.

Ключевые слова: проектирование интерфейсов, синтез программ, программирование в ограничениях, реляционное программирование, miniKanren.

Для цитирования: Косарев Д.С., Булычев Д.Ю. Строго типизированное встраивание реляционного языка программирования в OCaml. Труды ИСП РАН, том 1, вып. 2, 2019 г., стр. 15–19. DOI: 10.15514/ISPRAS–2019–1(2)–1.

Declarative GUI Layout Synthesis with Relational Constraint

Solvers

1D.S. Kosarev ORCID: 0000-0002-6773-5322 <d.kosarev@spbu.ru>

1P.A. Lozov ORCID: 0000-0003-3563-2828 <lozov.peter@gmail.com>

1D.Yu. Boulytchev ORCID: 0000-0001-8363-7143 <dboulytchev@math.spbu.ru>

2D.S. Fokin <denis.fokin@gmail.com>

1St. Petersburg State University,

Universitetski pr., 28, St. Petersburg, 198504, Russia.

2No affiliation.

Abstract. We describe a system which, given a set of designer-specified layout constraints (guidelines) and a description of graphic user interface (GUI) logical structure generates a set of concrete layouts, each of which complies with these guidelines by construction. We give a formal treatment of the task as a constraint satisfiability problem and describe the construction of a sound and complete solver based on the utilization of relational verifier-to-solver approach. We also describe a number of refinements which make the solver more efficient and applicable.

**Keywords:** interface design, program synthesis, constraint programming, relational programming, miniKanren.

**For citation:** Kosarev D.S., Lozov P.A., Boulytchev D.Yu., Fokin D.S. Declarative GUI Layout Synthesis with Relational Constraint Solvers. *Trudy ISP RAN/Proc. ISP RAS*, vol. 1, issue 2, 2019. pp. 15-19 (in Russian). DOI: 10.15514/ISPRAS-2019-1(2)-1.

1. Введение

Реляционное программирование [1] — это привлекательный подход, основанный на представлении программ как отношений. В результате реляционные программы можно “запускать” в различных “направлениях”, что позволяет, например, симулировать обратимые вычисления. Это интересно не только с теоретической точки зрения, но обладает и практической ценностью: некоторые задачи выглядят проще, если их рассматривать как запросы к некоторой реляционной спецификации [2]. Можно назвать некоторое количество примеров, которые подтверждают это наблюдение: алгоритм проверки типов для просто типизированного *𝜆*-исчисления (и в то же время алгоритм вывода типов и проверки населенности типов); интерпретатор, способный синтезировать “квайны” — программы, вычисляющиеся в самих себя; сортировка списков, способная породить все перестановки, т.п.

Многие языки логического программирования (например, Prolog, Mercury [3] или Curry [4]) в некоторой степени могут считаться реляционными. Нами был выбран miniKanRen[[1]](#footnote-1) как модельный язык, потому что он был специально спроектирован как предметно-ориентированный язык, встраиваемый в Scheme/Racket. Он довольно минималистичен — может быть реализован с помощью малого количества структур данных и комбинаторов [5, 6] — и поэтому можно найти реализации, где его уже реализовали для других языков программирования общего назначения, включая Scala, Haskell и Standard ML. Парадигму в основе miniKanren можно описать как “легковесное логическое программирование”[[2]](#footnote-2).

В данной работе исследуется задача встраивания miniKanren в OCaml[[3]](#footnote-3) — статически

типизированный функциональный язык с богатой системой типов. Статическая типизация

приносит ряд улучшений. Во-первых, типизация даёт некоторый уровень корректности,

отсеивая патологические программы, выдающие патологические результаты. В контексте

реляционного программирования типизация также помогает интерпретировать результаты

запросов. Часто ответы на реляционные запросы содержат свободные переменные, которые

можно заменять на произвольные значения. В типизированном случае эти переменные

становятся типизированными, что упрощает понимание ответов, особенно с большим количеством свободных переменных. Во-вторых, некоторые программы на miniKanren требуют реализации и использования специальных ограничений. В слабо типизированном случае, где всё может быть чем угодно, некоторые символы и структуры данных могут “просачиваться” в нежелательные места [7]. Чтобы это предотвращать в miniKanren для Scheme были добавлены дополнительные ограничения (“absentо”, “symbolо”). Эти ограничения важны для слабых динамических систем типов, так как они отсекают ответы во время исполнения. В статически типизированных языках эта работа может быть переложена на систему типов (при условии, что программист реализует их правильно), что не только улучшит производительность, но и уменьшит количество примитивов miniKanren, которые требуется реализовать.

В работе представлена реализация[[4]](#footnote-4) реляционных комбинаторов и синтаксических расширений для языка OCaml под названием OCanren, которая в том числе поддерживает ограничения-неравенства [8] и оптимизации faster-miniKanren[[5]](#footnote-5), которые применяются в наиболее полных реализациях. Новизна заключается в следующем:

* Реализованное встраивание позволяет программисту использовать статическую типизацию и вывод типов в реляционных спецификациях. В частности, ошибки типизации находятся во время компиляции и типы логических переменных выводятся из контекста.
* Реализация основана на полиморфной унификации, которая также как и полиморфное сравнение языка OCaml, может использоваться для произвольных типов. Реализация полиморфной унификации использует небезопасные возможности компилятора OCaml и полагается на специфичные особенности представления значений в памяти. При этом доказано, что корректность типов не нарушается.
* Описан масштабируемый подход (глава 9) к использованию типов при написании реляционных спецификаций, которые позволяет конвертировать данные из реляционного домена и в него. Подход основан на обобщённом программировании и позволяет более кратко и единообразно писать реляционные спецификации.
* Представлен упрощенный способ интеграции реляционного и функционального кода. Он основан на известном методе [9, 10] реализации поливариадических функций, и позволяет скрыть процессе реификации (глава 7) от программиста-пользователя.

В разделе 2 представлен обзор. Затем мы кратко описываем основные конструкции miniKanren в его оригинальном виде и вводим некоторые понятия реляционного программирования. Изложения языка со всеми деталями не предполагается, интересующиеся читатели могут обратиться к книге [1] по реляционному программированию. В разделе 4 описываются необходимые конструкции для реализации реляционного языка, в это раз в контексте OCaml. В разделе 5 вводится полиморфная унификация, а также показано, что этот алгоритм унификации с использованием треугольной подстановки не нарушает корректности типов. Затем представлены возможности по поддержке пользовательских типов данных путём их инъекции в логический домен, а также шаблон обобщённого программирования, который позволяет реализовывать преобразования в логический домен и обратно. Также представлен (раздел 6) более сложный подход, где данные представляются без тегирования ради эффективности. В разделе 7 описаны конструкции верхнего уровня и обсуждается задача интеграции функционального и реляционного кода. Затем представлены несколько примеров (раздел 9) написания реляционных программ, с помощью нашей библиотеки. В разделе 10 приведены результаты оценки производительности нашего подхода и реализаций miniKanren на языке Scheme.

2. Обзор

При реализации miniKanren в строго статически типизированном языке естественным образом возникают сложности, так как реляционное программирование родилось в среде Scheme/Racket — динамически типизированной и удобной для метапрограммирования среде. Оригинальная реализация miniKanren не обращает внимания на особенности, важные для таких языков как ML и Haskell. Например, унификация, одна из основных конструкций miniKanren, обязана исполняться для разных данных, и одного только параметрического полиморфизма может быть недостаточно.

Существуют несколько способов преодолеть эту проблему. Во-первых, можно смоделировать нетипизированный подход и предоставить реализацию унификации для некоторого конкретного типа, достаточно богатого, чтобы представить все другие типы данных. Некоторые библиотеки[[6]](#footnote-6) реляционного программирования для Haskell и предыдущая[[7]](#footnote-7) реализация для OCaml пошли по этому пути. В результате оригинальная реализация на Scheme может быть повторена со всей её краткостью, но реляционные спецификации станут слабо типизированы. Сходный подход встречался ранее при встраивании Prolog в Haskell [11].

Другим подходом является использование *ad hoc* полиморфизма и написание специфичной

унификации для каждого из интересующих типов. Некоторые реализации miniKanren, например Molog[[8]](#footnote-8) and MiniKanrenT[[9]](#footnote-9) для Haskell, выбрали данный подход. Строгая типизация сохраняется, но требуется написание большого количества стереотипного (англ. boilerplate) кода; часто применяется некоторая оптимизация этого, например с помощью Template Haskell [12]. Можно определять [13] отдельные классы типов для выполнения и унификации, и выделения свободных логических переменных. Для нас выглядит несколько искусственно требование выделять логические переменные в пользовательских типах данных особым способом, так как эти логические переменные придется обрабатывать и в модулях программы, не имеющих отношения к логическому программированию.

Существует ещё один подход, но нам неизвестны реализации miniKanren на его основе. Можно реализовать унификацию для обобщённого представления типов данных в виде сумм произведений и неподвижных точек функторов для них [14, 15]. Унификация сможет работать для любого типа данных, для которого сделано представление. Мы полагаем, что

с таким подходом будет меньше стереотипного кода.

Из выше сказанного можно заключить, что типизированное встраивание miniKanren в

OCaml состоит из сочетания обобщённого программирования [16] и *ad hoc* полиморфизма.

Использования обобщённого программирования в OCaml уже довольно хорошо исследовано [17]. Возможности для *ad hoc* полиморфизма в OCaml развиты слабо: нет ничего сравнимого с классами типов языка Haskell, использование объектно-ориентированного программирования позволяет эмулировать желаемое поведение, но не без недостатков. Существующие предложения по развитию *ad hoc* полиморфизма (например, неявные модули [18]) требуют модификации компилятора, чего мы хотели бы избежать. Поэтому мы пошли другим путём и реализовали полиморфную унификацию один раз и для всех логических типов, и эта реализация существенно использует особенности компилятора OCaml, так как возможности для менее *ad hoc* подхода пока не интегрированы в язык. Для работы с пользовательскими типами данных в реляционных частях программ предлагается использовать специальные логические представления (глава 6), которые освобождают пользователя от задачи поддержки переменных в его типах данных. Использование же обобщённого программирования позволяет систематическим образом получать преобразования в логические представления и обратно.

3. miniKanren — краткое изложение

В этом разделе miniKanren будет кратко описан в его оригинальном виде на основе канонических примеров. Предметно-ориентированный язык организован как набор комбинаторов и макросов для Scheme/Racket, которые должны описывать поиск решения для некоторой “цели” (англ. goal). Ядро языка состоит из четырех реляционных конструкции для создания целей:

* Ограничение синтаксической унификации [19] в форме (== *𝑡*1 *𝑡*2), где *𝑡*1, *𝑡*2 — это некоторые “термы”. Унификация является основой для проведения поиска: если существует унификатор двух термов, то цель считается выполненной, наиболее общий унификатор сохраняется как частичное решение, и поиск в данной ветви дерева поиска.
* Ограничения-неравенства [8] в форме (≠ *𝑡*1 *𝑡*2), где *𝑡*1, *𝑡*2 — это некоторые “термы”. Данное ограничения запрещает поиск в поддеревьях, где данные темы равны с учетом соответствующей подстановки.
* Условная конструкция в форме  
  conde, где каждый *𝑔𝑖 𝑗* является некоторой целью. Построенная с использованием conde цель, рассматривает коллекцию подцелей в квадратных скобках как неявную конъюнкцию (т.е. [𝑔𝑖1 𝑔𝑖2 … 𝑔𝑖𝑘] считается конъюнкцией всех gij) и пытается решать их независимо, т.е. conde работает как дизъюнкция.

(conde

**[**g11 g12 … g1k1 **]**

**[**g21 g22 … g2k2 **]**

…

**[**gn1 gn2 … gnkn **]**)

* Создание новых (англ. fresh) логических переменных осуществляется с помощью Конструкции fresh, где каждый 𝑔𝑖является некоторой целью. Создаются свежие логические переменные x1 x2 … xk и осуществляется поиск цели, являющейся конъюнкцией целей g1 … gn, внутри которых могут использоваться только что введенные логические переменные.

(fresh (x1 x2 … xk )

g1

…

gn)

В качестве примера рассмотрим отношение appendoдля конкатенации списков (по традиции

реляционные объекты именуются с суффиксомo).

(define (append𝑜x y xy)

(conde

[(== '() x) (== y xy)]

[(fresh (h t ty)

(== `(,h . ,t) x)

(== `(,h . ,ty) xy)

(append𝑜t y ty))]))

Рис. 6. Реляционная реализация сложения чисел Пеано  
Pic. 6. Relational implementation of Peano numbers addition

Отношение “appendox y xy” интерпретируется как “конкатенация x и y даёт xy”. Если

список x пуст (строка 3), то, вне зависимости от содержимого y, чтобы отношение выполнялось значение xy должно быть таким же как y. Иначе, x нужно разделить на голову h и хвост t, для этого понадобятся введённые свежие переменные. Также понадобится дополнительная переменная ty, чтобы хранить список, находящий в отношении appendoс t и y. Дополнив тривиальными соображениями (строки 5–7), получим полную реализацию отношения.

Цели, полученные с помощью упомянутых выше конструкций, можно запускать с помощью

примитива run:

run 𝑛 (𝑞1 . . . 𝑞𝑘 ) 𝐺

Здесь 𝑛— это число желаемых ответов (для всех ответов используется “\*”); 𝑞𝑖 — свежие

искомые переменные; а 𝐺— цель, использующая эти переменные.

Конструкция run осуществляет поиск ответов для данной цели, и возвращает потенциально

бесконечный список ответов, состоящий из значений искомых переменных, которые хранят

ответы на данный запрос. Например

run 1 (q) (appendoq '(3 4) '(1 2 3 4) )

возвращает список ((1 2)), который хранит одно значение искомой переменной*𝑞*. Процесс

реконструирования ответов из внутреннего представления называет *реификацией[[10]](#footnote-10)* [2].

4. Потоки, состояния и цели

Этот раздел содержит некоторые детали реализации. Несмотря на то, что это почти повторение оригинальной реализации [5, 8] для OCaml, он нужен, чтобы аккуратно ввести некоторые понятия.  
Процедура поиска реализована с использованием монады для ленивых потоков с возвратами [20]:

type 𝛼 stream  
val mplus: 𝛼 stream -> 𝛼 stream -> 𝛼 stream  
val bind: 𝛼 stream -> (𝛼 -> 𝛽 stream) -> 𝛽 stream

Важной компонентой является реализация следующих типов данных:

**type** env = *. . .*

**type** subst = *. . .*

**type** constraints = *. . .*

**type** state = env \* subst \* constraint

Тип состояния state описывает позицию в лениво строящемся дереве поиска: тип env

соответствует *окружению*, содержащему дополнительную информацию (в частности, необходимую для создания свежих переменных); subst хранит подстановку, которая содержит отображения некоторых логических переменных; тип constraints представляет ограничения-неравенства, которые надо соблюдать (раздел 8). В самом простом случае env содержит только номер последней введённой свежей переменной, subst — это конечное отображение, а constraints — просто список подстановок.

Тип целей goal является преобразованием одного состояния в ленивый поток состояний:

**type** goal = state -> state stream

В терминах поиска выполнение цели соответствует одному шагу поиска: для конкретного

узла дерева поиска вычисляются его непосредственные потомки. Со стороны пользователя

тип goal является абстрактным и все состояния полностью скрыты.

Также присутствует набор заранее реализованных комбинаторов:

**val** (&&&) : goal -> goal -> goal

**val** (|||) : goal -> goal -> goal

**val** call\_fresh : (*𝑣* -> goal) -> goal

…

Конъюнкция “&&&” объединяет результаты своих целей-аргументов с помощью bind,

дизъюнкция “|||” конкатенирует результаты с помощью mplus, примитив call\_fresh принимает абстрагированную цель и применяет её к только что созданной свежей переменной. Тип *𝑣* соответствует типу свежей логической переменной, детали которого мы обсудим позже (раздел 6). Эти комбинаторы являются базовыми для реализации более удобных конструкций реляционного программирования (conde, fresh и т.д.)

Наконец, существуют два примитива для создания примитивных целей: ограничения унификации и ограничения-неравенства. Тип термов 𝑡 также оставим абстрактным и вернёмся к нему позже.

**val** (===) : 𝑡 -> 𝑡 -> goal

**val** (=/=) : 𝑡 -> 𝑡 -> goal

В различных реализациях miniKanren все конструкции можно реализовывать сходным образом. Но в случае с OCaml реализация полиморфной унификации (раздел 5) потребовала использования нетривиальных решений, которые отсутствуют в оригинальном miniKanren.

С использованием объявленных выше типов, сигнатура примитива run выглядит так:

**val** run : goal -> state stream

Данная функция создает начальное состояние и применяет к нему цель. Состояния в получившемся потоке описывают различные решения данной цели. Так как поток строится постепенно, для проведения поиска следуют получать состояния по одному.

Конкретные ответы реконструируются из состояний. Как правило, несколько переменных реифицируются в этом состоянии, т.е. оттуда достаются их конкретные значения. Такжедля свободных переменных извлекаются ограничения-неравенства, представленные как список “запрещённых” термов. Так как эти запрещённые термы могут в свою очередь содержать другие свободные переменные, то реификация ограничений осуществляется рекурсивно.

В случае OCaml реификация является нетривиальной частью реализации, так как, как мы увидим, она не может быть реализована на типобезопасном фрагменте языка.

5. Полиморфная унификация

Довольно естественно попытаться реализовать полиморфную унификацию в языке, оснащенном полиморфным сравнением — удобной, но порою спорной функциональностью[[11]](#footnote-11). Как и полиморфное сравнение, полиморфная унификация производит обход значений, используя знания о представлении значений во время выполнения программы. Неоспоримым преимуществом данного подхода является то, что, во-первых, для использования унификации программисту не придется писать стереотипного кода, а во-вторых, то, что данный подход позволяет получить наиболее эффективную реализацию. С другой стороны, недостатки полиморфного сравнения также наследуются, в частности унификация будет зависать на циклических структурах данных и не работать со значениями-функциями. Так как мы не ожидаем никакого разумного поведения в этих случаях, основной проблемой останется то, что компилятор не будет способен предупредить об использовании этих крайних случаев. Другим недостатком является то, что реализацию придется изменять каждый раз, когда представление данных в компиляторе поменяется. Также полиморфная унификация реализована в слабо типизированной манере с использование модуля Obj, и её необходимо аккуратно тестировать.

Важным различием между полиморфным сравнением и унификацией является то, что сравнение только изучает свои операнды, а унификация постепенно конструирует подстановку-результат. Подстановка отображает переменные в термы и используется далее для реификации ответов. Итого, нам необходимо показать, что в результате не конструируются неправильно типизированные значения. Может показаться, что это свойство будет соблюдаться естественным образом, так как сутью синтаксической унификации является проверка того, что некоторые объекты считаются равными. Тем не менее существуют различные системы типов и различные реализации унификации, к тому же иногда *одинаковые значения* могут быть типизированы *по-разному*, поэтому ниже будет представлено обоснование корректности в абстрактном случае, а его части будут переиспользованы для конкретных.

Для начала рассмотрим три алфавита:

* 𝜏, . . .— типы;
* 𝑥𝜏 , . . *.* — типизированные логические переменные;
* 𝐶𝑘 𝜏 *(𝑘 ⩾ 0), . . .* — конструкторы типа 𝜏.

Множество всех правильно построенных типов задается индуктивно:

𝑡𝜏 = 𝑥𝜏 | 𝐶𝑘 𝜏 (𝑡𝜏1 , 𝑡𝜏2 , . . . , 𝑡𝜏𝑘 ),

подразумевая, что для любого конструктора и для всех его вхождений его аргументы имеют соответствующие типы.

Мы считаем, что всем термам во время компиляции соответствует некоторый тип. Воспользуется этим свойством чтобы реализовать алгоритм унификации на OCaml, используя некоторые представления для термов и типов:

**val** unify : term -> term -> subst option -> subst option,

где “term” соответствует типу представления типизированных термов, а “subst” — это

подставнока (частичное отображение переменных в термы). Унификация может закончиться неуспешно (поэтому в типе результат присутствует “option”), выполняется в контексте некоторой подстановки (поэтому “subst” в третьем аргументе) и может запускаться по цепочке (для этого “option” в третьем аргументе).

Используется тот же самый алгоритм унификации с треугольной подстановкой, что и в

минимальной реализации [5]. Также опущены некоторые не очень важные детали (такие

как “occurs check”), но необходимые в настоящей реализации, а также воздержимся от

обсуждения алгоритма самого по себе: превосходное описание [21] вместе с доказательством корректности можно найти в источниках.

Следующий псевдокод демонстрирует схему реализации:

**let rec** unify 𝑡𝜏1 𝑡𝜏2 𝑠𝑢𝑏𝑠𝑡 =

**let rec** walk 𝑠 𝑡𝜏 =

**match** 𝑡𝜏 **with**

| 𝑥𝜏 **when** 𝑥𝜏 ∈ 𝑑𝑜𝑚(𝑠) -> walk 𝑠 (𝑠 𝑥𝜏)

| \_ -> 𝑡𝜏

**in**

**match** 𝑠𝑢𝑏𝑠𝑡 **with**

| None -> None

| Some 𝑠 ->

**match** walk 𝑠 𝑡𝜏1, walk 𝑠 𝑡𝜏2 **with**

| 𝑥𝜏1 , 𝑥𝜏2 when 𝑥𝜏1 = 𝑥𝜏2 -> 𝑠𝑢𝑏𝑠𝑡

| 𝑥𝜏1 , 𝑞𝜏2 -> Some (𝑠 [𝑥𝜏1 ←𝑞𝜏 2])

| 𝑞𝜏 1, 𝑥𝜏2 -> Some (𝑠 [𝑥𝜏2←𝑞𝜏1])

| 𝐶𝜏 (𝑡𝜏11,… , 𝑡𝜏𝑘𝑘), 𝐶𝜏 (𝑝𝜏11, …, 𝑝𝜏𝑘𝑘) ->

unify 𝑡𝜏𝑘𝑘 𝑝𝜏𝑘 k (... (unify 𝑡𝜏11 𝑝𝜏11 𝑠𝑢𝑏𝑠𝑡 ) ...)

| \_, \_ -> None

Напоминаем читателю, что в верхних индексах указаны типы, которые мы считаем частью

значений. Например, в строке 1 имеется в виду, что мы запускаем unify от термов 𝑡1 и

𝑡2, у которых один и тот же тип *𝜏*. Предположим, что самый первый запуск унификации

происходит от термов одного типа, и что любая подстановка может быть получена только

с помощью одной или нескольких унификаций.

Покажем, что в этих предположения все атрибуты с типами не нужны: они не влияют на

исполнение unify и могут быть стёрты. Единственное место, где невозможно предъявить

атрибут явно — это строка 4, где с помощью функции walk возвращается результат применения подстановки. Можно показать по индукции, что любая подстановка обладает следующим свойством: если в подстановке *𝑠* определено значение переменной *𝑥𝜏* , то это значение имеет тип *𝜏*, и, следовательно, walk *𝑠 𝑡𝜏* всегда возвращает значения типа *𝜏*.

Очевидно, что это свойство выполняется для пустых подстановок. Рассмотрим произвольную подстановку *𝑠*, для которой это свойство выполнено. В 10й строке совершаются два вызова — walk 𝑠 𝑡𝜏 1 и walk 𝑠 𝑡𝜏 2 — и разбираются их результаты. По индукционному предположению у них тип *𝜏* и производить сопоставление с образцом в строке 14 можно. В сроке 11 подстановка возвращается без изменений, в строках 12 и 13 — изменяется с сохранением свойства. Наконец, в 15й строке осуществляется несколько вызовов unify, но все они запускаются на термах одинакового типа, а значит, что подстановка обладает интересующим нас свойством. Индукция

по структуре терма завершает доказательство.

Итак, атрибуты с типами не важны — они никогда не анализируются и не ограничивают

сопоставления с образцом, а значит могут быть полностью стёрты. Также стоит обратить

внимание, что представлением термов может служить обычное представление среды исполнения OCaml. Но это нельзя сделать естественным способом, придется воспользоваться низкоуровневым интерфейсом модуля Obj[[12]](#footnote-12). Также понадобится некоторый способ отличать вхождения логических переменных в терм, в оригинальной реализации эта проблема решалась с помощью договоренности — роскошь, которую мы не может себе позволить. Мы вернемся к этому вопросу в следующем разделе.

Реализация унификации называется *полиморфной*, потому что на верхнем уровне ей

можно приписать тип

**val** unify : *𝛼* -> *𝛼* -> subst option -> subst option

Тип подстановки не является полиморфным, что означает, что компилятор полностью

теряет типы значений, хранящихся в подстановке. Они восстанавливаются во время реификации ответов (раздел 7). Вне унификации компилятор поддерживает типизацию, что означает что все термы, подтермы и логические переменные согласуются по типам во всех контекстах.

6. Представление термов и их погружение в логический домен

Полиморфная унификация из предыдущего раздела работает для значений произвольных типов, при условии, что можно отличать логические переменные от других термов. Это предположение зависит от вида представления термов. В оригинальной реализации на языке Scheme все термы были обычными s-выражениями, а логические переменные (в простом случае) — одноэлементными векторами. Пользователь обязан выполнять это соглашение и воздерживаться от работы с пользовательскими значениями, состоящими из векторов.

В нашем случае мы хотим сохранить и строгую статическую типизацию и автоматический вывод типов. Так как была выбрана полиморфная унификация, нежелательно представлять логические значения различных типов по-разному (технически это возможно, но будет противоречить выбранному легковесному подходу). Из этого следует, что термы с логическими переменными должны быть типизированы не так, как типизируется термы, объявленные пользователем, иначе будет возможно использовать термы в контекстах, где логические переменные не отображаются должным образом. В то же время нежелательно, чтобы типы логических значений были полностью отделены от пользовательских типов: могут пригодиться конструкторы, объявленные пользователем и т.п. Эти рассуждения подводят к идее логического представления для пользовательских типов данных. Неформально, логическим представлением типа 𝑡 является тип 𝜌𝑡, для которого определена пара функций:

↑ : 𝑡 → 𝜌𝑡 − инъекция

↓ : 𝜌𝑡 →𝑡 − проекция.

Тип 𝜌𝑡 структурно повторяет 𝑡, но может содержать логические переменные. Также инъекция — это тотальная функция, а проекция — частичная.

Важно спроектировать представление значений следуя некоторой обобщённой схеме, иначе комбинаторы miniKanren будет невозможно правильно протипизировать. Также желательно представить обобщённый способ получения пар инъекции/проекции (а ещё лучше, автоматизировать), чтобы снять с программиста груз ответственности за правильную их реализацию. Наконец, представление должно позволять надежным образом отличать логические переменные от других значений.

В этом разделе рассмотрен два способа реализовать логические представленя. Первый

достаточно просто реализовать, но к сожалению реализация будет показывать низкую

производительность на практически значимых примерах. Чтобы исправить этот недостаток,

был разработан другой способ представления значений, который переиспользует некоторые

компоненты первого. В разделе 10 представлены результаты экспериментального сравнения

производительности.

6.1. Тегированные логические значения

Наиболее естественным решением будет тегированное логическое представление. Вводится

полиморфный тип [*𝛼*][[13]](#footnote-13), который соответствует логическому представлению типа *𝛼*.

**type** [*𝛼*] = Var **of** int | Value **of** *𝛼*

Неформально выражаясь, любое значение типа [*𝛼*] является либо значением типа *𝛼*, либо

свободной логической переменной. Обращаем внимание, что конструкторы этого типа не

используются пользователем напрямую, так как единственным способом создания переменных

является конструкция “fresh”, т.е. тип логических значений является абстрактным. Используя

этот тип можно определить сигнатуры примитивов унификации, неравенства и создания

переменных следующим образом:

**val** (===) : [*𝛼*] -> [*𝛼*] -> goal

**val** (=/=) : [*𝛼*] -> [*𝛼*] -> goal

**val** call\_fresh : ([*𝛼*] -> goal) -> goal

И ограничение унификации, и неравенства использует одну и ту же полиморфную унификацию, но их тип ограничен только логическими значениями.

Кроме переменных, другие логические значения также могут быть получены с помощью

инъекции. И наоборот, иногда логические значения могут быть спроецированы в обычные.

Для этого предоставляются два базовых примитива[[14]](#footnote-14), которые могут быть использованы

для *поверхностной* инъекции/проекции. Как и ожидается, инъекция тотальная, а проекция

— частичная функция.

**val** (↑∀): *𝛼* -> [*𝛼*]

**val** (↓∀): [*𝛼*] -> *𝛼*

**let** (↑∀) x = Value x

**let** (↓∀) = **function** Value x -> x | \_ -> failwith "not a value"

…

Пара поверхностных функций хорошо работает для примитивных типов, но чтобы реализовать инъекцию/проекцию для произвольных, необходимо воспользоваться идеей представления типов в виде неподвижных точек функторов [15]. Для наших нужд желательно сделать функторы полностью полиморфными: такими типами, где логические значения можно писать в произвольных позициях. К тому же данный подход позволит переиспользовать имеющийся код с меньшим количеством изменений.

Продемонстрируем этот подход на каноническом примере связных списков. Рассмотрим

обычный тип данных OCaml для полиморфных списков:

**type** *𝛼* list = Nil | Cons **of** *𝛼* \* *𝛼* list

В этом типе можно использовать логические переменные только на месте элементов

списка, но не вместо “хвоста”, так как там ожидается значение типа *𝛼* list, зафиксированного

в конструкторе Cons. Чтобы получить подходящее логическое представление, вначале абстрагируем тип полностью полиморфного функтора:

**type** (*𝛼*, *𝛽*) L = Nil | Cons **of** *𝛼* \* *𝛽*

Таким образом, оригинальный тип может быть выражен как

**type** *𝛼* list = (*𝛼*, *𝛼* list) L,

а его логическое представление как

**type** *𝛼* list*𝑜* = [([*𝛼*], *𝛼* list*𝑜* ) L]

Более того, с помощью специфичной для функтора функции fmap

**val** fmapL : (*𝛼* -> *𝛼*′) -> (*𝛽* -> *𝛽*′) -> (*𝛼*, *𝛽*) L -> (*𝛼*′, *𝛽*′) L

можно реализовать инъекцию и проекцию:

**let rec** ↑list l = ↑∀(fmapL (↑∀) ↑list l)

**let rec** ↓list l = fmapL (↓∀) ↓list (↓∀ l)

Специфичные для функторов функции fmap могут быть легко написаны, и даже получены

автоматически с помощью различных методов обобщённого программирования для OCaml.

А с помощью fmap уже будут написаны инъекции и проекции пользовательских типов

данных.

Теперь вернемся к вопросу определения переменных во время полиморфной унификации.

Так как типы не известны, мы не можем отличать переменные с помощью простого сопоставления с образцом. В реализации переменные проверяются следующим способом:

* в окружении хранится дополнительное значение-“якорь”, создаваемое вместе с начальным состоянием; этот “якорь” не изменяется для всех производных состояний во время запуска run;
* в тип логических переменных добавлен “якорь”, который при берется из окружения и при создании переменных кладется в них.  
  **type** [*𝛼*] = Var **of** int \* anchor | Value **of** *𝛼*
* во время унификации, для проверки переменных проверяется конъюнкция следующих свойств:
  + рассматриваемое значение является boxed;
  + тег и раскладка в памяти соответствует переменным, т.е. значениям, построенным с помощью конструктора Var из типа [*𝛼*];
  + адрес якоря соответствует адресу якоря в текущем окружении.

Учитывая, что состояние абстрактно для пользователя, можно гарантировать, что только

переменные, созданные в текущем запуске примитива run, пройдут тест на переменные,

так как указатель на якорь уникален среди всех указателей и никак может попасть во вне

из-за использования примитива создания переменных.

Осталось описать фазу реификации, которая может быть представлена с помощью следующей функции:

**val** reify : state -> [*𝛼*] -> [*𝛼*]

Функция принимает состояние и логическое значение, и рекурсивно совершает подстановку

всех переменных в логическом значении до тех пор, пока остаются связанные переменные.

Так как наша реализация подстановки не является полиморфной, то функция reify также

реализована в небезопасном стиле. Но легко показать, что reify не создает неправильно

типизированных термов. Все термы в подстановке корректно типизированы, подстановка

осуществляет замену переменных на термы такого же типа, а унификация не изменяет

типы унифицируемых термов. Следовательно, reify всегда подставляет в правильно типизированном терме подтермы на термы таких же типов, что обуславливает безопасность системы типов.

Кроме осуществления подстановки функция reify также реифицирует ограничения-

неравенства. При этом к каждой свободной переменной приписывается список реифицированных термов, представляющий ограничения-неравенства для этой свободной переменной. Заметим, что ограничения неравенства могут быть созданы только для одинаково типизированных термов, что обосновывает типовую безопасность реификации. Обратим также внимание, что нужны дополнительные проверки для избежания зацикливания реификации, так как реификация ответов и ограничений взаимно рекурсивны, а реификация ответа может вызвать саму себя по цепочке ограничений-неравенств. В примере ниже

**let** foo q =

fresh (r s)

(q === ↑∀ (**Some** r)) &&&

(r =/= s) &&&

(s =/= r)

…

ответ для переменной *𝑞* будет содержать неравенства для переменной *𝑟* ; реификация *𝑟*

приведет ограничения с переменной *𝑠*, которая снова запустит реификацию переменной

*𝑟* , и т.д.

6.1. Логические переменные без тегирования. Отслеживание типов

Решение, предложенное выше, страдает существенным недостатком: чтобы произвести

унификацию, необходимо погрузить термы в логический домен, что сделает их в два раза

больше. В результате, эта реализация проиграет по производительности оригинальной.

Было разработано другое представление без этого недостатка. Идея в том, чтобы хранить

данные (не переменные) в памяти без использования тегирования вообще.

**type** [*𝛼*] = *𝛼*

Как следствие станет невозможно конструировать данные с переменными внутри путём использования конструкторов, а придется использовать специальные функции-конструкторы. В нашей реализации тип [*𝛼*] абстрактный, поэтому эти модификации не меняют интерфейса взаимодействия с данными из реляционного кода. Но при этом всё ещё возможно представлять переменные старым способом, переиспользовав алгоритм проверки на переменные, а реализация полиморфной унификации остается *почти* такой же. Некоторой проблемой является то, что можно вводить переменные в нелогические и нетегированные данные.

Эти свободные переменные не помещают унификации (она сумеет их отличить), но их

нельзя оставить для нелогического домена как они есть.

Идея заключается в том, чтобы использовать в общем случае некорректное представление

в памяти — ему нельзя придать тип в языке OCaml — для внутренних нужд, а применять

тегирование только на фазе реификации. Но тогда необходимо решить каким будет тип

функции reify. Прямолинейное решение **val** reify : state -> [*𝛼*] -> [*𝛼*] не годится, так как тип аргумента совпадает с типом результата [*𝛼*]. Хотелось бы что-то похожее на

**val** reify : state -> [*𝛼*] -> (``tagged'' [*𝛼*]).

Если *𝛼* — тип без параметров, то можно затегировать результат конструкторами Var

или Value в зависимости того переменная это или нет. Всё несколько сложнее для типов

с параметрами, например, для типа целочисленных списков [ [int] list]. Здесь надо не

только тегировать не верхнем уровне, но и запускать реификацию с тегированием рекурсивно

для подвыражений внутри. Итого, необходим следующий (мета)тип для функции реификации, где *𝛽* — тип затегированного *𝛼*:

**val** reify : state -> [*𝛼*] -> (``tagged''[*𝛽*]).

Эти наблюдения обосновывают выбранную конкретную реализацию.

Оригинальное определение типа [*𝛼*] будет тегированным представлением. Введем дополнительный однопараметрический тип “*𝛼* ilogic”, представляющий логические значения типа *𝛼*.

**type** *𝛼* ilogic = *𝛼*

**val** inj : *𝛼* -> *𝛼* ilogic

Для пользователя он будет абстрактным, а его значения будут появляться с помощью функции поверхностной инъекции inj. Для рекурсивных типов всё несколько сложнее: необходимо применять инъекцию рекурсивно для всех подвыражений. Рассмотрим эту инъекцию на примере списков.

**type** *𝛼* list = (*𝛼*, *𝛼* list) L,

**type** *𝛼* injected = (*𝛼*, *𝛼* injected) L ilogic,

**type** *𝛼* list*𝑜* = [([*𝛼*], *𝛼* list*𝑜* ) L]

В типе нетегированного представления необходимо расставить тип ilogic для всех подвыражений, так образом инъекция целочисленных списков с типом int list будет порождать значения типа int ilogic injected. На практике для построения логических представлений списков стоит использовать специальные функции-конструкторы вместо конструкторов алгебраического типа.

**let** cons h tl : 'a -> 'a injected -> 'a injected =

inj (Cons (h, tl))

**let** nil () : unit -> 'a injected = inj Nil

Также необходимо реализовать реификацию значений из логического домена. На данный

момент подход к реификации не ограничивает тип, куда реификация будет производится,

но каноническим является использование упомянутого выше [*𝛼*]. Все реификаторы являются

двупараметрическими типами, инкапсулирущими преобразование из логического домена

в некоторый кодомен. Например, для списков необходимо получить реификатор с типом

(int ilogic injected, [int] list𝑜) reifier.

Вот пример реификатора для списков, который можно построить, предъявив реификатор

ra для элементов списка. Сами реификаторы по сути являются монадами Reader, которые

хранят в себе информацию для определения переменных в терме. Типы реификаторов

абстрактные, чтобы пользователь не смог провести реификацию переменных, созданных

не в том состоянии. При реализации реификации используется интерфейс аппликативных

функторов с операциями pure и <\*>. Также используются комбинатороы неподвижных

точек для реификаторов (Reifier.fix) и для call-by-value языков (Reifier.zed), а ещё специальная функция chain. Примитив reify проводит поверхностную реификацию, а функция rework рекурсивно запускает реификацию для подвыражений и ограничений-неравенств, если таковые присутствуют.

**val** chain: ('a reifier -> 'b reifier) -> ('a -> 'b) reifier

**let** fmapt f g xs = pure (fmap*𝑙𝑖𝑠𝑡*) <\*> f <\*> g <\*> xs

**let** reify: (*𝛼*, *𝛽*) reifier -> (*𝛼* injected, *𝛽* list*𝑜* ) reifier =

**fun** ra ->

Reifier.fix (**fun** self ->

OCanren.reify <..> chain

(Reifier.zed

(Reifier.rework ~fv:(fmapt ra self)) ) )

Функции chain, OCanren.reify и комбинаторы неподвижной точки определены один

раз для всех типов, и пользователю для своего типа данных необходимо реализовать только

fmapt, reify и функции-конструкторы, упомянутые выше. Для облегчения этого было

разработано синтаксическое расширение, которое по объявлению типа порождает всё необходимое во время компиляции. Пример его использования можно найти в разделе 9.

7. Реификация и примитивы верхнего уровня

Чтобы облегчить пользователю представление структуры GUI и гайдлайна, мы разработали два текстовых предметно-ориентированных языка. В реализации выразительность языка гайдлайнов расширена: мы разрешаем в правилах использовать пользовательские ограничения, присутствуют конструкции для задания табличных расположений, и т.п. Ни одно из расширений существенно не изменяет подход, описанный в предыдущих главах, и все расширения могут быть учтены. Мы также параметризуем систему набором констант, которые определяют свойства окружения (например, размер холста, где мы располагаем элементы управления). Пример описания структуры для диалогового окна настроек поиска сервиса Google Drive представлена на рис. 10. Текстовое описание гайдлайна слишком длинное, чтобы представить его здесь.

ordered main\_group (

Label "Type" describes TextEdit "Any" {width=200}

Label "Owner" describes TextEdit "Anyone" {width=200}

Label "Has the words" describes

TextEdit "Enter words found in the file" {width=400}

Label "Item name" describes

TextEdit "Enter a term that matches part of the file name" {

width = 400

}

(Label "Location" describes

(TextEdit "Anywhere" { width = 200 }) dominates

ordered location\_checkboxes (

Label "In trash" describes CheckBox in\_trash\_checkbox

Label "Starred" describes CheckBox starred\_checkbox

Label "Encrypted" describes CheckBox encrypted

)

)

Label "Date modified" describes TextEdit "Any time" {

width=200

}

Label "Approvals" describes

ordered approvals\_checkboxes (

Label "Awaiting my approval" describes

CheckBox awaiting\_checkbox

Label "Requested by me" describes

CheckBox requestred\_checkbox

)

Label "Shared to" describes

TextEdit "Enter a name or email address..." {width=400}

Label "Follow-ups" describes

TextEdit "---" {width=200, height=35}

)

Button "Search" approves^ main\_group

Button "Reset" cancels^ main\_group

Рис. 10. Описание структуры диалогового окна настроек поиска из сервиса Google Drive  
Pic. 10. Google Drive search settings dialog structure description

Для оценки нашего прототипа на промышленных примерах мы также реализовали отображение элементов управления с помощью Qt/QML. На рис. 11 можно увидеть два расположения структуры диалогового окна Google Drive, синтезированного с учетом двух немного различных версий гайдлайна от JetBrains [9], нарисованные с помощью QtQuick Controls[[15]](#footnote-15) и темы оформления Material Design. Гайдлайн регламентирует, что “слишком длинный” текст должен располагаться под меткой. На правом расположении константа, регламентирующая свойство “слишком длинный” была уменьшена. Синтез занял примерно 5 секунд.

Некоторые другие примеры можно найти на рис. 12.

|  |  |
| --- | --- |
| C:\Users\lozov\AppData\Local\Microsoft\Windows\INetCache\Content.Word\pic11b.png | C:\Users\lozov\AppData\Local\Microsoft\Windows\INetCache\Content.Word\pic11a.png |
| (a) Если метка описывает “слишком длинный” элемент управления, то располагать их вертикально (a) If label describes controls which are “too long”, then place them vertically |
| (b) То же самое, но константа для свойства “слишком длинный” уменьшена (b) The same, but the constant for “too long” was decreased |

Рис. 11. Синтезированные расположения элементов для диалогового окна из Google Drive  
Pic. 11. Google Drive search settings dialog layouts

7. Обзор

Дизайн и реализация GUI — горячая тема уже много десятилетий, поэтому можно найти много инструментов, подходов, статей и отчётов по теме. Большая часть из них (если не все) предлагают декларативные и автоматические решения. Но если присмотреться, то их понятия “декларативности” и “автоматизации” отчаются от наших.

Во-первых, необходимо упомянуть некоторое количество инструментов для реализации GUI и визуализации данных, например, React [22], Jetpack Compose [23], SwiftUI [24], Streamlit [25], D3 [26] и другие. Они предоставляют пользователю наборы примитивов, которые

|  |  |
| --- | --- |
| ordered (  Label ”Short label”  describes TextEdit ”Text 1”  Label ”Loooooong label”  describes TextEdit ”Text 2”  ) |  |
| ordered (  Label ”Looooong label”  describes TextEdit ”Text 1”  Label ”Medium label”  describes TextEdit ”Text 2”  Label ”Short”  Describes TextEdit ”Text 3”  ) | C:\Users\lozov\AppData\Local\Microsoft\Windows\INetCache\Content.Word\pic12-2.png |
| ordered (  Label ”Short label”  describes TextEdit ”Text 1”  Label ”Check box label”  describes CheckBox \_  Label ”Looooong label”  describes TextEdit ”Text 2”  ) | C:\Users\lozov\AppData\Local\Microsoft\Windows\INetCache\Content.Word\pic12-3.png |
| ordered (  Label ”L 1”  describes CheckBox \_  Label ”Lab 2”  describes CheckBox \_  Label ”Label 3”  describes CheckBox \_  Label ”Label 4”  describes CheckBox \_  Label ”L 5”  describes CheckBox \_  Label ”Lab 6”  describes CheckBox \_  ) |  |

Рис. 12. Примеры структур (слева) и синтезированных расположений (справа) с учётом гайдлайна от JetBrains  
Pic. 12. Examples of structures (left) and synthesized layouts (right) w.r.t. JetBrains guidelines

позволяют отображать данные и пользовательский интерфейс. Например, Streamlit предоставляет набор встроенных примитивов отображения: “колонки”, “контейнеры”, “модальные диалоги” и т.п. [27], а также множество внешних компонент. Эти примитивы позволяют пользователям абстрагироваться от конкретного вычисления координат и относительного выравнивания. Также они предоставляют разумное поведение при изменении размера холста. Однако, выбор конкретного примитива остаётся на усмотрение разработчика, а не определяется системой. И если расположение элементов должно по какой-то причине измениться, то эти изменения нужно реализовывать вручную. В нашем случае, разработчики не задают конкретное расположение, а только логическую структуру пользовательского интерфейса. Гайдлайн определяет конкретное расположение элементов управления, в зависимости от внешних ограничений, таких как разрешение экрана или настройки региона (например, письмо справа-налево). Пока логическая структура не изменяется, не потребуется никакого взаимодействия с программистом для отображения интерфейса по-другому. С другой стороны, данные инструменты могут использоваться совместно c нашим как способ визуализации интерфейса, так как они предоставляют сходные примитивы расположения. В этой работе мы таким образом задействовали Qt/QML.

Программирование в ограничениях уже использовалось для размещения элементов управления GUI. Одним из примеров реактивного языка программирования в ограничениях является язык Wallingford [28] и система Cassowary [29]. Wallingford позволяет прикреплять ограничения различной силы к значениям в программе. Система реагирует на изменения времени и обновляет значения, не нарушая ограничения. Например, можно реализовать элемент управления с шириной равной синусу текущего времени. Система Cassowary и её потомки позволяют вычислять размеры и положения элементов динамически, например, при изменении размера холста. Она поддерживает множество ограничений, в том числе глубину по оси Z; арифметические операции (например, ширина одного элемента может быть половиной высоты другого); элементы, накладывающиеся друг на друга и т.д. Эти система предназначена для задач автоматической изменения размеров элементов при изменении общего размера холста. Также, она не предоставляет поддержки правил общего вида для корректного позиционирования элементов. Мы сомневаемся в выразимости неоднозначных расположений элементов в данных системах, например, если вертикально или горизонтальное взаимоположение определяется на основе размеров. С другой стороны, количество различных ограничений больше, чем у нас. Например, поддержка накладывающихся друг на друга элементов управления запрещена у нас с самого начала, и такие расположения не создаются вообще.

В последние годы появились методы на основе машинного обучения. Некоторые работы ставят гораздо более амбициозные цели, чем наши.

Существует направление исследования, занимающееся порождением кода UI на основе картинок [30]. По полученному от дизайнера рисунку интерфейса, инструмент распознает элементы управления и их относительное местоположение, и создает код реализации для инструмента создания GUI. Подход ортогонален и не совместим с предлагаемым нами. Он требует взаимодействия с дизайнером во время получения каждой части интерфейса, у нас же дизайнер задействован только при описании гайдлайна. Также наш подход позволяет создавать много интерфейсов автоматически. В инструменте [31] была поставлена несколько другая задача: по картинке интерфейса синтезируется его реализация в терминах примитивов Android GUI, масштабируемая между различными устройствами и без типичных ошибок. Для выполнения этой задачи из картинки извлекается элементы управления, их местоположения, а также некоторые ограничения. Эти ограничения напоминают наши примитивы расположения, но специализированы для семантики виджета ConstraintLayout [32] из Android. На основе этих ограничений и набору *свойств надёжности*, разработанному авторами, вероятностная модель, обученная на большем наборе существующих интерфейсов, порождает код. Эти реализации, более устойчивы к измерению размера и разрешения экрана, чем полученные только путём распознавания картинок. Любопытно, что авторы мотивируют свою работу, заявляя, что “одно и то же расположение должно отображаться более чем на 15000 устройствах Android с ≈100 различных плотностей пикселей; требовать от программистов разработки и поддержки всех сочетаний входа чрезвычайно нежелательно”. Именно это делает наша система за несколько минут со 100% точностью, поэтому мы считаем свой подход более общим.

Также решалась задача согласованного во всём приложении оформления GUI [33]. Подход основан на идее заполнения: в предположении, что уже есть набор согласованных расположений, решается проблема добавления ещё одного элемента так, чтобы новое расположение было согласовано со старым. Эта задача связана с нашей, так как мы можем рассматривать уже существующее расположение как неявно заданный гайдлайн, но мы можем указать несколько потенциальных недостатков. Во-первых, учитывается только добавление элементов, без удаления. Во-вторых, добавление/удаление компонент не обязательно приводит к “монотонному” изменению расположения: добавление ещё одного поля ввода может существенно поменять расположение. Наконец, начальный согласованный дизайн редко появляется из воздуха, наверняка это результат следования уже существующему гайдлайну, который был описан явно.

В системе [34] поставлена гораздо более широкая задача синтеза без гайдлайна, а только на основе эстетических, эргономических и других метрик. Для синтеза используется генетический алгоритм, а качество вычисляется на основе отзывов пользователей. Сам синтез работает часами. Хотя подход потенциально позволяет создавать эстетически приятные интерфейсы без участия дизайнера, задача получения корректного расположения для существенно разных структур обходится стороной.

Встречается интересная задача [35] исследования различных расположений элементов. Она ставит своей целью помочь дизайнерам разрабатывать убедительные и разнообразные расположения элементов управления. Вводится набор ограничений, с помощью которого дизайнеры составляют требования к результату. Любопытно, что в наших терминах этот набор ограничений является смесью структурных ограничений и ограничений на расположение: например, можно указывать и порядок элементов, и выравнивание. По набору ограничений система порождает множество расположений, удовлетворяющих им. Изменяя ограничения дизайнер исследует возможные интерфейсы. Для решения ограничений используется метод ветвей и границ. С ростом числа возможных решений применяются эвристики для отсечения эстетически неподходящих ответов. Несмотря на то, что эта работа похожа на нашу, она нацелена на дизайнеров и может рассматриваться скорее как средство разработки гайдлайна, чем как средство получения расположения элементов, удовлетворяющего гайдлайну.

8. Заключение

Данная работа посвящена синтезу GUI на основе гайдлайна. Наш подход позволяет автоматически располагать элементы управления GUI так, что результат по построению соответствует набору правил, сформулированных дизайнером. Прототип реализации позволяет синтезировать реалистичные примеры промышленных интерфейсов, с учетом промышленных гайдлайнов за разумное время.

Одним из интересных вопросов, является необходимость использования реляционного программирования для решения нашей задачи. Да, набор правил, регламентирующих систему переписывания, в принципе, может быть реализован без использования реляционного подхода. Однако мы считаем, что в этом случае большая часть работы по обоснованию корректности решения должна быть повторена заново. В нашем же случае обоснование тривиально следует из полноты поиска miniKanren и полноты по опровержению (англ. refutational completeness) нашего решения. Мы также предсказываем, что решение потребует изобретения заново недетерминизма и поиска с возвратами, которые уже есть в реляционном программировании. Также дуальность между экземплярами в структуре и реляционными отношениями, изначально не ожидаемая нами, по нашему мнению, демонстрирует, что реляционное программирование — это подходящий подход для решения поставленной задачи.

Список литературы / References

1. Haft M., Humm B., Siedersleben J. The Architect's Dilemma — Will Reference Architectures Help? Quality of Software Architectures and Software Quality, 2005, pp. 106-122.
2. Ingram S. (2016) The Thumb Zone: Designing For Mobile Users. Smashing Magazine (online). Available at: <https://www.smashingmagazine.com/2016/09/the-thumb-zone-designing-for-mobile-users>, accessed 30.08.2024.
3. Ergonomics of human-system interaction — Part 210: Human-centred design for interactive   
   systems. ISO 9241-210:2019, International Organization for Standardization, 2019. Available at: <https://www.iso.org/standard/77520.html>.
4. Gerber E., Carroll M. The psychological experience of prototyping. Design Studies, vol. 33, isuue 1, 2012, pp 64-84. DOI: 10.1016/j.destud.2011.06.005.
5. Friedman D.P., William W.E., Kiselyov O., Hemann J. The Reasoned Schemer. The MIT Press, 2nd edition, Cambridge, USA, 2005, 224 p.
6. Lozov P., Verbitskaia E., Boulytchev D. Relational Interpreters for Search Problems. In miniKanren and Relational Programming Workshop, 2019.
7. Leonardo de Moura, Bjørner N. Z3: An Efficient SMT Solver. Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, 2008, pp. 337-340.
8. Bengfort J. Thin vs. Thick vs. Zero Client: What’s the Right Fit for Your Business? Online. Available at: <https://biztechmagazine.com/article/2018/10/thin-vs-thick-vs-zero-client-whats-right-fit-your-business-perfcon>, accessed: 30.08.2024.
9. Intellij platform UI guidelines: Layout (online). JetBrains s.r.o., 2000-2022. Available at: <https://jetbrains.github.io/ui/principles/layout>, accessed: 30.08.2024.
10. Kosarev D., Boulytchev D. Typed Embedding of a Relational Language in OCaml. Electronic Proceedings in Theoretical Computer Science, 2016, pp. 1-22.
11. Kiselyov O., Chung-chieh Shan, Friedman D.P., Amr S. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). In Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, New York, USA, 2005, pp. 192-203.
12. Rozplokhas D., Vyatkin A., Boulytchev D. Certified Semantics for Relational Programming. Programming Languages and Systems, APLAS 2020, Lecture Notes in Computer Science, vol. 12470, Springer, Cham, pp. 167-185.
13. Comon H. Disunification: A Survey. Computational Logic — Essays in Honor of Alan Robinson. MIT Press, 1991, 322-359.
14. Alvis C.E., Willcock J.J., Carter K.M., Byrd W.E., Friedman D.P. cKanren: miniKanren with Constraints. Proceedings of the 2011 Annual Workshop on Scheme and Functional Programming, 2011.
15. Byrd W.E., Friedman D.P. αKanren A Fresh Name in Nominal Logic Programming. In Scheme and Functional Programming, 2007.
16. Abramov S., Glück R. From Standard to Non-Standard Semantics by Semantics Modifiers. International Journal of Foundations of Computer Science, vol. 12, issue 2, 2001, pp. 171-211. DOI: 10.1142/S0129054101000448.
17. Abramov S., Glück R. Combining Semantics with Non-standard Interpreter Hierarchies. FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science, Springer Berlin Heidelberg, 2000, pp. 201-213.
18. Byrd W.E., Holk E., Friedman D.P. MiniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). Proceedings of the Annual Workshop on Scheme and Functional Programming, Association for Computing Machinery, New York, USA, 2012, pp. 8-29.
19. Byrd W.E., Ballantyne M., Rosenblatt G., Might M. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). Proceedings of ACM Program. Lang., Association for Computing Machinery, New York, USA, 2017, pp. 8:1-8:26.
20. Kosarev D., Lozov P., Boulytchev D. Relational Synthesis for Pattern Matching. Programming Languages and Systems, Springer International Publishing, Cham, 2020, pp.293-310.
21. Guthmann O., Strichman O., Trostanetski A. Minimal Unsatisfiable Core Extraction for SMT. 2016 Formal Methods in Computer-Aided Design (FMCAD), Mountain View, CA, USA, 2016, pp. 57-64. DOI: 10.1109/FMCAD.2016.7886661.
22. React: A JavaScript Library for Building User Interfaces. Meta Platforms, Inc. Available at: <https://reactjs.org/>, accessed: 30.08.2024.
23. Jetpack Compose. Android Developers. Available at: <https://developer.android.com/jetpack/compose>, accessed: 30.08.2024.
24. SwiftUI. Apple Inc. Available at: <https://developer.apple.com/xcode/swiftui>, accessed: 30.08.2024.
25. Streamlit framework site. Available at: <https://docs.streamlit.io>, accessed: 30.08.2024.
26. The JavaScript library for bespoke data visualization. Available at: <https://d3js.org>, accessed: 30.08.2024.
27. Streamlit layouts and containers. Available at: <https://docs.streamlit.io/develop/api-reference/layout>, accessed: 30.08.2024.
28. Borning A. Wallingford: Toward a Constraint Reactive Programming Language. Companion Proceedings of the 15th International Conference on Modularity, Association for Computing Machinery, New York, NY, USA, 2016, pp. 45-49. DOI: 10.1145/2892664.2892667.
29. Badros G.J., Borning A., Stuckey P.J. The Cassowary Linear Arithmetic Constraint Solving Algorithm. ACM Trans. Comput.-Hum. Interact., vol. 8, issue 4, Association for Computing Machinery, New York, NY, USA, 2001, pp. 267-306. DOI: 10.1145/504704.504705.
30. Bo Cai, Jian Luo, Zhen Feng. A novel code generator for graphical user interfaces. Scientific Reports, vol. 13, 2023. DOI: 10.1038/s41598-023-46500-6.
31. Bielik P., Fischer M., Vechev M. Robust relational layout synthesis from examples for Android. Proc. ACM Program. Lang., vol. 2, Association for Computing Machinery, New York, NY, USA, 2018. DOI: 10.1145/3276526.
32. Android ConstraintLayout widget. Accessed: 30.08.2024, available at: <https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintLayout>.
33. Brückner L., Leiva L.A., Oulasvirta A. Learning GUI Completions with User-defined Constraints. ACM Trans. Interact. Intell. Syst., vol. 12, Association for Computing Machinery, New York, NY, USA, 2022. DOI: 10.1145/3490034.
34. Shiripour M., Dayama N.R., Oulasvirta A. Grid-based Genetic Operators for Graphical Layout Generation. Proc. ACM Hum.-Comput. Interact., vol. 5, Association for Computing Machinery, New York, NY, USA, 2021. DOI: 10.1145/3461730.
35. Swearngin A., Wang C., Oleson A., Fogarty J., Amy J. Ko. Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints. Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, 2020. Available at: <https://api.semanticscholar.org/CorpusID:210177012>, accessed: 30.08.2024.

Информация об авторах / Information about authors

Дмитрий Сергеевич КОСАРЕВ является ассистентом кафедры Системного программирования мат-мех. факультета СПбГУ. Его научные интересы включают функциональное программирование, компиляторы и реляционное программирование.

Dmitry Sergeevich KOSAREV is an assistant at the Chair of System Programming of Mathematics and Mechanics Faculty of SPBU. His research interests include functional programming, compilers, and relational programming.

Петр Алексеевич ЛОЗОВ является кандидатом физико-математических наук. Сфера научных интересов: статический анализ, трансляция языков программирования, функционально-реляционное программирование.

Petr Alekseevich LOZOV is a candidate of physical and mathematical sciences. Research interests: static analysis, translation of programming languages, functional-relational programming.

Дмитрий Юрьевич БУЛЫЧЕВ – доцент кафедры системного программирования мат.-мех. факультета СПбГУ, кандидат физико-математических наук. Область научных интересов - языки и инструменты программирования, компиляторы, функциональное, логическое и реляционное программирование, анализ и синтез программ.

Dmitry Yuryevich BOULYTCHEV is an associate professor of the System Programming Chair of the Faculty of Mathematics and Mechanics of SPBU, Candidate of Physical and Mathematical Sciences. His research interests include programming languages ​​and tools, compilers, functional, logical and relational programming, program analysis and synthesis.

Денис Сергеевич ФОКИН – руководитель направления статического и динамического анализа департамента облачных вычислений в Российском исследовательском институте Huawei. Научные интересы включают статический анализ программ, в том числе, абстрактную интерпретацию и символьное исполнение.

Denis Sergeevich FOKIN is the head of the static and dynamic analysis division of the cloud computing department at the Huawei Russian Research Institute. His research interests include static analysis, including abstract interpretation and symbolic execution.

1. http://minikanren.org [↑](#footnote-ref-1)
2. Детальное сравнение miniKanren и Prolog можно найти здесь: <http://minikanren.org/minikanren-and-prolog.html> (проверено: 10 января 2025). [↑](#footnote-ref-2)
3. http://ocaml.org [↑](#footnote-ref-3)
4. https://github.com/PLTools/OCanren [↑](#footnote-ref-4)
5. https://github.com/michaelballantyne/faster-minikanren [↑](#footnote-ref-5)
6. https://github.com/JaimieMurdock/HK, https://github.com/rntz/ukanren [↑](#footnote-ref-6)
7. https://github.com/lightyang/minikanren-ocaml [↑](#footnote-ref-7)
8. https://github.com/acfoltzer/Molog [↑](#footnote-ref-8)
9. https://github.com/jvranish/MiniKanrenT [↑](#footnote-ref-9)
10. Реификация (англ. reification) — превращение чего-то абстрактного (т.е. существующего в виде идеи) во что-то конкретное. Кембриджский словарь. <https://dictionary.cambridge.org/dictionary/english/reification> (проверено 28 января 2025 г.) [↑](#footnote-ref-10)
11. Например, <https://blogs.janestreet.com/the-perils-of-polymorphic-compare> (проверено 10 января 2025) [↑](#footnote-ref-11)
12. https://ocaml.org/manual/5.3/api/Obj.html (проверено 28 января 2025 г.) [↑](#footnote-ref-12)
13. В конкретном синтаксисе “*𝛼* logic”. [↑](#footnote-ref-13)
14. В конкретном синтаксисе это “inj” and “prj”. [↑](#footnote-ref-14)
15. <https://doc.qt.io/qt-5/qtquick-controls2-qmlmodule.html> (проверено: 26.08.2024) [↑](#footnote-ref-15)