Деревья с балансировкой по весу — параметризованное семейство самобалансирующихся деверьев поиска, широко применяющееся в функциональном программировании. Для них существует однопроходная версия алгоритмов модификации, более быстрая, но менее изученная, особенно в вопросе допустимых значений параметров. В данной работе предложена верифицированная в Coq реализация операции вставки ключа и формально доказана допустимость нескольких практически значимых значений параметров. Для некоторых из них установление корректности является новым результатом даже с учётом неформальных доказательств.

СОДЕРЖАНИЕ

Введение

Формальная верификация — доказательство в рамках некоторой формальной системы соответствия программы её спецификации — становится тем важнее, чем большую роль в жизни человека начинают играть компьютерные системы. Вместе с развитием инструментов, возрастает и их применение в индустрии, даже компания Facebook отказалась от слогана «Move Fast and Break Things»[2] и стала применять отдельные методы формальной верификации[10]. Программные ошибки становились причиной крушения космических кораблей[9] и гибели пациентов[15]; нет смысла пытаться перечислить случаи сбоев или утечек персональных данных в широко используемых сервисах. Особенно важно верифицировать алгоритмы со сложным доказательством корректности, поскольку ошибки могут появиться не только в их реализации, но и в самих доказательствах.

Coq[19][6] — система для работы с формальными доказательствами, основанная на соответствии Карри–Ховарда. Она широко применяется для верификации программного обеспечения, к ярким образцам можно отнести верифицированный компилятор C CompCert[14] и верификацию реализации HMAC для SHA-256 в OpenSSL[5] (не только соответствия программы протоколу, но и криптографических свойств самого протокола). Возможно и доказательство чисто математических утверждений: к примеру, с помощью Coq была формализована основная теорема алгебры[11].

Дерево, сбалансированные по весу (далее WBT), — достаточно популярный вид самобалансирующихся двоичных деревьев поиска, впервые представленный Невергельтом и Рейнгольдом[16]; для поддержания баланса после модификаций они использовали те же вращения, что и в АВЛ-дереве[20], но выполняли их ещё на спуске от корня к изменяемому узлу.

Однако в их алгоритме содержались существенные ошибки, и классическим стал алгоритм модификации, предложенный Блумом и Мельхорном[8], с отдельным проходом от узла к корню для балансировки. Позже Лэй и Вуд представили исправленный однопроходный алгоритм модификации вместе с детальным доказательством корректности[13], но их работа долгое время оставалась незамеченной. Только спустя почти 30 лет на неё обратили внимание Барт и Вагнер; проведя эмпирический анализ, они обнаружили значительное (от 23%) улучшение производительности по сравнению с двухпроходным вариантом в практической задаче из сферы энергетики, в некоторых синтетических тестах результаты даже оказались лучше, чем у красно-чёрных деревьев[3].

В Haskell, WBT используется как основа для де-факто стандартных контейнеров `Data.Set` и `Data.Map`. В 2010 году в них была обнаружена ошибка: использование недопустимых значений параметров (алгоритм допускает настройку степени сбалансированности дерева, в том числе и в однопроходной версии) приводило к разбалансировке дерева. Хираи и Ямамото подошли к исправлению этой ошибки фундаментально — они определили вид множества допустимых параметров и формально доказали с помощью Coq его полноту и корректность[12]. Основываясь на их работе, Нипков и Дирикс разработали в Isabelle, другой системе для работы с формальными доказательствами, верифицированную (в том числе и на соответствие требованиям к деревьям поиска) реализацию WBT[17]. Всё это относится исключительно к двухпроходному варианту алгоритма, сказать что-либо о корректности алгоритма и допустимости параметров для однопроходной версии это не позволяет.

Целью данной работы является разработка и верификация средствами Coq реализации однопроходной версии алгоритма модификации WBT и исследование множества допустимых параметров.

1 ОПИСАНИЕ АЛГОРИТМА

Существует два подхода к определению деревьев с балансировкой по весу: с использованием функции баланса и без, напрямую через размеры поддеревьев. Первый подход чаще применяется в публикациях с фокусом на теоретическом аспекте, поскольку располагает к большей краткости, а второй — в публикациях, освещающих скорее вопрос программирования алгоритма, поскольку отражает код эффективных реализаций. В рамках этого раздела будет использован первый подход, чтобы как можно быстрее познакомить читателя с алгоритмом, но в конце будет описан и второй, который будет применяться далее.

Пусть T — двоичное дерево, и если оно непусто, то T_l и T_r — его левое и правое поддеревья, соответственно. Обозначая количество узлов в дереве как $|T|$, определим функции *веса* и *баланса*:

$$w(T) = 1 + |W|, \qquad \beta(T) = \frac{w(T_l)}{w(T)}. \tag{1.1}$$

При зафиксированном $\alpha \in [0, 1/2]$, дерево T называется *деревом с ограниченным балансом* (множество таких деревьев обозначим BB_α) либо если оно пусто, либо если $T_l, T_r \in BB(\alpha)$, и

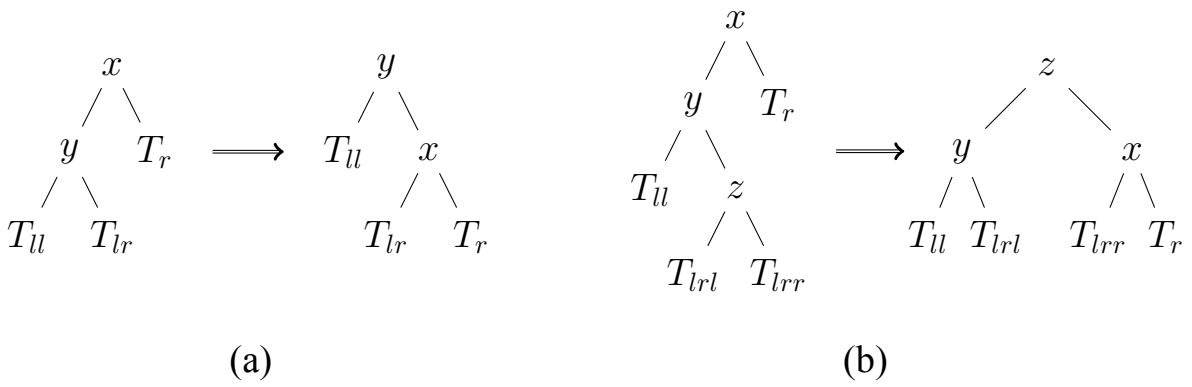
$$\alpha \leqslant \beta(T) \leqslant 1 - \alpha. \tag{1.2}$$

Более известным названием, особенно в смысле структуры данных, является «*дерево, сбалансированное по весу*». Важным свойством, позволяющим использовать такие деревья как основу для эффективной реализации абстрактных типов данных «множество» и «ассоциативный массив», является то, что их высота не превосходит[16]

$$\frac{\log(n+1) - 1}{\log 1/(1-\alpha)}. \tag{1.3}$$

Если дерево поиска T принадлежит BB_α, то для некоторых значений α дерево T', полученное из T вставкой или удалением одного ключа, можно

привести к виду BB_α с помощью изображённых на рисунке 1.1 вращений и их зеркальных отражений. Один способ, более известный и исследованный, аналогичен определению операций для АВЛ-дерева. Сначала выполняется модификация как для простого дерева поиска, а после в каждом затронутом узле выполняется процедура восстановления сбалансированности, сначала для родителя добавленного или удалённого узла, потом для родителя родителя, и так далее, вплоть до корневого узла.



(a) (b)

Рис. 1.1 – Простое (a) и двойное (b) правые вращения

Процедура восстановления сбалансированности для узла, корня поддерева T, такова:

- если $\beta(T) \in [\alpha, 1-\alpha]$, не делать ничего;
- если $\beta(T) > 1-\alpha$, выполнить одно из вращений, изображённых на рисунке 1.1: простое, если $\beta(T_l) \geqslant 1-\gamma$, или двойное, иначе;
- если $\beta(T) < \alpha$, выполнить вращение, зеркальное к одному из изображённых на рисунке 1.1 вращений: простому, если $\beta(T_r) \leqslant \gamma$, или двойному, иначе.

γ — ещё один параметр алгоритма. Двухпроходная процедура работает корректно для $\alpha \in [2/11, 1-\sqrt{2}/2]$, в качестве γ можно взять, например, $1/(2-\alpha)$[12].

В однопроходном алгоритме модификации необходимо гарантировать сбалансированность узла до рекурсивного спуска в одно из поддеревьев,

когда ещё не известно, изменится ли множество ключей (и соответственно, размер поддерева). Добавляемый ключ может присутствовать, а удаляемый — отсутствовать в исходном дереве, в таком случае модификация называется *избыточной*. Учёт этого факта требует интуитивно простой конструкции, которую достаточно трудно определить формально. Пусть выполняется модификация дерева T, U — некоторое поддерево T. Пусть операция была выполнена по алгоритму для простого, не самобалансирующегося дерева поиска, и при этом не оказалась избыточной, в результате чего было получено дерево T', в котором U соответствует поддерево U'. Далее запись $\beta'(U)$ будет обозначать значение $\beta(U')$ в этой гипотетической ситуации. Например, если вставка или удаление выполняется для ключа меньшего, чем ключ в корне дерева T, то для T_r и всех его поддеревьев значения β и β' равны; а для самого T

$$\beta'(T) = \begin{cases} \frac{w(T_l)+1}{w(T)+1}, & \text{если выполняется вставка,} \\ \frac{w(T_l)-1}{w(T)-1}, & \text{если выполняется удаление.} \end{cases} \tag{1.4}$$

Чтобы получить однопроходные алгоритмы вставки и удаления ключа для WBT, необходимо в алгоритмы для простого дерева поиска добавить следующую процедуру, выполняемую перед спуском в поддерево:

- если $\beta'(T) \in [\alpha, 1-\alpha]$, не делать ничего;
- если $\beta'(T) > 1-\alpha$, выполнить одно из вращений, изображённых на рисунке 1.1: простое, если $\max(\beta(T_l), \beta'(T_l)) \geqslant 1-\gamma$, или двойное, иначе;
- если $\beta'(T) < \alpha$, выполнить вращение, зеркальное к одному из изображённых на рисунке 1.1 вращений: простому, если $\min(\beta(T_r), \beta'(T_r)) \leqslant \gamma$, или двойному, иначе.

Чтобы сложность работы алгоритма была логарифмической, для каждого узла необходимо хранить размер его поддерева; но размер модифицируемого дерева нельзя предсказать, не зная заранее, не окажется ли модификация избыточной. Поскольку основная мотивация для однопроходного

алгоритма — избавление от прохода снизу вверх в случае неизбыточных операций (для избыточных он не требуется и двухпроходном варианте), сохранённые в узлах размеры поддеревьев обновляются в предположении неизбыточности операции. Если же операция оказывается избыточной, то, чтобы исправить некорректные значения, всё-таки выполняется обратный проход по дереву.

Что касается альтернативного, более удобного на практике варианта алгоритма, единственное отличие состоит в замене неравенств 1.2 на эквивалентную систему

$$\begin{cases} w(T_l) \leqslant \Delta \cdot w(T_r), \\ w(T_r) \leqslant \Delta \cdot w(T_l), \end{cases} \tag{1.5}$$

где $\Delta = (1 - \alpha)/\alpha$. Аналогичным образом неравенства с γ заменяются на эквивалентные в терминах $\Gamma = \gamma/(1 - \gamma)$. Далее в работе под параметрами алгоритма подразумевается пара $\langle \Delta, \Gamma \rangle$.

2 РЕАЛИЗАЦИЯ АЛГОРИТМА В COQ

Для верификации алгоритма необходимо иметь его реализацию в виде конкретной программы. При этом понятие программы может лежать на широком спектре от последовательности байт, кодирующей исходный код на некотором языке программирования, вплоть до терма Gallina — языка программирования, на котором основан Coq. Выбор точки на этом спектре задаёт баланс между, соответственно, максимально широким пониманием термина «программа» и удобством процесса верификации. С одной стороны, программы на Gallina должны соблюдать строгие, даже на фоне других чисто функциональных языков программирования, ограничения. С другой стороны, чем ближе к противоположному концу спектра, тем более сложными и техническими становятся доказательства.

В данной работе выбор был сделан в пользу использования естественных для Coq'а программ на Gallina, это позволило сфокусироваться на доказательстве корректности алгоритма балансировки. Чтобы приблизить реализацию к практической применимости, код был написан с оглядкой на возможную экстракцию в OCaml или Haskell; например, тип данных, используемый для хранения, абстрагирован через `Int`, что позволяет при экстракции заменить его на один из встроенных в целевой язык целочисленных типов.

2.1 Теория MSets

Одно из немедленных преимуществ выбранного подхода — возможность воспользоваться стандартной библиотекой Coq. В её состав входит, в числе прочего, теория `MSets`, определяющая интерфейс абстрактного типа данных «множество» и несколько реализаций для этого интерфейса. Общая логика выделена в отдельные модули и функторы, что упрощает разработку новых реализаций.

К примеру, интерфейс RawSets и функтор Raw2Sets позволяют сформулировать тип множества как допускающий «плохие» значения, для которых результаты операций не обязаны соответствовать соответствовать спецификации (например, тип двоичного дерева, в котором «плохими» являются деревья, не являющиеся деревьями поиска), а после этого выделить тип-подмножество, что позволяет использовать итоговый модуль без оглядки на предусловия.

Наиболее полезен для целей данной работы модуль MSetGenTree.Ops, предоставляющий базу для реализации двоичных деревьев поиска. В нём определён тип двоичного дерева (в каждом узле хранится дополнительная информация, чей тип задаётся как параметр модуля) и реализацию не-модифицирующих операций над ним. В расширяющем его модуле MSetGenTree.Props определяется определяется предикат «быть деревом поиска» и набор связанных тактик, а также верифицируются операции, определённые в Ops. Эта пара модулей вместе с верифицированными операциями модификации как раз удовлетворяют RawSets.

Объявляемый MSets интерфейс несколько больше чем минимально необходимый для деревьев поиска (требуются, например, операции объединения и пересечения двух множеств), поэтому в работе используется копия MSets, из которой удалены теоретико-множественные операции и функции высшего порядка filter и partition.

2.2 Проблема остановки

При разработке на Gallina, проблемой может оказаться требование гарантированного завершения программы — бесконечно работающую программу можно было бы использовать для доказательства произвольного утверждения. В случае операций над WBT, вызывает трудности не само требование, а то, как проверяется его выполнение: в рекурсивных функциях

Gallina возможна только структурная рекурсия по фиксированному аргументу. То есть, должен быть зафиксирован аргумент, для которого в любых рекурсивных вызовах допустимы только подтермы его значения при исходном вызове функции. Однопроходная модификация WBT это ограничение, очевидно, не соблюдает: перед рекурсивным вызовом для одного из поддеревьев может быть выполнено вращение, а при этом создаются новые термы, не входящие в старое дерево, и именно в такой терм может «спуститься» функция.

Один из способов решения такой проблемы — фундированная рекурсия. Необходимо построить на множестве наборов аргументов фундированный порядок так, чтобы рекурсивные вызовы применялись только к наборам аргументов меньшим, относительно этого порядка, исходного. Очевидно, из такого построения следует, что программа всегда останавливается: отсутствие бесконечно убывающих цепочек влечёт отсутствие бесконечной рекурсии. Примечателен этот подход тем, что он сводится к структурной рекурсии: в качестве структурно убывающего аргумента выступает доказательство отсутствия убывающих цепочек с началом в текущем наборе аргументов. Зачастую порядок не определяется с нуля, а переносится с некоторого известного фундированного множества с помощью отображения в такое множество; для операций над WBT в качестве такого отображения подходит размер дерева: при спуске в одно из поддеревьев он уменьшится хотя бы на единицу. Воспользоваться этой техникой без погружения в технические детали позволяет команда `Function`:

```
Function add x s {measure cardinal s} := match s with
| Leaf => singleton x
| Node _ l y r =>
  match X.compare x y with
  | Eq => s
```

```
  | Lt =>
    if boundedBy Delta (1 + weight l) (weight r)
    then node (add x l) y r
    else match l with
    | Node _ ll ly lr =>
      match X.compare x ly with
      | Eq => s
      | Lt =>
        if boundedBy Gamma (weight lr) (weight ll)
        then node (add x ll) ly (node lr y r)
        else match lr with
        | Node _ lrl lry lrr =>
          node (add x (node ll ly lrl)) lry (node lrr y r)
        | Leaf => (* impossible *) node (add x l) y r
        end
      ...
end.
all: intros; simpl; lia. Defined.
```

Для доказательство убывания размера входного дерева достаточно применить тактику `lia` — решающую процедуру для бесквантворной линейной арифметики над целыми числами[7]. Дополнительным бонусом становится генерация схемы функциональной индукции, о чём речь пойдёт дальше.

3 ВЕРИФИКАЦИЯ АЛГОРИТМА

Верификацию модифицирующей операции можно разделить на две независимые части: верификация свойств общих для двоичного дерева поиска (сохранение упорядоченности ключей и, собственно, соответствие семантике операции изменения набора ключей) и специфичных для деревьев, сбалансированных по весу (корректность сохранённых данных о величине поддерева и, конечно, сохранения сбалансированности). Это разделение закреплено размещением доказательств в не зависящие друг от друга функторы `Props` и `BalanceProps`.

Для доказательство всех этих свойств рекурсивной функции, конечно, хочется воспользоваться индукцией. Однако в формальном доказательстве слово «индукция» недостаточно конкретно, необходимо указать конкретную схему индукции. Простая индукция по определению типа `tree` не подойдёт по той же причине, по которой нельзя было объявить операции как `Fixpoint`. Не сработает и трюк с обобщением по размеру дерева и последующим применением `nat_ind`: размер дерева-аргумента при рекурсивных вызовах может уменьшаться сильнее, чем на единицу. Необходима индукция по фундированному множеству.

И снова от работы с техническими деталями спасает `Function`-объявление: для объявленных с её помощью функции автоматически генерируется схема функциональной индукции. Основываясь на этой схеме, тактика `functional induction add x t` преобразует цель из свойства `P (add x t)` в набор целей вида `P res` для всех возможных `res`, получаемых в результате однократной замены `add` на её тело с последующим разбором случаев в `if`- и `match`-выражениях. В набор гипотез к каждой цели добавляются равенства, полученные в результате разбора случаев, и гипотезы индукций для каждого рекурсивного вызова `add`.

В случае операции добавления элемента, доказательство каждого из четырёх свойств разбивается на 24 случая. Чтобы сделать достижимыми как задачу написания доказательств, так и их понимания, активно используется автоматизация доказательств.

3.1 Свойства дерева поиска

Библиотека `MSets` содержит формализацию свойств двоичного дерева поиска и несколько полезных для работы с ними тактик. Что касается предикатов, в рамках данной работы необходимо работать всего с несколькими, имеющими достаточно интуитивные (и, безусловно, интуиционистские) определения:

```
(* ”ключ x присутствует в дереве” *)
Inductive InT (x : elt) : tree -> Prop :=
  | IsRoot : forall c l r y, X.eq x y
     -> InT x (Node c l y r)
  | InLeft : forall c l r y, InT x l
     -> InT x (Node c l y r)
  | InRight : forall c l r y, InT x r
     -> InT x (Node c l y r).
(* ”все ключи из дерева s меньше/больше чем x” *)
Definition lt_tree x s := forall y, InT y s -> X.lt y x.
Definition gt_tree x s := forall y, InT y s -> X.lt x y.
(* ”дерево является деревом поиска” *)
Inductive bst : tree -> Prop :=
  | BSLeaf : bst Leaf
  | BSNode : forall c x l r, bst l -> bst r ->
     lt_tree x l -> gt_tree x r -> bst (Node c l x r).
```

Отметив, что принимаемые доказательства `bst` всегда абстрагируется с помощью класса предикатов `Ok`, рассмотрим лемму, формализующую семантику операции добавления элемента в множество:

```
Lemma add_spec' : forall s x y `{Ok s},
  InT y (add x s) <-> X.eq y x \/ InT y s.
```

Тут есть относительно нетривиальная (особенно с учётом гипотез индукции) пропозициональная структура, а потому в качестве основы доказательства используется тактика `intuition`. Она базируется на решающей процедуре для исчисления высказываний, но если для каких-то атомарных формул не находится пропозиционального доказательства, тактика не завершается с ошибкой, а оставляет их для дальнейшего доказательства, напоследок пробуя разрешить их с помощью `auto with *`.

Большая часть целей, к которым сводит задачу `intuition`, решается комбинацией из применения конструкторов `InT` к одной из гипотез, которую, возможно, необходимо извлечь из построения `InT`. С применением конструкторов и гипотез справляется `auto`, а для инвертирования индуктивных предикатов над деревьями `MSets` предоставляет тактику `invtree f`. Не хватает этих шагов только для случаев, когда добавляемый элемент был обнаружен в дереве; тут необходимо применить транзитивность равенства, что можно сделать с помощью `eauto`.

Несколько более трудоёмким оказывается доказательство сохранения свойства упорядоченности ключей:

```
Instance add_ok t x `(Ok t) : Ok (add x t).
```

Во-первых, аналогично прошлой тактике тут требуется конструирование и инвертирование, но уже для предикатов `lt_tree` и `gt_tree`, не определённых как индуктивные. Леммы для конструирования определяются в `MSets`, инвертирование было реализовано следующим образом:

```
Lemma lt_tree_inv : forall y s l x r,
  lt_tree y (Node s l x r) ->
  lt_tree y l /\ X.lt x y /\ lt_tree y r.
Proof. intuition; unfold lt_tree; auto. Qed.
Lemma gt_tree_inv : ...
Ltac inv_xt_tree := try match goal with
  | H : lt_tree _ (Node _ _ _ _) |- _ =>
    apply lt_tree_inv in H;
    decompose [and] H; clear H;
    inv_xt_tree
  | H : gt_tree _ (Node _ _ _ _) |- _ => ...
end.
```

Во-вторых, полученных инвертированием гипотез не всегда достаточно. К примеру, простое правое вращение преобразует дерево `node (node ll ly lr) y r` в `node ll ly (node lr y r)`, но предикат `bst` (для исходного дерева) в явном виде не подразумевает, что ключи из поддерева `r` должны быть больше `ly`, это следует только из транзитивности отношения порядка на множестве ключей. Такое рассуждение оформлено в лемме `gt_tree_trans` из `MSets`, а после применения тактики `inv_xt_tree` применить эту лемму можно и без тяжеловесного `eauto`, для поиска промежуточного ключа хватает сопоставления цели с шаблоном `H1 : X.lt ?x ?y, H2 : gt_tree ?y ?s |- _`.

В-третьих, предположение индукции говорит только о выполнении предиката `bst` для результата рекурсивного вызова, его использование как левого или правого поддерева требует выполнения для него предиката `lt_tree` или `gt_tree`, соответственно. На этот раз, определение этих предикатов на основе `InT`, наоборот, удобно, поскольку это позволяет применить `add_spec`.

Чтобы закрывать такие цели, достаточно добавить в базу подсказок auto следующую тактику:

```
Ltac xt_tree_add :=
  intro; (* unfolds head *)
  rewrite add_spec;
  [ intros [ | ]; [ | inv ] | ].
```

3.2 Сбалансированность

Для доказательства сохранения у деревьев свойства сбалансированности необходимо, в первую очередь, это свойство сформулировать. Формализация тут крайне естественна, поскольку исходное, неформальное, определение, идеально ложится на понятие индуктивных предикатов:

```
Inductive balanced (n m: nat) : tree -> Prop :=
  | BalancedLeaf : balanced n m Leaf
  | BalancedNode : forall s l x r,
      balanced n m l -> balanced n m r ->
      m * (1 + cardinal l) <= n * (1 + cardinal r) ->
      m * (1 + cardinal r) <= n * (1 + cardinal l) ->
      balanced n m (Node s l x r)
.
```

Простоты ради, коэффициент Δ не выступает параметром напрямую, вводится два параметра `n` и `m` — его числитель и знаменатель. Кроме того, при работе во вселенной `Prop` разумнее применять `nat` чем `Int`; даже определение размера множества `cardinal` (предоставляемое `MSets`) имеет тип `tree -> nat`. Этот переход к другому типу требует небольшой технической работы, которая скрывается в определении предиката `delta_balanced` (`balanced` с коэффициентом Δ).

В доказательстве утверждения сбалансированности основную трудность представляют неравенства вида `a * (1 + cardinal t1) <= b * (1 + cardinal t2)`. При этом они должны следовать из гипотез аналогичного вида, полученных либо из условий сбалансированности входного дерева, либо как результаты сравнений, выполняемых программой для выбора подходящего вращения. Кроме того, если считать параметры Δ и Γ фиксированными, то все коэффициенты `a` и `b` оказываются константными, что позволяет считать цель утверждением из арифметики Пресбургера. Следовательно, завершить доказательство можно с помощью тактики `lia`.

Первый шаг на пути к главному доказательству данной работы вряд ли описан в какой-либо публикации, обсуждающей (неформально) корректность алгоритмов модификации деревьев с балансировкой по весу; по крайней мере, если корректность не доказывается формальными методами. Этим шагом является верификация функции `size` — доказательство того, она действительно вычисляет размер дерева (то есть, корректности дополнительной информации, хранящейся в узлах дерева). Это тривиальное, но крайне важное условие, ведь именно `size` используется для принятия решений о необходимости выполнения вращений дерева. Весь процесс заключается в определении чисто технического индуктивного предиката и леммы, которая уже свзяывает значения `size` и `cardinal`:

```
Inductive sizedTree : tree -> Prop :=
  | SizedLeaf : sizedTree Leaf
  | SizedNode : forall l x r,
                sizedTree l ->
                sizedTree r ->
                sizedTree (Node
                  (1 + size l + size r)
```

```
                  l x r)
Lemma size_spec : forall tr, sizedTree tr ->
  i2z (size tr) = Z.of_nat (cardinal tr).
```

Доказательство леммы состоит из аккуратного применения редукций и тактик `lia` и `i2z` (последняя определяется в модуле `Int.MoreInt` и позволяет переходить от `Int` к `Z`). Благодаря тому, что `SizedNode` просто отражает использование в программе конструктора `Node`, доказательство сохранения сохранения свойства `sizedTree` сводится к функциональной индукции и применению для всех случаев тактик `invtree sizedTree; auto`. Итак, `sizedTree` позволяет привязать к условиям на ограниченность баланса их семантику, для этого сформулирована тактика

```
Ltac reflect_boundedBy := lazymatch goal with
  | H : boundedBy _ _ _ = _ |- _ =>
    simpl in H;
    MI.i2z;
    rewrite ?size_spec in H;
    [ simpl_boundedBy | assumption.. ]
  | _ => idtac
end.
```

Ещё одно очевидное свойство, которое необходимо доказать, касается влияния операций на размер дерева. Так, `add` либо сохранит его (если ключ уже есть), либо увеличит его на единицу. Опять же, доказательство состоит из индукции, упрощения сравниваемых термов и доказательства тривиального равенства с помощью `lia`. Для применения этого свойства используется тактика

```
Ltac rw_add_cardinal := match goal with
  | |- context [cardinal (add ?x ?tr)] =>
```

```
    let H := fresh in
    destruct (add_cardinal tr x) as [H | H];
    rewrite H
end.
```

После всей этой подготовительной работы, изложенная выше идея доказательства достаточно кратко выражается в Coq'e:

```
Hint Constructors balanced : core.
Hint Extern 8 => rewrite cardinal_node in * : core.
Hint Extern 9 => rw_add_cardinal : core.
Hint Extern 10 => lia : core.
Theorem add_balanced : forall t x,
  sizedTree t -> delta_balanced t ->
  delta_balanced (add x t).
Proof.
  intros t x Hsize Hbalance.
  functional induction add x t;
  try apply singleton_balanced;
  unfold_helpers; unfold_delta;
  invtree balanced; invtree sizedTree;
  reflect_boundedBy;
  auto 6.
Qed.
```

4 О ВЫБОРЕ ПАРАМЕТРОВ

При использовании сбалансированных по весу деревьев важную роль играет выбор параметров Δ и Γ. Чем меньше их значения, тем сильнее сбалансированно дерево и меньше его глубина, что напрямую влияет на время работы всех операций; с другой стороны, возрастает и количество необходимых вращений, что приводит к замедлению вставки и удаления ключей. При этом не для всех значений параметров алгоритм работает корректно. Для однопроходной версии известно только о допустимости множества $\{\langle \Delta, \Gamma \rangle \mid 3 \leqslant \Delta \leqslant 4,5$ и $\Delta \cdot \Gamma = 1 + \Delta\}$[13].

Даже вид самих чисел может иметь существенное значение; например, оптимальным с точки зрения сбалансированности дерева является набор $\langle 1 + \sqrt{2}, \sqrt{2} \rangle$, но иррациональность коэффициентов делает его непрактичным[18]. Наоборот, единственный допустимый целочисленный набор $\langle 3, 2 \rangle$ позволяет при сравнениях размеров поддеревьев использовать одну операцию умножения, а не две, как для рациональных параметров.

В итоге, единственным разумным способом подобрать оптимальный набор параметров является эмпирическая оценка производительности. Для двухпроходного алгоритма, большое влияние оказала работа Адамса[1], в которой рекомендуется использовать значения Δ не меньше четырёх. Для однопроходного варианта можно опираться на результаты полученные Бартом и Вагнер для модифицирующих операций[3]. Они проводили замеры для следующих наборов параметров: $\langle 3, 4/3 \rangle$ (единственный доказанно корректный вариант), $\langle 1 + \sqrt{2}, 2 \rangle$ и $\langle 3, 2 \rangle$ (доказанно корректных для двухпроходного алгоритма), $\langle 2, 3/2 \rangle$ и $\langle 3/2, 5/4 \rangle$ (гарантированно некорректных). Исходя из совокупности синтетических тестов, наиболее практичными оказались $\langle 3, 4/3 \rangle$, $\langle 3, 2 \rangle$ и, несмотря на значительное количество несбалансированных узлов, $\langle 2, 3/2 \rangle$. В случае использования реальных тестовых данных (последо-

вательность операций в алгоритме SWAG[4]), оптимальным оказался набор $\langle 3, 2\rangle$.

Исходя из потенциальной практической применимости, в рамках данной работы функтор `BalanceProps` был применён для параметров $\langle 3, 4/3\rangle$, $\langle 3, 2\rangle$ и $\langle 2, 3/2\rangle$. Проверка доказательства успешно выполнилась для пар $\langle 3, 4/3\rangle$ и $\langle 3, 2\rangle$, а для $\langle 2, 3/2\rangle$, ожидаемо, завершилась ошибкой. Как уже было сказано ранее, только для $\langle 3, 4/3\rangle$ доказательство корректности было опубликовано ранее. Таким образом, корректность однопроходной вставки элементов для набора параметров $\langle 3, 2\rangle$ является новым результатом.

Заключение

В рамках работы был успешно реализован и верифицирован алгоритм однопроходной вставки для деревьев с балансировкой по весу, код доступен по адресу `https://github.com/mizabrik/verified-top-down-wbt/tree/vMIPT`. Представленное доказательство проверено для некоторых практически значимых наборов параметров, и есть основания полагать, что оно подходит для любого допустимого набора параметров. В процессе доказательства активно использовалась автоматизация, и некоторые тактики могут быть полезны для дальнейшего анализа свойств WBT.

Полученные результаты являются аналогом таковых у Нипкова и Дирикса[17] и, частично, у Хираи и Ямамото[12], но для однопроходной версии алгоритма.

Возможности направлениями дальнейшей работы являются, в порядке возрастания амбициозности:

- Верификация функции удаления элемента;
- Достижение работоспособной экстракции в OCaml или Haskell;
- Формализация доказательства допустимости множества параметров, предъявленного Лэем и Вудом;
- Точное определение множества допустимых параметров.

Список литературы

[1] Stephen Adams. “Functional pearls efficient sets—a balancing act”. В: *Journal of functional programming* 3.4 (1993), с. 553—561.

[2] Drake Baer. *Mark Zuckerberg Explains Why Facebook Doesn’t ’Move Fast And Break Things’ Anymore*. Май 2014. URL: `https://www.businessinsider.com/mark-zuckerberg-on-facebooks-new-motto-2014-5`.

[3] Lukas Barth и Dorothea Wagner. “Engineering top-down weight-balanced trees”. В: *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2020, с. 161—174.

[4] Lukas Barth и Dorothea Wagner. “Shaving peaks by augmenting the dependency graph”. В: *Proceedings of the Tenth ACM International Conference on Future Energy Systems*. 2019, с. 181—191.

[5] Lennart Beringer и др. “Verified Correctness and Security of OpenSSL {HMAC}”. В: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, с. 207—221.

[6] Yves Bertot и Pierre Castéran. *Interactive theorem proving and program development: Coq’Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[7] Frédéric Besson. “Fast reflexive arithmetic tactics the linear case and beyond”. В: *International Workshop on Types for Proofs and Programs*. Springer. 2006, с. 48—62.

[8] Norbert Blum и Kurt Mehlhorn. “On the average number of rebalancing operations in weight-balanced trees”. В: *Theoretical Computer Science* 11.3 (1980), с. 303—320.

[9] Mars Climate Orbiter Mishap Investigation Board. *Phase I Report*. 1999.

[10] Cristiano Calcagno и др. “Moving Fast with Software Verification”. В: *NASA Formal Methods*. Под ред. Klaus Havelund, Gerard Holzmann и Rajeev Joshi. Cham: Springer International Publishing, 2015, с. 3—11. ISBN: 978-3-319-17524-9.

[11] Herman Geuvers, Freek Wiedijk и Jan Zwanenburg. “A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals”. В: *Types for Proofs and Programs*. Под ред. Paul Callaghan и др. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, с. 96—111. ISBN: 978-3-540-45842-5.

[12] Yoichi Hirai и Kazuhiko Yamamoto. “Balancing weight-balanced trees”. В: *Journal of Functional Programming* 21.3 (2011), с. 287—307.

[13] Tony W Lai и Derick Wood. “A top-down updating algorithm for weight-balanced trees”. В: *International Journal of Foundations of Computer Science* 4.04 (1993), с. 309—324.

[14] Xavier Leroy. “Formal verification of a realistic compiler”. В: *Communications of the ACM* 52.7 (2009), с. 107—115.

[15] Nancy G Leveson и Clark S Turner. “An investigation of the Therac-25 accidents”. В: *Computer* 26.7 (1993), с. 18—41.

[16] Jürg Nievergelt и Edward M Reingold. “Binary search trees of bounded balance”. В: *SIAM journal on Computing* 2.1 (1973), с. 33—43.

[17] Tobias Nipkow и Stefan Dirix. “Weight-Balanced Trees”. В: *Archive of Formal Proofs* (март 2018). `https://isa-afp.org/entries/Weight_Balanced_Trees.html`, Formal proof development. ISSN: 2150-914x.

[18] Salvador Roura. “A new method for balancing binary search trees”. В: *International Colloquium on Automata, Languages, and Programming*. Springer. 2001, с. 469—480.

[19] Coq development team. *The Coq proof assistant*. 1989—2021. URL: `http://coq.inria.fr/`.

[20] Георгий Максимович Адельсон-Вельский и Евгений Михайлович Ландис. “Один алгоритм организации информации”. В: *Доклады Академии Наук*. Т. 146. 2. Российская академия наук. 1962, с. 263—266.