Правительство Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Полубелова Марина Игоревна

Использование сертифицированных библиотек в несертифицированных системах

Курсовая работа

Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
магистр информационных технологий, ст. преп. Григорьев С. В.

Санкт-Петербург
2015

Оглавление

Введение

Сертификационное программирование — направление, в котором при разработке программы используются формальные описания и доказательства ее полезных свойств. Появилось данное направление в связи с развитием средств для интерактивного доказательства и автоматической проверки теорем (proof assistants). В качестве примеров proof assistants можно привести Coq [8], Agda [7], F* [2], Idris [11] и Epigram [9]. В данной работе используется инструмент F*, так как он является единственным инструментом [5], который позволяет одновременно писать программы и их доказывать.

Области, в которых применяется сертификационное программирование, в основном связаны с обеспечением безопасного доступа к данным. Классическими примерами являются обмен сообщениями с помощью криптографических протоколов и распределенные системы, для которых нужно гарантировать, что данные доступны пользователям согласно их привилегиям, а также не нарушаются свойства самих данных (например, файл доступен только для чтения, а над ним выполняется операция записи). Однако доказательство корректности программы требуется не только в рассмотренных областях, но и при разработки остальных алгоритмов. Использование такого подхода увеличит надежность разрабатываемых систем и уменьшит трудозатраты на тестирование.

Системы, обеспечивающие интерактивное доказательство теорем, позволяют доказывать корректность алгоритма только в рамках своей системы. При извлечении проверифицированного кода в другой язык программирования теряется смысл верификации: либо отсутствуют некоторые необходимые проверки, либо подмножество исходного языка не может быть выражено в терминах целевого языка (например, кванторы существования и всеобщности). Даже если написать программу полностью на языке, который использует верификатор, то на практике часто требуется взаимодействие с кодом, который никак не проверифицирован.

Рассмотрим пример, демонстрирующий одну из указанных проблем. В листинге 1 доказана корректность функции `f` с помощью инструмента F* [2]. Ключевое слово `val` используется для определения типов и эффектов функции `f`; ключевое слово `Tot` означает, что вычисленный результат всегда имеет тип int без каких-либо эффектов вида: вхождение в бесконечный цикл, возникновение исключений (exceptions), взаимодействие с вводом или выводом и другие. После извлечения функции `f`, написанной на F*, в код на F# [13], представленный в листинге 2, потеряны некоторые свойства функции `f`, а именно, что она имеет эффект `Tot`. Теперь компилятор языка F# позволит вызвать эту функцию с некорректными, с точки зрения F*, параметрами, например, указанными в листинге 3. Хотя с точки зрения языка F# все будет правильно: для функции `g` компилятор выведет тип `int -> int`, несмотря на то, что

в ветке с `else` происходит вызов исключения.

```
val f : g : (int -> Tot int) -> int -> Tot int
let f g x = g x
```

Листинг 1: Пример проверифицированной функции в F*

```
let f : (int -> int) -> int -> int =
          (fun (g : int -> int) (x : int) -> g x)
```

Листинг 2: Код на F#, полученный из листинга 1

```
f (fun x: int -> if x > 0
                 then x + 1
                 else failwith "Expected non negative number!" ) 5
```

Листинг 3: Пример вызова функции f

В данной работе рассматривается задача извлечения проверифицированного кода в целевой язык программирования с сохранением его надежности. Для этого необходимо выделить подмножество языка F*, которое можно выразить с помощью конструкций языка F#. При этом генерация всех динамических проверок, например, на соответствие структуре данных, значительно снизит производительность системы, поэтому необходимо решение, которое будет удовлетворять не только требованиям надежности, но и производительности.

1. Постановка задачи

Целью данной работы является обеспечение надежного использования сертифицированных библиотек в несертифицированных системах. Для её достижения были поставлены следующие задачи:

- проверифицировать структуру данных и операции над ней;
- из кода, проверифицированного с помощью инструмента F*, получить код, написанный на языке программирования F#;
- проанализировать получившийся код;
- предложить решение обнаруженных проблем.

2. Обзор

В данной работе для верификации программ выбран функциональный язык программирования F* [2, 5], который используется как proof assistant и как язык общего назначения, ориентированный на верификацию. Его система типов включает в себя: полиморфизм, зависимые типы, monadic effect, refinement types и вычисление слабейших предусловий [4]. Указанные свойства позволяют написать спецификацию программы точно и компактно [10]. Для доказательства того, что программа соответствует спецификации, система проверки типов инструмента F* использует SMT solving (Z3 [12]) и свойства, написанные пользователем. Проверифицированный код можно извлечь в языки программирования F# и OCaml.

Кроме того, что язык F* является единственным инструментом [5], который позволяет одновременно писать программы и их доказывать. Данный инструмент обладает еще рядом отличительных свойств.

- Поддержка примитивных эффектов, таких как состояние, исключения, расходящиеся функции и ввод/ввывод:
  - `Tot t` — гарантирует, что вычисленный результат имеет тип `t` без каких-либо эффектов вида: вхождение в бесконечный цикл, возникновение исключений, взаимодействие с вводом или выводом и другие;
  - `ML t` — означает, что терм `t` может иметь произвольный эффект (это может быть циклом, изменение кучи, вызов исключения и другие);
  - `Dv t` — вычисления могут расходиться;
  - `ST t` — вычисления могут расходиться, происходят операции чтения, записи или выделение новых ссылок в куче;
  - `Exn t` — вычисления могут расходиться или происходит вызов исключений.

  Эффекты `{Tot, ML, Dv, ST, Exn}` образуют решетку, в которой `Tot` расположен внизу, `ML` — наверху, а `ST` никак не связан с `Exn`.

- Способом доказательства завершаемости рекурсивных функций: значения аргументов функций должны убывать в лексикографическом порядке. Язык так же позволяет указывать параметр, который соответствует завершаемости алгоритма, с помощью ключевого слова `decreases`.

- Написание спецификации программы с помощью зависимых и уточняющих типов.
  - Зависимые типы имеют следующую структуру:
    $x_1 : t_1 \to \ldots \to x_n : t_n[x_1 \ldots x_{n-1}] \to E\ t[x_1 \ldots x_n]$.

Нотация $t[x_1 \dots x_n]$ обозначает, что переменные $x_1 \dots x_n$ могут свободно входить в аргумент t. E содержит эффект вычисления тела функции.

– Уточняющие (refinement) типы имеют следующий вид:
$x : t\{\phi(x)\}$ — добавляется ограничение к типу t, а именно, элементы x должны удовлетворять предикату $\phi(x)$.

3. Основная часть

3.1. Верификация библиотеки dList

Структура данных dList [3] — это вариант представления структуры данных List, благодаря которому конкатенация списков возможна за константное время.

Описание структуры dList на языке F* выглядит следующим образом:

```
type dList 't =
    | Nil : dList 't
    | Unit : 't -> dList 't
    | Join : dList 't -> dList 't  -> nat -> dList 't
```

В dList используется вспомогательный параметр типа `nat` (алиас для неотрицательных целых чисел) для хранения длины списка.

В данной работе для рассматриваемой структуры данных проверифицированы следующие операции:

- *head* — возвращает первый элемент списка;
- *tail* — возвращает список без первого элемента;
- *length* — возвращает длину списка;
- *append* — возвращает результат конкатенации двух списков;
- *cons* — прибавляет элемент к началу списка;
- *snoc* — прибавляет элемент к концу списка;
- *fold* — свертка списка.

Уже доказательство корректности первой операции требует изменения в определении структуры данных dList: необходимо проверять, является ли пустым один из аргументов `Join`:

```
val isCorrectJoined : l : dList1 't -> Tot bool
let rec isCorrectJoined l =
        match l with
        | Nil -> true
        | Unit x -> true
        | Join Nil _ _ -> false
        | Join x y l -> isCorrectJoined x && isCorrectJoined y
```

```
type dList1 't = l : dList 't {isCorrectJoined l}
```

В дальнейшем используется последнее определение для структуры dList. Доказательства корректности операций представлены ниже.

- Для операции *head* в случае, когда аргумент является пустым списком `Nil`, результат не определен. Для доказательства этот факт можно интерпретировать по-разному, здесь же считается, что операция *head* всегда применяется к листу, состоящего хотя бы из одного элемента.

```
val isCons: dList 'a -> Tot bool
let isCons xs =
    match xs with
    | Unit x -> true
    | Join x y z -> true
    | _ -> false
```

```
val head : l : dList 't {isCons l} -> Tot 't
let rec head l =
    match l with
    | Unit x -> x
    | Join x y _ -> head x
```

- Для операции *append* обычно проверяют, что длина результирующего списка равна сумме длин исходных списков.
- Для операции *length* обычно проверяют, что результат является неотрицательным числом.
- Для операции *tail* можно доказать уже некоторые свойства. Например, конкатенация результата операции *head* с результатом операции *tail* есть исходный список, то есть `append (head l) (tail l) = l`.

```
val tail : l:dList 't -> Tot(l1:dList 't
{ (length l = 0 ==> length l1 = 0) \/
  (length l > 0 ==> length l1 = length l - 1 /\ append (Unit (head l)) l1 = l)})
let tail l = if length l = 0 then Nil else step l Nil
```

```
val step: xs:dList 't -> acc:dList 't -> Tot(l: dList 't)
let rec step xs acc =
    match xs with
    | Nil -> acc
    | Unit _ -> acc
    | Join x y _ ->
        step x (match y with
                | Nil -> acc
                | _ -> match acc with
                       | Nil -> y
                       | _ -> Join y acc (length y + length acc))
```

- Для операции *cons* тоже доказываются некоторые свойства:
  - результат содержит хотя бы один элемент;
  - первым элементом результирующего списка является добавленный элемент;
  - хвостом результирующего списка является исходный список;
  - полученный результат можно получить и другим способом: с применением операции *append*;
  - длина списка увеличилась на 1.

```
val cons: hd : 't -> l: dList 't -> Tot(l1 : dList 't
{isCons l1 /\ head l1 = hd /\ tail l1 = l /\
l1 = append (Unit hd) l /\ length l + 1 = length l1})
let cons hd l =
    match l with
    | Nil -> Unit hd
    | _ -> Join (Unit hd)  l (length l + 1)
```

- Доказанные свойства операции *snoc* совпадают с некоторыми свойствами операции *cons*:

```
val snoc: tl : 't -> l: dList 't -> Tot(l1 : dList 't
{isCons l1 /\ l1 = append l (Unit tl) /\ length l + 1 = length l1})
let snoc tl l =
    match l with
    | Nil -> Unit tl
    | _ -> Join l (Unit tl) (length l + 1)
```

- Для операции *fold* используется реализация[1], которая имеет линейную сложность. Одной из особенностью инструмента F* является способ доказательства завершаемости рекурсивной функции, а именно, проверяется, что значения аргументов функции уменьшаются в лексикографическом порядке. Однако можно явно указывать, какой параметр соответствует завершаемости алгоритма. Здесь таким параметром является количество "связующих" элементов списка (`Nil`, `Unit`, `Join`). Функция `cntdList` возвращает указанный параметр для списка типа `dList`, а функция `cntList` — для обычного списка, каждый элемент которого является список типа `dList`.

```
val fold : ('a -> 't -> Tot 'a) -> 'a -> dList 't -> Tot 'a
let fold f state l = walk [] l state f

val finish : rights : list (dList1 't) -> xs : 'a -> f : ('a -> 't -> Tot 'a)
-> Tot 'a  (decreases %[cntList rights; 1])
val walk : rights : list (dList1 't)  -> l : dList1 't -> xs : 'a
-> f : ('a -> 't -> Tot 'a)  -> Tot 'a (decreases %[cntdList l + cntList rights; 0])
let rec walk rights l xs f =
    match l with
    | Nil         -> finish rights xs f
    | Unit x      -> finish rights (f xs x) f
    | Join x y _  -> walk (y::rights) x xs f
and finish rights xs f =
    match rights with
    | []     -> xs
    | hd::tl -> walk tl hd xs f
```

3.2. Кодогенерация из F* в F#

Инструмент F* позволяет извлекать код в языки программирования F# и OCaml. В данном разделе рассматриваются проблемы, которые возникают в целевом коде, на примере проверифицируемой структуры данных dList.

В предыдущем разделе были описаны два типа для структуры данных dList: один без ограничений, другой с ними. В целевом коде эта информация не сохранилась:

[1]Реализация взята с сайта `http://stackoverflow.com/questions/5324623/functional-o1-append-and-on-iteration-from-first-element-list-data-structure/5334068#5334068`

```
type 't dList =
| Nil
| Unit of 't
| Join of 't dList1 * 't dList1 * Prims.nat

type 't dList1 = 't dList
```

Аналогично и с функциями: ограничения на уровне типов никак не отразились в целевой код. Например, корректность функции `head` была доказана в предположении, что она применяется только к спискам, состоящим хотя бы из одного элемента. Компилятор языка F# укажет только на не полный pattern matching.

```
let rec head = (fun ( l  :  't dList ) -> (match (l) with
| Unit (x) -> begin
x
end
| Join (x, y, _5_83) -> begin
(head x)
end))
```

Одним из возможных решений указанных проблем является генерация всех динамических проверок. С одной стороны, такой подход нельзя осуществить полностью, так как часть языковых конструкций F* не выразима в терминах языка F#. С другой стороны, динамическая проверка на корректность выполнения каждой операции значительно снизит производительность всей программы. Существующие подходы к решению данной проблемы в других proof assistants заключаются либо в расширении синтаксиса целевого языка [1], либо в использовании особенностей исходного языка [6] (например, аксиоматический подход для реализации смешанных вычислений в Coq).

Заключение

При выполнении данной работы были получены следующие результаты:

- описана на F* и проверифицирована структура данных dList и операции над ней;
- проанализирован код на F#, извлеченный из соответствующего кода на F*;
- выявлены проблемы с надежностью кода, извлеченного из проверифицированного кода, из-за отсутствия динамических проверок на входные данные, в предположении которых выполнялись доказательства корректности программы;
- рассмотрены существующие подходы к решению выявленных проблем.

Дальнейшее направление

Дальнейшем направлением данной работы является реализация подхода, обеспечивающего надежное использование сертифицированных библиотек в несертифицированных системах. Для этого необходимо выделить подмножество языка F*, которое можно выразить в терминах языка F#. При этом необходимо сообщать пользователю, какая часть кода является надежной, а какая нет.

Список литературы

[1] Chen Juan, Chugh Ravi, Swamy Nikhil. Type-preserving Compilation of End-to-end Verification of Security Enforcement // SIGPLAN Not. — 2010. — Vol. 45, no. 6. — P. 412–423. — URL: http://doi.acm.org/10.1145/1809028.1806643.

[2] F* Tutorial. — URL: https://www.fstar-lang.org/tutorial/.

[3] Hughes R J M. A Novel Representation of Lists and Its Application to the Function "Reverse" // Inf. Process. Lett. — 1986. — Vol. 22, no. 3. — P. 141–144. — URL: http://dx.doi.org/10.1016/0020-0190(86)90059-1.

[4] Pierce Benjamin C. Types and Programming Languages. — Cambridge, MA, USA : MIT Press, 2002.

[5] Swamy Nikhil, Hriţcu Cătălin, Keller Chantal. Dependent Types and Multi-Monadic Effects in F* // 43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). — ACM, 2016. — P. 256–270. — URL: https://www.fstar-lang.org/papers/mumon/.

[6] Tanter Éric, Tabareau Nicolas. Gradual Certified Programming in Coq. — 2015. — P. 26–40. — URL: http://doi.acm.org/10.1145/2816707.2816710.

[7] Сайт проекта Agda. — URL: http://wiki.portal.chalmers.se/agda/pmwiki.php.

[8] Сайт проекта Coq. — URL: https://coq.inria.fr/.

[9] Сайт проекта Epigram. — URL: https://github.com/mietek/epigram2.

[10] Сайт проекта F*. — URL: https://www.fstar-lang.org/.

[11] Сайт проекта Idris. — URL: http://www.idris-lang.org/.

[12] Сайт проекта Z3. — URL: http://z3.codeplex.com/.

[13] Язык программирования F#. — URL: http://fsharp.org/.