Lecture 7
Generate-Test-Aggregate in Coq
NII Lectures Series

Frédéric Loulergue

Université d’Orléans – LIFO – PaMDA Team

October-November 2013
Outline

1. Introduction
2. GTA in Coq
3. Summary
Generate-Test-Aggregate

User’s perspective:
- **G**: Generator to produce all candidates
- **T**: Tester for validity check of candidates
- **A**: Aggregator to build the final result

Optimization:
- Filter embedding: $GTA \Rightarrow GA$
- Semiring Fusion: $GA \Rightarrow D&C$

Papers: [1, 2]
The Knapsack problem

Given:
- a knapsack that could contain items for at most a volume \( v \)
- a set of items each characterised by a value and a volume

Find the most valuable selection of items

Items and knapsack

- pens 🖋️: ¥200, 1\(dl\)
- scissorsissors: ¥350, 1\(dl\)
- phone 📱: ¥5000, 2\(dl\)

Knapsack: 3\(dl\)
Choose an adequate generator

A generator produces:

- a multi-set of candidates
- it should generate all **necessary** candidates
- it could generate **invalid** candidates

The Knapsack problem

- $G$ generates all the subsets of the set of items
- $G \{\text{item 1}, \text{item 2}, \text{item 3}\} = $
- $\{}$, $\{\text{item 1}\}$, $\{\text{item 2}\}$, $\{\text{item 3}\}$, $\{\text{item 1}, \text{item 2}\}$, $\{\text{item 1}, \text{item 3}\}$, $\{\text{item 2}, \text{item 3}\}$
Choose an adequate tester

- it removes invalid items according to a predicate
- it should be a homomorphism

The Knapsack problem

- \( T \) is based on the predicate:

\[
p \equiv s \mapsto \sum_{i \in s} \text{volume} \leq 3
\]
For better performances

\[ p \equiv s \mapsto \bigoplus_{i \in s} volume'(i) \leq 3 \]

where
- \[ volume'(i) = \max \ volume(i) (3 + 1) \]
- \[ x \oplus y = \max (x + y) (3 + 1) \]
Choose an adequate aggregator
It produces a summary of the valid candidates

The Knapsack problem
Examples:

▶ the best candidate
▶ the two first best candidates
▶ the number of candidates
▶ …

\[ \{\}, \{\text{ceptor}\}, \{\text{ceptor} \rightarrow \text{ceptor}\}, \{\text{ceptor} \rightarrow \text{ceptor}\} \cup \{\text{ceptor}, \text{ceptor}\} = (\{\text{ceptor}, \text{ceptor}\}, 5350\text{¥}) \]
The naive program \texttt{GTA} has complexity $O(2^n n)$

with GTA framework: $O(n)$

Two transformations:

- Filter embedding: \texttt{GTA} $\Rightarrow$ \texttt{GA}

- Semiring Fusion: \texttt{GA} $\Rightarrow$ \texttt{D&C}
A verified version of GTA

The Goal

▶ to allow the user to write GTA programs
▶ type class mechanism to automatically derive an efficient
divide-and-conquer program
▶ prove the fusion theorems in Coq
▶ extraction towards:
  ▶ OCaml + BSML
  ▶ Scala + Spark

Joint work with

▶ Kento Emoto (Kyushu University of Technology)
▶ Julien Tesson (LACL, Université Paris-Est Créteil)
▶ Frédéric Dabrowski (LIFO, Université d’Orléans)
Status

It is an ongoing work!

- ✓ to allow the user to write GTA programs
- ✓ type class mechanism to automatically perform semiring fusion and filter embedding fusion
- ✗ prove the fusion theorems in Coq
- ✗ type class mechanism to automatically derive parallel BSML program from a GTA specification
DEM0
GTA in Coq

We had to define data-structures: for example Bags.

▶ multiset in Coq standard library is not abstract, it is a specific implementation using functions to represent multisets
▶ the multiset axiomatisation in the CoLoR library is nice but does not use type classes

How to axiomatise:

▶ try to find a minimal axiomatisation so that its realisation will not be too complicated
▶ then additional results are derived from this axiomatisation
Automatic parallelisation

Basically we need:

- an automatic way to derive parallel versions of homomorphisms
- but in a generalised setting with user-defined equivalence relations rather than Coq $=$ equality

$\Rightarrow$ ongoing work
Proof of the fusion theorem

Main Theorem 3 (Filter-embedding Semiring Fusion). Given a set $A$, a finite monoid $(M, \circ)$, a monoid homomorphism $\text{hom}$ from $([A], +)$ to $(M, \circ)$, a semiring $(S, \oplus, \otimes)$, a semiring homomorphism $\text{aggregate}$ from $([A], \sqcup, \times)$ to $(S, \oplus, \otimes)$, a function $\text{ok} : M \to \text{Bool}$, and a polymorphic semiring generator $\text{generate}$, the following equation holds:

$$\text{aggregate} \circ \text{filter} (\text{ok} \circ \text{hom}) \circ \text{generate}_{\sqcup, \times} (\lambda x \to [x])$$

$$= \text{postprocess}_M \text{ok} \circ \text{generate}_{\oplus_M, \otimes_M} (\lambda x \to \text{aggregate}_M [x])$$

Proof. Combining Theorems 1 and 2.

Filter-embedding Semiring Fusion is not restricted to parallel algorithms. It can be used to calculate efficient programs from specifications that use arbitrary polymorphic semiring generators.

It is worth noting that it is possible to remove the finiteness requirement for monoids and define a lifted semiring of finite mappings of unbounded and unknown size. We require the finiteness only in order to be able to describe the complexity of the resulting parallel algorithms more accurately.
Main Theorem 3 (Filter-embedding Semiring Fusion). Given a set $A$, a finite monoid $(M, \odot)$, a monoid homomorphism $\text{hom}$ from $([A], \oplus)$ to $(M, \odot)$, a semiring $(S, \oplus, \otimes)$, a semiring homomorphism $\text{aggregate}$ from $(\uplus ([A]), \cup, \times_\uplus)$ to $(S, \oplus, \otimes)$, a function $\text{ok} : M \rightarrow \text{Bool}$, and a polymorphic semiring generator $\text{generate}$, the following equation holds:

$$\text{aggregate} \circ \text{filter} (\text{ok} \circ \text{hom}) \circ \text{generate}_{\cup, \times_\uplus} (\lambda x \rightarrow \uplus ([x])) = \text{postprocess}_M \text{ok} \circ \text{generate}_{\oplus_M, \otimes_M} (\lambda x \rightarrow \text{aggregate}_M \uplus ([x]))$$

Proof. Combining Theorems 1 and 2.

Filter-embedding Semiring Fusion is not restricted to parallel algorithms. It can be used to calculate efficient programs from specifications that use arbitrary polymorphic semiring generators.

It is worth noting that it is possible to remove the finiteness requirement for monoids and define a lifted semiring of finite mappings of unbounded and unknown size. We require the finiteness only in order to be able to describe the complexity of the resulting parallel algorithms more accurately.
Proof of the fusion theorem

Currently: finiteness condition removed, so

- need to deal with a Map data-structure instead of tuples
- prove of associativity and distributivity for the semi-ring are difficult and not yet done
- the user is may be not aware that to obtain an efficient version, finitness optimisation is necessary

We will consider another version: \( M \) is finite
Finite and tuples with dependent types?

DEMO
Summary

GTA framework:
- usable for writing and deriving sequential functions
- soon able to derive parallel BSML versions
- not yet completely dependable (admits) ... but hopefully soon

In the future:
- scala extraction for Cloud Computing
- additional examples and library of generators/testers
- ...
Thank you!

Coming soon

- https://traclifo.univ-orleans.fr/SYDPACC
- packages for the OPAM (OCaml) package manager