Lecture 13
Deductive and Transformational Synthesis

Armando Solar-Lezama and Xiaokang Qiu
with some slides from Adam Chlipala
A Transformation System for Developing Recursive Programs

R.M. Burstall and John Darlington 1977
“We make no claim for any sort of completeness of the system; it embodies only one family of program transformations, and we have no formal delineation of this family. ... However, we hope that the examples will give the reader pleasure...”
Key premise

(1) Start with a clean high-level representation of the problem

(2) Apply semantics preserving transformations

(3) Arrive at a less clean but more efficient representation
Clean high-level spec

\[
\begin{align*}
dot(x, y, 0) & \leftarrow 0 \\
dot(x, y, n + 1) & \leftarrow \dot(x, y, n) + x[n + 1]y[n + 1]
\end{align*}
\]

\[
\begin{align*}
fib(0) & \leftarrow 0 \\
fib(1) & \leftarrow 1 \\
fib(x + 2) & \leftarrow fib(x + 1) + fib(x)
\end{align*}
\]

\[
\begin{align*}
concat(nil, z) & \leftarrow z \\
concat(cons(x, y), z) & \leftarrow cons(x, concat(y, z))
\end{align*}
\]
Example

\[dot(x, y, 0) \Leftarrow 0\]
\[dot(x, y, n + 1) \Leftarrow dot(x, y, n) + x[n + 1]y[n + 1]\]

\[f(a, b, c, d, n) \Leftarrow dot(a, b, n) + dot(c, d, n)\]
Key transformation rules

Instantiation

- \( f(a, b, c, d, n) \leftarrow \text{dot}(a, b, n) + \text{dot}(c, d, n) \)
- \( f(a, b, c, d, 0) \leftarrow \text{dot}(a, b, 0) + \text{dot}(c, d, 0) \)
- \( f(a, b, c, d, n + 1) \leftarrow \text{dot}(a, b, n + 1) + \text{dot}(c, d, n + 1) \)

Unfolding

- \( f(a, b, c, d, n + 1) \leftarrow \text{dot}(a, b, n + 1) + \text{dot}(c, d, n + 1) \)
- \( f(a, b, c, d, n + 1) \leftarrow \text{dot}(a, b, n) + a[n + 1]b[n + 1] + \text{dot}(c, d, n) + c[n + 1]d[n + 1] \)

Folding

- \( f(a, b, c, d, n + 1) \leftarrow \text{dot}(a, b, n) + a[n + 1]b[n + 1] + \text{dot}(c, d, n) + c[n + 1]d[n + 1] \)
- \( f(a, b, c, d, n + 1) \leftarrow f(a, b, c, d, n) + a[n + 1]b[n + 1] + c[n + 1]d[n + 1] \)
Key transformation rules

Algebraic laws
• \( f(a, b, c, d, n + 1) \leftarrow \text{dot}(a, b, n) + a[n + 1]b[n + 1] + \text{dot}(c, d, n) + c[n + 1]d[n + 1] \)
• \( f(a, b, c, d, n + 1) \leftarrow \text{dot}(a, b, n) + \text{dot}(c, d, n) + a[n + 1]b[n + 1] + c[n + 1]d[n + 1] \)

Abstraction
• \( E[F_1 \rightarrow u_1, ... F_n \rightarrow u_n] \) where \( \langle u_1 ... u_n \rangle = \langle F_1, ..., F_n \rangle \)
Fibonacci

1. \( f(0) \triangleq 1 \)
given
2. \( f(1) \triangleq 1 \)
given
3. \( f(x + 2) \triangleq f(x + 1) + f(x) \)
given
4. \( g(x) \triangleq \langle f(x + 1), f(x) \rangle \)
definition (eureka)
5. \( g(0) \triangleq \langle f(1), f(0) \rangle \)
   \triangleq \langle 1, 1 \rangle 
   instantiation
   unfolding with 1 and 2
6. \( g(x + 1) \triangleq \langle f(x + 2), f(x + 1) \rangle \)
   \triangleq \langle f(x + 1) + f(x), f(x + 1) \rangle 
   \triangleq \langle u + v, u \rangle \text{ where } \langle u, v \rangle = \langle f(x + 1), f(x) \rangle 
   \triangleq \langle u + v, u \rangle \text{ where } \langle u, v \rangle = g(x) 
   instantiate 4
   unfold with 3
   abstract
   fold with 4
7. \( f(x + 2) \triangleq u + v \text{ where } \langle u, v \rangle = \langle f(x + 1), f(x) \rangle \)
   \triangleq u + v \text{ where } \langle u, v \rangle = g(x) 
   abstract 3
   fold with 4
Dreams $\Rightarrow$ Programs

Waldinger & Manna 1979
Deductive synthesis

Similar ideas to Burstall and Darlington with a solid theoretical footing

A lot more ambitious

“...Such a system accepts specifications that express the purpose of the program to be constructed, without giving any hint of the algorithm to be employed. With no further human intervention, the system attempts to transform these specifications into a program that achieves the expressed purpose”
Specifications

\[ \text{lessall}(x \ l) \iff \text{compute } x \lt \text{all}(l) \]
\[ \text{where } x \text{ is a number and } l \text{ is a list of numbers.} \]

\[ \text{gcd}(x \ y) \iff \text{compute max} \{z : z \mid x \text{ and } z \mid y\} \]
\[ \text{where } x \text{ and } y \text{ are nonnegative integers and } x \neq 0 \text{ or } y \neq 0. \]
Basic transformation rules

\[
t \Rightarrow t' \quad \text{if } P
\]

\[
\text{true and } Q \Rightarrow Q
\]

\[
u | v \Rightarrow \text{true} \quad \text{if } u \text{ is an integer and } v = 0
\]
Derivation

Goal 1: compute $x < all(l)$.

Rule 1: $P(all(l)) \Rightarrow true$ if $l$ is the empty list

Goal 2: prove $l$ is the empty list. $\times$

Rule 2: Conditional formation

$S$ where $Q$ \Rightarrow if $(P)$ $S$ where $P$ and $Q$
else $S$ where $\neg P$ and $Q$

$lessall(x, l) \iff$ compute $x < all(l)$

where $x$ is a number and

$l$ is a list of numbers.

lesall(x, l) \iff if empty(l) compute x < all(l) where l is empty
else compute x < all(l) where l is not empty
Derivation

Rule 1: \( P(\text{all}(l)) \Rightarrow \text{true} \) if \( l \) is the empty list

Rule 3: \( P(\text{all}(l)) \Rightarrow P(\text{head}(l)) \) and \( P(\text{all}(\text{tail}(l))) \)
if \( l \) is a nonempty list.

Goal 4: compute \( x < \text{head}(l) \) and \( x < \text{all}(\text{tail}(l)) \).

Rule 4: Given \( F(x) \Rightarrow \) compute \( P(x) \) when \( Q \)
\( P(t) \Rightarrow F(t) \) if \( Q(t) \) and \( F(t) \) terminates

\[ \text{lesall}(x, l) \Leftarrow \text{if empty}(l) \text{ compute } x < \text{all}(l) \text{ where } l \text{ is empty} \]
\[ \text{else compute } x < \text{all}(l) \text{ where } l \text{ is not empty} \]

\[ \text{lesall}(x, l) \Leftarrow \text{if empty}(l) \text{ true} \]
\[ \text{else compute } x < \text{all}(l) \text{ where } l \text{ is not empty} \]
Challenges

Proving that conditions are satisfied could be non-trivial

Transformation rules can be very domain-specific and you need a lot of them

Knowing what to apply when can be challenging
  • Very large space of possible derivations
Modern incarnations

Synthesis modulo recursive functions
  • Etienne Kneuss, Ivan Kuraj, Viktor Kuncak and Philippe Suter

Key idea: Apply deductive rules to simplify the problem
  • Fall back on CEGIS to solve the more restricted problems
Modern incarnations

Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant
   • Benjamin Delaware, Clément Pit--Claudel, Jason Gross, Adam Chlipala

Rely on modern proof assistant to help automate routine parts of the derivation

Lift the level of abstraction of transformation rules so they are easier to apply
Fiat: A Program Derivation Framework in Coq

Program synthesis by stepwise refinement

1. Write mathematical specifications without worrying about performance.
2. Apply **optimization scripts** to refine specifications into efficient programs.
3. Coq’s usual proof checker validates that scripts have preserved semantics.

Big delta from past related work:
“high automation without giving up high assurance”
Step 1: Specification

query "NumOrders" (author : string) : nat :=
  Count (For (o in "Orders") (b in "Books")
    Where (author = b!"Author")
    Where (b!"ISBN" = o!"ISBN")
  Return ()

Notes:
- Example of an SQL-style query
- Similar to monadic query comprehension notation.
- Parsed & type-checked within normal Coq code (extensible parser, etc.)
Step 2: Optimization Script
Choose data structures:

Definition BookStorage : IndexFor Book.

Definition OrderStorage : IndexFor Order.

Define abstraction relation connecting them to specification:

Definition Bookstore_AbsR := ...

Call library tactic to implement specs in terms of this relation:

plan Bookstore_AbsR.
Step 3: Reasonable Code!

meth "NumOrders" (p : rep, s : string) : nat :=
  ret (p, fold_left
    (fun (count : nat) (x : Tuple) =>
      count + bcount (snd p) (Some x!"ISBN", [])
    (bfind (fst p) (Some s, (None, []))) 0)

Calls to methods of library data structures (nested AVL trees)
Modern Transformation systems

Spiral
- Led by Markus Puschel, Franz Franchetti and Jose Moura

Key ideas:
- Focus on a specific domain
- Build in heuristics and transformations that capture domain insights
- Throw a lot of computing power!
Example: DropThird

Many transformations required before you can code in C

Many choices have to be made along the way

Most optimal choices depend on wordsize, cache size, cache speed, etc.
The strategy

Start with a Synchronous Data Flow language to describe the task at a high level.

The same language can express the low level implementation
  • Make each actor correspond to a single low level operation.

Any sequence of transformations that transforms a HL program into an equivalent LL program encodes an implementation.
Default Implementation

Automatic Transform algorithm

Defines a default implementation for any program

Leveraged by the user for more involved implementations
The Full Picture

For every task there is a space of programs that describe it. The **Automatic Transform** algorithm defines a path from any program to a LL program.

To get a different implementation, define a **Halfway Transformation**, and let the automatic algorithm do the rest.

If the transformation is difficult, simply provide a **Sketch**.
Half way transformations- example

System Expert provides high level decomposition
System Takes care of Lowering F.F_1, F.F_2 and F.F_3
Correctness is guaranteed as long as

\[(F.F_3) \times (F.F_2) \times (F.F_1) = F\]

Fully Specifying F.F._1, F.F._2 and F.F._3 is still too difficult. We would like to be able to sketch them
Sketching

An implementation **sketch**:
- Specifies a number of steps in the implementation of a task
- Implicitly defines a matrix decomposition
- Gives constraints on what each step is allowed to do
- System derives an implementation satisfying the constraints and semantically equivalent to original filter

Example. A fragment of TSL for FAST compaction:

```
filter = [shift(1:16 by 0 || 1)] ;
[shift(1:16 by 0 || 2)] ;
[shift(1:16 by 0 || 4)] ;
```