Derivation of Program Properties During Generation

Christoph Herrmann

IFIP WG 2.11 Meeting
Halmstad/Sweden, June 25, 2012

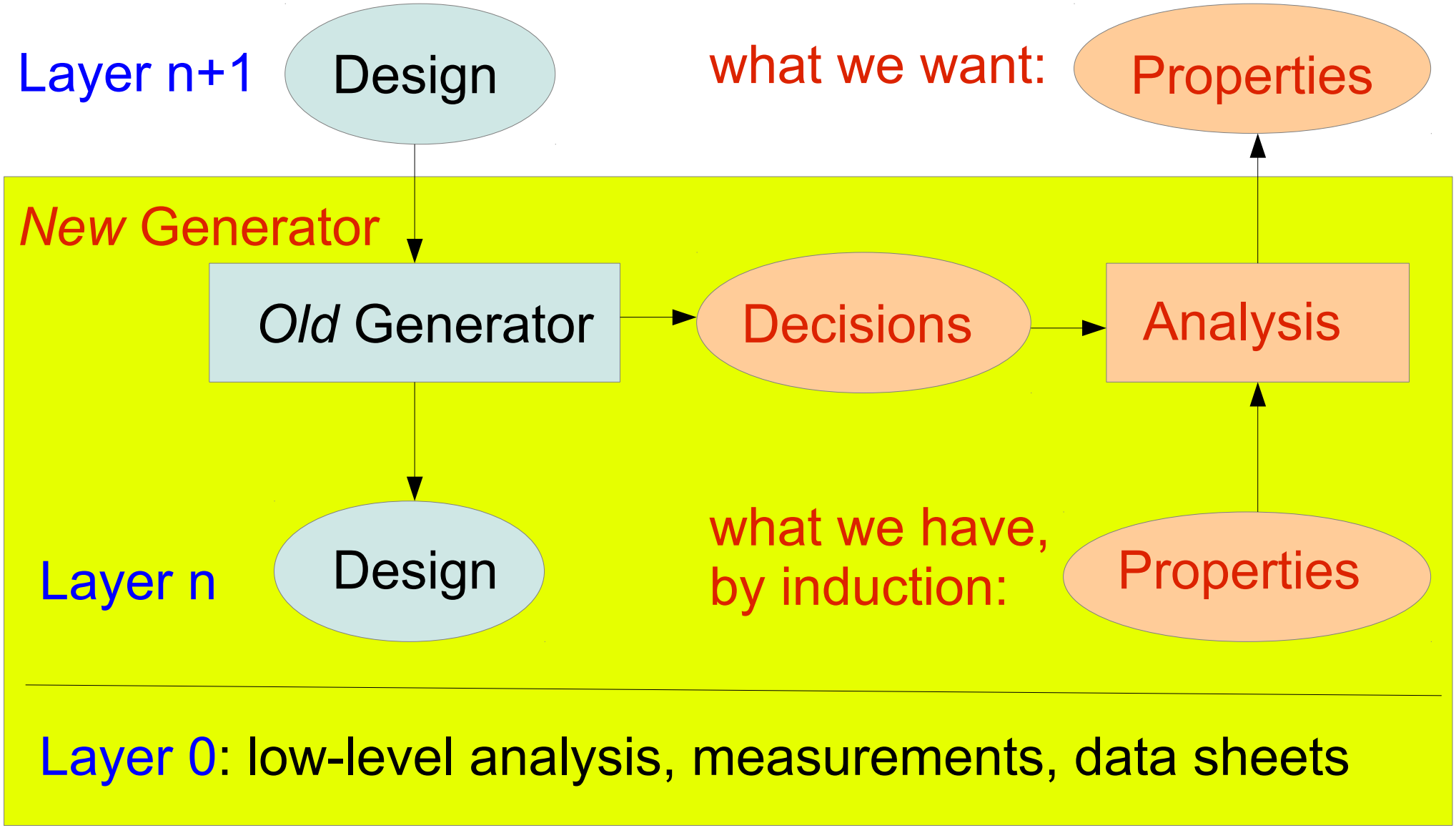
Aims / Motivation

- automatic derivation of program/system properties
  - small changes can have large effects
  - human interaction kills productivity and introduces errors
- properties are parameterized
  - because programs are parametrized
  - fight limits of decidability by patterns
- derivation during generation
  - rule out general cases that we do not need
  - enable more precise analysis results

Potential Application Areas

- autonomous and resource-critical systems
  - be sure memory space and energy are sufficient
  - reaction in time required
  - consider devices other than CPU as well
- business applications in the large scale
  - millions of business objects updated in few hours
- high-performance computing
  - as much parallelism as necessary, but
  - as little energy consumption as possible

Adding Analysis to Generation



Example Uses of Layers (1)

- autonomous vehicles
  - 3: strategic layer: describes global plan
  - 2: execution layer: coordinates actions of one vehicle
  - 1: operational layer: controls each unit, processes data
  - 0: assembly code
    - maybe fixed blocks or fixed loops (exploit pipelining)

Example Uses of Layers (2)

- relational database queries

  3: specification in a query language (SQL)

  2: collective data base operations (join, filter)

  1: imperative loop language (for, :=, +, if)

  0: assembly code (mov, add, cmp)

Example Uses of Layers (3)

- high-performance computing

  4: coordination language (HDC, Eden)

  3: algorithmic skeletons (HOF divide-and-conquer)

  2: architectural skeletons (MPI map, MPI reduce)

  1: imperative program (C, Fortran)

  - skeleton implementation
  - compiled customizing functions

  0: assembly code

Problem with Existing Approaches: Lack of Productivity

- monolithic analyses
  - neglect at least low-level details (assembly code) or high-level structures (symbolic loop bounds)
  - require large amount of human interaction to annotate the program to help the analysis tools
  - human effort has to be repeated in case of changes
- dependently-typed languages
  - can transfer the properties between layers of abstraction (provided formalization of assembly code etc. given)
  - require large amount of human interaction to provide appropriate proofs

Industrial Practice

- flight control software
  - no loops with parameterized bounds
  - changes in the code: complete rerun of analysis
- business applications
  - OO design: no concept how to integrate properties
  - generation includes only trivial program logic
  - profiling in case of known resource problems
  - quality control: mainly testing, testing, testing (customers are not interested in formal correctness but that the system behaves as they expect)

Advantages of the Suggested Method: Coupling of Generation and Analysis

- no code is generated that cannot be analysed
- analysis benefits from the generator
  - in the design by exploiting expert knowledge
  - in the application by exploiting special cases
- manual effort must be spent only on the generator, not on the generated code (in contrast to use of typical analysis tools)
- no type connection between the code generator and the generated code, which makes writing generators more productive (at the loss of formal guarantees)

Ex. Choices: Data Base Query

- Case A: all fields available in one physical table
  - generate a loop and the analysis for the loop
- Case B: the information is distributed among several tables (due to normalization)
  - alternatives, e.g.:

    (a) first perform inner product, then do a single loop

    (b) collect the information with nested loops
  - generator can produce several implementations, provide an analysis for each and introduce a run-time selection depending on parameter values

Ex. Choices: Autonomous Systems

- different strategies to solve a task collectively
- each strategy has individual costs for memory, execution time, energy consumption
- generator produces for each strategy code and a formula for each property
- user of the system can then decide which strategy is best in a particular situation
  - by quickly evaluating the cost formula, or
  - by using an external solver

Ex. Choices: High-Perf. Computing

- where to duplicate work/data, where to reuse
- I/O: which data centralized, which distributed
- how to distribute data and computations
- parallel skeletons: provide given implementations and according cost formulae manually
- polytope model
  - automatically generates code which is optimal according to a cost function
  - specialized for certain kinds of programs (loops) and computational models (systolic arrays)

Variations of Automation

- meta-language with dependent types: structure of particular program fixed (reflected by proof)
- skeletons: program structure basically fixed, but unreflected choice between variations possible
- polytope model: only the class of programs is fixed, high variety of automatically producible code depending on objective function
- DSL-compilers: highly involved code generation, sometimes very restricted tasks (e.g. FFTW)

Ex. Knowledge: Loops

- E = [FOR i:=0 TO n DO A(i)]
  - time [A(i)] = c time(E) = (n+1)*c
  - time [A(i)] = i*c time(E) = (n+1)*n/2*c
  - space [A(i+1)] ≤ space [A(i)]

    space(E) = space(A(0)) + c
- different bounds and stride other than 1:
  - convert into form above by substituting bounds and argument of A before analysing the loop

Ex. Knowledge: Sequences

- E = [sort (sort xs)]
  - generate E = generate [sort xs]
  - time (generate E) = time (generate [sort xs])
- E = [mergesort xs]
  - time (generate E) ≤ c0 * (n * log2 n) + c1 * n + c2

    where n = length(xs)
- E = [map f (map g xs)]
  - E = [map (f ◌ g) xs]
  - in parallel setting avoid unnecessary gather and scatter and their communication times in the analysis

From Programs To Systems

- people might want to use DSLs to describe entire systems and generate code for them, not just single programs, e.g.:
  - cooperating autonomous systems
  - parallel processing
  - interacting business applications
- we add the effects of the devices we need to operate the system to the program
  - this is obvious if we consider a simulation (interpretation)
  - but generation is nothing but partial application of an interpreter
- a usual program analysis would neglect the effect of devices
  - but when controlling a vehicle the energy consumption for the CPU is less important than the one for the motor

Ex. Knowledge: Devices

- Vehicle

  energy(drive n meters) = energy(start) + n * c

- Camera

  time [take picture, take picture] = c + 2 * time [take picture]

  a composition is more expensive than the sum of the single actions

- Harddisk

  time(read block from hard disc)

  ≤ c1 * (endposition head - startposition head) + c2

It would be very difficult to derive such costs from a generated low-level program alone!

Analysis during Generation

- generation often comes with a kind of analysis
- analysis of generated code without exploiting knowledge of the generator is „stupid“, because it solves an artificial problem
- several different approaches exist
  - each has its merits at certain points
  - none alone is sufficient to cover all the needs
- a general way to integrate analysis and generation in the software design process would be useful

Embedded Meta-DSL?

- is it possible to have a nice set of (say Haskell) definitions
  - that can be instantiated for the mentioned domains,
  - to generate code we would like to have, including dependently typed or parallel code where desired,
  - to integrate analyses of different mathematical models and decision procedures?
- probably yes!
  - Haskell provides a good amount of formal rigor while flexible
  - the specification would not be part of the run time, so it would not impose particular execution models
- but if this is not accepted as basis for software design documents, then it will be practically useless

Ingredients of such a Meta-DSL

- user-definable domains
  - for programs
  - for mathematical models, e.g. the polytope model
- way to specify costs for Layer 0
- defining a language at Layer n+1 in terms of a language at Layer n, specifying a generator and an according analysis
- challenge: to provide a useful (infra-)structure by the Meta-DSL without restricting the applicability

How to Deal with Object-Orientation

- automatically generated OO classes are difficult to reuse safely, especially if generation is repeated after use with different parameters
- fortunately, use of inheritance is sometimes just for configuration, i.e. no object creation is necessary and the actual instance is fixed, i.e., can be replaced by a procedure
- where useful (simulation, real world mappings) generate classes, but do not inherit from them in other code
- analysis still difficult, but a problem of the person who writes the generator

Summary

- analysis can benefit from generator knowledge
- application domains develop isolated solutions
- there exists a large variety of domain knowledge that could be used explicitly (safely)
- it would be useful if analysis and generation would become an integral part of the software development process and the specification
- Haskell is a promising candidate as a Meta-Meta-DSL for formalization of software designs