Conclusion*

Andreas Lochbihler

Department of Computer Science
ETH Zurich

*Thanks to David Basin for slide material

Programming paradigms

- Haskell is a so-called pure functional programming language
  - It is "pure" in that functions have by default no side effects
  - Features list comprehension, higher-order functions, strong type system, algebraic data types, lazy evaluation, . . .
- Other FP languages: LISP, Scheme, SML, Ocaml, F#, Scala, . . .
  - None of their type systems guarantees side-effect freeness
- Other programming paradigms

  imperative: e.g., Pascal, C, and FORTRAN
  object-oriented: e.g., Smalltalk, Java, and C++
  logic: e.g., Prolog
- From computer scientists one can expect some "multi-linguality"

Programming languages: why study another one?

- All general-purpose programming languages are equivalent

  Formally: they are Turing complete and can express the same class of (partial, recursive) functions

- Although the same functions can be computed, algorithms are formalized differently

  Formalization as a RAM is different from formalization in Java, which is different from formalization in Haskell . . .

- Approaches have their pros and cons

  We summarize some of the important differences

Values versus states (I)

- A functional program describes values

```
sumSquares :: Int -> Int
sumSquares 0 = 0                          -- Direct description
sumSquares n = n*n + sumSquares (n-1)     --
```

Described by equations constituting the function definition.

Values versus states (II)

- An imperative program describes how memory is updated.

```
s = 0;
i = 0;
while (i < n) {
  i = i+1;
  s = i*i + s;
}
```

  - Consists of commands that update the content of memory.
  - Memory models primitive data.
  - No analogy to data-driven solutions where computation driven by need to generate data rather than explicitly by programmer.

```
sumSquares' n = sum (map (^2) [1 .. n])
```

Functions and variables

- Functional programs
  - Function returns value depending only on input

$$\begin{aligned} sumSquares\ n &= n * n + sumSquares\ (n-1) \\ sumSquares\ 6 &= 6 * 6 + sumSquares\ (6-1) \end{aligned}$$

  - Analogous to functions in mathematics

$$\begin{aligned} f(x,y) &= 2 \cdot x - y \\ f(6,3) &= 2 \cdot 6 - 3 \end{aligned}$$

- This referential transparency fails for imperative programs
- Critical difference: variables in functional programs do not vary!

Program verification (functional)

- Functional programs describe their properties

$$sumSquares\ 0 = 0$$

or for $n > 0$,

$$sumSquares\ n = n * n + sumSquares(n - 1)$$

- Proofs through equational reasoning or by induction

$$sumSquares\ 1 = 1 * 1 + sumSquares\ (1 - 1) = 1 + 0 = 1$$

Program verification (imperative)

- Imperative programs lack referential transparency

```
class test {
  static int y=0;

  static int f(int x) {
    y = y+1;
    return(y);  };

  public static void main (String args[]){
     System.out.println(f(0));
     System.out.println(f(0)); }
}
```

- Verification with respect to program points using pre-/postconditions describing sets of states
- It is (relatively) difficult to verify non-trivial programs

Compositionality

- Functional programs
  - Functions serve both as building blocks and as glue
  - Referential transparency enables compositional reasoning
- Imperative programs
  - Control-structures such as `if-then-else` are glue that construct commands from commands

```
if b then c1 else c2          while b do c
```

  - Glue is fixed!
  - Less compositional reasoning due to lack of referential transparency

Types and abstraction

- Functional programs
  - Strong type system $\Rightarrow$ fewer errors at runtime
  - Polymorphism and type classes (Haskell/ghc)
  - Abstraction over data and control
- Imperative programs
  - Typically a weaker type system $\Rightarrow$ more runtime errors
  - Abstraction over data through object orientation and generics
  - Cumbersome abstraction over control

Efficiency

- Functional programs
  - Programs are high-level and far from machine architecture, but referential transparency enables further optimizations.
  - Lazy evaluation more difficult to implement
- Imperative programs
  - Complexity is easier to assess and control
  - Closer to hardware level
  - Direct influence on representation of data in memory
- Haskell programs can often be optimized to $\leq 2 *$ speed of C
- Other factors are often more important than efficiency, e.g., developer productivity, correctness, safety (no buffer overflows), libraries, . . .

Haskell Outlook

Haskell usage

- Haskell has a thriving and exceptionally friendly user community

  Website: www.haskell.org
  Maling List: haskell-cafe@haskell.org
  Blogs: planet.haskell.org
  IRC: #haskell on freenode.net
  Reddit: http://www.reddit.com/r/haskell/
  Libraries: hackage.haskell.org

- Commercial users of functional programming
  - Website: cufp.org (includes a job board)
  - Credit Suisse: Modeling and analysis of financial products
  - Galois: High-assurance applications (e.g., for the US DoD)
  - Bluespec: Hardware design languages and automation

Haskell research

- Functional programming is an active research topic

  Conferences: ICFP, Haskell Symposium, IFL, POPL

  Abstraction: embedded languages, generic programming

  Concurrency: Data Parallel Haskell, STM

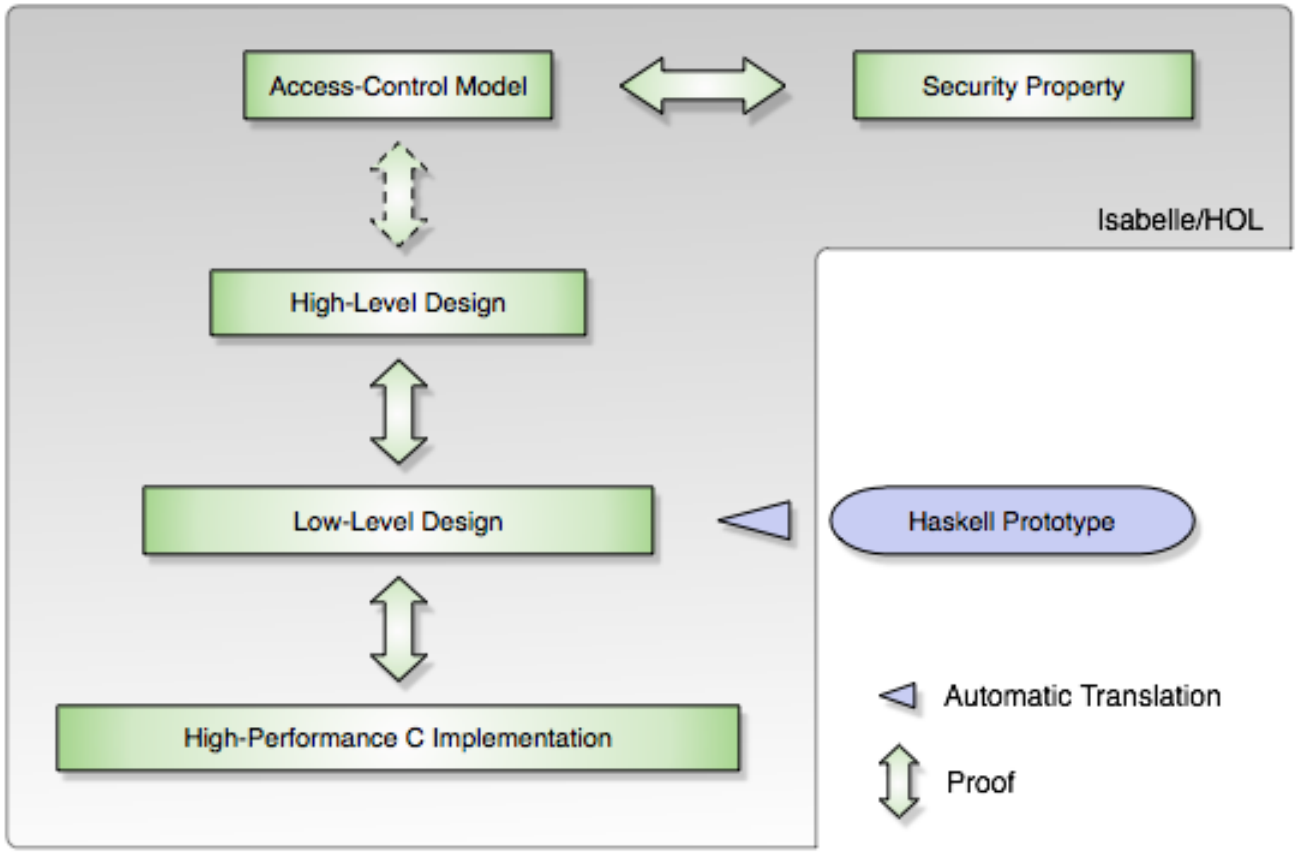
  Efficiency: compilation techniques, data structures

  Correctness: type and proof systems, testing

  $\Rightarrow$ Haskell is likely to become even better in the future

Research highlight: the L4.verified project

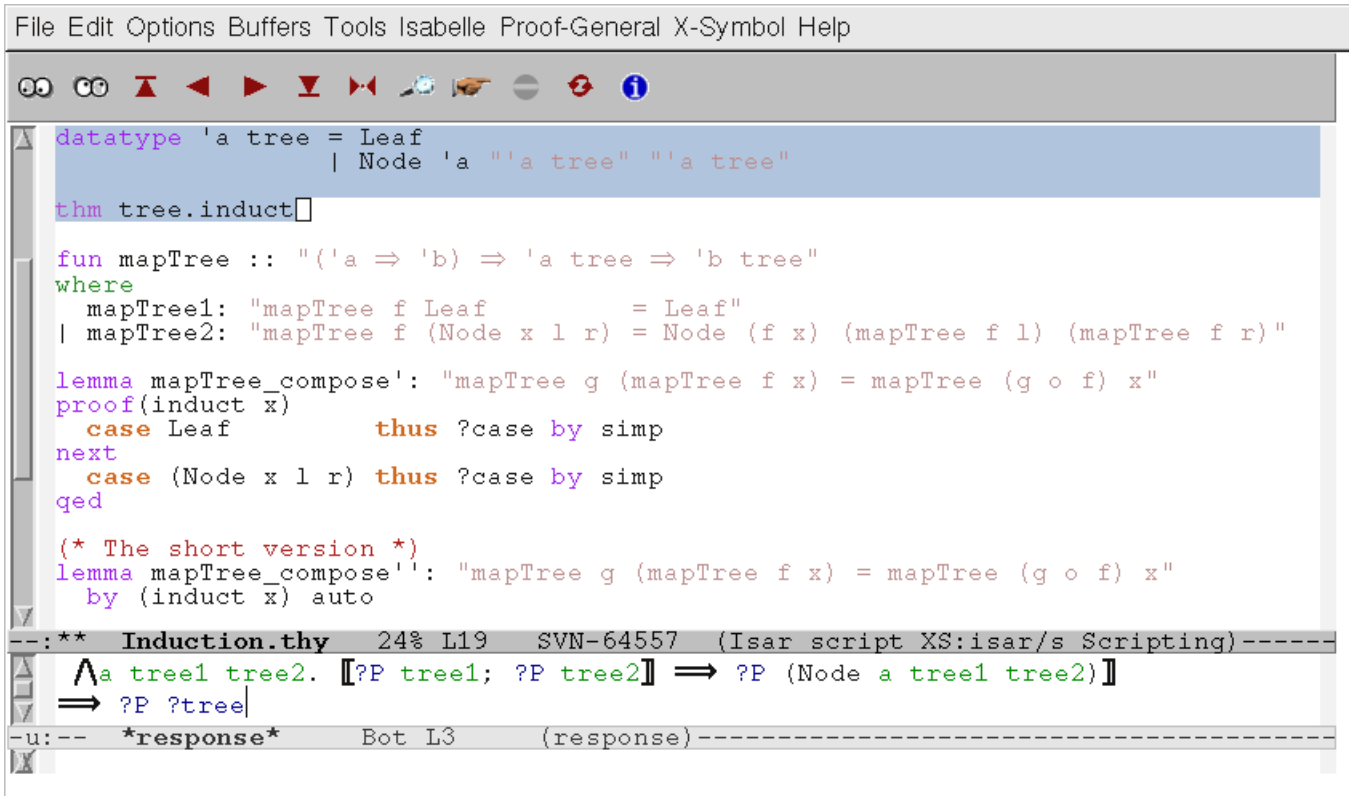
- A truly trustworthy, high-performance operating system kernel with a machine-checked proof of its functional correctness w.r.t. a high level, formal description of its expected behaviour.



http://ertos.nicta.com.au/research/l4.verified/

Mechanized theorem proving

- Complex formal proofs are constructed using proof assistants
- In our research group, we use the Isabelle/HOL proof assistant



- There will be a **lab course next semester.**
  Chance for an exciting student project, bachelor/master thesis.

End of part I

- Starting next class: part II on Formal Methods

  Peter Müller takes over.

- If you are interested in a student project, bachelors thesis, or masters thesis using functional programming or logic, please contact one of the organizers of this course or visit

  https://www1.ethz.ch/infsec/education/studentProjects