Today's Topics

- Part I: Type Checking/Type Inference
  A necessity of compilers and interpreters
- Part II: Parallelism in Functional Programming Languages
  A hot research topic
- Part III: The Story of Haskell
  Behind the scenes of Haskell (and Functional Programming)

Part I: Type Checking/Type Inference

...of central importance for implementing functional programming languages like Haskell (as well as languages of other paradigms)

We distinguish...

- monomorphic and
- polymorphic

type checking/type inference.

Type Checking/Type Inference

Important, too, in this context...

- Overloading (as a special case of polymorphism (ad hoc polymorphism)
- Type classes (such as `Num`, `Eq`, etc.)

Reference

The following presentation is based on...

- Chapter 13
  Simon Thompson. *Haskell – The Craft of Functional Programming*, Addison-Wesley, 2nd edition, 1999.
- Chapter 5
  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999 (In German).

Type-Based Programming Languages

We distinguish...

- Programming languages with...
  - *weak* (checked at run-time)
  - *strong* (checked at compile-time)

  typing

Benefits

...of using typed programming languages

- *More reliable code*:
  ...many programming flaws can be detected at compile-time; type correctness is a proof of correctness on the abstraction level of types
- *More efficient code*: ...no type-checks required at run-time
- *More effective program development*:
  ...Type information is additional program documentation
  ...the *understanding*, *advancing* and *maintaining* of programs gets simpler, like the search for pre-defined library functions ("is there a library function, which removes duplicates from a list, i.e., applied to the list `[2,3,2,1,3,4]` yields the result `[2,3,1,4]`? Search for a function with the type `(Eq a) => [a] -> [a]`".)

What's it all about? 1(4)

Haskell expressions all have a defined type.

This type can be...

- monomorphic
- polymorphic
  - restricted by type class constraints

...and *explicitly* be given as in the below examples:

```
'w'  :: Char                             -- monomorphic
flip :: (a -> b -> c) -> (b -> a -> c)   -- polymorphic
elem :: Eq a => a -> [a] -> Bool         -- polymorphic with
                                            type class constraint
```


What's it all about? 2(4)

Types (of expressions) can automatically be inferred by Haskell compilers and interpreters as in the below example:

```
magicType = let
              pair x y z = z x y
              f y = pair y y
              g y = f (f y)
              h y = g (g y)
            in h (\x->x)
```

What's it all about? 3(4)

The call of `:t magicType` in Hugs yields:

```
Main> :t magicType
magicType ::
   (((((((a -> a) -> (a -> a) -> b) -> b) ->
   (((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) ->
   ((((a -> a) -> (a -> a) -> b) -> b) ->
   (((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) -> d) -> d) ->
   ((((((a -> a) -> (a -> a) -> b) -> b) -> (((a -> a) ->
   (a -> a) -> b) -> b) -> c) -> c) -> (((((a -> a) ->
   (a -> a) -> b) -> b) -> (((a -> a) ->
   (a -> a) -> b) -> b) -> c) -> c) -> d) -> d) -> e) -> e
```

Quite a complex type, isn't it?

What's it all about? 4(4)

Central question...

- How does Hugs succeed to automatically infer this type information?

The systematical investigation leads us to notions such as...

- *Type analysis/-checking*
- *Type systems* and
- *Type inference*

First, an informal approach, driven by examples...

Monomorphic Type Checking

We first consider a simplified situation without...

- polymorphism

Characteristic for this situation is...

- An expression has
  - either a unique single type
  - or no type at all (not well-typed)

Convention:
The polymorphism of polymorphic or overloaded functions is explicitly resolved (by indexing) as in...

```
+_{Int} :: Int -> Int -> Int
length_{Char} :: [Char] -> Int
```


Type Analysis

Central idea...

- Evaluating the application context an expression is embedded in

In the following a couple of examples for illustration...

Type Checking Function Applications

The context of function applications allows conclusions about the types involved...

```
f must be of (function) type...       e must be of type s...
       s -> t                                     s
          \                                      /
           \___________              ____________/
                       \        /
                        \      /
                        \/   \/
                          f e
                          /\
                          /
                         /
f e must be of (result) type t...
```


Type Checking (Other) Expressions 1(2)

Correctly typed...

```
Char -> Int     Char
          \      \
           \      \
           \/     \/
            ord   'c'   +_{Int}    3_{Int}
            /\            /\          /\
             |             |           |
             |             |           |
            Int  (Int -> Int -> Int)  Int
```


Type Checking Expressions 2(2)

Incorrectly typed...

```
Char -> Int     Char
          \       \
           \       \
           \/      \/
            ord   'c'   +_{Int}      False
            /\            /\          /\
             |             |           |
             |             |           |
            Int  (Int -> Int -> Int)  Bool

            /\    /\     /\           /\
             |_____|      |____________|
                |               |
                |               |
        Expected and        Expected ...Int
        given argument      Given    ...Bool
        type coincide       => TYPE ERROR
        => TYPE CORRECT
```


Type Checking Function Definitions

```
f :: t1 -> t2 -> ... -> tk -> t
f a1 a2 ... ak
  | b1   = e1
  | b2   = e2
  ...
  | bk   = ek
```

In type checking monomorphic function definitions, three things have to be checked...

- Each guard `bi` must be of type `Bool`
- The result value of each expression `ei` must be of type `t`
- The pattern of each argument `pi` must be consistent with the type of that formal parameter, i.e., with type `ti`.

Consistency of Patterns 1(2)

Informally:
A pattern is *consistent* with a type, if it *matches* (some of) the values of this type.

In detail:

- A variable is consistent with any type
- A literal is consistent with its type
- A pattern `p:q` is consistent with type `[t]`, if `p` is consistent with type `t` and `q` is consistent with type `[t]`
- ...

Consistency of Patterns 2(2)

Example:

- `(42:xs)` ...is consistent with `[Int]`
- `(x:xs)` ...is consistent with any type of lists

Polymorphic Type Checking 1(2)

Characteristic for the situation of polymorphic type checking...

- An expression can
  - have several types (while being well-typed)

Central for the solution of the polymorphic type checking problems are...

- *Constraint satisfaction*
- *Unification*

Polymorphic Type Checking 2(2)

Consider...

```
length :: [a] -> Int
```

Informal interpretation of the type...

```
length :: [a] -> Int
```

...as an abbreviation of

```
length :: [t] -> Int
```

where `t` is an arbitrary monomorphic type, thus in total an abbreviation of...

```
[Int] -> Int
[(Bool,Char)] -> Int
...
```

Polymorphic Type Checking – Continuing the Example

In the example of...

```
length ['c','a']
```

we can conclude from the calling context to the type...

```
length :: [Char] -> Int
```


Observation

The preceding examples allow the following conclusions...

The application contexts of expressions impose

- ...different constraints on the expression type.

This way type checking boils down to the problem, if...

- Types can be determined for the various expressions such that all constraints are met.

Further Examples – Example 1

Consider:

```
f (x,y) = (x , ['a' .. y])
```

Observation:

- `f` ...expects pairs as arguments, where
  - 1st component ...no imposed constraints
  - 2nd component ...`y` must be of type `Char` because of being used in the range of an enumerated type `['a' .. y]`

Hence, conclusion for type `f`...

```
f :: (a , Char) -> (a , [Char])
```


Example 2

Consider:

```
g (m,zs) = m + length zs
```

Observation:

- `g` ...expects pairs as arguments, where
  - 1st component ...`m` must be of a numerical type because of being used as an operand of `+`
  - 2nd component ...`zs` must be of type `[b]` because of being used as an argument of the function `length` with `length :: [b] -> Int`

Hence, conclusion for type `g`...

```
g :: (Int, [b]) -> Int
```

Example 3 1(2)

Consider the composition of the two preceding examples:

```
g . f
```

Observation:

- `g . f` ...in a composition `g . f`, the return value of `f` becomes the argument of `g`

  In this example, this means...

  - Result type `f` ...`(a , [Char])`
  - Argument type `g` ...`(Int , [b])`
  - Wanted ...types for `a` and `b`, which satisfy the above two constraints

Example 3 2(2)

Illustration:

```
                      g . f
                    /\     /\
                   /         \
                  /           \
(Int, [b]) -> Int        (a , Char) -> (a , [Char])
     /\                                      /\
      \                                      /
       \                                    /
        \                                  /
     The input of g                       /
                          is the output of f
```

Hence, conclusion for the type of `g . f`...

```
g . f :: (Int, [Char]) -> Int
```


Unification in the case of the Example

Crucial for drawing the conclusion in the previous example is...

- Unification

Illustration:

```
                   ---------------------------------------
                                                          |
 _________________|_______________                        |
|                 |               |    (Int,[Int])        |
|  (Bool,Char)    |               |                       |
|                 |               |    (Int,[[c]])        |
|  (c->c,Char)    |  (Int,[Char]) |                       |
|                 |               |    ...                |
|  ...            |               |                       |
|                  ---------------------------------------
|                                 |
|_________________________________|

     (a,[Char])                        (Int,[b])
```

Unification

Introducing terminology. We say...

- *Instance* of a type ...given by replacing a type variable by a (concrete) type expression
- *Common instance* of two type expressions ...if the instance is an instance of both type expressions

Unification problem...

- Search for the *most general common* (*type*) *instance*

In the previous example:

- The most general common type instance of `(a,Char)` and `(Int,[b])`

  ...`(Int, [Char])`

More on Unification 1(3)

In general, unification does not lead to unique types...

Example:

```
    (a,[a])         ([b],c)
          \         /
           \/    \/
          ([b],[[b]])
  most general common instance
```

Observation:

- (a,[a]) ...requires that the second component is a list of elements of the type of the elements of the first component
- ([b],c) ...requires that the first component is of some list type
- Together ...the most general common type instance of (a,[a]) and ([b],c) is ([b],[[b]])

More on Unification 2(3)

Note:

- ([Bool],[[Bool]]), ([[c]],[[[c]]]) are...
  - instances of ([b],[[b]])
  - but no unifier: ([b],[[b]]) is not an instance of either of them

More on Unification 3(3)

Unification can fail...

```
  [Int] -> [Int]   a -> [a]
          \         /
           \       /
           \/    \/
            (?,?)
```

Unification of the...

- Argument type requires ...a must be of type [Int]
- Result type requires ...a must be of type Int
- Together ...unification fails (inconsistent constraints)

Type Checking Expressions

Here... polymorphic function application

```
           f e
        /\     /\
       /        \
      /          \
   s -> t         u
   | /\         /\ \
   |  \         /   \
   |   \       /     |
   |   Unification   |
   |   of s and u    |
   \/                \/
   s' -> t'          s'
        /\
         \
          \
     Result type t'
```

Observation:

- s and u need not to coincide; they only need to be *unifiable*

Example – `map` and `ord`

Consider...

```
map :: (a -> b) -> [a] -> [b]
ord :: Char -> Int
```

Unification of `a -> b` and `Char -> Int` yields...

```
map :: (Char -> Int) -> [Char] -> [Int]
```

Hence, we receive

```
map ord :: [Char] -> [Int]
```


Example – `foldr`

Reminder...

```
foldr f s []     = s
foldr f s (x:xs) = f x (foldr f s xs)
```

Application example...

```
foldr (+) 0 [3,5,34] = 42
```

...this suggests that the most general type is

```
foldr :: (a -> a -> a) -> a -> [a] -> a
```


Example – `foldr` 2(2)

More careful reasoning shows...

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Illustration:

```
                          b
                         /
                        /   a
                       \/  /
foldr f s []      = s  /      b
                      \/ ------------
foldr f s (x:xs)  = f x (foldr f s xs)
                    ------------------
                              b
```


Type Checking Polymorphic Function Definitions

```
f :: t1 -> t2 -> ... -> tk -> t
f a1 a2 ... ak
  | b1   = e1
  | b2   = e2
  ...
  | bk   = ek
```

Three things have to be checked...

- Each guard `bi` must be of type `Bool`
- The return value of each expression `ei` must be of a type `si`, which is as least as general as type `t`, i.e. `t` must be an instance of `si`
- The pattern of each argument `pi` must be consistent with the type of the formal parameter, i.e., with the type `ti`.

Typ Checking and Type Classes

Consider...

```
member []     y = False
member (x:xs) y = (x==y) || member xs y
```

In the above example, the usage of (==) forces...

```
member :: Eq a => [a] -> a -> Bool
```


Type Checking and Overloading

Consider the function application...

```
member e
```

with

```
e      :: Ord b => [[b]]
```

Without further context information unification yields...

```
member :: [[b]] -> [b] -> Bool
```

Thus, we receive...

```
member e :: [b] -> Bool
```

More carefully, we have to take the contexts into account, too!

Context Analysis

```
(Eq [b] , Ord b)
```

Analysing and simplifying the contexts yields...

- Context constraints refer to type variables
  `instance Eq a => Eq [a] where...`
  ...this yields `(Eq b, Ord b)`
- Repeat until no more instances apply
- Simplification of the context by means of the information given by `class` yields...
  `class Eq a => Ord a where...`
- Hence ...`Ord b`
- Summing up ...`member e :: Ord b => [b] -> Bool`

Summary – A Three-Stage Process

The three-stage process consists of

- Unification
- Analysis (with instances)
- Simplification

...is a typical pattern of the context-aware type analysis in Haskell.

Type Systems and Type Inference

Informally...

- *Type systems* are...
  - logical systems, which allow us to formalize statements of the form "exp is of type t" and to prove them by means of the axioms and rules of the type system
- *Type inference* denotes...
  - the process to automatically derive the type of an expression by means of the axioms and rules of a type system
    Key words: Type inference algorithm, Unification

A Typical Part of a Type Grammar

...generates the language of types

$$\begin{array}{lll} \tau & ::= & Int|Float|Char|Bool \quad \text{(simple type)} \\ & | & \alpha \quad \text{(type variable)} \\ & | & \tau \rightarrow \tau \quad \text{(function type)} \end{array}$$

$$\begin{array}{lll} \sigma & ::= & \tau \quad \text{(type)} \\ & | & \forall\, \alpha.\ \sigma \quad \text{(type binding)} \end{array}$$

We say:

- τ ...a type
- σ ...a type schema

A Typical Part of a Type System 1(2)

...associates with each (typable) expression of the language a type of the type language

VAR $$\frac{}{\Gamma \vdash var : \Gamma(var)}$$

CON $$\frac{}{\Gamma \vdash con : \Gamma(con)}$$

COND $$\frac{\Gamma \vdash exp : Bool \quad \Gamma \vdash exp_1 : \tau \quad \Gamma \vdash exp_2 : \tau}{\Gamma \vdash \text{ if } exp \text{ then } exp1 \text{ else } exp2 : \tau}$$

APP $$\frac{\Gamma \vdash exp : \tau' \rightarrow \tau \quad \Gamma \vdash exp' : \tau'}{\Gamma \vdash exp\ exp' : \tau}$$

ABS $$\frac{\Gamma[var \mapsto \tau'] \vdash exp : \tau}{\Gamma \vdash /x\text{->}exp : \tau' \rightarrow \tau}$$

...

where Γ is a so-called *type assumption*.

Typical Part of a Type System 2(2)

Type assumptions are...

- partial functions, which map variables to type schemas

Here, $\Gamma[var_1 \mapsto \tau_1, \ldots, var_n \mapsto \tau_n]$ is the function, which yields the type τ_i for each var_i and is as Γ otherwise.

The Schematic Unification Algorithm

$$
\begin{array}{rcl}
\mathcal{U}(\alpha,\alpha) & = & [] \\
\mathcal{U}(\alpha,\tau) & = & \begin{cases} [\tau/\alpha] & \text{if } \alpha \notin \tau \\ error & \text{otherwise} \end{cases} \\
\mathcal{U}(\tau,\alpha) & = & \mathcal{U}(\alpha,\tau) \\
\mathcal{U}(\tau_1 \to \tau_2, \tau_3 \to \tau_4) & = & \mathcal{U}(U\tau_2, U\tau_4)U \\
 & & \quad \text{where } U = \mathcal{U}(\tau_1,\tau_3) \\
\mathcal{U}(\tau,\tau') & = & \begin{cases} [] & \text{if } \tau = \tau' \\ error & \text{otherwise} \end{cases}
\end{array}
$$

Remarks:

- U ...(most general) unifier (essentially a substituion)
- Application of equations sequentially from top to bottom

Example on Unification/Most General Unification

Consider... `a -> (Bool,c)` and `Int -> b`

- *Unifier* ...Substitution `[Int/a,Float/c,(Bool,Float)/b]`
- *Most general unifier* ...Substitution `[Int/a,(Bool,c)/b]`

Example on the Unification Algorithm

Task

...Unifying the type expressions `a -> c` and `b -> Int -> a`

Solution

$$
\begin{array}{rcl}
 & & \mathcal{U}(\texttt{a -> c},\texttt{b -> Int -> a}) \\
(\text{mit } U = \mathcal{U}(\texttt{a},\texttt{b}) = [\texttt{b/a}]) & = & \mathcal{U}(U\texttt{c}, U(\texttt{Int -> a}))U \\
 & = & \mathcal{U}(\texttt{c, Int -> b})[b/a] \\
 & = & [\texttt{Int -> b/c}][\texttt{b/a}] \\
 & = & [\texttt{Int -> b/c,b/a}]
\end{array}
$$

In total ...`b -> Int -> b`

Essence of Automatic Type Inference Algorithms

...syntax-directed application of the rules of the type inference systems

The key...

- Modification of the type inference system such that there is always only a single rule applicable

Further Reading 1(3)

Types and type systems, type inference...

- For functional languages in general
  - Anthony J. Field, Peter G. Robinson. *Functional Programming*. Addison-Wesley, 1988 (Chapter 7)
- Haskell-specific
  - Simon Peyton Jones, John Hughes. *Report on the Programming Language Haskell 98*. `http://www.haskell.org/report/`

Further Reading 2(3)

- Overview
  - J. C. Mitchell. *Type Systems for Programming Languages*. In J. van Leeuwen (Hrsg.). *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics*. Elsevier Science Publishers, 367-458, 1990.
- Foundations of polymorphic type systems
  - Robin Milner. *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences 17, 248-375, 1978.
  - L. Damas, Robin Milner. *Principal Type Schemes for Functional Programming Languages*. In Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages (POPL'82), 207-218, 1982

Further Reading 3(3)

- Unification algorithm
  - J. A. Robinson. *A Machine-Oriented Logic Based on the Resolution Principle*. Journal of the ACM 12(1), 23-42, 1965.
- Type systems and type inference
  - Luca Cardelli. *Basic Polymorphic Type Checking*. Science of Computer Programming 8, 147-172, 1987.

Part II: Parallelism in Functional Programming Languages

Parallelism

- Implicit
- Explicit
- Skeletons

Reference

The following presentation is based on...

- Chapter 21
  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*, Springer, 2006. (In German).

Related and relevant in this context...

- Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*, The MIT Press, 1989.
- Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, Simon L. Peyton Jones. *Algorithms + Strategy = Parallelism*. Journal of Functional Programming, 8(1):23-60, 1998.
- Philip W. Trinder, Hans-Wolfgang Loidl, Robert F. Pointon. *Parallel and Distributed Haskells*. Journal of Functional Programming, 12(4&5):469-510, 2002.

Parallelism in Imperative Languages

In particular...

- Data-parallel Languages (e.g. High Performance Fortran)
- Libraries (PVM, MPI) $\leadsto$ *Message Passing Model* (C, C++, Fortran)

Parallelism in Functional Languages

In particular...

- Implicit (expression) parallelism
- Explicit parallelism
- Algorithmic skeletons

Implicit Parallelism

...resp. *expression parallelism*

Consider the functional expression of the form `f(e1,...,en)`:

Note:

- Arguments (and functions) can be evaluated in parallel.
- Advantages: Parallelism *for free*! No effort for the programmer.
- Disadvantages: Results often unsatisfying. E.g. granularity, load distribution, etc. not taken into account.

Thus:

- Easy to detect parallelism (i.e., for the compiler), but hard to fully exploit.

Explicit Parallelism

By...

- Introducing meta-statements (e.g. to control the data and load distribution, communication)
- Advantages: Often superior results by explicit hands-on control of the programmer
- Disadvantages: High programming effort, loss of functional elegance

Algorithmic Skeletons

Compromise between...

- *explicit imperative* parallel programming
- *implicit functional* parallel programming

In the following

- Massively parallel systems
- Algorithmic skeletons

Massively Parallel Systems

...characterized by

- large number of processors with
  - local memory
  - communication by message exchange
- MIMD-Parallel Processor Architecture (*Multiple Instruction/Multiple Data*)
- Here: SPMD-Programming Style (*Single Program/Multiple Data*)

Algorithmic Skeletons

Algorithmic Skeletons...

- represent typical patterns for parallelization (*Farm*, *Map*, *Reduce*, *Branch&Bound*, *Divide&Conquer*,...)
- are easy to instantiate for the programmer
- allow parallel programming at a high level of abstraction

Realization of Algorithmic Skeletons

...in functional languages

- by special higher-order functions
- with parallel implementation
- embedded in sequential languages

Thus

- Hiding of parallel implementation details in the skeleton
- Elegance and (parallel) efficiency for special application patterns

Example: Parallel Map on Distributed List

Consider the higher-order function `map` on lists...

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x) : (map f xs)
```

Observation

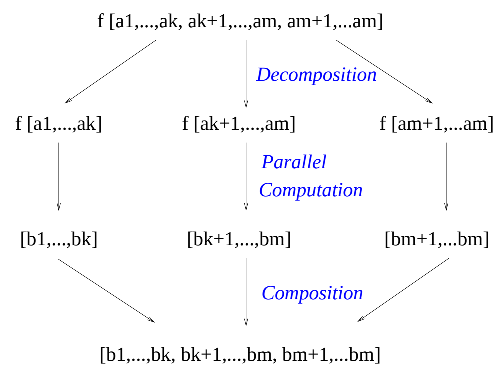
- Application of `f` to a list element does not depend on other list elements

Apparent

- Dividing the list into sublists followed by *parallel* application of map to the sublists (parallelization pattern *Farm*)

Parallel Map on Distributed Lists

For illustration...



Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer, 2006, S. 445.

On the Implementation

Implementing the parallel map function requires...

- special data structures, which take into account the aspect of distribution (ordinary lists are inefficient for this purpose)

Skeletons on distributed data structures

- so-called *data-parallel skeletons*

Note the difference:

- *Data-parallelism*: Supposes an a priori distribution of data on different processors
- *Task-parallelism*: Processes and data to be distributed are not known a priori, hence dynamically generated

Programming of a Parallel Application

...using algorithmic skeletons

- Recognizing problem-inherent parallelism
- Selecting an adequate data distribution (granularity)
- Selecting a suitable skeleton from a library
- Problem-specific instantiation of the skeleton(s)

Remark:

- Some languages (e.g. Eden) support also the implementation of skeletons

Data Distribution on Processors

...is

- crucial for
  - structure of the complete algorithm
  - efficiency

The hardness of the distribution problems depends on...

- Independence of all data elements (like in the map-example): Distribution is easy
- Independence of subsets of data elements
- Complex dependences of data elements: Adequate distribution is challenging

An auxiliary means

- So-called *covers* (investigated by various authors)

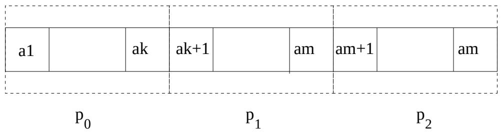
Covers

...describe

- Decomposition and communication pattern of a data structure

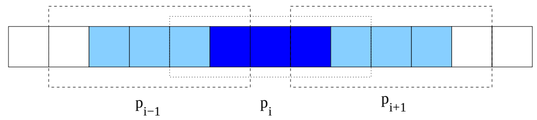
Example: Simple List Cover

Distributing a list on 3 processors p_1, p_2, and p_3:



Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer, 2006, S. 446.

Example: List Cover with Overlapping Elements



Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer, 2006, S. 446.

General Cover Structure

```
Cover = {
  Type S a       -- Whole object
       C b       -- Cover
       U c       -- Local sub-objects

split :: S a -> C (U a)    -- Decomposing the original object
glue :: C (U a) -> S a     -- Composing the original object
}
```

It is required:

```
glue . split = id
```

Note: No (valid) Haskell

Realization in a Programming Language

...implementing covers requires support for

- the specification of covers
- the programming of algorithmic skeletons on covers
- the provision of often used skeletons in libraries

...is

- currently a hot research topic in functional programming

Last but not least

Implementing skeletons...

- by message passing via skeleton hierarchies

Further Reading

- Hans-Werner Loidl et al. *Comparing Parallel Functional Languages: Programming and Performance*. Higher-Order and Symbolic Computation, 16(3):203-251, 2003.

Part III: The Story of Haskell

16 Years of Haskell: A Retrospective on the occasion of its 15th Anniversary

by

Simon Peyton Jones

Wearing the Hair Shirt: A Retrospective on Haskell

`http://research.microsoft.com/users/simonpj/papers/haskell-retrospective/`

Haskell at HOPL III

More recently...

- Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. *A History of Haskell: Being Lazy with Class*. In Proceedings of the Third ACM SIGPLAN 2007 Conference on History of Programming Languages (HOPL III), (San Diego, California, June 09 - 10, 2007), 12-1 - 12-55.

Check out the ACM Digital Library (`www.acm.org/dl`) for this article!

Last but not least

Final (oral) examination...

- In principle, any time. Just make an appointment by email (`knoop@complang.tuwien.ac.at`) or phone (58801-18510).
- Topics: Assignments plus lecture materials.