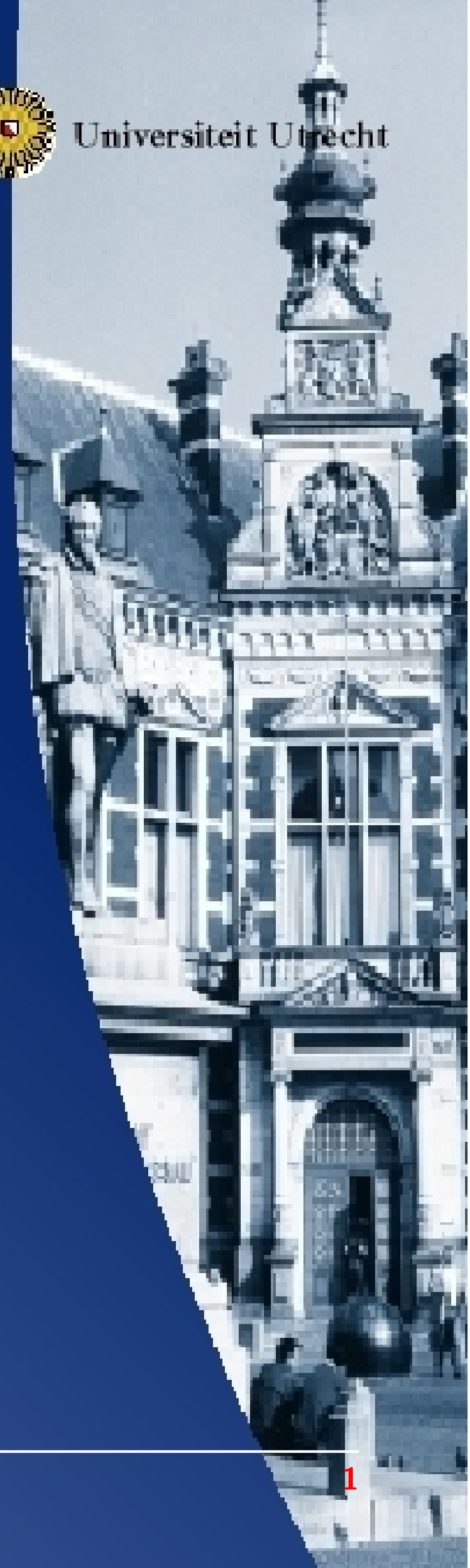


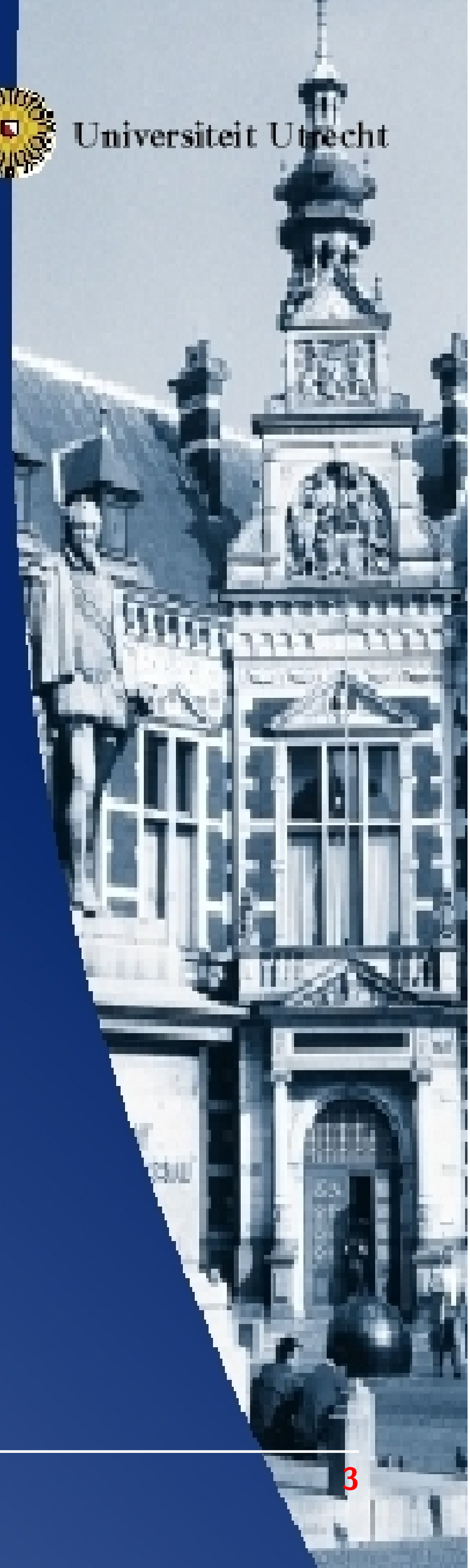
A second prototype of a first prototype for Generic H∀SKELL

Andres Löh

15. November 2000

Overview

- Generic definitions (MPC-style)
- Implementation status
- Goals of future work on the prototype

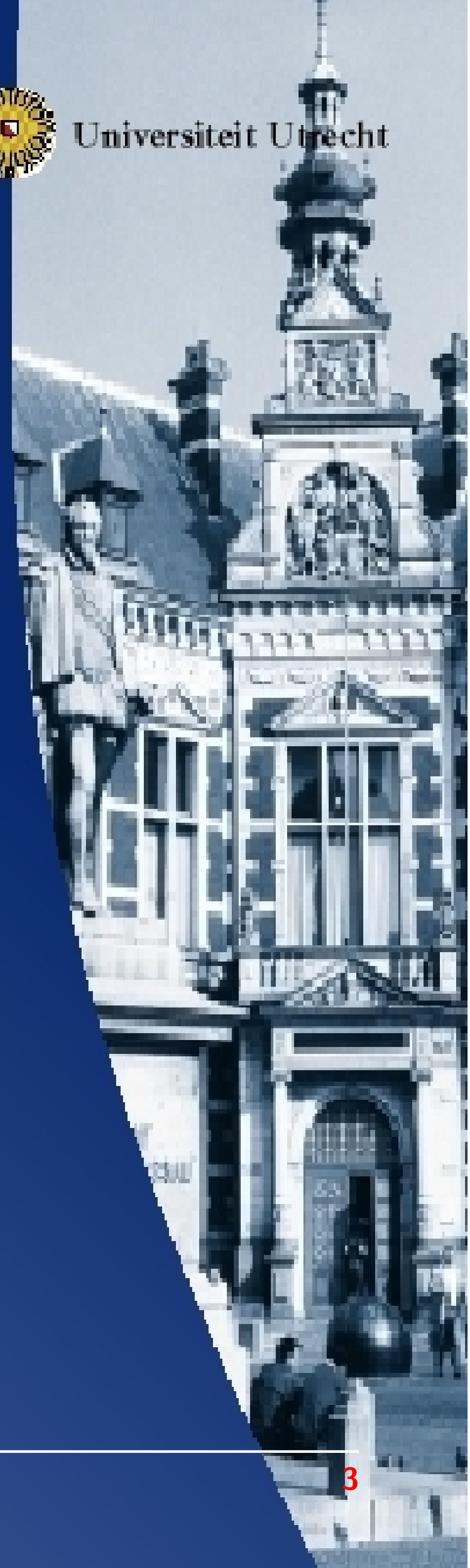


Generic programming

- View Haskell **data** definitions as definitions for structured sum-of-product types
- Replace Haskell's n-ary sums and products by binary sums and products
- Write functions based on structural recursion over datatypes,
  i. e. give clauses for $+$, $\times$ etc.
- Use the same function for arbitrary datatypes
- Similar thing as **derive** in Haskell, but more general and well-defined
- Examples include map, $encode$, $size$, $reduce$ ($crush$), $cata$ ($fold$)

Generic programming

- View Haskell **data** definitions as definitions for structured sum-of-product types
- Replace Haskell's n-ary sums and products by binary sums and products
- Write functions based on structural recursion over datatypes, i. e. give clauses for $+$, $\times$ etc.
- Use the same function for arbitrary datatypes
- Similar thing as **derive** in Haskell, but more general and well-defined
- Examples include map, $encode$, $size$, $reduce$ ($crush$), $cata$ ($fold$)



Generic programming

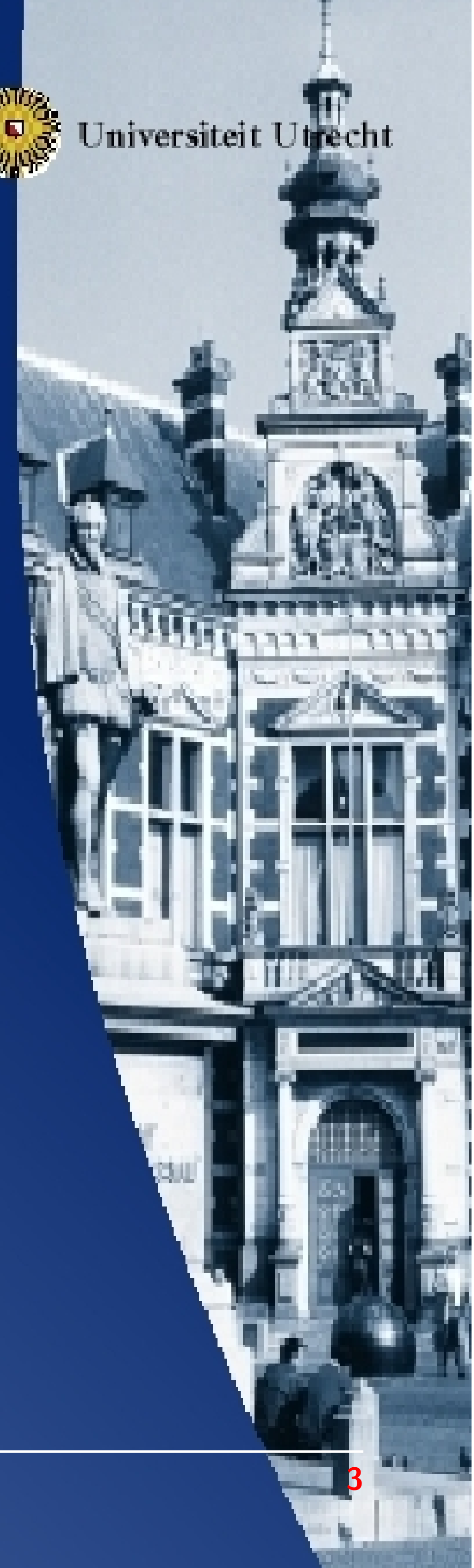
- View Haskell **data** definitions as definitions for structured sum-of-product types
- Replace Haskell's n-ary sums and products by binary sums and products
- Write functions based on structural recursion over datatypes,
  i. e. give clauses for $+$, $\times$ etc.
- Use the same function for arbitrary datatypes
- Similar thing as **derive** in Haskell, but more general and well-defined
- Examples include map, $encode$, $size$, $reduce$ $(crush)$, $cata$ $(fold)$

Generic programming

- View Haskell **data** definitions as definitions for structured sum-of-product types
- Replace Haskell's n-ary sums and products by binary sums and products
- Write functions based on structural recursion over datatypes,
  i. e. give clauses for $+$, $\times$ etc.
- Use the same function for arbitrary datatypes
- Similar thing as **derive** in Haskell, but more general and well-defined
- Examples include map, $encode$, $size$, $reduce$ $(crush)$, $cata$ $(fold)$

Generic programming

- View Haskell **data** definitions as definitions for structured sum-of-product types
- Replace Haskell's n-ary sums and products by binary sums and products
- Write functions based on structural recursion over datatypes,
  i. e. give clauses for $+$, $\times$ etc.
- Use the same function for arbitrary datatypes
- Similar thing as **derive** in Haskell, but more general and well-defined
- Examples include map, $encode$, $size$, $reduce$ $(crush)$, $cata$ $(fold)$



Generic programming

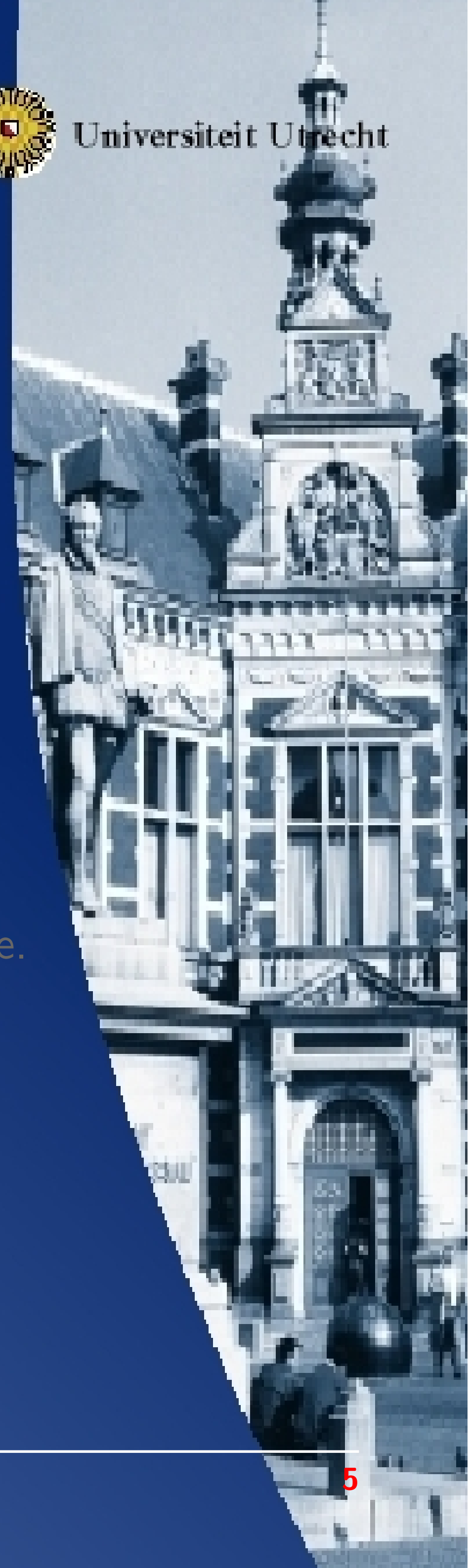
- View Haskell **data** definitions as definitions for structured sum-of-product types
- Replace Haskell's n-ary sums and products by binary sums and products
- Write functions based on structural recursion over datatypes,
  i. e. give clauses for $+$, $\times$ etc.
- Use the same function for arbitrary datatypes
- Similar thing as **derive** in Haskell, but more general and well-defined
- Examples include map, $encode$, $size$, $reduce$ ($crush$), $cata$ ($fold$)

MPC-style definitions — example

This function encodes a value of arbitrary type into a list of bits.

$$
\begin{aligned}
&\mathit{Encode}\langle \kappa :: \Box \rangle && :: \kappa \to \star \\
&\mathit{Encode}\langle \star \rangle\ t && = t \to [\mathit{Bit}] \\
&\mathit{Encode}\langle \kappa_1 \to \kappa_2 \rangle\ t && = \forall a. \mathit{Encode}\langle \kappa_1 \rangle\ a \to \mathit{Encode}\langle \kappa_2 \rangle\ (t\ a)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{encode}\langle\!\langle t :: \kappa \rangle\!\rangle && :: \mathit{Encode}\langle \kappa \rangle\ t \\
&\mathit{encode}\langle\!\langle 1 \rangle\!\rangle && = [\,] \\
&\mathit{encode}\langle\!\langle + \rangle\!\rangle\ \mathit{eA}\ \mathit{eB}\ (\mathit{inl}\ a) && = \mathtt{O} : \mathit{eA}\ a \\
&\mathit{encode}\langle\!\langle + \rangle\!\rangle\ \mathit{eA}\ \mathit{eB}\ (\mathit{inr}\ b) && = \mathtt{I} : \mathit{eB}\ b \\
&\mathit{encode}\langle\!\langle \times \rangle\!\rangle\ \mathit{eA}\ \mathit{eB}\ (a, b) && = \mathit{eA}\ a +\!\!+\ \mathit{eB}\ b
\end{aligned}
$$

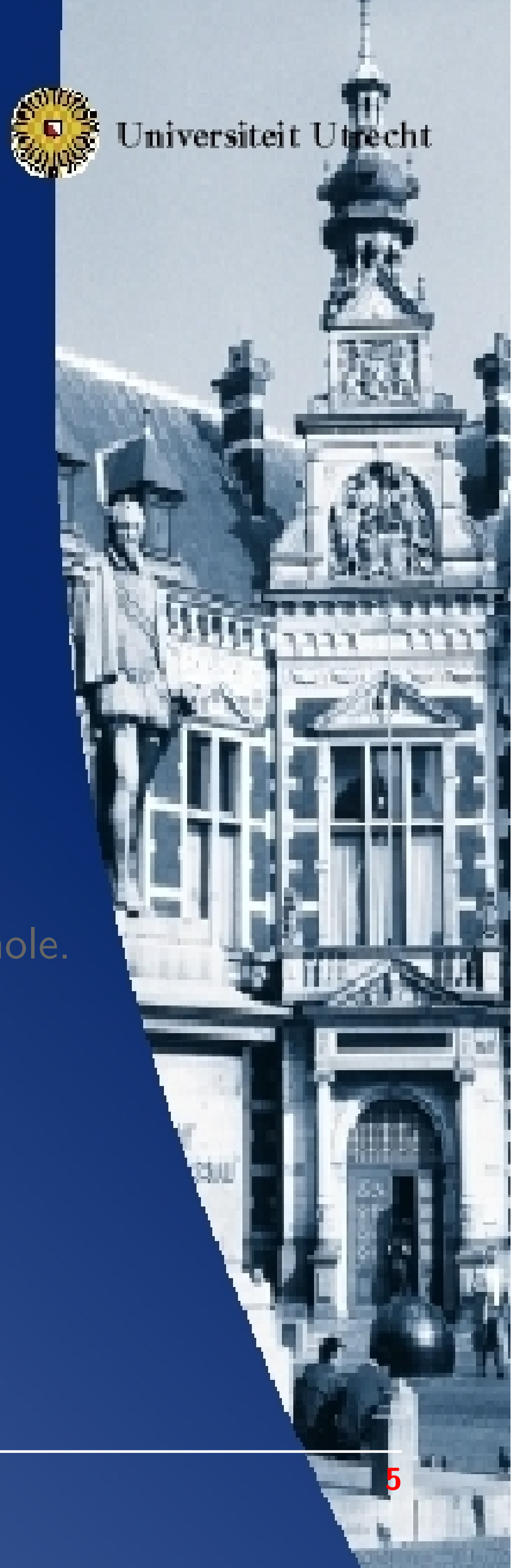


MPC-style definitions — continued

- Based on Hinze's paper at MPC 2000
- No restriction to regular types.
- One polytypic declaration works for types of arbitrary kind.
  It consists of a type ($Encode$) and a function ($encode$) definition.
- If specialised to types of different kinds, the type of the generic function varies.
- The line

  $$Encode\langle\kappa_1 \to \kappa_2\rangle\ t \qquad = \forall a. Encode\langle\kappa_1\rangle\ a \to Encode\langle\kappa_2\rangle\ (t\ a)$$

  can be left out, as it has to be as it is for the definition to be well-typed as a whole.
- The user has to supply:
  - the type of the function specialised to a type of kind $\star$
  - clauses for (), constant types, $+$ and $\times$

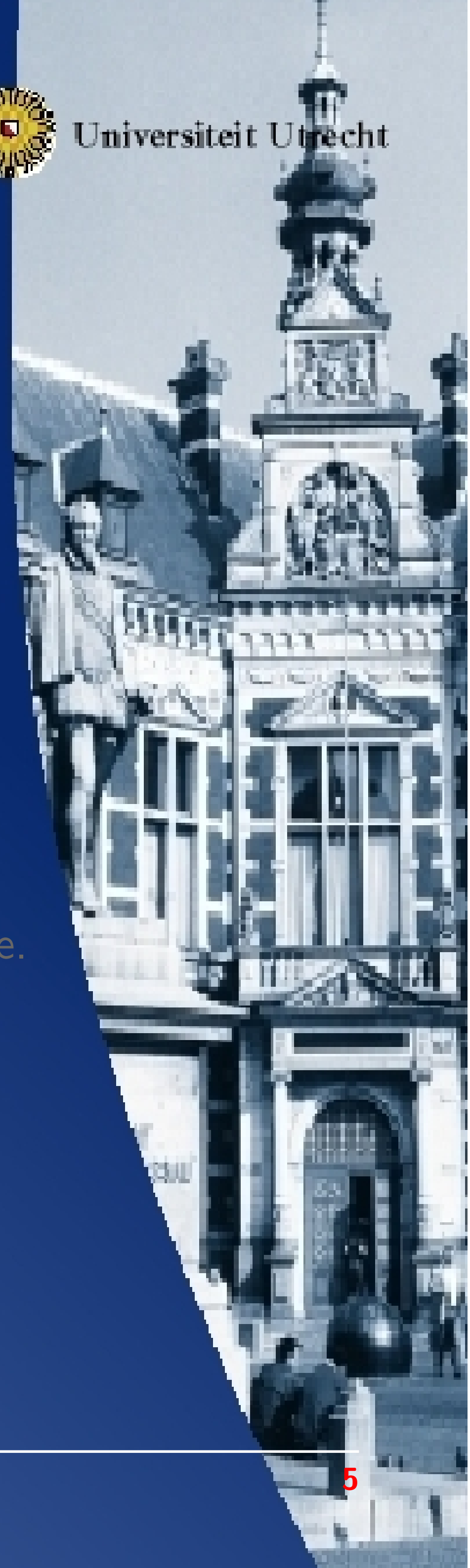


MPC-style definitions — continued

- Based on Hinze's paper at MPC 2000
- No restriction to regular types.
- One polytypic declaration works for types of arbitrary kind.
  It consists of a type (Encode) and a function (encode) definition.
- If specialised to types of different kinds, the type of the generic function varies.
- The line

$$\mathit{Encode}\langle\kappa_1 \to \kappa_2\rangle\ t \qquad = \forall a.\mathit{Encode}\langle\kappa_1\rangle\ a \to \mathit{Encode}\langle\kappa_2\rangle\ (t\ a)$$

  can be left out, as it has to be as it is for the definition to be well-typed as a whole.
- The user has to supply:
  - the type of the function specialised to a type of kind $\star$
  - clauses for $()$, constant types, $+$ and $\times$



MPC-style definitions — continued

- Based on Hinze's paper at MPC 2000
- No restriction to regular types.
- One polytypic declaration works for types of arbitrary kind.
  It consists of a type ($Encode$) and a function ($encode$) definition.
- If specialised to types of different kinds, the type of the generic function varies.
- The line

$$Encode\langle\kappa_1 \to \kappa_2\rangle\ t \qquad = \forall a. Encode\langle\kappa_1\rangle\ a \to Encode\langle\kappa_2\rangle\ (t\ a)$$

  can be left out, as it has to be as it is for the definition to be well-typed as a whole.
- The user has to supply:
  - the type of the function specialised to a type of kind $\star$
  - clauses for (), constant types, $+$ and $\times$

MPC-style definitions — continued

- Based on Hinze's paper at MPC 2000
- No restriction to regular types.
- One polytypic declaration works for types of arbitrary kind.
  It consists of a type ($Encode$) and a function ($encode$) definition.
- If specialised to types of different kinds, the type of the generic function varies.
- The line

  $$Encode\langle\kappa_1 \to \kappa_2\rangle\ t \qquad = \forall a. Encode\langle\kappa_1\rangle\ a \to Encode\langle\kappa_2\rangle\ (t\ a)$$

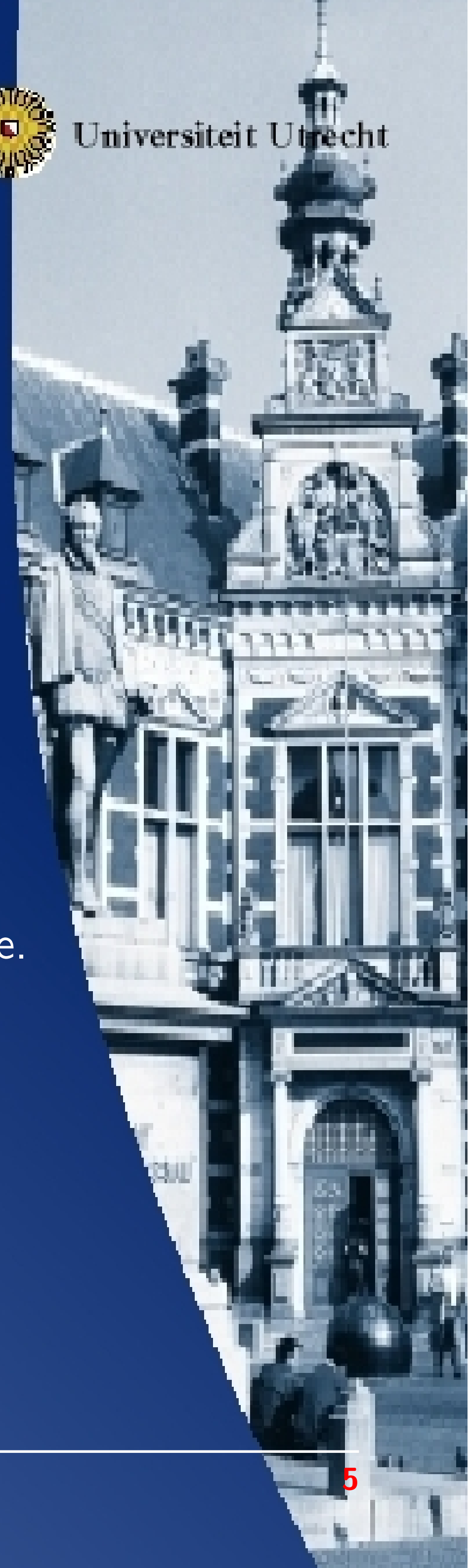
  can be left out, as it has to be as it is for the definition to be well-typed as a whole.
- The user has to supply:
  - the type of the function specialised to a type of kind $\star$
  - clauses for (), constant types, $+$ and $\times$

MPC-style definitions — continued

- Based on Hinze's paper at MPC 2000
- No restriction to regular types.
- One polytypic declaration works for types of arbitrary kind.
  It consists of a type ($Encode$) and a function ($encode$) definition.
- If specialised to types of different kinds, the type of the generic function varies.
- The line

$$Encode\langle\kappa_1 \rightarrow \kappa_2\rangle\ t \quad = \forall a . Encode\langle\kappa_1\rangle\ a \rightarrow Encode\langle\kappa_2\rangle\ (t\ a)$$

  can be left out, as it has to be as it is for the definition to be well-typed as a whole.
- The user has to supply:
  - the type of the function specialised to a type of kind $\star$
  - clauses for (), constant types, $+$ and $\times$



MPC-style definitions — continued

- Based on Hinze's paper at MPC 2000
- No restriction to regular types.
- One polytypic declaration works for types of arbitrary kind.
  It consists of a type (Encode) and a function (encode) definition.
- If specialised to types of different kinds, the type of the generic function varies.
- The line

  $$\mathit{Encode}\langle\kappa_1 \longrightarrow \kappa_2\rangle\ t \quad = \forall a.\mathit{Encode}\langle\kappa_1\rangle\ a \longrightarrow \mathit{Encode}\langle\kappa_2\rangle\ (t\ a)$$

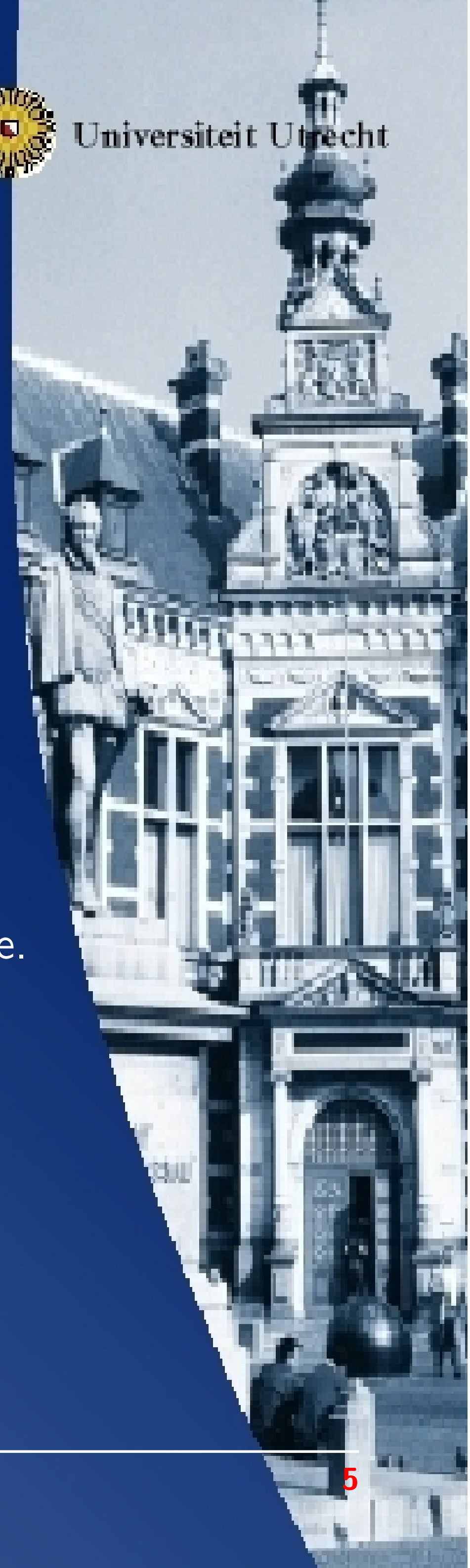
  can be left out, as it has to be as it is for the definition to be well-typed as a whole.
- The user has to supply:
  - the type of the function specialised to a type of kind $\star$
  - clauses for $()$, constant types, $+$ and $\times$

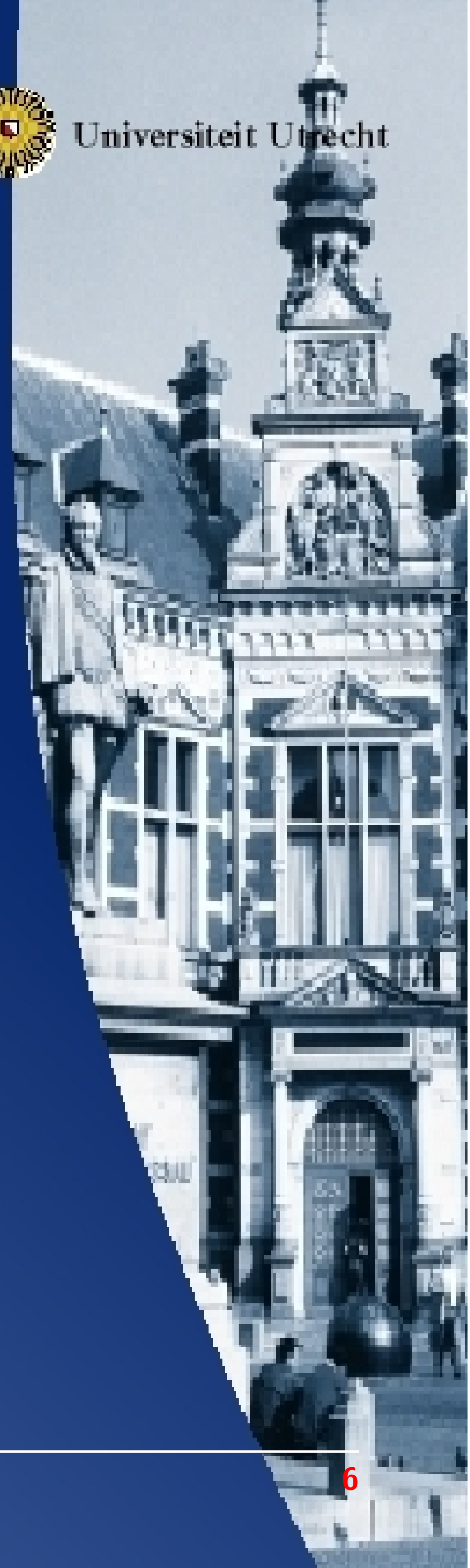
MPC-style definitions — continued

- Based on Hinze's paper at MPC 2000
- No restriction to regular types.
- One polytypic declaration works for types of arbitrary kind.
  It consists of a type ($Encode$) and a function ($encode$) definition.
- If specialised to types of different kinds, the type of the generic function varies.
- The line

$$Encode\langle\kappa_1 \rightarrow \kappa_2\rangle\ t \quad = \forall a. Encode\langle\kappa_1\rangle\ a \rightarrow Encode\langle\kappa_2\rangle\ (t\ a)$$

  can be left out, as it has to be as it is for the definition to be well-typed as a whole.
- The user has to supply:
  - the type of the function specialised to a type of kind $\star$
  - clauses for $()$, constant types, $+$ and $\times$





Implementation

- Based on the work of Jan de Wit
- Written in Haskell 98 using `UU_Scanner`, `UU_Parsing` and `UU_Pretty` (and therefore needing GHC/Hugs extensions)
- Is supposed to be a quick hack in order to produce results soon
- We try to follow Hinze's "Habilitationsschrift" which contains a chapter outlining implementation issues

Implementation — continued

Input A single file (`.ghs`) written in a subset of Haskell
(for example no modules, no type classes, only builtin infix operators)
with extensions for defining generic functions

Step 1 The file is parsed and grouped to

- **data** definitions
- **polyvalue** definitions
  (i. e. MPC-style generic function definitions)
- other stuff

Implementation — continued

Input A single file (`.ghs`) written in a subset of Haskell
(for example no modules, no type classes, only builtin infix operators)
with extensions for defining generic functions

Step 1 The file is parsed and grouped to

- **data** definitions
- **polyvalue** definitions
  (i. e. MPC-style generic function definitions)
- other stuff

Implementation — continued

Step 2 All defined datatypes are translated into structural equivalent types constructed only out of binary sums and products from constant types, i. e. no constructor names, no field labels.
Furthermore, we provide embedding functions from a "real" datatype to its structural equivalent type.

Step 3 The **polyvalue** definitions are translated line by line into ordinary Haskell functions.

Step 4 For every pair of a generic function and a datatype definition a specialisation is generated from the function to the datatype.
(This is possible because MPC-style definitions are not specific to a kind!)

Output A single file (`.hs`) containing Haskell 98 with extensions
(we need rank-2 type signatures)

Implementation — continued

Step 2 All defined datatypes are translated into structural equivalent types constructed only out of binary sums and products from constant types, i. e. no constructor names, no field labels.
Furthermore, we provide embedding functions from a "real" datatype to its structural equivalent type.

Step 3 The **polyvalue** definitions are translated line by line into ordinary Haskell functions.

Step 4 For every pair of a generic function and a datatype definition a specialisation is generated from the function to the datatype.
(This is possible because MPC-style definitions are not specific to a kind!)

Output A single file (`.hs`) containing Haskell 98 with extensions (we need rank-2 type signatures)

Implementation — continued

Step 2 All defined datatypes are translated into structural equivalent types constructed only out of binary sums and products from constant types, i. e. no constructor names, no field labels.
Furthermore, we provide embedding functions from a "real" datatype to its structural equivalent type.

Step 3 The **polyvalue** definitions are translated line by line into ordinary Haskell functions.

Step 4 For every pair of a generic function and a datatype definition a specialisation is generated from the function to the datatype.
(This is possible because MPC-style definitions are not specific to a kind!)

Output A single file (`.hs`) containing Haskell 98 with extensions (we need rank-2 type signatures)

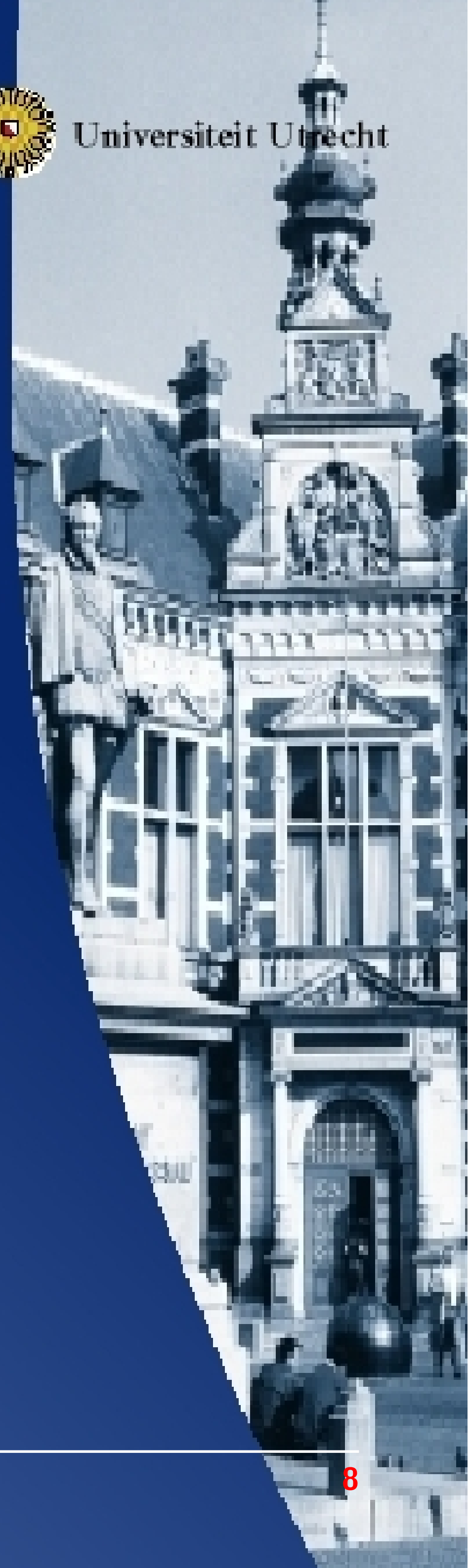
Implementation — continued

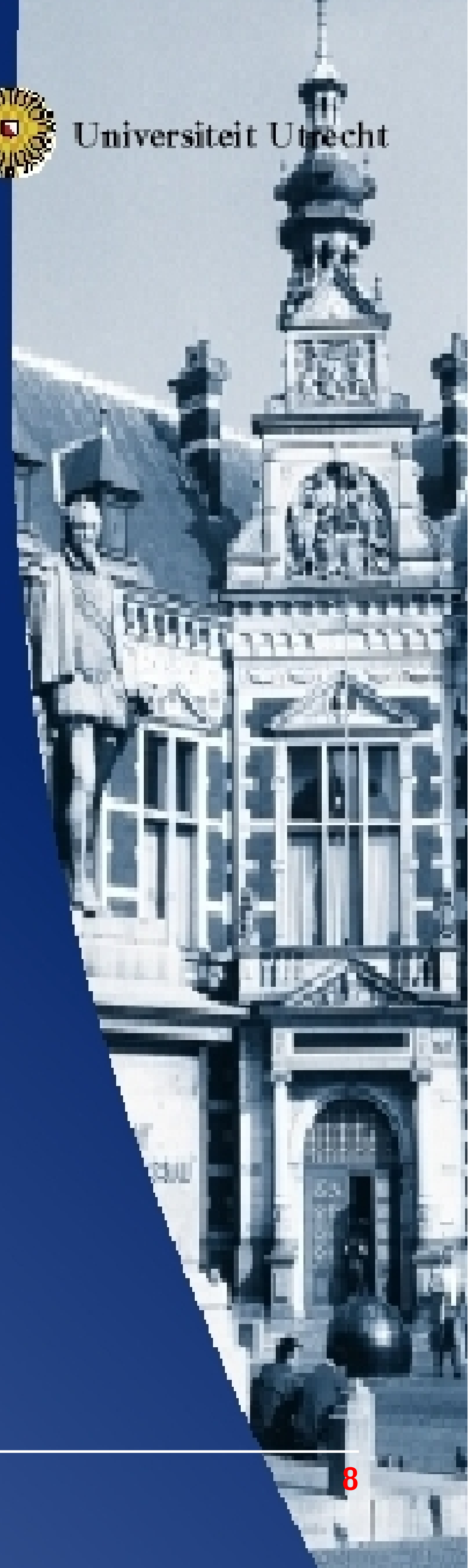
Step 2 All defined datatypes are translated into structural equivalent types constructed only out of binary sums and products from constant types, i. e. no constructor names, no field labels.
Furthermore, we provide embedding functions from a "real" datatype to its structural equivalent type.

Step 3 The **polyvalue** definitions are translated line by line into ordinary Haskell functions.

Step 4 For every pair of a generic function and a datatype definition a specialisation is generated from the function to the datatype.
(This is possible because MPC-style definitions are not specific to a kind!)

Output A single file (`.hs`) containing Haskell 98 with extensions (we need rank-2 type signatures)





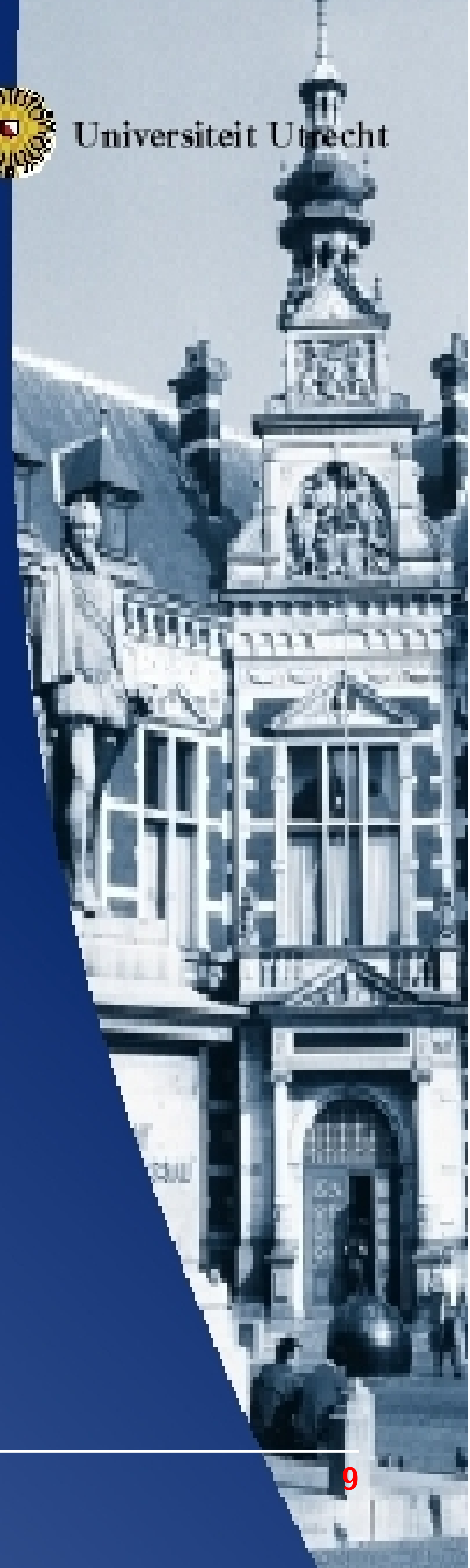
Implementation — continued

Step 2 All defined datatypes are translated into structural equivalent types constructed only out of binary sums and products from constant types, i. e. no constructor names, no field labels.
Furthermore, we provide embedding functions from a "real" datatype to its structural equivalent type.

Step 3 The **polyvalue** definitions are translated line by line into ordinary Haskell functions.

Step 4 For every pair of a generic function and a datatype definition a specialisation is generated from the function to the datatype.
(This is possible because MPC-style definitions are not specific to a kind!)

Output A single file (`.hs`) containing Haskell 98 with extensions
(we need rank-2 type signatures)

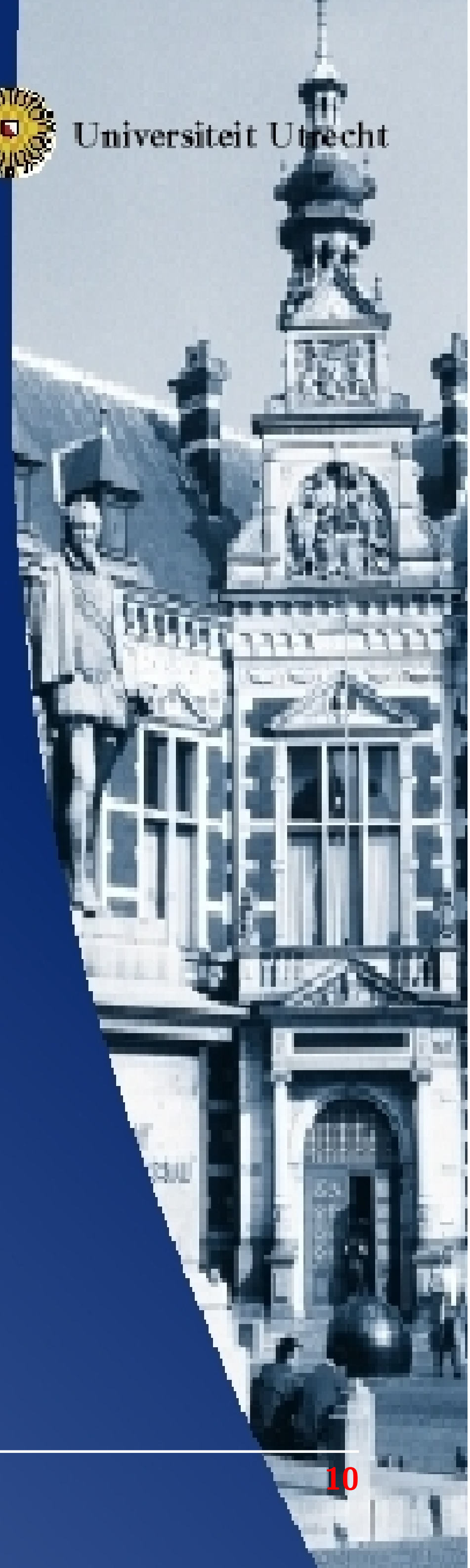


Example (Syntax demonstration)

We revisit the *encode* function definition from the beginning:

$$
\begin{array}{lll}
Encode\langle \kappa :: \Box \rangle & :: & \kappa \rightarrow \star \\
Encode\langle \star \rangle \; t & = & t \rightarrow [\,Bit\,] \\
Encode\langle \kappa_1 \rightarrow \kappa_2 \rangle \; t & = & \forall a. Encode\langle \kappa_1 \rangle \; a \rightarrow Encode\langle \kappa_2 \rangle \; (t \; a)
\end{array}
$$

$$
\begin{array}{lll}
encode\langle\!\langle t :: \kappa \rangle\!\rangle & :: & Encode\langle \kappa \rangle \; t \\
encode\langle\!\langle 1 \rangle\!\rangle & = & [\,] \\
encode\langle\!\langle + \rangle\!\rangle \; eA \; eB \; (inl \; a) & = & \mathtt{O} : eA \; a \\
encode\langle\!\langle + \rangle\!\rangle \; eA \; eB \; (inr \; b) & = & \mathtt{I} : eB \; b \\
encode\langle\!\langle \times \rangle\!\rangle \; eA \; eB \; (a, b) & = & eA \; a \mathbin{+\!\!+} eB \; b
\end{array}
$$



Example (Syntax demonstration) — continued

This is how that transforms into the syntax of the prototype compiler:

```
polyvalue encode {| t |} :: t -> [Bit]

encode{| 1 |}             x           = []
encode{| + |} eA eB (LEFT  xl)    = O : (eA xl)
encode{| + |} eA eB (RIGHT xr)    = I : (eB xr)
encode{| * |} eA eB (PROD x1 x2) = eA x1 ++ eB x2
```

Example (Syntax demonstration) — continued

This is how that transforms into the syntax of the prototype compiler:

```
polyvalue encode {| t |} :: t -> [Bit]

encode{| 1 |}             x         = []
encode{| + |} eA eB (LEFT  xl)    = O : (eA xl)
encode{| + |} eA eB (RIGHT xr)    = I : (eB xr)
encode{| * |} eA eB (PROD x1 x2) = eA x1 ++ eB x2
```

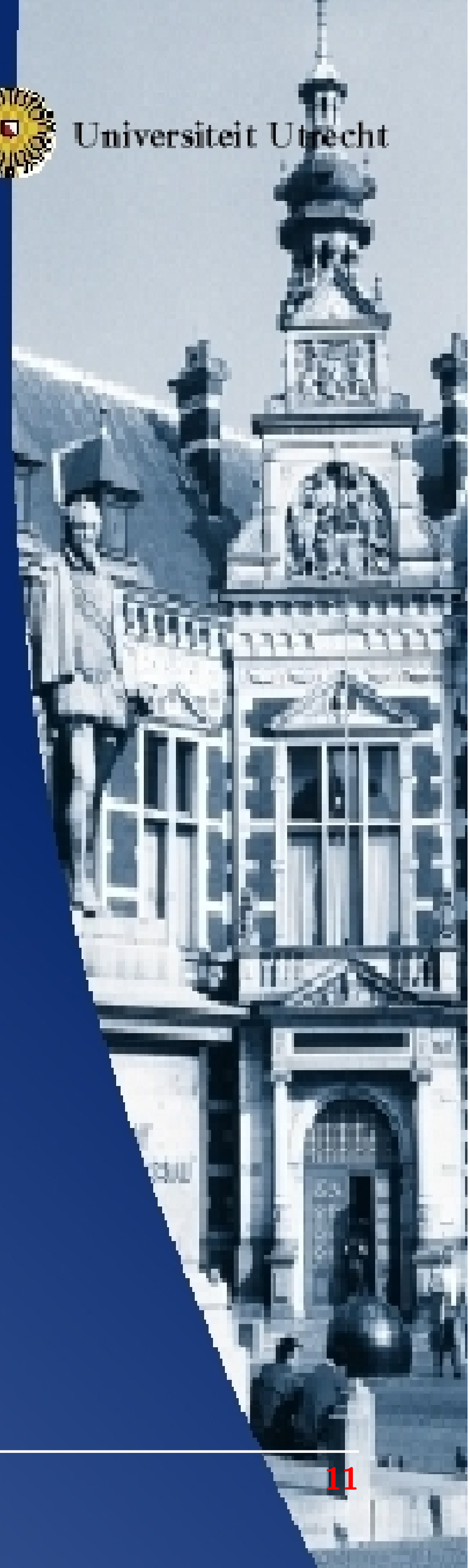
We provide the following datatypes:

```
data Bit        = O | I
data List     a = Nil | Cons a (List a)
data GRose c a = Node a (c (GRose c a))
```

Example (Call of compiler)

Now we can call the compiler on this file:

```
$ gh2hs --verbose EncodeTalk.ghs
```



Example (Call of compiler)

Now we can call the compiler on this file:

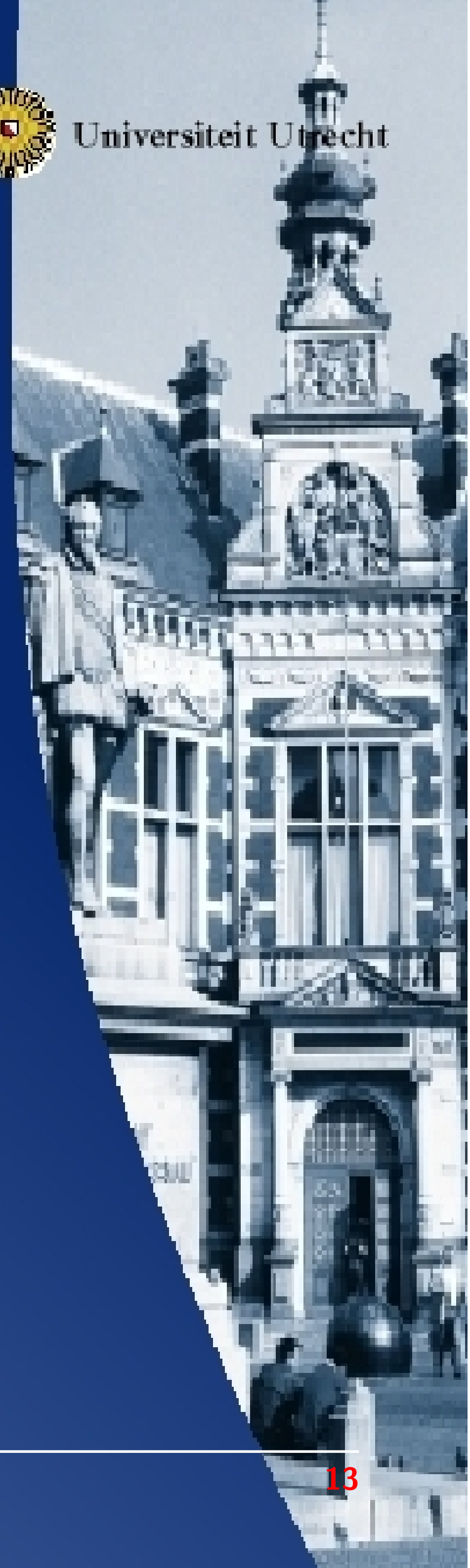
```
$ gh2hs --verbose EncodeTalk.ghs
Generic Haskell compiler, version 0.0.4
Options are: [Verbose]
Scanned.
Parsed.

File EncodeTalk.ghs read.
Kinds inferred.
Structure types generated.
Isomorphisms generated.
Iso-adapters for kind-*-datatypes generated.
Type-synonyms for polytypic values generated.
Iso-adapters generated.
Components generated.
Requirements analyzed.
Specializations generated.
$ _
```

Example (Generated code)

The inferred kinds of the datatypes are written as a comment to the output file. The builtin list and pair types are added to the list.

```
-- Datatypes
data Bit  = O | I
data List a = Nil | Cons a (List a)
data GRose f a = Node a (f (GRose f a))
-- LIST :: (* -> *)
-- PAIR :: (* -> (* -> *))
-- Bit :: *
-- List :: (* -> *)
-- GRose :: ((* -> *) -> (* -> *))
```



Example (Generated code) — continued

These are the generated structural types. Note that the translations of the builtin list type and the user-defined one are identical.

```
-- Structure types
type LIST__ a = SUM UNIT (PROD a (LIST a))
type PAIR__ a b = PROD a b
type Bit__  = SUM UNIT UNIT
type List__ a = SUM UNIT (PROD a (List a))
type GRose__ f a = PROD a (f (GRose f a))
```

Universiteit Utrecht

Example (Generated code) — continued

Isomorphisms for mapping types to the structural types are generated.
These are lifted to the type of generic functions.

```
-- Type synonyms for polytypic values
type EncodeType t = t -> [Bit]

-- Iso-adapters for polyvalues
isoMapEncodeType :: Iso a1 a1__ -> Iso (EncodeType a1) (EncodeType a1__)
isoMapEncodeType isoMapt = ((isoMapFUN isoMapt) (isoMapLST isoMapBit))
```

Example (Generated code) — continued

The translation of the lines of the generic function definition is almost trivial.

```
-- Components
encodeUNIT :: EncodeType UNIT
encodeUNIT x = []
encodeSUM :: EncodeType a1 -> EncodeType b1 -> EncodeType (SUM a1 b1)
encodeSUM eA eB (LEFT xl) = (O:(eA xl))
encodeSUM eA eB (RIGHT xr) = (I:(eB xr))
encodePROD :: EncodeType a1 -> EncodeType b1 -> EncodeType (PROD a1 b1)
encodePROD eA eB (PROD x1 x2) = ((eA x1)++(eB x2))
```

Universiteit Utrecht

Example (Generated code) — continued

Finally, we get a look at the specialised functions for *Bit*, *List* and *GRose*.

```
encodeBit :: Bit -> [Bit]
encodeBit  = ((osi (isoMapEncodeType isoBit)) encodeBit__)
encodeBit__ :: Bit__ -> [Bit]
encodeBit__  = ((encodeSUM encodeUNIT) encodeUNIT)
encodeList :: (a01 -> [Bit]) -> List a01 -> [Bit]
encodeList encodea = ((osi (isoMapEncodeType isoList))
                          (encodeList__ encodea))
encodeList__ :: (a01 -> [Bit]) -> List__ a01 -> [Bit]
encodeList__ encodea = ((encodeSUM encodeUNIT)
                          ((encodePROD encodea) (encodeList encodea)))
encodeGRose :: (forall a11 . (a11 -> [Bit]) -> a01 a11 -> [Bit])
            -> (a21 -> [Bit]) -> GRose a01 a21 -> [Bit]
encodeGRose encodef encodea = ((osi (isoMapEncodeType isoGRose))
                                  ((encodeGRose__ encodef) encodea))
encodeGRose__ :: (forall a11 . (a11 -> [Bit]) -> a01 a11 -> [Bit])
              -> (a21 -> [Bit]) -> GRose__ a01 a21 -> [Bit]
encodeGRose__ encodef encodea = ((encodePROD encodea)
                                  (encodef ((encodeGRose encodef) encodea)))
```

Example (Usage of generated code)


```
> encodeBit O
O
> encodeBit I
I
> encodeList encodeBit $ I `Cons` (I `Cons` (O `Cons` Nil))
IIIIIOO
> let
    emptyO = Node O Nil; emptyI = Node I Nil
  in
    encodeGRose encodeList encodeBit
      $ Node I (emptyI `Cons` (emptyI `Cons` (emptyO `Cons` Nil)))
IIIOIIOIOOO
> _
```

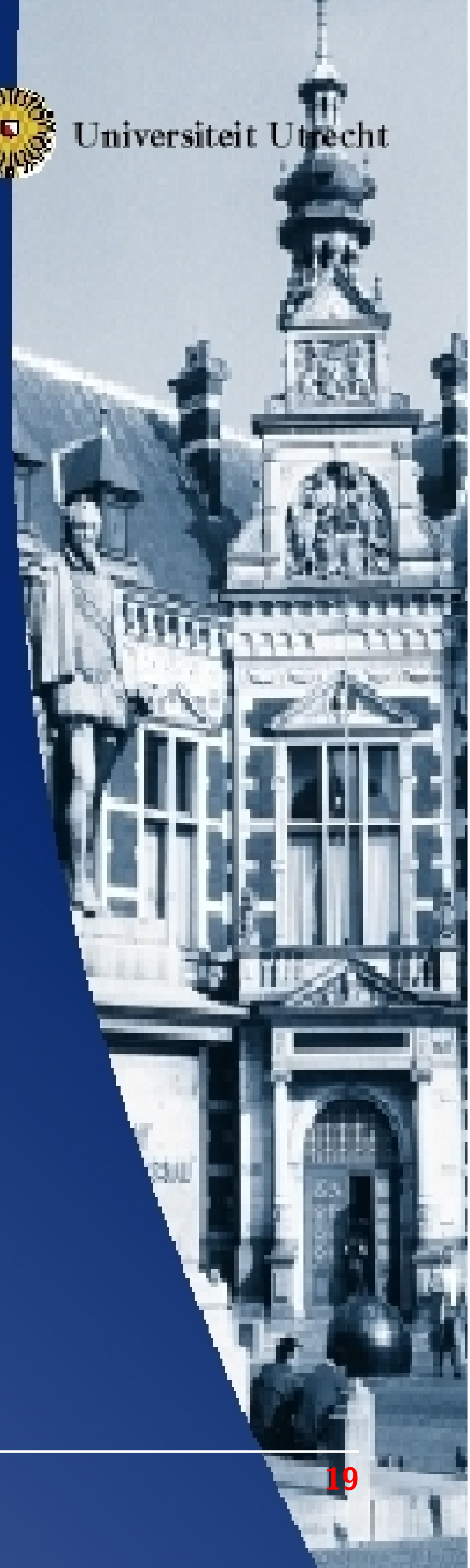
Example (Usage of generated code)

```
> encodeBit O
O
> encodeBit I
I
> encodeList encodeBit $ I `Cons` (I `Cons` (O `Cons` Nil))
IIIIIOO
> let
    emptyO = Node O Nil; emptyI = Node I Nil
  in
    encodeGRose encodeList encodeBit
      $ Node I (emptyI `Cons` (emptyI `Cons` (emptyO `Cons` Nil)))
IIIOIIOIOOO
> _
```

- Names of generated functions are built by extending the name of the function with the name of the type.
- Nothing (yet) is done to prevent name clashes.

Future goals (decreasing priority)

- Wipe out some small technical deficiencies
- Clean up code a bit, switch to Attribute Grammar system
- Extend the parser to parse more or less Haskell 98
  plus generic programming extensions
- Provide fixed-kind instantiations of MPC-style generic definitions
- Allow for better control over the specialisation process
- Add typechecking



Some remaining deficiencies are no problems

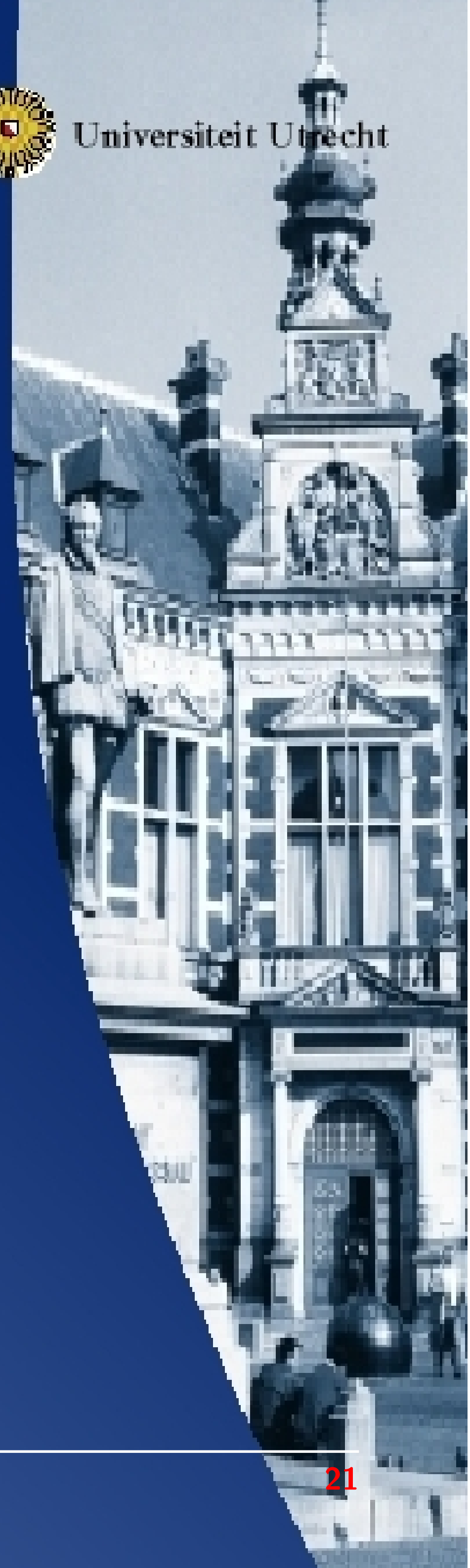
Hinze already provides solutions in his thesis for

- mapping more-than-rank-2 type signatures to rank-2 type signatures
- allowing user-defined types in the signatures of generic functions
- providing access to constructor names and labels of fields (needed for a reimplementation of *show*)

Extending the parser

We need a better parser. It should parse most of Haskell 98 to make the prototype usable for Haskell users.

- `hsparser` by Sven Panne, Simon Marlow and Noel Winstanley is a `happy`-based parser for full Haskell 1.4 that could be adapted
- Write a parser ourselves, using `UU_Parsing`
  (and we could have a more suitable abstract syntax)



More genericity

Let's have a look at the MPC-style generic function *count*:

$$
\begin{array}{ll}
Count\langle \kappa :: \Box \rangle & :: \kappa \rightarrow \star \\
Count\langle \star \rangle \; t & = t \rightarrow Int \\
Count\langle \kappa_1 \rightarrow \kappa_2 \rangle \; T & = \forall a. Count\langle \kappa_1 \rangle \; a \rightarrow Count\langle TB \rangle \; (t \; a)
\end{array}
$$

$$
\begin{array}{ll}
count\langle\!\langle t :: \kappa \rangle\!\rangle & :: Count\langle \kappa \rangle \; t \\
count\langle\!\langle 1 \rangle\!\rangle & = [\,] \\
count\langle\!\langle + \rangle\!\rangle \; cA \; cB \; (inl \; a) & = cA \; a \\
count\langle\!\langle + \rangle\!\rangle \; cA \; cB \; (inr \; b) & = cB \; b \\
count\langle\!\langle \times \rangle\!\rangle \; eA \; eB \; (a, b) & = cA \; a + cB \; b
\end{array}
$$

We want to be able to write the (still generic) instances *sum* and *size* of *count* for $\star \rightarrow \star$-kinded types:

$$
\begin{array}{ll}
sum\langle\!\langle f :: \star \rightarrow \star \rangle\!\rangle & :: f \; Int \rightarrow Int \\
sum\langle\!\langle f \rangle\!\rangle & = count\langle\!\langle f \rangle\!\rangle \; id
\end{array}
$$

$$
\begin{array}{ll}
size\langle\!\langle f :: \star \rightarrow \star \rangle\!\rangle & :: F \; A \rightarrow Int \\
size\langle\!\langle f \rangle\!\rangle & = count\langle\!\langle f \rangle\!\rangle \; (const \; 1)
\end{array}
$$

Specialise the specialisation process

- Give the user special syntax to use the generic functions at a specific type everywhere (we need to parse more than one file)
- Specialisations to applied type contructors are also possible: write $encode\langle\!\langle List\ \ Int\rangle\!\rangle$ rather than $encode\langle\!\langle List\rangle\!\rangle\ \ encode\langle\!\langle Int\rangle\!\rangle$
- Let the user leave out the type, but that requires . . .

Typechecking

- Possibilities have to be investigated
- Providing typechecking for full Haskell 98 will probably be a lot of work

Thank you for listening