

Typing Haskell with an Attribute Grammar

Atze Dijkstra and S. Doaitse Swierstra

Institute of Information and Computing Sciences,
Utrecht University,
P.O.Box 80.089,
Padualaan 14, Utrecht, Netherlands,
{atze,doaitse}@cs.uu.nl,
WWW home page: <http://www.cs.uu.nl>

Abstract. A great deal has been written about type systems. Much less has been written about implementing them. Even less has been written about implementations of complete compilers in which all aspects come together. This paper fills this gap by describing the implementation of a series of compilers for a simplified variant of Haskell. By using an attribute grammar system, aspects of a compiler implementation can be described separately and added in a sequence of steps, thereby giving a series of increasingly complex (working) compilers. Also, the source text of both this paper and the executable compilers come from the same source files by an underlying minimal weaving system. Therefore, source and explanation is kept consistent.

1 Introduction and overview

Haskell98 [31] is a complex language, not to mention its more experimental incarnations. Though also intended as a research platform, realistic compilers for Haskell [1] have grown over the years and understanding and experimenting with those compilers is not an easy task. Experimentation on a smaller scale usually is based upon relatively simple and restricted implementations [20], often focusing only on a particular aspect of the language and/or its implementation. This paper aims at walking somewhere between this complexity and simplicity by

- Describing the implementation of essential aspects of Haskell (or any other (functional) programming language), hence the name Essential Haskell (EH) used for simplified variants of Haskell¹ in these notes.
- Describing these aspects separately in order to provide a better understanding.
- Adding these aspects on top of each other in an incremental way, thus leading to a sequence of compilers, each for a larger subset of complete Haskell (and extensions).

¹ The 'E' in EH might also be expanded to other aspects of the compiler, like being an **Example**.

- Using tools like the Utrecht University Attribute Grammar (UUAG) system [3], hereafter referred to as the AG system, to allow for separate descriptions for the various aspects.

The remaining sections of this introduction will expand on this by looking at the intentions, purpose and limitations of these notes in more detail. This is followed by a short description of the individual languages for which we develop compilers throughout these notes. The last part of the introduction contains a small tutorial on the AG system used in these notes. After the introduction we continue with discussing the implementation of the first three compilers (sections 2, 3 and 4) out of a (currently) sequence of ten compilers. On the web site [11] for this project the full distribution of the code for these compilers can be found. We conclude these notes by reflecting upon our experiences with the AG system and the creation of these notes (section 5).

1.1 Purpose

For whom is this material intended?

- For students who wish to learn more about the implementation of functional languages. This paper also informally explains the required theory, in particular about type systems.
- For researchers who want to build (e.g.) a prototype and to experiment with extensions to the type system and need a non-trivial and realistic starting point. This paper provides documentation, design rationales and an implementation for such a starting point.
- For those who wish to study a larger example of the tools used to build the compilers in these notes. We demonstrate the use of the AG system, which allows us to separately describe the various aspects of a language implementation. Other tools for maintaining consistency between different versions of the resulting compilers and the source code text included in these notes are also used, but will not be discussed.

For this intended audience these notes provide:

- A description of the implementation of a type checker/inferencer for a subset of Haskell. We describe the first three languages of a (currently) sequence of ten, that end in a full implementation of an extended Haskell.
- A description of the semantics of Haskell, lying between the more formal [16,14] and more implementation oriented [21,33] and similar to other combinations of theory and practice [34].
- A gradual instead of a big bang explanation.
- Empirical support for the belief that the complexity of a compiler can be managed by splitting the implementation of the compiler into separate aspects.

- A working combination of otherwise usually separately proven or implemented features.

We will come back to this in our conclusion (see section 5).

We restrict ourselves in the following ways, partly because of space limitations, partly by design:

- We do not discuss extensions to Haskell implemented in versions beyond the last version presented in these notes. See section 1.3 for a preciser description of what can and cannot be found in these notes with respect to Haskell features.
- We concern ourselves with typing only. Other aspects, like pretty printing and parsing, are not discussed. However, the introduction to the AG system (see section 1.4) gives some examples of the pretty printing and the interaction between parsing, AG code and Haskell code.
- We do not deal with type theory or parsing theory as a subject on its own. This paper is intended to describe “how to implement” and will use theory from that point of view. Theoretical aspects are touched upon from a more intuitive point of view.

Although informally and concisely introduced where necessary, familiarity with the following will make reading and understanding these notes easier:

- Functional programming, in particular using Haskell
- Compiler construction in general
- Type systems, λ -calculus
- Parser combinator library and AG system [3,38]

For those not familiar with the AG system a short tutorial has been included at the end of this introduction (see section 1.4). It also demonstrates the use of the parser combinators used throughout the implementation of all EH versions.

We expect that by finding a balance between theory and implementation, we serve both those who want to learn and those who want to do research. It is also our belief that by splitting the big problem into smaller aspects the combination can be explained in an easier way.

In the following sections we give examples of the Haskell features present in the series of compilers described in the following chapters. Only short examples are given, so the reader gets an impression of what is explained in more detail and implemented in the relevant versions of the compiler.

1.2 A short tour

Though all compilers described in these notes deal with a different issue, they all have in common that they are based on the λ -calculus, most of the time using the syntax

and semantics of Haskell. The first version of our series of compilers therefore accepts a language that most closely resembles the λ -calculus, in particular typed λ -calculus extended with **let** expressions and some basic types and type constructors such as *Int*, *Char* and tuples.

EH version 1: λ -calculus. An EH program is a single expression, contrary to a Haskell program which consists of a set of declarations forming a module.

```
let i :: Int
    i = 5
in i
```

All variables need to be typed explicitly; absence of an explicit type is considered to be an error. The corresponding compiler (EH version 1, section 2) checks the explicit types against actual types.

Besides the basic types *Int* and *Char*, composite types can be formed by building tuples and defining functions:

```
let id :: Int → Int
    id =  $\lambda x \rightarrow x$ 
    fst :: (Int, Char) → Int
    fst =  $\lambda(a, b) \rightarrow a$ 
in id (fst (id 3, 'x'))
```

Functions accept one parameter only, which can be a pattern. All types are monomorphic.

EH version 2: Explicit/implicit typing. The next version (EH version 2, section 3) no longer requires the explicit type specifications, which thus may have to be inferred by the compiler.

The reconstructed type information is monomorphic, for example the identity function in:

```
let id =  $\lambda x \rightarrow x$ 
in let v = id 3
    in id
```

is inferred to have the type *id* :: *Int* → *Int*.

EH version 3: Polymorphism. The third version (EH version 3, section 4) performs standard Hindley-Milner type inferencing [8,9] which also supports parametric polymorphism. For example,

```
let id =  $\lambda x \rightarrow x$ 
in id 3
```

is inferred to have type *id* :: $\forall a. a \rightarrow a$.

1.3 Haskell language elements not described

As mentioned before, only a subset of the full sequence of compilers is described in these notes. Currently, as part of an ongoing work [11], in the compilers following the compilers described in these notes, the following Haskell features are dealt with:

- EH 4.** Quantifiers everywhere: higher ranked types [36,32,7,28] and existentials [30,25,27].
See also the longer version of these notes handed out during the AFP04 summer-school [13].
- EH 5.** Data types.
- EH 6.** Kinds, kind inference, kind checking, kind polymorphism.
- EH 7.** Non extensible records, subsuming tuples.
- EH 8.** Code generation for a GRIN (Graph Reduction Intermediate Notation) like backend [6,5].
- EH 9.** Class system, explicit implicit parameters [12].
- EH 10.** Extensible records [15,22].

Also missing are features which fall in the category syntactic sugar, programming in the large and the like. Haskell incorporates many features which make programming easier and/or manageable. Just to mention a few:

- Binding group analysis
- Syntax directives like infix declarations
- Modules [10,37].
- Type synonyms
- Syntactic sugar for **if**, **do**, list notation and comprehension.

We have deliberately not dealt with these issues. Though necessary and convenient we feel that these features should be added after all else has been dealt with, so as not to make understanding and implementing essential features more difficult.

1.4 An AG mini tutorial

The remaining part of the introduction contains a small tutorial on the AG system. The tutorial explains the basic features of the AG system. The explanation of remaining features is postponed to its first use throughout the main text. These places are marked with *AG*. The tutorial can safely be skipped if the reader is already familiar with the AG system.

Haskell and Attribute Grammars (AG). Attribute grammars can be mapped onto functional programs [23,19,4]. Vice versa, the class of functional programs (catamorphisms [39]) mapped onto can be described by attribute grammars. The AG system exploits this correspondence by providing a notation (attribute grammar) for computations over trees

which additionally allows program fragments to be described separately. The AG compiler gathers these fragments, combines these fragments, and generates a corresponding Haskell program.

In this AG tutorial we start with a small example Haskell program (of the right form) to show how the computation described by this program can be expressed in the AG notation and how the resulting Haskell program generated by the AG compiler can be used. The ‘repmim’ problem [4] is used for this purpose. A second example describing a ‘pocket calculator’ (that is, expressions) focusses on more advanced features and typical AG usage patterns.

Repmim a la Haskell. Repmim stands for “replacing the integer valued leaves of a tree by the minimal integer value found in the leaves”. The solution to this problem requires two passes over a tree structure, computing the minimum and computing a new tree with the minimum as its leaves respectively. It is often used as the typical example of a circular program which lends itself well to be described by the AG notation. When described in Haskell it is expressed as a computation over a tree structure:

```
data Tree = Tree_Leaf Int
          | Tree_Bin  Tree Tree
deriving Show
```

The computation itself simultaneously computes the minimum of all integers found in the leaves of the tree and the new tree with this minimum value. The result is returned as a tuple computed by function *r*:

```
repmim :: Tree → Tree
repmim t
  = t'
where (t', tmin) = r t tmin
        r (Tree_Leaf i) m = (Tree_Leaf m, i)
        r (Tree_Bin lt rt) m = (Tree_Bin lt' rt', lmin 'min' rmin)
        where (lt', lmin) = r lt m
              (rt', rmin) = r rt m
```

We can use this function in some setting, for example:

```
tr      = Tree_Bin (Tree_Leaf 3) (Tree_Bin (Tree_Leaf 4) (Tree_Leaf 5))
tr'     = repmim tr
main :: IO ()
main = print tr'
```

The resulting program produces the following output:

```
Tree_Bin (Tree_Leaf 3) (Tree_Bin (Tree_Leaf 3) (Tree_Leaf 3))
```

The computation of the new tree requires the minimum. This minimum is passed as a parameter *m* to *r* at the root of the tree by extracting it from the result of *r*. The result

tuple of the invocation $r\ t\ tmin$ depends on itself via the minimum $tmin$ so it would seem we have a cyclic definition. However, the real dependency is not on the tupled result of r but on its elements because it is the element $tmin$ of the result tuple which is passed back and not the tuple itself. The elements are not cyclically dependent so Haskell's laziness prevents a too eager computation of the elements of the tuple which might otherwise have caused an infinite loop during execution. Note that we have two more or less independent computations that both follow the tree structure, and a weak interaction, when passing the $tmin$ value back in the tree.

Repmín a la AG. The structure of *repmín* is similar to the structure required by a compiler. A compiler performs several computations over an *abstract syntax tree (AST)*, for example for computing its type and code. This corresponds to the *Tree* structure used by *repmín* and the tupled results. In the context of attribute grammars the elements of this tuple are called *attribute's*. Occasionally the word *aspect* is used as well, but an aspect may also refer to a group of attributes associated with one particular feature of the AST, language or problem at hand.

Result elements are called *synthesized* attributes. On the other hand, a compiler may also require information that becomes available at higher nodes in an AST to be available at lower nodes in an AST. The m parameter passed to r in *repmín* is an example of this situation. In the context of attribute grammars this is called an *inherited attribute*.

Using AG notation we first define the AST corresponding to our problem (for which the complete compilable solution is given in Fig. 1):

```
DATA Tree
  | Leaf int : {Int}
  | Bin lt : Tree
    rt : Tree
```

The **DATA** keyword is used to introduce the equivalent of Haskell's **data** type. A **DATA** $\langle node \rangle$ defines a *node* $\langle node \rangle$ (or *nonterminal*) of an AST. Its alternatives, enumerated one by one after the vertical bar |, are called *variants*, *productions*. The term *constructor* is occasionally used to stress the similarity with its Haskell counterpart. Each variant has members, called *children* if they refer to other nodes of the AST and *fields* otherwise. Each child and field has a name (before the colon) and a type (after the colon). The type may be either another **DATA** node (if a child) or a monomorphic Haskell type (if a field), delimited by curly braces. The curly braces may be omitted if the Haskell type is a single identifier. For example, the **DATA** definition for the *repmín* problem introduces a node (nonterminal) *Tree*, with variants (productions) *Leaf* and *Bin*. A *Bin* has children *lt* and *rt* of type *Tree*. A *Leaf* has no children but contains only a field *int* holding a Haskell *Int* value.

The keyword **ATTR** is used to declare an attribute for a node, for instance the synthesized attribute *min*:

```
ATTR Tree [| min : Int]
```

```

DATA Tree
  | Leaf int : {Int}
  | Bin lt : Tree
    rt : Tree

ATTR Tree [| min : Int]

SEM Tree
  | Leaf lhs . min = @int
  | Bin lhs . min = @lt.min 'min' @rt.min

ATTR Tree [rmin : Int ||]
  -- The next SEM may be generated automatically

SEM Tree
  | Bin lt . rmin = @lhs.rmin
    rt . rmin = @lhs.rmin

DATA Root
  | Root tree : Tree

SEM Root
  | Root tree . rmin = @tree.min

ATTR Root Tree [| tree : Tree]

SEM Tree
  | Leaf lhs . tree = Tree_Leaf@lhs.rmin
  | Bin lhs . tree = Tree_Bin @lt.tree @rt.tree
  -- The next SEM may be generated automatically

SEM Root
  | Root lhs . tree = @tree.tree

DERIVING Tree : Show
{
|Tree_Bin (Tree_Leaf 3) (Tree_Bin (Tree_Leaf 4) (Tree_Leaf 5))
tr' = sem_Root (Root_Root tr)
main :: IO ()
main = print tr'
}


```

Fig. 1. Full AG specification of `repmin`

SEM Tree

```
| Leaf lhs.min = @int  
| Bin lhs.min = @lt.min 'min' @rt.min
```

A synthesized attribute is declared for the node after **ATTR**. Multiple declarations of the same attribute for different nonterminals can be grouped on one line by enumerating the nonterminals after the **ATTR** keyword, separated by whitespace. The attribute declaration is placed inside the square brackets at one or more of three different possible places. All attributes before the first vertical bar | are inherited, after the last bar synthesized, and in between both inherited and synthesized. For example, attribute *min* is a result and therefore positioned as a synthesized attribute, after the last bar.

Rules relating an attribute to its value are introduced using the keyword **SEM**. For each production we distinguish a set of input attributes, consisting of the synthesized attributes of the children referred to by $@\langle child \rangle.\langle attr \rangle$ and the inherited attributes of the parent referred to by $@\mathbf{lhs}.\langle attr \rangle$. For each output attribute we need a rule that expresses its value in terms of input attributes and fields.

The computation for a synthesized attribute for a node has to be defined for each variant individually as it usually will differ between variants. Each rule is of the form

$$|\langle variant \rangle \langle node \rangle.\langle attr \rangle = \langle Haskell \text{ expr} \rangle$$

If multiple rules are declared for a $\langle variant \rangle$ of a node, the $\langle variant \rangle$ part may be shared. The same holds for multiple rules for a child (or **lhs**) of a $\langle variant \rangle$, the child (or **lhs**) may then be shared.

The text representing the computation for an attribute has to be a Haskell expression and will end up almost unmodified in the generated program, without any form of checking. Only attribute and field references, starting with a @, have meaning to the AG system. The text, possibly stretching over multiple lines, has to be indented at least as far as its first line. Otherwise it is to be delimited by curly braces.

The basic form of an attribute reference is $@\langle node \rangle.\langle attr \rangle$ referring to a synthesized attribute $\langle attr \rangle$ of child node $\langle node \rangle$. For example, $@lt.min$ refers to the synthesized attribute *min* of child *lt* of the *Bin* variant of node *Tree*.

The $\langle node \rangle.$ part of $@\langle node \rangle.\langle attr \rangle$ may be omitted. For example, *min* for the *Leaf* alternative is defined in terms of $@int$. In that case $@\langle attr \rangle$ refers to a locally (to a variant for a node) declared attribute, or to the field with the same name as defined in the **DATA** definition for that variant. This is the case for the *Leaf* variant's *int*. We postpone the discussion of locally declared attributes.

The minimum value of *repm* passed as a parameter corresponds to an inherited attribute *rmin*:

```
ATTR Tree [rmin : Int ||]
```

The value of *rmin* is straightforwardly copied to its children. This “simply copy” behavior occurs so often that we may omit its specification. The AG system uses so called

copy rules to automatically generate code for copying if the value of an attribute is not specified explicitly. This is to prevent program clutter and thus allows the programmer to focus on programming the exception instead of the usual. We will come back to this later; for now it suffices to mention that all the rules for *rmin* might as well have been omitted.

The original *repm* function did pass the minimum value coming out *r* back into *r* itself. This did happen at the top of the tree. Similarly we define a *Root* node sitting on top of a *Tree*:

```
DATA Root
  | Root tree : Tree
```

At the root the *min* attribute is passed back into the tree via attribute *rmin*:

```
SEM Root
  | Root tree.rmin = @tree.min
```

The value of *rmin* is used to construct a new tree:

```
ATTR Root Tree [|| tree : Tree]
SEM Tree
  | Leaf lhs.tree = Tree_Leaf@lhs.rmin
  | Bin lhs.tree = Tree_Bin @lt.tree @rt.tree
```

```
SEM Root
  | Root lhs.tree = @tree.tree
```

For each **DATA** the AG compiler generates a corresponding Haskell **data** type declaration. For each node *<node>* a data type with the same name *<node>* is generated. Since Haskell requires all constructors to be unique, each constructor of the data type gets a name of the form *<node>_<variant>*.

In our example the constructed tree is returned as the one and only attribute of *Root*. It can be shown if we tell the AG compiler to make the generated data type an instance of the *Show* class:

```
DERIVING Tree : Show
```

Similarly to the Haskell version of *repm* we can now show the result of the attribute computation as a plain Haskell value by using the function *sem_Root* generated by the AG compiler:

```
{
tr = Tree_Bin (Tree_Leaf 3) (Tree_Bin (Tree_Leaf 4) (Tree_Leaf 5))
tr' = sem_Root (Root_Root tr)
main :: IO ()
main = print tr'
}
```

Because this part is Haskell code it has to be delimited by curly braces, indicating that the AG compiler should copy it unchanged into the generated Haskell program.

In order to understand what is happening here, we take a look at the generated Haskell code. For the above example the following code will be generated (edited to remove clutter):

```

data Root = Root_Root (Tree)
-- semantic domain
type T_Root = ( (Tree))
-- cata
sem_Root :: (Root) -> (T_Root)
sem_Root ((Root_Root (_tree)))
  = (sem_Root_Root ((sem_Tree (_tree))))
sem_Root_Root :: (T_Tree) -> (T_Root)
sem_Root_Root (tree_) =
  let ( _treeImin,_treeItree) = (tree_ (_treeOrmin))
      (_treeOrmin) = _treeImin
      (_lhs0tree) = _treeItree
  in ( _lhs0tree)

data Tree = Tree_Bin (Tree) (Tree)
          | Tree_Leaf (Int)
          deriving ( Show)
-- semantic domain
type T_Tree = (Int) -> ( (Int),(Tree))
-- cata
sem_Tree :: (Tree) -> (T_Tree)
sem_Tree ((Tree_Bin (_lt) (_rt)))
  = (sem_Tree_Bin ((sem_Tree (_lt))) ((sem_Tree (_rt))))
sem_Tree ((Tree_Leaf (_int))) = (sem_Tree_Leaf (_int))
sem_Tree_Bin :: (T_Tree) -> (T_Tree) -> (T_Tree)
sem_Tree_Bin (lt_) (rt_) =
  \ _lhsIrmin ->
    let ( _ltImin,_ltItree) = (lt_ (_ltOrmin))
        ( _rtImin,_rtItree) = (rt_ (_rtOrmin))
        (_lhs0min) = _ltImin 'min' _rtImin
        (_rtOrmin) = _lhsIrmin
        (_ltOrmin) = _lhsIrmin
        (_lhs0tree) = Tree_Bin _ltItree _rtItree
    in ( _lhs0min,_lhs0tree)
sem_Tree_Leaf :: (Int) -> (T_Tree)
sem_Tree_Leaf (int_) =
  \ _lhsIrmin ->

```

```

let (_lhs0min) = int_
    (_lhs0tree) = Tree_Leaf _lhsIrmin
in  ( _lhs0min, _lhs0tree)

```

In general, generated code is not the most pleasant² of prose to look at, but we will have to use the generated functions in order to access the AG computations of attributes from the Haskell world. The following observations should be kept in mind when doing so:

- For node $\langle node \rangle$ also a type $T_{\langle node \rangle}$ is generated, describing the function type that maps inherited to synthesized attributes. This type corresponds one-to-one to the attributes defined for $\langle node \rangle$: inherited attributes to parameters, synthesized attributes to elements of the result tuple (or single type if exactly one synthesized attribute is defined).
- Computation of attribute values is done by semantic functions with a name of the form $sem_{\langle node \rangle \langle variant \rangle}$. These functions have exactly the same type as their constructor counterpart of the generated data type. The only difference lies in the parameters which are of the same type as their constructor counterpart, but prefixed with T_{\cdot} . For example, data constructor $Tree_Bin :: Tree \rightarrow Tree \rightarrow Tree$ corresponds to the semantic function $sem_Tree_Bin :: (T_Tree) \rightarrow (T_Tree) \rightarrow (T_Tree)$.
- A mapping from the Haskell **data** type to the corresponding semantic function is available with the name $sem_{\langle node \rangle}$.

In the Haskell world one now can follow different routes to compute the attributes:

- First construct a Haskell value of type $\langle node \rangle$, then apply $sem_{\langle node \rangle}$ to this value and the additionally required inherited attributes values. The given function *main* from AG variant of *repmim* takes this approach.
- Circumvent the construction of Haskell values of type $\langle node \rangle$ by using the semantic functions $sem_{\langle node \rangle \langle variant \rangle}$ directly when building the AST instead of the data constructor $\langle node \rangle \langle variant \rangle$ (This technique is called *deforestation* [42]).

In both cases a tuple holding all synthesized attributes is returned. Elements in the tuple are sorted lexicographically on attribute name, but it still is awkward to extract an attribute via pattern matching because the size of the tuple and position of elements changes with adding and renaming attributes. For now, this is not a problem as sem_Root will only return one value, a *Tree*. Later we will see the use of wrapper functions to pass inherited attributes and extract synthesized attributes via additional wrapper data types holding attributes in labeled fields.

² In addition, because generated code can be generated differently, one cannot count on it being generated in a specific way. Such is the case here too, this part of the AG implementation may well change in the future.

Parsing directly to semantic functions. The given *main* function uses the first approach: construct a *Tree*, wrap it inside a *Root*, and apply *sem.Root* to it. The following example takes the second approach; it parses some input text describing the structure of a tree and directly invokes the semantic functions:

```
instance Symbol Char
  pRepmIn :: IsParser p Char => p T_Root
  pRepmIn = pRoot
    where pRoot = sem.Root.Root ($) pTree
          pTree = sem.Tree_Leaf ($) pInt
                ⋈ sem.Tree_Bin  ($) pSym 'B' ⋈ pTree ⋈ pTree
          pInt  = (λc → ord c - ord '0') ($) '0' ⟨..⟩ '9'
```

The parser recognises the letter 'B' as a *Bin* alternative and a single digit as a *Leaf*. Fig. 2 gives an overview of the parser combinators which are used [38]. The parser is invoked from an alternative implementation of *main*:

```
main :: IO ()
main = do tr ← parseIOMessage show pRepmIn "B3B45"
        print tr
```

We will not discuss this alternative further nor will we discuss this particular variant of parser combinators. However, this approach is taken in the rest of these notes wherever parsing is required.

Combinator	Meaning	Result
$p \langle * \rangle q$	p followed by q	result of p applied to result of q
$p \langle \vee \rangle q$	p or q	result of p or result of q
$pSucceed\ r$	empty input ε	r
$f \langle \$ \rangle p$	$\equiv pSucceed\ f \langle * \rangle p$	
$pKey\ "x"$	symbol/keyword x	" x "
$p \langle ** \rangle q$	p followed by q	result of q applied to result of p
$p \langle opt \rangle r$	$\equiv p \langle \vee \rangle pSucceed\ r$	
$p \langle ? \rangle q$	$\equiv p \langle ** \rangle q \langle opt \rangle id$	
$p \langle * \rangle q, p \langle * \rangle q, f \langle \$ \rangle p$	variants throwing away result of angle missing side	
$pFoldr\ listAlg\ p$	sequence of p 's	$foldr\ c\ n$ (result of all p 's)
$pList\ p$	$pFoldr\ (:), []\ p$	
$pChainr\ s\ p$	p 's (>1) separated by s 's	result of s 's applied to results of p 's aside

Fig. 2. Parser combinators

More features and typical usage: a pocket calculator. We will continue with looking at a more complex example, a pocket calculator which accepts expressions. The calculator prints a pretty printed version of the entered expression, its computed value and some statistics (the number of additions performed). An interactive terminal session of the pocket calculator looks as follows:

```
$ build/bin/expr
Enter expression: 3+4
Expr='3+4', val=7, add count thus far=1
Enter expression: [a=3+4:a+a]
Expr='[a=3+4:a+a]', val=14, add count thus far=3
Enter expression: ^Cexpr: interrupted
$
```

This rudimentary calculator allows integer values, their addition and binding to identifiers. Parsing is character based, no scanner is used to transform raw text into tokens. No whitespace is allowed and a **let** expression is syntactically denoted by [`<nm>=<expr>:<expr>`].

The example will allow us to discuss more AG features as well as typical use of AG. We start with integer constants, addition followed by an attribute computation for the pretty printing:

```
DATA AGIf
  | AGIf expr : Expr
DATA Expr
  | IConst int : {Int}
  | Add e1 : Expr e2 : Expr
SET AllNT = AGIf Expr
```

The root of the tree is now called *AGIf* to indicate (as a naming convention) that this is the place where interfacing between the Haskell world and the AG world takes place.

The definition demonstrates the use of the **SET** keyword which allows the naming of a group of nodes. This name can later be used to declare attributes for all the named group of nodes at once.

The computation of a pretty printed representation follows the same pattern as the computation of *min* and *tree* in the *repm* example, because of its compositional and bottom-up nature. The synthesized attribute *pp* is synthesized from the values of the *pp* attribute of the children of a node:

```
ATTR AllNT [| pp : PP_Doc]
SEM Expr
  | IConst lhs.pp = pp @int
  | Add lhs.pp = @e1.pp >|< "+" >|< @e2.pp
```

The pretty printing uses a pretty printing library with combinators for values of type *PP_Doc* representing pretty printed documents. The library is not further discussed here; an overview of some of the available combinators can be found in Fig. 3.

Combinator	Result
$p_1 \succ\! \!< p_2$	p_1 besides p_2 , p_2 at the right
$p_1 \succ\!# \!< p_2$	same as $\succ\! \!<$ but with an additional space in between
$p_1 \succ\!< p_2$	p_1 above p_2
$pp_parens\ p\ p$	p inside parentheses
$text\ s$	string s as <i>PP_Doc</i>
$pp\ x$	pretty print x (assuming instance <i>PP x</i>) resulting in a <i>PP_Doc</i>

Fig. 3. Pretty printing combinators

As a next step we add **let** expressions and use of identifiers in expressions. This demonstrates an important feature of the AG system: we may introduce new alternatives for a *<node>* as well as may introduce new attribute computations in a separate piece of program text. We first add new AST alternatives for *Expr*:

```
DATA Expr
| Let nm : {String} val : Expr body : Expr
| Var nm : {String}
```

One should keep in mind that the exensibility offered is simplistic of nature, but surprisingly flexible at the same time. The idea is that node variants, attribute declarations and attribute rules for node variants can all occur textually separated. The AG compiler gathers all definitions, combines, performs several checks (e.g. are attribute rules missing), and generates the corresponding Haskell code. All kinds of declarations can be distributed over several text files to be included with a **INCLUDE** directive (not discussed any further).

Any addition of new node variants requires also the corresponding definitions of already introduced attributes:

```
SEM Expr
| Let lhs.pp = "[" >|< @nm >|< "=" >|< @val.pp >|< ":" >|< @body.pp >|< "]"
| Var lhs.pp = pp @nm
```

The use of variables in the pocket calculator requires us to keep an administration of values bound to variables. An association list is used to provide this environmental and scoped information:

```
ATTR Expr [env : [(String, Int)]] ||]
SEM Expr
| Let body.env = (@nm, @val.val) : @lhs.env
```

SEM *AGIf*
 | *AGIf* *expr .env* = []

The scope is enforced by extending the inherited attribute *env* top-down in the AST. Note that there is no need to specify a value for *@val.env* because of the copy rules discussed later. In the *Let* variant the inherited environment, which is used for evaluating the right hand sided of the bound expression, is extended with the new binding, before being used as the inherited *env* attribute of the body. The environment *env* is queried when the value of an expression is to be computed:

ATTR *AllNT* [| *val* : *Int* |]
SEM *Expr*
 | *Var* **lhs.val** = *maybe 0 id (lookup @nm @lhs.env)*
 | *Add* **lhs.val** = *@e1.val + @e2.val*
 | *Let* **lhs.val** = *@body.val*
 | *ICnst* **lhs.val** = *@int*

The attribute *val* holds this computed value. Because its value is needed in the ‘outside’ Haskell world it is passed through *AGIf* (as part of **SET** *AllNT*) as a synthesized attribute. This is also the case for the previously introduced *pp* attribute as well as the following *count* attribute used to keep track of the number of additions performed. However, the *count* attribute is also passed as an inherited attribute. Being both inherited and synthesized it is defined between the two vertical bars in the **ATTR** declaration for *count*:

ATTR *AllNT* [| *count* : *Int* |]
SEM *Expr*
 | *Add* **lhs.count** = *@e2.count + 1*

The attribute *count* is said to be *threaded* through the AST, the AG solution to a global variable or the use of state monad. This is a result of the attribute being inherited as well as synthesized and the copy rules. Its effect is an automatic copying of the attribute in a preorder traversal of the AST.

Copy rules are attribute rules inserted by the AG system if a rule for an attribute *<attr>* in a production of *<node>* is missing. AG tries to insert a rule that copies the value of another attribute with the same name, searching in the following order:

1. Local attributes.
2. The synthesized attribute of the children to the left of the child for which an inherited *<attr>* definition is missing, with priority given to the nearest child fulfilling the condition. A synthesized *<attr>* of a parent is considered to be at the right of any child’s *<attr’>*.
3. Inherited attributes (of the parent).

In our example the effect is that for the *Let* variant of *Expr*

- (inherited) `@lhs.count` is copied to (inherited) `@val.count`,
- (synthesized) `@val.count` is copied to (inherited) `@body.count`,
- (synthesized) `@body.count` is copied to (synthesized) `@lhs.count`.

Similar copy rules are inserted for the other variants. Only for variant *Add* of *Expr* a different rule for `@lhs.count` is explicitly specified, since here we have a non-trivial piece of semantics: i.e. we actually want to count something.

Automatic copy rule insertion can be both a blessing and curse. A blessing because it takes away a lot of tedious work and minimises clutter in the AG source text. On the other hand it can be a curse, because a programmer may have forgotten an otherwise required rule. If a copy rule can be inserted the AG compiler will silently do so, and the programmer will not be warned.

As with our previous example we can let a parser map input text to the invocations of semantic functions. For completeness this source text has been included in Fig. 4. The result of parsing combined with the invocation of semantic functions will be a function taking inherited attributes to a tuple holding all synthesized attributes. Even though the order of the attributes in the result tuple is specified, its extraction via pattern matching should be avoided. The AG system can be instructed to create a wrapper function which knows how to extract the attributes out of the result tuple:

WRAPPER *AGItf*

The attribute values are stored in a data type with labeled fields for each attribute. The attributes can be accessed with labels of the form `<attr>_Syn_<node>`. The name of the wrapper is of the form `wrap_<node>`; the wrapper function is passed the result of the semantic function and a data type holding inherited attributes:

```
run :: Int → IO ()
run count
  = do hPutStr stdout "Enter expression: "
       hFlush stdout
       l ← getLine
       r ← parseIOMessage show pAGItf l
       let r' = wrap_AGItf r (Inh_AGItf{count_Inh_AGItf = count})
           putStrLn ("Expr=" ++ disp (pp_Syn_AGItf r') 40 "" ++
                    "' , val=" ++ show (val_Syn_AGItf r') ++
                    ", add count thus far=" ++ show (count_Syn_AGItf r')
           )
       run (count_Syn_AGItf r')

main :: IO ()
main = run 0
```

We face a similar problem with the passing of inherited attributes to the semantic function. Hence inherited attributes are passed to the wrapper function via a data type with name `Inh_<node>` and a constructor with the same name, with fields having labels of the

form $\langle attr \rangle Inh.\langle node \rangle$. The *count* attribute is an example of an attribute which must be passed as an inherited attribute as well as extracted as a synthesized attribute.

This concludes our introduction to the AG system. Some topics have either not been mentioned at all or only shortly touched upon. We provide a list of those topics together with a reference to the first use of the features which are actually used later in these notes. Each of these items is marked with \mathcal{AG} to indicate that it is about the AG system.

- Type synonym, only for lists (see section 14).
- Left hand side patterns for simultaneous definition of rules (see section 42).
- Set notation for variant names in rules (see section 31).
- Local attributes (see section 31).
- Additional copy rule via **USE** (see section 3.2).
- Additional copy rule via **SELF** (see section 3.2).
- Rule redefinition via $:=$ (see section 3.3).
- Cycle detection and other (experimental) features, commandline invocation, etc.

We will come back to the AG system itself in our conclusion.

```

instance Symbol Char
  pAGIf :: IsParser p Char => p T AGIf
  pAGIf = pRoot
    where pRoot      = sem_AGIf_AGIf ($) pExpr
          pExpr      = pChainr (sem_Expr_Add ($) pSym '+' ) pExprBase
          pExprBase = (sem_Expr_IConst.foldl (\l r -> l * 10 + r) 0)
                    ($) pList1 ((\lc -> ord c - ord '0') ($) '0' \_.) '9')
                    \() sem_Expr_Let
                    ($) pSym '[' \() pNm \* pSym '=' \() pExpr
                    \* pSym ':' \() pExpr
                    \* pSym ']'
                    \() sem_Expr_Var ($) pNm
          pNm        = (:"") ($) 'a' \_.) 'z'

```

Fig. 4. Parser for calculator example

2 EH 1: typed λ -calculus

In this section we build the first version of our series of compilers: the typed λ -calculus packaged in Haskell syntax in which all values need to explicitly be given a type. The compiler checks if the specified types are in agreement with actual value definitions. For example

```

let i :: Int
    i = 5
in i

```

is accepted, whereas

```

let i :: Char
    i = 5
in i

```

produces a pretty printed version of the erroneous program, annotated with errors:

```

let i :: Char
    i = 5
    {- ***ERROR(S):
        In '5':
        Type clash:
        failed to fit: Int <= Char
        problem with : Int <= Char -}
    {- [ i:Char ] -}
in i

```

Type signatures have to be specified for identifiers bound in a **let** expression. For λ -expressions the type of the parameter can be extracted from these type signatures unless a λ -expression occurs at the position of an applied function. In that case a type signature for the λ -expression is required in the expression itself. This program will not typecheck because this EH version does not allow polymorphic types in general and on higher ranked (that is, parameter) positions in particular.

```

let v :: (Int, Char)
    v = ( (\f -> (f 3, f 'x'))
         :: (Char -> Char) -> (Int, Char)
         ) (\x -> x)
in v

```

The implementation of a type system will be the main focus of this and following sections. As a consequence the full environment/framework needed to build a compiler will not be discussed. This means in particular that error reporting, generation of a pretty printed annotated output, parsing and the compiler driver are not described.

We start with the definition of the AST and how it relates to concrete syntax, followed by the introduction of several attributes required for the implementation of the type system.

2.1 Concrete and abstract syntax

The *concrete syntax* of a (programming) language describes the structure of acceptable sentences for that language, or more down to earth, it describes what a compiler for that language accepts with respect to the textual structure. On the other hand, *abstract syntax* describes the structure used by the compiler itself for analysis and code generation. Translation from the more user friendly concrete syntax to the machine friendly abstract syntax is done by a parser; from the abstract to the concrete representation is done by a pretty printer.

Let us focus our attention first on the abstract syntax for EH1, in particular the part defining the structure for expressions (the remaining syntax can be found in Fig. 5).

```
DATA Expr
  | IConst int    : {Int}
  | CConst char  : {Char}
  | Con nm       : {HsName}
  | Var nm       : {HsName}
  | App func     : Expr
    arg         : Expr
  | Let decls    : Decls
    body       : Expr
  | Lam arg      : PatExpr
    body       : Expr
  | AppTop expr  : Expr
  | Parens expr  : Expr
  | TypeAs tyExpr : TyExpr
    expr       : Expr
```

Integer constants are represented by *IConst*, lowercase (uppercase) identifier occurrences by *Var* (*Con*), an *App* represents the application of a function to its argument, *Lam* and *Let* represent lambda expressions and let expressions.

AG: Type synonyms (for lists). The AG notation allows type synonyms for one special case, AG's equivalent of a list. It is an often occurring idiom to encode a list of nodes, say **DATA** *L* with elements *<node>* as:

```
DATA L
  | Cons hd : <node>
    tl    : L
  | Nil
```

AG allows the following notation as a shorthand:

```
TYPE L = [ <node> ]
```

The EH fragment (which is incorrect for this version of because type signatures are missing)

```
let ic @(i,c) = (5, 'x')
    id      =  $\lambda x \rightarrow x$ 
in id i
```

is represented by the following piece of abstract syntax tree:

```
AGItf_AGItf
  Expr_Let
    Decls_Cons
      Decl_Val
        PatExpr_VarAs "ic"
          PatExpr_AppTop
            PatExpr_App
              PatExpr_App
                PatExpr_Con ",2"
                PatExpr_Var "i"
                PatExpr_Var "c"
            Expr_AppTop
              Expr_App
                Expr_App
                  Expr_Con ",2"
                  Expr_IConst 5
                  Expr_CConst 'x'
          Decls_Cons
            Decl_Val
              PatExpr_Var "id"
            Expr_Lam
              PatExpr_Var "x"
              Expr_Var "x"
          Decls_Nil
      Expr_AppTop
        Expr_App
          Expr_Var "id"
          Expr_Var "i"
```

The example also demonstrates the use of patterns, which is almost the same as in Haskell: EH does not allow a type signature for the elements of a tuple.

Looking at this example and the rest of the abstract syntax in Fig. 5 we can make several observations of what one is allowed to write in EH and what can be expected from the implementation.

```

DATA AGlTf
  | AGlTf   expr   : Expr

DATA Decl
  | TySig   nm     : {HsName}
                tyExpr : TyExpr
  | Val     patExpr : PatExpr
                expr   : Expr

TYPE Decls   = [Decl]
SET AllDecl  = Decl Decls

DATA PatExpr
  | IConst  int    : {Int}
  | CConst  char   : {Char}
  | Con     nm     : {HsName}
  | Var     nm     : {HsName}
  | VarAs   nm     : {HsName}
                patExpr : PatExpr
  | App     func   : PatExpr
                arg    : PatExpr
  | AppTop  patExpr : PatExpr
  | Parens  patExpr : PatExpr

SET AllPatExpr = PatExpr

DATA TyExpr
  | Con     nm     : {HsName}
  | App     func   : TyExpr
                arg    : TyExpr
  | AppTop  tyExpr : TyExpr
  | Parens  tyExpr : TyExpr

SET AllTyExpr = TyExpr
SET AllExpr  = Expr
SET AllNT    = AllTyExpr AllDecl AllPatExpr AllExpr

```

Fig. 5. Abstract syntax for EH (without Expr)

- There is a striking similarity between the structure of expressions *Expr* and patterns *PatExpr* (and as we will see later type expressions *TyExpr*): they all contain *App* and *Con* variants. This similarity will sometimes be exploited to factor out common code, and, if factoring out cannot be done, leads to similarities between pieces of code. This is the case with pretty printing (not included in these notes), which is quite similar for the different kinds of constructs.
- Type signatures (*Decl_TySig*) and value definitions (*Decl_Val*) may be freely mixed. However, type signatures and value definitions for the same identifier are still related.
- Because of the textual decoupling of value definitions and type signatures, a type signature may specify the type for an identifier occurring inside a pattern:

```

let a      :: Int
      (a, b) = (3, 4)
in ...

```

Currently we do not allow this, but the following however is:

```

let ab      :: (Int, Int)
      ab @(a, b) = (3, 4)
in ...

```

because the specified type for *ab* corresponds to the top of a pattern of a value definition.

- In EH composite values are created by tupling, denoted by (\dots) . The same notation is also used for patterns (for unpacking a composite value) and types (describing the structure of the composite). In all these cases the corresponding AST consists of a *Con* applied to the elements of the tuple. For example, the value $(2, 3)$ corresponds to

$$\text{Expr_App (Expr_App (Expr_Con " , 2") (Expr_IConst 2)) (Expr_IConst 3)}$$

- For now there is only one value constructor: for tuples. The EH constructor for tuples also is the one which needs special treatment because it actually stands for an infinite family of constructors. This can be seen in the encoding of the name of the constructor which is composed of a " , " together with the arity of the constructor. For example, the expression $(3, 4)$ is encoded as an application *App* of *Con " , 2"* to the two *Int* arguments: $(2\ 3\ 4)$. In our examples we will follow the Haskell convention, in which we write $(.)$ instead of $'2'$. By using this encoding we also get the unit type $()$ as it is encoded by the name $" , 0"$.
- The naming convention for tuples and other naming conventions are available through the following definitions for Haskell names *HsName*.

```

data HsName = HNm String
           deriving (Eq, Ord)
instance Show HsName where

```

```

show (HNm s) = s
hsnArrow, hsnUnknown, hsnInt, hsnChar, hsnWild :: HsName
hsnArrow           = HNm "->"
hsnUnknown         = HNm "???"
hsnInt             = HNm "Int"
hsnChar            = HNm "Char"
hsnWild            = HNm "_"

hsnProd            :: Int → HsName
hsnProd i          = HNm (' ', ' ': show i)

hsnIsArrow, hsnIsProd :: HsName → Bool
hsnIsArrow hsn     = hsn ≡ hsnArrow
hsnIsProd (HNm (' ', ' ': _)) = True
hsnIsProd _        = False

hsnProdAry         :: HsName → Int
hsnProdAry (HNm (_ : ar)) = read ar

```

- Each application is wrapped on top with an *AppTop*. This has no meaning in itself but it simplifies the pretty printing of expressions³. We need *AppTop* for patterns, but for the rest it can be ignored.
- The location of parentheses around an expression is remembered by a *Parens* alternative. We need this for the reconstruction of the parenthesis in the input.
- *AGItf* is the top of a complete abstract syntax tree. As noted in the AG primer this is the place where interfacing with the ‘outside’ Haskell world takes place. It is a convention in these notes to give all nonterminals in the abstract syntax a name with *AGItf* in it, if it plays a similar role.

2.2 Types

We will now turn our attention to the way the type system is incorporated into EH1. We focus on the pragmatics of the implementation and less on the corresponding type theory.

What is a type Compiler builders consider a *type* to be a description of the interpretation of a value whereas a value is to be understood as a bitpattern. This means in particular that machine operations such as integer addition, are only applied to patterns that are to be interpreted as integers. More generally, we want to prevent unintended interpretations of bitpatterns, which might likely lead to the crash of a program.

The flow of values, that is, the copying between memory locations, through the execution of a program may only be such that a copy is allowed only if the corresponding types relate to each other in some proper fashion. A compiler uses a type system to analyse this flow and to make sure that built-in functions are only applied to patterns

³ As it also complicates parsing it may disappear in future versions of EH.

that they are intended to work on. The idea is that if a compiler cannot find an erroneous flow of values, with the notion of erroneous defined by the type system, the program is guaranteed not to crash because of unintended use of bitpatterns.

In this section we start by introducing a type language in a more formal setting as well as a more practical setting. The formal setting uses typing rules to specify the static semantics of EH whereas in the practical setting the AG system is used, providing an implementation. In the following section we discuss the typing rules, the mechanism for enforcing the equality of types (called *fitting*) and the checking itself. Types will be introduced informally, instead of taking a more formal approach [40,41,34,2].

Types are described by a type language. The type language for EH1 allows some basic types and two forms of composite types, functions and tuples, and is described by the following grammar:

$$\begin{aligned} \sigma &= \text{Int} \mid \text{Char} \\ &\mid (\sigma, \dots, \sigma) \\ &\mid \sigma \rightarrow \sigma \end{aligned}$$

The following definition however is closer to the one used in our implementation:

$$\begin{aligned} \sigma &= \text{Int} \mid \text{Char} \mid \rightarrow \mid , \mid , , \mid \dots \\ &\mid \sigma \sigma \end{aligned}$$

The latter definition also introduces the possibility of describing types like *Int Int*. We nevertheless use this one since it is used in the implementation of later versions of EH where it will prove useful in expressing the application of type constructors to types. Here we just have to make sure no types like *Int Int* will be created; in a (omitted) later version of EH we perform kind inferencing/checking to prevent the creation of such types from showing up.

The corresponding encoding using AG notation differs in the presence of an *Any* type, also denoted by \square . In section 2.3 we will say more about this. It is used to smoothen the type checking by (e.g.) limiting the propagation of erroneous types:

DATA *TyAGItf*
 $\mid \text{AGItf } ty : \text{Ty}$

DATA *Ty*
 $\mid \text{Con } nm : \{\text{HsName}\}$
 $\mid \text{App } func : \text{Ty}$
 $\quad \quad \quad arg : \text{Ty}$
 $\mid \text{Any}$

The formal system and implementation of this system use different symbols to refer to the same concept. For example, *Any* in the implementation is the same as \square in the typing rules. Not always is such a similarity pointed out explicitly but instead a notation *name₁//name₂* is used to simultaneously refer to both symbols *name₁* and *name₂*, for

example $Any//\square$. The notation also implies that the identifiers and symbols separated by $'//'$ are referring to the same concept.

The definition of Ty will be used in both the Haskell world and the AG world. In Haskell we use the corresponding **data** type generated by the AG compiler, for example in the derived type TyL :

type $TyL = [Ty]$

The data type is used to construct type representations. In the AG world we define computations over the type structure in terms of attributes. The corresponding semantic functions generated by the AG system can then be applied to Haskell values.

2.3 Checking types

The type system of a programming language is described by typing rules. A *typing rule*

- Relates language constructs to types.
- Constrains the types of these language constructs.

Type rules For example, the following is the typing rule (taken from Fig. 6) for function application

$$\frac{\begin{array}{l} \Gamma \stackrel{expr}{\vdash} e_2 : \sigma^a \\ \Gamma \stackrel{expr}{\vdash} e_1 : \sigma^a \rightarrow \sigma \end{array}}{\Gamma \stackrel{expr}{\vdash} e_1 e_2 : \sigma} \quad (\text{e-app1})$$

It states that an application of e_1 to e_2 has type σ provided that the argument has type σ^a and the function has a type $\sigma^a \rightarrow \sigma$.

All rules we will use are of the form

$$\frac{\begin{array}{l} prerequisite_1 \\ prerequisite_2 \\ \dots \end{array}}{consequence} \quad (\text{rule-name})$$

with the meaning that if all $prerequisite_i$ can be proven we may conclude the *consequence*.

A *prerequisite* can take the form of any logical predicate or has a more structured form, usually called a *judgement*:

$$context \stackrel{judgetype}{\vdash} construct : property \rightsquigarrow more\ results$$

The part “ $\rightsquigarrow more\ results$ ” needs not always be present if there are no more results for a judgement. The notation reads as

$$\boxed{\Gamma \stackrel{expr}{\vdash} e : \sigma}$$

$$\frac{\Gamma \stackrel{expr}{\vdash} e_2 : \sigma^a}{\Gamma \stackrel{expr}{\vdash} e_1 : \sigma^a \rightarrow \sigma} \text{ (e-app1)} \quad \frac{i \mapsto \sigma^i, \Gamma \stackrel{expr}{\vdash} e : \sigma^e}{\Gamma \stackrel{expr}{\vdash} \lambda i \rightarrow e : \sigma^i \rightarrow \sigma^e} \text{ (e-lam1)}$$

$$\frac{\Gamma \stackrel{expr}{\vdash} e_2 : \sigma_2}{\Gamma \stackrel{expr}{\vdash} e_1 : \sigma_1} \text{ (e-prod1)} \quad \frac{i \mapsto \sigma^i, \Gamma \stackrel{expr}{\vdash} e^i : \sigma^i}{\Gamma \stackrel{expr}{\vdash} \mathbf{let} \ i :: \sigma^i; \ i = e^i \mathbf{in} \ e : \sigma^e} \text{ (e-let1)}$$

$$\frac{(i \mapsto \sigma) \in \Gamma}{\Gamma \stackrel{expr}{\vdash} i : \sigma} \text{ (e-ident1)} \quad \frac{}{\Gamma \stackrel{expr}{\vdash} \mathbf{minint} .. \mathbf{maxint} : \mathbf{Int}} \text{ (e-int1)}$$

Fig. 6. Type rules for expressions

In the interpretation *judgetype* the *construct* has property *property* assuming *context* and with optional additional *more results*.

If the *context* or *more results* itself consists of multiple parts, these parts are separated by a semicolon ';'. An underscore '_' has a similar role as in Haskell to indicate a property is not relevant for a type rule (see rule e-app1B, Fig. 7)

Although a rule formally is to be interpreted purely equational, it may help to realise that from an implementors point of view this (more or less) corresponds to an implementation template, either in the form of a function *judgetype*:

$$\begin{aligned} \mathit{judgetype} &= \lambda \mathit{construct} \rightarrow \\ &\quad \lambda \mathit{context} \rightarrow \dots(\mathit{property}, \mathit{more_results}) \end{aligned}$$

or a piece of AG:

$$\mathbf{ATTR} \ \mathit{judgetype} \ [\mathit{context} : \dots \parallel \mathit{property} : \dots \mathit{more_results} : \dots]$$

SEM *judgetype*

| *construct*

$$\mathbf{lhs} . (\mathit{property}, \mathit{more_results}) = \dots @\mathbf{lhs} . \mathit{context} \dots$$

Typing rules and implementation templates differ in that the latter prescribes the order in which the computation of a property takes place, whereas the former simply postulates

relationships between parts of a rule. In general typing rules presented throughout these notes will be rather explicit in the flow of information and thus be close to the actual implementation.

Environment The rules in Fig. 6 refer to Γ , which is often called *assumptions*, *environment* or *context* because it provides information about what may be assumed about identifiers. Identifiers ξ are distinguished on the case of the first character, capitalized I 's starting with an uppercase, uncapitalized i 's otherwise

$$\begin{array}{l} \xi = i \\ | I \end{array}$$

For type constants we will use capitalized identifiers I , whereas for identifiers bound to an expression in a **let**-expression we will use lower case identifiers (i, j, \dots) .

An environment Γ is a vector of bindings, a partial finite map from identifiers to types:

$$\Gamma = \overline{\xi \mapsto \sigma}$$

Concatenation of such collections as well as scrutinizing a collection is denoted with a comma ','. For example, ' $i \mapsto \sigma, \Gamma$ ' represents a concatenation as well as a pattern match. For rules this does not make a difference, for the implementation there is a direction involved as we either construct from smaller parts or deconstruct (pattern match) into smaller parts.

If shadowing is involved, that is duplicate entries are added, left/first (w.r.t. to the comma ',') entries shadow right/later entries. In particular, when we locate some variable in a Γ the first occurrence will be taken.

If convenient we will also use a list notation:

$$\Gamma = [\xi \mapsto \sigma]$$

This will be done if specific properties of a list are used or if we borrow from Haskell's repertoire of list functions. For simplicity we also use (association) lists in our implementation.

A list structure suffices to encode the presence of an identifier in a Γ , but it cannot be used to detect multiple occurrences caused by duplicate introductions. Thus in our implementation we use a stack of lists instead:

type *AssocL* $k\ v = [(k, v)]$

newtype *Gam* $k\ v = \text{Gam } [\text{AssocL } k\ v]$ **deriving** *Show*

emptyGam $:: \text{Gam } k\ v$

gamUnit $:: k \rightarrow v \rightarrow \text{Gam } k\ v$

gamLookup $:: \text{Eq } k \Rightarrow k \rightarrow \text{Gam } k\ v \rightarrow \text{Maybe } v$

gamToAssocL $:: \text{Gam } k\ v \rightarrow \text{AssocL } k\ v$

<i>gamPushNew</i>	$:: \text{Gam } k \ v \rightarrow \text{Gam } k \ v$	
<i>gamPushGam</i>	$:: \text{Gam } k \ v \rightarrow \text{Gam } k \ v \rightarrow \text{Gam } k \ v$	
<i>gamAddGam</i>	$:: \text{Gam } k \ v \rightarrow \text{Gam } k \ v \rightarrow \text{Gam } k \ v$	
<i>gamAdd</i>	$:: k \rightarrow v \rightarrow \text{Gam } k \ v \rightarrow \text{Gam } k \ v$	
<i>emptyGam</i>		$= \text{Gam } [[]]$
<i>gamUnit</i>	$k \ v$	$= \text{Gam } [(k, v)]$
<i>gamLookup</i>	$k \ (\text{Gam } ll)$	$= \text{foldr } (\lambda mv \rightarrow \text{maybe } mv \ \text{Just } (\text{lookup } k \ l))$ $\text{Nothing } ll$
<i>gamToAssocL</i>	$(\text{Gam } ll)$	$= \text{concat } ll$
<i>gamPushNew</i>	$(\text{Gam } ll)$	$= \text{Gam } ([] : ll)$
<i>gamPushGam</i>	$g1 \ (\text{Gam } ll2)$	$= \text{Gam } (\text{gamToAssocL } g1 : ll2)$
<i>gamAddGam</i>	$g1 \ (\text{Gam } (ll2 : ll2))$	$= \text{Gam } ((\text{gamToAssocL } g1 \ ++ \ ll2) : ll2)$
<i>gamAdd</i>	$k \ v$	$= \text{gamAddGam } (k \mapsto v)$

Entering and leaving a scope is implemented by means of pushing and popping a Γ . Extending an environment Γ will take place on the top of the stack only. A *gamUnit* used as an infix operator will print as \mapsto .

A specialization *ValGam* of *Gam* is used to store and lookup the type of value identifiers.

```
data ValGamInfo = ValGamInfo{vgiTy :: Ty} deriving Show
type ValGam = Gam HsName ValGamInfo
```

The type is wrapped in a *ValGamInfo*. Later versions of EH can add additional fields to this data type.

```
valGamLookup :: HsName → ValGam → Maybe ValGamInfo
valGamLookup = gamLookup

valGamLookupTy :: HsName → ValGam → (Ty, ErrL)
valGamLookupTy n g
  = case valGamLookup n g of
    Nothing → (Ty_Any, [Err_NamesNotIntrod [n]])
    Just vgi → (vgiTy vgi, [])
```

Later the variant *valGamLookup* will do additional work, but for now it does not differ from *gamLookup*. The additional variant *valGamLookupTy* is specialized further to produce an error message in case the identifier is missing from the environment.

Checking Expr The rules in Fig. 6 do not provide much information about how the type σ in the consequence of a rule is to be computed; it is just stated that it should relate in some way to other types. However, type information can be made available to parts of the abstract syntax tree, either because the programmer has supplied it somewhere or because the compiler can reconstruct it. For types given by a programmer the compiler has to check if such a type correctly describes the value of an expression for which the

type is given. This is called *type checking*. If no type information has been given for a value, the compiler needs to reconstruct or infer this type based on the structure of the abstract syntax tree and the semantics of the language as defined by the typing rules. This is called *type inferencing*. In EH1 we exclusively deal with type checking.

$\Gamma; \sigma^k \stackrel{expr}{\vdash} e : \sigma$

$$\begin{array}{c}
\frac{\Gamma; \sigma^a \stackrel{expr}{\vdash} e_2 : _}{\Gamma; \square \rightarrow \sigma^k \stackrel{expr}{\vdash} e_1 : \sigma^a \rightarrow \sigma} \quad \text{(e-app1B)} \qquad \frac{i \mapsto \sigma^i, \Gamma; \sigma^r \stackrel{expr}{\vdash} e : \sigma^e}{\Gamma; \sigma^i \rightarrow \sigma^r \stackrel{expr}{\vdash} \lambda i \rightarrow e : \sigma^i \rightarrow \sigma^e} \quad \text{(e-lam1B)} \\
\\
\frac{\Gamma; \sigma_2^k \stackrel{expr}{\vdash} e_2 : \sigma_2 \quad \Gamma; \sigma_1^k \stackrel{expr}{\vdash} e_1 : \sigma_1}{\Gamma; (\sigma_1^k, \sigma_2^k) \stackrel{expr}{\vdash} (e_1, e_2) : (\sigma_1, \sigma_2)} \quad \text{(e-prod1B)} \qquad \frac{i \mapsto \sigma^i, \Gamma; \sigma^i \stackrel{expr}{\vdash} e^i : _ \quad i \mapsto \sigma^i, \Gamma; \sigma^k \stackrel{expr}{\vdash} e : \sigma^e}{\Gamma; \sigma^k \stackrel{expr}{\vdash} \mathbf{let} \ i :: \sigma^i; \ i = e^i \mathbf{in} \ e : \sigma^e} \quad \text{(e-let1B)} \\
\\
\frac{(i \mapsto \sigma^i) \in \Gamma \quad \frac{fit}{\vdash} \sigma^i \leq \sigma^k : \sigma}{\Gamma; \sigma^k \stackrel{expr}{\vdash} i : \sigma} \quad \text{(e-ident1B)} \qquad \frac{\frac{fit}{\vdash} Int \leq \sigma^k : \sigma}{\Gamma; \sigma^k \stackrel{expr}{\vdash} \mathbf{minint} \dots \mathbf{maxint} : \sigma} \quad \text{(e-int1B)}
\end{array}$$

Fig. 7. Type checking for expression (checking variant)

We now can tailor the type rules in Fig. 6 towards an implementation which performs type checking, in Fig. 7. We also start with the discussion of the corresponding AG implementation. The rules now take an additional context, the expected (or known) type σ^k (attribute *knTy*, simultaneously referred to by $\sigma^k // knTy$) as specified by the programmer, defined in terms of AG as follows:

ATTR *AllExpr* [*knTy* : *Ty* ||]

The basic idea underlying this implementation for type checking, as well as in later versions of EH also for type inferencing, is that

- A *known* (or *expected*) type $\sigma^k // knTy$ is passed top-down through the syntax tree of an expression, representing the maximal type (in terms of \leq , see Fig. 8 and discussion below) the type of an expression can be. At all places where this expression is used it also is assumed that the type of this expression equals σ^k .

- A result type $\sigma//ty$ is computed bottom-up for each expression, representing the minimal type (in terms of \leq) the expression can have.
- At each node in the abstract syntax tree it is checked whether $\sigma \leq \sigma^k$ holds. The result of $lhs \leq rhs$ is rhs which is subsequently used by the type checker, for example to simply return or use in constructing another, usually composite, type.
- In general, for $lhs \leq rhs$ the rhs is an expected type whereas lhs is the bottom-up computed result type.

$$\boxed{\frac{fit}{\vdash} \sigma^l \leq \sigma^r : \sigma}$$

$$\frac{\frac{fit}{\vdash} \sigma_2^a \leq \sigma_1^a : \sigma^a \quad \frac{fit}{\vdash} \sigma_1^r \leq \sigma_2^r : \sigma^r}{fit \vdash \sigma_1^a \rightarrow \sigma_1^r \leq \sigma_2^a \rightarrow \sigma_2^r : \sigma^a \rightarrow \sigma^r} \quad (\text{f-arrow1})$$

$$\frac{\frac{fit}{\vdash} \sigma_1^l \leq \sigma_2^l : \sigma^l \quad \frac{fit}{\vdash} \sigma_1^r \leq \sigma_2^r : \sigma^r}{fit \vdash (\sigma_1^l, \sigma_1^r) \leq (\sigma_2^l, \sigma_2^r) : (\sigma^l, \sigma^r)} \quad (\text{f-prod1}) \quad \frac{I_1 \equiv I_2}{fit \vdash I_1 \leq I_2 : I_2} \quad (\text{f-con1})$$

$$\frac{}{fit \vdash \square \leq \sigma : \sigma} \quad (\text{f-anyl1}) \quad \frac{}{fit \vdash \sigma \leq \square : \sigma} \quad (\text{f-anyr1})$$

Fig. 8. Rules for fit

An additional judgement type named *fit* (Fig. 8) is needed to check an actual type against an expected (known) type. The judgement specifies the matching $\sigma_1 \leq \sigma_2$ of two types σ_1 and σ_2 . The meaning of \leq is that the left hand side (lhs) type σ_1 of \leq can be used where the right hand side (rhs) type σ_2 is expected. Expressed differently, \leq checks whether a value of type σ_1 can flow (that is, be stored) into a memory location of type σ_2 . This is an asymmetric relation because “a value flowing into a location” does not imply that it can flow the other way, so \leq conceptually has a direction, even though in the rules in Fig. 8 \leq is a test on equality of the two type arguments.

The rules for \leq also specify a result type. Strictly this result is not required for the *fit* judgement to hold but in the implementation it is convenient to have the implementation *fitsIn* of \leq return the smallest type σ for which of $\sigma_1 \leq \sigma$ and $\sigma_2 \leq \sigma$ hold. This is

useful in particular in relation to the use of \square in rule `f-any1` and rule `f-anyr1`; we will come back to this later.

For example, \leq is used in rule `e-int1B` which checks that its actual *Int* type matches the known type σ^k . The implementation of the type rule `e-int1B` performs this check and returns the type σ in attribute *ty*:

```

ATTR AllExpr [| ty : Ty]
SEM Expr
  | CConst loc.fTy = tyChar
  | IConst loc.fTy = tyInt
  | IConst CConst
    loc.fo = @fTy  $\leq$  @lhs.knTy
    .ty = foTy @fo

```

AG: Set notation for variants. The rule for (e.g.) attribute *fo* is specified for *IConst* and *CConst* together. Instead of specifying only one variant a whitespace separated list of variant names may be specified after the vertical bar '|'. It is also allowed to specify this list relative to all declared variants by specifying for which variants the rule should *not* be declared. For example: `* - IConst CConst` if the rule was to be defined for all variants except *IConst* and *CConst*.

AG: Local attributes. The attribute *fTy* is declared locally. In this context ‘local’ means that the scope is limited to the variant of a node. Attribute *fTy* defined for variant *IConst* is available only for other attribute rules for variant *IConst* of *Expr*. Note that no explicit rule for synthesized attribute *ty* is required; a copy rule is inserted to use the value of the locally declared attribute *ty*. This is a common AG idiom when a value is required for later use as well or needs to be redefined in later versions of EH.

Some additional constants representing built-in types are also required:

```

tyInt   = Ty_Con hsnInt
tyChar = Ty_Con hsnChar

```

The local attribute *fTy* (by convention) holds the type as computed on the basis of the abstract syntax tree. This type *fTy* is subsequently compared to the expected type **lhs.knTy** via the implementation *fitsIn* of the rules for `fit` \leq . In infix notation *fitsIn* prints as \leq . The function *fitsIn* returns a *FIOut* (**fitsIn output**) data structure in attribute *fo*. *FIOut* consists of a record containing amongst other things field *foTy*:

```

data FIOut = FIOut{foTy :: Ty      ,foErrL :: ErrL}
emptyFO   = FIOut{foTy = Ty_Any,foErrL = [] }

foHasErrs :: FIOut  $\rightarrow$  Bool

```

$$foHasErrs = \neg.null.foErrL$$

Using a separate attribute fTy instead of using its value directly has been done in order to prepare for a redefinition of fTy in later versions⁴.

$Ty_Any//Any//\square$ plays a special role. This type appears at two places in the implementation of the type system as a solution to the following problems:

- Invariant to our implementation is the top-down passing of an expected type. However, this type is not always fully known in a top-down order. For example, in rule **e-app1B** (Fig. 7) the argument of the expected function type $\square \rightarrow \sigma^k$ is not known because this information is only available from the environment Γ which is used further down in the AST via rule **e-ident1B**. In this use of \square it represents a “don’t know” of the type system implementation. As such \square has the role of a type variable (as introduced for type inferencing in section 3).
- An error occurs at a place where the implementation of the type system needs a type to continue (type checking) with. In that case \square is used to prevent further errors from occurring. In this use of \square it represents a “don’t care” of the type system implementation. As such \square will be replaced by more a more specific type as soon as it matches (via \leq) such a type.

In both cases \square is a type exclusively used by the implementation to smoothen type checking. The rules for \leq for \square in Fig. 8 state that \square is equal to any type. The effect is that the result of \leq is a more specific type. This suits our “don’t know” and “don’t care” use. Later, when discussing the AG implementation for these rules this issue reappears. In later EH versions we will split the use of \square into the proper use of a type lattice, and will it thus disappear.

The role of \square may appear to be similar to \top and \perp known from type theory. However, \square is used only as a mechanism for the type system implementation. It is not offered as a feature to the user (i.e. the EH programmer) of the type system.

$Ty_Any//Any//\square$ is also used at the top level where the actual expected type of the expression neither is specified nor matters because it is not used:

$$\begin{array}{l} \mathbf{SEM\ AGItf} \\ | \mathbf{AGItf\ expr.knTy} = Ty_Any \end{array}$$

The rule **f-arrow1** in Fig. 8 for comparing function types compares the types for arguments in the opposite direction. Only in later versions of EH when \leq really behaves asymmetrically we will discuss this aspect of the rules which is named *contravariance*. In the rules in Fig. 8 the direction makes no difference; the correct use of the direction for now only anticipates issues yet to come.

The Haskell counterpart of $\overset{fit}{\vdash} \sigma_1 \leq \sigma_2 : \sigma$ is implemented by *fitsIn*:

$$fitsIn :: Ty \rightarrow Ty \rightarrow FIOut$$

⁴ This will happen with other attributes as well.

```

fitsIn ty1 ty2
  = f ty1 ty2
where
  res t          = emptyFO{foTy = t}
  f Ty_Any t2    = res t2
  f t1 Ty_Any   = res t1
  f t1 @(Ty_Con s1)
    t2 @(Ty_Con s2)
    | s1 ≡ s2    = res t2

  f t1 @(Ty_App (Ty_App (Ty_Con c1) ta1) tr1)
    t2 @(Ty_App (Ty_App (Ty_Con c2) ta2) tr2)
    | hsnIsArrow c1 ∧ c1 ≡ c2
    = comp ta2 tr1 ta1 tr2 (λa r → [a] ‘mkTyArrow’ r)
  f t1 @(Ty_App tf1 ta1)
    t2 @(Ty_App tf2 ta2)
    = comp tf1 ta1 tf2 ta2 Ty_App
  f t1 t2 = err [Err_UnifyClash ty1 ty2 t1 t2]
  err e   = emptyFO{foErrL = e}
  comp tf1 ta1 tf2 ta2 mkComp
    = foldr1 (λfo1 fo2 → if foHasErrs fo1 then fo1 else fo2)
              [ffo, afo, res rt]
where ffo = f tf1 tf2
        afo = f ta1 ta2
        rt  = mkComp (foTy ffo) (foTy afo)

```

The function *fitsIn* checks whether the *Ty_App* structure and all type constants *Ty_Con* are equal. If not, a non-empty list of errors is returned as well as type *Ty_Any//Any//□*. Matching a composite type is split in two cases for *Ty_App*, one for function types (the first case), and one for the remaining type applications (the second case). For the current EH version the second case only concerns tuple types. Both matches for composite types use *comp* which performs multiple \leq 's and combines the results. The difference lies in the treatment of contravariant behavior as discussed earlier.

The type rules leave in the open how to handle a situation when a required constraint is broken. For a compiler this is not good enough, being the reason *fitsIn* gives a “will-do” type *Ty_Any* back together with an error for later processing. Errors themselves are also described via AG:

```

DATA Err
| UnifyClash ty1      : {Ty} ty2      : {Ty}
  ty1detail : {Ty} ty2detail : {Ty}

```

```

DATA Err
| NamesNotIntrod nmL : {[HsName]}

```

The *Err* datatype is available as a datatype in the same way *Ty* is. The error datatype is also used for signalling undeclared identifiers:

SEM Expr
 | *Var loc*.(*gTy*, *nmErrs*)
 = *valGamLookupTy @nm @lhs.valGam*
 fTy = @*gTy*
 fo = @*fTy* ≤ @*lhs.knTy*
 ty = *foTy @fo*

AG: Left hand side patterns. The simplest way to define a value for an attribute is to define one value for one attribute at a time. However, if this value is a tuple, its fields are to be extracted and assigned to individual attributes (as in *tyArrowArgRes*). AG allows a pattern notation of the form(s) to make the notation for this situation more concise:

| *<variant>* *<node>*.(*<attr₁>* , *<attr₂>* , ...) =
 | *<variant>* (*<node₁>*.*<attr₁>*, *<node₁>*.*<attr₂>*, ...) =

Again, the error condition is signalled by a non empty list of errors if a lookup in Γ fails. These errors are gathered so they can be incorporated into an annotated pretty printed version of the program.

Typing rule **e-ident1B** uses the environment Γ to retrieve the type of an identifier. This environment *valGam* for types of identifiers simply is declared as an inherited attribute, initialized at the top of the abstract syntax tree. It is only extended with new bindings for identifiers at a declaration of an identifier.

ATTR AllDecl AllExpr [*valGam* : *ValGam* ||]

SEM AGItf

| *AGItf expr.valGam* = *emptyGam*

One may wonder why the judgement $\vdash^{\text{fit}} \sigma_1 \leq \sigma_2 : \sigma$ and its implementation *fitsIn* returns a type at all; the idea of checking was to only pass explicit type information σ^k (or *knTy*) from the top of the abstract syntax tree to the leaf nodes. Note that this idea breaks when we try to check the expression *id 3* in

let *id* :: *Int* → *Int*
 id = $\lambda x \rightarrow x$
in *id 3*

What is the *knTy* against which 3 will be checked? It is the argument type of the type of *id*. However, in rule **e-app1B** and its AG implementation, the type of *id* is not the (top-to-bottom travelling) $\sigma^k // \text{knTy}$, but it will be the argument part of the (bottom-to-top travelling) resulting function type of *e₁ // func.ty*:

SEM Expr

```
| App loc .knFunTy    = [Ty.Any] 'mkTyArrow' @lhs.knTy
  func.knTy          = @knFunTy
  (arg.knTy, loc.fTy) = tyArrowArgRes @func.ty
  loc .ty            = @fTy
```

The idea here is to encode the partially known function type as $\square \rightarrow \sigma^k$ (passed to *func.knTy*) and let *fitsIn* fill in the missing details, that is to find a type for \square . This is the place where it is convenient to have *fitsIn* return a type in which $\square//Ty_Any$'s are replaced by a more concrete type. From that result the known/expected type of the argument can be extracted.

Note that we are already performing a little bit of type inferencing. This is however only done locally to *App* as the \square in $\square \rightarrow \sigma^k$ is guaranteed to have disappeared in the result type of *fitsIn*. If this is not the case, the EH program contains an error. This is a mechanism we repeatedly use, so we summarize it here:

- Generally, the semantics of the language requires a type σ to be of a specific form. Here σ equals the type of the function (not known at the *App* location in the AST) which should have the form $\square \rightarrow \sigma^k$.
- The specific form may contain types about which we know nothing, here encoded by \square , in later EH versions by type variables.
- *fitsIn*// \leq is used to enforce σ to have the right form. Here this is done by pushing the form as σ^k down the AST for the function (attribute *func.knTy*). The check $\sigma \leq \sigma^k$ is then performed in the *Var* variant of *Expr*.
- Enforcing may or may not succeed. In the latter case error messages are generated and the result of enforcing is \square .

The type construction and inspection done in the *App* variant of *Expr* requires some additional type construction functions, of which we only include *mkTyArrow*:

```
algTy      :: MkConApp Ty
algTy      = (Ty.Con, Ty.App, id, id)
mkTyArrow :: TyL → Ty → Ty
mkTyArrow = flip (foldr (mkArrow algTy))
```

The function is derived from a more general function *mkArrow*:

```
type MkConApp t = (HsName → t, t → t → t, t → t, t → t)
```

```
mkArrow :: MkConApp t → t → t → t
mkArrow alg @(con, -, -, -) a r = mkApp alg [con hsnArrow, a, r]
```

```
mkApp :: MkConApp t → [t] → t
mkApp (_, app, top, -) ts
  = case ts of
    [t] → t
```

– $\rightarrow \text{top} (\text{foldl1 } \text{app } \text{ts})$

A *MkConApp* contains four functions, for constructing a value similar to *Con*, *App*, *AppTop* and *ICnst* respectively. These functions are used by *mkApp* to build an *App* like structure and by *mkArrow* to build function like structures. The code for (e.g.) parsers (omitted from these notes), uses these functions parameterized with the proper four semantics functions as generated by the AG system. So this additional layer of abstraction improves code reuse. Similarly, function *mkTyProdApp* constructs a tuple type out of types for the elements.

The functions used for scrutinizing a type are given names in which (by convention) the following is encoded:

- What is scrutinized.
- What is the result of scrutinizing.

For example, *tyArrowArgRes* dissects a function type into its argument and result type. If the scrutinized type is not a function, “will do” values are returned:

$\text{tyArrowArgRes} :: \text{Ty} \rightarrow (\text{Ty}, \text{Ty})$

$\text{tyArrowArgRes } t$
 = **case** t **of**
 $\text{Ty_App } (\text{Ty_App } (\text{Ty_Con } nm) a) r$
 | $\text{hsnIsArrow } nm \rightarrow (a, r)$
 – $\rightarrow (\text{Ty_Any}, t)$

Similarly *tyProdArgs* is defined to return the types of the elements of a tuple type. The code for this and other similar functions have been omitted for brevity.

Constructor Con, tuples. Apart from constructing function types only tupling allows us to build composite types. The rule **e-prod1B** for tupling has no immediate counterpart in the implementation because a tuple (a, b) is encoded as the application $(,) a b$. The alternative *Con* takes care of producing a type $a \rightarrow b \rightarrow (a, b)$ for $(,)$.

SEM Expr
 | *Con* **loc**. $ty = \text{let } \text{resTy} = \text{tyArrowRes } @\text{lhs.knTy}$
 in $\text{tyProdArgs } \text{resTy } 'mkTyArrow' \text{resTy}$

This type can be constructed from *knTy* which by definition has the form $\square \rightarrow \square \rightarrow (a, b)$ (for this example). The result type of this function type is taken apart and used to produce the desired type. Also by definition (via construction by the parser) we know the arity is correct.

Note that, despite the fact that the cartesian product constructors are essentially polymorphic, we do not have to do any kind of unification here, since they either appear in the right hand side of declaration where the type is given by an explicit type declaration, or they occur at an argument position where the type has been implicitly specified by

the function type. Therefore we indeed can use the a and b from type $\square \rightarrow \square \rightarrow (a, b)$ to construct the type $a \rightarrow b \rightarrow (a, b)$ for the constructor $(,)$.

λ -expression Lam. For rule **e-lam1B** the check whether $knTy$ has the form $\sigma_1 \rightarrow \sigma_2$ is done by letting $fitsIn$ match the $knTy$ with $\square \rightarrow \square$. The result (forced to be a function type) is split up by $tyArrowArgRes$ into argument and result type. The function $gamPushNew$ opens a new scope on top of $valGam$ so as to be able to check duplicate names introduced by the pattern arg :

SEM Expr

Lam loc $funTy$	=	$[Ty_Any] \text{ 'mkTyArrow' } Ty_Any$
$foKnFun$	=	$@funTy \leq @lhs.knTy$
$(arg.knTy, body.knTy)$	=	$tyArrowArgRes (foTy @foKnFun)$
$arg.valGam$	=	$gamPushNew @lhs.valGam$
loc ty	=	$@lhs.knTy$

Type annotations (for λ -expression). In order to make λ -expressions typecheck correctly it is the responsibility of the EH programmer to supply the correct type signature. The *TypeAs* variant of *Expr* takes care of this by simply passing the type signature as the expected type:

SEM Expr

TypeAs loc fo	=	$@tyExpr.ty \leq @lhs.knTy$
$expr.knTy$	=	$@tyExpr.ty$

The obligation for the EH programmer to specify a type is dropped in later versions of EH.

Checking PatExpr Before we can look into more detail at the way new identifiers are introduced in **let**- and λ -expressions we take a look at patterns. The rule **e-let1B** is too restrictive for the actual language construct supported by EH because the rule only allows a single identifier to be introduced. The following program allows inspection of parts of a composite value by naming its components through pattern matching:

```
let  $p :: (Int, Int)$ 
     $p @ (a, b) = (3, 4)$ 
in  $a$ 
```

The rule **e-let1C** from Fig. 9 together with the rules for patterns from Fig. 10 reflects the desired behaviour. These rules differ from those in Fig. 7 in that a pattern instead of a single identifier is allowed in a value definition and the parameter position of a λ -expression.

Again the idea is to distribute a known type over the pattern by dissecting it into its constituents. However, patterns do not return a type but type bindings for the identifiers in-

$$\boxed{\Gamma; \sigma^k \stackrel{expr}{\vdash} e : \sigma}$$

$$\frac{\begin{array}{l} \Gamma^p, \Gamma; \sigma^i \stackrel{expr}{\vdash} e^i : _ \\ \Gamma^p, \Gamma; \sigma^k \stackrel{expr}{\vdash} e : \sigma^e \\ \sigma^i \stackrel{pat}{\vdash} p : \Gamma^p \\ p \equiv i \vee p \equiv i @ \dots \end{array}}{\Gamma; \sigma^k \stackrel{expr}{\vdash} \mathbf{let} \ i :: \sigma^i; p = e^i \mathbf{in} \ e : \sigma^e} \quad (\text{e-let1C})$$

$$\frac{\begin{array}{l} \Gamma^p, \Gamma; \sigma^r \stackrel{expr}{\vdash} e : \sigma^e \\ \sigma^p \stackrel{pat}{\vdash} p : \Gamma^p \end{array}}{\Gamma; \sigma^p \rightarrow \sigma^r \stackrel{expr}{\vdash} \lambda p \rightarrow e : \sigma^p \rightarrow \sigma^e} \quad (\text{e-lam1C})$$

Fig. 9. Type checking for let-expression with pattern

$$\boxed{\sigma^k \stackrel{pat}{\vdash} p : \Gamma^p}$$

$$\frac{}{\sigma^k \stackrel{pat}{\vdash} i : [i \mapsto \sigma^k]} \quad (\text{p-var1}) \quad \frac{\begin{array}{l} \text{dom}(\Gamma_1^p) \cap \text{dom}(\Gamma_2^p) = \emptyset \\ \sigma_2^k \stackrel{pat}{\vdash} p_2 : \Gamma_2^p \\ \sigma_1^k \stackrel{pat}{\vdash} p_1 : \Gamma_1^p \end{array}}{(\sigma_1^k, \sigma_2^k) \stackrel{pat}{\vdash} (p_1, p_2) : \Gamma_1^p, \Gamma_2^p} \quad (\text{p-prod1})$$

Fig. 10. Building environments from patterns

side a pattern instead. The new bindings are subsequently used in **let**- and λ -expressions bodies.

A tuple pattern with rule `p-prod1` is encoded in the same way as tuple expressions; that is, pattern (a, b) is encoded as an application $(,) a b$ with an `AppTop` on top of it. We dissect the known type of a tuple in rule `p-prod1` into its element types at `AppTop` using function `tyProdArgs`. For this version of EH we only have tuple patterns; we can indeed assume that we are dealing with a tuple type.

ATTR `AllPatExpr` [`knTy` : `Ty` ||]

ATTR `PatExpr` [`knTyL` : `TyL` ||]

SEM `PatExpr`

```

| AppTop loc .knProdTy
    = @lhs.knTy
    .(knTyL, aErrs)
    = case tyProdArgs @knProdTy of
        tL | @patExpr.arity  $\equiv$  length tL
             $\rightarrow$  (reverse tL, [])
        _  $\rightarrow$  (repeat Ty.Any
            , [Err_PatAriy
                @knProdTy @patExpr.arity])
| App loc .(knArgTy, knTyL)
    = hdAndTl @lhs.knTyL
    arg.knTy = @knArgTy

```

The list of these elements is passed through attribute `knTyL` to all `App`'s of the pattern. At each `App` one element of this list is taken as the `knTy` of the element AST.

The complexity in the `AppTop` alternative of `PatExpr` arises from repair actions in case the arity of the pattern and its known type do not match. In that case the subpatterns are given as many \square 's as known type as necessary.

Finally, for the distribution of the known type throughout a pattern we need to properly initialize `knTyL`:

SEM `Decl`

```

| Val patExpr.knTyL = []

```

SEM `Expr`

```

| Lam arg .knTyL = []

```

The arity of the patterns is needed as well:

ATTR `PatExpr` [|| `arity` : `Int`]

SEM `PatExpr`

```

| App lhs.arity = @func.arity + 1
| Con Var AppTop IConst CConst
    lhs.arity = 0

```

As a result of this unpacking, at a *Var* alternative attribute *knTy* holds the type of the variable name introduced. The type is added to attribute *valGam* that is threaded through the pattern for gathering all introduced bindings:

```

ATTR AllPatExpr [| valGam : ValGam |]
SEM PatExpr
  | Var VarAs loc.ty      = @lhs.knTy
                        .varTy      = @ty
                        .addToGam = if @lhs.inclVarBind  $\wedge$  @nm  $\neq$  hsnWild
                                then gamAdd @nm
                                (ValGamInfo @varTy)
                                else id
  | Var      lhs.valGam = @addToGam @lhs.valGam
  | VarAs    lhs.valGam = @addToGam @patExpr.valGam

```

The addition to *valGam* is encoded in the attribute *addToGam*, a function which only adds a new entry if the variable name is not equal to an underscore '_' and has not been added previously via a type signature for the variable name, signalled by attribute *inclVarBind* (defined later).

Checking declarations In a **let**-expression type signatures, patterns and expressions do meet. Rule e-let1C from Fig. 9 shows that the idea is straightforward: take the type signature, distribute it over a pattern to extract bindings for identifiers and pass both type signature (as *knTy*) and bindings (as *valGam*) to the expression. This works fine for single combinations of type signature and the corresponding value definition for a pattern. However, it does not work for:

- Mutually recursive value definitions.

```

let f :: ...
     f =  $\lambda x \rightarrow \dots g \dots$ 
     g :: ...
     g =  $\lambda x \rightarrow \dots f \dots$ 
in ...

```

In the body of *f* the type *g* must be known and vice-versa. There is no ordering of what can be defined and checked first. In Haskell *f* and *g* together would be in the same binding group.

- Textually separated signatures and value definitions.

```

let f :: ...
     ...
     f =  $\lambda x \rightarrow \dots$ 
in ...

```


This type signature is then used as the known type of the pattern and the expression.

SEM Decl

| Val **loc**.knTy = @sigTy

The flag *hasTySig* is used to signal the presence of a type signature for a value and a correct form of the pattern. We allow patterns of the form ‘*ab @ (a, b)*’ to have a type signature associated with *ab*. No type signatures are allowed for ‘*(a, b)*’ without the ‘*ab @*’ alias (because there is no way to refer to the anonymous tuple) nor is it allowed to specify type signature for the fields of the tuple (because of simplicity, additional plumbing would be required).

ATTR PatExpr [| mbTopNm : {Maybe HsName}]

SEM PatExpr

| Var VarAs **loc**.mbTopNm = if @nm ≡ hsnWild then Nothing else Just @nm

| * – Var VarAs

loc.mbTopNm = Nothing

The value of *hasTySig* is also used to decide on the binding of the top level identifier of a pattern, via *inclVarBind*.

ATTR PatExpr [inclVarBind : Bool]

SEM PatExpr

| AppTop patExpr.inclVarBind = True

SEM Decl

| Val patExpr.inclVarBind = ¬ @hasTySig

SEM Expr

| Lam arg .inclVarBind = True

If a type signature for an identifier is already defined there is no need to rebind the identifier by adding one more binding to *valGam*.

New bindings are not immediately added to *valGam* but are first gathered in a separately threaded attribute *patValGam*, much in the same way as *gathTySigGam* is used.

ATTR AllDecl [| patValGam : ValGam]

SEM Decl

| Val patExpr.valGam = @lhs.patValGam

lhs .patValGam = @patExpr.valGam

expr .valGam = @lhs.valGam

SEM Expr

| Let decls.patValGam = @decls.gathTySigGam

‘gamPushGam’ @lhs.valGam

loc .(lValGam, gValGam) = gamPop @decls.patValGam

decls.valGam = @decls.patValGam

body.valGam = @decls.patValGam

Newly gathered bindings are stacked on top of the inherited *valGam* before passing them on to both declarations and body.

Some additional functionality for pushing and popping the stack *valGam* is also needed:

$$\begin{aligned} \text{gamPop} &:: \text{Gam } k \ v \rightarrow (\text{Gam } k \ v, \text{Gam } k \ v) \\ \text{assocLToGam} &:: \text{AssocL } k \ v \rightarrow \text{Gam } k \ v \\ \\ \text{gamPop} &(\text{Gam } (l : ll)) = (\text{Gam } [l], \text{Gam } ll) \\ \text{assocLToGam } l &= \text{Gam } [l] \end{aligned}$$

Extracting the top of the stack *patValGam* gives all the locally introduced bindings in *lValGam*. An additional error message is produced if any duplicate bindings are present in *lValGam*.

Checking TyExpr All that is left to do now is to use the type expressions to extract type signatures. This is straightforward as type expressions (abstract syntax for what the programmer specified) and types (as internally used by the compiler) have almost the same structure:

$$\begin{aligned} \text{ATTR } \text{TyExpr} &[| \text{ty} : \text{Ty} |] \\ \text{SEM } \text{TyExpr} & \\ &| \text{Con } \mathbf{lhs.ty} = \text{Ty_Con } @nm \\ &| \text{App } \mathbf{lhs.ty} = \text{Ty_App } @func.ty @arg.ty \end{aligned}$$

Actually, we need to do more because we also have to check whether a type is defined. A variant of *Gam* is used to hold type constants:

$$\begin{aligned} \mathbf{data} \ \text{TyGamInfo} &= \text{TyGamInfo}\{\text{tgiTy} :: \text{Ty}\} \ \mathbf{deriving} \ \text{Show} \\ \\ \mathbf{type} \ \text{TyGam} &= \text{Gam } \text{HsName } \text{TyGamInfo} \\ \\ \text{tyGamLookup} &:: \text{HsName} \rightarrow \text{TyGam} \rightarrow \text{Maybe } \text{TyGamInfo} \\ \text{tyGamLookup } nm \ g & \\ &= \mathbf{case} \ \text{gamLookup } nm \ g \ \mathbf{of} \\ &\quad \text{Nothing} \ | \ \text{hsnIsProd } nm \rightarrow \text{Just } (\text{TyGamInfo } (\text{Ty_Con } nm)) \\ &\quad \text{Just } \text{tgi} && \rightarrow \text{Just } \text{tgi} \\ &\quad \text{--} && \rightarrow \text{Nothing} \end{aligned}$$

This Γ is threaded through *TyExpr*:

$$\text{ATTR } \text{AllTyExpr} [| \text{tyGam} : \text{TyGam} |]$$

At the root of the AST *tyGam* is initialized with the fixed set of types available in this version of the compiler:

$$\begin{aligned} \text{SEM } \text{AGItf} & \\ &| \text{AGItf } \mathbf{loc.tyGam} = \text{assocLToGam} \\ &\quad [(\text{hsnArrow}, \text{TyGamInfo } (\text{Ty_Con } \text{hsnArrow})) \end{aligned}$$

```

, (hsnInt, TyGamInfo tyInt)
, (hsnChar, TyGamInfo tyChar)
]

```

Finally, at the *Con* alternative of *TyExpr* we need to check if a type is defined:

```

SEM TyExpr
| Con loc.(tgi, nmErrs) = case tyGamLookup @nm @lhs.tyGam of
    Nothing → (TyGamInfo Ty_Any
    , [Err_NamesNotIntrod @nm])
    Just tgi → (tgi, [])

```

3 EH 2: monomorphic type inferencing

The next version of EH drops the requirement that all value definitions need to be accompanied by an explicit type signature. For example, the example from the introduction:

```

let i = 5
in i

```

is accepted by this version of EH:

```

let i = 5
    {- [ i:Int ] -}
in i

```

The idea is that the type system implementation has an internal representation for “knowing it is a type, but not yet which one” which can be replaced by a more specific type if that becomes known. The internal representation for a yet unknown type is called a *type variable*, similar to mutable variables for (runtime) values.

The implementation attempts to gather as much information as possible from a program to reconstruct (or infer) types for type variables. However, the types it can reconstruct are limited to those allowed by the used type language, that is, basic types, tuples and functions. All types are assumed to be monomorphic, that is, polymorphism is not yet allowed. The next version of EH deals with polymorphism.

So

```

let id =  $\lambda x \rightarrow x$ 
in let v = id 3
    in id

```

will give

```

let id = \x -> x
    {- [ id:Int -> Int ] -}
in let v = id 3
    {- [ v:Int ] -}
    in id

```

If the use of *id* to define *v* is omitted, less information (namely the argument of *id* is an int) to infer a type for *id* is available. Because no more specific type information for the argument (and result) of *id* could be retrieved the representation for “not knowing which type”, that is, a type variable, is shown:

```

let id = \x -> x
    {- [ id:v_1_1 -> v_1_1 ] -}
in id

```

On the other hand, if contradictory information is found we will have

```

let id = \x -> x
    {- [ id:Int -> Int ] -}
in let v = (id 3, id 'x')
    {- ***ERROR(S):
        In '(id 3, id 'x')':
        ... In 'x':
        Type clash:
        failed to fit: Char <= Int
        problem with : Char <= Int -}
    {- [ v:(Int,Int) ] -}
    in v

```

However, the next version of EH dealing with Haskell style polymorphism (section 4) accepts this program.

Partial type signatures are also allowed. A partial type signature specifies a type only for a part, allowing a cooperation between the programmer who specifies what is (e.g.) already known about a type signature and the type inferencer filling in the unspecified details. For example:

```

let id :: ... -> ...
    id = \x -> x
in let f :: (Int -> Int) -> ...
    f = \i -> \v -> i v
    v = f id 3

```

```

in let v = f id 3
in v

```

The type inferencer pretty prints the inferred type instead of the explicit type signature:

```

let id :: Int -> Int
    id = \x -> x
    {- [ id:Int -> Int ] -}
in let f :: (Int -> Int) -> Int -> Int
    f = \i -> \v -> i v
    v = f id 3
    {- [ v:Int, f:(Int -> Int) -> Int -> Int ] -}
in let v = f id 3
    {- [ v:Int ] -}
in v

```

The discussion of the implementation of this feature is postponed until section 3.6 in order to demonstrate the effects of an additional feature on the compiler implementation in isolation.

3.1 Type variables

In order to be able to represent yet unknown types the type language needs *type variables* to represent this:

$$\begin{aligned}
 \sigma &= \text{Int} \mid \text{Char} \\
 &\mid (\sigma, \dots, \sigma) \\
 &\mid \sigma \rightarrow \sigma \\
 &\mid v
 \end{aligned}$$

The corresponding type structure *Ty* needs to be extended with an alternative for a variable:

```

DATA Ty
  | Var tv : {TyVarId}

```

A type variable is identified by a unique identifier, a *UID*:

```

newtype UID = UID [Int] deriving (Eq, Ord)

```

```

type UIDL = [UID]

```

```

instance Show UID where

```

```

  show (UID ls) = concat.intersperse "_".map show.reverse $ ls

```

```
type TyVarId = UID
```

```
type TyVarIdL = [TyVarId]
```

The idea is to thread a counter as global variable through the AST, incrementing it whenever a new unique value is required. The implementation used throughout all EH compiler versions is more complex because an *UID* actually is a hierarchy of counters, each level counting in the context of an outer level. This is not discussed any further; we will ignore this aspect and just assume a unique *UID* can be obtained. However, a bit of its implementation is visible in the pretty printed representation as a underscore separated list of integer values, occasionally visible in sample output of the compiler.

3.2 Constraints

Although the typing rules at Fig. 9 still hold we need to look at the meaning of \leq (or *fitsIn*) in the presence of type variables. The idea here is that what is unknown may be replaced by that which is known. For example, when the check $v \leq \sigma$ is encountered, the easiest way to make $v \leq \sigma$ true is to state that the (previously) unknown type v equals σ . An alternative way to look at this is that $v \leq \sigma$ is true under the constraint that v equals σ .

Remembering and applying constraints Next we can observe that once a certain type v is declared to be equal to a type σ this fact has to be remembered.

```
C = [v ↦ σ]
```

A set of *constraints* C (appearing in its non pretty printed form as `Cnstr` in the source text) is a set of bindings for type variables, represented as an association list:

```
newtype C = C (AssocL TyVarId Ty) deriving Show
```

```
cnstrTyLookup :: TyVarId → C → Maybe Ty
```

```
cnstrTyLookup tv (C s) = lookup tv s
```

```
emptyCnstr :: C
```

```
emptyCnstr = C []
```

```
cnstrTyUnit :: TyVarId → Ty → C
```

```
cnstrTyUnit tv t = C [(tv, t)]
```

If *cnstrTyUnit* is used as an infix operator it is printed as \mapsto in the same way as used in type rules.

Different strategies can be used to cope with constraints [17,29]. Here constraints C are used to replace all other references to v by σ , for this reason often named a *substitution*. In this version of EH the replacement of type variables with newly types is done immediately after constraints are obtained as to avoid finding a new and probably conflicting

constraint for a type variable. Applying constraints means substituting type variables with the bindings in the constraints, hence the class *Substitutable* for those structures which have references to type variables hidden inside and can replace, or substitute those type variables:

```
infixr 6 >
class Substitutable s where
  (>) :: C → s → s
  ftv  :: s → TyVarIdL
```

The operator \succ applies constraints C to a *Substitutable*. Function *ftv* extracts the free type variable references as a set of *TVarId*'s.

A C can be applied to a type:

```
instance Substitutable Ty where
  (>) = tyAppCnstr
  ftv = tyFtv
```

This is another place where we use the AG notation and the automatic propagation of values as attributes throughout the type representation to make the description of the application of a C to a *Ty* easier. The function *tyAppCnstr* is defined in terms of the following AG. The plumbing required to provide the value of attribute *repl* (*tv*s) available as the result of Haskell function *tyAppCnstr* (*tyFtv*) has been omitted:

```
ATTR TyAGIf AllTy [cnstr : C ||
ATTR AllAllTy [ || repl : SELF ]
ATTR TyAGIf [ || repl : Ty ]
SEM Ty
  | Var lhs.repl = maybe @repl id (cnstrTyLookup @tv @lhs.cnstr)

ATTR TyAGIf AllTy [|| tvs USE{∪}{[]} : TyVarIdL]
SEM Ty
  | Var lhs.tvs = [@tv]
```

AG: Attribute of type SELF. The type of an attribute of type **SELF** depends on the node in which a rule is defined for the attribute. The generated type of an attribute $\langle attr \rangle$ for $\langle node \rangle$ is equal to the generated Haskell datatype of the same name $\langle node \rangle$. The AG compiler inserts code for building $\langle node \rangle$'s from the $\langle attr \rangle$ of the children and other fields. Insertion of this code can be overridden by providing a definition ourselves. In this way a complete copy of the AST can be built as a Haskell value. For example, via attribute *repl* a copy of the type is built which only differs (or, may differ) in the original in the value for the type variable.

AG: Attribute together with USE. A synthesized attribute $\langle attr \rangle$ may be declared together with **USE**{ $\langle op \rangle$ }{ $\langle zero \rangle$ }. The $\langle op \rangle$ and $\langle zero \rangle$ allow the insertion of copy

rules which behave similar to Haskell's *foldr*. The first piece of text $\langle op \rangle$ is used to combine the attribute values of two children by textually placing this text as an operator between references to the attributes of the children. If no child has an $\langle attr \rangle$, the second piece of text $\langle zero \rangle$ is used as a default value for $\langle attr \rangle$. For example, `tvS USE { 'union' } { [] }` (appearing in pretty printed form as `tvS USE{U}{[]}`) gathers bottom-up the free type variables of a type.

The application of a *C* is straightforwardly lifted to lists:

instance *Substitutable a* \Rightarrow *Substitutable [a]* **where**

$s \succ l = \text{map } (s \succ) l$
 $\text{ftv } l = \text{unionL.map ftv } \$ l$

$\text{unionL} :: \text{Eq } a \Rightarrow [[a]] \rightarrow [a]$
 $\text{unionL} = \text{foldr union } []$

A *C* can also be applied to another *C*:

instance *Substitutable C* **where**

$s1 @ (C \text{ sl}_1) \succ s2 @ (C \text{ sl}_2)$
 $= C (\text{sl}_1 \# \text{map } (\lambda (v, t) \rightarrow (v, s1 \succ t)) \text{sl}'_2)$
where $\text{sl}'_2 = \text{deleteFirstBy } (\lambda (v1, -) (v2, -) \rightarrow v1 \equiv v2) \text{sl}_2 \text{sl}_1$
 $\text{ftv } (C \text{ sl})$
 $= \text{ftv.map snd } \$ \text{sl}$

Substituting a substitution is non-commutative as constraints s_1 in $s_1 \succ s_2$ take precedence over s_2 . To make this even clearer all constraints for type variables in s_1 are removed from s_2 , even though for a list implementation this would not be required.

Computing constraints The only source of constraints is the check *fitsIn* which determines whether one type can flow into another one. The previous version of EH could only do one thing in case a type could not fit in another: report an error. Now, if one of the types is unknown, which means that it is a type variable, we have the additional possibility of returning a constraint on that type variable. The implementation *fitsIn* of \leq additionally has to return constraints:

data *FIOut* = *FIOut*{*foTy* :: *Ty* , *foErrL* :: *ErrL*, *foCnstr* :: *C* }

emptyFO = *FIOut*{*foTy* = *Ty_Any*, *foErrL* = [] , *foCnstr* = *emptyCnstr*}

Computation and proper combination of constraints necessitates *fitsIn* to be rewritten:

fitsIn :: *Ty* \rightarrow *Ty* \rightarrow *FIOut*

fitsIn *ty*₁ *ty*₂
 $= f \text{ ty}_1 \text{ ty}_2$

where

res t $= \text{emptyFO}\{\text{foTy} = t\}$

$$\begin{aligned}
\text{bind } tv \ t &= (\text{res } t)\{\text{foCnstr} = tv \mapsto t\} \\
\text{occurBind } v \ t \mid v \in \text{fv } t &= \text{err } [\text{Err_UnifyOccurs } ty_1 \ ty_2 \ v \ t] \\
&\mid \text{otherwise} &= \text{bind } v \ t
\end{aligned}$$

$$\begin{aligned}
\text{comp } tf_1 \ ta_1 \ tf_2 \ ta_2 \ \text{mkComp} \\
&= \text{foldr1 } (\lambda fo_1 \ fo_2 \rightarrow \text{if } \text{foHasErrs } fo_1 \ \text{then } fo_1 \ \text{else } fo_2) \\
&\quad [\text{ffo}, \text{afo}, \text{rfo}] \\
\text{where } \text{ffo} &= f \ tf_1 \ tf_2 \\
\text{fs} &= \text{foCnstr } \text{ffo} \\
\text{afo} &= f \ (\text{fs } \succ \ ta_1) \ (\text{fs } \succ \ ta_2) \\
\text{as} &= \text{foCnstr } \text{afo} \\
\text{rt} &= \text{mkComp } (\text{as } \succ \ \text{foTy } \text{ffo}) \ (\text{foTy } \text{afo}) \\
\text{rfo} &= \text{emptyFO}\{\text{foTy} = \text{rt}, \text{foCnstr} = \text{as } \succ \ \text{fs}\}
\end{aligned}$$

$$\begin{aligned}
f \ \text{Ty_Any} \quad t_2 &= \text{res } t_2 \\
f \ t_1 \quad \text{Ty_Any} &= \text{res } t_1 \\
f \ t_1 \ @(\text{Ty_Con } s1) \\
t_2 \ @(\text{Ty_Con } s2) \\
\mid s1 \equiv s2 &= \text{res } t_2
\end{aligned}$$

$$\begin{aligned}
f \ t_1 \ @(\text{Ty_Var } v1) \ (\text{Ty_Var } v2) \\
\mid v1 \equiv v2 &= \text{res } t_1 \\
f \ t_1 \ @(\text{Ty_Var } v1) \ t_2 &= \text{occurBind } v1 \ t_2 \\
f \ t_1 \quad t_2 \ @(\text{Ty_Var } v2) &= \text{occurBind } v2 \ t_1
\end{aligned}$$

$$\begin{aligned}
f \ t_1 \ @(\text{Ty_App } (\text{Ty_App } (\text{Ty_Con } c1) \ ta_1) \ tr_1) \\
t_2 \ @(\text{Ty_App } (\text{Ty_App } (\text{Ty_Con } c2) \ ta_2) \ tr_2) \\
\mid \text{hsnIsArrow } c1 \ \wedge \ c1 \equiv c2 \\
&= \text{comp } ta_2 \ tr_1 \ ta_1 \ tr_2 \ (\lambda a \ r \rightarrow [a] \text{'mkTyArrow' } r) \\
f \ t_1 \ @(\text{Ty_App } tf_1 \ ta_1) \\
t_2 \ @(\text{Ty_App } tf_2 \ ta_2) \\
&= \text{comp } tf_1 \ ta_1 \ tf_2 \ ta_2 \ \text{Ty_App} \\
f \ t_1 \quad t_2 &= \text{err } [\text{Err_UnifyClash } ty_1 \ ty_2 \ t_1 \ t_2] \\
\text{err } e &= \text{emptyFO}\{\text{foErrL} = e\}
\end{aligned}$$

Although this version of the implementation of *fitsIn* resembles the previous one it differs in the following aspects:

- The datatype *FIOut* returned by *fitsIn* has an additional field *foCnstr* holding found constraints. This requires constraints to be combined for composite types like the *App* variant of *Ty*.
- The function *bind* creates a binding for a type variable to a type. The use of *bind* is shielded by *occurBind* which checks if the type variable for which a binding is created does not occur free in the bound type too. This is to prevent (e.g.) $a \leq a \rightarrow a$ to succeed. This is because it is not clear if $a \mapsto a \rightarrow a$ should be the resulting

- constraint or $a \mapsto (a \rightarrow a) \rightarrow (a \rightarrow a)$ or one of infinitely many other possible solutions. A so called *infinite type* like this is inhibited by the so called *occurs check*.
- An application *App* recursively fits its components with components of another *App*. The constraints from the first fit *fit* are applied immediately to the following component before fitting that one. This is to prevent $a \rightarrow a \leq Int \rightarrow Char$ from finding two conflicting constraints $[a \mapsto Int, a \mapsto Char]$ instead of properly reporting an error.

3.3 Reconstructing types for Expr

Constraints are used to make knowledge found about previously unknown types explicit. The typing rules in Fig. 6 (and Fig. 7, Fig. 9) in principle do not need to be changed. The only reason to adapt some of the rules to the variant in Fig. 11 is to clarify the way constraints are used.

$\Gamma; \sigma^k \stackrel{expr}{\vdash} e : \sigma \rightsquigarrow C$

$\frac{\Gamma; \sigma^a \stackrel{expr}{\vdash} e_2 : _ \rightsquigarrow C_2 \quad \Gamma; v \rightarrow \sigma^k \stackrel{expr}{\vdash} e_1 : \sigma^a \rightarrow \sigma \rightsquigarrow C_1 \quad v \text{ fresh}}{\Gamma; \sigma^k \stackrel{expr}{\vdash} e_1 e_2 : C_2 \sigma \rightsquigarrow C_{2..1}} \quad (\text{e-app2})$	$\frac{\Gamma^p, \Gamma; \sigma^r \stackrel{expr}{\vdash} e : \sigma^e \rightsquigarrow C_3 \quad \sigma^p \stackrel{pat}{\vdash} p : _ ; \Gamma^p \rightsquigarrow C_2 \quad \stackrel{fit}{\vdash} v_1 \rightarrow v_2 \leq \sigma^k : \sigma^p \rightarrow \sigma^r \rightsquigarrow C_1 \quad v_i \text{ fresh}}{\Gamma; \sigma^k \stackrel{expr}{\vdash} \lambda p \rightarrow e : C_3 \sigma^p \rightarrow \sigma^e \rightsquigarrow C_{3..1}} \quad (\text{e-lam2})$
$\frac{(i \mapsto \sigma^i) \in \Gamma \quad \stackrel{fit}{\vdash} \sigma^i \leq \sigma^k : \sigma \rightsquigarrow C}{\Gamma; \sigma^k \stackrel{expr}{\vdash} i : \sigma \rightsquigarrow C} \quad (\text{e-ident2})$	$\frac{\stackrel{fit}{\vdash} (v_1, v_2, \dots, v_n) \leq \sigma^r : (\sigma_1, \sigma_2, \dots, \sigma_n) \rightsquigarrow C \quad _ \rightarrow \dots \rightarrow \sigma^r \equiv \sigma^k \quad v_i \text{ fresh}}{\Gamma; \sigma^k \stackrel{expr}{\vdash} .n : \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow (\sigma_1, \sigma_2, \dots, \sigma_n) \rightsquigarrow C} \quad (\text{e-con2})$
$\frac{\stackrel{fit}{\vdash} Int \leq \sigma^k : \sigma \rightsquigarrow C}{\Gamma; \sigma^k \stackrel{expr}{\vdash} \text{minint} \dots \text{maxint} : \sigma \rightsquigarrow C} \quad (\text{e-int2})$	

Fig. 11. Type inferencing for expressions (using constraints)

The type rules in Fig. 11 enforce an order in which checking and inferring types has to be done.

Actually, the rules in Fig. 11 should be even more specific in how constraints flow around if we want to be closer to the corresponding AG description. The AG specifies a C to be threaded instead of just returned bottom-up:

ATTR $AllExpr$ [$| tyCnstr : C |$]

Its use in an expression application is as follows:

SEM $Expr$

$| App \text{ loc.knFunTy} := [mkNewTyVar @lUniq] 'mkTyArrow' @lhs.knTy$
 $.ty := @arg.tyCnstr > @fTy$

AG: Redefining an attribute value. Normally a value for an attribute may be associated with an attribute only once, using $=$ in a rule. It is an error if multiple rules for an attribute are present. If $:=$ is used instead, any previous definition is overridden and no error message is generated. In this context previous means “textually occurring earlier”. Because the AG system’s flexibility finds its origin in the independence of textual locations of declarations and definitions, $:=$ should be used with care. For these notes the order in which redefinitions appear is the same as their textual appearance in these notes, which again is the same as the sequence of versions of EH.

This definition builds on top of the previous version by redefining some attributes (indicated by $:=$ instead of $=$). If this happens a reference to the location (in these notes) of the code on top of which the new code is added can be found⁵.

To correspond better with the related AG code the rule e-app2 should be:

$$\frac{\begin{array}{l} C_1; \Gamma; \sigma^a \vdash^{expr} e_2 : - \rightsquigarrow C_2 \\ C^k; \Gamma; v \rightarrow C^k \sigma^k \vdash^{expr} e_1 : \sigma^a \rightarrow \sigma \rightsquigarrow C_1 \\ v \text{ fresh} \end{array}}{C^k; \Gamma; \sigma^k \vdash^{expr} e_1 e_2 : C_2 \sigma \rightsquigarrow C_2} \quad (\text{e-app2B})$$

The flow of constraints is made explicit as they are passed through the rules, from the context (left of \vdash) to a result (right of \rightsquigarrow). We feel this does not benefit clarity, even though it is correct. It is our opinion that typing rules serve their purpose best by providing a basis for proof as well as understanding and discussion. An AG description serves its purpose best by showing how it really is implemented. Used in tandem they strengthen each other.

⁵ This is not an ideal solution to display combined fragments. A special purpose editor would probably do a better job of browsing textually separated but logically related pieces of code.

An implementation by necessity imposes additional choices, in order to make a typing rule into an algorithmic solution. For example, our AG description preserves the following invariant:

- A resulting type has all known constraints applied to it, here ty .

but as this invariant is not kept for $knTy$ and $valGam$ it requires to

- Explicitly apply known constraints to the inherited known type $knTy$.
- Explicitly apply known constraints to types from a Γ , here $valGam$.

The type rules in Fig. 11 do not mention the last two constraint applications (rule `e-app2B` does), and this will also be omitted for later typing rules. However, the constraint applications are shown by the AG code for the `App` alternative and the following `Var` alternative:

```

SEM Expr
  | Var loc.fTy      := @lhs.tyCnstr > @gTy
    fo                := @fTy ≤ (@lhs.tyCnstr > @lhs.knTy)
    lhs.tyCnstr = foCnstr @fo > @lhs.tyCnstr

```

The rules for constants all resemble the one for `Int`, rule `e-int2`. Their implementation additionally takes care of constraint handling:

```

SEM Expr
  | IConst CCnst
    loc.fo      := @fTy ≤ (@lhs.tyCnstr > @lhs.knTy)
    lhs.tyCnstr = foCnstr @fo > @lhs.tyCnstr

```

The handling of products does not differ much from the previous implementation. A rule `e-con2` has been included in the typing rules, as a replacement for rule `e-prod1B` (Fig. 7) better resembling its implementation. Again the idea is to exploit that in this version of EH tupling is the only way to construct an aggregate value. A proper structure for its type is (again) enforced by `fitsIn`.

```

SEM Expr
  | Con loc.fo      = let gTy      = mkTyFreshProdFrom @lUniq (hsnProdArity @nm)
    foKnRes = gTy ≤ (@lhs.tyCnstr > tyArrowRes @lhs.knTy)
    in foKnRes{foTy = tyProdArgs (foTy foKnRes)
    'mkTyArrow' (foTy foKnRes)}
    .ty      := foTy @fo
    lhs.tyCnstr = foCnstr @fo > @lhs.tyCnstr

```

Finally,

```

SEM Expr
  | Lam loc .(argTy, resTy, funTy)
    := let [a, r] = mkNewTyVarL 2 @lUniq

```

```

                                in (a, r, [a] 'mkTyArrow' r)
foKnFun      := @funTy ≤ (@lhs.tyCnstr > @lhs.knTy)
arg .knTy     := @argTy
.tyCnstr     = foCnstr @foKnFun > @lhs.tyCnstr
body.knTy    := @resTy
loc .bodyTyCnstr = @body.tyCnstr
.ty          := [@bodyTyCnstr > @arg.ty] 'mkTyArrow' @body.ty

```

which uses some additional functions for creating type variables

```

mkNewTyVar :: UID → Ty
mkNewTyVar u = let (←, v) = mkNewUID u in mkTyVar v

mkNewUIDTyVarL :: Int → UID → ([UID], TyL)
mkNewUIDTyVarL sz u = let vs = mkNewUIDL sz u in (vs, map mkTyVar vs)

mkNewTyVarL :: Int → UID → TyL
mkNewTyVarL sz u = snd (mkNewUIDTyVarL sz u)

```

Some observations are in place:

- The main difference with the previous implementation is the use of type variables to represent unknown knowledge. Previously \square was used for that purpose, for example, the rule `e-lam2` and its implementation show that fresh type variables v_i in $v_1 \rightarrow v_2$ are used instead of $\square \rightarrow \square$ to enforce a $\dots \rightarrow \dots$ structure. If \square still would be used, for example in:

```

let id = λx → x
in id 3

```

the conclusion would be drawn that $id :: \square \rightarrow \square$, whereas $id :: v \rightarrow v$ would later on have bound $v \mapsto Int$ (at the application `id 3`). So, \square represents “unknown knowledge”, a type variable v represents “not yet known knowledge” to which the inferring process later has to refer to make it “known knowledge”.

- Type variables are introduced under the condition that they are “fresh”. For a typing rule this means that these type variables are not in use elsewhere, often more concretely specified with a condition $v \notin fv(\Gamma)$. Freshness in the implementation is implemented via unique identifiers UID.

3.4 Reconstructing types for PatExpr

In the previous version of EH we were only interested in bindings for identifiers in a pattern. The type of a pattern was already known via a corresponding type signature. For this version this is no longer the case so the structure of a pattern reveals already some type structure. Hence we compute types for patterns too and use this type as the known type if no type signature is available.

$$\boxed{\sigma^k \stackrel{pat}{\vdash} p : \sigma; \Gamma^p \rightsquigarrow C}$$

$$\frac{\begin{array}{l} \stackrel{fit}{\vdash} C_1 \sigma^k \leq \sigma^d : \sigma \rightsquigarrow C_2 \\ \sigma^d \rightarrow () \equiv \sigma^p \\ - \stackrel{pat}{\vdash} p : \sigma^p; \Gamma^p \rightsquigarrow C_1 \\ p \equiv p_1 p_2 \dots p_n, n \geq 1 \end{array}}{\sigma^k \stackrel{pat}{\vdash} p : \sigma; \Gamma^p \rightsquigarrow C_{2..1}} \quad (\text{p-apptop2})$$

$$\frac{\begin{array}{l} \text{dom}(\Gamma_1^p) \cap \text{dom}(\Gamma_2^p) = \emptyset \\ \sigma_1^a \stackrel{pat}{\vdash} p_2 : -; \Gamma_2^p \rightsquigarrow C_2 \\ - \stackrel{pat}{\vdash} p_1 : \sigma^d \rightarrow (\sigma_1^a, \sigma_2^a, \dots, \sigma_n^a); \Gamma_1^p \rightsquigarrow C_1 \end{array}}{- \stackrel{pat}{\vdash} p_1 p_2 : C_2(\sigma^d \rightarrow (\sigma_2^a, \dots, \sigma_n^a)); \Gamma_1^p, \Gamma_2^p \rightsquigarrow C_{2..1}} \quad (\text{p-app2})$$

$$\frac{\sigma^k \neq \square}{\sigma^k \stackrel{pat}{\vdash} i : \sigma^k; [i \mapsto \sigma^k] \rightsquigarrow []} \quad (\text{p-var2}) \quad \frac{v_i \text{ fresh}}{- \stackrel{pat}{\vdash} I : \sigma; (v_1, v_2, \dots, v_n) \rightarrow (v_1, v_2, \dots, v_n) \rightsquigarrow []} \quad (\text{p-con2})$$

Fig. 12. Type inferencing for pattern (using constraints)

Finally, the type itself and additional constraints are returned:

```

SEM PatExpr
| IConst loc   .ty      = tyInt
| CConst loc   .ty      = tyChar
| AppTop loc   .fo      = @lhs.knTy ≤ @knResTy
                        .ty      = foTy @fo
                        patExpr.tyCnstr = foCnstr @fo > @lhs.tyCnstr
                        lhs   .ty      = @patExpr.tyCnstr > @ty
| App   arg   .knTy    := @func.tyCnstr > @knArgTy
| Con   loc   .ty      = Ty_Any

```

The careful reader may have observed that the direction of \leq for fitting actual (synthesized, bottom-up) and known type (inherited, top-down) is the opposite of the direction used for expressions. This is a result of a difference in the meaning of an expression and a pattern. An expression builds a value from bottom to top as seen in the context of an abstract syntax tree. A pattern dissects a value from top to bottom. The flow of data is opposite, hence the direction of \leq too.

3.5 Declarations

Again, at the level of declarations all is tied together. Because we first gather information about patterns and then about expressions two separate threads for gathering constraints are used, *patTyCnstr* and *tyCnstr* respectively.

```

SEM Expr
| Let decls.patTyCnstr = @lhs.tyCnstr
  .tyCnstr      = @decls.patTyCnstr

ATTR AllDecl [| tyCnstr : C   patTyCnstr : C |]

SEM Decl
| Val   patExpr.tyCnstr   = @lhs.patTyCnstr
  lhs   .patTyCnstr = @patExpr.tyCnstr
  expr  .tyCnstr   = @lhs.tyCnstr

SEM AGIf
| AGIf expr .tyCnstr   = emptyCnstr

```

If a type signature has been given it is used as the known type for both expression and pattern. If not, the type of a pattern is used as the known type for an expression.

```

SEM Decl
| Val expr.knTy = if @hasTySig then @knTy else @patExpr.ty

```

3.6 Partial type signatures: a test case for extensibility

Partial type signatures allow the programmer to specify only a part of a type in a type signature. The description of the implementation of this feature is separated from the discussion of other features to show the effects of an additional feature on the compiler. In other words, the following is an impact analysis.

First, both abstract syntax and the parser (not included in these notes) contain an additional alternative for parsing the “. . .” notation chosen for unspecified type information designated by *Wild* for wildcard:

```
DATA TyExpr
  | Wild
```

A wildcard type is treated in the same way as a type variable as it also represents unknown type information:

```
SEM TyExpr
  | Wild loc.tyVarId = @lUniq
    .tgi      = TyGamInfo (mkNewTyVar @tyVarId)
```

```
SEM TyExpr
  | Wild lhs.ty = tgiTy @tgi
```

Changes also have to be made to the omitted parts of the implementation, in particular the pretty printing of the AST and generation of unique identifiers. We mention the necessity of this but omit the relevant code.

The pretty printing of a type signature is enhanced a bit further by either printing the type signature (if no wildcard types are present in it) or by printing the type of the type signature combined with all found constraints. The decision is based on the presence of wildcard type variables in the type signature:

```
ATTR TyExpr [| tyVarWildL USE{ ++ }{ [] } : TyVarIDL]
```

```
SEM TyExpr
  | Wild lhs.tyVarWildL = [ @tyVarId ]
```

The set of all constraints is retrieved at the root of the AST and passed back into the tree:

```
ATTR AllDecl [ finValGam : ValGam || ]
```

```
ATTR AllNT [ finTyCnstr : C || ]
```

```
SEM Expr
  | Let decls.finValGam = @lhs.finTyCnstr > @lValGam
```

```
SEM Decl
  | TySig loc .finalTy = vgiTy.fromJust.valGamLookup @nm
    $ @lhs.finValGam
```

```
SEM AGItf
```

```
| AGIf expr .finTyCnstr = @expr.tyCnstr
```

4 EH 3: polymorphic type inferencing

The third version of EH adds polymorphism, in particular so-called parametric polymorphism which allows functions to be used on arguments of differing types. For example

```
let id :: a -> a
    id = \x -> x
    v = (id 3, id 'x')
in v
```

gives $v :: (Int, Char)$ and $id :: \forall a.a \rightarrow a$. The polymorphic identity function id accepts a value of any type a , giving back a value of the same type a . Type variables in the type signature are used to specify polymorphic types. Polymorphism of a type variable in a type is made explicit in the type by the use of a universal quantifier `forall`, or \forall . The meaning of this quantifier is that a value with a universally quantified type can be used with different types for the quantified type variables.

The type signature may be omitted, and in that case the same type will still be inferred. However, the reconstruction of the type of a value for which the type signature is omitted has its limitations, the same as for Haskell98 [31]. Haskell98 also restricts what can be described by type signatures.

Polymorphism is allowed for identifiers bound by a **let**-expression, not for identifiers bound by another mechanism such as parameters of a lambda expression. The following variant of the previous example is therefore not to be considered correct:

```
let f :: (a -> a) -> Int
    f = \i -> i 3
    id :: a -> a
    id = \x -> x
in f id
```

It will give the following output:

```
let f :: (a -> a) -> Int
    f = \i -> i 3
    {- ***ERROR(S):
        In '\i -> i 3':
        ... In 'i':
            Type clash:
            failed to fit: c_2_0 -> c_2_0 <= v_7_0 -> Int
```



```

In '\i -> (i 3,i 'x)':
... In ''x'':
  Type clash:
    failed to fit: Char <= Int
    problem with : Char <= Int -}
id = \x -> x
{- [ id:forall a . a -> a, f:forall a . (Int -> a) -> (a,a) ] -}
in let v = f id
    {- [ v:(Int,Int) ] -}
in v

```

Because *i* is not allowed to be polymorphic it can either be used on *Int* or *Char*, but not both.

These problems can be overcome by allowing higher ranked polymorphism in type signatures. Later versions of EH deal with this problem, but this is not included in these notes. This version of EH resembles Haskell98 in these restrictions.

The reason not to allow explicit types to be of assistance to the type inferencer is that Haskell98 and this version of EH have as a design principle that all explicitly specified types in a program are redundant. That is, after removal of explicit type signatures, the type inferencer can still reconstruct all types. It is guaranteed that all reconstructed types are the same as the removed signatures or more general, that is, the type signatures are a special case of the inferred types. This guarantee is called the principal type property [9,26,18]. However, type inferencing also has its limits. In fact, the richer a type system becomes, the more difficult it is for a type inferencing algorithm to make the right choice for a type without the programmer specifying additional helpful type information.

4.1 Type language

The type language for this version of EH adds quantification by means of the universal quantifier \forall :

$$\begin{aligned}
\sigma &= \text{Int} \mid \text{Char} \\
&\mid (\sigma, \dots, \sigma) \\
&\mid \sigma \rightarrow \sigma \\
&\mid v \mid f \\
&\mid \forall \alpha. \sigma
\end{aligned}$$

A *f* stands for a fixed type variable, a type variable which may not be constrained but still stands for an unknown type. A *v* stands for a plain type variable as used in the previous EH version. A series of consecutive quantifiers in $\forall \alpha_1. \forall \alpha_2. \dots \sigma$ is abbreviated to $\forall \bar{\alpha}. \sigma$.

The type language suggests that a quantifier may occur anywhere in a type. This is not the case, quantifiers may only be on the top of a type; this version of EH takes care to ensure this. A second restriction is that quantified types are present only in a Γ whereas no \forall 's are present in types used throughout type inferencing expressions and patterns. This is to guarantee the principle type property.

The corresponding abstract syntax for a type needs additional alternative to represent a quantified type. For a type variable we also have to remember to which category it belongs, either *plain* or *fixed*:

```
DATA Ty
  | Var tv   : {TyVarId}
             categ : TyVarCateg
```

```
DATA TyVarCateg
  | Plain
  | Fixed
```

```
DATA Ty
  | Quant tv : {TyVarId}
             ty : Ty
```

```
SET AllTyTy = Ty
SET AllTy   = AllTyTy
SET AllAllTy = AllTy TyVarCateg
```

together with convenience functions for constructing these types:

```
mkTyVar :: TyVarId → Ty
mkTyVar tv = Ty_Var tv TyVarCateg_Plain

mkTyQu :: TyVarIdL → Ty → Ty
mkTyQu tvL t = foldr ( $\lambda tv t \rightarrow Ty\_Quant\ tv\ t$ ) t tvL
```

We will postpone the discussion of type variable categories until section 92.

The syntax of this version of EH only allows type variables to be specified as part of a type signature. The quantifier \forall cannot be explicitly denoted. We only need to extend the abstract syntax for types with an alternative for type variables:

```
DATA TyExpr
  | Var nm : {HsName}
```

4.2 Type inferencing

Compared to the previous version the type inferencing process does not change much. Because types used throughout the type inferencing of expressions and patterns do not contain \forall quantifiers, nothing has to be changed there.

Changes have to be made to the handling of declarations and identifiers though. This is because polymorphism is tied up with the way identifiers for values are introduced and used.

$$\boxed{\Gamma; \sigma^k \stackrel{expr}{\vdash} e : \sigma \rightsquigarrow C}$$

$$\begin{array}{c}
\Gamma^q, \Gamma; \sigma^k \stackrel{expr}{\vdash} e : \sigma^e \rightsquigarrow C_3 \\
\Gamma^q \equiv [(i \mapsto \forall \bar{\alpha}. \sigma) \mid (i \mapsto \sigma) \leftarrow C_{2..1} \Gamma^p, \bar{\alpha} \equiv \text{ftv}(\sigma) - \text{ftv}(C_{2..1} \Gamma)] \\
\Gamma^p, \Gamma; \sigma^p \stackrel{expr}{\vdash} e^i : _ \rightsquigarrow C_2 \\
\frac{\square \stackrel{pat}{\vdash} p : \sigma^p; \Gamma^p \rightsquigarrow C_1}{\Gamma; \sigma^k \stackrel{expr}{\vdash} \mathbf{let} p = e^i \mathbf{in} e : \sigma^e \rightsquigarrow C_{3..1}} \quad (\text{e-let3})
\end{array}$$

$$\begin{array}{c}
(\Gamma^q - [i \mapsto _]) \vdash [i \mapsto \sigma^q] \vdash \Gamma; \sigma^k \stackrel{expr}{\vdash} e : \sigma^e \rightsquigarrow C_3 \\
\Gamma^q \equiv [(i \mapsto \forall \bar{\alpha}. \sigma) \mid (i \mapsto \sigma) \leftarrow C_{2..1} \Gamma^p, \bar{\alpha} \equiv \text{ftv}(\sigma) - \text{ftv}(C_{2..1} \Gamma)] \\
(\Gamma^p - [i \mapsto _]) \vdash [i \mapsto \sigma^q] \vdash \Gamma; \sigma^j \stackrel{expr}{\vdash} e^i : _ \rightsquigarrow C_2 \\
\sigma^q \equiv \forall \bar{\alpha}. \sigma^i \\
\sigma^j \equiv [\alpha_j \mapsto f_j] \sigma^i, f_j \text{ fresh} \\
\bar{\alpha} \equiv \text{ftv}(\sigma^i) \\
p \equiv i \vee p \equiv i @ \dots \\
\frac{\sigma^i \stackrel{pat}{\vdash} p : _; \Gamma^p \rightsquigarrow C_1}{\Gamma; \sigma^k \stackrel{expr}{\vdash} \mathbf{let} i :: \sigma^i; p = e^i \mathbf{in} e : \sigma^e \rightsquigarrow C_{3..1}} \quad (\text{e-let-tysig3})
\end{array}$$

$$\frac{\begin{array}{c} (i \mapsto \forall [\alpha_j]. \sigma^i) \in \Gamma \\ \stackrel{ft}{\vdash} [\alpha_j \mapsto v_j] \sigma^i \leq \sigma^k : \sigma \rightsquigarrow C \\ v_j \text{ fresh} \end{array}}{\Gamma; \sigma^k \stackrel{expr}{\vdash} i : \sigma \rightsquigarrow C} \quad (\text{e-ident3})$$

Fig. 13. Type inferencing for expressions with quantifier \forall

A quantified type, also often named *type scheme*, is introduced in rule **e-let3** and rule **e-let-tysig3** and instantiated in rule **e-ident3**, see Fig. 13. We will first look at the instantiation.

Instantiation A quantified type is introduced in the type inferencing process whenever a value identifier having that type is occurs in an expression:

SEM Expr

$$| \text{Var } \mathbf{loc}.fTy := @\mathbf{lhs}.tyCnstr \succ tyInst @lUniq @gTy$$

We may freely decide what type the quantified type variables may have as long as each type variable stands for a monomorphic type. However, at this point it is not known which type a type variable stands for, so fresh type variables are used instead. This is called *instantiation*, or specialization. The resulting instantiated type partakes in the inference process as usual.

The removal of the quantifier and replacement of all quantified type variables with fresh type variables is done by *tyInst*:

$$\begin{aligned} tyInst' &:: (TyVarId \rightarrow Ty) \rightarrow UID \rightarrow Ty \rightarrow Ty \\ tyInst' \text{ mkFreshTy } uniq \text{ ty} &= s \succ ty' \\ \text{where } i \text{ u } (Ty_Quant \ v \ t) &= \mathbf{let} \ (u', v') = mkNewUID \ u \\ &\quad (s, t') = i \ u' \ t \\ &\quad \mathbf{in} \ ((v \mapsto (mkFreshTy \ v')) \succ s, t') \\ i \ _ \ t &= (emptyCnstr, t) \\ (s, ty') &= i \ uniq \ ty \\ tyInst &:: UID \rightarrow Ty \rightarrow Ty \\ tyInst &= tyInst' \text{ mkTyVar} \end{aligned}$$

Function *tyInst* strips all quantifiers and substitutes the quantified type variables with fresh ones. It is assumed that quantifiers occur only at the top of a type.

Quantification The other way around, quantifying a type, happens when a type is bound to a value identifier and added to a Γ . The way this is done varies with the presence of a type signature. Rule *e-let3* and rule *e-let-tysig3* (Fig. 13) specify the respective variations.

A type signature itself is specified without explicit use of quantifiers. These need to be added for all introduced type variables, except the ones specified by means of ‘...’ in a partial type signature:

SEM Decl

$$\begin{aligned} | TySig \ \mathbf{loc}.sigTy &= tyQuantify (\in @tyExpr.tyVarWildL) @tyExpr.ty \\ .gamSigTy &:= @sigTy \end{aligned}$$

A type signature simply is quantified over all free type variables in the type using

$$\begin{aligned} tyQuantify &:: (TyVarId \rightarrow Bool) \rightarrow Ty \rightarrow Ty \\ tyQuantify \ tvIsBound \ ty &= mkTyQu (filter (\neg.tvIsBound) (ftv \ ty)) \ ty \end{aligned}$$

Type variables introduced by a wildcard may not be quantified over because the type inferencer will fill in the type for those type variables.

We now run into a problem which will be solved no sooner than the next version of EH. In a declaration of a value (variant *Val* of *Decl*) the type signature acts as a known type *knTy* against which checking of the value expression takes place. Which type do we use for that purpose, the quantified *sigTy* or the unquantified *tyExpr.ty*?

- Suppose the *tyExpr.ty* is used. Then, for the erroneous

```

let id :: a → a
      id = λx → 3
in ...

```

we end up with fitting $v_1 \rightarrow Int \leq a \rightarrow a$. This can be accomplished via constraints $[v_1 \mapsto Int, a \mapsto Int]$. However, *a* was supposed to be chosen by the caller of *id*. Now it is constrained by the body of *id* to be an *Int*. Somehow constraining *a* whilst being used as part of a known type for the body of *id* must be inhibited.

- Alternatively, *sigTy* may be used. However, the inferencing process and the fitting done by *fitsIn* cannot (yet) handle types with quantifiers.

For now, this can be solved by replacing all quantified type variables of a known type with type constants:

```

SEM Decl
  | Val loc.knTy := tyInstKnown @lUniq @sigTy

```

by using a variant of *tyInst*:

```

tyInstKnown :: UID → Ty → Ty
tyInstKnown = tyInst' (λtv → Ty_Var tv TyVarCateg_Fixed)

```

This changes the category of the fresh type variable replacing the quantified type variable to ‘fixed’. A *fixed type variable* is like a plain type variable but may not be constrained, that is, bound to another type. This means that *fitsIn* has to be adapted to prevent this from happening. The difference with the previous version only lies in the handling of type variables. Type variables now may be bound if not fixed, and to be equal only if their categories match too. For brevity the new version of *fitsIn* is omitted.

Generalization/quantification of inferred types How do we determine if a type for some expression bound to an identifier in a value declaration is polymorphic? If a (non partial) type signature is given, the signature itself describes the polymorphism via type variables explicitly. However, if for a value definition a corresponding type signature is missing, the value definition itself gives us all the information we need. We make use of the observation that a binding for a value identifier acts as a kind of boundary for that expression.

```

let id = λx → x
in ...

```

The only way the value associated with id ever will be used outside the expression bound to id , is via the identifier id . So, if the inferred type $v_1 \rightarrow v_1$ for the expression $\lambda x \rightarrow x$ has free type variables (here: $[v_1]$) and these type variables are not used in the types of other bindings, in particular those in the global Γ , we know that the expression $\lambda x \rightarrow x$ nor any other type will constrain those free type variables. The type for such a type variable apparently can be freely chosen by the expression using id , which is exactly the meaning of the universal quantifier. These free type variables are the candidate type variables over which quantification can take place, as described by the typing rules for **let**-expressions in Fig. 13 and its implementation:

```

SEM Expr
| Let loc .lSubsValGam = @decls.tyCnstr > @lValGam
      .gSubsValGam = @decls.tyCnstr > @gValGam
      .gTyTvL      = frv @gSubsValGam
      .lQuValGam   = valGamQuantify @gTyTvL @lSubsValGam
      body.valGam  := @lQuValGam
                    'gamPushGam' @gSubsValGam

```

All available constraints in the form of $decls.tyCnstr$ are applied to both global ($gValGam$) and local ($lValGam$) Γ . All types in the resulting local $lSubsValGam$ are then quantified over their free type variables, with the exception of those referred to more globally, the $gTyTvL$. We use $valGamQuantify$ to accomplish this:

```

valGamQuantify :: TyVarIdL → ValGam → ValGam
valGamQuantify globTvL = valGamMapTy (λt → tyQuantify (∈ globTvL) t)

valGamMapTy :: (Ty → Ty) → ValGam → ValGam
valGamMapTy f = gamMapEls (λvgi → vgi{vgiTy = f (vgiTy vgi)})

gamMap :: ((k, v) → (k', v')) → Gam k v → Gam k' v'
gamMap f (Gam ll) = Gam (map (map f) ll)

gamMapEls :: (v → v') → Gam k v → Gam k v'
gamMapEls f = gamMap (λ(n, v) → (n, f v))

```

The condition that quantification only may be done for type variables not occurring in the global Γ is a necessary one. For example:

```

let h :: a → a → a
      f = λx → let g = λy → (h x y, y)
          in g 3
in f 'x'

```

If the type $g :: a \rightarrow (a, a)$ would be concluded, g can be used with y an *Int* parameter, as in the example. Function f can then be used with x a *Char* parameter. This would go wrong because h assumes the types of its parameters x and y are equal. So, this justifies the error given by the compiler for this version of EH:

```

let h :: a -> a -> a
    f = \x -> let g = \y -> (h x y,y)
                {- [ g:Int -> (Int,Int) ] -}
                in g 3
    {- [ f:Int -> (Int,Int), h:forall a . a -> a -> a ] -}
in f 'x'
{- ***ERROR(S):
   In 'f 'x'' :
     ... In 'x'':
       Type clash:
         failed to fit: Char <= Int
         problem with : Char <= Int -}

```

All declarations in a **let**-expression together form what in Haskell is called a binding group. Inference for these declarations is done together and all the types of all identifiers are quantified together. The consequence is that a declaration that on its own would be polymorphic, may not be so in conjunction with an additional declaration which uses the previous declaration:

```

let id1 = λx → x
    id2 = λx → x
    v1 = id1 3
in let v2 = id2 3
    in v2

```

The types of the function *id1* and value v_1 are inferred in the same binding group. However, in this binding group the type for *id1* is $v_1 \rightarrow v_1$ for some type variable v_1 , without any quantifier around the type. The application *id1* 3 therefore infers an additional constraint $v_1 \mapsto \text{Int}$, resulting in type $\text{Int} \rightarrow \text{Int}$ for *id1*

```

let id1 = \x -> x
    id2 = \x -> x
    v1 = id1 3
    {- [ v1:Int, id2:forall a . a -> a, id1:Int -> Int ] -}
in let v2 = id2 3
    {- [ v2:Int ] -}
    in v2

```

On the other hand, *id2* is used after quantification, outside the binding group, with type $\forall a.a \rightarrow a$. The application *id2* 3 will not constrain *id2*.

In Haskell binding group analysis will find groups of mutually dependent definitions, each of these called a binding group. These groups are then ordered according to “define

before use” order. Here, for EH, all declarations in a **let**-expression automatically form a binding group, the ordering of two binding groups d_1 and d_2 has to be done explicitly using sequences of **let** expressions: **let** d_1 **in let** d_2 **in**...

Being together in a binding group can create a problem for inferencing mutually recursive definitions, for example:

```
let  $f_1 = \lambda x \rightarrow g_1 x$ 
     $g_1 = \lambda y \rightarrow f_1 y$ 
     $f_2 :: a \rightarrow a$ 
     $f_2 = \lambda x \rightarrow g_2 x$ 
     $g_2 = \lambda y \rightarrow f_2 y$ 
in 3
```

This results in

```
let  $f_1 = \lambda x \rightarrow g_1 x$ 
     $g_1 = \lambda y \rightarrow f_1 y$ 
     $f_2 :: a \rightarrow a$ 
     $f_2 = \lambda x \rightarrow g_2 x$ 
     $g_2 = \lambda y \rightarrow f_2 y$ 
    {- [  $g_2$ :forall  $a . a \rightarrow a$ ,  $g_1$ :forall  $a . \text{forall } b . a \rightarrow b$ 
        ,  $f_1$ :forall  $a . \text{forall } b . a \rightarrow b$ ,  $f_2$ :forall  $a . a \rightarrow a$  ] -}
in 3
```

For f_1 it is only known that its type is $v_1 \rightarrow v_2$. Similarly g_1 has a type $v_3 \rightarrow v_4$. More type information cannot be constructed unless more information is given as is done for f_2 . Then also for g_2 may the type $\forall a.a \rightarrow a$ be reconstructed.

Type expressions Finally, type expressions need to return a type where all occurrences of type variable names (of type *HsName*) coincide with type variables (of type *TyVarId*). Type variable names are identifiers just as well so a *TyGam* similar to *ValGam* is used to map type variable names to freshly created type variables.

SEM *TyExpr*

```
| Var (loc.tgi, lhs.tyGam) = case tyGamLookup @nm @lhs.tyGam of
    Nothing  $\rightarrow$  let  $t = \text{mkNewTyVar } @\text{Uniq}$ 
         $tgi = \text{TyGamInfo } t$ 
    in ( $tgi$ ,  $\text{gamAdd } @nm\ tgi\ @\b{lhs}.tyGam$ )
    Just  $tgi \rightarrow (tgi, @\b{lhs}.tyGam)$ 
```

SEM *TyExpr*

```
| Var lhs.ty = tgiTy @tgi
```

Either a type variable is defined in *tyGam*, in that case the type bound to the identifier is used, otherwise a new type variable is created.

5 Remarks, experiences and conclusion

AG system. At the start of these notes we did make a claim that our “describe separately” approach contributes to a better understood implementation of a compiler, in particular a Haskell compiler. Is this true? We feel that this is the case, and thus the benefits outweigh the drawbacks, based on some observations made during this project:

The AG system provides mechanisms to split a description into smaller fragments, combine those fragments and redefine part of those fragments. An additional fragment management system did allow us to do the same with Haskell fragments. Both are essential in the sense that the simultaneous ‘existence’ of a sequence of compiler versions, all in working order when compiled, with all aspects described with the least amount of duplication, presentable in a consistent form in these notes could not have been achieved without these mechanisms and supporting tools.

The AG system allows focusing on the places where something unusual needs to be done, similar to other approaches [24]. In particular, copy rules allow us to forget about a large amount of plumbing.

The complexity of the language Haskell, its semantics, and the interaction between features is not reduced. However, it becomes manageable and explainable when divided into small fragments. Features which are indeed independent can also be described independently of each other by different attributes. Features which evolve through different versions, like the type system, can also be described separately, but can still be looked upon as a group of fragments. This makes the variation in the solutions explicit and hence increases the understanding of what really makes the difference between two subsequent versions.

On the downside, fragments for one aspect but for different compiler versions end up in different sections of these notes. This makes their understanding more difficult because one now has to jump between pages. This is a consequence of the multiple dimensions we describe: variation in language elements (new AST), additional semantics (new attributes) and variation in the implementation. Paper, on the other hand, provides by definition a linear, one dimensional rendering of this multidimensional view. We can only expect this to be remedied by the use of proper tool support (like a fragment editor or browser). On paper, proper cross referencing, colors, indexing or accumulative merging of text are most likely to be helpful.

The AG system, though in its simplicity surprisingly usable and helpful, could be improved in many areas. For example, no type checking related to Haskell code for attribute definitions is performed, nor will the generated Haskell code when compiled by a Haskell compiler produce sensible error messages in terms of the original AG code. The AG system also lacks features necessary for programming in the large. For exam-

ple, all attributes for a node live in a global namespace for that node instead of being packaged in some form of module.

Performance is expected to give problems for large systems. This seems to be primarily caused by the simple translation scheme in which all attributes together live in a tuple just until the program completes. This inhibits garbage collection of intermediate attributes that are no longer required. It also stops GHC from performing optimizations; informal experimentation with a large AG program resulted in GHC taking approximately 10 times more time with optimization flags on. The resulting program only ran approximately 15% faster. The next version of the AG system will be improved in this area [35].

AG vs Haskell. Is the AG system a better way to do Haskell programming? In general, no, but for Haskell programs which can be described by a catamorphism the answer is yes (see also section 1.4). In general, if the choices made by a function are mainly driven by some datastructure, it is likely that this datastructure can be described by an AST and the function can be described by the AG's attribution. This is the case for an abstract syntax tree or analysis of a single type. It is not the case for a function like *fitsIn* (section 35) in which decisions are made based on the combination of two (instead of just one) type.

About these notes EH and its code. The linear presentation of code and explanation might suggest that this is also the order in which the code and these notes came into existence. This is not the case. A starting point was created by programming a final version (at that time EH version 6, not included in these notes). From this version the earlier versions were constructed. After that, later versions were added. However, these later versions usually needed some tweaking of earlier versions. The consequence of this approach is that the rationale for design decisions in earlier versions become clear only in later versions. For example, an attribute is introduced only so later versions only need to redefine the rule for this single attribute. However, the initial rule for such an attribute often just is the value of another attribute. At such a place the reader is left wondering. This problem could be remedied by completely redefining larger program fragments. This in turn decreases code reuse. Reuse, that is, sharing of common code turned out to be beneficial for the development process as the use of different contexts provides more opportunities to test for correctness. No conclusion is attached to this observation, other than being another example of the tension between clarity of explanation and the logistics of compiler code management.

Combining theory and practice. Others have described type systems in a practical setting as well. For example, Jones [21] describes the core of Haskell98 by a monadic style type inferencer. Pierce [34] explains type theory and provides many small implementations performing (mainly) type checking for the described type systems in his book. On the other hand, only recently the static semantics of Haskell has been described formally [14]. Extensions to Haskell usually are formally described but once they find

their way into a production compiler the interaction with other parts of Haskell is left in the open or is at best described in the manual.

The conclusion of these observations might be that a combined description of a language, its semantics, its formal analysis (like the type system), and its implementation is not feasible. Whatever the cause of this is, certainly one contributing factor is the sheer size of all these aspects in combination. We feel that our approach contributes towards a completer description of Haskell, or any other language if described by the AG system. Our angle of approach is to keep the implementation and its explanation consistent and understandable at the same time. However, this document clearly is not complete either. Formal aspects are present, let alone a proof that the implementation is sound and complete with respect to the formal semantics. Of course one may wonder if this is at all possible; in that case our approach may well be a feasible second best way of describing a compiler implementation.

EH vs Haskell. The claim of our title also is that we provide an implementation of Haskell, thereby implying recent versions of Haskell, or at least Haskell98. However, these notes does not include the description of (e.g.) a class system; the full version of EH however does.

Acknowledgements. We thank both (anonymous) reviewers for their extremely valuable and helpful comments.

References

1. The Glasgow Haskell Compiler. <http://www.haskell.org/ghc/>, 2004.
2. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
3. Arthur Baars. Attribute Grammar System. <http://www.cs.uu.nl/~groups/ST/twiki/bin/view/Center/AttributeGrammarSystem>, 2004.
4. Richard S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21:239–250, 1984.
5. Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages, PhD Thesis*. Chalmers University of Technology, 1999.
6. Urban Boquist and Thomas Johnsson. The GRIN Project: A Highly Optimising Back End For Lazy Functional Languages. In *Selected papers from the 8th International Workshop on Implementation of Functional Languages*, 1996.
7. Didier Botlan, Le and Didier Remy. ML-F, Raising ML to the Power of System F. In *ICFP*, 2003.
8. Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of Principles of Programming Languages (POPL)*, pages 207–212. ACM, ACM, 1982.
9. Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.
10. Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A Formal Specification of the Haskell 98 Module System. In *Haskell Workshop*, pages 17–29, 2002.

11. Atze Dijkstra. EHC Web. <http://www.cs.uu.nl/groups/ST/Ehc/WebHome>, 2004.
12. Atze Dijkstra and Doaitse Swierstra. Explicit implicit parameters. Technical Report UU-CS-2004-059, Institute of Information and Computing Science, 2004.
13. Atze Dijkstra and Doaitse Swierstra. Typing Haskell with an Attribute Grammar (Part I). Technical Report UU-CS-2004-037, Department of Computer Science, Utrecht University, 2004.
14. Karl-Filip Faxen. A Static Semantics for Haskell. *Journal of Functional Programming*, 12(4):295, 2002.
15. Benedict R. Gaster and Mark P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, Languages and Programming Group, Department of Computer Science, Nottingham, November 1996.
16. Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type Classes in Haskell. *ACM TOPLAS*, 18(2):109–138, March 1996.
17. Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Generalizing Hindley-Milner Type Inference Algorithms. Technical Report UU-CS-2002-031, Institute of Information and Computing Science, University Utrecht, Netherlands, 2002.
18. J.R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.
19. Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, pages 154–173, 1987.
20. Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.
21. Mark P. Jones. Typing Haskell in Haskell. <http://www.cse.ogi.edu/~mpj/thih/>, 2000.
22. Mark P. Jones and Simon Peyton Jones. Lightweight Extensible Records for Haskell. In *Haskell Workshop*, number UU-CS-1999-28. Utrecht University, Institute of Information and Computing Sciences, 1999.
23. M.F. Kuiper and S.D. Swierstra. Using Attribute Grammars to Derive Efficient Functional Programs. In *Computing Science in the Netherlands CSN'87*, November 1987.
24. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types In Languages Design And Implementation*, pages 26–37, 2003.
25. Konstantin Laufer and Martin Odersky. Polymorphic Type Inference and Abstract Data Types. Technical Report LUC-001, Loyola University of Chicago, 1994.
26. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.
27. John C. Mitchell and Gordon D. Plotkin. Abstract Types Have Existential Type. *ACM TOPLAS*, 10(3):470–502, July 1988.
28. Martin Odersky and Konstantin Laufer. Putting Type Annotations to Work. In *Principles of Programming Languages*, pages 54–67, 1996.
29. Martin Odersky, Martin Sulzmann, and Martin Wehr. Type Inference with Constrained Types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, 1997.
30. Nigel Perry. The Implementation of Practical Functional Programming Languages, 1991.
31. Simon Peyton Jones. *Haskell 98, Language and Libraries, The Revised Report*. Cambridge Univ. Press, 2003.
32. Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. <http://research.microsoft.com/Users/simonpj/papers/putting/index.htm>, 2004.

33. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
34. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
35. Joao Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University, 1999.
36. Chung-chieh Shan. Sexy types in action. *ACM SIGPLAN Notices*, 39(5):15–22, May 2004.
37. Mark Shields and Simon Peyton Jones. First-class Modules for Haskell. In *Ninth International Conference on Foundations of Object-Oriented Languages (FOOL 9)*, Portland, Oregon, December 2001.
38. Utrecht University Software Technology Group. UUST library. <http://cvs.cs.uu.nl/cgi-bin/cvsweb.cgi/uust/>, 2004.
39. S.D. Swierstra, P.R. Azero Alocer, and J. Saraiva. Designing and Implementing Combinator Languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, number 1608 in LNCS, pages 150–206. Springer-Verlag, 1999.
40. Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.
41. Phil Wadler. Theorems for free! In *4'th International Conference on Functional Programming and Computer Architecture*, September 1989.
42. Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Theoretical Computer Science, (Special issue of selected papers from 2'nd European Symposium on Programming)*, number 73, pages 231–248, 1990.