Typing Haskell with an Attribute Grammar

Atze Dijkstra and Doaitse Swierstra

September 9, 2004

Abstract

Much has been written about type systems. Much less has been written about implementing type systems. Even less has been written about implementations of real compilers where all aspects of a compiler come together. This paper helps filling the final gap by describing the implementation of a compiler for a simplified variant of Haskell. By using an attribute grammar system, aspects of a compiler implementation are described separately and added in a sequence of steps, thereby giving a series of increasingly complex (working) compilers. Also, the source text of both paper and executable compilers come from the same source files by an underlying minimal weaving system. Therefore, sources and (this) explanation are kept consistent.

Contents

List of Figures

1 Introduction and overview

Haskell98[**?**] is a complex language, not to mention its more experimental incarnations. Though also intended as a research platform, realistic compilers for Haskell [**?**] have grown over the years and understanding of and experimentation with those compilers is not an easy task. Experimentation on a smaller scale usually is based upon relatively simple and restricted implementations [**?**], often focussing 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 Haskell[1] in this paper.
- Describing these aspects separately in order to provide a better understanding.
- Adding these aspects on top of each other, each addition gives rise to a complete compiler for a Haskell subset, finally leading to a compiler for full Haskell (and extensions).
- Using tools like the AG system, to allow for separate descriptions yet also to allow the creation of working software based on these separate descriptions.

The following sections will expand on this by looking at the intentions and purpose of this paper in more detail. This is followed by a short description of the final language for which we develop compilers throughout this paper.

1.1 Purpose

For whom is this paper intended?

- For students who wish to learn more about the implementation of functional languages. This paper also informally explains required theory, in particular theory about type systems.

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

- 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 encounter a larger example of the tools used to build the compilers in this paper. This paper uses the AG system [**?**] to separately describe aspects of the language implementation, and parser combinators [**?**, **?**] to compactly describe executable syntax. Other tools for maintaining consistency between different versions of the resulting compilers and the source code text included in this paper are also used but not discussed in here.

For this intended audience this paper provides

- A description of the implementation of a type checker/inferencer for an extended subset of Haskell. Extended in the sense of allowing higher ranked polymorphism [**?**, **?**, **?**] and existentials [**?**, **?**, **?**]. A subset in the sense of (e.g.) leaving out the class system. However, it is the intention of the authors to gradually extend the compiler towards the full functionality offered by Haskell.
- A description of the semantics of Haskell, lying between the more formal [**?**, **?**] and more implementation oriented [**?**, **?**] or similar to [**?**].
- A gradual instead of a big bang explanation. It should be noted however that gradual explanation is different from gradual evolution; the former being based on a complete version, the latter following a path to an at the time unknown final version.
- A somewhat informal proof of 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.

However, this paper does *not* provide

- 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 at occasions informally and concisely introduced where necessary, familiarity with the following will make reading and understanding this paper easier

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

It is our belief and hope that by following this fine line between theory and its implementation, we serve both who want to learn and those who want to do research are served. It is also our belief that by splitting the big problem into smaller aspects the combination can be explained in an easier way. Finally, we believe that this can only be accomplished if supported by proper tooling, currently the AG system and a weaving system TBD: [cite...], and in the future perhaps by integrated environments [**?**]. TBD: [cite Proxima, Programmatica]

In the following sections we give examples of the Haskell features present in the series of compilers described in section 2 throughout section 7 . Only short examples are given, intended to get a feel for 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 this paper deal with a different aspect, 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 most closely resembles the λ-calculus, in particular typed λ-calculus extended with **let** expressions and some basic types like *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. For example:

```
let i :: Char
    i = 5
in  i
```

is not accepted.

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

```
let id :: Int → Int
    id = λx → x
    fst :: (Int, Char) → Int
    fst = λ(a, b) → a
in  id 3
```

Functions accept one parameter only, which can be a pattern. Functions are not polymorphic (yet).

EH version 2: Explicit/implicit typing. The next version (EH version 2, section 3) allows the explicitly specified types to be omitted. The consequently missing type information has to be inferred by the compiler.

```
let i = 5
in  i
```

will reconstruct the type specification i :: *Int*.

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

```
let id = λx → x
in  let v = id 3
    in  id
```

the type *id* :: *Int* → *Int* will be reconstructed.

EH version 3: Polymorphism. The third version (EH version 3, section 4) performs standard Hindley-Milner type inference [**?**, **?**] which infers also parametric polymorphism. For example,

$$\begin{array}{l}\textbf{let}\ id = \lambda x \to x \\ \textbf{in}\ \ id\ 3\end{array}$$

gives type $id :: \forall\ a.a \to a$.

A type for a value can also be specified explicitly

$$\begin{array}{l}\textbf{let}\ id :: a \to a \\ \quad\ \ id = \lambda x \to x \\ \textbf{in}\ \ id\ 3\end{array}$$

The type signature is checked against the inferred type.

EH version 4: Higher ranked types. Standard Hindley-Milner type inference cannot infer polymorphic parameters, so-called rank-2 (or even higher ranked) polymorphism. In general, this is a hard if not impossible thing to do [**?**, **?**, **?**, **?**], so the fourth version (EH version 4, section 5) does not infer this type information but allows explicitly specified polymorphism for (e.g.) parameters.

For example, the following is allowed.

$$\begin{array}{l}\textbf{let} f :: (\forall\ a.a \to a) \to (Int, Char) \\ \quad\ \ f = \lambda i \to (i\ 3, i\ \texttt{'x'}) \\ \textbf{in}\ \ f\end{array}$$

Note that the type signature cannot be omitted here.

This version also provides some notational sugaring by allowing one to omit the explicit quantifiers from the type signature. For example, if the $\forall$ in the previous example is omitted the correct location for the quantifier is inferred, based on the occurrences of type variables in a type expression:

$$\begin{array}{l}\textbf{let} f :: (a \to a) \to (Int, Char) \\ \quad\ \ f = \lambda i \to (i\ 3, i\ \texttt{'x'}) \\ \textbf{in}\ \ f\end{array}$$

infers $f :: (\forall\ a.a \to a) \to (Int, Char)$

Specifying a complete type signature can be difficult for complicated types, so it is permitted to leave argument and results of a function unspecified using a *partial type signature*.

$$\begin{array}{l}\textbf{let}\ id\ ::\ ... \to ... \\ \quad\ \ id = \lambda x \to x \\ \quad\ \ f\ \ ::\ (a \to a) \to ... \\ \quad\ \ f\ \ = \lambda i \to (i\ 3, i\ \texttt{'x'}) \\ \textbf{in}\ f\ id\end{array}$$

Only the part which cannot be inferred is given in the signature.

Finally, type information can be hidden, or encapsulated, by using existential quantification

$$\begin{array}{l}\textbf{let}\ id\ \ \ :: \forall\ a.a \to a \\ \quad\ \ xy\ \ :: \exists\ a.(a, a \to Int) \\ \quad\ \ xy\ \ = (3, id) \\ \quad\ \ ixy :: (\exists\ a.(a, a \to Int)) \to Int \\ \quad\ \ ixy = \lambda(v, f) \to f\ v \\ \quad\ \ xy' = ixy\ xy \\ \quad\ \ pq\ \ :: \exists\ a.(a, a \to Int)\end{array}$$

$pq\ = ($'x'$, id)$ -- ERROR
in xy'

The value *xy* contains an *Int* and a function making an *Int* from the value of which the type has been hidden. The extraction is done by function *ixy*. An attempt to construct *pq* fails.

When a value of an existentially quantified type is opened, that is, identifiers are bound to its components, the hidden type becomes visible in the form of a fresh type constant. The explicit $\exists$ may also be omitted, for example $xy :: (a, a \rightarrow Int)$ is interpreted as $xy :: \exists\, a.(a, a \rightarrow Int)$.

EH version 5: Data types. The fifth version (EH version 5, section 6) adds **data** types and opening/unpacking/scrutinizing a data type value by means of a **case** expression.

```
let data List a = Nil | Cons a (List a)
in  let v = case Cons 3 Nil of
              Nil        → 5
              Cons x y → x
    in  v
```

EH version 6: Kinding. The previous version allows incorrect programs because data types can be used incorrectly:

```
let data List a = Nil | Cons a (List a)
    v :: List
in  v
```

The type of *v* is not a type of a value, and thus the type of *v* itself is not well-typed. The sixth version (EH version 6, section 7) adds kind (that is, the type of a type) inferencing. For example, the previous example gives

```
let data List a = Nil  | Cons a (List a)
    v :: List
    {- ***ERROR(S):
         In ‘List’:
           Type clash:
             failed to fit: * -> * <= *
             problem with : * -> * <= * -}
    {- [ List:List/* -> * ] -}
    {- [ Cons:forall a . a -> List a -> List a
       , unCons:forall a . List a -> (a,List a), Nil:forall a . List a
       , unNil:forall a . List a -> (), v:List ] -}
in v
```

With the notion of the kind of a type also comes the notion of polymorphism for kinds, which this version allows:

```
let data Eq a b = Eq (∀ f.f a → f b)
    id = λx → x
in  Eq id
```

infers for type constructor *Eq*

```
let data Eq a b = Eq (forall f . f a -> f b)
    {- [ Eq:Eq/Forall a . a -> a -> * ] -}
    {- [ Eq:forall a . forall b . (forall c . c a -> c b) -> Eq a b
       , unEq:forall a . forall b . Eq a b -> (forall c . c a -> c b) ] -}
in 3
```

1.3 Haskell language elements not described

The last compiler in the series of compilers in this paper still lacks many features available in Haskell. Just to mention a few:

- Binding group analysis
- Syntax directives like infix declarations
- Class system
- Modules [**?**, **?**].
- Type synonyms
- Syntactic sugar for **if**, **do**, list notation and comprehension.
- Code generation

1.4 About the presented code

The fact that multiple versions of a compiler are described and are the basis of the story around which the explanation has been woven brings some consequences

- Versions are built on top of each other. However, in practice this meant that the last version was built and subsequently simplified to earlier versions. It is not the case that the versions represent a timeline, a tale of how the different versions came into being. It also means that all versions are dependent on each other and are designed as a whole. Any desired change in the last version may imply a change in the first version. Metaphorically speaking, in order to change the grown-up compiler you may have to tweak its childhood.

  In an ideal situation this would not have been necessary; but it would be unfair if we did not mention the work that went into getting the stepwise built-up compilers to work as neatly as is does now.
- Since we do not only want to sketch the approach but want to present a complete compiler we also have to deal with many non-interesting details. The complete compiler text can be found on the website accompanying this paper [**?**].

1.5 Literature

TBD:

here???? Or merged with previous.

Similar, implementation describing

Rules only

Haskell

2 EH 1: typed λ-calculus

In this section we will build the first version of our series of compilers: the typed λ-calculus packaged in Haskell syntax in which all values need 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
```

Although the implementation of a type system will be the main focus of this section, any such implementation lives in co-existence with the complete environment/framework needed to build a compiler. Therefore, aspects conveniently omitted from subsequent sections, like parsing, connection with the abstract syntax, the use of the AG system and error reporting will also be touched upon.

First we will start with the EH language elements implemented and how they correspond to abstract syntax, followed by the translation done by parsing from concrete syntax to abstract syntax. Our first aspect described using the AG notation will be a pretty printed representation of the abstract syntax tree, reflecting the original input as closely as possible. The second aspect concerns type checking, which involves several attributes for computing a type associated with certain parts of the abstract syntax tree.

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 on the abstract syntax for EH, in particular the part defining the structure for expressions (the full syntax can be found in figure 1). A **DATA** definition is the analogue of a Haskell **data** definition, used to define a piece of the abstract syntax.

EHAbsSyn

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
```

The notation of the AG system is used to define an expression *Expr* to be a range of alternatives (or productions); for example a single variable *Var* represents the occurrence of an identifier (referring to a value introduced by a declaration), or *App* represents the application of a function to an argument. The EH fragment

```
let ic @(i, c) = (5, 'x')
    id         = λx → x
in  id i
```

is represented by the following piece of abstract syntax tree:

```
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"
```

Note that the program is incorrect for this version of EH because type signatures are missing.

```
DATA AGItf
  | AGItf 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
```

Figure 1: Abstract syntax for EH (without Expr)

Looking at this example and the rest of the abstract syntax in figure 1 we can make several observations of what is allowed to notate in EH and what can be expected from the implementation.

- Applications (*App*) and constants (*Con*) occur in expressions (*Expr*), patterns (*PatExpr*) and type expressions (*TyExpr*). This similarity is sometimes 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, which is quite similar for the different kinds of constructs.

- In the abstract syntax an alternative belongs to a nonterminal (or **DATA**), for example "*App* of *Expr*". On request the AG system generates corresponding Haskell data types with the same name as the **DATA** defined, alternatives are mapped to constructors with the name of the **DATA** combined with the name of the alternative, separated by _. For example: *Expr_App*. If necessary, the same convention will be used when referring to an alternative[2].

- Type signatures (*Decl_TySig*) and value definitions (*Decl_Val*) may be mixed freely. However, type signatures and value definitions for the same identifier are still related. For this version of EH, each identifier introduced by means of a value definition must have a corresponding type signature specification.

- Because of the textual decoupling of value definitions and type signatures, a type signature might specify the type for an identifier occurring inside a pattern:

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

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

  ```
  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.

- Composite values are created by tupling, denoted by (..,..). 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 structure consists of a *Con* applied to the elements of the tuple. On the value level a *Con* stands for a value constructor, on the type level for a type constructor. For now there is only one constructor: for tuples.

- The constructor for tuples also is the one which needs special treatment because it actually stands for a 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 get the unit type () for free as it is encoded by the name `",0"`. We also can encode the tuple with one element which is for tuples themselves quite useless as tuples have at least two elements. However, later on, in section 5 , this turns out to be convenient. This and other naming conventions are available via the following definitions for Haskell names *HsName*.

  EHCommon

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

  EHCommon

[2]In this way we overcome a problem in Haskell, where it is required that the constructors for all data types are different.

```
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

hsnProdArity                    :: HsName → Int
hsnProdArity (HNm (_ : ar))     = read ar
```

- Each application is wrapped on top with an *AppTop*. This has no meaning in itself but as we will see in section 2.3 it simplifies the pretty printing of expressions.
- The location of parenthesis around an expression is remembered by a *Parens* alternative. This is to avoid the effort of finding back appropriate places to insert parenthesis.
- *AGItf* is the top of a complete abstract syntax tree. The top of an abstract syntax tree is the place where interfacing (hence the convention *Itf*) with the outside, that is, the Haskell world takes place. At *AGItf*
  - Initialization of inherited attributes takes place.
  - Synthesized attributes are routed back into the tree as inherited attributes.
  - Synthesized attributes are passed to the outside, to the Haskell world.

  It is a convention in this paper to give all nonterminals in the abstract syntax a name with *AGItf* in it, if it plays a similar role.
- The remaining alternatives for the non-terminal *Expr* stand for their EH counterparts, for example *IConst* for an *Int* constant, and *Lam* for a λ-expression.
- Groups of nonterminals may be given a name through the **SET** directive. For example, *AllNT* (All NonTerminals) is the name for all nonterminals occurring in the abstract syntax. This provides an indirection when referring to nonterminals in attribute definitions (**ATTR**). Later compiler versions can change the definition for *AllNT* without the need to also modify the list of nonterminals for which an attribute definition **ATTR** is declared.

2.2 Parsing: from concrete to abstract (syntax)

An abstract syntax tree is obtained as the result of parsing. The parser combinator library used (see [?] and figure 2) to specify the parser for EH allows us to simultaneously define the syntactic structure and the connection to the abstract syntax. For example, the parser for the simplest of expressions

```
pExprBase = sem_Expr_IConst ⟨$⟩ pInt
          ⟨|⟩ sem_Expr_CConst ⟨$⟩ pChr
          ⟨|⟩ sem_Expr_Var    ⟨$⟩ pVar
          ⟨|⟩ sem_Expr_Con    ⟨$⟩ pCon
          ⟨|⟩ pParenProd exprAlg pExpr
```

recognises variables via *pVarid*. The parser *pVarid* is defined in a general purpose scanner library [?] about which we will say no more than that it provides basic parsers tailored towards the recognition of Haskell lexical elements. All parsers from this library return their result as a string, which can conveniently be passed as an argument to the semantic function for a variable, *sem_Expr_Var*. This latter function is generated by the AG system form the attribute grammar. This is also the case for the recognition of integer and character constants with the help appropriate helper functions that convert the string returned by the scanner parsers *pInteger* and *pChar* respectively.

```
pChr     = head ⟨$⟩ pChar
pInt     = read ⟨$⟩ pInteger
pKeyw k = pKey (show k)
pCon     = HNm ⟨$⟩ pConid
pVar     = HNm ⟨$⟩ pVarid
```

The use of semantic functions deserves some closer inspection. For each of the alternatives of a non-terminal the AG system generates a *semantic function*, that takes as argument the semantic functions corresponding to its right hand side, and constructs a function mapping inherited to synthesized attributes. The structure of such a function, that is, its type is similar to its related and also generated datatype. Similar in the sense that both take their components as their first arguments. For example, both *Expr_Var* as a data constructor and *sem_Expr_Var* take a string as their first argument. It is this similarity we exploit because we shortcut the usual two-step process of first creating the abstract syntax structure and then applying the semantic function. Instead the semantic function is applied immediately, without creating the intermediate structure. For all practical purposes, for now, this is all we need in order to be able to use these functions (see also [**?**]).

Combinator	Meaning	Result
p ⟨∗⟩ *q*	*p* followed by *q*	result of *p* applied to result of *q*
p ⟨|⟩ *q*	*p* or *q*	result of *p* or result of *q*
pSucceed r	empty input ε	*r*
f ⟨$⟩ *p*	≡ *pSucceed f* ⟨∗⟩ *p*	
pKey `"x"`	symbol/keyword x	`"x"`
p ⟨∗∗⟩ *q*	*p* followed by *q*	result of *q* applied to result of *p*
p ‘*opt*‘ *r*	≡ *p* ⟨|⟩ *pSucceed r*	
p ⟨??⟩ *q*	≡ *p* ⟨∗∗⟩ *q* ‘*opt*‘ *id*	
p ⟨∗ *q*, *p* ∗⟩ *q*, *f* ⟨$ *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

Figure 2: Parser combinators

From *pExprBase* the parser *pExpr* for the more complex expressions is built.

EHParser

```
exprAlg     = (sem_Expr_Con, sem_Expr_App
              , sem_Expr_AppTop, sem_Expr_Parens)
pExpr       = pExprPrefix ⟨∗⟩ pExpr
              ⟨|⟩ pExprApp
pExprApp    = pApp exprAlg pExprBase
pExprPrefix = sem_Expr_Let       ⟨$ pKey "let"
                ⟨∗⟩ pDecls        ⟨∗ pKey "in"
              ⟨|⟩ sem_Expr_Lam     ⟨$ pKey "\\"
                ⟨∗⟩ pPatExprBase ⟨∗ pKey "->"
```

An application *pExprApp* is a juxtapositioning of a non empty series of *pExprBase*’s. A *pExprApp* itself can be prefixed with parsers making the expression into a **let**- or λ-expression. The parser *pExprPrefix* which recognizes this prefix and returns a function doing this prefixing in terms of the abstract syntax. The result of *pExprPrefix* is a function and can therefore be applied immediately to *pExprApp* by the sequencing combinator ⟨∗⟩ .

Parser *pExprApp* is defined in terms of an algebra *exprAlg* and an abstract parser recognising applications

```
pApp alg p = mkApp alg ⟨$⟩ pList1 p
```

First, let us look at the algebra *exprAlg*, again referring to Swierstra [?] for a more thorough treatment. The abstract syntax of EH (see figure 1, page 9) has *Con*, *App* and *AppTop* alternatives for *Expr*, *TyExpr* and *PatExpr*. For all three kinds of expressions an application *f x y* maps to the same structure *AppTop* (*App* (*App f x*) *y*), where for a 2-tuple pattern *f* ≡ *Con* `",2"`, and for a function type *f* ≡ *Con* `"->"` (representing $x \to y$). The *Con*, *App* and *AppTop* depend on the particular kind of expression. The following function *mkApp* does this job of creating the application given a list of elements (here [f,x,y]) to make the application from, and the specific variants for *Con*, *App* and *AppTop*. Note that no *AppTop* is placed around a singleton list as this is not an application and the function *mkApp* is not used on an empty list.

EHCommon

```
type MkConAppAlg t = (HsName → t, t → t → t, t → t, t → t)
mkApp :: MkConAppAlg t → [t] → t
mkApp (_, app, top, _) ts
   = case ts of
       [t] → t
       _   → top (foldl1 app ts)
```

EHCommon

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

EHCommon

```
mkConApp :: MkConAppAlg t → HsName → [t] → t
mkConApp alg @(con, _, _, _) c ts = mkApp alg (con c : ts)
```

EHCommon

```
mkProdApp :: MkConAppAlg t → [t] → t
mkProdApp alg ts = mkConApp alg (hsnProd (length ts)) ts
```

It is precisely the group of functions specifying what should be done for *Con*, *App* and *AppTop* which is called an algebra.

The algebra is also used in the parser *pParenProd* recognising either a unit (), parentheses (*pE*) around a more elementary parser *pE* or a tuple (*pE*, *pE*, ...) of elements *pE*.

```
pParenProd alg @(_, _, _, par) pE
   = pParens pP
     where
       pP = mkProdApp alg ⟨$⟩ pSucceed [ ]
          ⟨|⟩ pE
             ⟨∗∗⟩  (  (λes e → mkProdApp alg (e : es))
                        ⟨$⟩ pList1 (pComma ∗⟩ pE)
                   ⟨|⟩ pSucceed par
                   )
```

The definition for *pParenProd* quite literally follows this enumeration of alternatives, where in case of parentheses around a single *pE* the parentheses do not carry any semantic meaning. Note that the parser is in a left factorized form in order to make parsing take linear time.

The parsers for patterns and type expressions also use these abstractions, for example, the parser for type expressions

```
tyExprAlg    = (sem_TyExpr_Con, sem_TyExpr_App
               , sem_TyExpr_AppTop, sem_TyExpr_Parens)
pTyExprBase  = sem_TyExpr_Con ⟨$⟩ pCon
```

```
                ⟨|⟩ pParenProd tyExprAlg pTyExpr
pTyExpr         = pChainr
                    (mkArrow tyExprAlg ⟨$ pKeyw hsnArrow)
                    pTyExprBase
```

defines a *tyExprAlg* to be used to recognise parenthesized and tupled type expressions. The parser for *pTyExpr* uses *pChainr* to recognise a list of more elementary types separated by →.

The parser for patterns is because of its similarity with the previous expression parsers given without further explanation

```
patExprAlg = (sem_PatExpr_Con, sem_PatExpr_App, sem_PatExpr_AppTop, sem_PatExpr_Parens)
pPatExpr   = pApp patExprAlg pPatExprBase

pPatExprBase = pVar ⟨**⟩ ( flip sem_PatExpr_VarAs ⟨$ pKey "@" ⟨*⟩ pPatExprBase
                         ⟨|⟩ pSucceed sem_PatExpr_Var
                         )
               ⟨|⟩ sem_PatExpr_Con ⟨$⟩ pCon
               ⟨|⟩ pParenProd patExprAlg pPatExpr
```

Finally, the parsers for the program itself and declarations should have few surprises by now, except for the use of *pBlock* recognising a list of more elementary parsers where the offside rule of Haskell is used. We will not look at this any further.

```
pDecls = foldr sem_Decls_Cons sem_Decls_Nil
                     ⟨$⟩ pBlock pOCurly pSemi pCCurly pDecl
pDecl  = sem_Decl_Val   ⟨$⟩ pPatExprBase ⟨* pKey "="  ⟨*⟩ pExpr
     ⟨|⟩ sem_Decl_TySig ⟨$⟩ pVar         ⟨* pKey "::" ⟨*⟩ pTyExpr

pAGItf = sem_AGItf_AGItf ⟨$⟩ pExpr
```

Once all parsing results are passed to semantic functions we enter the world of attributes as offered by the AG system. The machinery for making the compiler actually producing some output is not explained here but can be found in the sources. From this point onwards we will forget about that and just look at computations performed on the abstract syntax tree through the separate views as provided by the AG system.

2.3 Pretty printing: from abstract to concrete

The first aspect we define is *pp* that constructs the pretty printed representation for the abstract syntax tree, starting with the definition for *Expr*

EHPretty

```
ATTR AllNT AGItf [|| pp USE{>-<}{empty} : PP_Doc]
SEM Expr
  | IConst  loc.pp = pp (show @int)
  | CConst  loc.pp = pp (show @char)
  | Var     loc.pp = pp @nm
  | Con     loc.pp = ppCon @nm
  | Let     loc.pp = "let"
                     >#< (@decls.pp >-< @ppExtra)
                     >-< @errLetPP
                     >-< "in" >#< @body.pp
                     >-< @errBodyPP
  | App     loc.pp = @func.pp >#< @arg.pp
  | Parens  loc.pp = pp_parens @expr.pp
```

EHPretty

```
SEM Expr
```

```
  | Let loc.ppExtra = ppCmt (ppGam @lValGam)
```
EHPretty
```
  | AppTop loc.pp = ppAppTop (@expr.appFunNm, @expr.appFunPP)
                             @expr.appArgPPL @expr.pp
  | Lam    loc.pp = "\\" >|< @arg.pp >#< "->" >#< @body.pp
```

The *pp* attribute is defined (via **ATTR**) as a synthesized (because after the two |) attribute for all nonterminals of the abstract syntax. This output attribute will also be passed to "the outside world" (because it is also defined for *AGItf*). If a rule for *pp* for an alternative of a nonterminal is missing, the default definition is based on **USE** which provides a default value for alternatives without nonterminals as children (here: *empty*) and a function used to compose new values based on the values of the children (here: >−<).

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

Figure 3: Pretty printing combinators

The computation of the *pp* attribute is straightforward in the sense that only a few basic combinators are needed (see [?] and figure 3). The additional complexity arises from the encoding for applications. The problem is that straightforward printing prints as (2, 3) as ,2 2 3. This is solved at the *AppTop* alternative by gathering all arguments (here: [2, 3]) for a function (here: tuple constructor ,2, or (,)) into *appArgPPL*:

EHPretty
```
ATTR Expr [|| appFunNm : {HsName} appFunPP : PP_Doc appArgPPL : PP_DocL]
SEM Expr
  | Con     lhs.appFunNm  = @nm
  | App     lhs.appFunNm  = @func.appFunNm
               .appArgPPL = @func.appArgPPL ++ [@arg.pp]
               .appFunPP  = @func.appFunPP
  | * − Con App
            lhs.appFunNm  = hsnUnknown
  | * − App lhs.appArgPPL = [ ]
               .appFunPP  = @pp
```

The *appArgPPL* and the two other required attributes *appFunNm* for the name and *appFunPP* for the pretty printing of the function which is applied all are used to gather information from deeper within the syntax tree for further processing higher up in the tree by *ppAppTop*

EHCommon
```
ppAppTop :: PP arg ⇒ (HsName, arg) → [arg] → PP_Doc → PP_Doc
ppAppTop (conNm, con) args dflt
  = if      hsnIsArrow conNm then ppListSep "" "" ("  " ++ show hsnArrow ++ "  ") args
    else if hsnIsProd conNm  then ppListSep "(" ")" "," args
                             else dflt
```

EHCommon
```
type PP_DocL = [PP_Doc]
ppListSep :: (PP s, PP c, PP o, PP a) ⇒ o → c → s → [a] → PP_Doc
ppListSep o c s pps = o >|< hlist (intersperse (pp s) (map pp pps)) >|< c
ppCommaList :: PP a ⇒ [a] → PP_Doc
ppCommaList = ppListSep "[" "]" ","
```

```
ppCon :: HsName → PP_Doc
ppCon nm = if    hsnIsProd nm
           then pp_parens (text (replicate (hsnProdArity nm − 1) ',' ))
           else  pp nm
ppCmt :: PP_Doc → PP_Doc
ppCmt p = "{-" >#< p >#< "-}"
```

Additionally, because the pretty printing with non-optimally placed parentheses resembles the structure of the abstract syntax tree more, this variant still is computed for the *App* and *Con* alternative in a local *pp* attribute. This *pp* value is used for producing error messages (see section 2.6), but not for the computation of *pp* at the *AppTop* alternative. For the *App* alternative parentheses are placed without consideration if this is necessary and the *Con* alternative prints (via *ppCpn*) the constructor following the Haskell convention.

We will not look into the pretty printing for patterns and type expressions because it is almost an exact replica of the code for expressions. The remainder of the *pp* related attribute computations is also omitted as they are rather straightforward uses of >||< and >−<.

2.4 Types

We will now turn our attention towards the way a type system is incorporated into EH. As we are focussed more on implementation than theory, our point of view is a more pragmatic one in that a type system is an aid or tool for program analysis and a way to ensure specific desirable properties of a program.

TBD: better intro here

2.4.1 What is a type

A *type* is a description of the interpretation of a value where value has to read here 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. The flow of values, that is, the copying between memory locations, through the execution of a program may only be such that the a copy is allowed only if the types of source and destination relate to each other in some proper fashion: a value from a source must fit in a destination. If we copy or move a bit patterns it keeps its interpretation. This is to prevent unintended use of bitpatterns, which might likely lead to the crash of a program.

A compiler uses a type system to analyse this flow and to make sure that built-in functions are only applied to patterns 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 for use in typing rules, and in a more practical setting using the AG system, that gives us an implementation too. In the following section we discuss the typing rules, the mechanism for enforcing the fitting of types and the checking itself. Types will be introduced informally and from a practical point of view instead of a more formal approach [**?**, **?**, **?**, **?**].

Types are described using a type language. Our initial type language, for this first version of EH, allows some basic types and two forms of composite types, functions and tuples. A type σ can be

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

Actually, the following definition is close to the one used by the implementation

$\sigma = Int \mid Char \mid \rightarrow \mid , \mid ,, \mid ...$
$\quad \mid \sigma\ \sigma$

This definition also introduces the possibility of describing types like *Int Int*. This being rather meaningless the first version is to be preferred We nevertheless use the second one since it is used by the implementation of later versions of EH. Here we just have to make sure no types like *Int Int* are created.

The corresponding encoding using AG notation differs in the presence of an *Any* type. This type is also denoted by $\mathbb{I}$ to indicate that it takes the role of $\perp$ as well as $\top$ usually present in type languages.

EHTyAbsSyn

```
DATA TyAGItf
  | AGItf ty : Ty

DATA Ty
  | Con nm   : {HsName}
  | App func : Ty
        arg  : Ty
  | Any

SET AllTy = Ty
```

No *AppTop* alternative is present since we want to keep this definition as simple as possible. Not only will we define attributions for *Ty*, but we will also use it to define plain Haskell values of the corresponding Haskell data type.

No *Parens* alternative is present either, again to keep the type structure as simple as possible. However, this makes pretty printing more complicated because the proper locations for parenthesis have to be computed.

2.4.2 Type pretty printing

As before, *TyAGItf* is the place where interfacing with the Haskell world takes place. The AG system also introduces proper data types to be used in the Haskell world, the following is the data type as generated by the AG system:

```
-- Ty ----------------------------
data Ty = Ty_Any
        | Ty_App (Ty) (Ty)
        | Ty_Con (String)
        deriving ( Eq,Show)
-- TyAGItf -----------------------
data TyAGItf = TyAGItf_AGItf (Ty)
             deriving ( Eq,Show)
```

This allows us to create types as plain Haskell values, and to define functionality in the AG world which is subsequently applied to types represented by Haskell data type values. As an demonstration of how this is done, pretty printing is used once again. Using the same design and tools as used for pretty printing *Expr* (and pattern/type expressions) abstract syntax we can define pretty printing for a type:

EHTyPretty

```
ATTR TyAGItf AllTy [|| pp : PP_Doc]
SEM Ty
  | Con loc.pp     = ppCon @nm
  | App loc.ppDflt = @func.pp >#< @arg.pp
```

```
             .pp      = if @isSpineRoot
                          then ppParNeed@parNeed @lhs.parNeed
                                             (ppAppTop (@appFunNm, @appFunPP) @appArgPPL @ppDflt)
                          else @ppDflt
        | Any loc.pp  = pp hsnUnknown
```

Some additional attributes are necessary to determine whether an *App* is at the top of the spine of *App*'s

EHTyCommonAG

```
ATTR Ty [|| appFunNm : HsName]
SEM Ty
  | Con lhs.appFunNm = @nm
  | App loc.appFunNm = @func.appFunNm
  | Any lhs.appFunNm = hsnUnknown
```

EHTyCommonAG

```
ATTR Ty [appSpinePos : Int ||]
SEM TyAGItf
  | AGItf ty   .appSpinePos = 0
SEM Ty
  | App   func.appSpinePos = @lhs.appSpinePos + 1
          arg  .appSpinePos = 0
          loc  .isSpineRoot  = @lhs.appSpinePos ≡ 0
```

Parenthesis. Because no explicit *Parens* alternative is present in the type structure, appropriate places to insert parenthesis have to be computed. This is solved by passing the need for parentheses as contextual information using the inherited attribute *parNeed*. However, as this part is not necessary for understanding the implementation of a type system it can safely be skipped.

EHTyPretty

```
ATTR Ty [parNeed : ParNeed   parNeedL : ParNeedL ||]
SEM TyAGItf
  | AGItf ty  .parNeed   = ParNotNeeded
              .parNeedL = [ ]
SEM Ty
  | App   loc.(parNeed, argsParNeedL)
                        = if @isSpineRoot
                          then let (here, _, args) = parNeedApp @appFunNm
                               in  (here, args)
                          else  (ParNotNeeded, @lhs.parNeedL)
          (arg.parNeed, func.parNeedL)
                        = hdAndTl @argsParNeedL
```

And supporting Haskell definitions:

EHCommon

```
data ParNeed = ParNotNeeded | ParNeededLow | ParNeeded | ParNeededHigh | ParOverrideNeeded
             deriving (Eq, Ord)
type ParNeedL = [ParNeed]
parNeedApp :: HsName → (ParNeed, ParNeed, ParNeedL)
parNeedApp conNm
  = let pr | hsnIsArrow conNm = (ParNeededLow, ParNotNeeded, [ParNotNeeded, ParNeeded])
           | hsnIsProd   conNm = (ParOverrideNeeded, ParNotNeeded, repeat ParNotNeeded)
           | otherwise         = (ParNeeded, ParNeededHigh, repeat ParNeededHigh)
    in  pr
ppParNeed :: PP p ⇒ ParNeed → ParNeed → p → PP_Doc
ppParNeed locNeed globNeed p
```

```
    = par (pp p)
  where par = if globNeed > locNeed then pp_parens else id
```

The idea here is that a *Ty* and its context jointly determine whether parentheses are needed. This need is encoded in *ParNeed*, the contextual need is passed to *ppParNeed* as the parameter *globNeed* and the local need as *locNeed*. In *ppParNeed* parentheses are added if the contextual need is higher than the local need. For example, at *AppTop* the printing of a tuple always adds parentheses via function *ppAppTop*, so no additional parentheses are needed. This is reflected in the result of *parNeedApp* which tells us that for a product locally never (encoded by *ParOverrideNeeded*, first element of result 3-tuple) additional parentheses are needed, and for the function part of the application (which is useless here since it is meant for the product constructor) and its arguments in principle not (encode by *ParNotNeeded*, second element of result tuple). The list (third element of result tuple) is meant for the arguments of the application.

The structure of the solution, that is, gather information bottom to top (e.g. *appArgPPL*) and use it higher up in the tree, is a recurring pattern. The other way around, for example *parNeedL* distributing the need for parentheses over the arguments of an application, is also a recurring pattern.

Tying in with the Haskell world. What remains, is the way the functionality is made available to the Haskell world:

EHTyPretty

```
ppTy :: Ty → PP_Doc
ppTy ty
  = let t = wrap_TyAGItf
              (sem_TyAGItf (TyAGItf_AGItf ty))
              Inh_TyAGItf
    in  pp_Syn_TyAGItf t
instance PP Ty where
  pp t = ppTy t
```

To make this work, we need to tell the AG system to generate a wrapper function and datatypes *Inh_TyAGItf* and *Syn_TyAGItf* with selector functions for defined attributes to pass inherited and synthesized attributes respectively: TBD: Update AG manual, pg 21 w.r.t. wrap

EHTyPretty

```
WRAPPER TyAGItf
```

In the code that is generated by the AG system inherited attributes become function arguments and synthesized attributes components of a cartesian product; in both cases they are identified by position. This makes interfacing to the Haskell world cumbersome. In order to overcome these problems so-called wrapper functions can be generated, which make it possible to access result e.g. by using functions generated acoording to a fixed naming convention. An example of such a fuction is *pp_Syn_TyAGItf*, which when applied to *t* accesses the synthesized *pp* attribute at an attributed nonterminal *t* of type *TyAGItf*.

2.5 Checking types

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

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

2.5.1 Type rules

For example, the following is the typing rule (taken from figure 4) for function application

$$\frac{\Gamma \overset{expr}{\vdash} e_2 : \sigma^a \qquad \Gamma \overset{expr}{\vdash} e_1 : \sigma^a \to \sigma}{\Gamma \overset{expr}{\vdash} e_1\ e_2 : \sigma}\ \text{(e-app1)}$$

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 $\sigma^a \to \sigma$.

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

$$\frac{\Gamma \overset{expr}{\vdash} e_2 : \sigma^a \qquad \Gamma \overset{expr}{\vdash} e_1 : \sigma^a \to \sigma}{\Gamma \overset{expr}{\vdash} e_1\ e_2 : \sigma}\ \text{(e-app1)} \qquad \frac{[i \mapsto \sigma^i] +\!\!+ \Gamma \overset{expr}{\vdash} e : \sigma^e}{\Gamma \overset{expr}{\vdash} \lambda i \to e : \sigma^i \to \sigma^e}\ \text{(e-lam1)}$$

$$\frac{\Gamma \overset{expr}{\vdash} e_2 : \sigma_2 \qquad \Gamma \overset{expr}{\vdash} e_1 : \sigma_1}{\Gamma \overset{expr}{\vdash} (e_1, e_2) : (\sigma_1, \sigma_2)}\ \text{(e-prod1)} \qquad \frac{[i \mapsto \sigma^i] +\!\!+ \Gamma \overset{expr}{\vdash} e^i : \sigma^i \qquad [i \mapsto \sigma^i] +\!\!+ \Gamma \overset{expr}{\vdash} e : \sigma^e}{\Gamma \overset{expr}{\vdash} \mathbf{let}\ i :: \sigma^i; i = e^i \mathbf{in}\ e : \sigma^e}\ \text{(e-let1)}$$

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

Figure 4: Type checking for expression

All rules we will use are of the form

$$\frac{\begin{array}{c} prerequisite_1 \\ prerequisite_2 \\ ... \end{array}}{consequence}\ \text{(rule-name)}$$

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 called a *judgement*:

$$context \overset{judgetype}{\vdash} construct : property \leadsto more\ results$$

This reads as

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

Although the rule formally are to be interpreted purely equationally 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

$$
\begin{aligned}
&judgetype = \lambda construct \rightarrow \\
&\qquad\qquad \lambda context \rightarrow ...(property, more\_results)
\end{aligned}
$$

or a piece of AG

```
ATTR judgetype [context : ... ||
                    property : ...more_results : ...]
SEM judgetype
  | construct
      lhs.(property, more_results) = ... @lhs.context ...
```

Typing rules and these templates for implementation however differ in that an implementation prescribes the order in which the computation for a property takes place, whereas a typing rule simply postulates relationships between parts of a rule without specifiying how this should be enforced or computed. If it benefits clarity of explanation, typing rules presented throughout this paper will be more explicit in the flow of information so as to be closer to the corresponding implementation.

2.5.2 Environment

The rules in figure 4 also refer to Γ, often called *assumptions*, the *environment* or a *context* because it provides information about what may be assumed about, say, identifiers. Identifiers are distinguished on the case of the first character, capitalized I's starting with an uppercase, uncapitalized i's otherwise

$$
\begin{aligned}
\xi &= i \\
&\mid I
\end{aligned}
$$

Type constants use capitalized identifiers I for their names, whereas for identifiers bound to an expression in a **let**-expression we will use lower case identifiers such as i.

An environment Γ is a set of bindings notated as a list

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

Because we assume the reader to be familiar with Haskell we will also use Haskell operators like concatenation $+\!\!+$ on lists in our type rules. We also make use of other behavior of lists, for example when looking up something in a Γ. If two definitions for one identifier are present the first one will be taken, therefore effectively shadowing the second. Only when the list notation becomes cumbersome a vector notation $\bar{\cdot}$ will be used instead.

A list structure is enough to encode the presence of an identifier in a Γ, but it cannot be used to detect multiple occurrences caused by duplicate introductions. In the AG implementation a stack of lists is used instead

EHCommon

```
type AssocL k v = [(k, v)]
```

EHGam

```
newtype Gam k v = Gam [AssocL k v]
emptyGam     :: Gam k v
gamLookup    :: Eq k ⇒ k → Gam k v → Maybe v
gamPushNew   :: Gam k v  → Gam k v
gamPushGam   :: Gam k v  → Gam k v → Gam k v
gamAdd       :: k → v    → Gam k v → Gam k v
```

EHGam

```
emptyGam                  = Gam [[ ]]
gamLookup       k (Gam ll) = foldr (λl mv → maybe mv Just (lookup k l)) Nothing ll
gamPushNew        (Gam ll) = Gam ([ ] : ll)
gamPushGam g1 (Gam ll2)    = Gam (gamToAssocL g1 : ll2)
gamAdd          k v        = gamAddGam (k ↦ v)
```

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

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

EHGam

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

EHGam

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

Later on the variant *valGamLookup* will do some additional work, for now it does not differ from *gamLookup*.

2.5.3 Checking Expr

The rules in figure 4 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 indeed correctly describes the value 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 inference*. Here we deal with type checking, the next version of EH (section 3) deals with type inference.

$$\boxed{\overset{fit}{\vdash} \sigma^l \leqslant \sigma^r : \sigma}$$

$$\frac{\begin{array}{l}\overset{fit}{\vdash} \sigma_2^a \leqslant \sigma_1^a : \sigma^a \\ \overset{fit}{\vdash} \sigma_1^r \leqslant \sigma_2^r : \sigma^r\end{array}}{\overset{fit}{\vdash} \sigma_1^a \to \sigma_1^r \leqslant \sigma_2^a \to \sigma_2^r : \sigma^a \to \sigma^r} \quad \text{(f-arrow1)} \qquad \frac{\begin{array}{l}\overset{fit}{\vdash} \sigma_1^l \leqslant \sigma_2^l : \sigma^l \\ \overset{fit}{\vdash} \sigma_1^r \leqslant \sigma_2^r : \sigma^r\end{array}}{\overset{fit}{\vdash} (\sigma_1^l, \sigma_1^r) \leqslant (\sigma_2^l, \sigma_2^r) : (\sigma^l, \sigma^r)} \quad \text{(f-prod1)}$$

$$\frac{I_1 \equiv I_2}{\overset{fit}{\vdash} I_1 \leqslant I_2 : I_2} \quad \text{(f-con1)}$$

$$\frac{}{\overset{fit}{\vdash} \text{⌶} \leqslant \sigma_2 : \sigma_2} \quad \text{(f-any-l1)} \qquad \frac{}{\overset{fit}{\vdash} \sigma_1 \leqslant \text{⌶} : \sigma_1} \quad \text{(f-any-r1)}$$

Figure 5: Rules for fit

$$\boxed{\Gamma, \sigma^k \overset{expr}{\vdash} e : \sigma}$$

$$\frac{\begin{array}{c}\Gamma, \sigma^a \overset{expr}{\vdash} e_2 : \_ \\ \Gamma, \bot \to \sigma^k \overset{expr}{\vdash} e_1 : \sigma^a \to \sigma\end{array}}{\Gamma, \sigma^k \overset{expr}{\vdash} e_1\ e_2 : \sigma} \ \text{(e-app1B)} \qquad \frac{[i \mapsto \sigma^i] +\!\!+ \Gamma, \sigma^r \overset{expr}{\vdash} e : \sigma^e}{\Gamma, \sigma^i \to \sigma^r \overset{expr}{\vdash} \lambda i \to e : \sigma^i \to \sigma^e} \ \text{(e-lam1B)}$$

$$\frac{\begin{array}{c}\Gamma, \sigma^k_2 \overset{expr}{\vdash} e_2 : \sigma_2 \\ \Gamma, \sigma^k_1 \overset{expr}{\vdash} e_1 : \sigma_1\end{array}}{\Gamma, (\sigma^k_1, \sigma^k_2) \overset{expr}{\vdash} (e_1, e_2) : (\sigma_1, \sigma_2)} \ \text{(e-prod1B)} \qquad \frac{\begin{array}{c}[i \mapsto \sigma^i] +\!\!+ \Gamma, \sigma^i \overset{expr}{\vdash} e^i : \_ \\ [i \mapsto \sigma^i] +\!\!+ \Gamma, \sigma^k \overset{expr}{\vdash} e : \sigma^e\end{array}}{\Gamma, \sigma^k \overset{expr}{\vdash} \mathbf{let}\ i :: \sigma^i ; i = e^i \mathbf{in}\ e : \sigma^e} \ \text{(e-let1B)}$$

$$\frac{\begin{array}{c}(i \mapsto \sigma^i) \in \Gamma \\ \overset{fit}{\vdash} \sigma^i \leqslant \sigma^k : \sigma\end{array}}{\Gamma, \sigma^k \overset{expr}{\vdash} i : \sigma} \ \text{(e-ident1B)} \qquad \frac{\overset{fit}{\vdash} Int \leqslant \sigma^k : \sigma}{\Gamma, \sigma^k \overset{expr}{\vdash} minint \mathinner{..} maxint : \sigma} \ \text{(e-int1B)}$$

Figure 6: Type checking for expression (checking variant)

We now can tailor the type rules in figure 4 towards an implementation which performs type checking, in figure 6. The rules now take an additional context, the expected (or known) type σ^k (attribute *knTy*) as specified by the programmer. An additional type of judgement *fit* (figure 5) is needed to check an actual type against a known type. For example, the rule e-int1B checks that its actual *Int* type matches the known type σ^k. Both the definition of σ^k in terms of AG

EHInferExpr

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

initialized on top by

EHInferExpr

```
    SEM AGItf
      | AGItf expr.knTy = Ty_Any
```

and the implementation of the type rule e-int1B

EHInferExpr

```
    ATTR Expr [|| ty : Ty]
    SEM Expr
      | CConst loc.fTy = tyChar
      | IConst  loc.fTy = tyInt
      | IConst CConst
                 loc.fo   = @fTy ≤ @lhs.knTy
                    .ty   = foTy @fo
```

correspond one-to-one to their formal counterpart together with definitions for type constants

EHTy

```
    tyInt    = Ty_Con hsnInt
    tyChar = Ty_Con hsnChar
```

The local attribute *fTy* (by convention) contains the type as computed on the basis of the abstract syntax tree. This type *fTy* is then compared to the expected type **lhs**.*knTy* via the implementation *fitsIn* of the rules for *fit*/$\leqslant$. In infix notation *fitsIn* prints as $\leqslant$. *fitsIn* returns a *FIOut* (**f**its**I**n **out**put) datastructure from which the resulting type can be retrieved by using the accessor function *foTy*. The split into separate attributes has been made in this way so later redefinitions of *fTy* can be made independently of the definition for *fo*.

The Haskell counterpart of $\overset{fit}{\vdash} \sigma_1 \leqslant \sigma_2 : \sigma$ is implemented by *fitsIn*. The function *fitsIn* checks if a value of type σ_1 can flow (that is, 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 around, so $\leqslant$, or *fitsIn* conceptually has a direction, even though the following implementation in essence is a test on equality of types.

The rule f-arrow1 in figure 5 for comparing function types compares the types for arguments in the other direction relative to the function type itself. Only in section 5.2, page 57 when $\leqslant$ really behaves asymmetrically we will discuss this aspect of the rules which is named *contravariance*. In the rules in figure 5 the direction makes no difference; the correct use of the direction for now only anticipates issues yet to come.

EHTyFitsIn

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

EHTyFitsIn

```
foHasErrs :: FIOut → Bool
foHasErrs = ¬.null.foErrL
```

EHTyFitsIn

```
fitsIn :: Ty → Ty → FIOut
fitsIn ty1 ty2
      =   u ty1 ty2
  where
    res t                 = emptyFO{foTy = t}
    u Ty_Any t2           = res t2
    u t1          Ty_Any  = res t1
    u t1 @(Ty_Con s1)
      t2 @(Ty_Con s2)
         | s1 ≡ s2        = res t2
```

EHTyFitsIn

```
    u 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)
    u t1 @(Ty_App tf1 ta1)
      t2 @(Ty_App tf2 ta2)
         = comp tf1 ta1 tf2 ta2 Ty_App
    u 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  = u tf1 tf2
              afo  = u ta1 ta2
              rt   = mkComp (foTy ffo) (foTy afo)
```

The function *fitsIn* checks if the *Ty_App* structure and all type constants *Ty_Con* are equal. If not, a non-empty list of errors will be returned as well as type *Ty_Any* (*Any*/⌶).

The constant *Ty_Any* denoted by *Any*/⌶ plays two roles

- *Ty_Any* plays the role of $\bot$ when it occurs at the lhs of ⌶ $\leqslant \sigma$ because it is returned as the result of an erroneous condition signaled by a previous *fitsIn*. It represents a value which never can or may be used. In that case it does not matter how it will be used because an error already has occurred.

- *Ty_Any* plays the role of $\top$ when it occurs at the rhs of $\sigma \leqslant \bot$. This happens when nothing is known about the type of the memory location a value will be put in. In that case we conclude that that location has type σ. This is the case for rule `e-app1` where we cannot determine what the expected type for the argument of the function in the application is.

Ty_Any represents "don't care" and "don't know" respectively. Even if a correspondence with $\top$ and $\bot$ from type theory is outlined, one should be aware that *Ty_Any* is used to smoothen the type checking, and is not meant to be a (new) type theoretic value.

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 *Any* back together with an error for later processing (in section 2.6). Errors themselves are also described via AG

EHErrorAbsSyn

```
DATA Err
   | UnifyClash        ty1       : Ty ty2       : Ty
                       ty1detail : Ty ty2detail : Ty
   | NamesNotIntrod nmL       : {[HsName]}
```

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

EHInferExpr

```
SEM Expr
   | Var loc.(gTy, nmErrs)
                   = case valGamLookup @nm @lhs.valGam of
                         Nothing → (Ty_Any
                                   , [Err_NamesNotIntrod [@nm]])
                         Just vgi  → (vgiTy vgi, [ ])
             .fTy = @gTy
             .fo  = @fTy ⩽ @lhs.knTy
             .ty  = foTy @fo
```

Again, the error condition is signalled by a non empty list of errors if a lookup in the Γ fails.

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 abstrcat syntax tree. The update with bindings for identifiers will be dealt with later.

EHInfer

```
ATTR AllDecl AllExpr [valGam : ValGam || ]
SEM AGItf
   | AGItf expr.valGam = emptyGam
```

One may wonder why the judgement $\overset{fit}{\vdash} \sigma_1 \leqslant \sigma_2 : \sigma$ and its implementation *fitsIn* do return a type at all. For 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. However, this idea breaks when we try to check the expression *id* 3 in

```
let id :: Int → Int
    id = λx → x
in  id 3
```

Application *App*. 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) σ^k (or *knTy*), but will be the argument of the (bottom-to-top) resulting type of e_1 (or *@func.ty*)

EHInferExpr

```
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 as $\mathtt{I} \rightarrow \sigma^k$ (passed to *func.knTy*) the partially known function type and let *fitsIn* fill in the missing details, that is find a type for $\mathtt{I}$. From that result the known/expected type of the argument can be extracted. For this to work, some additional convenience functions for constructing a type and deconstructing it are required.

EHTy

```
algTy :: MkConAppAlg Ty
algTy = (Ty_Con, Ty_App, id, id)

mkTyArrow :: TyL → Ty → Ty
mkTyArrow = let mk = mkArrow algTy in flip (foldr mk)

mkTyApp :: TyL → Ty
mkTyApp = mkApp algTy

mkTyConApp :: HsName → TyL → Ty
mkTyConApp = mkConApp algTy
```

EHTy

```
mkTyProdApp :: TyL → Ty
mkTyProdApp = mkProdApp algTy
```

EHTy

```
tyArrowArgRes  :: Ty → (Ty, Ty)
tyArrowArgsRes :: Ty → (TyL, Ty)
tyAppFunArgs   :: Ty → (Ty, TyL)
tyProdArgs     :: Ty → TyL
```

EHTy

```
tyArrowArgRes t
  = case t of
      (Ty_App (Ty_App (Ty_Con nm) a) r)
        | hsnIsArrow nm → (a, r)
      _                 → (Ty_Any, t)
tyArrowArgsRes t
  = case t of
      (Ty_App (Ty_App (Ty_Con nm) a) r)
        | hsnIsArrow nm → let (as, r') = tyArrowArgsRes r in (a : as, r')
      _                 → ([], t)
tyAppFunArgs
  = extr []
  where extr as t
          = case t of
              Ty_App f a → extr (a : as) f
              _          → (t, as)
```

EHTy

The algebra based *mkArrow* used for building abstract syntax trees during parsing is now reused to build type structures.

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 $(,)$.

EHInferExpr

```
SEM Expr
  | Con loc.ty = let resTy = snd (tyArrowArgsRes @lhs.knTy)
                 in  tyProdArgs resTy `mkTyArrow` resTy
```

This type can be constructed from *knTy* which by definition has the form $⌶ \to ⌶ \to (a, b)$ (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.

λ-expression *Lam***.** Finally, for rule e-lam1B the check if *knTy* has the form $\sigma \to \sigma$ is done by letting *fitsIn* match the *knTy* with $⌶ \to ⌶$. The result (forced to be a function type) is split up by *tyArrowArgRes*.

EHInferExpr

```
SEM Expr
  | Lam loc.funTy                = [Ty_Any] `mkTyArrow` Ty_Any
            .foKnFun             = @funTy ≼ @lhs.knTy
        (arg.knTy, body.knTy)    = tyArrowArgRes (foTy @foKnFun)
        arg.valGam               = gamPushNew @lhs.valGam
        loc.ty                   = @lhs.knTy
```

2.5.4 Checking PatExpr

Before we can look into more detail at the way new identifiers are introduced in **let**- and λ-expressions we have to look at patterns. The rule e-let1B is too restrictive for the actual language construct supported by EH. 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 figure 7 together with the rules for patterns from figure 8 reflect the desired behaviour. These rules differ from those in figure 6 in that a pattern instead of a single identifier is allowed in a value definition and argument of a λ-expression.

$$\boxed{\Gamma, \sigma^k \overset{expr}{\vdash} e : \sigma}$$

$$\frac{\begin{array}{c}\Gamma^p +\!\!+\, \Gamma, \sigma^i \overset{expr}{\vdash} e^i : \_ \\ \Gamma^p +\!\!+\, \Gamma, \sigma^k \overset{expr}{\vdash} e : \sigma^e \\ \sigma^i \overset{pat}{\vdash} p : \Gamma^p \\ p \equiv i \vee p \equiv i\ @ ...\end{array}}{\Gamma, \sigma^k \overset{expr}{\vdash} \mathbf{let}\ i :: \sigma^i ; p = e^i \mathbf{in}\ e : \sigma^e} \ (\text{e-let1C}) \qquad \frac{\begin{array}{c}\Gamma^p +\!\!+\, \Gamma, \sigma^r \overset{expr}{\vdash} e : \sigma^e \\ \sigma^p \overset{pat}{\vdash} p : \Gamma^p\end{array}}{\Gamma, \sigma^p \to \sigma^r \overset{expr}{\vdash} \lambda p \to e : \sigma^p \to \sigma^e} \ (\text{e-lam1C})$$

Figure 7: Type checking for let-expression with pattern

Again the idea is to distribute a known type over the pattern and dissect it into its constituents. However, patterns do not return a type but type bindings for the identifiers inside a pattern instead. The bindings are subsequently used in **let**- and λ-expressions.

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.

EHInferPatExpr

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

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

$$\frac{}{\sigma^k \overset{pat}{\vdash} i : [i \mapsto \sigma^k]}\ (\text{p-var1}) \qquad \frac{\begin{array}{c} dom\ (\Gamma^p_1) \cap dom\ (\Gamma^p_2) = \emptyset \\ \sigma^k_2 \overset{pat}{\vdash} p_2 : \Gamma^p_2 \\ \sigma^k_1 \overset{pat}{\vdash} p_1 : \Gamma^p_1 \end{array}}{(\sigma^k_1, \sigma^k_2) \overset{pat}{\vdash} (p_1, p_2) : \Gamma^p_1 +\!\!+ \Gamma^p_2}\ (\text{p-prod1})$$

Figure 8: Building environments from patterns

```
SEM PatExpr
  | AppTop loc.knProdTy
                   = @lhs.knTy
               .(knTyL, aErrs)
                   = case tyProdArgs @knProdTy of
                       tL | @patExpr.arity ≡ length tL
                          → (reverse tL, [ ])
                       tL → (reverse.take @patExpr.arity
                                .(++repeat Ty_Any) $ tL
                             , [Err_PatArity
                                @knProdTy @patExpr.arity])
  | App      loc.(knArgTy, knTyL)
                   = hdAndTl @lhs.knTyL
             arg .knTy = @knArgTy
```

EHInferPatExpr

```
SEM Decl
  | Val  patExpr.knTyL = [ ]
SEM Expr
  | Lam arg      .knTyL = [ ]
```

For this version of EH the only pattern is a tuple pattern. Therefore, we can use this (current) limitation to unpack a known product type *knTy* into its elements distributed over its arguments via *knTyL*. The additional complexity 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 $\top$'s as known type as necessary, determined by the actual arity of the application.

EHInferPatExpr

```
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 *knTy* holds the type of the variable name introduced. The type is added to the *valGam* threaded through the pattern, gathering all introduced bindings.

EHInferPatExpr

```
ATTR PatExpr [| valGam : ValGam |]
SEM PatExpr
  | Var VarAs loc.ty         = @lhs.knTy
              .addToGam = if @lhs.inclVarBind ∧ @nm ≢ hsnWild
```

```
                                   then gamAdd @nm
                                            (ValGamInfo @ty)
                                   else id
      | Var       lhs.valGam    = @addToGam @lhs.valGam
      | VarAs     lhs.valGam    = @addToGam @patExpr.valGam
```

2.5.5 Checking declarations

In a **let**-expression type signatures, patterns and expressions do meet. Rule e-let1C from figure 7 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. It does not work

- For mutually recursive value definitions.

  ```
  let f :: ...
      f = λx → ...g ...
      g :: ...
      g = λx → ...f ...
  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 this would make *f* and *g* together a binding group.

- For textually separated signatures and value definitions.

  ```
  let f :: ...
      ...
      f = λx → ...
  in  ...
  ```

  Syntactically the signature and value definition for an identifier need not be defined adjacently or in any specific order.

In Haskell dependency analysis determines that *f* and *g* form a so-called *binding group*, which contains declarations that have to be subjected to type analysis together. However, due to the presence of the type signatures it is possible to gather all signatures first and only then type check the value definitions. Therefore, for this version of EH it is not really an issue as we always require a signature to be defined. For later versions of EH it actually will become an issue, so for simplicity all bindings in a **let**-expression are analysed together as a single (binding) group.

Though only stating something about one combination of type signature and value definition, rule e-let1C still describes the basic strategy. First extract all type signatures, then distribute those signatures over patterns followed by expressions. The difference lies in doing it simultaneously for all declarations in a **let**-expression. So, first all signatures are collected: TBD: AG pattern: walk over tree, put result back into tree TBD: AG pattern: gather, walk, update (via gamAdd)

EHInfer

```
ATTR AllDecl [tySigGam : ValGam | gathTySigGam : ValGam ||]
SEM Decl
  | TySig lhs  .gathTySigGam = gamAdd
                                  @nm (ValGamInfo @sigTy)
                                  @lhs.gathTySigGam
SEM Expr
  | Let     decls.gathTySigGam = emptyGam
```

```
              .tySigGam      = @decls.gathTySigGam
```

EHInfer

```
  SEM Decl
    | Val loc.(sigTy, hasTySig) = case @patExpr.mbTopNm of
                                    Nothing → (Ty_Any, False)
                                    Just nm → case gamLookup nm @lhs.tySigGam of
                                                Nothing → (Ty_Any, False)
                                                Just vgi → (vgiTy vgi, True)
```

EHInfer

```
  SEM Decl
    | TySig loc.sigTy = @tyExpr.ty
```

Attribute *gathTySigGam* is used to gather type signatures. Attribute *tySigGam* is used to distribute the gathered type signatures over the declarations. In a value declaration we check if a pattern has a type signature which is to be used as the known type of the pattern and the expression.

EHInfer

```
  SEM Decl
    | Val loc.knTy = @sigTy
```

If the pattern has an identifier at the top level (pattern (a, b) has not, *ab* @(a, b) has) and a signature for this top level identifier everything is in order, indicated by a *True* value in *hasTySig*.

EHInfer

```
  ATTR PatExpr [|| mbTopNm : {Maybe HsName}]
  SEM PatExpr
    | Var VarAs loc.mbTopNm = if @nm ≡ hsnWild then Nothing else Just @nm
    | Con CConst IConst App AppTop
                loc.mbTopNm = Nothing
```

The value of *hasTySig* is also used to decide to what the top level identifier of a pattern should be bound, via *inclVarBind* used at page 28.

EHInfer

```
  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 via an addition of a binding to *valGam* (see).

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.

EHInfer

```
  ATTR AllDecl [| patValGam : ValGam |]
  SEM Decl
    | Val patExpr.valGam    = @lhs.patValGam
          lhs    .patValGam = @patExpr.valGam
          expr   .valGam    = @lhs.valGam
```

EHInfer

```
  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. Note that this implicitly makes this a three-pass algorithm over declarations.

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

EHGam

```
gamUnit     :: k → v        → Gam k v
gamPop      :: Gam k v      → (Gam k v, Gam k v)
gamToAssocL :: Gam k v      → AssocL k v
assocLToGam :: AssocL k v   → Gam k v
gamAddGam   :: Gam k v      → Gam k v → Gam k v
```

EHGam

```
gamUnit     k v                 = Gam [[(k, v)]]
gamPop      (Gam (l : ll))      = (Gam [l], Gam ll)
gamToAssocL (Gam ll)            = concat ll
assocLToGam l                   = Gam [l]
gamAddGam   g1 (Gam (l2 : ll2)) = Gam ((gamToAssocL g1 ++ l2) : ll2)
```

Extracting the top of the stack *patValGam* gives all the locally introduced bindings in *lValGam*. An additional error message is produced (later on, section 2.6) if any duplicate bindings are present in *lValGam*.

2.5.6 Checking TyExpr

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

EHInferTyExpr

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

ATTR TyExpr [|| ty : Ty]
SEM TyExpr
  | Con lhs.ty = Ty_Con @nm
  | App lhs.ty = Ty_App @func.ty @arg.ty
```

2.6 Reporting program errors

Errors are defined by AG too (see also page 25).

EHErrorAbsSyn

```
DATA Err
  | NamesDupIntrod nmL : {[HsName]}
  | PatArity       ty   : Ty          arity : {Int}
  | NestedIn       wher : {PP_Doc}    errL  : ErrL
TYPE ErrL = [Err]
```

Errors are gathered and at convenient places grouped made part of the pretty printing *pp*.

EHGatherError

```
ATTR AllNT [|| errL USE{++}{[]} : ErrL]
SEM Expr
  | App    lhs.errL = mkNestErr @pp
                        (@func.errL ++ @arg.errL)
  | Lam    lhs.errL = mkNestErr @pp
```

```
                   (foErrL @foKnFun ++ @arg.errL
                     ++ @body.errL)
  | Var       lhs.errL = mkNestErr @pp
                   (@nmErrs ++ foErrL @fo)
  | AppTop lhs.errL = mkNestErr @pp (@expr.errL)
  | IConst CConst
              lhs.errL = mkNestErr @pp (foErrL @fo)
```

EHGatherError

```
SEM TyExpr
  | Con       lhs.errL = mkNestErr @pp (@nmErrs)
SEM PatExpr
  | AppTop lhs.errL = mkNestErr @pp (@patExpr.errL ++ @aErrs)
SEM Decl
  | Val       lhs.errL = @patExpr.errL ++ @expr.errL ++ @sigMissErrs
```

EHError

```
mkNestErr :: PP_Doc → ErrL → ErrL
mkNestErr wher errL
  = if null errL then [ ] else [Err_NestedIn wher errL]
```

The pretty print of collected errors is made available for inclusion in the pretty printing of the abstract syntax tree (see page 14) We have chosen to only report the collected errors at the end of a group of declaration or at a specific declaration. Many different choices are possible here, depending on the output medium.

EHGatherError

```
SEM Decls
  | Cons loc.errPP      = ppErrs @hd.errL
         lhs.errL       = [ ]
SEM Expr
  | Let  loc.errLetPP   = ppErrs @dupErrs
            .errBodyPP  = ppErrs @body.errL
         lhs.errL       = [ ]
```

Errors themselves also need to be pretty printed, but we have omitted this part. The issue here is to keep the generated output reasonably compact because the nesting can give a long trace of locations where the error did occur.

2.7 Tying it all together

Finally, all needs to be tied together to obtain a working program. This involves a bit of glue code to (for example) combine scanner (not discussed here), parser, semantics, passing compiler options and the generation of output. Though not always trivial at times it mainly is non-trivial because glueing the pieces can mean that many missing details have to be specified. For brevity, these details are omitted.

2.8 Literature

TBD:

Local type inference [?] also has top-down, bottom-up mixing.

3 EH 2: monomorphic type inference

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

$\textbf{let}\ i = 5$
$\textbf{in}\ \ i$

gives

```
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 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.

So

$\textbf{let}\ id = \lambda x \rightarrow x$
$\textbf{in}\ \ \textbf{let}\ v = id\ 3$
$\quad\ \ \textbf{in}\ \ id$

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

```
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
```

This may look a bit strange for a Haskell programmer, but we will concern ourselves with polymorphism no sooner then with the next version of EH (section 4).

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 = λx → x
in  let f  :: (Int → Int) → ...
        f  = λi → λv → 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 explicity type signature:

```
let id :: ... → ...
    id = λx → x
in  let f  :: (Int → Int) → ...
        f  = λi → λv → i v
        v  = f id 3
    in  let v = f id 3
        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.

3.1 Type variables

In order to be able to represent yet unknown types the type language needs *type variable*s to represent this.

$$
\begin{aligned}
\sigma = &\ Int \mid Char \\
\mid &\ (\sigma, ..., \sigma) \\
\mid &\ \sigma \to \sigma \\
\mid &\ v
\end{aligned}
$$

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

EHTyAbsSyn

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

The AG system allows us to separately describe the extension with a new variant as well as describe separately the additionaly required attribution, for example the pretty printing of the type

EHTyPretty

```
SEM Ty
  | Var loc.pp = pp ("v_" ++ show @tv)
```

```
SEM Ty
  | Var lhs.appArgPPL = [ ]
          .appFunPP  = @pp
```

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

EHCommon

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

EHCommon

```
type UIDL = [UID]
instance Show UID where
  show (UID (l : ls)) = foldl (λls l → show l ++ "_" ++ ls) (show l) ls
```

EHTy

```
type TyVarId = UID
```

EHTy

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

Generation of unique identifiers is not explained here, it can be found in the source code itself. For now we will ignore this aspect and just assume a unique *UID* can be obtained.

3.2 Constraints

Although the typing rules at figure 7, page 27 still hold we need to look at the meaning of $\leqslant$ (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, if the check $v \leqslant \sigma$ is encountered making the previously unknown type v equal to σ is the easiest way to make $v \leqslant \sigma$ true. An alternative way to look at this is that $v \leqslant \sigma$ is true under the constraint that v equals σ.

3.2.1 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.

$$\mathcal{C} = [v \mapsto \sigma]$$

A set of *constraints* $\mathcal{C}$ is a set of bindings for type variables, represented as

EHCnstr

```
newtype C = C (AssocL TyVarId Ty) deriving Show
cnstrLookup :: TyVarId → C → Maybe Ty
cnstrLookup tv (C s) = lookup tv s
```

EHCnstr

```
emptyCnstr :: C
emptyCnstr = C [ ]
cnstrUnit :: TyVarId → Ty → C
cnstrUnit tv t = C [(tv, t)]
```

If *cnstrUnit* 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 [**?**, **?**]. Here constraints $\mathcal{C}$ are used to replace all other references to v by σ, for this reason often named a *substitution*. Mostly this will be 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

EHCnstr

```
infixr 6 ≻
class Substitutable s where
```

```
(≻) :: C → s → s
ftv  :: s → TyVarIdL
```

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

A $\mathcal{C}$ can be applied to a type

EHCnstr

```
instance Substitutable Ty where
  s ≻ ty = st ty
    where st t = case t of
                   Ty_App fn arg → Ty_App (st fn) (st arg)
                   Ty_Var v      → maybe t id (cnstrLookup v s)
                   otherwise     → t
  ftv t = case t of
            Ty_App fn arg       → ftv fn ∪ ftv arg
            Ty_Var tv           → [tv]
            otherwise           → [ ]
```

and is lifted straightforwardly to lists

EHCnstr

```
instance Substitutable a ⇒ Substitutable [a] where
  s ≻ l = map (s≻) l
  ftv l = unionL.map ftv $ l
```

A $\mathcal{C}$ can also be applied to another $\mathcal{C}$

EHCnstr

```
instance Substitutable C where
  s1 @(C sl₁) ≻ s2 @(C sl₂) = C (sl₁ ++ map (λ(v,t) → (v,s1 ≻ t)) sl'₂)
                               where sl'₂ = deleteFirstsBy (λ(v1,_) (v2,_) → v1 ≡ v2) sl₂ sl₁
  ftv            (C sl)      = ftv.map snd $ sl
```

but one must be aware that $\succ$ 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.

3.2.2 Computing constraints

The only source of constraints is the check *fitsIn* which finds out if a type can flow into another. The previous version of EH had one option in case a type could not fit in another: report an error. Now, if one of the types is unknown, that is, a type variable we have the additional option to return a constraint on that type variable. The implementation *fitsIn* of $\leqslant$ has to be rewritten to include additional cases for type variables and the return of constraints

EHTyFitsIn

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

EHTyFitsIn

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

EHTyFitsIn

```
fitsIn :: Ty → Ty → FIOut
fitsIn ty₁ ty₂
      =    u ty₁ ty₂
  where
    res t                          = emptyFO{foTy = t}
```

EHTyFitsIn

```
bind tv t                       = (res t){foCnstr = tv ↦ t}
occurBind v t | v ∈ ftv t       = err [Err_UnifyOccurs ty₁ ty₂ v t]
              | otherwise       = bind v t
```

EHTyFitsIn

```
comp tf₁ ta₁ tf₂ ta₂ mkComp
    = foldr1 (λfo₁ fo₂ → if foHasErrs fo₁ then fo₁ else fo₂)
             [ffo, afo, rfo]
    where ffo = u tf₁ tf₂
          fs  = foCnstr ffo
          afo = u (fs ≻ ta₁) (fs ≻ ta₂)
          as  = foCnstr afo
          rt  = mkComp (as ≻ foTy ffo) (foTy afo)
          rfo = emptyFO{foTy = rt, foCnstr = as ≻ fs}
```

EHTyFitsIn

```
u Ty_Any        t₂              = res t₂
u t₁            Ty_Any          = res t₁
u t₁ @(Ty_Con s1)
  t₂ @(Ty_Con s2)
    | s1 ≡ s2                   = res t₂
```

EHTyFitsIn

```
u t₁ @(Ty_Var v1) (Ty_Var v2)
    | v1 ≡ v2                   = res t₁
u t₁ @(Ty_Var v1) t₂            = occurBind v1 t₂
u t₁            t₂ @(Ty_Var v2) = occurBind v2 t₁
```

EHTyFitsIn

```
u 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)
u t₁ @(Ty_App tf₁ ta₁)
  t₂ @(Ty_App tf₂ ta₂)
    = comp tf₁ ta₁ tf₂ ta₂ Ty_App
u t₁            t₂              = err [Err_UnifyClash ty₁ ty₂ t₁ t₂]
err e                           = emptyFO{foErrL = e}
```

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

- The datatype *FIOut* returned by has an additional field *foCnstr* holding found constraints.
- 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 type bound to. This is to prevent (e.g.) $a \leqslant a \to a$ to succeed. This is because it is not clear if $a \mapsto a \to a$ should be the resulting constraint or $a \mapsto (a \to a) \to (a \to a)$ or one of infinitely many other possible solutions. A so called *infinite type* like this is inhibited by the so called *occur check*.
- An application *App* recursively fits its components with components of another *App*. The constraints from the first fit *ffo* are applied immediately to the following component before fitting that one. This is to prevent $a \to a \leqslant Int \to Char$ 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 found knowledge about previously unknown types explicit. The typing rules in figure 4 (and figure 6, figure 7) in principle need to be changed. The only reason to adapt some of the rules to the variant in figure 9 is to clarify the way constraints are used.

$$\boxed{\Gamma, \sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow \mathcal{C}}$$

$$\frac{\begin{array}{c}\Gamma, \sigma^a \overset{expr}{\vdash} e_2 : \_ \rightsquigarrow \mathcal{C}_2 \\ \Gamma, v \rightarrow \sigma^k \overset{expr}{\vdash} e_1 : \sigma^a \rightarrow \sigma \rightsquigarrow \mathcal{C}_1 \\ v \text{ fresh}\end{array}}{\Gamma, \sigma^k \overset{expr}{\vdash} e_1\ e_2 : \mathcal{C}_2 \sigma \rightsquigarrow \mathcal{C}_{2..1}} \quad (\text{e-app2})$$

$$\frac{\begin{array}{c}\Gamma^p +\!\!+ \Gamma, \sigma^r \overset{expr}{\vdash} e : \sigma^e \rightsquigarrow \mathcal{C}_3 \\ \sigma^p \overset{pat}{\vdash} p : \_, \Gamma^p \rightsquigarrow \mathcal{C}_2 \\ \overset{fit}{\vdash} v_1 \rightarrow v_2 \leqslant \sigma^k : \sigma^p \rightarrow \sigma^r \rightsquigarrow \mathcal{C}_1 \\ v_i \text{ fresh}\end{array}}{\Gamma, \sigma^k \overset{expr}{\vdash} \lambda p \rightarrow e : \mathcal{C}_3 \sigma^p \rightarrow \sigma^e \rightsquigarrow \mathcal{C}_{3..1}} \quad (\text{e-lam2})$$

$$\frac{\begin{array}{c}(i \mapsto \sigma^i) \in \Gamma \\ \overset{fit}{\vdash} \sigma^i \leqslant \sigma^k : \sigma \rightsquigarrow \mathcal{C}\end{array}}{\Gamma, \sigma^k \overset{expr}{\vdash} i : \sigma \rightsquigarrow \mathcal{C}} \quad (\text{e-ident2})$$

$$\frac{\begin{array}{c}\overset{fit}{\vdash} (v_1, v_2, ..., v_n) \leqslant \sigma^r : (\sigma_1, \sigma_2, ..., \sigma_n) \rightsquigarrow \mathcal{C} \\ \_ \rightarrow ... \rightarrow \sigma^r \equiv \sigma^k \\ v_i \text{ fresh}\end{array}}{\Gamma, \sigma^k \overset{expr}{\vdash} \ ,\!n : \sigma_1 \rightarrow ... \rightarrow \sigma_n \rightarrow (\sigma_1, \sigma_2, ..., \sigma_n) \rightsquigarrow \mathcal{C}} \quad (\text{e-con2})$$

$$\frac{\overset{fit}{\vdash} Int \leqslant \sigma^k : \sigma \rightsquigarrow \mathcal{C}}{\Gamma, \sigma^k \overset{expr}{\vdash} minint \mathinner{..} maxint : \sigma \rightsquigarrow \mathcal{C}} \quad (\text{e-int2})$$

Figure 9: Type inferencing for expressions (using constraints)

The type rules in figure 9 enforce an order in which checking and inferring types has to be done.

TBD: ...

Actually, the rules in figure 9 are more cluttered with constraints flowing around if we want to approximate the corresponding AG description, for example for expression application

```
SEM Expr
  | App loc .knFunTy          = [Ty_Any] `mkTyArrow` @lhs.knTy
        func.knTy             = @knFunTy
        (arg.knTy, loc.fTy)   = tyArrowArgRes @func.ty
        loc .ty               = @fTy
```

EHInferExpr

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

EHInferExpr

```
SEM Expr
  | App loc.knFunTy := [mkNewTyVar @lUniq]
                         `mkTyArrow` (@lhs.tyCnstr ≻ @lhs.knTy)
           .ty      := @arg.tyCnstr ≻ @fTy
```

This definition builds on top of the previous version by redefining some attributes (indicated by ⊨ instead of =), the appearance on screen or paper should have a different color or shade of gray to indicate it is a repetition from a previous appearance elsewhere in the text.

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

$$\frac{\begin{array}{c} \mathcal{C}_1, \Gamma, \sigma^a \overset{expr}{\vdash} e_2 : \_ \leadsto \mathcal{C}_2 \\ \mathcal{C}^k, \Gamma, v \to \mathcal{C}^k\sigma^k \overset{expr}{\vdash} e_1 : \sigma^a \to \sigma \leadsto \mathcal{C}_1 \\ v \text{ fresh} \end{array}}{\mathcal{C}^k, \Gamma, \sigma^k \overset{expr}{\vdash} e_1\ e_2 : \mathcal{C}_2\sigma \leadsto \mathcal{C}_2} \qquad (\text{e-app2B})$$

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 $\leadsto$). 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 eachother.

An implementation by necessity imposes additional choices to make a typing rule into an algorithmic solution. For example, our description preserves the 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 figure 9 do not mention the last two constraint applications (rule e-app2B does), 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.(gTy, nmErrs)
                = case valGamLookup @nm @lhs.valGam of
                    Nothing → (Ty_Any
                              , [Err_NamesNotIntrod [@nm]])
                    Just vgi → (vgiTy vgi, [ ])
            .fTy = @gTy
            .fo  = @fTy ≤ @lhs.knTy
            .ty  = foTy @fo
SEM Expr
  | Var loc.fTy      := @lhs.tyCnstr ⊢ @gTy
           .fo       := @fTy ≤ (@lhs.tyCnstr ⊢ @lhs.knTy)
        lhs.tyCnstr = foCnstr @fo ⊢ @lhs.tyCnstr
```

EHInferExpr

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

```
ATTR Expr [|| ty : Ty]
SEM Expr
  | CConst loc.fTy = tyChar
  | IConst loc.fTy = tyInt
  | IConst CConst
          loc.fo  = @fTy ≤ @lhs.knTy
             .ty  = foTy @fo
```

EHInferExpr

```
SEM Expr
  | IConst CConst
    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 (figure 6) 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) forced by *fitsIn*.

```
SEM Expr
  | Con loc.ty = let resTy = snd (tyArrowArgsRes @lhs.knTy)
                 in  tyProdArgs resTy `mkTyArrow` resTy
```

EHInferExpr

```
SEM Expr
  | Con loc.fo       = let gTy      = mkTyFreshProdFrom @lUniq (hsnProdArity @nm)
                           foKnRes = gTy ⩽ (@lhs.tyCnstr ≻ snd (tyArrowArgsRes @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.funTy               = [Ty_Any] `mkTyArrow` Ty_Any
           .foKnFun             = @funTy ⩽ @lhs.knTy
        (arg.knTy, body.knTy)   = tyArrowArgRes (foTy @foKnFun)
        arg.valGam              = gamPushNew @lhs.valGam
        loc.ty                  = @lhs.knTy
```

EHInferExpr

```
SEM Expr
  | Lam loc.funTy    := let [a, r] = mkNewTyVarL 2 @lUniq
                        in  [a] `mkTyArrow` r
           .foKnFun  := @funTy ⩽ (@lhs.tyCnstr ≻ @lhs.knTy)
        arg.tyCnstr   = foCnstr @foKnFun ≻ @lhs.tyCnstr
        loc.ty       := [@body.tyCnstr ≻ @arg.ty] `mkTyArrow` @body.ty
```

which uses some additional functions for creating type variables

EHTy

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

EHTy

```
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 $\top$ 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 $\top \rightarrow \top$ to enforce a $.. \rightarrow ..$ structure. If $\top$ still would be used, for

  ```
  let id = λx → x
  in  id 3
  ```

the conclusion would be drawn that $id :: \text{⌶} \to \text{⌶}$, whereas $id :: v \to v$ would later on have bound $v \mapsto Int$ (at the application id 3). So, ⌶ represents "unknown knowledge", a type variable v represents "not yet known knowledge" to which the inferencing 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 $v \notin \mathsf{ftv}\,(\Gamma)$. 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.

$$\boxed{\sigma^k \overset{pat}{\vdash} p : \sigma, \Gamma^p \leadsto \mathcal{C}}$$

$$\frac{\begin{array}{c} \overset{fit}{\vdash} \mathcal{C}_1\sigma^k \leqslant \sigma^d : \sigma \leadsto \mathcal{C}_2 \\ \sigma^d \to () \equiv \sigma^p \\ \_ \overset{pat}{\vdash} p : \sigma^p, \Gamma^p \leadsto \mathcal{C}_1 \\ p \equiv p_1\ p_2 \ldots p_n, n \geqslant 1 \end{array}}{\sigma^k \overset{pat}{\vdash} p : \sigma, \Gamma^p \leadsto \mathcal{C}_{2..1}} \quad (\text{p-apptop2})$$

$$\frac{\begin{array}{c} dom\ (\Gamma^p_1) \cap dom\ (\Gamma^p_2) = \emptyset \\ \sigma^a_1 \overset{pat}{\vdash} p_2 : \_, \Gamma^p_2 \leadsto \mathcal{C}_2 \\ \_ \overset{pat}{\vdash} p_1 : \sigma^d \to (\sigma^a_1, \sigma^a_2, \ldots, \sigma^a_n), \Gamma^p_1 \leadsto \mathcal{C}_1 \end{array}}{\_ \overset{pat}{\vdash} p_1\ p_2 : \mathcal{C}_2(\sigma^d \to (\sigma^a_2, \ldots, \sigma^a_n)), \Gamma^p_1 +\!\!+\, \Gamma^p_2 \leadsto \mathcal{C}_{2..1}} \quad (\text{p-app2})$$

$$\frac{\sigma^k \not\equiv \text{⌶}}{\sigma^k \overset{pat}{\vdash} i : \sigma^k, [i \mapsto \sigma^k] \leadsto [\,]} \quad (\text{p-var2}) \qquad \frac{v_i \text{ fresh}}{\_ \overset{pat}{\vdash} I : \sigma, (v_1, v_2, \ldots, v_n) \to (v_1, v_2, \ldots, v_n) \leadsto [\,]} \quad (\text{p-con2})$$

Figure 10: Type inferencing for pattern (using constraints)

Computation of the type of a pattern is similar to and yet more straightforward than for expressions. The rule e-pat2 from figure 10 binds the identifier to the known type and if no such known type is available it invents a fresh one, by means of *tyEnsureNonBotTop*

EHInferPatExpr

```
ATTR PatExpr [| tyCnstr : C | ty : Ty]
SEM PatExpr
  | Var VarAs loc     .ty    := tyEnsureNonBotTop @lUniq @lhs.knTy
  | VarAs     patExpr.knTy = @ty
```

EHTy

```
tyEnsureNonBotTop :: UID → Ty → Ty
tyEnsureNonBotTop u t = if t ≢ Ty_Any then t else mkNewTyVar u
```

For tuples we again make use of the fact that the *Con* alternative will always represent a tuple. From section 6 when datatypes are introduced and onwards this will no longer be the case. Here, we already make the required rule p-con2 more general than is required here in that it will reflect the idea of a pattern. A pattern (in essence) can be represented by a function taking a value of some type and dissecting it into a tuple containing all its constituents. For now, because we have only tuples to dissect, the function returned by the *Con* alternative is just the identity on tuples of the correct size. The application rule p-app2 consumes an element of this tuple representing the dissected value and uses it for checking and inferring the constituent.

The implementation of this representation convention returns the dissecting function type in *patFunTy*

EHInferPatExpr

```
ATTR PatExpr [|| patFunTy : Ty]
SEM PatExpr
  | Con loc.patFunTy = let prTy = mkTyFreshProdFrom @lUniq (hsnProdArity @nm)
                       in  ([prTy] `mkTyArrow` prTy)
  | App lhs.patFunTy = @func.patFunTy
  | ∗ − App Con
        lhs.patFunTy = Ty_Any
```

which is at the top distributed as described for the previous version.

```
ATTR PatExpr [knTy : Ty   knTyL : TyL ||]
SEM PatExpr
  | AppTop loc.knProdTy
                      = @lhs.knTy
                 .(knTyL, aErrs)
                      = case tyProdArgs @knProdTy of
                          tL | @patExpr.arity ≡ length tL
                             → (reverse tL, [ ])
                          tL → (reverse.take @patExpr.arity
                                  .(++repeat Ty_Any) $ tL
                               , [Err_PatArity
                                  @knProdTy @patExpr.arity])
  | App      loc.(knArgTy, knTyL)
                      = hdAndTl @lhs.knTyL
             arg.knTy = @knArgTy
```

EHInferPatExpr

```
SEM PatExpr
  | AppTop loc.patFunTy              = @patExpr.patFunTy
                 .(knResTy, knProdTy) := tyArrowArgRes @patFunTy
```

Finally, the type itself and additional constraints are returned

EHInferPatExpr

```
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 $\leqslant$ 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 $\leqslant$ also.

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.

EHInfer

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

EHInfer

```
ATTR AllDecl [| tyCnstr : C    patTyCnstr : C |]
SEM Decl
  | Val    patExpr.tyCnstr      = @lhs.patTyCnstr
           lhs      .patTyCnstr = @patExpr.tyCnstr
           expr     .tyCnstr    = @lhs.tyCnstr
SEM AGItf
  | AGItf 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 such.

EHInfer

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

3.6 Partial type signatures: a test case for extendibility

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, an impact analysis.

First, both abstract syntax and the parser contain an additional alternative for parsing the "..." notation chosen for unspecified type information designated by *Wild* for wildcard:

EHAbsSyn

```
DATA TyExpr
  | Wild
```

EHParser

```
tyExprAlg   = (sem_TyExpr_Con, sem_TyExpr_App
              , sem_TyExpr_AppTop, sem_TyExpr_Parens)
pTyExprBase = sem_TyExpr_Con  <$> pCon
              <|> sem_TyExpr_Wild  <$ pKey "..."
              <|> pParenProd tyExprAlg pTyExpr
pTyExpr     = pChainr
                (mkArrow tyExprAlg <$ pKeyw hsnArrow)
                pTyExprBase
```

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

EHInferTyExpr

```
SEM TyExpr
  | Wild loc.tyVarId = @lUniq
```

```
              .tgi      = TyGamInfo (mkNewTyVar @tyVarId)
SEM TyExpr
   | Wild lhs.ty = tgiTy @tgi
```
EHInferTyExpr

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

Finally, in order to decide which type to pretty print all 'wild' type variables are gathered so we can check if any wild type variables were introduced.

EHInferTyExpr

```
SEM TyExpr [|| tyVarWildL USE{ ++ }{[]} : TyVarIdL]
   | Wild lhs.tyVarWildL = [@tyVarId]
```

Pretty printing then uses the final type which has all found constraints incorporated.

EHInfer

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

4 EH 3: polymorphic type inference

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 = λx → 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 ∀. 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.

TBD: more on this..., mention: specialization, instantiation, impredicativeness

The type signature may be omitted, 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 [**?**]. 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 = λi → i 3
    id :: a → a
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
                   problem with : c_2_0 <= Int -}
    id :: a -> a
    {- [ id:forall a . a -> a, f:forall a . (a -> a) -> Int ] -}
in f id
```

The problem here is that the polymorphism of f in a means that the caller of f can freely choose what this a is for a particular call. However, from the viewpoint of the body of f this limits the choice of a to no choice at all. If the caller has all the freedom to make the choice, the callee has none. This is encoded as a type constant `c_` chosen for a during type checking the body of f. This type constant by definition is a type a programmer can never define or denote. The consequence is that an attempt to use i in the body of f, which has type `c_..`$\rightarrow$`c_..` cannot be used with an *Int*. The use of type constants will be explained later.

Another example of the limitations of polymorphism in this version of EH is the following variation

let $f = \lambda i \rightarrow i\ 3$
   $id :: a \rightarrow a$
in let $v = f\ id$
   in f

for which the compiler will infer types

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

EH version 3 allows parametric polymorphism but not polymorphic parameters. The parameter i has a monomorphic type, which is made even more clear when we make an attempt to use this i polymorphically in

let $f = \lambda i \rightarrow (i\ 3, i\ \text{'x'})$
   $id = \lambda x \rightarrow x$
in let $v = f\ id$
   in v

for which the compiler will infer types

```
let f = \i -> (i 3,i 'x')
    {- ***ERROR(S):
         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.

However, the next version (section 5) does permit polymorphic parameters.

The "monomorphic parameter" problem could have been solved by allowing a programmer to explicitly specify a $\forall$ for the type of the parameter i of f. The type of f would then be $(\forall\ a.a \rightarrow a) \rightarrow (Int, Char)$ instead of $\forall\ a.(a \rightarrow a) \rightarrow (Int, Char)$. In this version of EH (resembling Haskell98) it is not permitted to specify polymorphism explicitly, but the next version of EH does permit this.

The reason not to allow explicit types 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 [**?**, **?**, **?**].

However, type inferencing also has its limitsTBD: [cite...]. In fact, the richer a type system becomes, the more difficulty a type inferencing algorithm has in making the right choice for a type. In the next version it is allowed to specify types which otherwise could not have been found by a type inferencing algorithm.

4.1 Type language

The type language for this version of EH adds quantification with the universal quantifier $\forall$

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

A f stands for a fixed type variable, a type variable which may not be constrained but still stands for an unknown type. A series of consecutive quantifiers in $\forall\ \alpha_1.\forall\ \alpha_2. ... \sigma$ is abbreviated to $\forall\overline{\alpha}.\sigma$.

The corresponding abstract syntax for type expressions also need an additional alternative EHTyAbsSyn

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

DATA *TyVarCateg*
  | *Plain*
  | *Fixed*

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

together with a convenience function for making such a quantified type EHTy

$$\begin{array}{l} mkTyVar :: TyVarId \rightarrow Ty \\ mkTyVar\ tv = Ty\_Var\ tv\ TyVarCateg\_Plain \end{array}$$

$$\begin{array}{l} mkTyQu :: TyVarIdL \rightarrow Ty \rightarrow Ty \\ mkTyQu\ tvL\ t = foldr\ (\lambda tv\ t \rightarrow Ty\_Quant\ tv\ t)\ t\ tvL \end{array}$$

The discussion of type variable categories is postponed until section 4.2.1.

However, the syntax of this version of EH only allows type variables to be specified as part of a type signatures. The quantifier ∀ cannot be explicitly denoted.

EHAbsSyn

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

As a consequence the parser for type expressions has to include an alternative in *pTyExprBase* to parse type variables.

EHParser

```
tyExprAlg    = (sem_TyExpr_Con, sem_TyExpr_App
               , sem_TyExpr_AppTop, sem_TyExpr_Parens)
pTyExprBase  = sem_TyExpr_Con  ⟨$⟩ pCon
               ⟨|⟩ sem_TyExpr_Var  ⟨$⟩ pVar
               ⟨|⟩ sem_TyExpr_Wild ⟨$ pKey "..."
               ⟨|⟩ pParenProd tyExprAlg pTyExpr
pTyExpr      = pChainr
                 (mkArrow tyExprAlg ⟨$ pKeyw hsnArrow)
                 pTyExprBase
```

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. A second restriction is that quantified types are present only in a Γ whereas no ∀'s are present in types used throughout type inferencing expressions and patterns. This is to guarantee the principle type property TBD: more refs [..].

4.2 Type inference

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 ∀ 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.

A quantified type, or also often named *type scheme*, is introduced in rule e-let3 and rule e-let-tysig3 and instantiated in rule e-ident3, see figure 11. We will first look at the instantiation.

4.2.1 Instantiation

EHInferExpr

```
SEM Expr
   | Var loc.fTy := @lhs.tyCnstr ≻ tyInst @lUniq @gTy
```

A quantified type is used whenever a value identifier having that type is referred to in an expression. 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 with all quantified type variables is done by *tyInst*:

EHTyInstantiate

$$\boxed{\Gamma, \sigma^k \overset{expr}{\vdash} e : \sigma \leadsto \mathcal{C}}$$

$$\frac{\begin{array}{c} \Gamma^q +\!\!+ \Gamma, \sigma^k \overset{expr}{\vdash} e : \sigma^e \leadsto \mathcal{C}_3 \\ \Gamma^q \equiv [(i \mapsto \forall \overline{\alpha}.\sigma) \mid (i \mapsto \sigma) \leftarrow \mathcal{C}_{2..1}\Gamma^p, \overline{\alpha} \equiv \mathsf{ftv}\ (\sigma) - \mathsf{ftv}\ (\mathcal{C}_{2..1}\Gamma)] \\ \Gamma^p +\!\!+ \Gamma, \sigma^p \overset{expr}{\vdash} e^i : \_ \leadsto \mathcal{C}_2 \\ \top \overset{pat}{\vdash} p : \sigma^p, \Gamma^p \leadsto \mathcal{C}_1 \end{array}}{\Gamma, \sigma^k \overset{expr}{\vdash} \mathbf{let}\ p = e^i \mathbf{in}\ e : \sigma^e \leadsto \mathcal{C}_{3..1}} \quad \text{(e-let3)}$$

$$\frac{\begin{array}{c} (\Gamma^q - [i \mapsto \_] +\!\!+ [i \mapsto \sigma^q]) +\!\!+ \Gamma, \sigma^k \overset{expr}{\vdash} e : \sigma^e \leadsto \mathcal{C}_3 \\ \Gamma^q \equiv [(i \mapsto \forall \overline{\alpha}.\sigma) \mid (i \mapsto \sigma) \leftarrow \mathcal{C}_{2..1}\Gamma^p, \overline{\alpha} \equiv \mathsf{ftv}\ (\sigma) - \mathsf{ftv}\ (\mathcal{C}_{2..1}\Gamma)] \\ (\Gamma^p - [i \mapsto \_] +\!\!+ [i \mapsto \sigma^q]) +\!\!+ \Gamma, \sigma^j \overset{expr}{\vdash} e^i : \_ \leadsto \mathcal{C}_2 \\ \sigma^q \equiv \forall \overline{\alpha}.\sigma^i \\ \sigma^j \equiv [\alpha_j \mapsto f_j]\ \sigma^i, f_j \text{ fresh} \\ \overline{\alpha} \equiv \mathsf{ftv}\ (\sigma^i) \\ p \equiv i \vee p \equiv i\ @... \\ \sigma^i \overset{pat}{\vdash} p : \_, \Gamma^p \leadsto \mathcal{C}_1 \end{array}}{\Gamma, \sigma^k \overset{expr}{\vdash} \mathbf{let}\ i :: \sigma^i ; p = e^i \mathbf{in}\ e : \sigma^e \leadsto \mathcal{C}_{3..1}} \quad \text{(e-let-tysig3)}$$

$$\frac{\begin{array}{c} (i \mapsto \forall\ [\alpha_j].\sigma^i) \in \Gamma \\ \overset{fit}{\vdash} [\alpha_j \mapsto v_j]\ \sigma^i \leqslant \sigma^k : \sigma \leadsto \mathcal{C} \\ v_j \text{ fresh} \end{array}}{\Gamma, \sigma^k \overset{expr}{\vdash} i : \sigma \leadsto \mathcal{C}} \quad \text{(e-ident3)}$$

Figure 11: Type inferencing for expressions with quantifier $\forall$

```
tyInst' :: (TyVarId → Ty) → UID → Ty → Ty
tyInst' mkFreshTy uniq ty
  = s ≻ ty'
  where i u (Ty_Quant v t) = let (u',v') = mkNewUID u
                                 (s,t') = i u' t
                             in  ((v ↦ (mkFreshTy v')) ≻ s, t')
        i _ t              = (emptyCnstr, t)
        (s, ty')           = i uniq ty
tyInst :: UID → Ty → Ty
tyInst = tyInst' mkTyVar
```

Function *tyInst* strips all quantifiers and substitutes the quantified type variables with fresh ones.

4.2.2 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 (figure 11) differ only in the use of a type signature, if present.

```
SEM Decl
  | TySig loc.sigTy = @tyExpr.ty
SEM Decl
  | TySig loc.sigTy := tyQuantify (∈ @tyExpr.tyVarWildL) @tyExpr.ty
```

EHInfer

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

EHTyQuantify

```
tyQuantify :: (TyVarId → Bool) → Ty → Ty
tyQuantify tvIsBound ty = mkTyQu (filter (¬.tvIsBound) (ftv ty)) ty
```

EHTyQuantify

```
tyQuantifyClosed :: Ty → Ty
tyQuantifyClosed = tyQuantify (const False)
```

Type variables introduced by a wildcard may not be quantified over.

We now run into a problem which will be solved no sooner than the next version of EH. The type signature acts as a known type *knTy* against which checking 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 $v_1 \rightarrow Int \leqslant a \rightarrow a$. This can be done via constraints $[v_1 \mapsto Int, a \mapsto Int]$. However, *a* was supposed to be chosen by the caller of *id* while 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 = @sigTy
```

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

by using a variant of *tyInst*

EHTyInstantiate

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

This changes the category of a 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 that 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 are equal only if their categories match too. For brevity the new version of *fitsIn* is omitted.

4.2.3 Generalization/quantification of inferred types

How do we determine if a type for some expression is polymorphic? If a 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 body 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 figure 11 and its implementation

```
SEM Expr
  | Let decls.patValGam          = @decls.gathTySigGam
                                     `gamPushGam` @lhs.valGam
        loc .(lValGam, gValGam)  = gamPop @decls.patValGam
        decls.valGam             = @decls.patValGam
        body.valGam              = @decls.patValGam
SEM Expr
  | Let decls.patTyCnstr = @lhs.tyCnstr
            .tyCnstr     = @decls.patTyCnstr
```

EHInfer

```
SEM Expr
  | Let loc .lSubsValGam  = @decls.tyCnstr ⊢ @lValGam
            .gSubsValGam  = @decls.tyCnstr ⊢ @gValGam
            .gTyTvL       = ftv @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 available more globally, the *gTyTvL*.

EHGam

$valGamQuantify :: TyVarIdL \rightarrow ValGam \rightarrow ValGam$
$valGamQuantify\ globTvL$
$\quad = gamMap\ (\lambda(n,t) \rightarrow (n, t\{vgiTy = tyQuantify\ (\in globTvL)\ (vgiTy\ t)\}))$

EHGam

$gamMap :: ((k,v) \rightarrow (k',v')) \rightarrow Gam\ k\ v \rightarrow Gam\ k'\ v'$
$gamMap\ f\ (Gam\ ll) = Gam\ (map\ (map\ f)\ ll)$

The condition that quantification only may be done for type variables not occurring in the global Γ is a necessary one. Take 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:v_14_1 -> (v_14_1,v_14_1) ] -}
              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 no 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 $v_1 \mapsto Int$, resulting in type $Int \rightarrow Int$ 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 to binding groups d_1 and d_2 has to be done explicitly by **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 f1 = λx → g1 x
    g1 = λy → f1 y
    f2 :: a → a
    f2 = λx → g2 x
    g2 = λy → f2 y
in f1 3
```

This results in

```
let f1 = \x -> g1 x
    g1 = \y -> f1 y
    f2 :: a -> a
    f2 = \x -> g2 x
    g2 = \y -> f2 y
    {- [ g2:forall a . a -> a, g1:forall a . forall b . a -> b
       , f1:forall a . forall b . a -> b, f2:forall a . a -> a ] -}
in f1 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.

TBD: example polymorphic recursion

4.2.4 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.

EHGam

```
data TyGamInfo = TyGamInfo{ tgiTy :: Ty } deriving Show
```

EHGam

```
type TyGam = Gam HsName TyGamInfo
```

EHInferTyExpr

```
SEM TyExpr
  | Var (loc.tgi, lhs.tyGam) = case tyGamLookup @nm @lhs.tyGam of
                                 Nothing → let t   = mkNewTyVar @lUniq
                                               tgi = TyGamInfo t
```

```
                                   in  (tgi, gamAdd @nm tgi @lhs.tyGam)
                         Just tgi  → (tgi, @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.

4.3 Literature

TBD:

Polymorphic recursion [?, ?, ?, ?, ?]

5 EH 4: ∀ and ∃ everywhere

This version of EH adds explicit types with quantifiers at all positions in a type, existential quantification and an interpretation of unquantified types to quantified types.

Higher ranked explicit polymorphism. For example in

```
let f  :: (a → a) → (Int, Char)
    f  = λi → (i 3, i 'x')
    id :: a → a
    v  = f id
in f
```

f has type $f :: (\forall\ a.a \rightarrow a) \rightarrow (Int, Char)$, which means that *i* has type $\forall\ a.a \rightarrow a$ in the body of *f*. Therefore *i* can be used polymorphically in the body of *f*.

The quantifier ∀ in the type of *f* is on a so called higher ranked position in the type, rank 1 being in front of the type or in front of a result of a function type. A rank 2 position is in front of the argument of a function type. Higher ranked positions are defined recursively by the same definition. So, in

$$\forall_1\ a.a \rightarrow (\forall_2\ b.b \rightarrow b) \rightarrow ((\forall_3\ c.c \rightarrow c) \rightarrow (\forall_4\ d.d \rightarrow d)) \rightarrow \forall_5\ e.e \rightarrow e \rightarrow a$$

$\forall_1$ and $\forall_5$ are on a rank 1 position, $\forall_2$ and $\forall_4$ on rank 2 and $\forall_3$ on rank 3.

Standard Hindley-Milner inferencing as described for the previous version of EH can infer rank 1 universal quantifiers. We will not do better than that, but the inferencer described for this version of EH will not throw away any type information about higher ranked quantifiers defined via a type signature. No attempt is made to be smart in reconstructing higher ranked types, only smartness is implemented by not forgetting higher ranked type information.

Existentially quantified types. Quantifiers on higher ranked positions are also necessary to make existential types useful

```
let id  :: ∀ a.a → a
    xy  :: ∃ a.(a, a → Int)
    xy  = (3, id)
    ixy :: (∃ a.(a, a → Int)) → Int
    ixy = λ(v, f) → f v
```

$$
\begin{array}{l}
\quad xy' = ixy\ xy \\
\quad pq\ \ :: \exists\, a.(a, a \rightarrow Int) \\
\quad pq\ \ = (\texttt{'x'}, id) \quad \text{-- ERROR} \\
\textbf{in}\ xy'
\end{array}
$$

An existentially quantified type is specified with keyword `exists`, or ∃. The type variable which is existentially quantified represents a type but we do not know which one. Existential quantification hides, or forgets, more specific type information. In the given example *xy* is a tuple for which we have forgotten that its first component is an *Int*. All we know is that we can apply the second component to the first component (as done by *ixy*). This constraint is not upheld for *pq*, so an error is produced:

```
let id :: forall a . a -> a
    xy :: exists a . (a,a -> Int)
    xy = (3,id)
    ixy :: (exists a . (a,a -> Int)) -> Int
    ixy = \(v,f) -> f v
    xy' = ixy xy
    pq :: exists a . (a,a -> Int)
    pq = ('x',id)
    {- ***ERROR(S):
         In `('x',id)':
           ... In `id':
                 Type clash:
                   failed to fit: forall a . a -> a <= Char -> Int
                   problem with : Char <= Int -}
    {- [ xy':Int, pq:(C_0_3_0,C_0_3_0 -> Int)
       , ixy:(exists a . (a,a -> Int)) -> Int
       , xy:(C_0_1_0,C_0_1_0 -> Int), id:forall a . a -> a ] -}
in xy'
```

Opening an existentially quantified type. The inverse of existential quantification of a type variable is often called 'opening'. It means that we get to know the original type. This of course cannot TBD: (???? type carrying code, analysis, etc) be done as this information was forgotten in the first place. Instead, the compiler 'invents' a fresh new type, a type constant, which acts as a placeholder for the opened type variable. This fresh type is guaranteed to be different from any type the programmer can construct. Only if functions accepting parameters of this type are available anything useful can be done with it, as in the example.

It also means that we can create values with which we cannot do much useful. For example

$$
\begin{array}{l}
\textbf{let}\ v_1 :: \exists\, a.a \\
\quad v_2 = v_1 \\
\textbf{in}\ v_2
\end{array}
$$

gives

```
let v1 :: exists a . a
    v2 = v1
    {- [ v2:C_0_0_0, v1:C_0_0_0 ] -}
in v2
```

Both examples also demonstrate the place where opening an existentially quantified type is done. Opening an existential type is done when the type is bound to an identifier. However, only the type variables for the top level existential quantifiers are opened. If an existentially quantified type is buried in a composite type it will only be opened if bound to a value identifier. For example:

$$\begin{array}{l}\textbf{let}\ v_1 :: (\exists\, a.a, \exists\, b.b) \\ \quad v_2 = v_1 \\ \quad (a,b) = v_1 \\ \quad (c,d) = v_1 \\ \textbf{in}\ \ v_2\end{array}$$

gives

```
let v1 :: (exists a . a,exists b . b)
    v2 = v1
    (a,b) = v1
    (c,d) = v1
    {- [ d:C_1_5_0, c:C_1_4_0, b:C_1_3_0, a:C_1_2_0
       , v2:(exists a . a,exists b . b)
       , v1:(exists a . a,exists b . b) ] -}
in v2
```

Also, opening the same type twice, as done in the given example for v_1, will twice give fresh type constants too.

This behavior simplifies the use of existential types in that no additional language construct for opening is necessary. More about this in TBD: [literature ...].

Guessing locations for quantifiers. Quantifiers need not always be specified. For example in

$$\begin{array}{l}\textbf{let}\ id \quad :: a \rightarrow a \\ \quad ixy \ = \lambda(v,f) \rightarrow f\ v \\ \quad pq \ \ = (\texttt{'x'}, id) \\ \quad xy \ \ :: (a, a \rightarrow Int) \\ \quad xy \ \ = (3, id) \\ \textbf{in}\ \ \textbf{let}\ xy' = ixy\ xy \\ \qquad\quad pq' = ixy\ pq \\ \quad \textbf{in}\ \ xy'\end{array}$$

no quantifiers are specified in the type signatures, for ixy a type signature is even absent. The following interpretation of the meaning of a type is used to determine where a quantifier should be inserted when a type is quantified.

- If a type variable a occurs somewhere in σ_1 and σ_2 in $\sigma_1 \rightarrow \sigma_2$ but not outside the function type, a will be universally quantified, i.e. with $\forall$.
- If a type variable a occurs somewhere in σ_1 and σ_2 in (σ_1, σ_2) but not outside the tuple type, a will be existentially quantified, i.e. with $\exists$.

The idea is that the first rule covers the notion that if an a is passed in and comes out a function, this function will work for all a, hence the universal quantification. On the other hand, the second rule covers the notion that if an a is stored together with another and nothing more is known about a we might as well hide *a*, hence the existential quantification. More rules are needed but we will look into this further when we look at the implementation.

For the given example the following will be concluded

```
let id :: a -> a
    ixy = \(v,f) -> f v
    pq = ('x',id)
    xy :: (a,a -> Int)
    xy = (3,id)
    {- [ pq:(Char,forall a . a -> a)
       , ixy:forall a . (exists b . (b,b -> a)) -> a
       , xy:(C_0_1_0,C_0_1_0 -> Int), id:forall a . a -> a ] -}
in let xy' = ixy xy
       pq' = ixy pq
       {- [ pq':Char, xy':Int ] -}
   in xy'
```

Note that an explicit type is needed to hide type information (as for *xy*), but is not required to pass it to a function expecting an existential.

Outline of the rest. First we will look at the type structure (section 5.1). The *fitsIn* function will have to be redesigned almost completely (section 5.2), partly because of the presence of quantifiers everywhere, partly because *fitsIn* now really is asymmetric because (e.g.) forgetting type information is one-way only. Quantification as well as instantiation become more complicated because of the presence of quantifiers anywhere in a type (section 5.4, section 5.3). Finally type inferencing expressions and patterns will have to be modified (section 5.5), luckily not much because the hard work is done in *fitsIn*.

5.1 Type language

The type language for this version of EH adds quantification with the existential quantifier $\exists$

$$
\begin{aligned}
\sigma = {} & \mathit{Int} \mid \mathit{Char} \mid c \\
\mid {} & (\sigma, ..., \sigma) \\
\mid {} & \sigma \rightarrow \sigma \\
\mid {} & v \mid f \\
\mid {} & \mathcal{Q}\ \alpha.\sigma, \mathcal{Q} \in \{\forall, \exists\}
\end{aligned}
$$

We also need an infinite supply of type constants *c*. Quantifiers $\mathcal{Q}$ may now appear anywhere in a type.

The *Quant* alternative of the type structure has to made more general to be able to encode $\exists$ too.

EHTyAbsSyn

DATA *Ty*
  | *Quant qu* : *TyQu*
          tv : {*TyVarId*}
          ty : *Ty*

DATA *TyQu*
  | *Forall*
  | *Exists*

Some additional functions on quantifiers *TyQu* are defined here too. These may seem unnecessary but the extra layer of abstraction is convenient when the range of quantifiers is extended in section 7 :

EHTy

```
tyquForall TyQu_Exists   = TyQu_Forall
tyquForall q             = q
tyquExists TyQu_Forall   = TyQu_Exists
tyquExists q             = q
tyquIsForall TyQu_Forall = True
tyquIsForall _           = False
tyquIsExists TyQu_Exists = True
tyquIsExists _           = False
```

These functions inquire the kind of quantifier and flip between universal and existential quantifier.

5.2 Fitting, subsumption

First we will look at the issues arising with the presence of quantifiers in a type. Next we will look at the implementation.

5.2.1 Issues

$\leqslant$ is a partial ordering. With the presence of quantifiers during type inferencing, the function *fitsIn*, implementating the so called *subsumption* relation $\leqslant$ between types, now becomes a partial ordering on types. For example,

```
let v1 :: ∃ a.a
    v1 = 3
    v2 :: Int
    v2 = v1
in  v1
```

makes the type inferencer check $Int \leqslant \exists\ a.a$, if the type of the actual value 3 fits in the specified type $\exists\ a.a$. This corresponds to the flow of 3 into some location, which in turn is then later on used elsewhere. This checks out well, but not the other way around because we cannot assume that some unknown type is an *Int*.

Similarly for universal quantification the check $\forall\ a.a \rightarrow a \leqslant Int \rightarrow Int$ holds as we can use a more generally applicable function *id* in the more restricted required setting $Int \rightarrow Int$.

```
let ii :: Int → Int
    ii  = id
    id :: ∀ a.a → a
    id = ii
in  ii
```

Losing generality, or specialization/instantiation to a specific type is ok, but the other way around, for the definition of *id* it is not.

So, in a nutshell

$$\forall\ a.a \leqslant \sigma \leqslant \exists\ a.a$$

meaning that a universally quantified type can be used at a place where any other type is expected, say some type σ (e.g. an *Int*), which in turn can be used at a place where all is forgotten about a type. Or, in other words, first we choose some type for a, then we forget this choice.

Let us look at some other examples to get a better idea of what *fitsIn*/$\leqslant$ has to deal with.

Impredicativeness. The following example (from Botlan [**?**])

$$\begin{array}{l}\textbf{let } \mathit{choose} :: a \to a \to a \\ \quad\; \mathit{id} \quad\;\; :: a \to a \\ \quad\; v \quad\;\;\; = \mathit{choose}\ \mathit{id} \\ \textbf{in } v\end{array}$$

demonstrates a choice we can make. This choice coincides with the question what the type of v is.

- $v :: \forall\ b.(b \to b) \to b \to b$. This is Haskell's answer. This answer is given because all types are monomorphic, that is, without quantifiers during inferencing. The type variable a of the type of *choose* is bound to the instantiated type if *id*. So, *id* is instantiated to $v_1 \to v_1$, giving $\mathit{choose}\ \mathit{id} :: (v_1 \to v_1) \to v_1 \to v_1$ resulting in the given quantified type. Function v can safely be applied to a function of type $\mathit{Int} \to \mathit{Int}$.
- $v :: (\forall\ c.c \to c) \to (\forall\ d.d \to d)$. This is system-F's answer [**?**]. This answer is given because the type variable a of *choose* is bound to the type of the parameter as it is known, with quantifiers, in its uninstantiated original form. Now v cannot be applied to a function of type $\mathit{Int} \to \mathit{Int}$ because this function is not general enough to be used as if it were of type $\forall\ a.a \to a$. Paradoxically enough a more general function will be returned; this relates to a phenomenon called *contravariance* which we will discuss later.

Allowing type variables to be bound to, or instantiated with, quantified types is called *impredicativeness* TBD: [cite...].

The critical observation for this version of EH is that it is difficult TBD: [cite...] to infer that a type variable should be bound to a quantified type, but that it is relatively easy not to forget that a type is quantified if we already knew in the first place that it was quantified. The latter is what we do except in situations where it would break Haskell's choice, thereby still inferring types in a standard Hindley-Milner way but using *fitsIn*/$\leqslant$ to allow richer types still to match properly.

Subsumption $\leqslant$ needs to instantiate types. These examples also demonstrate that *fitsIn* needs to instantiate types. In the previous version of EH all types partaking in the inference process were expected to be fully instantiated. By definition this can no longer be done if as much as possible type information is to be retained. Still, at some point instantiation has to be done, in particular at the latest moment possible. This latest moment is the place where a type really is compared with another one, in *fitsIn*.

Subsumption $\leqslant$ depends on context. *fitsIn* is used as a tool to enforce the $\leqslant$ relation between types. It is used to check the type of an actual parameter with its expected one, as in the previous example for *choose* (page 58). It is also used to check the type of a value against its known type as in the earlier example with *ii* and *id* (page 57). However, the precise use of *fitsIn* in these contexts differs slightly in the way instantiation is done.

- For an application $f\ a$ of $f :: \sigma_2 \to ...$ to $a :: \sigma_1$ we have to check the type σ_1 of an actual argument against the type σ_2 of an expected via $\sigma_1 \leqslant \sigma_2$. As we learned from looking at the *choose* example, Haskell's convention is to instantiate σ_1 before binding it to a type variable v in the case $\sigma_2 \equiv v$. This information is passed as an additional parameter to *fitsIn*, notated by ν_{il}, named an *instantiating context*.
- For checking an expression $e :: \sigma_2$ of a declaration $v :: \sigma_1; v = e$ to its known type σ_1 we check $\sigma_2 \leqslant \sigma_1$. In this case we want to avoid instantiating σ_2. The necessity of avoiding this becomes clear if we look at a situation where $\sigma_2 \equiv (\exists\ a.a, ...)$ and $\sigma_1 \equiv (v, ...)$, coinciding with a situation where an explicit type signature is absent. Now, if $\exists\ a.a$ is instantiated with type constants

before it is bound to v all information about the existential is irretrievably lost, something we do only when an existential is bound to an identifier. So, in this case we say that *fitsIn* needs to told it is checking types in a *strong context*, notated by ν_s.

- A third context will also be mentioned here for completeness, a so called *weak context* ν_w. It is used whenever an expression can have >1 alternative expressions as a result, which is the case for **case**-expressions, to be dealt with no sooner than the introduction of datatypes in the next version of EH (section 6).

fitsIn/$\leqslant$ therefore needs an option ν to describe these variations

```
ν = ν_s     -- strong context
  | ν_il    -- instantiating context, for expr App
  | ν_i     -- instantiating context, for patterns
```

These contextual variations actually are configurations of lowlevel boolean flags for *fitsIn*

```
fioBindRFirst = fi+_{r-bind}   -- prefer binding of a rhs tvar over instantiating
              | fi-_{r-bind}
fioBindLFirst = fi+_{l-bind}   -- prefer binding of a lhs tvar over instantiating
              | fi-_{l-bind}
fioLeaveRInst = fi+_{r-inst}   -- leave rhs (of fitsIn) instantiated
              | fi-_{r-inst}
```

where the $+$ variants stand for *True*. A *True* value for the flag *fioBindRFirst* states that in case of $\sigma \leqslant v$ a constraint $(v \mapsto \sigma)$ will result, otherwise first σ will be instantiated and v be bound to the instantiated σ. Similary we have *fioBindLFirst* for $v \leqslant \sigma$. Finally, *fioLeaveRInst* determines if an instantiation done for σ in ... $\leqslant \sigma$ will return the instantiated σ or σ itself. Summarizing, *fioBindRFirst* and *fioBindLFirst* turn off greedy instantiating and *fioLeaveRInst* leaves visible what has been instantiated.

Context variations and these flags relate as follows

	fioBindRFirst	*fioBindLFirst*	*fioLeaveRInst*
ν_s	fi^+_{r-bind}	fi^+_{l-bind}	fi^-_{r-inst}
ν_{il}	fi^-_{r-bind}	fi^+_{l-bind}	fi^-_{r-inst}
ν_i	fi^-_{r-bind}	fi^-_{l-bind}	fi^+_{r-inst}

So, for example the ν_i context variant used for an expression application has as its effect that instantiation will be done a la Hindley-Milner.

Finally, all of this is encoded as follows

EHTyFitsIn

```
data FIOpts = FIOpts{fioLeaveRInst :: Bool ,fioBindRFirst :: Bool
                    , fioBindLFirst :: Bool ,fioUniq       :: UID
                    , fioCoContra   :: CoContraVariance
                    } deriving Show
ν_s :: FIOpts
ν_s = FIOpts        {fioLeaveRInst = False,fioBindRFirst = True
                    , fioBindLFirst = True ,fioUniq       = uidStart
                    , fioCoContra   = ⊕
                    }
```

EHTyFitsIn

```
ν_il :: FIOpts
ν_il = ν_s{fioBindRFirst = False}
ν_i :: FIOpts
ν_i = ν_il{fioLeaveRInst = True,fioBindLFirst = False}
```

Co- and contravariance. For tuples the check $(\sigma_1, \sigma_2) \leqslant (\sigma_3, \sigma_4)$ will break down into $\sigma_1 \leqslant \sigma_3$ and $\sigma_2 \leqslant \sigma_4$ because conceptually a tuple value can be put into a location if that location expects a tuple and the elements of the tuple also fit.

For functions this works differently. Checking $\sigma_1 \rightarrow \sigma_2 \leqslant \sigma_3 \rightarrow \sigma_4$ means that a $f :: \sigma_3 \rightarrow \sigma_4$ is expected but a $g :: \sigma_1 \rightarrow \sigma_2$ is available. This happens for example in

$$\begin{array}{l}\textbf{let}\ g :: \sigma_1 \rightarrow \sigma_2 \\ \quad f :: \sigma_3 \rightarrow \sigma_4 \\ \quad f = g \\ \quad a :: \sigma_3 \\ \textbf{in}\ f\ a\end{array}$$

So, what happens when f is called and what does it mean in terms of types σ? The caller of f in the application $f\ a$ expects that the function f accepts a σ_3. However, g is invoked instead, so a value of type σ_3 is passed to a function expecting a σ_1, which in terms of fitting means $\sigma_3 \leqslant \sigma_1$. The observation here is that the direction of $\leqslant$ for fitting the argument types of f and g is opposite to the direction of fitting f and g. This behavior is called *contravariance*.

In general, fitting a composite type breaks down into fitting the components of the composite type. If the direction of $\leqslant$ for fitting a component is the same as for the composite type, it is said that that component of the type is *covariant*, if the direction is opposite the component is *contravariant*.

The following notation is used to denote this variance

$$\begin{array}{lll}\mathcal{V} = & \oplus & \text{-- CoVariant} \\ \quad\ \mid & \ominus & \text{-- ContraVariant} \\ \quad\ \mid & \odot & \text{-- CoContraVariant (both co/contra)}\end{array}$$

with a corresponding Haskell definition

EHCommon

$\textbf{data}\ \mathit{CoContraVariance} = \oplus \mid \ominus \mid \odot\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})$

in which the same notation is used for the alternatives of *CoContraVariance*.

5.2.2 Implementation

Typing rules. Let us look more precisely at $\leqslant$ which we now also will describe with rules, in figure 13 and figure 12. Rule f-prod4 and rule f-arrow4 both follow the discussion about co- and contravariance. These rules are both instances of the by now usual *App* structure which will be used by *fitsIn*.

The fine detail here lies with rule f-arrow4 which specifies a strong context ν_s for fitting its arguments. This means that for higher ranked positions in a type any implicit instantiation of types is inhibited, that is, it is not visible for the inference process. TBD: more explanation why...

The rules for quantified types also deserve a bit more attention.

- Rule f-forall-r2 applies in

  $$\begin{array}{l}\textbf{let}\ ii\ :: \mathit{Int} \rightarrow \mathit{Int} \\ \quad ii\ = id \\ \quad id :: \forall\ a.a \rightarrow a \\ \quad id = ii \\ \textbf{in}\ \ ii\end{array}$$

  to the check $\mathit{Int} \rightarrow \mathit{Int} \leqslant \forall\ a.a \rightarrow a$ which has to be done for $id = ii$. If $\forall\ a.a \rightarrow a$ is instantiated with type variables v, the check would succeed with a constraint $(v \mapsto \mathit{Int})$. This is not correct.

$$\boxed{\nu \overset{fit}{\vdash} \sigma^l \leqslant \sigma^r : \sigma \leadsto \mathcal{C}}$$

$$\frac{\begin{array}{c}\nu_s \overset{fit}{\vdash} \mathcal{C}_1\sigma^a_2 \leqslant \mathcal{C}_1\sigma^a_1 : \sigma^a \leadsto \mathcal{C}_2 \\ \nu \overset{fit}{\vdash} \sigma^r_1 \leqslant \sigma^r_2 : \sigma^r \leadsto \mathcal{C}_1\end{array}}{\nu \overset{fit}{\vdash} \sigma^a_1 \to \sigma^r_1 \leqslant \sigma^a_2 \to \sigma^r_2 : \sigma^a \to \mathcal{C}_2\sigma^r \leadsto \mathcal{C}_{2..1}} \quad \text{(f-arrow4)}$$

$$\frac{\begin{array}{c}\nu \overset{fit}{\vdash} \mathcal{C}_1\sigma^l_1 \leqslant \mathcal{C}_1\sigma^l_2 : \sigma^l \leadsto \mathcal{C}_2 \\ \nu \overset{fit}{\vdash} \sigma^r_1 \leqslant \sigma^r_2 : \sigma^r \leadsto \mathcal{C}_1\end{array}}{\nu \overset{fit}{\vdash} (\sigma^l_1, \sigma^r_1) \leqslant (\sigma^l_2, \sigma^r_2) : (\sigma^l, \mathcal{C}_2\sigma^r) \leadsto \mathcal{C}_{2..1}} \quad \text{(f-prod4)}$$

Figure 12: Fitting/subsumption for type applications

$$\boxed{\nu \overset{fit}{\vdash} \sigma^l \leqslant \sigma^r : \sigma \leadsto \mathcal{C}}$$

$$\frac{\begin{array}{c}\nu \overset{fit}{\vdash} \rho^i \leqslant \sigma_2 : \sigma \leadsto \mathcal{C} \\ (\_, \rho^i) \equiv inst_v(\overline{\alpha}, \rho_1)\end{array}}{\nu \overset{fit}{\vdash} \forall\overline{\alpha}.\rho_1 \leqslant \sigma_2 : \sigma \leadsto \mathcal{C}} \quad \text{(f-forall-l)}$$

$$\frac{\begin{array}{c}fi^+_{r-inst} \overset{fit}{\vdash} \sigma_1 \leqslant \rho^i : \sigma \leadsto \mathcal{C} \\ (\_, \rho^i) \equiv inst_v(\overline{\beta}, \rho_2)\end{array}}{fi^+_{r-inst} \overset{fit}{\vdash} \sigma_1 \leqslant \forall\overline{\beta}.\rho_2 : \sigma \leadsto \mathcal{C}} \quad \text{(f-forall-r1)} \qquad \frac{\begin{array}{c}fi^-_{r-inst} \overset{fit}{\vdash} \sigma_1 \leqslant \rho^i : \_ \leadsto \mathcal{C} \\ (\_, \rho^i) \equiv inst_f(\overline{\beta}, \rho_2)\end{array}}{fi^-_{r-inst} \overset{fit}{\vdash} \sigma_1 \leqslant \forall\overline{\beta}.\rho_2 : \mathcal{C}\ (\forall\overline{\beta}.\rho_2) \leadsto \mathcal{C}} \quad \text{(f-forall-r2)}$$

$$\frac{\begin{array}{c}\nu \overset{fit}{\vdash} \rho^i \leqslant \sigma_2 : \sigma \leadsto \mathcal{C} \\ (\_, \rho^i) \equiv inst_c(\overline{\alpha}, \rho_1)\end{array}}{\nu \overset{fit}{\vdash} \exists\overline{\alpha}.\rho_1 \leqslant \sigma_2 : \sigma \leadsto \mathcal{C}} \quad \text{(f-exists-l)}$$

$$\frac{\begin{array}{c}fi^+_{r-inst} \overset{fit}{\vdash} \sigma_1 \leqslant \rho^i : \sigma \leadsto \mathcal{C} \\ (\_, \rho^i) \equiv inst_c(\overline{\beta}, \rho_2)\end{array}}{fi^+_{r-inst} \overset{fit}{\vdash} \sigma_1 \leqslant \exists\overline{\beta}.\rho_2 : \sigma \leadsto \mathcal{C}} \quad \text{(f-exists-r1)} \qquad \frac{\begin{array}{c}fi^-_{r-inst} \overset{fit}{\vdash} \sigma_1 \leqslant \rho^i : \sigma \leadsto \mathcal{C} \\ (\overline{v}, \rho^i) \equiv inst_v(\overline{\beta}, \rho_2)\end{array}}{fi^-_{r-inst} \overset{fit}{\vdash} \sigma_1 \leqslant \exists\overline{\beta}.\rho_2 : \mathcal{C}\ (\exists\overline{\beta}.\rho_2) \leadsto \mathcal{C} \backslash^{dom}_{\overline{v}}} \quad \text{(f-exists-r2)}$$

Figure 13: Fitting/subsumption for quantified types

Recall that by succesfully passing this check $Int \rightarrow Int$ will be used as if it were a $\forall\ a.a \rightarrow a$, which definitely will not work for all types. So, in order to let this check fail we instantiate with the fixed variant f of type variables, indicated by $inst_f$. These fixed type variables cannot be further constrained but quantification over them is allowed (see ...).

- Dually, rule f-exists-r2 applies in

  ```
  let f :: (∃ a.a) → Int
      v = f 3
  in  v
  ```

  to the check $Int \leqslant \exists\ a.a$ for the application f 3. The body of f only knows it gets passed a value of some unknown type, no assumptions about it are made in the body of f. Consequently, type variable may be instantiated with any type by the caller of f. This is simulated by instantiating with fresh constrainable type variables v, via $inst_v$. In that case we also are not interested in any found constraints concerning the fresh type variables, so these are removed.

Co- and contravariance. The encoding of co- and contravariance behavior is solved a bit more general then really is required at this point. The idea is that for a type application $c\ a\ b...$ the c determines how the fitting of its arguments should be done. For example, for $c \equiv \rightarrow$, the first argument should be fitted with the opposite variance and options ν should be made strong. This is described via a environment encoding this information

EHTyFitsIn

```
type CoCoInfo = [(CoContraVariance → CoContraVariance, FIOpts → FIOpts)]
data CoCoGamInfo = CoCoGamInfo{ccgiCoCoL :: CoCoInfo}
type CoCoGam = Gam HsName CoCoGamInfo
mkStrong :: FIOpts → FIOpts
mkStrong fi = fi{fioLeaveRInst = False, fioBindRFirst = True, fioBindLFirst = True}
cocoGamLookup :: Ty → CoCoGam → CoCoInfo
cocoGamLookup ty g
  = case ty of
      Ty_Con nm → case gamLookup nm g of
                     Just ccgi                → ccgiCoCoL ccgi
                     Nothing | hsnIsProd nm → take (hsnProdArity nm) (repeat (id, id))
                     _                        → unknownCoCoL
      _         → unknownCoCoL
  where unknownCoCoL = repeat (const ⊙, id)
```

EHTyFitsIn

```
cocoGam :: CoCoGam
cocoGam = assocLToGam [(hsnArrow, CoCoGamInfo [(cocoOpp, mkStrong), (id, id)])]
```

It also shows that only for function and tuple types we know what to do in such a situation. Complications in this area will arise with the introduction of datatypes in section 6 .

***fitsIn* parameters and result.** So, let us know finally look at the implementation of $\leqslant$, and redo the implementation of *fitsIn*. First, *fitsIn* needs to pass information both up and downwards. Upwards was already implemented via

EHTyFitsIn

```
data FIOut = FIOut{foCnstr :: C       ,foTy        :: Ty
                  ,foUniq  :: UID     ,foCoContraL :: CoCoInfo
                  ,foErrL  :: ErrL
                  }
emptyFO    = FIOut{foCnstr = emptyCnstr,foTy          = Ty_Any
                  ,foUniq  = uidStart  ,foCoContraL = [ ]
```

```
            , foErrL  = [ ]
            }
```

which is extended with $\odot$ information and threads a UID value needed for instantiating types together with the downward information stored in

EHTyFitsIn

```
data FIIn = FIIn{fiFIOpts   :: FIOpts, fiUniq :: UID
                , fiCoContra :: CoContraVariance
                }
emptyFI   = FIIn{fiFIOpts   = ν_s     , fiUniq = uidStart
                , fiCoContra = ⊕
                }
```

The parameter *fiCoContra* is used to indicate if the comparison $\leqslant$ is flipped.

The fitting. The preliminaries of *fitsIn* have not changed much compared to the previous version (page 36). All internally defined functions now take an additional top-down *fi* :: *FIIn* parameter and some work has to be done for extracting and passing variance information (in function *res*).

EHTyFitsIn

```
fitsIn :: FIOpts → UID → Ty → Ty → FIOut
fitsIn opts uniq ty_1 ty_2
    =  fo
  where
    res fi t         = emptyFO{foUniq = fiUniq fi, foTy = t
                              , foCoContraL = cocoGamLookup t cocoGam}
    err fi e         = emptyFO{foUniq = fioUniq opts, foErrL = e}
    manyFO fos       = foldr1 (λfo_1 fo_2 → if foHasErrs fo_1 then fo_1 else fo_2) fos
    bind fi tv t     = (res fi t){foCnstr = tv ↦ t}
    occurBind fi v t
       | v ∈ ftv t   = err fi [Err_UnifyOccurs ty_1 ty_2 v t]
       | otherwise   = bind fi v t
```

The fitting of quantified types uses *unquant* which removes all top level quantifiers.

EHTyFitsIn

```
    unquant fi t @(Ty_Quant _ _ _) hide howToInst
       = let (u, uq)         = mkNewLevUID (fiUniq fi)
             (uqt, rtvs)     = tyInst1Quants uq howToInst t
             back            = if hide thenλfo → let s = cnstrFilter (const.¬.(∈ rtvs)) (foCnstr fo)
                                                 in  fo{foCnstr = s, foTy = s ≻ t}
                                          else id
         in  (fi{fiUniq = u}, uqt, back)
```

The instantiation is parameterized a flag *hide* telling if any found constraints for the fresh type variables *rtvs* should be hidden. A second parameter *howToInst* :: *HowToInst* specifies how to instantiate. When discussing the implementation for quantifiers we will look at this further.

The first cases of the actual implementation of $\leqslant$ are similar to the previous version with the exception of an alternative for flipping $t_1 \leqslant t_2$ into $t_2 \leqslant t_1$ if the variance is $\ominus$, and an additional guard on *fioBindLFirst* and *fioBindRFirst*.

EHTyFitsIn

```
    u fi t_1                 t_2
       | fiCoContra fi ≡ ⊖          = u    (fi{fiCoContra = ⊕
                                               , fiFIOpts = fioSwapVariance (fiFIOpts fi)})
                                           t_2 t_1
    u fi Ty_Any              t_2    = res fi t_2
    u fi t_1                 Ty_Any = res fi t_1
    u fi t_1 @(Ty_Con s1)    t_2 @(Ty_Con s2)
    | s1 ≡ s2                      = res fi t2
u fi t1 @(Ty_Var v1 f1)    (Ty_Var v2 f2)
    | v1 ≡ v2 ∧ f1 ≡ f2            = res fi t1
u fi t1 @(Ty_Var v1 f)      t2
    | fioBindLFirst (fiFIOpts fi) ∧ f ≡ TyVarCateg_Plain
                                   = occurBind fi v1 t2
u fi t1                    t2 @(Ty_Var v2 f)
    | fioBindRFirst (fiFIOpts fi) ∧ f ≡ TyVarCateg_Plain
                                   = occurBind fi v2 t1
```

The order in which all these case are listed is now important as the cases for fi^{-}_{l-bind} and fi^{-}_{r-bind} will be executed only after types are stripped of their top level quantifiers.

Compared to the rules in figure 13 an additional case has been included for an exact match of two quantified types when we want $t_1 \leqslant t_2$ and $t_2 \leqslant t_1$ both to hold. We will postpone discussion until section 6 .

EHTyFitsIn

```
u fi t1 @(Ty_Quant q1 _ _) t2 @(Ty_Quant q2 _ _)
    | fiCoContra fi ≡ ⊙ ∧ q1 ≡ q2
                                   = u fi2 uqt1 uqt2
    where (fi1, uqt1, _) = unquant fi t1 False instCoConst
          (fi2, uqt2, _) = unquant fi1 t2 False instCoConst
```

The function *unquant* has to be told how to do the instantiation, this is specified by a function which creates a type from a type variable and a quantifier.

EHTyInstantiate

```
type HowToInst = TyQu → TyVarId → Ty
instCoConst, instContra :: HowToInst
instCoConst q v = if tyquIsForall q then Ty_Var v TyVarCateg_Plain else mkTyCon ("C_" ⧺ show v)
instContra  q v = if tyquIsForall q then Ty_Var v TyVarCateg_Fixed else Ty_Var v TyVarCateg_Plain
```

The rules in figure 13 indicate for different combinations of options and quantifiers how to instantiate type variables. For example, the first case of

EHTyFitsIn

```
u fi t1                    t2 @(Ty_Quant _ _ _)
    | fiCoContra fi ≢ ⊙ ∧ fioLeaveRInst (fiFIOpts fi)
                                   = back2 (u fi2 t1 uqt2)
    where (fi2, uqt2, back2) = unquant fi t2 False instCoConst
u fi t1                    t2 @(Ty_Quant _ _ _)
    | fiCoContra fi ≢ ⊙ ∧ ¬ (fioLeaveRInst (fiFIOpts fi))
                                   = back2 (u fi2 t1 uqt2)
    where (fi2, uqt2, back2) = unquant fi t2 True instContra
```

implements rule f-forall-r1 and rule f-exists-r1. The behavior with respect to the different ways of instantiating is encoded in *instCoConst* which tells us that the universally quantified types should be instantiated with type variables, and existentially quantified types with type constants. The second case similarly covers rule f-forall-r2 and rule f-exists-r2 while

EHTyFitsIn

```
u fi t1 @(Ty_Quant _ _ _)  t2
    | fiCoContra fi ≢ ⊙            = u fi1 uqt1 t2
    where (fi1, uqt1, back1) = unquant fi t1 False instCoConst
```

covers the remaining rule f-forall-l and rule f-exists-l.

Checking two application of types, implementing both rule f-prod4 and rule f-arrow4, has changed with respect to the handling of co- and contravariance. From the resulting *foCoContraL* the first element describes the co- and contravariance behavior, as such it is used to update the *fi* :: *FIIn* accordingly.

EHTyFitsIn

```
u fi t₁ @(Ty_App tf₁ ta₁)    t₂ @(Ty_App tf₂ ta₂)
    = manyFO [ffo, afo, rfo]
    where ffo  = u fi tf₁ tf₂
          fs   = foCnstr ffo
          ((coUpd, fiUpd) : cor) = foCoContraL ffo
          fi'  = fi{fiCoContra = coUpd (fiCoContra fi), fiFIOpts = fiUpd (fiFIOpts fi)
                   , fiUniq     = foUniq ffo}
          afo = u fi' (fs ≻ ta₁) (fs ≻ ta₂)
          as  = foCnstr afo
          rt  = Ty_App (as ≻ foTy ffo) (foTy afo)
          rfo = afo{foTy = rt, foCnstr = as ≻ fs, foCoContraL = cor}
```

All is left of *fitsIn* are the remaining cases for type variables, now for the fi^{-}_{l-bind} and fi^{-}_{r-bind} cases EHTyFitsIn

```
u fi t₁ @(Ty_Var v1 f)        t₂
    | f ≡ TyVarCateg_Plain          = occurBind fi v1 t₂
u fi t₁                       t₂ @(Ty_Var v2 f)
    | f ≡ TyVarCateg_Plain          = occurBind fi v2 t₁
```

and starting it all EHTyFitsIn

```
u fi t₁                       t₂         = err fi [Err_UnifyClash ty₁ ty₂ t₁ t₂]
fo  = u (emptyFI{fiUniq = uniq, fiFIOpts = opts, fiCoContra = fioCoContra opts}) ty₁ ty₂
```

5.3 Instantiation

Function *fitsIn* is now one of the two places where instantiation of a type occurs.

- In *fitsIn* the instantiation of a quantified type now is entangled with matching two types. *fitsIn* peels off top level quantifiers layer by layer during the matching of types.
- When a type is bound to an identifier, we have to instantiate top level ∃'s to open the type.

These variants are implemented by *tyInst1Quants* and *tyInst1Exists* respectively: EHTyInstantiate

```
tyInst :: UID → Bool → HowToInst → Ty → (Ty, TyVarIdL)
tyInst uniq onlyExists howToInst ty
    = (s ≻ ty', tvl)
  where i u (Ty_Quant q v t) | ¬ (tyquIsForall q ∧ onlyExists)
                    = let (u', v')        = mkNewUID u
                          (s, t', tvl') = i u' t
                      in  ((v ↦ (howToInst q v')) ≻ s, t', v' : tvl')
        i _ t       = (emptyCnstr, t, [ ])
        (s, ty', tvl) = i uniq ty
tyInst1Quants :: UID → HowToInst → Ty → (Ty, TyVarIdL)
tyInst1Quants uniq howToInst ty = tyInst uniq False howToInst ty
tyInst1Exists :: UID → Ty → Ty
tyInst1Exists uniq ty = fst (tyInst uniq True instCoConst ty)
```

An additional *onlyExists* :: *Bool* is passed to the more general function *tyInst* to inhibit quantification of ∀'s.

Note that in the previous version of EH instantiation was done explicitly, as indicated by type rules. In this version instantiation is done implicitly by *fitsIn*.

5.4 Quantification

Quantification is more complicated because the place of omitted quantifiers has to be guessed. This guessing is done according to the rules in figure 14. The structure of a type and the occurrences of type variables are used to determine where which quantifier is inserted during the quantification of a type.

$$\boxed{bv, \mathcal{V} \overset{qu}{\vdash} \sigma : \sigma^q \rightsquigarrow fv}$$

$$\frac{v \notin bv}{bv, \oplus \overset{qu}{\vdash} v : \forall\, v.v \rightsquigarrow [v]} \ \text{(q-var-co)} \qquad \frac{v \notin bv}{bv, \ominus \overset{qu}{\vdash} v : \exists\, v.v \rightsquigarrow [v]} \ \text{(q-var-contra)}$$

$$\frac{\begin{array}{c} v \in (fv_1 \cap fv_2) - bv \\ [v] \cup bv, \ominus \overset{qu}{\vdash} \sigma_1 : \sigma_1^q \rightsquigarrow fv_1 \\ [v] \cup bv, \oplus \overset{qu}{\vdash} \sigma_2 : \sigma_2^q \rightsquigarrow fv_2 \end{array}}{bv, \_ \overset{qu}{\vdash} \sigma_1 \to \sigma_2 : \forall\, v.\sigma_1^q \to \sigma_2^q \rightsquigarrow (fv_1 \cup fv_2) - [v]} \ \text{(q-arrow)}$$

$$\frac{\begin{array}{c} v \in (fv_1 \cap fv_2) - bv \\ [v] \cup bv, \oplus \overset{qu}{\vdash} \sigma_1 : \sigma_1^q \rightsquigarrow fv_1 \\ [v] \cup bv, \oplus \overset{qu}{\vdash} \sigma_2 : \sigma_2^q \rightsquigarrow fv_2 \end{array}}{bv, \_ \overset{qu}{\vdash} (\sigma_1, \sigma_2) : \exists\, v.(\sigma_1^q, \sigma_2^q) \rightsquigarrow (fv_1 \cup fv_2) - [v]} \ \text{(q-prod)}$$

$$\frac{\begin{array}{c} v \in (fv_1 \cap fv_2) - bv \\ [v] \cup bv, \odot \overset{qu}{\vdash} \sigma_1 : \sigma_1^q \rightsquigarrow fv_1 \\ [v] \cup bv, \odot \overset{qu}{\vdash} \sigma_2 : \sigma_2^q \rightsquigarrow fv_2 \\ \mathcal{Q} \equiv \textbf{if } \mathcal{V} \equiv \oplus \textbf{ then } \forall \textbf{ else } \exists \\ \mathcal{V} \in \{\oplus, \ominus\} \end{array}}{bv, \mathcal{V} \overset{qu}{\vdash} \sigma_1\ \sigma_2 : \mathcal{Q}\ v.\sigma_1^q\ \sigma_2^q \rightsquigarrow (fv_1 \cup fv_2) - [v]} \ \text{(q-app)}$$

$$\frac{\begin{array}{c} v \notin bv \\ [v] \cup bv, \mathcal{V} \overset{qu}{\vdash} \sigma : \sigma^q \rightsquigarrow fv \end{array}}{bv, \mathcal{V} \overset{qu}{\vdash} \mathcal{Q}\ v.\sigma : \mathcal{Q}\ v.\sigma^q \rightsquigarrow fv - [v]} \ \text{(q-quant)}$$

Figure 14: Quantifier location inferencing

Informally, the following rules are obeyed

- The first step is to find for a type variable a the smallest part of a type where a occurs. For example, for $a \to (a, b, b, b \to c \to (b, c))$ this is the complete type for a, $(a, b, b, b \to c \to (b, c))$ for b and $c \to (b, c)$ for c.

$$\boxed{bv, \mathcal{V} \overset{quGam}{\vdash} \Gamma : \Gamma^q}$$

$$\frac{\begin{array}{c} bv, \mathcal{V} \overset{qu}{\vdash} \sigma : \sigma^q \leadsto \_ \\ bv, \mathcal{V} \overset{quGam}{\vdash} \Gamma : \Gamma^q \end{array}}{bv, \mathcal{V} \overset{quGam}{\vdash} [\xi \mapsto \sigma, \Gamma] : [\xi \mapsto \sigma^q, \Gamma^q]} \quad \text{(qg-cons4)}$$

Figure 15: Quantifier location inferencing for types in a Gamma

- If this smallest part is a function type it is assumed that universal quantification is the right choice (rule q-arrow).
- If this smallest part is a tuple type the type variable is existentially quantified (rule q-prod).
- For the remaining cases the position of the smallest part as part of a function type determines which quantifier is put in front. For example, in $a \rightarrow b$ the type variable a occurs in a contravariant (argument) position, b in a covariant position. Here the observation is that a apparently is not used at all by the corresponding function because it does not show up in the result type. So, we might as well hide a, hence the $\exists$ in rule q-var-contra. Conversely, the choice of what b is apparently is up to the caller of the function, hence the $\forall$ in rule q-var-co.
- The rule q-app covers remaining type constructors, which is irrelevant for this version of EH as there are no other type constructors. It becomes relevant for the next version of EH, when datatypes are introduced (section 6).

Because these rules represent a form of syntactic sugar they always can be overruled by explicit quantifiers, as indicated by rule q-quant. The quantification is lifted to a Γ in a straightforward way as specified by figure 15.

The implementation, in terms of AG, follows the rules with the exception of some details. First all free type variables are gathered:

EHTyQuantify

```
ATTR AllTy [|| frTvLL : {[TyVarIdL]}]
SEM TyAGItf
  | AGItf      loc.frTvL  = head @ty.frTvLL
SEM Ty
  | Var        loc.frTvL  = [@tv]
  | App        loc.frTvLL = @arg.frTvLL ++ @func.frTvLL
                  .frTvL  = listCombineUniq @frTvLL
               lhs.frTvLL = if @isSpineRoot then [@frTvL] else @frTvLL
  | Quant      loc.frTvL  = head @ty.frTvLL \\ @introTVarL
  | Any Con    loc.frTvL  = []
  | Quant Var Any Con
               lhs.frTvLL = [@frTvL]
```

EHCommon

```
listCombineUniq :: Eq a => [[a]] → [a]
listCombineUniq = nub.concat
```

At the top of a type application, say *App* (*App* (*Con* "->") (*Var* 1)) (*Var* 1) representing $a \rightarrow a$ we need to be able to determine which type variables occur in both arguments of the type constructor $\rightarrow$. Therefore a list *frTvLL* : [*TyVarIdL*] of type variables is gathered, each element corresponding with the free type variables of an argument. This list corresponds to the *fv*'s in the rules in figure 14. For the given example this would be [[1], [1]].

Next we determine which locations in the type structure are a candidate for quantification:

EHTyQuantify

```
ATTR AllTy [𝒱 : CoContraVariance || ]
SEM Ty
  | App   func.𝒱 = ⊙
          arg .𝒱 = if      @appIsLikeProd ∨ @isArrowRoot then ⊕
                   else if@appIsArrow                    then ⊖
                                                         else  ⊙
SEM TyAGItf
  | AGItf ty   .𝒱 = ⊕
```

EHTyQuantify

```
SEM Ty
  | Var App Quant
    loc.isQuLoc = @lhs.𝒱 ≢ ⊙
```

Quantifiability of a location is based on co- and contravariance information as passed from top to bottom, as prescribed by the rules in figure 14. We also need to know what an *App* represents, that is, if it is a function type (*appIsArrow*) or tuple type (*appIsLikeProd*):

EHTyCommonAG

```
SEM Ty
  | App loc.isArrowRoot = @appIsArrow ∧ @isSpineRoot
```

If a location in the type structure is a place where quantification may take place, the candidate free type variables *qHereTvL* are computed:

EHTyQuantify

```
SEM TyAGItf
  | AGItf loc.qHereTvL = [ ]
SEM Ty
  | Var    loc.qHereTvL = if @isQuLoc then [@tv] else [ ]
  | App    loc.qHereTvL = if @isQuLoc
                          then if     @appIsArrow ∨ @appIsLikeProd
                               then tvarOccurGE2 @frTvLL
                               else  @frTvL
                          else  [ ]
```

EHTyQuantify

```
tvarOccurCount :: [TyVarIdL] → AssocL TyVarId Int
tvarOccurCount = map (λvl @(v : _) → (v, length vl)).group.sort.concat
tvarOccurGE2 :: [TyVarIdL] → TyVarIdL
tvarOccurGE2 = map fst.filter ((>1).snd).tvarOccurCount
```

The auxiliary function *tvarOccurGE2* selects those type variables which occur at least twice in the arguments of a type constructor.

From the top of the type downwards then the function *tvIsBound* is constructed, ensuring that candidate free type variables are not in the *bv* of the rules in figure 14.

EHTyQuantify

```
ATTR TyAGItf AllTy [tvIsBound : {TyVarId → Bool} || ]
SEM TyAGItf
  | AGItf     loc.(qBndTvL, tvIsBound) = tvarsToQuant True @lhs.tvIsBound @qHereTvL
SEM Ty
  | App Var loc.(qBndTvL, tvIsBound) = tvarsToQuant @isQuLoc @lhs.tvIsBound @qHereTvL
  | Quant     loc.tvIsBound              = tvBoundAdd @lhs.tvIsBound @introTVarL
```

EHTyQuantify

```
tvBoundAdd :: (TyVarId → Bool) → TyVarIdL → TyVarId → Bool
tvBoundAdd tvIsBound tvL = λv → v ∈ tvL ∨ tvIsBound v
```

```
tvarsToQuant :: Bool → (TyVarId → Bool) → TyVarIdL → (TyVarIdL, TyVarId → Bool)
tvarsToQuant isQuLoc tvIsBound tvL
  = if isQuLoc
    then let boundables = filter (¬.tvIsBound) tvL
         in (boundables, tvBoundAdd tvIsBound boundables)
    else ([ ], tvIsBound)
```

The resulting selection of type variables *qBndTvL* is then used with the quantifier *hereQu* which in turn is based on *qExists*, telling us if it is a location where $\exists$ is to be used:

EHTyQuantify

```
SEM TyAGItf
  | AGItf     loc.hereQu   = TyQu_Forall
SEM Ty
  | App       loc.qAsExist = @appIsLikeProd ∨ @lhs.𝒱 ≡ ⊖ ∧ ¬ @appIsArrow
  | Var       loc.qAsExist = @lhs.𝒱 ≡ ⊖
  | App Var   loc.hereQu   = if @qAsExist then TyQu_Exists else TyQu_Forall
  | Quant     loc.hereQu   = @qu.self
```

Finally the quantified type *quTy* is constructed:

EHTyQuantify

```
ATTR TyAGItf [|| quTy : Ty]
ATTR AllTy TyQu TyVarCateg [|| quTy : SELF]
SEM TyAGItf
  | AGItf  lhs.quTy = mkTyQu @hereQu @qBndTvL @ty.quTy
SEM Ty
  | Var    lhs.quTy = mkTyQu @hereQu @qBndTvL (Ty_Var @tv @categ.quTy)
  | App    lhs.quTy = mkTyQu @hereQu @qBndTvL (Ty_App @func.quTy @arg.quTy)
  | Quant  lhs.quTy = Ty_Quant @qu.self @tv @ty.quTy
```

concluding with wrapping the AG functionality in the function *tyQuantify* which can be used in the Haskell world:

EHTyQuantify

```
tyQuantify :: (TyVarId → Bool) → Ty → Ty
tyQuantify tvIsBound ty
  = let t = wrap_TyAGItf
              (sem_TyAGItf (TyAGItf_AGItf ty))
              (Inh_TyAGItf {tvIsBound_Inh_TyAGItf = tvIsBound})
    in  quTy_Syn_TyAGItf t
```

5.5 Type inference

Type inferencing for this version of EH and the previous version are very similar. Figure 16 holds the adapted rules for expressions, figure 17 for patterns. The main differences are as follows:

- All rules are passed an additional context parameter indicating the way $\leqslant$ has to be done with respect to strength ν. See page 59 for the relevant discussion.
- The ν is mostly passed on unchanged, except in the argument of an expression application (rule e-app4) and a pattern *Con* (rule p-con4). The latter is due to a different way of handling tuple constructors. Instantiation in a pattern rule p-con4 instantiates as greedily as possible. TBD: needs more discussion
- Types of tuple constructors (and destructors) are now stored in the Γ for types, *valGam*. The lookup for types in *valGam* (*valGamLookup*) now takes care of returning a proper quantified type for tuples, thereby resembling more the normal retrieval of types from a Γ. The rule p-var4

$$\boxed{\nu, \Gamma, \sigma^k \overset{expr}{\vdash} e : \sigma \leadsto \mathcal{C}}$$

$$\frac{\begin{gathered}\nu_{il}, \Gamma, \sigma^a \overset{expr}{\vdash} e_2 : \_ \leadsto \mathcal{C}_2 \\ \nu, \Gamma, v \rightarrow \sigma^k \overset{expr}{\vdash} e_1 : \sigma^a \rightarrow \sigma \leadsto \mathcal{C}_1 \\ v \text{ fresh}\end{gathered}}{\nu, \Gamma, \sigma^k \overset{expr}{\vdash} e_1\ e_2 : \mathcal{C}_2 \sigma \leadsto \mathcal{C}_{2..1}} \quad \text{(e-app4)}$$

$$\frac{\begin{gathered}\nu, \Gamma^p \mathbin{+\!\!+} \Gamma, \sigma^r \overset{expr}{\vdash} e : \sigma^e \leadsto \mathcal{C}_3 \\ \nu, \Gamma, \sigma^p \overset{pat}{\vdash} p : \_, \Gamma^p \leadsto \mathcal{C}_2 \\ \nu \overset{fit}{\vdash} v_1 \rightarrow v_2 \leqslant \sigma^k : \sigma^p \rightarrow \sigma^r \leadsto \mathcal{C}_1 \\ v_i \text{ fresh}\end{gathered}}{\nu, \Gamma, \sigma^k \overset{expr}{\vdash} \lambda p \rightarrow e : \mathcal{C}_3 \sigma^p \rightarrow \sigma^e \leadsto \mathcal{C}_{3..1}} \quad \text{(e-lam4)}$$

$$\frac{\begin{gathered}(\xi \mapsto \sigma) \in \Gamma \\ \nu \overset{fit}{\vdash} \sigma \leqslant \sigma^k : \sigma \leadsto \mathcal{C}\end{gathered}}{\nu, \Gamma, \sigma^k \overset{expr}{\vdash} \xi : \sigma \leadsto \mathcal{C}} \quad \text{(e-ident4)}$$

$$\frac{\begin{gathered}\nu, \Gamma^q \mathbin{+\!\!+} \Gamma, \sigma^k \overset{expr}{\vdash} e : \sigma^e \leadsto \mathcal{C}_3 \\ \Gamma^q \equiv map\ (\lambda(n, \sigma) \rightarrow (n, inst^{\exists}\ (\sigma)))\ \Gamma^{p'} \\ \mathsf{ftv}\ (\mathcal{C}_{2..1}\Gamma), \oplus \overset{quGam}{\vdash} \Gamma^p : \Gamma^{p'} \\ \nu_s, \Gamma^p \mathbin{+\!\!+} \Gamma, \sigma^p \overset{expr}{\vdash} e^i : \_ \leadsto \mathcal{C}_2 \\ \nu_s, \Gamma, \mathbb{I} \overset{pat}{\vdash} p : \sigma^p, \Gamma^p \leadsto \mathcal{C}_1\end{gathered}}{\nu, \Gamma, \sigma^k \overset{expr}{\vdash} \mathbf{let}\ p = e^i \mathbf{in}\ e : \sigma^e \leadsto \mathcal{C}_{3..1}} \quad \text{(e-let4)}$$

$$\frac{\begin{gathered}\nu, (\Gamma^q - [i \mapsto \_] \mathbin{+\!\!+} [i \mapsto \sigma^q]) \mathbin{+\!\!+} \Gamma, \sigma^k \overset{expr}{\vdash} e : \sigma^e \leadsto \mathcal{C}_3 \\ \Gamma^q \equiv map\ (\lambda(n, \sigma) \rightarrow (n, inst^{\exists}\ (\sigma)))\ \Gamma^{p'} \\ \mathsf{ftv}\ (\mathcal{C}_{2..1}\Gamma), \oplus \overset{quGam}{\vdash} \Gamma^p : \Gamma^{p'} \\ \nu_s, (\Gamma^p - [i \mapsto \_] \mathbin{+\!\!+} [i \mapsto \sigma^q]) \mathbin{+\!\!+} \Gamma, \sigma^q \overset{expr}{\vdash} e^i : \_ \leadsto \mathcal{C}_2 \\ [\,], \oplus \overset{qu}{\vdash} \sigma^i : \sigma^q \leadsto \_ \\ p \equiv i \vee p \equiv i\ @... \\ \nu_s, \Gamma, \sigma^i \overset{pat}{\vdash} p : \_, \Gamma^p \leadsto \mathcal{C}_1\end{gathered}}{\nu, \Gamma, \sigma^k \overset{expr}{\vdash} \mathbf{let}\ i :: \sigma^i ; p = e^i \mathbf{in}\ e : \sigma^e \leadsto \mathcal{C}_{3..1}} \quad \text{(e-let-tysig4)}$$

$$\frac{\nu \overset{fit}{\vdash} Int \leqslant \sigma^k : \sigma \leadsto \mathcal{C}}{\nu, \Gamma, \sigma^k \overset{expr}{\vdash} minint\,..\,maxint : \sigma \leadsto \mathcal{C}} \quad \text{(e-int4)}$$

Figure 16: Type checking/inferencing for expression

$$\boxed{\nu, \Gamma, \sigma^k \overset{pat}{\vdash} p : \sigma, \Gamma^p \leadsto \mathcal{C}}$$

$$\frac{\begin{array}{c} \nu \overset{fit}{\vdash} \mathcal{C}_1 \sigma^k \leqslant \sigma^d : \sigma \leadsto \mathcal{C}_2 \\ \sigma^d \rightarrow () \equiv \sigma^p \\ \nu, \Gamma, \_ \overset{pat}{\vdash} p : \sigma^p, \Gamma^p \leadsto \mathcal{C}_1 \\ p \equiv p_1\ p_2\ ...\ p_n, n \geqslant 1 \end{array}}{\nu, \Gamma, \sigma^k \overset{pat}{\vdash} p : \sigma, \Gamma^p \leadsto \mathcal{C}_{2..1}} \quad \text{(p-apptop4)}$$

$$\frac{\begin{array}{c} dom\ (\Gamma^p_1) \cap dom\ (\Gamma^p_2) = \emptyset \\ \nu, \Gamma, \sigma^a_1 \overset{pat}{\vdash} p_2 : \_, \Gamma^p_2 \leadsto \mathcal{C}_2 \\ \nu, \Gamma, \_ \overset{pat}{\vdash} p_1 : \sigma^d \rightarrow (\sigma^a_1, \sigma^a_2, ..., \sigma^a_n), \Gamma^p_1 \leadsto \mathcal{C}_1 \end{array}}{\nu, \Gamma, \_ \overset{pat}{\vdash} p_1\ p_2 : \mathcal{C}_2(\sigma^d \rightarrow (\sigma^a_2, ..., \sigma^a_n)), \Gamma^p_1 + \!\!\!+\ \Gamma^p_2 \leadsto \mathcal{C}_{2..1}} \quad \text{(p-app4)}$$

$$\frac{\sigma \equiv inst^{\exists}\ (\sigma^k)}{\nu, \Gamma, \sigma^k \overset{pat}{\vdash} i : \sigma, [i \mapsto \sigma] \leadsto [\,]} \quad \text{(p-var4)} \qquad \frac{\begin{array}{c} (unI \mapsto \sigma^u) \in \Gamma \\ \nu_i \overset{fit}{\vdash} \sigma^u \leqslant v_1 \rightarrow v_2 : \sigma \leadsto \_ \\ v_i \text{ fresh} \end{array}}{\nu, \Gamma, \_ \overset{pat}{\vdash} I : \sigma, [\,] \leadsto [\,]} \quad \text{(p-con4)}$$

Figure 17: Type checking/inferencing for pattern

now covers the case for (tuple)constructors too. This change also prepares for the introduction of datatypes in the next version of EH.

- A **let**-expression in rule e-let4 and rule e-let-tysig4 quantify bindings via the rules in figure 15 and figure 14. Additionaly, to take care of always opening existentially quantified types bound by a value identifier, a function $inst^{\exists}$ is used. Function $inst^{\exists}$ corresponds to *tyInst1Exists*.

Changes in the implementation are also small, mostly to take care of the additional parameters to *fitsIn* (v, a *UID* for instantiations) and the previous remarks.

5.5.1 Handling of tuples

The alternative for *Con* looks up the value associated with the tuple constructor name in *valGam*.

EHInferExpr

```
SEM Expr
  | Con loc.(gTy, nmErrs) := case valGamLookup @nm @lhs.valGam of
                                 Nothing → (Ty_Any, [Err_NamesNotIntrod [@nm]])
                                 Just vgi → (vgiTy vgi, [ ])
        .fTy              := @lhs.tyCnstr ≻ @gTy
        .fo               := fitsIn @lhs.fiOpts @lUniq2 @fTy (@lhs.tyCnstr ≻ @lhs.knTy)
```

Previously, the type was constructed in situ, now it is delegated to *valGamLookup*:

EHGam

```
valGamLookup :: HsName → ValGam → Maybe ValGamInfo
valGamLookup nm g
  = case gamLookup nm g of
      Nothing
         | hsnIsProd nm
             → let pr = mkPr nm in mkRes (tyProdArgs pr ‘mkTyArrow‘ pr)
         | hsnIsUn nm ∧ hsnIsProd (hsnUnUn nm)
             → let pr = mkPr (hsnUnUn nm) in mkRes ([pr] ‘mkTyArrow‘ pr)
        where mkPr nm = mkTyFreshProd (hsnProdArity nm)
              mkRes t  = Just (ValGamInfo (tyQuantifyClosed t))
      Just vgi → Just vgi
      _        → Nothing
```

This modification also introduces a new convention where *valGam* contains for a value constructor *X* a binding for the type of the function which constructs the value, and a type of the function which dissects the value into a tuple of all fields of the value. The convention is that the constructor has name *X*, the dissector/deconstructor has name *unX*. For tuples these bindings are created on the fly. For example,for a 3-tuple the following bindings are simulated to be present in *valGam*:

```
,3   :: ∀ a.a → ∀ b.b → ∀ c.c → (a, b, c)
un,3 :: ∀ a b c.(a, b, c) → (a, b, c)
```

The *unX* binding corresponds to the type created in the rule p-con2 (figure 10, page 41). The *Con* alternative now also uses the *valGam* to find a binding for a tuple dissector/destructor:

EHInferPatExpr

```
SEM PatExpr
  | Con loc.(knUnTy, nmErrs) = case valGamLookup (hsnUn @nm) @lhs.valGam of
                                   Nothing → (Ty_Any, [Err_NamesNotIntrod [@nm]])
                                   Just vgi → (vgiTy vgi, [ ])
        .patFunTy          := let [a, r] = mkNewTyVarL 2 @lUniq
                                  fo     = fitsIn v_i @lUniq2 @knUnTy ([a] ‘mkTyArrow‘ r)
                              in  foTy fo
```

5.5.2 Declarations and options

Declarations als need some modifications to take care of the quantification and instantiation of toplevel existential quantifiers as specified in rule e-let4 and rule e-let-tysig4:

EHInfer

```
SEM Expr
  | Let decls.patValGam  := gamPushGam (valGamInst1Exists @lUniq @decls.gathTySigGam)
                                       @lhs.valGam
        loc  .lQuValGam := valGamInst1Exists @lUniq2.valGamQuantify @gTyTvL $ @lSubsValGam
```

```
ATTR AllExpr AllPatExpr [fiOpts : FIOpts || ]
SEM Expr
  | App  func     .fiOpts = $v_s$
         arg      .fiOpts = $v_{il}$
SEM Decl
  | Val  expr     .fiOpts = $v_s$
         patExpr.fiOpts = $v_s$
SEM AGItf
  | AGItf expr    .fiOpts = $v_s$
```

Setting up proper values for the "strength" v is also done here.

5.5.3 Type expressions

Type signatures may include quantifiers. This requires additional abstract syntax for type expressions:

EHAbsSyn

```
DATA TyExpr
  | Quant qu      : {TyQu}
          tyVar   : {HsName}
          tyExpr : TyExpr
```

and additional parsing

EHParser

```
pTyExprPrefix = sem_TyExpr_Quant
                    <$>      (TyQu_Forall <$ pKey "forall" <|> TyQu_Exists <$ pKey "exists")
                    <*>      pVar <* pKey "."
```

EHParser

```
tyExprAlg     = (sem_TyExpr_Con, sem_TyExpr_App
                ,sem_TyExpr_AppTop, sem_TyExpr_Parens)
pTyExprBase = sem_TyExpr_Con  <$> pCon
            <|> sem_TyExpr_Var   <$> pVar
            <|> sem_TyExpr_Wild  <$ pKey "..."
            <|> pParenProd tyExprAlg pTyExpr
pTyExpr       = pTyExprPrefix <*> pTyExpr
            <|> pTyExprBase <??> (flip (mkArrow tyExprAlg) <$ pKeyw hsnArrow <*> pTyExpr)
```

The parser *pTyExpr* is slightly complicated because of the right associativity of the function type constructor $\rightarrow$ in combination with quantifiers. For example, the type

$$\forall\ a.a \rightarrow \forall\ b.b \rightarrow (a, b)$$

parses to an abstract syntax fragment corresponding to

$$\forall\ a.(a \rightarrow (\forall\ b.(b \rightarrow (a, b))))$$

Rewriting to a form similar to the parser for expressions, with a prefix would lead to a parser with common prefixes (the *pTyExprBase*) in its alternatives. For LL(k) parsers such as the parser combinators used here this is not a good idea. Hence the construction where the quantifier is parsed as a prefix of the parts between → but still applies right associatively.

TBD: previous should be redone.

TBD: until here

EHInferExpr

```
SEM Expr
  | Var  loc.fTy      := @lhs.tyCnstr ≻ @gTy
  | IConst CConst Var
         loc.fo       := fitsIn @lhs.fiOpts @lUniq @fTy (@lhs.tyCnstr ≻ @lhs.knTy)
  | Lam loc.foKnFun := let fo = fitsIn @lhs.fiOpts @lUniq2 @funTy (@lhs.tyCnstr ≻ @lhs.knTy)
                       in  fo{foTy = foCnstr fo ≻ @funTy}
```

EHInferPatExpr

```
SEM PatExpr
  | AppTop Con loc.fo             := fitsIn @lhs.fiOpts @lUniq @lhs.knTy @knResTy
  | Con        loc.(knResTy, _)  = tyArrowArgRes @patFunTy
                  .ty             := foTy @fo
               lhs.tyCnstr        = foCnstr @fo ≻ @lhs.tyCnstr
  | Var VarAs  loc.ty             := tyInst1Exists @lUniq2 (tyEnsureNonBotTop @lUniq @lhs.knTy)
```

5.6 Literature

TBD:

Higher ranked types, [?, ?]

Cannot do inference for rank3, [?, ?, ?, ?]

Existentials, via universal [?]

6 EH 5: data types

This part is not included in this version of this paper.

7 EH 6: kind inference

This part is not included in this version of this paper.

8 EH 7: non extensible records

8.1 Current implementation examples

```
let id :: a → a
    id = λx → x
    v = (b = 'x', c = id, a = id 4)
    v :: ((a :: Int, b :: Char) | c :: a → a)
    v₂ = (3, 4)
```

$v_3 :: (\mathit{Int}, \mathit{Char})$
$v_3' = \textbf{case}\ v_3\ \textbf{of}$
$\quad (a, b) \rightarrow (b, a)$
$v_4 :: (a :: a, f :: a \rightarrow \mathit{Int})$
$v_4 = (a = 3, f = \mathit{id})$
$\mathit{v4fa} = v_4.f\ v_4.a$
$v_5 :: \mathit{Rec}\ (\mid c :: \mathit{Int} \mid)$
$v_6 :: (<c :: \mathit{Int}>)$
$\mathit{v6c} = v_6.c$
$\mathit{vs} = v.c\ v.a$
$\mathit{vc} = \textbf{case}\ v\ \textbf{of}$
$\quad (a = \mathit{aa}, b = \mathit{bb}, c) \rightarrow (c\ \mathit{aa}, c\ \mathit{bb})$
$\textbf{in}\ \mathit{vs}$

with output

```
let id :: a -> a
    id = \x -> x
    v = (b = 'x',c = id,a = id 4)
    v :: (a :: Int,b :: Char,c :: a -> a)
    v2 = (3,4)
    v3 :: (Int,Char)
    v3' = case v3 of
            (a,b) -> (b,a)
    v4 :: (a :: a,f :: a -> Int)
    v4 = (a = 3,f = id)
    v4fa = v4.f v4.a
    v5 :: (c :: Int)
    v6 :: (<c :: Int>)
    v6c = v6.c
    {- ***ERROR(S):
         In `v6.c':
           In `v6':
             Type clash:
               failed to fit: (<c :: Int>) <= (v_116_0|)
               problem with : Var <= Rec
           Missing label in row:
             Label: c
             Row  : ?? -}
    vs = v.c v.a
    vc = case v of
           (a = aa,b = bb,c = c) -> (c aa,c bb)
    {- [  ] -}
    {- [ vc:(Int,Char), vs:Int, v6c:forall a . a, v4fa:Int
       , v3':(Char,Int), v2:(Int,Int), v6:(<c :: Int>), v5:(c :: Int)
       , v4:(a :: C_0_3_0,f :: C_0_3_0 -> Int), v3:(Int,Char)
       , v:(a :: Int,b :: Char,c :: forall a . a -> a)
       , id:forall a . a -> a ] -}
in vs
```

8.2 Choices already made

Syntax: delimiters. The notation for tuples with parenthesis (...) is reused, no curly braces are used {...}.

- Curly braces are used for an alternative way of specifying list usually speficied via the layout rule (declarations, case alternatives). Especially the case alternative would lead to parsing problems with the matching of records.
- Tuples now automatically are records with a numerical selection as field selection

```
let r = (i = 3, 'x')
in  let vc = r.2
        vi = r.i
    in  vc
```

Rows themselves are delimited by (|...|), variants by (< ... >).

A row/record is always based on (or an extension of) an empty row/record. For a following version allowing extensible records this restriction (obviously) is removed.

Case expression, punning. Pattern matching a record is done via case:

```
let r = (p = 3, q = 'x')
in  let v1 = case r of
                (p = a, q = b) → a
        v2 = case r of
                (p, q = b) → p
        v3 = case (3, 'x') of
                (p, q) → p
        v4 = case r of
                (p, q) → p
        v5 = case (3, 'x') of
                (1 = p, q) → p
    in  v1
```

In these examples, the matching for v_1 is done on the basis of field names at the left of =. The corresponding field value is made available via the name at the right of =. If the latter name is omitted, so called punning takes place where both names are base on the single identifier occurring on a field position. The second example v_2 demonstrates this feature.

However, for tuples we run into a problem if we try to pun because the names for fields of a tuple default to $[1\,..]$. Tuples now would require explicit field name matching whereas (for compatibility/convenience reasons) the use as with v_3 is desirable. To fix this a somewhat arguable choice has been made:

- If at least one field is matched via explicit field matching (as in v_2) punning is done for the fields not matched via an explicit field match.
- If no field is matched explicitly, all fields are implicitly matched on their positional label, that is $[1\,..]$.

This design decision causes v_4 and v_5 to produce error messages:

```
let r = (p = 3,q = 'x')
    {- [  ] -}
    {- [ r:(p :: Int,q :: Char) ] -}
in let v1 = case r of
              (p = a,q = b) -> a
       v2 = case r of
              (p = p,q = b) -> p
       v3 = case (3,'x') of
              (p,q) -> p
       v4 = case r of
              (p,q) -> p
       {- ***ERROR(S):
            In `(p,q) -> p':
              In `(p,q)':
                Missing label in row:
                  Label: q
                  Row  : (|v_64_0,v_63_0|) -}
       v5 = case (3,'x') of
              (p,q = q) -> p
       {- ***ERROR(S):
            In `(p,q = q) -> p':
              In `(p,q = q)':
                Missing label in row:
                  Label: 2
                  Row  : (|v_81_0,q :: v_80_0|) -}
       {- [  ] -}
       {- [ v5:forall a . a, v4:forall a . a, v3:Int, v2:Int, v1:Int ] -}
   in v1
```

Relation with other proposals. For the rest the proposal by Jones [**?**] is followed.

This proposal also deviates from TRex (by extending from left to right) and is incompatible with Haskell because of the use of `'.'` as the builtin selection operator.

8.3 Issues, partly resolved or previous

A AG pattern: global variable simulation

This part is not included in this version of this paper.

B AG pattern: unique value generation

This part is not included in this version of this paper.

C AG pattern: gathering

This part is not included in this version of this paper.

D AG pattern: gathering from within and usage higher up

This part is not included in this version of this paper.

E EH: Additional checks

EHExtraChecks

```
gamToDups :: Gam HsName v → [HsName]
gamToDups g = [n | ns @(n : _) ← group.sort.map fst.gamToAssocL $ g, length ns > 1]

SEM Expr
  | Let loc.dupErrs = let nms = gamToDups @lValGam
                      in  if null nms then [] else [Err_NamesDupIntrod nms]

SEM Decl
  | Val loc.sigMissErrs = if @hasTySig then [] else [Err_MissingSig @patExpr.pp]
```

F EH: Missing glue

This part is not included in this version of this paper.

G EH: Connecting with the outside world

This part is not included in this version of this paper.