

# Typing Haskell with an Attribute Grammar (Part I)

*Atze Dijkstra*  
*Doaitse Swierstra*

institute of information and computing sciences, utrecht university

technical report UU-CS-2004-037

[www.cs.uu.nl](http://www.cs.uu.nl)

# Preface

## Type systems for functional languages (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.

## Context of this paper

A previous version of this paper has been presented and distributed at the AFP2004 summerschool<sup>1</sup>. This paper describes a part of a Haskell compiler under development [8], focussing on the type system of the language and its implementation. Subsequent papers will describe the remaining parts of the implementation.

Not all parts of the implementation are explained in this paper. In a subsequent paper (continuing with part II) for example data structures (**data** types, kind inference and records) will be introduced and explained. As a consequence at some points in this paper a forward reference will be made to material to be published later. This concerns mainly future design decisions which have an influence on design decisions made in this paper.

---

<sup>1</sup>The proceedings for the AFP2004 summerschool have yet to appear.



# Contents

<b>I</b>	<b>Type checking, inference and polymorphism</b>	<b>7</b>
<b>1</b>	<b>Introduction and overview</b>	<b>9</b>
1.1	Purpose . . . . .	9
1.2	A short tour . . . . .	10
1.3	Haskell language elements not described . . . . .	12
1.4	About the presented code . . . . .	13
<b>2</b>	<b>EH 1: typed <math>\lambda</math>-calculus</b>	<b>15</b>
2.1	Concrete and abstract syntax . . . . .	16
2.2	Parsing: from concrete to abstract (syntax) . . . . .	19
2.3	Pretty printing: from abstract to concrete . . . . .	22
2.4	Types . . . . .	25
2.4.1	What is a type . . . . .	25
2.4.2	Type pretty printing . . . . .	26
2.5	Checking types . . . . .	27
2.5.1	Type rules . . . . .	28
2.5.2	Environment . . . . .	29
2.5.3	Checking Expr . . . . .	30
2.5.4	Checking PatExpr . . . . .	35
2.5.5	Checking declarations . . . . .	37
2.5.6	Checking TyExpr . . . . .	39
2.6	Reporting program errors . . . . .	39
2.7	Tying it all together . . . . .	40
<b>3</b>	<b>EH 2: monomorphic type inference</b>	<b>41</b>
3.1	Type variables . . . . .	42
3.2	Constraints . . . . .	43
3.2.1	Remembering and applying constraints . . . . .	43
3.2.2	Computing constraints . . . . .	44
3.3	Reconstructing types for Expr . . . . .	46
3.4	Reconstructing types for PatExpr . . . . .	49
3.5	Declarations . . . . .	51
3.6	Partial type signatures: a test case for extendibility . . . . .	51
<b>4</b>	<b>EH 3: polymorphic type inference</b>	<b>53</b>
4.1	Type language . . . . .	55
4.2	Type inference . . . . .	56
4.2.1	Instantiation . . . . .	56
4.2.2	Quantification . . . . .	58
4.2.3	Generalization/quantification of inferred types . . . . .	59
4.2.4	Type expressions . . . . .	61
<b>5</b>	<b>EH 4: <math>\forall</math> and <math>\exists</math> everywhere</b>	<b>63</b>

5.1	Type language . . . . .	66
5.2	Fitting, subsumption . . . . .	67
5.2.1	Issues . . . . .	67
5.2.2	Implementation . . . . .	70
5.3	Instantiation . . . . .	75
5.4	Quantification . . . . .	76
5.5	Type inference . . . . .	79
5.5.1	Handling of tuples . . . . .	82
5.5.2	Declarations and options . . . . .	83
5.5.3	Type expressions . . . . .	83
	<b>References</b>	<b>84</b>
	<b>Index</b>	<b>86</b>

# List of Figures

2.1	Abstract syntax for EH (without Expr) . . . . .	17
2.2	Parser combinators . . . . .	20
2.3	Pretty printing combinators . . . . .	23
2.4	Type checking for expression . . . . .	28
2.5	Rules for fit . . . . .	30
2.6	Type checking for expression (checking variant) . . . . .	31
2.7	Type checking for let-expression with pattern . . . . .	35
2.8	Building environments from patterns . . . . .	36
3.1	Type inferencing for expressions (using constraints) . . . . .	46
3.2	Type inferencing for pattern (using constraints) . . . . .	49
4.1	Type inferencing for expressions with quantifier $\forall$ . . . . .	57
5.1	Fitting/subsumption for type applications . . . . .	71
5.2	Fitting/subsumption for quantified types . . . . .	71
5.3	Quantifier location inferencing . . . . .	76
5.4	Quantifier location inferencing for types in a Gamma . . . . .	77
5.5	Type checking/inferencing for expression . . . . .	80
5.6	Type checking/inferencing for pattern . . . . .	81



## **Part I**

# **Type checking, inference and polymorphism**





# Chapter 1

## Introduction and overview

Haskell98[25] is a complex language, not to mention its more experimental incarnations. Though also intended as a research platform, realistic compilers for Haskell [1] have grown over the years and understanding 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 [14], 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<sup>1</sup> 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.
- 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 [3] to separately describe aspects of the language implementation, and parser combinators [33, 34] to compactly describe executable syntax. Other

---

<sup>1</sup>The 'E' in EH might also be expanded to other aspects of the compiler, like being an Example

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 [26, 4, 22] and existentials [24, 19, 21]. 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 [10, 9] and more implementation oriented [15, 27] or similar to [28].
- 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 [3, 32]
- Type systems,  $\lambda$ -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, and in the future perhaps by integrated environments [30].

In the following sections we give examples of the Haskell features present in the series of compilers described in the following chapters. Only short examples are given, 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  $\lambda$ -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  $\lambda$ -calculus, in particular typed  $\lambda$ -calculus extended with **let** expressions and some basic types like *Int*, *Char* and tuples.

**EH version 1:  $\lambda$ -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, chapter 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, chapter 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  $i :: \text{Int}$ .

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  $id :: \text{Int} \rightarrow \text{Int}$  will be reconstructed.

**EH version 3: Polymorphism.** The third version (EH version 3, chapter 4 ) performs standard Hindley-Milner type inference [5, 6] which infers also parametric polymorphism. For example,

```
let id = λx → x
in id 3
```

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

A type for a value can also be specified explicitly

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

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 [13, 16, 17, 18], so the fourth version (EH version 4, chapter 5) does not infer this type information but allows explicitly specified polymorphism for (e.g.) parameters.

For example, the following is allowed.

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

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:

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

```
infers f :: (∀ a.a → a) → (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*.

```
let id :: ... → ...
    id = λx → x
    f :: (a → a) → ...
    f = λi → (i 3, i 'x')
in f id
```

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

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

```
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
    xy' = ixy xy
    pq :: ∃ a.(a, a → Int)
    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  $xy :: \exists a.(a, a \rightarrow Int)$ .

### 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 [7, 31].
- 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 [8].



## Chapter 2

# EH 1: typed $\lambda$ -calculus

In this section we will build the first version of our series of compilers: the typed  $\lambda$ -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 2.1). 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

```

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 AGlTf
  | AGlTf expr : Expr

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

TYPE Decls = [Decl]
SET AllDecl = Decl Decls

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

SET AllPatExpr = PatExpr

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

SET AllTyExpr = TyExpr

SET AllExpr = Expr

SET AllNT = AllTyExpr AllDecl
    AllPatExpr AllExpr

```

---

Figure 2.1: Abstract syntax for EH (without Expr)

Looking at this example and the rest of the abstract syntax in figure 2.1 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<sup>1</sup>.
- 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 chapter 5 , this turns out to be convenient. This and other naming conventions are available via the following definitions for Haskell names *HsName*.

```
data HsName = HNm String
            deriving (Eq, Ord)

instance Show HsName where
    show (HNm s) = s
```

EHCommon

EHCommon

<sup>1</sup>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 :: HsName
hsnArrow           = HNm "->"
hsnUnknown         = HNm "???"
hsnInt             = HNm "Int"
hsnChar            = HNm "Char"

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.
- *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 *ICnst* for an *Int* constant, and *Lam* for a  $\lambda$ -expression.
- Groups of nonterminals may be given a name through the **SET** directive. For example, *AllINT* (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 *AllINT* 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 [32] and figure 2.2) 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_ICnst <$> pInt
           <<> sem_Expr_CConst <$> 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 [32] 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 from 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 [35]).

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

Figure 2.2: Parser combinators

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

EHParse

```

exprAlg    = (sem_Expr_Con, sem_Expr_App
              , sem_Expr_AppTop)
pExpr      = pExprPrefix (*) pExpr
              (<|>) pExprApp
pExprPrefix = sem_Expr_Let    ($) pKey "let"
              (*) pDecls      (* pKey "in"
              (<|>) sem_Expr_Lam ($) pKey "\\\"
              (*) pPatExprBase (* pKey "->"
pExprApp    = pApp exprAlg pExprBase

```

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  $\lambda$ -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  $\langle * \rangle$ .

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

$$pApp\ alg\ p = mkApp\ alg\ \langle \$ \rangle\ pList1\ p$$

First, let us look at the algebra  $exprAlg$ , again referring to Swierstra [35] for a more thorough treatment. The abstract syntax of EH (see figure 2.1, page 17) 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  $f \equiv Con\ ",2"$ , and for a function type  $f \equiv Con\ "->"$  (representing  $x \rightarrow y$ ). 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.

**type**  $MkConAppAlg\ t = (HsName \rightarrow t, t \rightarrow t \rightarrow t, t \rightarrow t)$  EHCommon

$mkApp :: MkConAppAlg\ t \rightarrow [t] \rightarrow t$

$mkApp\ (\_, app, top)\ ts$

= **case**  $ts$  **of**

$[t] \rightarrow t$

$- \rightarrow top\ (foldl1\ app\ ts)$  EHCommon

$mkArrow :: MkConAppAlg\ t \rightarrow t \rightarrow t \rightarrow t$

$mkArrow\ alg\ @(con, -, -)\ a\ r = mkApp\ alg\ [con\ hsnArrow, a, r]$  EHCommon

$mkConApp :: MkConAppAlg\ t \rightarrow HsName \rightarrow [t] \rightarrow t$

$mkConApp\ alg\ @(con, -, -)\ c\ ts = mkApp\ alg\ (con\ c : ts)$  EHCommon

$mkProdApp :: MkConAppAlg\ t \rightarrow [t] \rightarrow t$

$mkProdApp\ alg\ ts = mkConApp\ alg\ (hsnProd\ (length\ ts))\ ts$

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  $(pE, pE, \dots)$  of elements  $pE$ .

$pParenProd\ alg\ pE$

=  $pParens\ pP$

**where**

$pP = mkProdApp\ alg\ \langle \$ \rangle\ pSucceed\ []$

$\langle \rangle\ pE$

$\langle ?? \rangle\ ( (\lambda es\ e \rightarrow mkProdApp\ alg\ (e : es))$

$\langle \$ \rangle\ pList1\ (pComma\ *)\ pE)$

$)$

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

$tyExprAlg = (sem\_TyExpr\_Con, sem\_TyExpr\_App,$   
 $\quad\quad\quad sem\_TyExpr\_AppTop)$

$pTyExprBase = sem\_TyExpr\_Con\ \langle \$ \rangle\ pCon$

$\langle \rangle\ pParenProd\ tyExprAlg\ pTyExpr$

$pTyExpr = pChainr$

$(mkArrow\ tyExprAlg\ \langle \$ \rangle\ pKeyw\ hsnArrow)$

### *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  $\rightarrow$ .

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)
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*

**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)
                >-< @errDeclsPP
                >-< "in" >#< @body.pp
                >-< @errBodyPP
                .ppExtra = ppCmt (ppGam @lValGam)
| App     loc.pp = pp_parens @func.pp
                >#< pp_parens @arg.pp
| AppTop loc.pp = ppParNeed @parNeed @lhs.parNeed
                (ppAppTop (@expr.appFunNm, @expr.appFunPP)

```

EHPretty

EHPretty

```

| Lam   loc.pp      = ppParNeed ParNeeded @lhs.parNeed
                    (@expr.appArgPPL)
                    ("\" >||< @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 <i>PP.Doc</i>
$pp\ x$	pretty print $x$ (assuming instance <i>PP x</i> ) resulting in a <i>PP.Doc</i>

Figure 2.3: Pretty printing combinators

The computation of the *pp* attribute is straightforward in the sense that only a few basic combinators are needed (see [32] and figure 2.3). The additional complexity arises from

- The encoding for applications. Straightforward printing prints as  $(2, 3)$  as  $, 2\ 2\ 3$ .
- Avoiding unnecessary parentheses. Without parentheses ‘ $f\ (f\ 2\ (3, 4))\ 5$ ’ would print as ‘ $f\ f\ 2\ (3, 4)\ 5$ ’ whereas incorporation of explicit parentheses to clarify the priority of all applications, would print as ‘ $(f\ (f\ (2\ (((3), (4))))\ (5)))$ ’.

The first problem is solved at the *AppTop* alternative by gathering all arguments (here:  $[2, 3]$ ) for a function (here: tuple constructor  $, 2$ , or  $(, )$ ) into *appArgPPL*:

```
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
| Var AppTop IConst CConst Lam
    lhs.appFunNm = hsnUnknown
| Var Con AppTop IConst CConst Lam
    lhs.appArgPPL = []
| Var Con AppTop IConst CConst Lam
    lhs.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*

```

ppAppTop :: PP arg => (HsName, arg) -> [arg] -> PP.Doc
ppAppTop (conNm, con) args
= if hsnIsArrow conNm then ppListSep "" "" (" " ++ show hsnArrow ++ " ") args
  else if hsnIsProd conNm then ppListSep "(" ")" " " ", " args
    else ppListSep "" "" " " (con : args)

```

EHPretty

EHCommon

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
ppCon :: HsName -> PP_Doc
ppCon nm = if hsnIsProd nm
           then pp_parens (text (replicate (hsnProdAry 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.

The second problem, avoiding unnecessary parentheses, 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.

EHPretty

```

ATTR Expr [parNeed : ParNeed parNeedL : ParNeedL ||]
SEM Expr
  | App (arg.parNeed, func.parNeedL) = hdAndTl @lhs.parNeedL
  | Lam body.parNeed                = ParNotNeeded
  | AppTop (loc.parNeed, expr.parNeed, expr.parNeedL)
           = parNeedApp @expr.appFunNm

SEM Decl
  | Val expr.parNeed                = ParNotNeeded
    .parNeedL                       = []

SEM AGItf
  | AGItf expr.parNeed              = ParNotNeeded
    .parNeedL                       = []

```

together with 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 an *Expr* 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 (encoded by *ParNotNeeded*, second element of result tuple). The list (third element of result tuple) is meant for the arguments of the application.

A few observations are in place here

- This mechanism may look slightly overcomplicated but is also used for patterns, type expressions and -later on- types too. In the case of types it is also necessary to take the right associativity of  $\rightarrow$  into account.
- 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.
- The pretty printed output is also merged with additional information, for example error messages and the binding of identifiers to types are also printed.

We will not look any further into these details as the code for patterns and type expressions is almost an exact replica of the code for expressions. The rest of the *pp* related attribute computations are 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.

### 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 [36, 37, 28, 2].

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  $\sigma$  can be

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

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

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

This definition also introduces the possibility of describing types like  $\text{Int Int}$ . 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  $\text{Int Int}$  are created.

The corresponding encoding using AG notation differs in the presence of an *Any* type. This type is also denoted by  $\perp$  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
```

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.

## 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 Ty [| pp : PP_Doc]
SEM Ty
  | Con loc.pp = ppCon @nm
  | App loc.pp = if @isSpineRoot
```

```

then ppParNeed @parNeed @lhs.parNeed
      (ppAppTop (@appFunNm, @appFunPP) @appArgPPL)
else @func.pp >#< pp_parens @arg.pp
| 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

**ATTR** *Ty* [| | *appFunNm* : *HsName*]

**SEM** *Ty*

```

| Con lhs.appFunNm = @nm
| App loc.appFunNm = @func.appFunNm
| Any lhs.appFunNm = hsnUnknown

```

**ATTR** *Ty* [*appSpinePos* : *Int* |]

**SEM** *TyAGIf*

```

| AGIf ty .appSpinePos = 0

```

**SEM** *Ty*

```

| App func.appSpinePos = @lhs.appSpinePos + 1
  arg .appSpinePos = 0
  loc .isSpineRoot = @lhs.appSpinePos ≡ 0

```

The remaining part, that is, finding out if and where parentheses are needed is similar to the way this was done before and is therefore omitted. What remains, is the way the functionality is made available to the Haskell world:

```

ppTy :: Ty → PP.Doc
ppTy ty
  = let t = wrap_TyAGIf
      (sem_TyAGIf (TyAGIf_AGIf ty))
      Inh_TyAGIf
  in pp_Syn_TyAGIf t

```

To make this work, we need to tell the AG system to generate a wrapper function and datatypes *Inh\_TyAGIf* and *Syn\_TyAGIf* with selector functions for defined attributes to pass inherited and synthesized attributes respectively:

**WRAPPER** *TyAGIf*

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 according to a fixed naming convention. An example of such a function is *pp\_Syn\_TyAGIf*, which when applied to *t* accesses the synthesized *pp* attribute at an attributed non-terminal *t* of type *TyAGIf*.

## 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 2.4) for function application

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

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

---

$\text{expr} \\ \Gamma \vdash e : \sigma$

  

$\frac{\begin{array}{c} \text{expr} \\ \Gamma \vdash e_2 : \sigma^a \\ \text{expr} \\ \Gamma \vdash e_1 : \sigma^a \rightarrow \sigma \end{array}}{\text{expr} \\ \Gamma \vdash e_1 e_2 : \sigma} \quad (\text{e-app1})$	$\frac{[i \mapsto \sigma^i] \text{++} \Gamma \vdash e : \sigma^e}{\text{expr} \\ \Gamma \vdash \lambda i \rightarrow e : \sigma^i \rightarrow \sigma^e} \quad (\text{e-lam1})$
$\frac{\begin{array}{c} \text{expr} \\ \Gamma \vdash e_2 : \sigma_2 \\ \text{expr} \\ \Gamma \vdash e_1 : \sigma_1 \end{array}}{\text{expr} \\ \Gamma \vdash (e_1, e_2) : (\sigma_1, \sigma_2)} \quad (\text{e-prod1})$	$\frac{[i \mapsto \sigma^i] \text{++} \Gamma \vdash e^i : \sigma^i}{[i \mapsto \sigma^i] \text{++} \Gamma \vdash e : \sigma^e} \quad (\text{e-let1})$
$\frac{(i \mapsto \sigma) \in \Gamma}{\text{expr} \\ \Gamma \vdash i : \sigma} \quad (\text{e-ident1})$	$\frac{}{\text{expr} \\ \Gamma \vdash \text{minint} \dots \text{maxint} : \text{Int}} \quad (\text{e-int1})$

---

Figure 2.4: Type checking for expression

All rules we will use are of the form

$$\frac{\begin{array}{c} \text{prerequisite}_1 \\ \text{prerequisite}_2 \\ \dots \end{array}}{\text{consequence}} \quad (\text{rule-name})$$

with the meaning that if all  $\text{prerequisite}_i$  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*:

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

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

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

or a piece of AG

$$\begin{aligned} \text{ATTR } \text{judgetype} & [\text{context} : \dots \mid \mid \\ &\quad \text{property} : \dots \text{more\_results} : \dots] \\ \text{SEM } \text{judgetype} & \\ | \text{construct} & \\ \text{lhs.}(\text{property}, \text{more\_results}) &= \dots @\text{lhs.context} \dots \end{aligned}$$

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 specifying 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 2.4 also refer to  $\Gamma$ , 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

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

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  $\Gamma$  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  $\Gamma$ . 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  $\vec{\cdot}$  will be used instead.

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

$$\begin{aligned} \text{type } \text{AssocL } k \ v &= [(k, v)] \\ \text{newtype } \text{Gam } k \ v &= \text{Gam } [\text{AssocL } k \ v] \\ \text{emptyGam} &:: \text{Gam } k \ v \\ \text{gamLookup} &:: \text{Eq } k \Rightarrow k \rightarrow \text{Gam } k \ v \rightarrow \text{Maybe } v \\ \text{gamPushNew} &:: \text{Gam } k \ v \rightarrow \text{Gam } k \ v \\ \text{gamPushGam} &:: \text{Gam } k \ v \rightarrow \text{Gam } k \ v \rightarrow \text{Gam } k \ v \\ \text{gamAdd} &:: k \rightarrow v \rightarrow \text{Gam } k \ v \rightarrow \text{Gam } k \ v \end{aligned}$$

EHCommon

EHGam

EHGam

```

emptyGam          = Gam [[]]
gamLookup         k (Gam ll) = foldr (\l 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  $\Gamma$ . Additions to a  $\Gamma$  will take place on the top of the stack only. A *gamUnit* used as an infix operator will print as  $\mapsto$ .

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

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 2.4 do not provide much information about how the type  $\sigma$  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.

---

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

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

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

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

$$\frac{}{\text{fit} \vdash \mathbb{I} \leq \sigma_2 : \sigma_2} \quad \text{(f-any-l1)}$$

$$\frac{}{\text{fit} \vdash \sigma_1 \leq \mathbb{I} : \sigma_1} \quad \text{(f-any-r1)}$$

---

Figure 2.5: Rules for fit

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

$$\frac{\Gamma, \sigma^a \stackrel{expr}{\vdash} e_2 : \_}{\Gamma, \mathbb{I} \rightarrow \sigma^k \stackrel{expr}{\vdash} e_1 : \sigma^a \rightarrow \sigma} \quad (\text{e-app1B}) \quad \frac{[i \mapsto \sigma^i] \# \Gamma, \sigma^r \stackrel{expr}{\vdash} e : \sigma^e}{\Gamma, \sigma^i \rightarrow \sigma^r \stackrel{expr}{\vdash} \lambda i \rightarrow e : \sigma^i \rightarrow \sigma^e} \quad (\text{e-lam1B})$$

$$\frac{\Gamma, \sigma_2^k \stackrel{expr}{\vdash} e_2 : \sigma_2}{\Gamma, \sigma_1^k \stackrel{expr}{\vdash} e_1 : \sigma_1} \quad (\text{e-prod1B}) \quad \frac{[i \mapsto \sigma^i] \# \Gamma, \sigma^i \stackrel{expr}{\vdash} e^i : \_}{[i \mapsto \sigma^i] \# \Gamma, \sigma^k \stackrel{expr}{\vdash} e : \sigma^e} \quad (\text{e-let1B})$$

$$\frac{(i \mapsto \sigma^i) \in \Gamma}{\stackrel{fit}{\vdash} \sigma^i \leq \sigma^k : \sigma} \quad (\text{e-ident1B}) \quad \frac{\stackrel{fit}{\vdash} Int \leq \sigma^k : \sigma}{\Gamma, \sigma^k \stackrel{expr}{\vdash} \text{minint} \dots \text{maxint} : \sigma} \quad (\text{e-int1B})$$

Figure 2.6: Type checking for expression (checking variant)

We now can tailor the type rules in figure 2.4 towards an implementation which performs type checking, in figure 2.6. The rules now take an additional context, the expected (or known) type  $\sigma^k$  (attribute  $knTy$ ) as specified by the programmer. An additional type of judgement  $fit$  (figure 2.5) 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  $\sigma^k$ . Both the definition of  $\sigma^k$  in terms of AG

**ATTR**  $Expr [knTy : Ty \ ||]$  EHInferExpr

initialized on top by

**SEM**  $AGItf$   
 $| AGItf \ expr.knTy = Ty\_Any$  EHInferExpr

and the implementation of the type rule e-int1B

**ATTR**  $Expr [|| \ ty : Ty]$  EHInferExpr

**SEM**  $Expr$   
 $| CConst \ \mathbf{loc}.fTy = tyChar$   
 $| IConst \ \mathbf{loc}.fTy = tyInt$   
 $| IConst \ CConst$   
 $\ \mathbf{loc}.fo = @fTy \leq @lhs.knTy$   
 $\ \ .ty = foTy @fo$

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

$tyInt = Ty\_Con \ hsnInt$   
 $tyChar = Ty\_Con \ hsnChar$  EHTy

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  $\mathbf{lhs}.knTy$  via the implementation



*fitsIn* of the rules for *fit*/ $\leq$ . In infix notation *fitsIn* prints as  $\leq$ . *fitsIn* returns a *FIOut* (*fitsIn* output) 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  $\vdash \overset{fit}{\sigma_1} \leq \sigma_2 : \sigma$  is implemented by *fitsIn*. The function *fitsIn* checks if a value of type  $\sigma_1$  can flow (that is, stored) into a memory location of type  $\sigma_2$ . This is an asymmetric relation because “a value flowing into a location” does not imply that it can flow the other way around, so  $\leq$ , 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 2.5 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 67 when  $\leq$  really behaves asymmetrically we will discuss this aspect of the rules which is named *contravariance*. In the rules in figure 2.5 the direction makes no difference; the correct use of the direction for now only anticipates issues yet to come.

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

```
foHasErrs :: FIOut → Bool
foHasErrs =  $\neg$ .null.foErrL
EHTyFitsIn
```

```
fitsIn :: Ty → Ty → FIOut
fitsIn ty1 ty2
  = u ty1 ty2
EHTyFitsIn
```

**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*/ $\perp$ ).

The constant *Ty\_Any* denoted by *Any*/ $\perp$  plays two roles

- *Ty\_Any* plays the role of  $\perp$  when it occurs at the lhs of  $\perp \leq \sigma$  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  $\sigma \leq \perp$ . 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  $\sigma$ . 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  $\perp$  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  $\Gamma$  fails.

Typing rule `e-ident1B` uses the environment  $\Gamma$  to retrieve the type of an identifier. This environment `valGam` for types of identifiers simply is declared as an inherited attribute, initialized at the top of the abstract syntax tree. 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  $\vdash^{\text{fit}} \sigma_1 \leq \sigma_2 : \sigma$  and its implementation `fitsIn` do return a type at all. For the idea of checking was to only pass explicit type information  $\sigma^k$  (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)  $\sigma^k$  (or `knTy`), but will be the argument of the (bottom-to-top) resulting type of `e1` (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  $\perp \rightarrow \sigma^k$  (passed to *func.knTy*) the partially known function type and let *fitsIn* fill in the missing details, that is find a type for  $\perp$ . 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.

```

algTy :: MkConAppAlg Ty
algTy = (Ty_Con, Ty_App, 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

mkTyProdApp :: TyL → Ty
mkTyProdApp = mkProdApp algTy

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

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)

```

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  $(,)$ .

```

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

```

EHTy

EHTy

EHTy

EHTy

EHTy

EHInferExpr

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

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

EHInferExpr

**SEM Expr**

<i>Lam loc.funTy</i>	= [Ty_Any] 'mkTyArrow' Ty_Any
<i>foKnFun</i>	= @funTy ≤ @lhs.knTy
( <i>arg.knTy, body.knTy</i> )	= tyArrowArgRes (foTy @foKnFun)
<i>arg.valGam</i>	= gamPushNew @lhs.valGam
<i>loc.ty</i>	= @lhs.knTy

## 2.5.4 Checking PatExpr

Before we can look into more detail at the way new identifiers are introduced in **let**- and  $\lambda$ -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 2.7 together with the rules for patterns from figure 2.8 reflect the desired behaviour. These rules differ from those in figure 2.6 in that a pattern instead of a single identifier is allowed in a value definition and argument of a  $\lambda$ -expression.

---

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

  

$\frac{\Gamma^p \# \Gamma, \sigma^i \stackrel{expr}{\vdash} e^i : \_ \quad \Gamma^p \# \Gamma, \sigma^k \stackrel{expr}{\vdash} e : \sigma^e \quad \sigma^i \stackrel{pat}{\vdash} p : \Gamma^p \quad p \equiv i \vee p \equiv i @ \dots}{\Gamma, \sigma^k \stackrel{expr}{\vdash} \mathbf{let} \ i :: \sigma^i ; p = e^i \mathbf{in} \ e : \sigma^e} \quad (\mathbf{e-let1C})$	$\frac{\Gamma^p \# \Gamma, \sigma^r \stackrel{expr}{\vdash} e : \sigma^e \quad \sigma^p \stackrel{pat}{\vdash} p : \Gamma^p}{\Gamma, \sigma^p \rightarrow \sigma^r \stackrel{expr}{\vdash} \lambda p \rightarrow e : \sigma^p \rightarrow \sigma^e} \quad (\mathbf{e-lam1C})$
---	---

---

Figure 2.7: 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  $\lambda$ -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** [ $knTy : Ty \quad knTyL : TyL \quad ||$ ]

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

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

Figure 2.8: 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  $\Gamma$ '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

```

```

                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 2.7 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:

**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
```

EHInfer

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 } topNm : HsName ]
```

**SEM PatExpr**

```
| Var VarAs lhs.mbTopNm = Just @nm
    .topNm = @nm
| Con CConst IConst App AppTop
    lhs.mbTopNm = Nothing
    .topNm = hsnUnknown
```

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 36.

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

```
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)
```

EHGam

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 33).

EHErrorAbsSyn

```
DATA Err
  | NamesDupIntrod nmL : { [HsName] }
  | PatAriety ty : Ty ariety : { 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 22) 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.errDeclsPP = 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.

## Chapter 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

```
let i = 5
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

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

will give

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

If the use of *id* to define *v* is omitted less information (namely the argument of *id* is an int) to infer a type for *id* is available

```
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 than with the next version of EH (chapter 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 = \x -> x
in let f :: (Int -> Int) -> ...
    f = \i -> \v -> i v
    v = f id 3
in let v = f id 3
in v
```

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

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

The 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 variables* to represent this.

$$\sigma = Int \mid Char$$

$$\begin{array}{l} | (\sigma, \dots, \sigma) \\ | \sigma \rightarrow \sigma \\ | v \end{array}$$

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

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

EHTyAbsSyn

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

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

EHTyPretty

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

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

```
newtype UID = UID [Int] deriving (Eq, Ord)
type UIDL = [UID]
instance Show UID where
  show (UID (l : ls)) = foldl (\ls l → show l ++ "_" ++ ls) (show l) ls
type TyVarId = UID
type TyVarIdL = [TyVarId]
```

EHCommon

EHCommon

EHTy

EHTy

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 2.7, page 35 still hold we need to look at the meaning of  $\leq$  (or *fitsIn*) in the presence of type variables. The idea here is that what is unknown may be replaced by that which is known. For example, if the check  $v \leq \sigma$  is encountered making the previously unknown type  $v$  equal to  $\sigma$  is the easiest way to make  $v \leq \sigma$  true. An alternative way to look at this is that  $v \leq \sigma$  is true under the constraint that  $v$  equals  $\sigma$ .

### 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  $\sigma$  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

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

EHCnstr

EHCnstr

```
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 [11, 23]. Here constraints  $\mathcal{C}$  are used to replace all other references to  $v$  by  $\sigma$ , 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  $>$  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 (\(v, t) → (v, s1 > t)) sl2)
  where sl2' = deleteFirstBy (\(v1, _) (v2, _) → v1 ≡ v2) sl2 sl1
  ftv (C sl) = ftv.map snd $ sl
```

but one must be aware that  $>$  is non-commutative as constraints  $s_1$  in  $s_1 > s_2$  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  $\leq$  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]
EHTyFitsIn
                | otherwise              = bind v t

  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.)  $a \leq a \rightarrow a$  to succeed. This is because it is not

clear if  $a \mapsto a \rightarrow a$  should be the resulting constraint or  $a \mapsto (a \rightarrow a) \rightarrow (a \rightarrow a)$  or one of infinitely many other possible solutions. A so called *infinite type* like this is inhibited by the so called *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  $a \rightarrow a \leq Int \rightarrow Char$  from finding two conflicting constraints  $[a \mapsto Int, a \mapsto Char]$  instead of properly reporting an error.

### 3.3 Reconstructing types for Expr

Constraints are used to make found knowledge about previously unknown types explicit. The typing rules in figure 2.4 (and figure 2.6, figure 2.7) in principle need to be changed. The only reason to adapt some of the rules to the variant in figure 3.1 is to clarify the way constraints are used.

---


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

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

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

$$\frac{(i \mapsto \sigma^i) \in \Gamma \quad \text{fit} \vdash \sigma^i \leq \sigma^k : \sigma \rightsquigarrow \mathcal{C}}{\Gamma, \sigma^k \vdash^{expr} i : \sigma \rightsquigarrow \mathcal{C}} \quad (\text{e-ident2})$$

$$\frac{\text{fit} \vdash (v_1, v_2, \dots, v_n) \leq \sigma^r : (\sigma_1, \sigma_2, \dots, \sigma_n) \rightsquigarrow \mathcal{C} \quad \_ \rightarrow \dots \rightarrow \sigma^r \equiv \sigma^k \quad v_i \text{ fresh}}{\Gamma, \sigma^k \vdash^{expr} \_n : \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow (\sigma_1, \sigma_2, \dots, \sigma_n) \rightsquigarrow \mathcal{C}} \quad (\text{e-con2})$$

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


---

Figure 3.1: Type inferencing for expressions (using constraints)

The type rules in figure 3.1 enforce an order in which checking and inferring types has to be done.

Actually, the rules in figure 3.1 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

```

**ATTR** *AllExpr* [| *tyCnstr* :  $\mathcal{C}$  |]

EHInferExpr

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  $\models$  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

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

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

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  $\Gamma$ , here *valGam*.

The type rules in figure 3.1 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*  $\rightarrow$  (*Ty\_Any*  
                           , [*Err\_NamesNotIntrod* [@nm]])  
               *Just vgi*  $\rightarrow$  (*vgiTy vgi*, [])  
           *fTy* = @gTy  
           *fo* = @fTy  $\leq$  @lhs.knTy  
           .*ty* = foTy @fo

**SEM** *Expr*

| *Var loc.fTy* := @lhs.tyCnstr > @gTy  
           *fo* := @fTy  $\leq$  (@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 additionally 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 2.6) 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* (*hsnProdAry* @*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

*mkNewTyVarL* :: *Int* → *UID* → *TyL*  
*mkNewTyVarL* *sz* *u* = **let** (*\_, vs*) = *mkNewUIDL* *sz* *u* **in** *map* *mkTyVar* *vs*

Some observations are in place

- The main difference with the previous implementation is the use of type variables to represent unknown knowledge. Previously  $\mathbb{I}$  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  $\mathbb{I} \rightarrow \mathbb{I}$  to enforce a  $\dots \rightarrow \dots$  structure. If  $\mathbb{I}$  still would be used, for

```

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

```

the conclusion would be drawn that  $id :: \mathbb{I} \rightarrow \mathbb{I}$ , whereas  $id :: v \rightarrow v$  would later on have bound  $v \mapsto Int$  (at the application  $id\ 3$ ). So,  $\mathbb{I}$  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  $v \notin \text{ftv}(\Gamma)$ . 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.

---


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

$$\frac{\begin{array}{l} \stackrel{fit}{\vdash} \mathcal{C}_1 \sigma^k \leq \sigma^d : \sigma \rightsquigarrow \mathcal{C}_2 \\ \sigma^d \rightarrow () \equiv \sigma^p \\ \stackrel{pat}{-} \vdash p : \sigma^p, \Gamma^p \rightsquigarrow \mathcal{C}_1 \\ p \equiv p_1 p_2 \dots p_n, n \geq 1 \end{array}}{\stackrel{pat}{\sigma^k \vdash} p : \sigma, \Gamma^p \rightsquigarrow \mathcal{C}_{2..1}} \quad (\text{p-apptop2})$$

$$\frac{\begin{array}{l} \text{dom}(\Gamma_1^p) \cap \text{dom}(\Gamma_2^p) = \emptyset \\ \sigma_1^a \stackrel{pat}{\vdash} p_2 : \sigma_2^a, \Gamma_2^p \rightsquigarrow \mathcal{C}_2 \\ \stackrel{pat}{-} \vdash p_1 : \sigma^d \rightarrow (\sigma_1^a, \sigma_2^a, \dots, \sigma_n^a), \Gamma_1^p \rightsquigarrow \mathcal{C}_1 \end{array}}{\stackrel{pat}{-} \vdash p_1 p_2 : \mathcal{C}_2(\sigma^d \rightarrow (\sigma_2^a, \dots, \sigma_n^a)), \Gamma_1^p \uplus \Gamma_2^p \rightsquigarrow \mathcal{C}_{2..1}} \quad (\text{p-app2})$$

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


---

Figure 3.2: Type inferencing for pattern (using constraints)



```

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

```

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

#### 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

```

EHInfer

#### 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.

#### SEM Decl

```

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

```

EHInfer

### 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:

#### DATA TyExpr

```

| Wild

```

EHAbsSyn

```

tyExprAlg = (sem_TyExpr_Con, sem_TyExpr_App
            , sem_TyExpr_AppTop)

```

EHParser

```

pTyExprBase = sem_TyExpr_Con <$> pCon
             <|> sem_TyExpr_Wild <$ pKey "... "

```

```

pTyExpr    <> pParenProd tyExprAlg 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:

```

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

```

EHInferTyExpr

EHInferTyExpr

```

SEM TyExpr
| Wild lhs.ty = tgiTy @tgi

```

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.

```

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

```

EHInferTyExpr

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

```

## Chapter 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  $\forall$ . The meaning of this quantifier is that a value with a universally quantified type can be used with different types for the quantified type variables.

The type signature may be omitted, 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 [25]. 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_6_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 = λi → i 3
    id :: a → 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 = λi → (i 3, i 'x')
    id = λx → 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 (chapter 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 [6, 20, 12].

However, type inferencing also has its limits. 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  $\forall$

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

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

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

```

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

DATA TyVarCateg
  | Plain
  | Fixed

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

```

EHTyAbsSyn

together with a convenience function for making such a quantified type

```

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

mkTyQu :: TyVarIdL → Ty → Ty
mkTyQu tvL t = foldr (\tv t → Ty_Quant tv t) t tvL

```

EHTy

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  $\forall$  cannot be explicitly denoted.

```

DATA TyExpr

```

EHAbsSyn



|  $Var\ nm : \{HsName\}$

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

EHParse

```

tyExprAlg    = (sem_TyExpr_Con, sem_TyExpr_App
               , sem_TyExpr_AppTop)
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  $\Gamma$  whereas no  $\forall$ 's are present in types used throughout type inferencing expressions and patterns. This is to guarantee the principle type property .

## 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  $\forall$  quantifiers, nothing has to be changed there.

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

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 4.1. 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

```

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

```

---


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

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

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

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


---

Figure 4.1: Type inferencing for expressions with quantifier  $\forall$

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  $\Gamma$ . The way this is done varies with the presence of a type signature. Rule *e-let3* and rule *e-let-tysig3* (figure 4.1) differ only in the use of a type signature, if present.

**SEM Decl**

| *TySig loc.sigTy* = @*tyExpr.ty*

EHIInfer

**SEM Decl**

| *TySig loc.sigTy* := *tyQuantify* ( $\in$  @*tyExpr.tyVarWildL*) @*tyExpr.ty*

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

*tyQuantify* :: (*TyVarId*  $\rightarrow$  *Bool*)  $\rightarrow$  *Ty*  $\rightarrow$  *Ty*

*tyQuantify tvIsBound ty* = *mkTyQu* (*filter* ( $\neg$ .*tvIsBound*) (*ftv ty*)) *ty*

EHTyQuantify

*tyQuantifyClosed* :: *Ty*  $\rightarrow$  *Ty*

*tyQuantifyClosed* = *tyQuantify* (*const False*)

EHTyQuantify

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  $\rightarrow$  a
    id =  $\lambda x \rightarrow 3$ 
in ...
```

we end up with fitting  $v_1 \rightarrow Int \leq a \rightarrow a$ . 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*

EHIInfer

**SEM Decl**

| *Val loc.knTy* := *tyInstKnown* @!Uniq @*sigTy*

by using a variant of *tyInst*

*tyInstKnown* :: *UID*  $\rightarrow$  *Ty*  $\rightarrow$  *Ty*

*tyInstKnown* = *tyInst'* ( $\lambda tv \rightarrow$  *Ty\_Var tv TyVarCateg.Fixed*)

EHTyInstantiate

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 =  $\lambda x \rightarrow x$ 
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  $\Gamma$ , 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 4.1 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
```

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

EHInfer

All available constraints in the form of *decls.tyCnstr* are applied to both global (*gValGam*) and local (*lValGam*)  $\Gamma$ . 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
  = 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  $\Gamma$  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_13_1 -> (v_13_1,v_13_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  $f_1 = \lambda x \rightarrow g_1 x$ 
     $g_1 = \lambda y \rightarrow f_1 y$ 
     $f_2 :: a \rightarrow a$ 
     $f_2 = \lambda x \rightarrow g_2 x$ 
     $g_2 = \lambda y \rightarrow f_2 y$ 
in  $f_1 3$ 
```

This results in

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

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.

#### 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.

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

[EHGam](#)

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

[EHGam](#)

```
SEM TyExpr
```

[EHInferTyExpr](#)

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



## Chapter 5

# EH 4: $\forall$ and $\exists$ 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  $\forall$  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
    xy' = ixy xy
    pq :: ∃ a.(a, a → Int)
    pq = ('x', id) -- ERROR
```



in  $xy'$

An existentially quantified type is specified with keyword `exists`, or  $\exists$ . 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 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

```
let v1 ::  $\exists a.a$ 
    v2 = v1
in v2
```

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:

```

let v1 :: ( $\exists a.a, \exists b.b$ )
    v2 = v1
    (a,b) = v1
    (c,d) = v1
in v2

```

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.

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

```

let id  :: a  $\rightarrow$  a
    ixy =  $\lambda(v,f) \rightarrow f v$ 
    pq  = ('x', id)
    xy  :: (a, a  $\rightarrow$  Int)
    xy  = (3, id)
in let xy' = ixy xy
    pq' = ixy pq
in xy'

```

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  $\sigma_1$  and  $\sigma_2$  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  $\sigma_1$  and  $\sigma_2$  in  $(\sigma_1, \sigma_2)$  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  $\exists$

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

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 be 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 later on (not included in this paper):

EHTy

$$\begin{aligned} \text{tyquForall } TyQu\_Exists &= TyQu\_Forall \\ \text{tyquForall } q &= q \end{aligned}$$

```

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

$\leq$  is a **partial ordering**. With the presence of quantifiers during type inferencing, the function *fitsIn*, implementing the so called *subsumption* relation  $\leq$  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  $Int \leq \exists a.a$ , 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  $\forall a.a \rightarrow a \leq Int \rightarrow Int$  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

$$\forall a.a \leq \sigma \leq \exists a.a$$

meaning that a universally quantified type can be used at a place where any other type is expected, say some type  $\sigma$  (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*/ $\leq$  has to deal with.

**Impredicativity.** The following example (from Botlan [4])

```

let choose :: a → a → a
      id     :: a → a
      v     = choose id
in v

```

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

- $v :: \forall b.(b \rightarrow b) \rightarrow b \rightarrow b$ . 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  $v_1 \rightarrow v_1$ , giving  $choose\ id :: (v_1 \rightarrow v_1) \rightarrow v_1 \rightarrow v_1$  resulting in the given quantified type. Function  $v$  can safely be applied to a function of type  $Int \rightarrow Int$ .
- $v :: (\forall c.c \rightarrow c) \rightarrow (\forall d.d \rightarrow d)$ . This is system-F's answer [29]. 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  $Int \rightarrow Int$  because this function is not general enough to be used as if it were of type  $\forall a.a \rightarrow a$ . 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 *impredicativity*.

The critical observation for this version of EH is that it is difficult 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  $fitsIn/\leq$  to allow richer types still to match properly.

**Subsumption  $\leq$  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  $\leq$  depends on context.**  $fitsIn$  is used as a tool to enforce the  $\leq$  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 68). 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 67). However, the precise use of  $fitsIn$  in these contexts differs slightly in the way instantiation is done.

- For an application  $f\ a$  of  $f :: \sigma_2 \rightarrow \dots$  to  $a :: \sigma_1$  we have to check the type  $\sigma_1$  of an actual argument against the type  $\sigma_2$  of an expected via  $\sigma_1 \leq \sigma_2$ . As we learned from looking at the  $choose$  example, Haskell's convention is to instantiate  $\sigma_1$  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  $v_{il}$ , named an *instantiating context*.
- For checking an expression  $e :: \sigma_2$  of a declaration  $v :: \sigma_1; v = e$  to its known type  $\sigma_1$  we check  $\sigma_2 \leq \sigma_1$ . In this case we want to avoid instantiating  $\sigma_2$ . The necessity of avoiding this becomes clear if we look at a situation where  $\sigma_2 \equiv (\exists a.a, \dots)$  and  $\sigma_1 \equiv (v, \dots)$ , 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 be told it is checking types in a *strong context*, notated by  $v_s$ .

- A third context will also be mentioned here for completeness, a so called *weak context*  $v_w$ . 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 (not included in this paper).

$fitsIn/\leq$  therefore needs an option  $v$  to describe these variations

```

v = v_s    -- strong context
  | v_il   -- instantiating context, for expr App
  | v_i    -- instantiating context, for patterns

```

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

```

fioBindRFirst = f i_r^+ -bind  -- prefer binding of a rhs tvar over instantiating
                | f i_r^- -bind
fioBindLFirst = f i_l^+ -bind  -- prefer binding of a lhs tvar over instantiating
                | f i_l^- -bind
fioLeaveRInst  = f i_r^+ -inst  -- leave rhs (of fitsIn) instantiated
                | f i_r^- -inst

```

where the  $+$  variants stand for *True*. A *True* value for the flag  $fioBindRFirst$  states that in case of  $\sigma \leq v$  a constraint ( $v \mapsto \sigma$ ) will result, otherwise first  $\sigma$  will be instantiated and  $v$  be bound to the instantiated  $\sigma$ . Similary we have  $fioBindLFirst$  for  $v \leq \sigma$ . Finally,  $fioLeaveRInst$  determines if an instantiation done for  $\sigma$  in  $\dots \leq \sigma$  will return the instantiated  $\sigma$  or  $\sigma$  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

	$fioBindRFirst$	$fioBindLFirst$	$fioLeaveRInst$
$v_s$	$f i_r^+ -bind$	$f i_l^+ -bind$	$f i_r^- -inst$
$v_{il}$	$f i_r^- -bind$	$f i_l^+ -bind$	$f i_r^- -inst$
$v_i$	$f i_r^- -bind$	$f i_l^- -bind$	$f i_r^+ -inst$

So, for example the  $v_i$  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

```

data FIOpts = FIOpts{fioLeaveRInst :: Bool ,fioBindRFirst :: Bool
                    , fioBindLFirst :: Bool ,fioUniq      :: UID
                    , fioCoContra  :: CoContraVariance
                    } deriving Show

v_s :: FIOpts
v_s = FIOpts {fioLeaveRInst = False, fioBindRFirst = True
            , fioBindLFirst = True ,fioUniq      = startUID
            , fioCoContra  = ⊕
            }

v_il :: FIOpts
v_il = v_s {fioBindRFirst = False}

v_i :: FIOpts
v_i = v_il {fioLeaveRInst = True, fioBindLFirst = False}

```

EHTyFitsIn

**Co- and contravariance.** For tuples the check  $(\sigma_1, \sigma_2) \leq (\sigma_3, \sigma_4)$  will break down into  $\sigma_1 \leq \sigma_3$  and  $\sigma_2 \leq \sigma_4$  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  $\sigma_1 \rightarrow \sigma_2 \leq \sigma_3 \rightarrow \sigma_4$  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

```

let  $g :: \sigma_1 \rightarrow \sigma_2$ 
       $f :: \sigma_3 \rightarrow \sigma_4$ 
       $f = g$ 
       $a :: \sigma_3$ 
in  $f a$ 

```

So, what happens when  $f$  is called and what does it mean in terms of types  $\sigma$ ? The caller of  $f$  in the application  $f a$  expects that the function  $f$  accepts a  $\sigma_3$ . However,  $g$  is invoked instead, so a value of type  $\sigma_3$  is passed to a function expecting a  $\sigma_1$ , which in terms of fitting means  $\sigma_3 \leq \sigma_1$ . The observation here is that the direction of  $\leq$  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  $\leq$  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

```

 $\mathcal{V} = \oplus$   -- CoVariant
      |  $\ominus$  -- ContraVariant
      |  $\odot$  -- CoContraVariant (both co/contra)

```

with a corresponding Haskell definition

```

data CoContraVariance =  $\oplus$  |  $\ominus$  |  $\odot$  deriving (Show, Eq)

```

[EHCommon](#)

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

## 5.2.2 Implementation

**Typing rules.** Let us look more precisely at  $\leq$  which we now also will describe with rules, in figure 5.2 and figure 5.1. 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  $v_s$  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.

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

- Rule *f-forall-r2* applies in

```

let  $ii :: Int \rightarrow Int$ 
       $ii = id$ 
       $id :: \forall a.a \rightarrow a$ 
       $id = ii$ 
in  $ii$ 

```

to the check  $Int \rightarrow Int \leq \forall a.a \rightarrow a$  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  $(v \mapsto Int)$ . This is not correct.

$$\boxed{v \vdash \sigma^l \leq \sigma^r : \sigma \rightsquigarrow \mathcal{C}}$$

$$\frac{
\begin{array}{c}
\text{fit} \\
v_s \vdash C_1 \sigma_2^a \leq C_1 \sigma_1^a : \sigma^a \rightsquigarrow C_2 \\
\text{fit} \\
v \vdash \sigma_1^r \leq \sigma_2^r : \sigma^r \rightsquigarrow C_1
\end{array}
}{
\text{fit} \\
v \vdash \sigma_1^a \rightarrow \sigma_1^r \leq \sigma_2^a \rightarrow \sigma_2^r : \sigma^a \rightarrow C_2 \sigma^r \rightsquigarrow C_{2..1}
} \quad (\text{f-arrow4})$$

$$\frac{
\begin{array}{c}
\text{fit} \\
v \vdash C_1 \sigma_1^l \leq C_1 \sigma_2^l : \sigma^l \rightsquigarrow C_2 \\
\text{fit} \\
v \vdash \sigma_1^r \leq \sigma_2^r : \sigma^r \rightsquigarrow C_1
\end{array}
}{
\text{fit} \\
v \vdash (\sigma_1^l, \sigma_1^r) \leq (\sigma_2^l, \sigma_2^r) : (\sigma^l, C_2 \sigma^r) \rightsquigarrow C_{2..1}
} \quad (\text{f-prod4})$$

Figure 5.1: Fitting/subsumption for type applications

$$\boxed{v \vdash \sigma^l \leq \sigma^r : \sigma \rightsquigarrow \mathcal{C}}$$

$$\frac{
\begin{array}{c}
\text{fit} \\
v \vdash \rho^i \leq \sigma_2 : \sigma \rightsquigarrow \mathcal{C} \\
(-, \rho^i) \equiv \text{inst}_v(\bar{\alpha}, \rho_1)
\end{array}
}{
\text{fit} \\
v \vdash \forall \bar{\alpha}. \rho_1 \leq \sigma_2 : \sigma \rightsquigarrow \mathcal{C}
} \quad (\text{f-forall-l})$$

$$\frac{
\begin{array}{c}
\text{fit} \\
f i_{r-inst}^+ \vdash \sigma_1 \leq \rho^i : \sigma \rightsquigarrow \mathcal{C} \\
(-, \rho^i) \equiv \text{inst}_v(\bar{\beta}, \rho_2)
\end{array}
}{
\text{fit} \\
f i_{r-inst}^+ \vdash \sigma_1 \leq \forall \bar{\beta}. \rho_2 : \sigma \rightsquigarrow \mathcal{C}
} \quad (\text{f-forall-r1})$$

$$\frac{
\begin{array}{c}
\text{fit} \\
f i_{r-inst}^- \vdash \sigma_1 \leq \rho^i : \_ \rightsquigarrow \mathcal{C} \\
(-, \rho^i) \equiv \text{inst}_f(\bar{\beta}, \rho_2)
\end{array}
}{
\text{fit} \\
f i_{r-inst}^- \vdash \sigma_1 \leq \forall \bar{\beta}. \rho_2 : \mathcal{C} (\forall \bar{\beta}. \rho_2) \rightsquigarrow \mathcal{C}
} \quad (\text{f-forall-r2})$$

$$\frac{
\begin{array}{c}
\text{fit} \\
v \vdash \rho^i \leq \sigma_2 : \sigma \rightsquigarrow \mathcal{C} \\
(-, \rho^i) \equiv \text{inst}_c(\bar{\alpha}, \rho_1)
\end{array}
}{
\text{fit} \\
v \vdash \exists \bar{\alpha}. \rho_1 \leq \sigma_2 : \sigma \rightsquigarrow \mathcal{C}
} \quad (\text{f-exists-l})$$

$$\frac{
\begin{array}{c}
\text{fit} \\
f i_{r-inst}^+ \vdash \sigma_1 \leq \rho^i : \sigma \rightsquigarrow \mathcal{C} \\
(-, \rho^i) \equiv \text{inst}_c(\bar{\beta}, \rho_2)
\end{array}
}{
\text{fit} \\
f i_{r-inst}^+ \vdash \sigma_1 \leq \exists \bar{\beta}. \rho_2 : \sigma \rightsquigarrow \mathcal{C}
} \quad (\text{f-exists-r1})$$

$$\frac{
\begin{array}{c}
\text{fit} \\
f i_{r-inst}^- \vdash \sigma_1 \leq \rho^i : \sigma \rightsquigarrow \mathcal{C} \\
(\bar{v}, \rho^i) \equiv \text{inst}_v(\bar{\beta}, \rho_2)
\end{array}
}{
\text{fit} \\
f i_{r-inst}^- \vdash \sigma_1 \leq \exists \bar{\beta}. \rho_2 : \mathcal{C} (\exists \bar{\beta}. \rho_2) \rightsquigarrow \mathcal{C} \setminus \frac{\text{dom}}{\bar{v}}
} \quad (\text{f-exists-r2})$$

Figure 5.2: Fitting/subsumption for quantified types



Recall that by successfully 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  $f :: (\exists a.a) \rightarrow Int$ 
       $v = f$  3
in  $v$ 

```

to the check  $Int \leq \exists a.a$  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 than 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  $v$  should be made strong. This is described via a environment encoding this information

```

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

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

```

EHTyFitsIn

EHTyFitsIn

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 later on (not included in this paper).

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

```

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

```

EHTyFitsIn

$$\begin{aligned} & , foErrL = [] \\ & \} \end{aligned}$$

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

```

data FIIIn = FIIIn{fiFIOpts  :: FIOpts,fiUniq :: UID
  ,fiCoContra :: CoContraVariance
  }
emptyFI = FIIIn{fiFIOpts  = vs    ,fiUniq = startUID
  ,fiCoContra =  $\oplus$ 
  }

```

EHTyFitsIn

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

**The fitting.** The preliminaries of *fitsIn* have not changed much compared to the previous version (page 45). All internally defined functions now take an additional top-down *fi* :: *FIIIn* 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 ty1 ty2
  = fo
where
  res fi t      = emptyFO{foUniq = fiUniq fi,foTy = t
    ,foCoContraL = cocoGamLookup t cocoGam }
  err fi e      = emptyFO{foUniq = fiUniq opts,foErrL = e }
  manyFO fos    = foldr1 ( $\lambda$ fo1 fo2 → if foHasErrs fo1 then fo1 else fo2) fos
  bind fi tv t = (res fi t) {foCnstr = tv  $\mapsto$  t }
  occurBind fi v t
    | v ∈ ftv t = err fi [Err.UnifyOccurs ty1 ty2 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 then  $\lambda$ fo → let s = cnstrFilter (const. $\neg$ .( $\in$  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  $\leq$  are similar to the previous version with the exception of an alternative for flipping  $t_1 \leq t_2$  into  $t_2 \leq t_1$  if the variance is  $\ominus$ , and an additional guard on *fiBindLFirst* and *fiBindRFirst*.

EHTyFitsIn

```

fi u fi t1      t2
  | fiCoContra fi ≡  $\ominus$       = u (fi{fiCoContra =  $\oplus$ 
    ,fiFIOpts = fiSwapVariance (fiFIOpts fi) })
    t2 t1
fi u fi Ty_Any    t2      = res fi t2
fi u fi t1      Ty_Any    = res fi t1
fi u fi t1 @(Ty_Con s1)  t2 @(Ty_Con s2)

```

$$\begin{array}{l}
| s1 \equiv s2 \qquad \qquad \qquad = res\ fi\ t2 \\
u\ fi\ t_1\ @(Ty\_Var\ v1\ f1) \quad (Ty\_Var\ v2\ f2) \\
| v1 \equiv v2 \wedge f1 \equiv f2 \qquad \qquad = res\ fi\ t_1 \\
u\ fi\ t_1\ @(Ty\_Var\ v1\ f) \quad t_2 \\
| fioBindLFirst\ (fiFIOpts\ fi) \wedge f \equiv TyVarCateg\_Plain \\
\qquad \qquad \qquad \qquad \qquad \qquad = occurBind\ fi\ v1\ t_2 \\
u\ fi\ t_1 \qquad \qquad \qquad t_2\ @(Ty\_Var\ v2\ f) \\
| fioBindRFirst\ (fiFIOpts\ fi) \wedge f \equiv TyVarCateg\_Plain \\
\qquad \qquad \qquad \qquad \qquad \qquad = occurBind\ fi\ v2\ t_1
\end{array}$$

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

Compared to the rules in figure 5.2 an additional case has been included for an exact match of two quantified types when we want  $t_1 \leq t_2$  and  $t_2 \leq t_1$  both to hold. We will postpone discussion until later (not included in this paper).

EHTyFitsIn

$$\begin{array}{l}
u\ fi\ t_1\ @(Ty\_Quant\ q1\ \_ \_) \quad t_2\ @(Ty\_Quant\ q2\ \_ \_) \\
| fiCoContra\ fi \equiv \odot \wedge q1 \equiv q2 \\
\qquad \qquad \qquad \qquad \qquad \qquad = u\ fi_2\ uqt_1\ uqt_2 \\
\mathbf{where}\ (fi_1, uqt_1, \_) = unquant\ fi\ t_1\ False\ instCoConst \\
\qquad \qquad \qquad (fi_2, uqt_2, \_) = unquant\ fi_1\ t_2\ False\ instCoConst
\end{array}$$

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 tyqulsForall q then Ty_Var v TyVarCateg_Plain else mkTyCon ("C_" ++ show v)
instContra q v = if tyqulsForall q then Ty_Var v TyVarCateg_Fixed else Ty_Var v TyVarCateg_Plain

```

The rules in figure 5.2 indicate for different combinations of options and quantifiers how to instantiate type variables. For example, the first case of

EHTyFitsIn

$$\begin{array}{l}
u\ fi\ t_1 \qquad \qquad \qquad t_2\ @(Ty\_Quant\ \_ \_ \_) \\
| fiCoContra\ fi \not\equiv \odot \wedge fioLeaveRInst\ (fiFIOpts\ fi) \\
\qquad \qquad \qquad \qquad \qquad \qquad = back2\ (u\ fi_2\ t_1\ uqt_2) \\
\mathbf{where}\ (fi_2, uqt_2, back2) = unquant\ fi\ t_2\ False\ instCoConst \\
u\ fi\ t_1 \qquad \qquad \qquad t_2\ @(Ty\_Quant\ \_ \_ \_) \\
| fiCoContra\ fi \not\equiv \odot \wedge \neg\ (fioLeaveRInst\ (fiFIOpts\ fi)) \\
\qquad \qquad \qquad \qquad \qquad \qquad = back2\ (u\ fi_2\ t_1\ uqt_2) \\
\mathbf{where}\ (fi_2, uqt_2, back2) = unquant\ fi\ t_2\ True\ instContra
\end{array}$$

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

$$\begin{array}{l}
u\ fi\ t_1\ @(Ty\_Quant\ \_ \_ \_) \quad t_2 \\
| fiCoContra\ fi \not\equiv \odot \qquad \qquad \qquad = u\ fi_1\ uqt_1\ t_2 \\
\mathbf{where}\ (fi_1, uqt_1, back1) = unquant\ fi\ t_1\ False\ instCoConst
\end{array}$$

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  $fi_{l-bound}^-$  and  $fi_{r-bound}^-$  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  $\exists$ '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  $\forall$ '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 5.3. 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.

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

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

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

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

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

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

Figure 5.3: 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  $a \rightarrow (a, b, b, b \rightarrow c \rightarrow (b, c))$  this is the complete type for  $a$ ,  $(a, b, b, b \rightarrow c \rightarrow (b, c))$  for  $b$  and  $c \rightarrow (b, c)$  for  $c$ .

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

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

Figure 5.4: 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-contr. 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 later on (not included in this paper).

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  $\Gamma$  in a straightforward way as specified by figure 5.4.

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

**ATTR**  $Ty \ [ \ [ \ frTvLL : \ { \ TyVarIdL \} \ ] \ ]$

**SEM**  $TyAGIf$

|  $AGIf$     **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**. $frTvL$  =  $head \ @ty.frTvLL \ \setminus \setminus \ @introTVarL$

|  $Quant \ Var$  **lhs**. $frTvLL$  =  $[@frTvL]$

|  $Any \ Con$     **lhs**. $frTvLL$  =  $[[]]$

$listCombineUniq :: Eq \ a \Rightarrow \ [[a]] \rightarrow [a]$

$listCombineUniq = nub.concat$

EHTyQuantify

EHCommon

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 5.3. For the given example this would be  $[[1], [1]]$ .

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

**ATTR**  $Ty$  [ $\mathcal{V} : CoContraVariance$  ||]

**SEM**  $Ty$

|  $App$   $func.$  $\mathcal{V} = \odot$   
    $arg.$  $\mathcal{V} = \mathbf{if}$   $@appIsLikeProd \vee @isArrowRoot$  **then**  $\oplus$   
                   **else if** $@appIsArrow$  **then**  $\ominus$   
   **else**  $\odot$

EHTyQuantify

**SEM**  $TyAGIf$

|  $AGIf$   $ty.$  $\mathcal{V} = \oplus$

EHTyQuantify

**SEM**  $Ty$

|  $Var$   $App$   $Quant$   $\mathbf{loc.isQuLoc} = @lhs.\mathcal{V} \neq \odot$

Quantifiability of a location is based on co- and contravariance information as passed from top to bottom, as prescribed by the rules in figure 5.3. We also need to know what an  $App$  represents, that is, if it is a function type ( $appIsArrow$ ) or tuple type ( $appIsLikeProd$ ):

**SEM**  $Ty$

|  $App$   $\mathbf{loc.isArrowRoot} = @appIsArrow \wedge @isSpineRoot$

EHTyCommonAG

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

**SEM**  $TyAGIf$

|  $AGIf$   $\mathbf{loc.qHereTvL} = []$

EHTyQuantify

**SEM**  $Ty$

|  $Var$   $\mathbf{loc.qHereTvL} = \mathbf{if}$   $@isQuLoc$  **then**  $[@tv]$  **else**  $[]$   
 |  $App$   $\mathbf{loc.qHereTvL} = \mathbf{if}$   $@isQuLoc$   
                   **then if**  $@appIsArrow \vee @appIsLikeProd$   
                           **then**  $tvarOccurGE2 @frTvLL$   
                           **else**  $@frTvL$   
                   **else**  $[]$

EHTyQuantify

$tvarOccurCount :: [TyVarIdL] \rightarrow AssocL TyVarId Int$

$tvarOccurCount = \mathit{map} (\lambda vl @ (v : \_) \rightarrow (v, \mathit{length} vl)).\mathit{group.sort.concat}$

$tvarOccurGE2 :: [TyVarIdL] \rightarrow TyVarIdL$

$tvarOccurGE2 = \mathit{map} \mathit{fst.filter} ((>1).\mathit{snd}).\mathit{tvarOccurCount}$

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 5.3.

**ATTR**  $TyAGIf$   $Ty$  [ $tvIsBound : \{TyVarId \rightarrow Bool\}$  ||]

**SEM**  $TyAGIf$

|  $AGIf$   $\mathbf{loc.(qBndTvL, tvIsBound)} = \mathit{tvvarsToQuant} \mathit{True} @lhs.\mathit{tvIsBound} @qHereTvL$

**SEM**  $Ty$

|  $App$   $Var$   $\mathbf{loc.(qBndTvL, tvIsBound)} = \mathit{tvvarsToQuant} @isQuLoc @lhs.\mathit{tvIsBound} @qHereTvL$   
 |  $Quant$   $\mathbf{loc.tvIsBound} = \mathit{tvBoundAdd} @lhs.\mathit{tvIsBound} @introTVarL$

EHTyQuantify

EHTyQuantify

$\mathit{tvBoundAdd} :: (TyVarId \rightarrow Bool) \rightarrow TyVarIdL \rightarrow TyVarId \rightarrow Bool$

$\mathit{tvBoundAdd} \mathit{tvIsBound} \mathit{tvL} = \lambda v \rightarrow v \in \mathit{tvL} \vee \mathit{tvIsBound} v$

$\mathit{tvvarsToQuant} :: Bool \rightarrow (TyVarId \rightarrow Bool) \rightarrow TyVarIdL \rightarrow (TyVarIdL, TyVarId \rightarrow Bool)$

```

tvVarsToQuant isQuLoc tvIsBound tvL
= if isQuLoc
  then let boundables = filter ( $\neg$ .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 =  $@appIsLikeProd \vee @lhs.V \equiv \ominus \wedge \neg @appIsArrow$ 
| Var loc.qAsExist =  $@lhs.V \equiv \ominus$ 
| 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 Ty 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  $\rightarrow$  Bool)  $\rightarrow$  Ty  $\rightarrow$  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 5.5 holds the adapted rules for expressions, figure 5.6 for patterns. The main differences are as follows:

- All rules are passed an additional context parameter indicating the way  $\leq$  has to be done with respect to strength  $v$ . See page 69 for the relevant discussion.
- The  $v$  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.
- Types of tuple constructors (and destructors) are now stored in the  $\Gamma$  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  $\Gamma$ . The rule p-var4 now covers the case for (tuple)constructors too. This change also prepares for the introduction of datatypes in the next version of EH.



---


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

$$\frac{
\begin{array}{c}
v_{il}, \Gamma, \sigma^a \vdash e_2 : \_ \rightsquigarrow \mathcal{C}_2 \\
v, \Gamma, v \rightarrow \sigma^k \vdash e_1 : \sigma^a \rightarrow \sigma \rightsquigarrow \mathcal{C}_1 \\
v \text{ fresh}
\end{array}
}{v, \Gamma, \sigma^k \vdash e_1 e_2 : \mathcal{C}_2 \sigma \rightsquigarrow \mathcal{C}_{2..1}} \quad (\text{e-app4})$$

$$\frac{
\begin{array}{c}
v, \Gamma^p \# \Gamma, \sigma^r \vdash e : \sigma^e \rightsquigarrow \mathcal{C}_3 \\
v, \Gamma, \sigma^p \vdash p : \_, \Gamma^p \rightsquigarrow \mathcal{C}_2 \\
v \vdash v_1 \rightarrow v_2 \leq \sigma^k : \sigma^p \rightarrow \sigma^r \rightsquigarrow \mathcal{C}_1 \\
v_i \text{ fresh}
\end{array}
}{v, \Gamma, \sigma^k \vdash \lambda p \rightarrow e : \mathcal{C}_3 \sigma^p \rightarrow \sigma^e \rightsquigarrow \mathcal{C}_{3..1}} \quad (\text{e-lam4})$$

$$\frac{
\begin{array}{c}
(\xi \mapsto \sigma) \in \Gamma \\
v \vdash \sigma \leq \sigma^k : \sigma \rightsquigarrow \mathcal{C} \\
\text{fit}
\end{array}
}{v, \Gamma, \sigma^k \vdash \xi : \sigma \rightsquigarrow \mathcal{C}} \quad (\text{e-ident4})$$

$$\frac{
\begin{array}{c}
v, \Gamma^q \# \Gamma, \sigma^k \vdash e : \sigma^e \rightsquigarrow \mathcal{C}_3 \\
\Gamma^q \equiv \text{map} (\lambda(n, \sigma) \rightarrow (n, \text{inst}^\exists(\sigma))) \Gamma^{p'} \\
\text{ftv} (\mathcal{C}_{2..1} \Gamma), \oplus \vdash \Gamma^p : \Gamma^{p'} \\
\text{quGam} \\
v_s, \Gamma^p \# \Gamma, \sigma^p \vdash e^i : \_ \rightsquigarrow \mathcal{C}_2 \\
v_s, \Gamma, \mathbb{I} \vdash p : \sigma^p, \Gamma^p \rightsquigarrow \mathcal{C}_1 \\
\text{pat}
\end{array}
}{v, \Gamma, \sigma^k \vdash \mathbf{let} p = e^i \mathbf{in} e : \sigma^e \rightsquigarrow \mathcal{C}_{3..1}} \quad (\text{e-let4})$$

$$\frac{
\begin{array}{c}
v, (\Gamma^q - [i \mapsto \_]) \# [i \mapsto \sigma^q] \# \Gamma, \sigma^k \vdash e : \sigma^e \rightsquigarrow \mathcal{C}_3 \\
\Gamma^q \equiv \text{map} (\lambda(n, \sigma) \rightarrow (n, \text{inst}^\exists(\sigma))) \Gamma^{p'} \\
\text{ftv} (\mathcal{C}_{2..1} \Gamma), \oplus \vdash \Gamma^p : \Gamma^{p'} \\
\text{quGam} \\
v_s, (\Gamma^p - [i \mapsto \_]) \# [i \mapsto \sigma^q] \# \Gamma, \sigma^q \vdash e^i : \_ \rightsquigarrow \mathcal{C}_2 \\
[, \oplus \vdash \sigma^i : \sigma^q \rightsquigarrow \_ \\
\text{qu} \\
p \equiv i \vee p \equiv i @ \dots \\
\text{pat} \\
v_s, \Gamma, \sigma^i \vdash p : \_, \Gamma^p \rightsquigarrow \mathcal{C}_1
\end{array}
}{v, \Gamma, \sigma^k \vdash \mathbf{let} i :: \sigma^i ; p = e^i \mathbf{in} e : \sigma^e \rightsquigarrow \mathcal{C}_{3..1}} \quad (\text{e-let-tysig4})$$

$$\frac{
\begin{array}{c}
v \vdash \text{Int} \leq \sigma^k : \sigma \rightsquigarrow \mathcal{C} \\
\text{fit}
\end{array}
}{v, \Gamma, \sigma^k \vdash \text{minint} \dots \text{maxint} : \sigma \rightsquigarrow \mathcal{C}} \quad (\text{e-int4})$$


---

Figure 5.5: Type checking/inferencing for expression

---

$$\nu, \Gamma, \sigma^k \stackrel{pat}{\vdash} p : \sigma, \Gamma^p \rightsquigarrow \mathcal{C}$$

  

$$\frac{\begin{array}{l} \stackrel{fit}{\nu \vdash \mathcal{C}_1 \sigma^k \leq \sigma^d : \sigma \rightsquigarrow \mathcal{C}_2} \\ \sigma^d \rightarrow () \equiv \sigma^p \\ \stackrel{pat}{\nu, \Gamma, - \vdash p : \sigma^p, \Gamma^p \rightsquigarrow \mathcal{C}_1} \\ p \equiv p_1 p_2 \dots p_n, n \geq 1 \end{array}}{\nu, \Gamma, \sigma^k \stackrel{pat}{\vdash} p : \sigma, \Gamma^p \rightsquigarrow \mathcal{C}_{2..1}} \quad (\text{p-apptop4})$$

  

$$\frac{\begin{array}{l} \text{dom}(\Gamma_1^p) \cap \text{dom}(\Gamma_2^p) = \emptyset \\ \stackrel{pat}{\nu, \Gamma, \sigma_1^a \vdash p_2 : -, \Gamma_2^p \rightsquigarrow \mathcal{C}_2} \\ \stackrel{pat}{\nu, \Gamma, - \vdash p_1 : \sigma^d \rightarrow (\sigma_1^a, \sigma_2^a, \dots, \sigma_n^a), \Gamma_1^p \rightsquigarrow \mathcal{C}_1} \end{array}}{\nu, \Gamma, - \stackrel{pat}{\vdash} p_1 p_2 : \mathcal{C}_2(\sigma^d \rightarrow (\sigma_2^a, \dots, \sigma_n^a)), \Gamma_1^p \# \Gamma_2^p \rightsquigarrow \mathcal{C}_{2..1}} \quad (\text{p-app4})$$

  

$$\frac{\sigma \equiv \text{inst}^\exists(\sigma^k)}{\nu, \Gamma, \sigma^k \stackrel{pat}{\vdash} i : \sigma, [i \mapsto \sigma] \rightsquigarrow []} \quad (\text{p-var4})$$

$$\frac{\begin{array}{l} (\text{unl} \mapsto \sigma^u) \in \Gamma \\ \stackrel{fit}{\nu_i \vdash \sigma^u \leq v_1 \rightarrow v_2 : \sigma \rightsquigarrow -} \\ v_i \text{ fresh} \end{array}}{\nu, \Gamma, - \stackrel{pat}{\vdash} I : \sigma, [] \rightsquigarrow []} \quad (\text{p-con4})$$

---

Figure 5.6: Type checking/inferencing for pattern

- A **let**-expression in rule **e-let4** and rule **e-let-tysig4** quantify bindings via the rules in figure 5.4 and figure 5.3. Additionally, 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*.

EHIInferExpr

```

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 (hsnProdAry 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 3.2, page 49). The *Con* alternative now also uses the *valGam* to find a binding for a tuple dissector/destructor:

EHIInferPatExpr

```

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 also need some modifications to take care of the quantification and instantiation of top-level 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 = vs
    arg .fiOpts = vil

```

```

SEM Decl
  | Val expr .fiOpts = vs
    patExpr.fiOpts = vs

```

```

SEM AGItf
  | AGItf expr .fiOpts = vs

```

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)
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  $\rightarrow$  but still applies right associatively.

**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*}

EHInferExpr

**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**)

EHInferPatExpr

# Bibliography

- [1] The Glasgow Haskell Compiler. <http://www.haskell.org/ghc/>, 2004.
- [2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
- [3] Arthur Baars. Attribute Grammar System. <http://www.cs.uu.nl/groups/ST/twiki/bin/view/Center/Attribu> 2004.
- [4] Didier Botlan, Le and Didier Remy. ML-F, Raising ML to the Power of System F. In *ICFP*, 2003.
- [5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of Principles of Programming Languages (POPL)*, pages 207–212. ACM, ACM, 1982.
- [6] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.
- [7] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A Formal Specification of the Haskell 98 Module System. In *Haskell Workshop*, pages 17–29, 2002.
- [8] Atze Dijkstra. EHC Web. <http://www.cs.uu.nl/groups/ST/Ehc/WebHome>, 2004.
- [9] Karl-Filip Faxen. A Static Semantics for Haskell. *Journal of Functional Programming*, 12:295, 2002.
- [10] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type Classes in Haskell. *ACM TOPLAS*, 18:109–138, mar 1996.
- [11] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Generalizing Hindley-Milner Type Inference Algorithms. Technical report, Institute of Information and Computing Science, University Utrecht, Netherlands, 2002.
- [12] J.R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec 1969.
- [13] Trevor Jim. Rank 2 type systems and recursive definitions. Technical report, MIT, 1995.
- [14] Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.
- [15] Mark P. Jones. Typing Haskell in Haskell. <http://www.cse.ogi.edu/~mpj/thih/>, 2000.
- [16] A.J. Kfoury and J.B. Wells. A Direct Algorithm for Type Inference in the Rank-2 Fragment of Second-Order lambda-Calculus. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 196–207, 1994.
- [17] A.J. Kfoury and J.B. Wells. Principality and Decidable Type Inference for Finite-Rank Intersection Types. In *Principles of Programming Languages*, pages 161–174, 1999.
- [18] A.J. Kfoury and J.B. Wells. Principality and Type Inference for Intersection Types Using Expansion Variables. *Theoretical Computer Science*, 311:1–70, 2003.

- [19] Konstantin Laufer and Martin Odersky. Polymorphic Type Inference and Abstract Data Types. Technical report, Loyola University of Chicago, 1994.
- [20] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.
- [21] John C. Mitchell and Gordon D. Plotkin. Abstract Types Have Existential Type. *ACM TOPLAS*, 10:470–502, Jul 1988.
- [22] Martin Odersky and Konstantin Laufer. Putting Type Annotations to Work. In *Principles of Programming Languages*, pages 54–67, 1996.
- [23] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type Inference with Constrained Types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, 1997.
- [24] Nigel Perry. The Implementation of Practical Functional Programming Languages, 1991.
- [25] Simon Peyton Jones. *Haskell 98, Language and Libraries, The Revised Report*. Cambridge Univ. Press, 2003.
- [26] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types, 2003.
- [27] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
- [28] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [29] J.C. Reynolds. Towards a theory of type structure. In *Proc. Coll. sur la Programmation*, LNCS, pages 408–425, 1974.
- [30] Martijn Schrage. Proxima. A generic presentation-oriented XML editor. <http://www.cs.uu.nl/groups/ST/Center/Proxima>, 2004.
- [31] Mark Shields and Simon Peyton Jones. First-class Modules for Haskell. In *Ninth International Conference on Foundations of Object-Oriented Languages (FOOL 9)*, Portland, Oregon, Dec 2001.
- [32] Utrecht University Software Technology Group. UUST library. <http://cvs.cs.uu.nl/cgi-bin/cvsweb.cgi/uust/>, 2004.
- [33] S. Doaitse Swierstra. Parser Combinators, from Toys to Tools. In *Haskell Workshop*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, 2000.
- [34] S. Doaitse Swierstra and Pablo R. Azero Alcocer. Fast, Error Correcting Parser Combinators: A Short Tutorial. In *SOFSEM'99 Theory and Practice of Informatics*, LNCS, pages 111–129, Nov 1999.
- [35] S.D. Swierstra, P.R. Azero Alcocer, and J. Saraiava. Designing and Implementing Combinator Languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, LNCS, pages 150–206. Springer-Verlag, 1999.
- [36] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.
- [37] Phil Wadler. Theorems for free! In *4'th International Conference on Functional Programming and Computer Architecture*, sep 1989.

# Index

$\lambda$ -calculus, 10

abstract syntax, 16  
algebra, 21  
assumptions, 29

binding group, 37

concrete syntax, 16  
constraint, 43  
context, 29  
contravariance, 32, 68, 70  
contravariant, 70  
covariant, 70

environment, 29  
existential quantification, 64

fixed type variable, 58  
fresh type variable, 49

impredicativity, 68  
infinite type, 46  
instantiating context, 68  
instantiation, 56

judgement, 28

occur check, 46

partial type signature, 12

semantic function, 20  
specialization, 56  
strong context, 69  
substitution, 44  
subsumption, 67

type, 25  
type checking, 30  
type inference, 30  
type scheme, 56  
type variable, 42  
typing rule, 27

weak context, 69