Stepping through Haskell

Stapsgewijs door Haskell
(met een samenvatting in het Nederlands)

Proefschrift ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de Rector Magnificus, Prof. dr. W. H. Gispen, ingevolge het besluit van het College voor Promoties in het openbaar te verdedigen op maandag 14 november 2005 des middags te 14.30 uur

door

Atze Dijkstra

geboren op 19 januari 1960 te Kerensheide (Beek), Nederland

promotor: Prof. dr. S. Doaitse Swierstra, Universiteit Utrecht

Department of Information and Computing Sciences, Universiteit Utrecht

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Printed by Ridderprint offsetdrukkerij BV, Ridderkerk.
Cover illustration *"λ strolling on the beach"* by Atze Dijkstra.

ISBN 90-393-4070-6

CONTENTS

Contents

1 Introduction

This thesis contains a description of an implementation of an extended subset of the programming language Haskell. At the same time it is an experiment in merging the description with the actual code of the implementation, thus guaranteeing some form of consistency. Similarly, we guarantee consistency between type rules and their implementation by using our *Ruler* system. The thesis is also about making description and implementation understandable, so that it can be used as a means for education and (Haskell) programming language research. In this thesis we take a new and stepwise approach to both description and implementation, which starts with a simple version of Haskell and then, step by step, we extend this simple version with (amongst other things) mechanisms for the use of explicit (type) information when implicit mechanisms are inadequate.

The reason to set out on this journey lies in the observation that Haskell [84] has become a complex language. Haskell includes many productivity increasing features, some of which are of a more experimental nature. Although also intended as a research platform, realistic compilers for Haskell [75] have grown over the years and understanding and experimenting with those compilers is not easy. Experimentation on a smaller scale is usually based upon relatively simple and restricted implementations [46], often focusing only on a particular aspect of the language and/or its implementation.

A second reason is that experimenting with Haskell, or language design in general, usually expresses itself in a theoretical setting, with a focus on the proof of formal properties. Experimentation also takes place in a practical setting, but often it is not at all obvious how theory and practice of an experiment relate. We feel that a description of a Haskell implementation which focusses on a joint presentation of the implementation and its formal representation (by means of type rules), and their mutual consistency, helps to bridge the gap between theory and practice.

The complexities of Haskell and its interaction with experimental features cannot be avoided; even more, it is desirable, as Haskell plays an important role in programming language research. Some of those experimental features turn into useful language constructs, increasing the language complexity as a consequence. The complexity can be made more manageable by looking at features individually, while not loosing sight of the context with which such a feature has to coexist. This thesis aims at walking somewhere between this

complexity and simplicity by describing features individually, as separate aspects of a more complex whole.

In the following section we first give an overview of our goals, the overall organisation of the road to their realisation, and the organisation of this thesis. The remainder of the introduction takes the reader on a short tour through the languages for which this thesis describes implementations.

1.1 Overview

Holy and less holy grails: thesis goals The holy grail of the EHC project [19], described by this thesis, is to create the following:

- A compiler for Haskell plus extensions, in particular an explicit form of Haskell. The goal is a compiler that can be used, and is not 'just' a toy.
- A description of the compiler, embedded in a generic infrastructure for further experimentation. The goal is to offer understanding of the implementation so it can be used for teaching, be a basis for experimentation, and be a bridge between (type) theory and practice.

Unlike a holy grail, we intend to reach our goal, but we do not expect this to happen overnight. However, our inspiration comes from:

- TEX, for which documented source code constitutes a stable point of reference [60].
- Pierce's book [91], which provides a combination of type theory and implementation of type checkers for discussed type systems, and Jones documented Haskell implementation [48].
- Work on the combination of explicit and implicit (type) information [92, 81, 87].

On a less ambitious scale, this thesis can be read with the following, somewhat overlapping, viewpoints in mind; each viewpoint comes with its own goal and design starting point:

- The thesis offers an explanation of the implementation of a Hindley-Milner type system (Chapter 3 through Chapter 5). The explanation is expressed and explained in terms of type rules, attribute grammar implementation for those type rules, and additional infrastructure (encoded in Haskell) to extend it to a full compiler.
- The thesis offers experiments in combining explicitly specified type information (by the programmer), and implicitly inferred type information (by the system) (Chapter 6 through Chapter 10). The design starting point here is to let programmer and system jointly specify (the types in) a program, instead of the programmer fighting

the system's limitations. This starting point is inspired by the observation that systems purely based on type inference hinder a programmer, because such a system does not allow the programmer to specify what the system cannot infer. In particular, we exploit (higher-ranked) type annotations and allow explicit parameter passing for implicit parameters (as specified by means of class predicates).

- The thesis offers an experiment in the description of a compiler, the required mechanisms to do such a job, and the tools to implement those mechanisms (Chapter 2, Chapter 11, and other tools). The design starting point is to partition the whole system into a sequence of steps, each representing a standalone language and implementation. Each step extends a previous step by adding some features. All our tools therefore somehow need to be aware of the notion of separate steps (or views, versions).
- The thesis offers an experiment in creating a maintainable and explainable compiler. The design starting point is that these aspects are best served by consistency, and consistency between artefacts is best implemented by avoiding the inconsistency created by duplication of shared material in the first place. The code presented in the thesis and used for the construction of different versions of the compiler are generated from common source code. The same holds for (the major part of) the type rules and their attribute grammar implementation (Chapter 11).

Thesis contribution Our work complements Pierce's book [91] in the sense that we bring together type rules and their implementation, instead of treating these separately. On the other hand, Pierce provides proofs for various properties, we don't. Later, when dealing with impredicativity (Chapter 7) and in the conclusion (Chapter 12), we come back to this. On a more concrete level, our contribution is summarized by the following:

- We describe a stepwise implementation of an extended subset of Haskell, with the guarantee that the presented source code and type rules[1] are consistent with the implementation. All the following extensions and tools have been implemented [19].
- We describe an algorithm for the propagation of impredicative type information, similar to Pottier nd Rémy [94, 95, 93], but taking the propagation a step further by combining quantifier information from multiple constraints on types (Chapter 7).
- We describe a proposal for explicitly passing parameters for functions expecting values for (implicit) class constraints (Chapter 9).
- We describe a proposal for partial type signatures (Chapter 10).
- We describe *Ruler*, a system for specifying type rules, which generates both LaTeX pretty printed and partial Attribute Grammar implementation for the type rules. The system is used throughout this thesis.

In the conclusion (Chapter 12) we will further detail this.

[1] See appendix B for which rules this is the case.

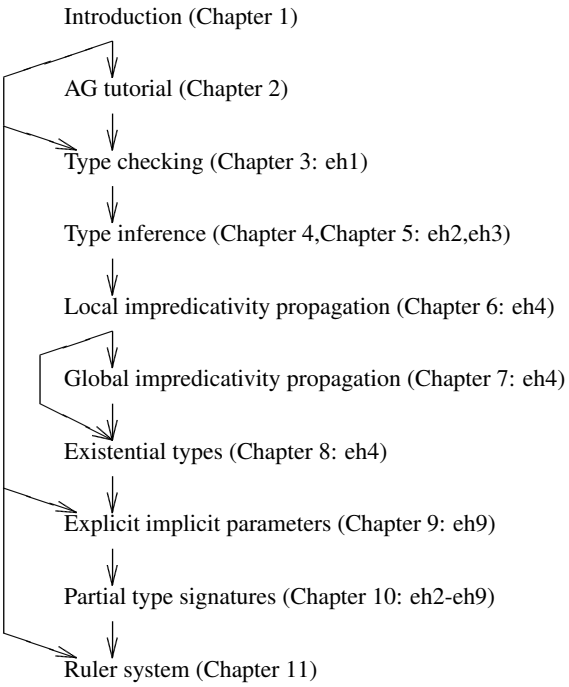

Figure 1.1.: Thesis roadmap

Thesis overview We assume the reader is familiar with Haskell. We do not assume familiarity with the Attribute Grammar system (AG) used throughout this thesis. An AG tutorial is included (Chapter 2), but can safely be skipped if the reader is already familiar with the AG system (see the roadmap in Fig. 1.1). For the remainder of this thesis we assume familiarity with type systems; we informally introduce the necessary concepts, but, as we focus on their implementation, the formalities of type systems are not discussed in this thesis.

Chapter 3 through Chapter 8 are to be read in the proper order, as they describe development steps of the first four of our series of compilers. Chapter 7 has a more experimental character as it is not (yet) integrated into the last compiler version, so this part can be skipped. Chapter 10 describes how we can make the specification of a type signature easier; this subject is relatively independent of the previous chapters. Chapter 9 and Chapter 11 can be read in isolation, as these chapters are based on (submitted) articles.

Throughout the description of the series of compilers we gradually shift from AG based

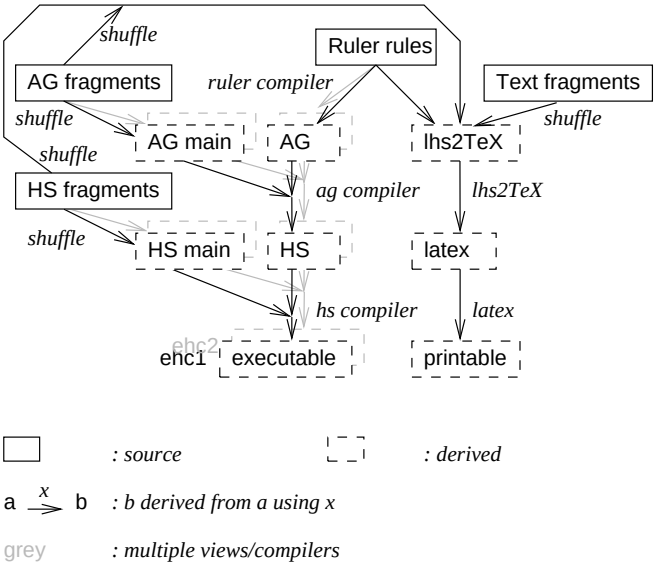

Figure 1.2.: System overview

explanation to type rule based explanation combined with examples. The code which implements those type rules is still referred to from the main text. In the paper version of this thesis we refer to the electronic version [19] by means of w^{w}_{W} in the margin (like here); in the electronic version we refer to the proper appendix with a margin reference (like this reference to the current page). We will come back to this in our conclusion (Chapter 12).

w^{w}_{W}

p 5, 1.1

System overview The core of this thesis consists of a sequence of compilers, together named the Essential Haskell project, abbreviated by EH. The construction of these compilers and this thesis is accomplished by the use of several tools (Fig. 1.2):

- Fragment administration and combination (of source text) is managed by a tool called *Shuffle*. *Shuffle* has the same responsibility as weaving and tangling in literate programming [6], but allows a more flexible recombination of source fragments. Although we occasionally will refer to *Shuffle*, we will not discuss *Shuffle* because of its resemblance to similar tools. w^{w}_{W}
- The *Ruler* system (Chapter 11) generates a visual rendering for inclusion in this thesis, and an AG implementation.
- The specification of the (remaining parts of the) implementation is expressed by an

EH	Feature(s)	Implemented	Chapter	In GHC
1	type checking	+	3	+
2	type inference	+	4	+
3	Hindley-Milner type inference	+	5	+
4	higher-ranked types, existential types	+	6, 7, 8	+, existential types tied to data types
5	data types	+	-	+
6	kind inference, kind polymorphism	+	-	+/-, no kind polymorphism
7	records (non-extensible)	+	-	+/-, tied to data types
8	code-generation	+/-	-	+
9	class system, explicit implicit params	+	9	+/-, class system only
10	records (extensible)	+	-	-
2 -	partial type signatures	+	10	-

Figure 1.3.: Implementation status

Attribute Grammar (AG) and Haskell code.

The partitioning into multiple steps, called *views* in the remainder of this thesis, influences the tools used. A view extends another view by adding new material or replacing old material. Either a tool manages views itself, or is unaware of views. For example, the *Ruler* system manages named views on type rules. However, Haskell compilers are not aware of our idea of views, so *Shuffle* takes care of properly generating source code for views.

Context of this thesis, project status This thesis represents a snapshot of the description of an ongoing project, the Essential Haskell (EH) project. From the www site [19] of the EH project both the source code and the electronic version of the thesis can be downloaded; the electronic version of the thesis includes more material than the paper version, and will be updated regularly.

Fig. 1.3 presents an overview of the features which are described in this thesis, and which features are present in GHC (version 6.4 [75]); the many features present in GHC but not in EH have not been included. We have described the most informative parts: the first chapters provide the basic structure of EH, presenting type rules, and AG implementation side by side. Remaining chapters describe experiments with explicit and implicit (type) information, and the *Ruler* tool.

The source code presented in this thesis is exactly the same code as used for the implementation. Although the source code forms the core around which the thesis has been constructed, only the first chapters of this thesis have incorporated source code. Later chapters involve more complex issues, which, we feel, are better presented using examples and type rules.

Much of the type rule related AG code is generated directly from the type rules by *Ruler* (Chapter 11). This is not (yet) done for all type rules (in particular, not for Chapter 9); appendix B provides a list of the type rules for which AG code is generated. The use of *Ruler* allows us to focus on type rules, but work still needs to be done (see also the conclusion, Chapter 12) to allow even more AG code to be replaced by *Ruler* generated AG code.

Between Chapter 8 and Chapter 9 data types, kind inference/checking, code generation for GRIN [14, 13], and extensible records appear (Fig. 1.3). Most of these features are required for the implementation of Chapter 9; their description will be included in later versions of the full description of EH, to appear after the publication of this thesis.

Parts of this thesis have been published in an earlier version in a different form: Chapter 1 through Chapter 5 are updated from an AFP (Advanced Functional Programming) tutorial [21], and Chapter 9 and Chapter 11 are available as technical reports [23, 24].

Additional work in the following areas is currently in progress:

- A compiler for GRIN [14, 13] to C– [3]. We are also looking at extending GRIN with exceptions.
- Extending EH with GADT's (Generalised Algebraic Data Types).

1.2 A short EH tour

Although all compilers described in this thesis deal with a different issue, they all have in common that they are based on the λ-calculus, most of the time using the syntax and semantics of Haskell. The first version of our series of compilers therefore accepts a language that most closely resembles the λ-calculus, in particular typed λ-calculus extended with **let** expressions and some basic types and type constructors such as *Int*, *Char* and tuples (see appendix A.2 for all terms used throughout this thesis).

We note that code generation is included in EH8. Although we call all compilers 'compiler', the versions before EH8 actually are type checkers as no translation to an equivalent executable program is computed.

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

let *i* :: *Int*

```
    i = 5
in  i
```

All variables need to be typed explicitly; absence of an explicit type is considered to be an error. The corresponding compiler (EH version 1, Chapter 3) 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*, more complex 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 (fst (id 3, 'x'))
```

All types are monomorphic.

EH version 2: Explicit/implicit typing The next version (EH version 2, Chapter 4) no longer requires the explicit type specifications, which in that case will be inferred by the compiler. For example for:

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

the compiler will infer the type specification *i* :: *Int*.

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

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

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

EH version 3: Polymorphism The third version (EH version 3, Chapter 5) performs standard Hindley-Milner type inferencing [16, 17] which also supports parametric polymorphism. For example,

```
let id = λx → x
in  id 3
```

is inferred to have type *id* :: *Int*; *id* has type *id* :: ∀ *a*.*a* → *a*.

A (polymorphic) type for a value can also be specified explicitly

let $id :: a \to a$
  $id = \lambda x \to x$
in $id\ 3$

The type signature is checked against the inferred type.

EH version 4: Higher ranked types Standard Hindley-Milner type inferencing cannot infer polymorphic parameters, so-called higher-ranked types. A higher-ranked type may have a quantified type at an argument position, allowing for polymorphic function arguments. In general, this is a hard thing to do and even impossible for rank-3 (and higher) types [40, 53, 54, 55], so the fourth version (EH version 4, Chapter 6) does not infer this type information, but allows for explicitly specified polymorphism for (e.g.) parameters.

For example, the following is allowed.

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

Note that the type signature is thus required here.

This version also provides some notational sugaring by allowing one to omit the explicit quantifiers from the type signature (separately discussed in Chapter 10). For example, if the universal quantifier $\forall$ 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 $f :: (a \to a) \to (Int, Char)$
  $f = \lambda i \to (i\ 3, i\ \texttt{'x'})$
in f

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

Specifying a complete type signature can be difficult for complicated types, so it is also permitted to leave argument and results of a function unspecified using a *partial type signature* (separately discussed in Chapter 10).

let $id :: \forall\ a.a \to a$
  $id = \lambda x \to x$
  $f\ \ :: (\forall\ a.a \to a) \to ...$
  $f\ \ = \lambda i \to (i\ 3, i\ \texttt{'x'})$
in $f\ id$

Here, for f only the part that cannot be inferred is given in the signature.

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

let $id\ \ :: \forall\ a.a \to a$
  $xy\ \ :: \exists\ a.(a, a \to Int)$
  $xy\ \ = (3, id)$
  $ixy :: (\exists\ a.(a, a \to Int)) \to Int$

```
    ixy = λ(v,f) → f v
    xy' = ixy xy
    pq :: ∃ a.(a, a → Int)
    pq = ('x', id)    -- ERROR
in  xy'
```

The tuple *xy* contains an *Int* (this type information is 'forgotten' via existential quantification) and a function constructing an *Int* from the value of which the type has been hidden. Access to the elements of such a tuple is done by pattern matching, as in the argument position of the function *ixy*. The attempt to construct *pq* fails.

When a value of an existentially quantified type is opened, that is, it is bound to a value identifier, the hidden type becomes visible in the form of a fresh type constant.

EH version 5: Data types The fifth version (EH version 5, not included in this thesis) adds **data** types and opening/unpacking/scrutinizing a data type value by means of a **case** expression.

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

EH version 6: Kinding The previous version allows incorrect programs because data types can be used incorrectly (type signature declarations without a corresponding value declaration are allowed until the code generation version of EH):

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

The type of *v* is not a type of a value, and thus the type of *v* itself is not well-typed. The sixth version (EH version 6, not included in this thesis) adds kind (that is, the type of a type) inferencing.

With the notion of the kind of a type we also allow the notion of polymorphism for kinds:

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

infers for type constructor *Eq*:

```
Eq :: ∀k.k → k → *
```

Explicit kind signatures for types are also allowed, similar to type signatures for values.

EH version 7: Non-extensible records The seventh version (EH version 7, not included in this thesis) extends tuples to (non-extensible) records. Fields can be named. For tuples the default field names are their position (starting at 1):

```
let r = (i = 3, c = 'x', id = λx → x)
    s = (r | c := 5)
in  let v = (r.id r.i, r.id r.c)
        vi = v.1
    in  vi
```

The proposal by Jones [49] is followed.

EH version 8: Code generation The eighth version (EH version 8, not included in this thesis) adds code generation for a GRIN (Graph Reduction Intermediate Notation) like backend [14, 13]. The generated code can be run (for testing purposes only) by a small GRIN interpreter (*grini*).

EH version 9: Implicit parameters, class system The ninth version (EH version 9, Chapter 9) adds a class system, and explicit parameter passing to implicit parameters:

```
let data List a = Nil | Cons a (List a)
    class Eq a where
      eq :: a → a → Bool
      ne :: a → a → Bool
    instance dEqInt ⇜ Eq Int where     -- (1)
      eq = primEqInt
      ne = λx y → not (eq x y)
    nub :: ∀ a.Eq a ⇒ List a → List a
    nub = λxx → case xx of
                  Nil         → Nil
                  Cons x xs → Cons x (nub (filter (ne x) xs))
    eqMod2 :: Int → Int → Bool
    eqMod2 = λx y → eq (mod x 2) (mod y 2)
    n1 = nub (!dEqInt ⇜ Eq Int!)          -- (2)
             (Cons 3 (Cons 3 (Cons 4 Nil)))
    n2 = nub (!(eq = eqMod2               -- (2)
                ,ne = λx y → not (eqMod2 x y)
                ) ⇜ Eq Int
              !)
             (Cons 3 (Cons 3 (Cons 4 Nil)))
in  ...
```

On top of a class system, we allow instances to be named (1), and passed explicitly (2) when expected to be passed implicitly.

EH version 10: Extensible records The tenth version (EH version 10, not included in this thesis) adds extensible records (again following Jones [49]), using the class system to allow:

$$
\begin{array}{l}
\textbf{let}\ add :: Int \to Int \to Int \\
\quad f \quad :: (r\backslash x, r\backslash y) \Rightarrow (r \mid x :: Int, y :: Int) \to Int \\
\quad f \quad = \lambda r \to add\ r.x\ r.y \\
\textbf{in} \\
\textbf{let}\ v_1 = f\ (x = 3, y = 4) \\
\quad v_2 = f\ (y = 5, a = \texttt{'z'}, x = 6) \\
\textbf{in}\ v_2
\end{array}
$$

Tuple access functions can also be used in a more general way:

$$
\begin{array}{l}
\textbf{let}\ snd = \lambda r \to r.2 \\
\textbf{in} \\
\textbf{let}\ v_1 = snd\ (3, 4) \\
\quad v_2 = snd\ (3, 4, 5) \\
\textbf{in}\ v_2
\end{array}
$$

2 Attribute Grammar (AG) system tutorial

This chapter contains a small tutorial on the Attribute Grammar (AG) system used to describe the greater part of the implementation of the compilers in this thesis. The tutorial explains the basic features of the AG system. The explanation of remaining features is postponed until the end of the tutorial in the form of a list of explanations which should be read within the context referred to (which occurs later in the thesis). The tutorial can safely be skipped if the reader is already familiar with the AG system.

2.1 Haskell and Attribute Grammars (AG)

Attribute grammars can be mapped onto functional programs [61, 41, 12]. Vice versa, the class of functional programs (catamorphisms [74, 28, 104]) mapped onto can be described by attribute grammars. The AG system exploits this correspondence by providing a notation (attribute grammar) for computations over trees which additionally allows program fragments to be described separately. The AG compiler gathers these fragments, combines these fragments, and generates a corresponding Haskell program.

In this AG tutorial we start with a small example Haskell program to show how the computation described by this program can be expressed in the AG notation and how the resulting Haskell program generated by the AG compiler can be used. The 'repmin' problem [12] is used for this purpose. A second example describing a 'pocket calculator' (that is, expressions) focusses on more advanced features and typical AG usage patterns.

2.2 Repmin a la Haskell

Repmin stands for "replacing the integer valued leaves of a tree by the minimal integer value found in the leaves". The solution to this problem requires two passes over a tree structure, computing the miminum and computing a new tree with the minimum as its leaves respectively. It is often used as the typical example of a circular program which

lends itself well to be described by the AG notation. When described in Haskell it is expressed as a computation over a tree structure:

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

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

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

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

```
tr  = Tree_Bin (Tree_Leaf 3) (Tree_Bin (Tree_Leaf 4) (Tree_Leaf 5))
tr' = repmin tr
```

This program produces the following output:

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

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

2.3 Repmin a la AG

The structure of *repmin* is similar to the structure required by a compiler. A compiler performs several computations over an *abstract syntax tree* (*AST*), for example for computing its type and generating code. This corresponds to the *Tree* structure used by *repmin* and the tupled results. In the context of attribute grammars the elements of this tuple are called

attributes. Occasionaly the word *aspect* is used as well, but an aspect may also refer to a group of attributes associated with one particular feature of the AST, language or problem at hand.

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

Using AG notation we first define the AST corresponding to our problem (for which the complete solution is given in Fig. 2.1):

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

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

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

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

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

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

Rules relating an attribute to its value are introduced using the keyword **sem**. For each production we distinguish a set of input attributes, consisting of the synthesized attributes of the children referred to by $@\langle child\rangle.\langle attr\rangle$ and the inherited attributes of the parent

```
data Tree
  | Leaf int  : {Int}
  | Bin  lt   : Tree
         rt   : Tree
attr Tree [|| min : Int]
sem Tree
  | Leaf lhs . min  = @int
  | Bin  lhs . min  = @lt.min `min` @rt.min
attr Tree [rmin : Int ||]
  -- The next SEM may be generated automatically
sem Tree
  | Bin  lt   . rmin = @lhs.rmin
         rt   . rmin = @lhs.rmin
data Root
  | Root tree : Tree
sem Root
  | Root tree. rmin = @tree.min
attr Root Tree [|| tree : Tree]
sem Tree
  | Leaf lhs . tree  = Tree_Leaf @lhs.rmin
  | Bin  lhs . tree  = Tree_Bin  @lt.tree @rt.tree
  -- The next SEM may be generated automatically
sem Root
  | Root lhs . tree  = @tree.tree
deriving Tree : Show
{
tr  = Tree_Bin (Tree_Leaf 3) (Tree_Bin (Tree_Leaf 4) (Tree_Leaf 5))
tr' = sem_Root (Root_Root tr)
}
{
main ::  IO ()
main =   print tr'
}
```

Figure 2.1.: Full AG specification of repmin

referred to by @**lhs**.⟨*attr*⟩. For each output attribute we need a rule that expresses its value in terms of input attributes and fields.

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

| ⟨*variant*⟩ ⟨*node*⟩.⟨*attr*⟩ = ⟨*Haskell expr*⟩

If multiple rules are declared for a ⟨*variant*⟩ of a node, they may all be listed under the same ⟨*variant*⟩. The same holds for multiple rules for a child (or **lhs**) of a ⟨*variant*⟩, the child (or **lhs**) may then be shared.

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

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

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

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

attr *Tree* [*rmin* : *Int* ||]

The value of *rmin* is straightforwardly copied to its children. This "simply copy" behavior occurs so often that we may omit its specification. The AG system uses so called copy rules to automically generate code for copying if the value of an attribute is not specified explicitly. This is to prevent program clutter and thus allows the programmer to focus on programming the exception instead of the rule. We will come back to this later; for now it suffices to mention that all the rules for *rmin* might as well have been omitted.

The original *repmin* function passed the minimum value coming out *r* back into *r* itself. This happened at the top of the tree; In AG we define a *Root* node sitting on top of a *Tree*:

data *Root*
| *Root tree* : *Tree*

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

sem *Root*
| *Root tree.rmin* = @*tree.min*

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

attr *Root Tree* [|| *tree* : *Tree*]

sem *Tree*
| *Leaf* **lhs**.*tree* = *Tree_Leaf* @**lhs**.*rmin*
| *Bin* **lhs**.*tree* = *Tree_Bin* @*lt.tree* @*rt.tree*

sem *Root*
| *Root* **lhs**.*tree* = @*tree.tree*

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

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

deriving *Tree* : *Show*

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

{
tr = *Tree_Bin* (*Tree_Leaf* 3) (*Tree_Bin* (*Tree_Leaf* 4) (*Tree_Leaf* 5))
tr' = *sem_Root* (*Root_Root tr*)
}

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

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

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

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

In general, generated code is not the most pleasant[1] of prose to look at, but we will have to use the generated functions in order to access the AG computations of attributes from the Haskell world. The following observations should be kept in mind when doing so:

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

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

- A mapping from the Haskell **data** type to the corresponding semantic function is available with the name *sem_⟨node⟩*.

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

- First construct a Haskell value of type ⟨*node*⟩, then apply *sem_⟨node⟩* to this value and the additionally required inherited attributes values. The given function *main* from AG variant of repmin takes this approach.
- Circumvent the construction of Haskell values of type ⟨*node*⟩ by using the semantic functions *sem_⟨node⟩_⟨variant⟩* directly when building the AST instead of the data constructor ⟨*node*⟩_⟨*variant*⟩ (This technique is called deforestation [115].).

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

2.4 Parsing directly to semantic functions

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

```
instance Symbol Char

pRepmin :: IsParser p Char ⇒ p T_Root
pRepmin = pRoot
        where pRoot = sem_Root_Root ⟨$⟩ pTree
              pTree = sem_Tree_Leaf ⟨$⟩ pInt
                    ⟨|⟩ sem_Tree_Bin  ⟨$ pSym 'B' ⟨*⟩ pTree ⟨*⟩ pTree
              pInt  = (λc → ord c − ord '0') ⟨$⟩ '0' ⟨..⟩ '9'
```

The parser recognises the letter 'B' as a *Bin* alternative and a single digit as a *Leaf*. appendix C.1 gives an overview of the parser combinators which are used [101]. The parser is invoked from an alternative implementation of *main*:

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

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

2.5 More features and typical usage: a pocket calculator

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

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

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

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

data *AGItf*
  | *AGItf* *expr* : *Expr*

data *Expr*
  | *IConst* *int* : {*Int*}
  | *Add* e_1 : *Expr*
  e_2 : *Expr*

set *AllNT* = *AGItf* *Expr*

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

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

The computation of a pretty printed representation follows the same pattern as the computation of *min* and *tree* in the *repmin* example, because of its compositional and bottom-up

nature. The synthesized attribute *pp* is synthesized from the values of the *pp* attribute of the children of a node:

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

The pretty printing uses a pretty printing library with combinators for values of type *PP_Doc* representing pretty printed documents. The library is not further discussed here; an overview of some of the available combinators can be found in appendix C.2.

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

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

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

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

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

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

```
attr Expr [env : {[(String, Int)]} ||]
sem Expr
  | Let    body.env = (@nm, @val.val) : @lhs.env
sem AGItf
  | AGItf  expr .env = []
```

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

```
attr AllNT [|| val : Int]
sem Expr
  | Var    lhs.val = maybe 0 id (lookup @nm @lhs.env)
  | Add    lhs.val = @e1.val + @e2.val
  | Let    lhs.val = @body.val
  | IConst lhs.val = @int
```

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

```
attr AllNT [| count : Int |]
sem Expr
  | Add lhs.count = @e2.count + 1
```

The attribute *count* is said to be *threaded* through the AST, the AG solution to a global variable or the use of state monad. This is a result of the attribute being inherited as well as synthesized and the copy rules. Its effect is an automatic copying of the attribute in a preorder traversal of the AST. The children nodes of the *Add* variant update the *count* value; the *Add* variant increments this value and passes the result to the parent node.

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

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

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

- (inherited) @**lhs**.*count* is copied to (inherited) @*val*.*count*,

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

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

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

As with our previous example we can let a parser map input text to the invocations of semantic functions. For completeness this source text has been included in Fig. 2.2. The result of parsing combined with the invocation of semantic functions will be a function taking inherited attributes to a tuple holding all synthesized attributes. Even though the order of the attributes in the result tuple is specified, position based extraction via pattern matching should be avoided. The AG system can be instructed to create a wrapper function which knows how to extract the attributes out of the result tuple:

wrapper *AGItf*

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

```
run :: Int → IO ()
run  count
  = do hPutStr stdout "Enter expression: "
       hFlush stdout
       l ← getLine
       r ← parseIOMessage show pAGItf l
       let r' = wrap_AGItf r (Inh_AGItf{count_Inh_AGItf = count})
       putStrLn ("Expr='"                    ++ disp (pp_Syn_AGItf r') 40 "" ++
                 "', val="                   ++ show (val_Syn_AGItf r')      ++
                 ", add count thus far=" ++ show (count_Syn_AGItf r')
                )
       run (count_Syn_AGItf r')
main :: IO ()
main = run 0
```

We face a similar (that is, position based) problem with the passing of inherited attributes to the semantic function. Hence inherited attributes are passed to the wrapper function via a data type with name *Inh_*⟨*node*⟩ and a constructor with the same name, with fields having

labels of the form ⟨*attr*⟩*_Inh_*⟨*node*⟩. The *count* attribute is an example of an attribute which must be passed as an inherited attribute as well as extracted as a synthesized attribute.

This concludes our introduction to the AG system. Some topics have either not been mentioned at all or only shortly touched upon. We provide a list of those topics together with a reference to their first use. All the following topics should be read within the context of their referred use.

- **Type synonyms (for lists)** *(Context and example at page 29)*: The AG notation allows type synomyms for one special case, AG's equivalent of a list (e.g *Decls* in Fig. 3.2). It is an often occurring idiom to encode a list of nodes, say **data** *L* with elements ⟨*node*⟩ as:

  data *L*
  | *Cons hd* : ⟨*node*⟩
  tl : *L*
  | *Nil*

  AG allows the following notation as a shorthand:

  type *L* = [⟨*node*⟩]

- **Left hand side patterns** *(Context and example at page 46)*: The simplest way to define a value for an attribute is to define one value for one attribute at a time. However, if this value is a tuple, its fields are to be extracted and assigned to individual attributes (as in *valGamLookupTy*). AG allows a pattern notation of the form(s) to make the notation for this situation more concise:

  | ⟨*variant*⟩ ⟨*node*⟩.(⟨*attr*$_1$⟩ , ⟨*attr*$_2$⟩ , ...) =
  | ⟨*variant*⟩ (⟨*node*$_1$⟩.⟨*attr*$_1$⟩, ⟨*node*$_1$⟩.⟨*attr*$_2$⟩, ...) =

- **Set notation for variants** *(Context and example at page 52)*: The rule for (e.g.) attribute *fo* is specified for *IConst* and *CConst* together. Instead of specifying only one variant a whitespace separated list of variant names may be specified after the vertical bar '|'. It is also allowed to specify this list relative to all declared variants by specifying for which variants the rule should *not* be declared. For example: ∗ − *IConst CConst* if the rule was to be defined for all variants except *IConst* and *CConst*.
- **Local attributes** *(Context and example at page 43)*: Attribute *fo_* and *ty* are declared locally. In this context 'local' means that the scope is limited to the variant of a node. For example, *fo_* (explained further below) also holds a (possibly empty) list of errors used by other attribute equations. *fo_* is only available for attribute equations for variant *IConst* of *Expr*.
- **Attribute together with use** *(Context and example at page 53)*: A synthesized attribute ⟨*attr*⟩ may be declared together with **use**{⟨*op*⟩}{⟨*zero*⟩}. The ⟨*op*⟩ and ⟨*zero*⟩

```
instance Symbol Char
pAGItf :: IsParser p Char ⇒ p T_AGItf
pAGItf = pRoot
    where pRoot     = sem_AGItf_AGItf ⟨$⟩ pExpr
          pExpr     = pChainr (sem_Expr_Add ⟨$ pSym '+') pExprBase
          pExprBase = (sem_Expr_IConst.foldl (λl r → l ∗ 10 + r) 0)
                        ⟨$⟩ pList1 ((λc → ord c − ord '0') ⟨$⟩ '0' ⟨..⟩ '9')
                      ⟨|⟩ sem_Expr_Let
                        ⟨$ pSym '[' ⟨∗⟩ pNm ⟨∗ pSym '=' ⟨∗⟩ pExpr
                        ⟨∗ pSym ':' ⟨∗⟩                    pExpr
                        ⟨∗ pSym ']'
                      ⟨|⟩ sem_Expr_Var ⟨$⟩ pNm
          pNm       = (:"") ⟨$⟩ 'a' ⟨..⟩ 'z'
```

Figure 2.2.: Parser for calculator example

allow the insertion of copy rules which behave similar to Haskell's *foldr*. The first piece of text ⟨*op*⟩ is used to combine the attribute values of two children by textually placing this text as an operator between references to the attributes of the children. If no child has an ⟨*attr*⟩, the second piece of text ⟨*zero*⟩ is used as a default value for ⟨*attr*⟩. For example, **use**{‘*gamAddGam*‘}{*emptyGam*} gathers bottom-up the type signature bindings.

- **Attribute of type self** *(Context and example at page 60)*: The use of the keyword **self** in:

  attr⟨*node*⟩[|| *a* : **self**]

  is equivalent to

  attr⟨*node*⟩[|| *a* : ⟨*node′*⟩]

  where:

  - The type ⟨*node′*⟩ of synthesized attribute *a* stands for the generated Haskell **data** type generated for ⟨*node*⟩.
  - For each ⟨*node*⟩ variant a local attribute *a* is inserted, defined to be a replica of the ⟨*node*⟩ variant in terms of ⟨*node′*⟩.
  - Copy rules automatically propagate the local *a* to the parent, unless an explicit definition for the parent's *a* is given.

  For example, via attribute *repl* a copy of the type is built which only differs (or, may differ) in the original in the value for the type variable.

- Rule redefinition via :=.
- Cycle detection and other (experimental) features, commandline invocation, etc.

3 EH 1: Typed λ-calculus

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

let i :: *Int*
  $i = 5$
in i

is accepted, whereas

let i :: *Char*
  $i = 5$
in i

produces a pretty printed version of the erroneous program, annotated with errors. Type errors are reported in terms of a failed 'fit' (<=) which is our mechanism for matching, or fitting (because of the asymmetry in later EH versions), two types:

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

Type signatures have to be specified for identifiers bound in a **let** expression. A **let** expression allows mutually recursive definitions. For λ-expressions the type of the parameter can be extracted from these type signatures, unless a λ-expression occurs at the position of an applied function. In that case a type signature for the λ-expression is required in the

expression itself. This program will not typecheck because a $Char \to Char$ function is applied to an Int.

```
let v :: (Int, Char)
    v = ( (λf → (f 3, f 'x'))
            :: (Char → Char) → (Int, Char)
        ) (λx → x)
in  v
```

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

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

3.1 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 (see Fig. 3.1 for the term and type expression language). 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; translation from the abstract to the concrete representation is done by a pretty printer.

Let us focus our attention first on the abstract syntax for EH1, in particular the part defining the structure for expressions (the remaining abstract syntax can be found in Fig. 3.2):

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

Values (expressions, terms):		
e ::=	$int \mid char$	literals
|	i	program variable
|	$e\ e$	application
|	**let** $\overline{d}$ **in** e	local definitions
|	$\lambda p \rightarrow e$	abstraction
|	$e :: t$	type annotated expression
Declarations of bindings:		
d ::=	$i :: t$	value type signature
|	$p = e$	value binding
Pattern expressions:		
p ::=	$int \mid char$	literals
|	i	pattern variable
|	$i\ @p$	pattern variable, with subpattern
|	$(p, ..., p)$	tuple pattern
Identifiers:		
ι ::=	i	lowercase: (type) variables
|	I	uppercase: (type) constructors
Type expressions:		
t ::=	$Int \mid Char$	type constants
|	$t \rightarrow t$	function type
|	$(t, ..., t)$	tuple type

Figure 3.1.: EH1 terms

$expr$: $Expr$

Integer constants are represented by *IConst*, lowercase (uppercase) identifier occurrences by *Var* (*Con*), an *App* represents the application of a function to its argument, *Lam* and *Let* represent lambda expressions and let expressions. The *Con*, *Paren*, and *AppTop* alternatives do not have a direct counterpart in EH1's term language: *Con* is used for constructors (for now only tuples), *Paren* is used to encode parenthesis (for pretty printing), and *AppTop* encodes the top of an application (*App*) spine; we will later come back to the use of these additional alternatives. The abstract syntax for value expressions, pattern expressions, and type expressions also are similar in their structure for constants, constructors (*Con*), variables (*Var*), and application (*App*). The term language not always directly reflects this structure; we will also come back to this.

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

$$\begin{array}{l}\textbf{let } ic\ @(i,c) = (5, \texttt{'x'}) \\ \quad id \qquad\quad = \lambda x \rightarrow x \\ \textbf{in } id\ i\end{array}$$

is represented by the following piece of abstract syntax tree:

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

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

Looking at this example and the rest of the abstract syntax in Fig. 3.2 we can make several observations of what one is allowed to write in EH and what can be expected from the implementation:

```
data AGItf
  | AGItf    expr    : Expr
data Decl
  | TySig    nm      : {HsName}
             tyExpr  : TyExpr
  | Val      patExpr : PatExpr
             expr    : Expr
type Decls   = [Decl]
set AllDecl  = Decl Decls
data PatExpr
  | IConst   int     : {Int}
  | CConst   char    : {Char}
  | Con      nm      : {HsName}
  | Var      nm      : {HsName}
  | VarAs    nm      : {HsName}
             patExpr : PatExpr
  | App      func    : PatExpr
             arg     : PatExpr
  | AppTop   patExpr : PatExpr
  | Parens   patExpr : PatExpr
set AllPatExpr = PatExpr
data TyExpr
  | Con      nm      : {HsName}
  | App      func    : TyExpr
             arg     : TyExpr
  | AppTop   tyExpr  : TyExpr
  | Parens   tyExpr  : TyExpr
set AllTyExpr  = TyExpr
set AllExpr    = Expr
set AllNT      = AllTyExpr AllDecl AllPatExpr AllExpr
```

Figure 3.2.: Abstract syntax for EH (without Expr)

- There is a striking similarity between the structure of expressions *Expr* and patterns *PatExpr* (and as we will see later type expressions *TyExpr*): they all contain *App* and *Con* variants. This similarity will sometimes be exploited to factor out common code, and, if factoring out cannot be done, leads to similarities between pieces of code. This is the case for the construction of application-like structures (by the parser and the type checker) and pretty printing (not included in this thesis).
- Type signatures (*Decl_TySig*) and value definitions (*Decl_Val*) may be freely mixed.

Declarations may be mutually recursive. However, type signatures and value definitions for the same identifier are still related.

- Because of the textual decoupling of value definitions and type signatures, a type signature may specify the type for an identifier occurring inside a pattern; for simplicity, we forbid this, for example:

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

  Additional analysis would be required to allow this. However, the following is allowed, because we allow type signatures for top-level identifiers:

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

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

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

  Expr_App (*Expr_App* (*Expr_Con* `",2"`) (*Expr_IConst* 2)) (*Expr_IConst* 3)

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

- The naming convention for tuples and other naming conventions are available through the following abstraction of Haskell names *HsName* (we remind the reader that a reference in the margin refers to additional, but not inlined, code).

  w^{W}_{W}

  data *HsName* = *HNm String*
  $\quad$ **deriving** (*Eq*, *Ord*)

  instance *Show HsName* **where**
  $\quad$ *show* (*HNm s*) = *s*

  hsnArrow, *hsnUnknown*, *hsnInt*, *hsnChar*, *hsnWild*
  $\quad :: HsName$
  hsnProd $:: Int \to HsName$
  hsnProdArity $:: HsName \to Int$

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

3.2 Types

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

3.2.1 What is a type

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

The flow of values must be such that "well-typed programs cannot go wrong". 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. If a compiler cannot find an erroneous flow of values, with the notion of erroneous defined by the type system, the program is guaranteed not to crash because of unintended use of bitpatterns.

In this section we start by introducing a type language in a more formal setting as well as a more practical setting. The formal setting uses typing rules to specify the static semantics of EH, whereas in the practical setting the AG system is used, providing an implementation. In the following section we discuss the typing rules, the mechanism for enforcing the equality of types (called *fitting*) and the checking itself. Types will be introduced informally, instead of taking a more formal approach [106, 111, 91, 7].

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

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

However, the following definition is closer to the one used in our implementation:

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

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

We explicitly distinguish between *type expressions* and *type signatures*. A type expression is a term t (Fig. 3.1) specified by the EH programmer, whereas a type signature σ (or type for short) is used internally by the type rules (and their implementation). Only when the difference is significant we distinguish between type expressions and signatures, otherwise we just use the word 'type'.

The corresponding encoding (for types) using AG notation differs in the presence of an *Any* type, also denoted by $\square$. In Section 3.3 we will say more about this. It is used to smoothen the type checking by (e.g.) limiting the propagation of erroneous types:

```
data TyAGItf
   | AGItf ty    : Ty
data Ty
   | Con   nm    : {HsName}
   | App   func  : Ty
           arg   : Ty
   | Any
```

The formal system and implementation of this system use different symbols to refer to the same concept. For example, *Any* in the implementation is the same as $\square$ in the typing rules. Such a similarity is not always pointed out explicitly but instead a notation $name_1 /\!/ name_2$ is used to simultaneously refer to both symbols $name_1$ and $name_2$, for example $Any /\!/ \square$. The notation also implies that the identifiers and symbols separated by '//' are referring to the same concept.

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

```
type TyL = [Ty]
```

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

3.3 Checking types

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

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

3.3.1 Type rules

We start with a simplified set of equational type rules (Fig. 3.3). The full algorithmic version (Fig. 3.5) of this chapter differs in the explicit handling of known type information, the use of patterns, and uncoupling of type signatures and corresponding value declarations. We discuss type expressions, used in rule E.ANN, later in this chapter (Section 3.3.6, page 54).

For example, the following is the typing rule (taken from Fig. 3.3) for function application:

$$\frac{\Gamma \vdash^{e} e_2 : \sigma_a \qquad \Gamma \vdash^{e} e_1 : \sigma_a \to \sigma}{\Gamma \vdash^{e} e_1\ e_2 : \sigma}\ \text{E.APP}_E$$

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

All rules we will use are of the form

$$\frac{\begin{array}{c} prerequisite_1 \\ prerequisite_2 \\ \ldots \end{array}}{consequence}\ \text{AST.RULENAME}_{view}$$

with the meaning that if all $prerequisite_i$ can be proven we may conclude the *consequence*. By convention rule names are typeset in SMALL CAPS font, and have the form AST.RULENAME where AST refers to the language element about which the rule states something (here expressions: E). The suffix *view*, typeset in *italic*, indicates the view on the rule (here the equational view: *E*). We omit the view when referring to a rule.

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

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

$$\boxed{\Gamma \vdash^{e} e : \sigma}$$

$$\frac{}{\Gamma \vdash^{e} int : Int}\ \text{E.INT}_E \qquad \frac{}{\Gamma \vdash^{e} char : Char}\ \text{E.CHAR}_E \qquad \frac{\iota \mapsto \sigma \in \Gamma}{\Gamma \vdash^{e} \iota : \sigma}\ \text{E.VAR}_E$$

$$\frac{\Gamma \vdash^{e} e_2 : \sigma_a \quad \Gamma \vdash^{e} e_1 : \sigma_a \to \sigma}{\Gamma \vdash^{e} e_1\ e_2 : \sigma}\ \text{E.APP}_E \qquad \frac{\Gamma \vdash^{e} e_1 : \sigma_1 \quad \Gamma \vdash^{e} e_2 : \sigma_2}{\Gamma \vdash^{e} (e_1, e_2) : (\sigma_1, \sigma_2)}\ \text{E.PROD}_E$$

$$\frac{(i \mapsto \sigma_i), \Gamma \vdash^{e} e : \sigma_e}{\Gamma \vdash^{e} \lambda i \to e : \sigma_i \to \sigma_e}\ \text{E.LAM}_E \qquad \frac{\vdash^{t} t : \sigma \quad \Gamma \vdash^{e} e : \sigma}{\Gamma \vdash^{e} (e :: t) : \sigma}\ \text{E.ANN}_E \qquad \frac{\Gamma_t, \Gamma \vdash^{d} d : \Gamma_t \quad \Gamma_t, \Gamma \vdash^{e} b : \sigma}{\Gamma \vdash^{e} \textbf{let}\ d\ \textbf{in}\ b : \sigma}\ \text{E.LET}_E$$

Within environment Γ, expression e has type σ.
e : Expression
σ : Type of expression
Γ : Environment $\overline{\iota \mapsto \sigma}$ for value identifiers

Figure 3.3.: Expression type rules (E)

$$\boxed{\Gamma \vdash^{d} d : \Gamma_t}$$

$$\frac{\Gamma \vdash^{e} e_i : \sigma_i \quad \Gamma_i \equiv [i \mapsto \sigma_i]}{\Gamma \vdash^{d} (i :: \sigma_i; i = e_i) : \Gamma_i}\ \text{D.SIG.VAL}_E$$

Within environment Γ, declaration d has type signature bindings Γ_t.
d : Declaration
Γ_t : Environment with type signature bindings
Γ : Environment with known bindings

Figure 3.4.: Declaration type rules (E)

The part "$\leadsto$ *more results*" does not have to be present if there are no more results for a judgement. The division between "*property*" and "*more results*" is somewhat arbitrary as both are results and properties. However, we consider "*property*" to be the most important result; for example, the type in the context of type checking. The notation reads as

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

If the *context* or *more results* itself consists of multiple parts, these parts are separated by a semicolon ';'. An underscore '_' has a similar role as in Haskell to indicate that a property is not relevant for a type rule (for example see rule E.APP, Fig. 3.5)

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

$$judgetype = \lambda construct \rightarrow \lambda context \rightarrow ...(property, more\_results)$$

or a piece of AG:

attr *judgetype* [*context* : ... ||
property : ...*more_results* : ...]
sem *judgetype*
| *construct*
lhs.(*property*, *more_results*) = ... @**lhs**.*context* ...

Typing rules and implementation templates differ in that the latter prescribes the order in which the computation of a property takes place, whereas the former simply postulates relationships between parts of a rule. In general, typing rules presented throughout this thesis will be rather explicit in the flow of information and thus be close to the actual implementation. In Chapter 11 we will exploit the similarity between type rules and their AG counterpart further when discussing the *Ruler* system used for describing the type rules in this thesis.

Finally, we include a (compact) description and a legenda in each set of type rules (like Fig. 3.3) of which its type scheme is introduced or is changed. The description explains how the type scheme for the type rules should be interpreted. The legenda describes the meaning and use of the meta-variables in the type scheme.

3.3.2 Environment

The rules in Fig. 3.3 refer to Γ, which is often called *assumptions*, *environment* or *context* because it provides information about what may be assumed about identifiers. Identifiers ι are distinguished on the case of the first character, capitalized *I*'s starting with an uppercase, uncapitalized *i*'s otherwise:

$$\iota = i \mid I$$

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

An environment Γ is a vector of bindings, a partial finite map from identifiers to types (or any other kind of information):

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

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

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

If convenient, we will also use a list notation:

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

This will be done if specific properties of a list are used or if we borrow from Haskell's repertoire of list functions. For simplicity we also use (assocation) lists in our implementation of an environment Γ, or more precisely, a stack of lists. A list structure suffices to encode the presence of an identifier in an environment Γ, but it cannot be used to detect multiple occurrences caused by duplicate introductions. Thus in our implementation we use a stack of lists instead. We will use the stack-like behavior by adding newly declared identifiers in the top list of the stack, which then can be treated separate from the rest of the stack:

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

newtype *Gam k v* = *Gam* [*AssocL k v*] **deriving** *Show*

emptyGam	::	*Gam k v*	
gamUnit	::	$k \to v$	$\to$ *Gam k v*
gamLookup	:: *Ord k* $\Rightarrow$	$k \to$ *Gam k v*	$\to$ *Maybe v*
gamToAssocL	::	*Gam k v*	$\to$ *AssocL k v*
gamPushNew	::	*Gam k v*	$\to$ *Gam k v*
gamPushGam	:: *Ord k* $\Rightarrow$	*Gam k v*	$\to$ *Gam k v* $\to$ *Gam k v*
gamPop	::	*Gam k v*	$\to$ (*Gam k v*, *Gam k v*)
gamAddGam	:: *Ord k* $\Rightarrow$	*Gam k v*	$\to$ *Gam k v* $\to$ *Gam k v*
gamAdd	:: *Ord k* $\Rightarrow$	$k \to v$	$\to$ *Gam k v* $\to$ *Gam k v*

Entering and leaving a scope is implemented by means of pushing and popping an environment Γ: *gamPushGam* pushes a Γ onto another, *gamAddGam* adds a Γ to another, *gamPushNew* pushes an empty Γ. Extending an environment Γ will take place on the top of the stack only. Left operands are added to right operands, possibly overwriting or hid-

ing entries in the right operand. For example *gamAddGam* g_1 g_2 adds g_1's entries to g_2, possibly hiding (overwriting) entries of g_2 unless they appear in an outer level.

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

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

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

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

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

Later (in Chapter 6) the variant *valGamLookup* will do additional work, but for now it does not differ from *gamLookup* except for the return of an error in case no entry is found in the *ValGam*. The additional variant *valGamLookupTy* is specialized further to produce an error message in case the identifier is missing from the environment. Later, we will discuss errors (like constructor *Err_NamesNotIntrod*). Here, we only note that the definition and pretty-printing of errors is done by using AG.

3.3.3 Checking expressions (Expr)

The rules in Fig. 3.3 do not provide much information about how the type σ in the consequence of a rule is to be computed; it is just stated that it should relate in some way to other types. However, type information can be made available to parts of the abstract syntax tree, either because the programmer has supplied it somewhere or because the compiler can reconstruct it. For types given by a programmer the compiler has to check if such a type correctly describes the value of an expression for which the type is given. This is called *type checking*. If no type information has been given for a value, the compiler needs to reconstruct or infer this type based on the structure of the abstract syntax tree and the semantics of the language as defined by the typing rules. This is called *type inferencing*. In EH1 we exclusively deal with type checking.

We can now tailor the type rules in Fig. 3.3 towards an implementation which performs type checking, in Fig. 3.5. Fig. 3.5 differs from Fig. 3.3 in the following aspects:

- We use an expected, or known type σ^k. Known type information is forwarded from the place it becomes known to where it is needed. Here it traverses from the top to the bottom of the AST. In a type rule it traverses from the consequence to the

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

$$\frac{\vdash^{\leqslant} Int \leqslant \sigma^k : \sigma}{\Gamma; \sigma^k \vdash^e int : \sigma}\ \text{E.INT}_K \qquad \frac{\vdash^{\leqslant} Char \leqslant \sigma^k : \sigma}{\Gamma; \sigma^k \vdash^e char : \sigma}\ \text{E.CHAR}_K \qquad \frac{\iota \mapsto \sigma_g \in \Gamma \quad \vdash^{\leqslant} \sigma_g \leqslant \sigma^k : \sigma}{\Gamma; \sigma^k \vdash^e \iota : \sigma}\ \text{E.VAR}_K$$

$$\frac{\Gamma; \square \to \sigma^k \vdash^e e_1 : \sigma_a \to \sigma \quad \Gamma; \sigma_a \vdash^e e_2 : \_}{\Gamma; \sigma^k \vdash^e e_1\ e_2 : \sigma}\ \text{E.APP}_K \qquad \frac{\Gamma; \sigma_1^k \vdash^e e_1 : \sigma_1 \quad \Gamma; \sigma_2^k \vdash^e e_2 : \sigma_2}{\Gamma; (\sigma_1^k, \sigma_2^k) \vdash^e (e_1, e_2) : (\sigma_1, \sigma_2)}\ \text{E.PROD}_K$$

$$\frac{\vdash^{\leqslant} \square \to \square \leqslant \sigma^k : \sigma_p \to \sigma_r \quad [\,]; \sigma_p \vdash^p p : \Gamma_p \quad \Gamma_p, \Gamma; \sigma_r \vdash^e e : \sigma_e}{\Gamma; \sigma^k \vdash^e \lambda p \to e : \sigma_p \to \sigma_e}\ \text{E.LAM}_K \qquad \frac{\Delta \vdash^t t : \sigma_a \quad \Gamma; \sigma_a \vdash^e e : \sigma_e \quad \vdash^{\leqslant} \sigma_a \leqslant \sigma^k : \_}{\Gamma; \sigma^k \vdash^e (e :: t) : \sigma_e}\ \text{E.ANN}_K$$

$$\frac{\Gamma_t; \Gamma_t, \Gamma; \Gamma_p \vdash^d d : \Gamma_t; \Gamma_p \quad \Gamma_p; \sigma^k \vdash^e b : \sigma}{\Gamma; \sigma^k \vdash^e \textbf{let}\ d\ \textbf{in}\ b : \sigma}\ \text{E.LET}_K$$

Within environment Γ, expecting the type of expression e to be σ^k, e has type σ.

- e : Expression
- σ^k : Expected/known type of expression
- σ : Type of expression
- Δ : Environment $\overline{\iota \mapsto \sigma}$ for type identifiers, cannot be modified (hence treated as a global constant in rule E.ANN)
- Γ : Environment $\overline{\iota \mapsto \sigma}$ for value identifiers

Figure 3.5.: Expression type rules (K)

prerequisites. For this reason it is (by convention) placed at the left side of the turnstyle '$\vdash$'.

- We uncouple type signatures from their corresponding value declarations. The single joint declaration for signature and value is split into two separate ones. Additional Γ's are required to make the type signature available at the value declaration. This complicates the type rules but it facilitates extension with different kinds of declarations in later EH versions.
- Patterns may be used in **let** and λ expressions instead of single identifiers.

$$\boxed{\Gamma_t^k;\Gamma_p^k;\Gamma \vdash^d d : \Gamma_t;\Gamma_p}$$

$$\frac{\Delta \vdash^t t : \sigma_i \quad \Gamma_i \equiv [i \mapsto \sigma_i]}{\_;\Gamma_p;\_ \vdash^d (i :: t) : \Gamma_i;\Gamma_p}\ \text{D.TYSIG}_K \qquad \frac{p \mapsto \sigma_s \in \Gamma_t^k \quad \Gamma;\sigma_s \vdash^e e : \_ \quad \Gamma_p^k;\sigma_s \vdash^p p : \Gamma_p}{\Gamma_t^k;\Gamma_p^k;\Gamma \vdash^d (p = e) : [\,];\Gamma_p}\ \text{D.VAL}_K$$

Declaration d has explicit type bindings Γ_t, within explicit bindings Γ_t^k and implicit type bindings Γ_p^k, and type checks within Γ, yielding additional bindings Γ_p.

d	: Declaration
Γ_t	: Environment with new type signature bindings
Δ	: Environment $\overline{\iota \mapsto \sigma}$ for type identifiers, cannot be modified (hence treated as a global constant in rule E.ANN)
Γ_t^k	: Collected Γ_t, used by patterns to extract bindings for pattern variables
Γ	: Environment with known bindings
Γ_p^k	: Known/gathered pattern variable bindings
Γ_p	: Γ_p^k+ new bindings

Figure 3.6.: Declaration type rules (K)

- The *int* in rule E.INT represents all possible denotations for integers of type *Int*, that is $\{minint, ..., -1, 0, 1, 2, ..., maxint\}$. Similarly, *char* in rule E.CHAR represents all character denotations. We assume ASCII encoding (no unicode).
- The type annotation requires an environment for the type expression, denoted by Δ. Although this environment is provided by the conclusion of rule E.ANN it is not shown, as Δ is a global constant for this EH version.

  We could have shown Δ as part of the type scheme for expressions, but this would have caused clutter. This is a typical example of the trade-off required between completeness and focus on the essentials of the description.
- The caption of both figures (holding the type rules for expressions and declarations) incorporates between parentheses the view on the type rules. Here we discuss view 'K', for type checking with known types.

We emphasize this difference by the use of colors: in the electronic version of this thesis we use blue for changes relative to the previous set of rules, grey for the unchanged part. This (of course) is better seen through media which support color, and worse in black and white print. The printed version of this thesis therefore does not use colors and typesets

$$\boxed{\vdash^{\leqslant} \sigma_l \leqslant \sigma_r : \sigma}$$

$$\frac{\vdash^{\cong} \sigma_l \cong \sigma_r : \sigma}{\vdash^{\leqslant} \sigma_l \leqslant \sigma_r : \sigma}\ \text{FIT}_K$$

$\leqslant$ delegates to $\cong$.
σ_l : Type to fit in σ_r
σ_r : Type in which σ_l must fit
σ : Result type, (in general) equals σ_r, but differs slightly in later versions

Figure 3.7.: Fitting of types (K)

(and prints) in black.

We also start with the discussion of the corresponding AG implementation. The rules now take an additional context, the expected (or known) type σ^k (attribute *knTy*, simultaneously referred to by σ^k *//knTy*) as specified by the programmer, defined in terms of AG as follows:

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

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

- A *known* (or *expected*) type σ^k *//knTy* is passed top-down through the syntax tree of an expression, representing the maximal type (in terms of $\leqslant$, see Fig. 3.7, Fig. 3.8 and discussion below) the type of an expression can be.
- A result type σ*//ty* is computed bottom-up for each expression, representing the minimal type (in terms of $\leqslant$) the expression can have.
- At each node in the abstract syntax tree it is checked whether $\sigma \leqslant \sigma^k$ holds. The result of *lhs* $\leqslant$ *rhs* is *rhs* which is subsequently used by the type checker, for example to simply return or use in constructing another, usually composite, type.
- In general, for *lhs* $\leqslant$ *rhs* the *rhs* is an expected type whereas *lhs* is the bottom-up computed result type.

An additional judgement type named *fit* (Fig. 3.7) is needed to check an actual type against an expected (known) type. The judgement specifies the matching $\sigma_1 \leqslant \sigma_2$ of two types σ_1 and σ_2. The meaning of $\leqslant$ is that the left hand side (lhs) type σ_1 of $\leqslant$ can be used where the right hand side (rhs) type σ_2 is expected. Expressed differently, $\leqslant$ checks whether a value of type σ_1 can flow (that is, be stored) into a memory location of type σ_2.

The relation $\leqslant$ is asymmetric because "a value flowing into a location" does not imply that

$$\boxed{\vdash^{\cong} \sigma_l \cong \sigma_r : \sigma}$$

$$\frac{I_1 \equiv I_2}{\vdash^{\cong} I_1 \cong I_2 : I_2}\ \text{M.CON}_K \qquad \frac{\vdash^{\cong} \sigma_2^a \cong \sigma_1^a : \sigma_a \quad \vdash^{\cong} \sigma_1^r \cong \sigma_2^r : \sigma_r}{\vdash^{\cong} \sigma_1^a \to \sigma_1^r \cong \sigma_2^a \to \sigma_2^r : \sigma_a \to \sigma_r}\ \text{M.ARROW}_K$$

$$\frac{\vdash^{\cong} \sigma_1^l \cong \sigma_2^l : \sigma_l \quad \vdash^{\cong} \sigma_1^r \cong \sigma_2^r : \sigma_r}{\vdash^{\cong} (\sigma_1^l, \sigma_1^r) \cong (\sigma_2^l, \sigma_2^r) : (\sigma_l, \sigma_r)}\ \text{M.PROD}_K \qquad \frac{}{\vdash^{\cong} \square \cong \sigma : \sigma}\ \text{M.ANY.L}_K$$

$$\frac{}{\vdash^{\cong} \sigma \cong \square : \sigma}\ \text{M.ANY.R}_K$$

σ_l matches σ_r, $\sigma \equiv \sigma_r$ with $\square$ eliminated from σ
σ_l : Type to match
σ_r : Type to match
σ : Result type

Figure 3.8.: Type matching (K)

it can flow the other way, so $\leqslant$ conceptually has a direction, even though the current version of $\leqslant$ is symmetric. To emphasize this, the rule for $\leqslant$ (in Fig. 3.7) delegates to the rules in Fig. 3.8. The rules in Fig. 3.8 test the equality of two types by matching their structure. Matching is denoted by $\cong$.

The rules for $\leqslant$ also specify a result type. Strictly this result is not required for the *fit* judgement to hold, but in the implementation it is convenient to have of $\leqslant$ (and its implementation *fitsIn*) return the smallest type σ for which of $\sigma_1 \leqslant \sigma$ and $\sigma_2 \leqslant \sigma$ hold. This is useful in relation to the use of $\square$ in rule M.ANY.L and rule M.ANY.R; we will come back to this later.

For example, $\leqslant$ is used in rule E.INT which checks that its actual *Int* type matches the known type σ^k. The implementation of rule E.INT performs this check and returns the type σ in attribute *ty*:

```
attr AllExpr [|| ty : Ty]
sem Expr
  | IConst loc.fo_ = tyInt <= @lhs.knTy
                .ty  = foTy @fo_
```

The implementation for rule E.CHAR is defined similarly. The constant *tyInt* represents the *Int* type constant.

The function *fitsIn* (printing as $\leqslant$ in infix notation) returns a *FIOut* (**f**its**I**n **out**put) data structure in attribute *fo_*. *FIOut* consists of a record containing amongst other things field *foTy*:

```
data FIOut = FIOut{foTy :: Ty      ,foErrL :: ErrL}
emptyFO    = FIOut{foTy = Ty_Any,foErrL = [ ]    }
foHasErrs :: FIOut → Bool
foHasErrs = not.null.foErrL
```

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

- Invariant to our implementation is the top-down passing of an expected type. However, this type is not always fully known in a top-down order. For example, in rule E.APP (Fig. 3.5) the argument of the expected function type $\square \rightarrow \sigma^k$ is not known because this information is only available from the environment Γ which is used further down in the AST via rule E.VAR. In this use of $\square$ it represents a "don't know" of the type system implementation. As such $\square$ has the role of a type variable (as introduced for type inferencing in Section 4).
- An error occurs at a place where the implementation of the type system needs a type to continue (type checking) with. In that case $\square$ is used to prevent further errors from occurring. In this use of $\square$ it represents a "don't care" of the type system implementation. As such $\square$ will be replaced by a more specific type as soon as it matches (via $\leqslant$) such a type.

In both cases $\square$ is a type exclusively used by the implementation to smoothen type checking. The rules for $\leqslant$ for $\square$ in Fig. 3.8 state that $\square$ is equal to any type. The effect is that the result of $\leqslant$ is a more specific type. This suits our "don't know" and "don't care" use. Later, when discussing the AG implementation for these rules, this issue reappears. In later EH versions we will split the use of $\square$ into the proper use of a type lattice, and it will thus disappear.

The role of $\square$ may appear to be similar to $\top$ and $\bot$ known from type theory. However, $\square$ is used only as a mechanism for the type system implementation. It is not offered as a feature to the user (i.e. the EH programmer) of the type system. A type expression t (Fig. 3.1) does not allow $\square$, but a type (signature) σ does.

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

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

The rule M.ARROW in Fig. 3.8 for comparing function types compares the types for arguments in the opposite direction. Only in Chapter 6 when $\leqslant$ really behaves asymmetrically we will discuss this aspect of the rules which is named *contravariance*. In the rules in

Fig. 3.8 the direction makes no difference; the correct use of the direction for now only anticipates issues yet to come.

The Haskell counterpart of $\vdash^{\leqslant} \sigma_l \leqslant \sigma_r : \sigma$ is implemented by *fitsIn*:

```
fitsIn :: Ty → Ty → FIOut
fitsIn ty1 ty2
  =  f ty1 ty2
  where
    res t                 = emptyFO{foTy = t}
    f Ty_Any t2           = res t2                          -- m.any.l
    f t1     Ty_Any       = res t1                          -- m.any.r
    f t1 @(Ty_Con s1)                                      -- m.con
      t2 @(Ty_Con s2)
        | s1 == s2        = res t2
    f t1 @(Ty_App (Ty_App (Ty_Con c1) ta1) tr1)            -- m.arrow
      t2 @(Ty_App (Ty_App (Ty_Con c2) ta2) tr2)
        | hsnIsArrow c1 ∧ c1 == c2
        = comp ta2 tr1 ta1 tr2 (λa r → [a] `mkArrow` r)
    f t1 @(Ty_App tf1 ta1)                                 -- m.prod
      t2 @(Ty_App tf2 ta2)
        = comp tf1 ta1 tf2 ta2 Ty_App
    f t1     t2           = err [Err_UnifyClash ty1 ty2 t1 t2]
    err e                 = emptyFO{foErrL = e}
    comp tf1 ta1 tf2 ta2 mkComp
        = foldr1 (λfo1 fo2 → if foHasErrs fo1 then fo1 else fo2)
                 [ffo, afo, res rt]
      where ffo = f tf1 tf2
            afo = f ta1 ta2
            rt  = mkComp (foTy ffo) (foTy afo)
```

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

In case an error is detected in both the type components in *comp*, only the leftmost are returned: *comp* is left-biased with respect to error reporting. This choice prevents too many errors, assuming leftmost errors are more informative; this is somewhat arbitrary.

The type rules leave open how to handle a situation when a required constraint is invalid. For a compiler this is not good enough, which is the reason why *fitsIn* gives a "will-do" type *Ty_Any* back together with an error for later processing. Errors themselves are also

described via AG:

```
data Err
  | UnifyClash      ty1        : {Ty}
                    ty2        : {Ty}
                    ty1detail  : {Ty}
                    ty2detail  : {Ty}
data Err
  | NamesNotIntrod  nmL        : {[HsName]}
type ErrL = [Err]
```

The *Err* datatype is available as a datatype in the same way *Ty* is.

Variable occurrences *Var* The error datatype is also used for signalling undeclared identifiers when a type for an identifier is retrieved from the environment $\Gamma /\!/ valGam$:

```
sem Expr
  | Var (loc.ty_g_, loc.nmErrs)
              = valGamLookupTy @nm @lhs.valGam
        loc.fo_ = @ty_g_ ⩽ @lhs.knTy
           .ty  = foTy @fo_
```

Again, the error condition is signalled by a non empty list of errors if a lookup in $\Gamma /\!/ valGam$ fails. These errors are gathered so they can be incorporated into an annotated pretty printed version of the program (this has been omitted).

Typing rule E.VAR uses the environment $\Gamma /\!/ valGam$ to retrieve the type of an identifier. This environment for types of identifiers is declared as an inherited attribute, initialized at the top of the abstract syntax tree. It is only extended with new bindings for identifiers at a declaration of an identifier.

```
attr AllDecl AllExpr [valGam : ValGam ||]
sem AGItf
  | AGItf expr.valGam = emptyGam
```

Function application *App* Type checking for rule E.APP constructs $\square \to \sigma^k$ as the expected type for the function to be applied. The resulting type *func.ty* is decomposed into argument and result type, of which the argument type is used as the known type for the argument child of the *App* node:

```
sem Expr
  | App func.knTy = [Ty_Any] `mkArrow` @lhs.knTy
      (loc.ty_a_, loc.ty_)
              = tyArrowArgRes @func.ty
      arg .knTy = @ty_a_
      loc .ty   = @ty_
```

This further clarifies the need for □. To see why, assume we do not use □. Then, in the following example, what would be the *knTy* against which 3 will be checked?

let $id :: Int \to Int$
  $id = \lambda x \to x$
in $id\ 3$

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

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

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

The type construction and inspection done in the *App* variant of *Expr* requires some additional type construction functions, for example *mkArrow* used in *App*. The function is part of the class *SemApp* defining (semantic) functions related to building application *App* like structures: w^{W}_{W}

class *SemApp a* **where**
  semApp :: $a \to a \to a$
  semAppTop :: $a \to a$
  semCon :: *HsName* $\to a$
  semParens :: $a \to a$
  mkApp :: $[a] \to a$
  mkConApp :: *HsName* $\to [a] \to a$
  mkProdApp :: $[a] \to a$
  mk1Arrow :: $a \to a \to a$

```
mkArrow  :: [a] → a → a
```

The instance for *SemApp Ty* is defined by:

```
instance SemApp Ty where
  semApp     = Ty_App
  semAppTop  = id
  semCon     = Ty_Con
  semParens  = id
```

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

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

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

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

```
tyArrowArgRes :: Ty → (Ty, Ty)
tyArrowArgRes t
  = case t of
      Ty_App (Ty_App (Ty_Con nm) a) r
        | hsnIsArrow nm → (a, r)
      _                 → (Ty_Any, t)
```

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

Constructor *Con*, tuples Apart from constructing function types only tupling allows us to build composite types. The rule E.PROD for tupling has no immediate counterpart in the implementation because a tuple (a, b) is encoded as the application $(,)\ a\ b$. We need a rule E.CON (replacing rule E.CON) to produce a type for $(,)$:

$$\frac{\overline{\sigma_p} \equiv [\sigma_1, ..., \sigma_n], \quad (\sigma_1, ..., \sigma_n) \equiv \sigma_r}{\Gamma; ... \to \sigma_r \vdash^e (,) : \overline{\sigma_p} \to \sigma_r}\ \text{E.CON}_K$$

The expected type σ_r of the complete tuple can be constructed from $knTy/\!/\sigma^k$, which by definition has the form $\square \to \square \to (a, b)$ (for this example). The result type of this func-

tion type is taken apart and used to produce the desired type $a \to b \to (a, b)$. The *Con* alternative implements this:

```
sem Expr
  | Con loc.ty     = tyProdArgs @ty_r_ `mkArrow` @ty_r_
           .ty_r_ = tyArrowRes @lhs.knTy
```

Note that, despite the cartesian product constructors being essentially polymorphic, we do not have to do any kind of unification here, since they either appear in the right hand side of a declaration where the type is given by an explicit type declaration, or they occur at an argument position where the type has been implicitly specified by the function type. Therefore we indeed can use the a and b from type $\square \to \square \to (a, b)$ to construct the type $a \to b \to (a, b)$ for the constructor $(,)$.

λ-expression *Lam* For rule E.LAM the check whether *knTy* has the form $\sigma_1 \to \sigma_2$ is done by letting *fitsIn* match the *knTy* with $\square \to \square$. The result (forced to be a function type) is split up by *tyArrowArgRes* into argument and result type.

```
sem Expr
  | Lam loc  .fo_fitF_ = ([Ty_Any] `mkArrow` Ty_Any) ⩽ @lhs.knTy
       (loc.ty_p_, loc.ty_r_)
                       = tyArrowArgRes (foTy @fo_fitF_)
       arg  .valGam = emptyGam
            .knTy   = @ty_p_
       body.knTy    = @ty_r_
            .valGam = gamAddGam @arg.valGam @lhs.valGam
       loc  .ty     = [@ty_p_] `mkArrow` @body.ty
```

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

```
sem Expr
  | TypeAs expr.knTy = @tyExpr.ty
      loc  .fo_      = @tyExpr.ty ⩽ @lhs.knTy
```

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

3.3.4 Checking pattern expressions

Before we can look into more detail at the way new identifiers are introduced in **let**- and λ-expressions we take a look at patterns. The rules in Fig. 3.9 demonstrate the basic idea:

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

$$\frac{\vdash^{\leqslant} \sigma^k \leqslant \mathit{Int} : \_}{\Gamma;\sigma^k \vdash^p \mathit{int} : \Gamma}\ \text{P.INT}_K \qquad \frac{\vdash^{\leqslant} \sigma^k \leqslant \mathit{Char} : \_}{\Gamma;\sigma^k \vdash^p \mathit{char} : \Gamma}\ \text{P.CHAR}_K \qquad \frac{\Gamma_i \equiv [\,i \mapsto \sigma^k\,]}{\Gamma;\sigma^k \vdash^p i : \Gamma_i, \Gamma}\ \text{P.VAR}_K$$

$$\frac{\Gamma_i \equiv [\,i \mapsto \sigma^k\,] \qquad \Gamma;\sigma^k \vdash^p p : \Gamma_p}{\Gamma;\sigma^k \vdash^p i\ @p : \Gamma_i, \Gamma_p}\ \text{P.VARAS}_K \qquad \frac{\begin{array}{c}\Gamma;\sigma^k \vdash^p p : \Gamma_p \\ \overline{\sigma} \equiv [\,\sigma_1, ..., \sigma_n\,], \quad (\sigma_1, ..., \sigma_n) \equiv \sigma^k \\ |\,\overline{\sigma}\,| \equiv n\end{array}}{\Gamma;\sigma^k \vdash^p p : \Gamma_p}\ \text{P.APPTOP}_K$$

$$\frac{\begin{array}{c}\sigma_f, \sigma_a \equiv (\sigma_1, ..., \sigma_{n-1}), \sigma_n, \quad (\sigma_1, ..., \sigma_{n-1}, \sigma_n) \equiv \sigma^k \\ \Gamma;\sigma_f \vdash^p p_1 : \Gamma_f \\ \Gamma_f;\sigma_a \vdash^p p_2 : \Gamma_a\end{array}}{\Gamma;\sigma^k \vdash^p p_1\ p_2 : \Gamma_a}\ \text{P.APP}_K$$

Knowing the type of pattern p to be σ^k, yielding additional bindings Γ (for identifiers introduced by p)

σ^k : Known type of pattern

Γ^k : Already gathered bindings (for this EH version initially [ ])

Γ : Γ^k+ new bindings

Figure 3.9.: Pattern expression type rules (K)

gather bindings for identifiers given an expected type of the pattern. For example, the following fragment specifies the type of p whereas we also need to know the types of a and b:

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

The expected type of a pattern is distributed over the pattern by dissecting it into its constituents. Patterns do not return a type, but instead return type bindings for the identifiers inside a pattern. The new bindings are subsequently used in **let**- and λ-expression bodies.

The typing for a tuple pattern is expressed as a combination of rule P.APP and rule P.APPTOP (Fig. 3.9). A tuple pattern 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. This facilitates the "one at a time" treatment of tuple elements, but complicates the treatment of overall aspects. We use the following strategy:

- We dissect the known type of a tuple into its elements at the top (*AppTop*) of a pattern, that is, the fully saturated pattern (see rule P.APPTOP). Here we also check whether the arity of the pattern and the arity of its expected type match.
- The known type dissection is distributed over the AST elements of the pattern (see rule P.APP), for use by pattern elements.
- At the leaves of a pattern we either bind its known type to an identifier (e.g. rule P.VAR) or we check the known type fits into the type of a constant (e.g. rule P.INT).

The AG implementation dissects the known type of a tuple into its element types at *AppTop* using function *tyProdArgs*. For this version of EH we only have tuple patterns. Instead of manipulating the expected tuple type over the *App* spine of the pattern, we directly decompose the tuple type into a list *knTyL* of constituent types. We also require the arity of the pattern in order to check (at *AppTop*) if the pattern is fully saturated: w^{w}_{w}

```
attr AllPatExpr [knTy : Ty ||]
attr PatExpr [knTyL : TyL ||]
sem PatExpr
  | AppTop loc      .tys_      = tyProdArgs @lhs.knTy
                    .arityErrs = if length @tys_ ==  @patExpr.arity
                                 then [ ]
                                 else [Err_PatArity @lhs.knTy @patExpr.arity]
            patExpr.knTyL      = reverse @tys_
sem PatExpr
  | App (loc.ty_a_, loc.tyi_l)
                    = tyLHdAndTl @lhs.knTyL
        func.knTyL = @tyi_l
            .knTy  = Ty_Any
       arg .knTy   = @ty_a_
```

The list of tuple elements is passed through attribute *knTyL* to all *App*'s of the pattern. At each *App* one element of this list is taken as the known type $knTy/\!/\sigma^k$ of the element AST. In case the arities of pattern and its expected type do not match, an error is produced and the tuple components are given $\square$ as their expected type (by *tyLHdAndTl*).

Finally, for the distribution of the known type throughout a pattern we need to properly initialize *knTyL*. Because a pattern occurs in other contexts, that is as a child of other AST nodes, other than an *AppTop*, we need to specify a default value.

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

The arity of the patterns is needed as well:

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

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

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

```
attr AllPatExpr [| valGam : ValGam |]
sem PatExpr
  | Var loc.valGam_i_ = if @lhs.inclVarBind ∧ @nm ≢ hsnWild
                        then gamUnit @nm (ValGamInfo @lhs.knTy)
                        else emptyGam
      lhs.valGam      = gamAddGam @valGam_i_ @lhs.valGam
```

A new entry is added if the variable name is not equal to an underscore '_' and has not been added previously via a type signature for the variable name, signalled by attribute *inclVarBind*. Because our check on duplicate introductions is based on duplicate entries in *valGam*, we inhibit addition if an entry has already been added via a type signature. This condition is indicated by *inclVarBind*.

3.3.5 Checking declarations

In a **let**-expression type signatures, patterns and expressions meet. The algorithmic version of rule E.LET in Fig. 3.5 is more complex than the equational version in Fig. 3.3 because of the presence of mutual recursive definitions and the uncoupling of type signatures and their corresponding value definition:

- 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 f and g together would be in the same binding group.

- 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 obligatory presence of type signatures in this version of EH it is possible to first gather all signatures and only then type check the value definitions. Therefore, for this version of EH mutual recursive definitions are less of 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.

Our AG implementation follows the strategy of rule E.LET (Fig. 3.5), rule D.TYSIG and rule D.VAL (Fig. 3.6):

- First extract all type signatures in:

  attr *AllDecl* [|| *gathTySigGam* **use**{‘*gamAddGam*‘}{*emptyGam*} : *ValGam*]

- Then distribute these gathered signatures in:

  attr *AllDecl* [*tySigGam* : *ValGam* ||]

  This attribute and *gathTySigGam* correspond to Γ_t in the type rules.

- Use the signature as the known type of a pattern in order to extract bindings inside a pattern in:

  attr *AllDecl* [| *patValGam* : *ValGam* |]

  This attribute corresponds to Γ_p in the type rules.

- Finally distribute all previously gathered information (for use by identifier occurrences in expressions) in:

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

  sem *AGItf*
  | *AGItf expr*.*valGam* = *emptyGam*

  This attribute corresponds to Γ in the type rules.

This flow of information is described by the following AG fragment:

```
sem Expr
  | Let decls.patValGam = gamPushGam @decls.gathTySigGam @lhs.valGam
          .tySigGam     = @decls.gathTySigGam
          .valGam       = @decls.patValGam
    body.valGam         = @decls.patValGam
```

Attribute *gathTySigGam* is populated with bindings in a type signature *TySig*:

```
sem Decl
  | TySig lhs.patValGam     = @lhs.patValGam
```

```
.gathTySigGam = gamUnit @nm (ValGamInfo @tyExpr.ty)
```

Bindings from patterns are gathered in a value declaration *Val*. The type signature for the topmost variable of the pattern is used as the expected type (*knTy*) of the pattern. *tySigGam* has to be queried for the type signature:

```
sem Decl
  | Val (loc.ty_sig_, loc.nmErrs) = let e = [Err_MissingSig @patExpr.pp]
                                        l n = gamLookup n @lhs.tySigGam
                                    in case@patExpr.mbTopNm of
                                         Nothing → (Ty_Any, e)
                                         Just nm → case l nm of
                                                     Nothing → (Ty_Any, e)
                                                     Just vgi → (vgiTy vgi, [ ])
      expr     .knTy          = @ty_sig_
               .valGam        = @lhs.valGam
      patExpr.valGam          = @lhs.patValGam
               .knTy          = @ty_sig_
      lhs      .patValGam     = @patExpr.valGam
               .gathTySigGam  = emptyGam
```

The actual bindings are added inside a patterns at variable occurrences.

We allow patterns of the form '*ab* @(*a*, *b*)' to have a type signature associated with *ab*. No type signatures are allowed for '(*a*, *b*)' without the '*ab* @' alias (because there is no way to refer to the anonymous tuple) nor is it allowed to specify type signature for the fields of the tuple (because of simplicity, additional plumbing would be required).

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

3.3.6 Checking type expressions

All that is left to do now is to use the type expressions to extract type signatures (type rules for type expressions are in Fig. 3.10). This is straightforward as type expressions (abstract syntax for what the programmer specified) and types (as internally used by the compiler) have almost the same structure (Section 3.2, page 31:

```
attr TyExpr [|| ty : Ty]
sem TyExpr
  | Con (loc.tgi_, loc.nmErrs)
          = case tyGamLookup @nm @lhs.tyGam of
              Nothing → (TyGamInfo Ty_Any, [Err_NamesNotIntrod [@nm]])
              Just @tgi_ → (@tgi_, [ ])
      loc.ty = tgiTy @tgi_
sem TyExpr
```

$$\boxed{\Delta \vdash^t t : \sigma}$$

$$\frac{I \mapsto \sigma \in \Delta}{\Delta \vdash^t I : \sigma}\ \text{T.CON}_K \qquad \frac{\Delta \vdash^t t_2 : \sigma_a \quad \Delta \vdash^t t_1 : \sigma_f}{\Delta \vdash^t t_1\ t_2 : \sigma_f \sigma_a}\ \text{T.APP}_K$$

Within environment Δ, type expression t has a (replica) type signature σ.
t : Type expression
σ : Type signature
Δ : Environment $\overline{\iota \mapsto \sigma}$ for type identifiers

Figure 3.10.: Type expression type rules (K)

```
| App loc.ty = Ty_App @func.ty @arg.ty
```

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

```
data TyGamInfo = TyGamInfo{tgiTy :: Ty} deriving Show
type TyGam = Gam HsName TyGamInfo
tyGamLookup :: HsName → TyGam → Maybe TyGamInfo
tyGamLookup nm g
  = case gamLookup nm g of
      Nothing | hsnIsProd nm → Just (TyGamInfo (Ty_Con nm))
      Just tgi               → Just tgi
      _                      → Nothing
```

This environment, denoted by Δ, is passed to a *TyExpr*:

```
attr AllTyExpr [tyGam : TyGam ||]
```

At the root of the AST *tyGam* is initialized with the fixed set of types available in this version of the compiler. Because Δ is fixed, and can be seen as a global constant, we have omitted Δ from all type rules, except those describing type expressions.

```
sem AGItf
  | AGItf loc.tyGam = assocLToGam
                        [(hsnArrow, TyGamInfo (Ty_Con hsnArrow))
                        ,(hsnInt,   TyGamInfo tyInt)
                        ,(hsnChar,  TyGamInfo tyChar)
                        ]
```

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

```
sem TyExpr
  | Con (loc.tgi_, loc.nmErrs)
          = case tyGamLookup @nm @lhs.tyGam of
              Nothing → (TyGamInfo Ty_Any, [Err_NamesNotIntrod [@nm]])
              Just @tgi_ → (@tgi_, [])
      loc.ty = tgiTy @tgi_
```

3.4 Conclusion and remarks

In this chapter we have described the first version of EH, that is, λ-calculus (plus tuples) packaged in Haskell notation. Types are simple (no polymorphism), explicit, and checked. The next version adds non-polymorphic type inference.

This EH version provides the basis for later EH versions, but is also influenced by the need of later versions. For example, a **let** expression is expressed in terms of declarations and a single expression in which the bindings introduced by declarations are used. This has the following consequences:

- The separation into **let** expressions and declarations allows flexible extension with new kinds of declarations without the need to introduce new kinds of **let** expressions.
- Common aspects of declarations can be factored out and be dealt with in the **let** expression.
- On the other hand, some kinds of declarations are related, and information constructed in one kind of declaration (for example, a type signature declaration) must be made available in the related declaration (a value declaration). Additional environments are required to pass the relevant information around; this is the price we pay for this flexibility.

We emphasize that the AG system allows us to independently specify these aspects. We consider this a strong point of the AG system (see also Chapter 12). By using a slightly more general approach we are able to anticipate later modifications. However, this raises the question of what "the right" approach is; we will discuss our choice (and related choices concerning our partitioning into steps) in the conclusion (Chapter 12).

4 EH 2: MONOMORPHIC TYPE INFERENCING

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

let $i = 5$
in i

is accepted by this version of EH. From now on, and when relevant, we will also give the output as produced by the corresponding EH compiler, because additional type information is inferred:

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

The idea is that the type system implementation has an internal representation for "knowing a type is some 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*.

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

So

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

will give

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

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

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

```
let id = \x -> x
    {- [ id:v_3_0 -> v_3_0 ] -}
in id
```

On the other hand, if contradictory information in case of a monomorphic *id* applied to values of different type, we will obtain the following error:

```
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
```

The next version of EH dealing with Haskell style polymorphism (Chapter 5) accepts this program. This version of EH also allows partial type signatures; we will discuss this feature in Chapter 10.

4.1 Type variables

In order to be able to represent yet unknown types the type language needs *type variable*s to represent this, and therefore we extend our set of types as follows:

$\sigma ::= Int \mid Char$
$\quad \mid \ (\sigma, ..., \sigma) \mid \sigma \rightarrow \sigma$
$\quad \mid \ v$

The corresponding type structure *Ty* needs to be extended with an alternative for a variable. Note that the AG system allows us to define this additional type variant independent of the previous definition, thus allowing an isolated explanation:

data *Ty*
  | *Var tv* : {*TyVarId*}

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

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

type *TyVarId* = *UID*

We thread a counter as global variable through the AST, incrementing it whenever a new unique value is required. The implementation used throughout all EH compiler versions is more complex because a *UID* actually is a hierarchy of counters, each level counting within the context of an outer level. This structure allows the outer level to be threaded, while avoiding this for an inner level when passed to a function, thus avoiding the introduction of possible cycles. This is not discussed any further; we will ignore this aspect and just assume a unique *UID* can be obtained. Its use is visible whenever we need a so called fresh type variable in a type rule.

4.2 Constraints

Although the typing rules in Fig. 3.5, page 40 still hold we need to look at the meaning of $\leqslant$ (or *fitsIn*) in the presence of type variables. The idea here is that what is unknown may be replaced by that which is known. For example, when the check $v \leqslant \sigma$ is encountered, the easiest way to make $v \leqslant \sigma$ true is to state that the (previously) unknown type v equals σ. An alternative way to look at this is that $v \leqslant \sigma$ is true under the constraint that v equals σ. Alternatively, we say that v binds to σ.

4.2.1 Remembering and applying constraints

As soon as we have determined that a type variable v equals a type σ, we must remember this and propagate it to wherever v is used. We use constraints, denoted by C, to remember such a binding of a type variable to a type:

$C = [v \mapsto \sigma]$

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

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

emptyCnstr :: *C*
emptyCnstr = *C* [ ]

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

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

Different strategies can be used to cope with constraints [94, 35, 80]. Here constraints C are used to replace all other references to v by σ, for this reason often named a *substitution*. In this version of EH the replacement of type variables with types is done immediately after constraints are obtained, 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 inside and can replace, or substitute those type variables:

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

The operator ⊕ applies constraints C to a *Substitutable*. Function *ftv* extracts the free type variable references as a set (implemented as a list) of *TVarId*'s.

A C can be applied to a type:

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

This is another place where we use the AG notation and the automatic propagation of values as attributes throughout the type representation to make the description of the application of a C to a *Ty* easier. The functions *tyAppCnstr* and *tyFtv* are defined in terms of the following AG:

```
attr TyAGItf AllTy [cnstr : C ||          ]
attr AllTyAndFlds  [          || repl : self ]
attr TyAGItf       [          || repl : Ty   ]
sem Ty
  | Var            lhs.repl = maybe @repl id (cnstrTyLookup @tv @lhs.cnstr)
attr TyAGItf AllTy [|| tvs use{∪}{[]} : TyVarIdL]
sem Ty
  | Var            lhs.tvs  = [@tv]
```

The application of a C is straightforwardly lifted to lists:

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

A C can also be applied to another C:

```
instance Substitutable C where
  s1 @(C sl1) ⊕ s2 @(C sl2)
     = C (sl1 ++ map (λ(v,t) → (v, s1 ⊕ t)) sl2′)
     where sl2′ = deleteFirstsBy (λ(v1,_) (v2,_) → v1 == v2) sl2 sl1
  ftv (C sl)
     = ftv.map snd $ sl
```

Substituting a substitution is non-commutative as constraints s_1 in $s_1 \oplus s_2$ 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.

4.2.2 Computing constraints

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

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

Computation and proper combination of constraints necessitates *fitsIn* to be rewritten in order to deal with type variables and constraints. The rules describing the desired behavior are shown in Fig. 4.1 and Fig. 4.2. We show the changed part of the full implementation for this version. The function *comp* deals with the proper combination of constraints for composite types:

```
fitsIn :: Ty → Ty → FIOut
fitsIn ty1 ty2
     =  f ty1 ty2
  where
    res t                              = emptyFO{foTy = t}
    bind tv t                          = (res t){foCnstr = tv ↦ t}
    occurBind v t | v ∈ ftv t          = err [Err_UnifyOccurs ty1 ty2 v t]
                  | otherwise          = bind v t
    comp tf1 ta1 tf2 ta2 mkComp
         = foldr1 (λfo1 fo2 → if foHasErrs fo1 then fo1 else fo2)
                  [ffo, afo, rfo]
       where ffo  = f tf1 tf2
             fs   = foCnstr ffo
             afo  = f (fs ⊕ ta1) (fs ⊕ ta2)
             as   = foCnstr afo
             rt   = mkComp (as ⊕ foTy ffo) (foTy afo)
```

$$\boxed{\vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{}{\vdash^{\cong} \square \cong \sigma : \sigma \rightsquigarrow [\,]}\ \text{M.ANY.L}_C \qquad \frac{}{\vdash^{\cong} \sigma \cong \square : \sigma \rightsquigarrow [\,]}\ \text{M.ANY.R}_C$$

$$\frac{I_1 \equiv I_2}{\vdash^{\cong} I_1 \cong I_2 : I_2 \rightsquigarrow [\,]}\ \text{M.CON}_C \qquad \frac{v_1 \equiv v_2}{\vdash^{\cong} v_1 \cong v_2 : v_2 \rightsquigarrow [\,]}\ \text{M.VAR}_C$$

$$\frac{C \equiv [v \mapsto \sigma]}{\vdash^{\cong} v \cong \sigma : \sigma \rightsquigarrow C}\ \text{M.VAR.L1}_C \qquad \frac{C \equiv [v \mapsto \sigma]}{\vdash^{\cong} \sigma \cong v : \sigma \rightsquigarrow C}\ \text{M.VAR.R1}_C$$

$$\frac{\begin{array}{c}\vdash^{\cong} \sigma_2^a \cong \sigma_1^a : \sigma_a \rightsquigarrow C_a \\ \vdash^{\cong} C_a\sigma_1^r \cong C_a\sigma_2^r : \sigma_r \rightsquigarrow C_r\end{array}}{\vdash^{\cong} \sigma_1^a \to \sigma_1^r \cong \sigma_2^a \to \sigma_2^r : C_r\sigma_a \to \sigma_r \rightsquigarrow C_rC_a}\ \text{M.ARROW}_C$$

$$\frac{\begin{array}{c}\vdash^{\cong} \sigma_1^l \cong \sigma_2^l : \sigma_l \rightsquigarrow C_l \\ \vdash^{\cong} C_l\sigma_1^r \cong C_l\sigma_2^r : \sigma_r \rightsquigarrow C_r\end{array}}{\vdash^{\cong} (\sigma_1^l, \sigma_1^r) \cong (\sigma_2^l, \sigma_2^r) : (C_r\sigma_l, \sigma_r) \rightsquigarrow C_rC_l}\ \text{M.PROD}_C$$

σ_l matches σ_r under constraints C, $\sigma \equiv C\ \sigma_r$
C : Additional constraints under which matching succeeds
σ_l : Type to match
σ_r : Type to match
σ : Result type

Figure 4.1.: Type matching rules (C)

```
rfo = emptyFO{foTy = rt, foCnstr = as ⊕ fs}
f t1 @(Ty_Var v1) (Ty_Var v2)
     | v1 == v2                  = res t1
f t1 @(Ty_Var v1) t2             = occurBind v1 t2
f t1              t2 @(Ty_Var v2) = occurBind v2 t1
```

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

- The datatype *FIOut* returned by *fitsIn* has an additional field *foCnstr* holding found constraints. This requires constraints to be combined for composite types like the *App* variant of *Ty*. The constraints returned by *fitsIn* further participate in type infer-

$$\boxed{\vdash^{\leqslant} \sigma_l \leqslant \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{\vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}{\vdash^{\leqslant} \sigma_l \leqslant \sigma_r : \sigma \rightsquigarrow C}\ \text{FIT}_C$$

$\leqslant$ delegates to $\cong$.
σ_l : Type to fit in σ_r
σ_r : Type in which σ_l must fit
σ : $\sigma \equiv C\ \sigma_r$

Figure 4.2.: Fitting of types (C)

encing.

- The function *bind* creates a binding for a type variable to a type. The use of *bind* is shielded by *occurBind* which checks if the type variable for which a binding is created does not occur free in the bound type too. This is to prevent (e.g.) $a \leqslant a \to a$ to succeed. This is because it is not clear if $a \mapsto a \to a$ should be the resulting constraint or $a \mapsto (a \to a) \to (a \to a)$ or one of infinitely many other possible solutions. A so called *infinite type* like this is inhibited by the so called *occurs check*.
- An application *App* recursively fits its components with components of another *App*. The constraints from the first fit *ffo* are applied immediately to the following component before fitting that one. This is to prevent $a \to a \leqslant Int \to Char$ from finding two conflicting constraints $[a \mapsto Int, a \mapsto Char]$ instead of properly reporting an error.

4.3 Type inference for expressions (Expr)

Constraints are used to make knowledge found about previously unknown types explicit. The typing rules in Fig. 3.5 and Fig. 3.6 in principle do not need to be changed. The only reason to adapt some of the rules to the variant in Fig. 4.3 is to clarify the way constraints are used.

The type rules in Fig. 4.3 enforce an order in which checking and inferring types has to be done. Constraints are threaded through the type rules. The flow of these constraints defines the computation order. In AG the threading of constraints is expressed by the following declaration of *tyCnstr*:

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

$$\frac{\vdash^{\leqslant} Int \leqslant C^k\sigma^k : \sigma \rightsquigarrow C}{\Gamma; C^k; \sigma^k \vdash^e int : \sigma \rightsquigarrow C\ C^k}\ \text{E.INT}_C \qquad \frac{\vdash^{\leqslant} Char \leqslant C^k\sigma^k : \sigma \rightsquigarrow C}{\Gamma; C^k; \sigma^k \vdash^e char : \sigma \rightsquigarrow C\ C^k}\ \text{E.CHAR}_C$$

$$\frac{\iota \mapsto \sigma_g \in \Gamma \qquad \vdash^{\leqslant} C^k\sigma_g \leqslant C^k\sigma^k : \sigma \rightsquigarrow C}{\Gamma; C^k; \sigma^k \vdash^e \iota : \sigma \rightsquigarrow C\ C^k}\ \text{E.VAR}_C \qquad \frac{\begin{array}{c}\overline{v}\text{ fresh},\ |\ I\ |\equiv|\ \overline{v}\ | \\ \sigma_p \equiv (\sigma_1, ..., \sigma_n), \quad [\sigma_1, ..., \sigma_n] \equiv \overline{v} \\ \vdash^{\leqslant} (\overline{v} \rightarrow \sigma_p) \leqslant C^k\sigma^k : \sigma \rightsquigarrow C\end{array}}{\Gamma; C^k; \sigma^k \vdash^e I : \sigma \rightsquigarrow C\ C^k}\ \text{E.CON}_C$$

$$\frac{\begin{array}{c}v\text{ fresh} \\ \Gamma; C^k; v \rightarrow \sigma^k \vdash^e e_1 : \sigma_a \rightarrow \sigma \rightsquigarrow C_f \\ \Gamma; C_f; \sigma_a \vdash^e e_2 : \_ \rightsquigarrow C_a\end{array}}{\Gamma; C^k; \sigma^k \vdash^e e_1\ e_2 : C_a\sigma \rightsquigarrow C_a}\ \text{E.APP}_C$$

$$\frac{\begin{array}{c}v_1, v_2\text{ fresh} \\ \vdash^{\leqslant} v_1 \rightarrow v_2 \leqslant C^k\sigma^k : \_ \rightsquigarrow C_F \\ C_FC^k; [\,]; v_1 \vdash^p p : \sigma_p; \Gamma_p \rightsquigarrow C_p; \_ \\ \Gamma_p, \Gamma; C_p; v_2 \vdash^e e : \sigma_e \rightsquigarrow C_e\end{array}}{\Gamma; C^k; \sigma^k \vdash^e \lambda p \rightarrow e : C_e\sigma_p \rightarrow \sigma_e \rightsquigarrow C_e}\ \text{E.LAM}_C$$

$$\frac{\begin{array}{c}\Delta \vdash^t t : \sigma_a \\ \vdash^{\leqslant} \sigma_a \leqslant C^k\sigma^k : \_ \rightsquigarrow C_F \\ \Gamma; C_FC^k; \sigma_a \vdash^e e : \sigma_e \rightsquigarrow C_e\end{array}}{\Gamma; C^k; \sigma^k \vdash^e (e :: t) : \sigma_e \rightsquigarrow C_e}\ \text{E.ANN}_C$$

$$\frac{\begin{array}{c}\Gamma_t; \Gamma_t, \Gamma; \Gamma_p; C^k; C_p \vdash^d d : \Gamma_t; \Gamma_p \rightsquigarrow C_p; C_d \\ \Gamma_p; C_d; \sigma^k \vdash^e b : \sigma \rightsquigarrow C_e\end{array}}{\Gamma; C^k; \sigma^k \vdash^e \textbf{let}\ d\ \textbf{in}\ b : \sigma \rightsquigarrow C_e}\ \text{E.LET}_C$$

Within environment Γ, expecting the type of expression e to be $C^k\sigma^k$, e has type σ, under constraints C.

- e : Expression
- σ^k : Expected/known type of expression
- σ : Type of expression
- Δ : Environment $\overline{\iota \mapsto \sigma}$ for type identifiers, cannot be modified (hence treated as a global constant in rule E.ANN)
- Γ : Environment $\overline{\iota \mapsto \sigma}$ for value identifiers
- C^k : Already known constraints
- C : C^k+ new constraints

Figure 4.3.: Expression type rules (C)

$$\boxed{\Gamma_t^k;\Gamma_p^k;\Gamma;C_p^k;C^k \vdash^d d : \Gamma_t;\Gamma_p \rightsquigarrow C_p;C}$$

$$\frac{\begin{array}{c}\Delta \vdash^t t : \sigma_i \\ \Gamma_i \equiv [\, i \mapsto \sigma_i \,]\end{array}}{\_;\Gamma_p;\_;C_p^k;C^k \vdash^d (i :: t) : \Gamma_i;\Gamma_p \rightsquigarrow C_p^k;C^k}\ \text{D.TYSIG}_C$$

$$\frac{\begin{array}{c}v \text{ fresh} \\ p \mapsto \sigma_s \in \Gamma_t^k \\ \sigma_p^k \equiv \sigma_s \wedge \sigma_e^k \equiv \sigma_s \ \vee\ \sigma_p^k \equiv v \wedge \sigma_e^k \equiv \sigma_p \\ C_p^k;\Gamma_p^k;\sigma_p^k \vdash^p p : \sigma_p;\Gamma_p \rightsquigarrow C_p;\_ \\ \Gamma;C^k;\sigma_e^k \vdash^e e : \_ \rightsquigarrow C_e\end{array}}{\Gamma_t^k;\Gamma_p^k;\Gamma;C_p^k;C^k \vdash^d (p = e) : [\,];\Gamma_p \rightsquigarrow C_p;C_e}\ \text{D.VAL}_C$$

Declaration d has explicit type bindings Γ_t, within explicit bindings Γ_t^k and implicit type bindings $C_p^k\Gamma_p^k$, and type checks within $C^k\Gamma$, yielding additional bindings Γ_p, under constraints C_p (for Γ_p) and C (for Γ).

- d : Declaration
- Γ_t : Environment with new type signature bindings
- Δ : Environment $\overline{\iota \mapsto \sigma}$ for type identifiers, cannot be modified (hence treated as a global constant in rule E.ANN)
- Γ_t^k : Collected Γ_t, used by patterns to extract bindings for pattern variables
- Γ : Environment with known bindings
- C_p^k : Known/gathered constraints during type inference of patterns (i. e. use of type signatures and pattern structure)
- C_p : C_p^k+ new constraints
- Γ_p^k : Known/gathered pattern variable bindings
- Γ_p : Γ_p^k+ new bindings
- C^k : Known/gathered constraints during type inference of expressions bound to patterns
- C : C^k+ new constraints

Figure 4.4.: Declaration type rules (C)

attr *AllExpr* [| *tyCnstr* : C |]

For a type rule the (already) known constraints C^k correspond to the use of **lhs**.*tyCnstr* and the result constraints (like C_a for rule E.APP) are assigned to **lhs**.*tyCnstr*. For rule E.APP this translates to the following implementation:

```
sem Expr
  | App (func.gUniq, loc.uniq1)
                  = mkNewLevUID @lhs.gUniq
      func.knTy   = [mkTyVar @uniq1 ] `mkArrow` @lhs.knTy
      (loc.ty_a_, loc.ty_)
                  = tyArrowArgRes @func.ty
      arg .knTy   = @ty_a_
      loc .ty     = @arg.tyCnstr ⊕ @ty_
```

The freshness of a type variable is guaranteed by threading a seed *gUniq* for unique values (*UID*'s) through the AST:

```
attr AllNT [| gUniq : UID |]
```

When a unique value is needed we use *gUniq*'s current value as the 'fresh' UID, and pass the adapted seed onwards to the first child which requires unique values as well.

Our type rules and their corresponding AG description preserve the following invariant:

- The resulting type *ty* has all known constraints applied to it.

This invariant is not preserved for *knTy* and *valGam*, which also can contain type variables. The application of constraints to these attributes is postponed until the following places (in the AST or other code):

- Such an attribute is used in a setting which may yield new constraints, that is, it is used by $\leqslant$ *//fitsIn*.
- Such an attribute is used to return a type of an expression, for example in rule E.VAR. We apply known constraints to the type extracted from Γ*//valGam*.

Variable occurrences *Var* The abovementioned invariant and non-invariant are illustrated by the AG code for rule E.VAR, in which **lhs**.*tyCnstr* is applied to both the expected type *knTy* and the type *ty_g_* extracted from *valGam*:

```
sem Expr
  | Var (loc.ty_g_, loc.nmErrs)
                  = valGamLookupTy @nm @lhs.valGam
      loc.fo_     = fitsIn (@lhs.tyCnstr ⊕ @ty_g_)
                           (@lhs.tyCnstr ⊕ @lhs.knTy)
      lhs.tyCnstr = foCnstr @fo_ ⊕ @lhs.tyCnstr
      loc.ty      = foTy @fo_
```

Newly found constraints (from $\leqslant$ *//fitsIn*) are combined with the already known constraints (**lhs**.*tyCnstr*).

Tuples The construction of tuples is handled by the combination of rule E.CON and rule E.APP. It is now the responsibility of rule E.CON to return the proper function type for constructing a tuple. In the previous version we could use the expected type, which was guaranteed to be available in *knTy*. This information is no longer available, so we use the arity *n* encoded in the name '*,n*' of the constructor to compute the constructor function type. This function is polymorphic, this is the only place where we need to deal with polymorphism for this version of EH. We compute the function type using fresh type variables:

```
sem Expr
  | Con (lhs.gUniq, loc.uniq1)
                    = mkNewLevUID @lhs.gUniq
        loc.tvars_  = map mkTyVar (mkNewUIDL (hsnProdArity @nm) @uniq1)
           .fo_     = fitsIn (@tvars_ `mkArrow` mkProdApp @tvars_)
                             (@lhs.tyCnstr ⊕ @lhs.knTy)
        lhs.tyCnstr = foCnstr @fo_ ⊕ @lhs.tyCnstr
        loc.ty      = foTy @fo_
```

The remaining rules follow the same strategy of applying constraints to types, matching types and propagating the resulting constraints. We omit their implementation, except for rule E.LET which we discuss later in this chapter.

Some observations are in place:

- The main difference with the previous implementation is the use of type variables to represent unknown knowledge. Previously □ was used for that purpose, for example, the rule E-LAM2 and its implementation show that fresh type variables v_i in $v_1 \to v_2$ are used instead of $\square \to \square$ to enforce a $.. \to ..$ structure. If □ still would be used, for example in:

  let *id* = $\lambda x \to x$
  in *id* 3

  the conclusion would be drawn that $id :: \square \to \square$: the absence of the (type variable) identity of x's type has as a consequence that we do not infer $id :: Int \to Int$ from the application *id* 3. So, □ represents "unknown knowledge", a type variable v represents "not yet known knowledge" to which the inferencing process later has to refer to make it "known knowledge".
- Type variables are introduced under the condition that they are "fresh". For a typing rule this means that these type variables are not in use elsewhere, often more concretely specified with a condition $v \notin ftv(\Gamma)$.

4.4 Type inference for pattern expressions (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. We now have to use the occurrence of an identifier in a pattern or expression to infer type information about the identifier. 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 (Fig. 4.5). Again, constraints are threaded through the pattern to accumulate information about type variables. For example, rule P.INT enforces the known type to be an *Int*:

$$\frac{\vdash^{\leqslant} C^k\sigma^k \leqslant \mathit{Int} : \_ \rightsquigarrow C_f}{C^k;\Gamma;\sigma^k \vdash^{p} \mathit{int} : \mathit{Int};\Gamma \rightsquigarrow C_f C^k;\square}\ \text{P.INT}_C$$

The use of $\leqslant$ in patterns is opposite compared to the use in expressions. Patterns are used to dissect values as opposed to expressions, which construct values. Hence values flow into the pattern, simulated by fitting the known type into the pattern type.

The final new ingredient is the use of *pattern function* type σ_{pf}, the type which encodes the structure of the dissection occurring in a composite pattern. A composite pattern takes a value and extracts its components. We encode this by a function type σ_{pf}, which takes this value and returns a tuple holding the components. The rules are organized in such a way that this type is computed in rule P.CON; by doing so, we already prepare for data types where the data constructor determines how a value should be dissected. However, for this version we only have tuple constructors, hence σ_{pf} is the identity function on tuples of size equal to the arity of the pattern. In rule P.CON (Fig. 4.5) we use the structure of the identifier to determine the arity and compute σ_{pf}.

The remainder of the computation of the type of a pattern is similar to and yet more straightforward than for expressions. The rule P.VAR (Fig. 4.5) binds the identifier to the known type and if no such known type is available it invents a fresh one, by means of *tyEnsureNonAny*:

```
sem PatExpr
  | Var (lhs.gUniq, loc.uniq1)
                      = mkNewLevUID @lhs.gUniq
      loc.ty_p_       = tyEnsureNonAny @uniq1 @lhs.knTy
        .valGam_i_    = if @lhs.inclVarBind ∧ @nm ≢ hsnWild
                        then gamUnit @nm (ValGamInfo @ty_p_)
                        else emptyGam
      lhs.valGam      = gamAddGam @valGam_i_ @lhs.valGam
        .patFunTy     = Ty_Any
      loc.ty          = @ty_p_
```

$$\boxed{C^k; \Gamma^k; \sigma^k \vdash^p p : \sigma; \Gamma \rightsquigarrow C; \sigma_{pf}}$$

$$\frac{\vdash^{\leqslant} C^k \sigma^k \leqslant Int : \_ \rightsquigarrow C_f}{C^k; \Gamma; \sigma^k \vdash^p int : Int; \Gamma \rightsquigarrow C_f C^k; \square}\ \text{P.INT}_C$$

$$\frac{\begin{array}{c}\overline{v} \text{ fresh}, \ |\ I\ | \equiv |\ \overline{v}\ | \\ \sigma_p \equiv (\sigma_1, ..., \sigma_n), \quad [\sigma_1, ..., \sigma_n] \equiv \overline{v}\end{array}}{C^k; \Gamma; \sigma^k \vdash^p I : \square; \Gamma \rightsquigarrow C^k; \sigma_p \to \sigma_p}\ \text{P.CON}_C$$

$$\frac{\begin{array}{c}\sigma_p \equiv \sigma^k, \quad \sigma_p \not\equiv \square \\ \Gamma_i \equiv [i \mapsto \sigma_p]\end{array}}{C^k; \Gamma; \sigma^k \vdash^p i : \sigma_p; \Gamma_i, \Gamma \rightsquigarrow C^k; \square}\ \text{P.VAR}_C$$

$$\frac{\begin{array}{c}\sigma_p \equiv \sigma^k, \quad \sigma_p \not\equiv \square \\ \Gamma_i \equiv [i \mapsto \sigma_p] \\ C^k; \Gamma; \sigma_p \vdash^p p : \_; \Gamma_p \rightsquigarrow C_p; \_\end{array}}{C^k; \Gamma; \sigma^k \vdash^p i\ @p : C_p \sigma_p; \Gamma_i, \Gamma_p \rightsquigarrow C_p; \square}\ \text{P.VARAS}_C$$

$$\frac{\begin{array}{c}\vdash^{\leqslant} C^k \sigma^k \leqslant \sigma_a : \sigma \rightsquigarrow C_f \\ C_f C^k; \Gamma; \sigma^k \vdash^p p : \sigma; \Gamma_p \rightsquigarrow C; \sigma_a \to \sigma_r \\ \overline{\sigma} \equiv [\sigma_1, ..., \sigma_n], \quad (\sigma_1, ..., \sigma_n) \equiv \sigma_r \\ |\ \overline{\sigma}\ | \equiv n\end{array}}{C^k; \Gamma; \sigma^k \vdash^p p : C\ \sigma; \Gamma_p \rightsquigarrow C; \square}\ \text{P.APPTOP}_C$$

$$\frac{\begin{array}{c}\sigma_f, \sigma_a \equiv (\sigma_1, ..., \sigma_{n-1}), \sigma_n, \quad (\sigma_1, ..., \sigma_{n-1}, \sigma_n) \equiv \sigma^k \\ C^k; \Gamma; \sigma_f \vdash^p p_1 : \sigma; \Gamma_f \rightsquigarrow C_f; \sigma_{pf} \\ C_f; \Gamma_f; \sigma_a \vdash^p p_2 : \sigma; \Gamma_a \rightsquigarrow C_a; \_\end{array}}{C^k; \Gamma; \sigma^k \vdash^p p_1\ p_2 : \sigma; \Gamma_a \rightsquigarrow C_a; \sigma_{pf}}\ \text{P.APP}_C$$

Knowing the type of pattern p to be $C^k \sigma^k$, p has type σ and bindings Γ (for identifiers introduced by p), under constraints C

σ^k : Known type of pattern

σ_{pf} : The type which encodes the value dissection as a function type, from value to tuple (holding the constituents of the value)

σ : Type of pattern p

C^k : Already known constraints

C : C^k+ new constraints

Γ^k : Already gathered bindings (for this EH version initially [ ])

Γ : Γ^k+ new bindings

Figure 4.5.: Pattern expression type rules (C)

The dissection occurring in a pattern is represented by the pattern function σ_{pf} of the form $\sigma \rightarrow (\sigma_1, ...)$. Conceptually this function takes the value (of type σ) to be dissected by the pattern into its constituents. For now, because we have only tuples to dissect, the function returned by the *Con* alternative is just the identity on tuples of the correct size:

```
sem PatExpr
  | Con (lhs.gUniq, loc.uniq1)
                        = mkNewLevUID @lhs.gUniq
        loc.tvars_      = map mkTyVar (mkNewUIDL (hsnProdArity @nm) @uniq1)
           .ty_p_       = mkProdApp @tvars_
        lhs.patFunTy    = [ @ty_p_ ] `mkArrow` @ty_p_
        loc.ty          = Ty_Any
```

At the top of a pattern, in rule P.APPTOP, this function $\sigma_{pf}//patFunTy$ is dissected into the argument $\sigma_a//ty\_a\_$ and result $\sigma_r//ty\_r\_$:

```
sem PatExpr
  | AppTop loc      .fo_fitR_     = fitsIn (@lhs.tyCnstr ⊕ @lhs.knTy)
                                           @ty_a_
           patExpr.tyCnstr        = foCnstr @fo_fitR_ ⊕ @lhs.tyCnstr
           (loc.ty_a_, loc.ty_r_) = tyArrowArgRes @patExpr.patFunTy
           loc      .tys_         = tyProdArgs @ty_r_
                    .arityErrs    = if length @tys_ == @patExpr.arity
                                    then [ ]
                                    else [Err_PatArity @lhs.knTy @patExpr.arity]
           patExpr.knTyL          = reverse @tys_
           lhs      .patFunTy     = Ty_Any
           loc      .ty           = @patExpr.tyCnstr ⊕ foTy @fo_fitR_
```

The argument σ_a, representing the value "going in the pattern", is matched with the expected type of the pattern; the result σ_r is dissected in rule P.APP as in the previous EH version:

```
sem PatExpr
  | App (loc.ty_a_, loc.tyi_l)
                        = tyLHdAndTl @lhs.knTyL
        func.knTyL      = @tyi_l
             .knTy      = Ty_Any
        arg  .knTy      = @ty_a_
        lhs  .patFunTy  = @func.patFunTy
        loc  .ty        = @func.ty
```

The pattern function type $\sigma_{pf}//patFunTy$ is constructed from fresh type variables. Each occurrence of a tuple pattern deals with different unknown types; hence fresh type variables are needed. The availability of polymorphism in later versions of EH allows us to describe this in a more general way.

4.5 Declarations (Let, Decl)

Again, at the level of declarations all is tied together (Fig. 4.3 and Fig. 4.4). We can no longer assume that type signatures are specified for all value expressions. The basic strategy for declarations (see Section 3.3.5, page 53) must be changed as follows:

- Parallel to *patValGam* and *valGam* we need to gather information about introduced type variables. C_p*//patTyCnstr* gathers information about the types of type variables for identifiers introduced as part of pattern expressions; *C//tyCnstr* gathers information from the use of those identifiers in expressions.
- If a type signature is defined for the toplevel identifier of a pattern in a value declaration (rule D.VAL) we use that type as the known type for both pattern and expression. Otherwise, a fresh type variable is used for the pattern and the pattern type for the value expression.
- The pattern constraints *patTyCnstr* is threaded independently through all declarations, only to be used as the starting point for *tyCnstr* in rule E.LET.

Here we omit the corresponding AG code: it follows the type rules faithfully. $\wp\!\!\!\wp_W$

4.6 Conclusion

In this chapter we have described the second version of EH, that is, monomorphic type inference. Types are still simple (no polymorphism), but may be omitted. Type inference uses the full program to reconstruct types. The next version adds polymorphic, that is Hindley-Milner, type inference.

4. EH 2: Monomorphic type inferencing

5 EH 3: Polymorphic type inferencing

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

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

gives:

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

The polymorphic identity function *id* accepts a value of any type *a*, and returns 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`, pretty-printed as ∀. The meaning of this quantifier is that a value with a universally quantified type can be used with different types for the quantified type variables.

The type signature may be omitted, and in that case the same type will still be inferred. However, the reconstruction of the type of a value for which the type signature is omitted has its limitations, the same as for Haskell98 [84]. Haskell98 restricts what can be described by type signatures by allowing a quantifier only at the beginning of a type signature. In this version of EH we also disallow the explicit use of a quantifier in a type expression (for a type signature); the quantifier is inserted by the implementation.

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 correct:

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

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_3_0 -> c_3_0 <= v_9_0 -> Int
                   problem with : c_3_0 <= Int -}
    id :: a -> a
    id = \x -> x
    {- [ f:forall a . (a -> a) -> Int, id:forall a . a -> a ] -}
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. In our implementation this is encoded as a type constant `c_` chosen for a during type checking the body of f. By definition this type constant is a type a programmer can never define nor denote. The consequence is that an attempt to use i in the body of f, which has type `c_..`→`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:

$$
\begin{array}{l}
\textbf{let}\ f = \lambda i \to i\ 3 \\
\quad \mathit{id} :: a \to a \\
\textbf{in}\ \ \textbf{let}\ v = f\ \mathit{id} \\
\quad \textbf{in}\ f
\end{array}
$$

for which the compiler will infer the following 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 yet 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:

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

The following error is produced:

```
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.

These problems can be overcome by allowing higher ranked polymorphism in type signatures. Later versions of EH deal with this problem (Chapter 6). This version of EH resembles Haskell98 in these restrictions.

The reason not to allow explicit types to be of assistance to the type inferencer is that Haskell98 and this version of EH have as a design principle that all explicitly specified types in a program are redundant. That is, after removal of explicit type signatures, the type inferencer can still reconstruct all types. It is guaranteed that all reconstructed types are the same as the removed signatures or more general, that is, a special case of the inferred types. This guarantee is called the principal type property [17, 76, 38]. However, type inferencing also has its limits. In fact, the richer a type system becomes, the more difficult it is for a type inferencing algorithm to make the right choice for a type without the programmer specifying additional type information.

5.1 Type language

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

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

An f stands for a fixed type variable, a type variable which may not be constrained but still stands for an unknown type. A f is used for fresh type constants, and corresponds to a so called *skolemized type variable*, a type variable which we want to restrict its scope for and inhibit its binding. We do not restrict its scope as we guarantee freshness: two fresh f's by definition do not match.

A v stands for a plain type variable as used in the previous EH version. A series of consecutive quantifiers in $\forall\alpha_1.\forall\alpha_2. ... \sigma$ is abbreviated to $\forall\overline{\alpha}.\sigma$.

The type language suggests that a quantifier may occur anywhere in a type. This is not the case, quantifiers may only be on the top of a type; this version of EH takes care to ensure this. A second restriction is that quantified types are present only in an environment Γ whereas no $\forall$'s are present in types used when type inferencing expressions and patterns. This is to guarantee the principal type property. We do not reflect this in the type language, as we drop this restriction in subsequent EH versions.

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

w^{W}_{W}

```
data Ty
  | Var    tv     : {TyVarId}
           categ  : TyVarCateg
data TyVarCateg
  | Plain
  | Fixed
data Ty
  | Quant  tv     : {TyVarId}
           ty     : Ty
```

We will postpone the discussion of type variable categories until Section 5.2.1.

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

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

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

$$\frac{\begin{array}{c}\iota \mapsto \sigma_g \in \Gamma \\ \sigma_i \equiv C_i\sigma',\ \forall\overline{v}.\sigma' \equiv \sigma_g,\ C_i \equiv \overline{[v \mapsto v_i]},\ \overline{v_i} \text{ fresh} \\ \vdash^{\leqslant} C^k\sigma_i \leqslant C^k\sigma^k : \sigma \rightsquigarrow C\end{array}}{\Gamma; C^k; \sigma^k \vdash^e \iota : \sigma \rightsquigarrow C\ C^k}\ \text{E.VAR}_{HM}$$

$$\frac{\begin{array}{c}\Delta \vdash^t t : \sigma_a \rightsquigarrow \Delta_t; \overline{v_t} \\ \sigma_q \equiv \forall(ftv\ (\sigma_a)\backslash(\overline{v_t})).\sigma_a \\ \sigma_i \equiv C_i\sigma',\ \forall\overline{v}.\sigma' \equiv \sigma_q,\ C_i \equiv \overline{[v \mapsto f]},\ \overline{f} \text{ fresh} \\ \vdash^{\leqslant} \sigma_i \leqslant C^k\sigma^k : \_ \rightsquigarrow C_F \\ \Gamma; C_F C^k; \sigma_i \vdash^e e : \sigma_e \rightsquigarrow C_e\end{array}}{\Gamma; C^k; \sigma^k \vdash^e (e :: t) : \sigma_a \rightsquigarrow C_e}\ \text{E.ANN}_{HM}$$

$$\frac{\begin{array}{c}\Gamma_t; \Gamma_t, \Gamma; \Gamma_p; C^k; C_p \vdash^d d : \Gamma_t; \Gamma_p \rightsquigarrow C_p; C_d \\ \Gamma_l, \Gamma_g \equiv \Gamma_p \\ \Gamma_q \equiv [i \mapsto \forall\overline{\alpha}.\sigma \mid (i \mapsto \sigma) \leftarrow C_d\Gamma_l, \overline{\alpha} \equiv ftv\ (\sigma) - ftv\ (C_d\Gamma_g)] \\ \Gamma_q, \Gamma_g; C_d; \sigma^k \vdash^e b : \sigma \rightsquigarrow C_e\end{array}}{\Gamma; C^k; \sigma^k \vdash^e \textbf{let}\ d\ \textbf{in}\ b : \sigma \rightsquigarrow C_e}\ \text{E.LET}_{HM}$$

Figure 5.1.: Expression type rules (HM)

5.2 Type inferencing

Compared to the previous version the type inferencing process does not change much. Because types used throughout the type inferencing of expressions and patterns do not contain ∀ 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, also often named *type scheme*, is introduced in rule E.LET and instantiated in rule E.VAR (see Fig. 5.1). We will first look at the instantiation.

We note that rule E.LET (Fig. 5.1) has become more complex than the versions appearing in standard treatments of HM type inference. This is a consequence of the combination of the following factors:

- Explicit type annotations are allowed, and have to be propagated to identifiers in

$$\boxed{\Gamma_t^k;\Gamma_p^k;\Gamma;C_p^k;C^k \vdash^d d : \Gamma_t;\Gamma_p \rightsquigarrow C_p;C}$$

$$\frac{\begin{array}{c}\Delta \vdash^t t : \sigma_i \rightsquigarrow \_;\overline{v_t} \\ \sigma_q \equiv \forall(ftv\ (\sigma_i)\backslash(\overline{v_t})).\sigma_i \\ \Gamma_i \equiv [\,i \mapsto \sigma_q\,]\end{array}}{\_;\Gamma_p;\_;C_p^k;C^k \vdash^d (i :: t) : \Gamma_i;\Gamma_p \rightsquigarrow C_p^k;C^k}\ \text{D.TYSIG}_{HM}$$

$$\frac{\begin{array}{c}v \text{ fresh} \\ p \mapsto \sigma_s \in \Gamma_t^k \\ \sigma_i \equiv C_i\sigma',\ \forall\overline{v}.\sigma' \equiv \sigma_s,\ C_i \equiv \overline{[v \mapsto f]},\ \overline{f} \text{ fresh} \\ \sigma_p^k \equiv \sigma_i \wedge \sigma_e^k \equiv \sigma_i\ \vee\ \sigma_p^k \equiv v \wedge \sigma_e^k \equiv \sigma_p \\ C_p^k;\Gamma_p^k;\sigma_p^k \vdash^p p : \sigma_p;\Gamma_p \rightsquigarrow C_p;\_ \\ \Gamma;C^k;\sigma_e^k \vdash^e e : \_ \rightsquigarrow C_e\end{array}}{\Gamma_t^k;\Gamma_p^k;\Gamma;C_p^k;C^k \vdash^d (p = e) : [\,];\Gamma_p \rightsquigarrow C_p;C_e}\ \text{D.VAL}_{HM}$$

Figure 5.2.: Declaration type rules (HM)

patterns.

- The **let** expression allows mutual recursive definitions, necessitating the introduction of bindings for value identifiers to placeholders (type variables) before normal type inference can proceed.
- Although data types are not included in this EH version, the required infrastructure is already available by exposing patterns to the global value environment (which will contain type bindings for data constructors)[1].

5.2.1 Instantiation

A quantified type is introduced in the type inferencing process whenever a value identifier having that type occurs in an expression (rule E.VAR, Fig. 5.1). 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, possibly finding more information about the type variables. Rule E.VAR shows how the type bound to an identifier is instantiated by replacing its quantified type variables with fresh ones. It is assumed that quantifiers occur only at the top of a type.

[1] In future EH versions, this part will be moved to the EH version dealing with data types.

5.2.2 Quantification

The other way around, quantifying a type, happens when a type is bound to a value identifier and added to an environment Γ. The way this is done varies with the presence of a type signature. Rule E.LET (Fig. 5.1), rule D.TYSIG, and rule D.VAL (Fig. 5.2) specify the respective variations:

- A type signature (for an identifier) is specified explicitly, in rule D.TYSIG (partial type signatures are supported, but discussed in Chapter 10). The (quantified) type signature is made available via rule E.LET to rule D.VAL where it must be instantiated as the expected type of both pattern and expression.
- A type (for an identifier) is inferred. Rule D.VAL has no type signature to use as the expected type; a type variable is used instead. At the boundary of its scope, in rule E.LET, we generalise over those type variables in the type which do not occur outside the scope.

A type signature itself is specified without explicit use of quantifiers. These need to be added for all introduced type variables in the type expression for the signature. Rule D.TYSIG shows how a quantified type is computed by wrapping the type in the quantifier $\forall // Ty\_Quant$. In the implementation we wrap the type in *Ty_Quant*, one for each free type variable. w4W

We now run into a problem which will be solved more elegantly in the next version of EH. In a declaration of a value (rule D.VAL) the type signature acts as a known type against which checking of the value expression takes place. Which type do we use for that purpose, the quantified or the unquantified type signature?

- Suppose the unquantified signature $a \to a$ is used in the following fragment. Then, for the erroneous

  let $id :: a \to a$
  $id = \lambda x \to 3$
  in ...

  we end up with fitting $v_1 \to Int \leqslant a \to a$. This can be accomplished via constraints $[v_1 \mapsto Int, a \mapsto Int]$. However, a may only be chosen by the caller of *id*. Instead it now is constrained by the body of *id* to be an *Int*. We must inhibit the binding of a as part of the known type of the body of *id*.
- Alternatively, quantified signature $\forall a.a \to a$ 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, encoded by a type variable with category *TyVarCateg_Fixed*. Rule D.VAL instantiates the type signature, which will be used as the expected type for both the pattern and value expression, with fixed type variables f. If no type signature is defined for the value declaration, a fresh type variable is used for the pattern and the pattern type is used for the expression; the implementation specifies this precisely. w4W

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

5.2.3 Generalisation/quantification of inferred types

How do we determine if a type for some expression bound to an identifier in a value declaration is polymorphic? If a type signature is given, the signature itself describes the polymorphism explicitly by means of type variables. 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 e

In expression e the value bound to id will only be used via id. So, if the inferred type $v_1 \rightarrow v_1$ for the expression $\lambda x \rightarrow x$ has free type variables (here: v_1) and these type variables are not used in the types of other bindings, in particular those in the global Γ, we know that the expression $\lambda x \rightarrow x$ nor any other type will constrain those free type variables. The type for such a type variable can apparently be freely chosen by the expression using id, which is exactly the meaning of the universal quantifier. These free type variables are the candidate type variables over which quantification can take place, as described by the typing rules for **let**-expressions in Fig. 5.1 and its implementation.

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

let $h :: a \rightarrow a \rightarrow a$
  $f = \lambda x \rightarrow$ **let** $g = \lambda y \rightarrow (h\ x\ y, y)$
    in $g\ 3$
in f `'x'`

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

```
let h :: a -> a -> a
    f = \x -> let g = \y -> (h x y,y)
                  {- [ g:Int -> (Int,Int) ] -}
              in g 3
    {- [ f:Int -> (Int,Int), h:forall a . a -> a -> a ] -}
in f 'x'
```

```
{- ***ERROR(S):
     In `f 'x'':
       ... In ``x'':
             Type clash:
               failed to fit: Char <= Int
               problem with : Char <= Int -}
```

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

let $id1 = \lambda x \rightarrow x$
  $id2 = \lambda x \rightarrow x$
  $v_1 = id1\ 3$
in **let** $v_2 = id2\ 3$
  in v_2

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 the smallest groups of mutually dependent definitions, each of these called a binding group. These groups are then ordered according to "define before use" order. Here, for EH, all declarations in a **let**-expression automatically form a binding group, the ordering of two binding groups d_1 and d_2 has to be done explicitly using sequences of **let** expressions: **let** d_1 **in let** d_2 **in**....

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

let $f_1 = \lambda x \rightarrow g_1\ x$

$$
\begin{aligned}
&g_1 = \lambda y \rightarrow f_1\ y \\
&f_2 :: a \rightarrow a \\
&f_2 = \lambda x \rightarrow g_2\ x \\
&g_2 = \lambda y \rightarrow f_2\ y \\
&\textbf{in}\ 3
\end{aligned}
$$

This results in

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

5.3 Conclusion

In this chapter we have described the third version of EH, that is, classic Hindley-Milner polymorphic type inference. The main difference with the previous version is the generalisation for types bound to value identifiers in **let**-expressions, and their instantiation for each use of those value identifiers.

Type expression can specify polymorphic types, but the specification of universal quantifiers at arbitrary positions in a type is dealt with in the next EH version.

Hindley-Milner type inference HM type inference has been introduced [17, 76], used [84, 77], and described before [89, 48]. Our implementation uses the same traditional technique of substitutions and their eager application to types and environments. This has the following advantages:

- We can focus more on the description.
- We can experiment with language extensions without the complication of alternate inference techniques.

However, both of these advantages also are disadvantages:

- For later EH versions the application of substitutions becomes more intricate, and thus less understandable.
- Recent language extensions require constraint solving not to be eager but to be delayed (e.g. GADT's [88, 93]).

In recent years, constraint based approaches [94, 80] are becoming more popular, and seem to more elegantly implement HM type inference and the type checking of more advanced language extensions. The approach is to first gather constraints and later solve these gathered constraints. This allows greater expressiveness, but moves the complexity into the constraint solver: constraint resolution is no longer syntax-directed.

It is (yet) unclear how a shift towards a constraint based approach influences the understandability of EH implementations, although we expect that for a first encounter with type inference the classical approach will benefit understanding the most. However, we also expect that constraint based approaches allow greater flexibility, extensibility, and improved error handling [35].

5. EH 3: Polymorphic type inferencing

6 EH 4: Local quantifier propagation

In the fourth EH version we deal, in the most general form possible, with the presence of quantifiers in types: we allow quantifiers, both universal (∀) and existential (∃), everywhere in a type signature. This offers great flexibility and richness when specifying type signatures, but we can no longer rely on type inferencing to find these type signatures for us. In general, it is impossible to infer types with universal quantifiers at arbitrary positions in a type; type inference for rank-2 is possible, but complex [40, 53, 54, 55].

In this thesis we therefore tackle this problem not by a clever inferencing algorithm, but by focussing on the propagation of explicit, programmer supplied type information to the places in a program where this information is relevant. We thus rely on the programmer to specify 'difficult' type signatures. Our implementation exploits these type signatures to type check and infer types for those parts for which no type signature has been given, similar to other approaches [110, 92].

We describe our solution in three parts:

- In this chapter we start with motivating examples. We then describe how we propagate type information, in particular the information related to the ∀ quantifier, 'locally' through the AST, where 'locally' means neighbouring (parent and children) nodes in the AST.
- In Chapter 7 we propagate type information 'globally' through the AST, where 'globally' means that we relax on the previous 'neighbouring' condition[1].
- In Chapter 8 we add existential quantification.

We also use a notational convention that allows the omission of explicit introduction of quantifiers in type expressions. We will discuss this in Chapter 10.

[1] It has been implemented as a separate branch from EH4 of the sequence of EH compilers. It is not yet part of the full sequence of compilers.

6.1 Motivating examples

The following is an example for demonstrating the usefulness of a universal quantifier at a higher-ranked position.

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

The *rank position* of an argument is defined to be one higher than the function type in which the argument occurs, with rank 1 as the base case: The ∀ quantifier in this example thus is in a rank-2 position. The *rank* of a type is the maximum of the rank positions of quantifiers in a type. The advantage of a higher-ranked type is that inside *f*'s body the argument-bound function *i* can be used polymorphically; in the same way as the **let**-bound function *id* can be used polymorphically.

Rank-2 polymorphism allows argument-bound and **let**-bound functions to be treated in the same way: both may be polymorphic. This is not the case for pure Hindley-Milner type inference, which excludes higher-ranked polymorphism. The advantage of this restriction is that removal of explicitly specified type signatures from a program still yields the same (or more general) typing of values (principal type property). However, this advantage turns into a hindrance when a programmer needs higher-ranked types, and is also willing to specify these types.

Shan [99] presents an overview of Haskell examples gathered from literature which exploit higher-ranked polymorphism. The examples either implement generic behavior or encapsulation. We repeat examples of both, but do not discuss the examples any further in detail; they are included to illustrate that higher-ranked types indeed are useful.

Generic use of higher-ranked polymorphism Generic traversals can be implemented by a function with the following interface [62]:

$$gmapT :: (\forall a.Term\ a \Rightarrow a \to a) \to (\forall b.Term\ b \Rightarrow b \to b)$$

The idea is that, given a transformation function for any type belonging to the class *Term*, another transformation can be constructed. The parameter of this function is a universally quantified function; hence *gmapT* is a higher-ranked (rank-2) function.

Another example of the use of rank-n types is their use in the translation of type-indexed functions with kind-indexed types used in generic programming [71].

Higher ranked polymorphism used for encapsulation The previous use of higher-ranked types deals with polymorphic functions; encapsulation deals with polymorphic val-

ues. For example, *runST* [64] runs a state thread, where *s* represents the state thread being run:

$$runST :: \forall a.(\forall s.ST\ s\ a) \rightarrow a$$

The implementation of *runST* cannot do anything with type *s*, since it cannot assume anything about it. As far as *runST*'s implementation is concerned *s* is hidden, or encapsulated. Haskell (confusingly) uses the ∀ quantifier for existential quantification.

This use of a higher-ranked value corresponds to existential quantification ∃. We allow the use of ∃ as a language construct in its own right (Chapter 8).

6.2 Design overview

The previous version of EH uses two mechanisms for the propagation of type information:

- Expected types are passed top-to-bottom through the AST, whereas result (or inferred) types travel bottom-to-top.
- Unknown types are encoded by type variables. Additional type information about these type variables is encoded in sets of constraints which travel through the complete AST.

In this version of EH we do not change this strategy. We extend the type language with universally quantified types and allow these types to participate in the type inference process. As a consequence, type variables can bind to quantified types; allowing this is called *impredicativity*. Throughout this and subsequent chapters describing EH4, we will further discuss impredicativity and its propagation, called *quantifier propagation*.

Type language The type language used in this chapter is the same as the type language used by the previous EH version. We repeat its definition:

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

Participation of ∀ types in the type inference process Standard HM type inference assumes a separation between type schemes and (monomorphic) types. A *type scheme* is a (possibly) quantified type, with the quantifier at the outer level of the type; a monomorphic type is completely quantifier free. In **let** expressions, type schemes are stored in environments Γ, whereas monomorphic types participate in the type inference process.

In this version of EH, we drop this restriction:

Values (expressions, terms):

e	$::= int \mid char$	literals
	$\mid i$	program variable
	$\mid e\ e$	application
	$\mid$ **let** $\overline{d}$ **in** e	local definitions
	$\mid \lambda p \rightarrow e$	abstraction

Declarations of bindings:

d	$::= i :: t$	value type signature
	$\mid p = e$	value binding

Pattern expressions:

p	$::= int \mid char$	literals
	$\mid i$	pattern variable
	$\mid i\ @p$	pattern variable, with subpattern
	$\mid (p, ..., p)$	tuple pattern

Type expressions:

t	$::= Int \mid Char$	type constants
	$\mid t \rightarrow t$	function type
	$\mid (t, ..., t)$	tuple type
	$\mid i$	type variable
	$\mid \forall i.t$	universal quantification

Identifiers:

ι	$::= i$	lowercase: (type) variables
	$\mid I$	uppercase: (type) constructors

Figure 6.1.: EH terms

- Types with or without quantifiers may live in environments Γ, and they may participate in the type inference process.
- Types retrieved from an environment Γ are no longer instantiated immediately after retrieval, because we want to retain quantifier information as long as possible.

Types are quantified either because a programmer has specified a type signature with a quantifier, or because the type inferencer has decided that a monomorphic type may be universally quantified over its (non-global) type variables. These quantified types may now enter the type inferencing process when extracted from an environment Γ or when passed top-to-bottom through the AST as the expected type of an expression.

This has the following consequences:

- Equating two types (by means of fitting) must take into account the presence of quantifiers.
- Instantiation of types is postponed until the latest moment possible, that is, until an uninstantiated type is to be matched with another type. Hence fitting must deal with instantiation as well.
- Type variables can also be bound to quantified types (called *impredicativity*). Here non-determinism arises because we can interchange binding and instantiation. We may first instantiate a type and then bind it to a type variable, or bind it directly to a type variable and delay its instantiation. Both are allowed to happen.
- Because our strategy is to propagate polymorphism instead of reconstructing it, our encoding of polymorphism places quantifiers at a position which guarantees that their instantiation happens as late as possible. We will come back to this in Chapter 10.
- If a type signature is passed top-down into an expression as the expected type, the type of expression has to match this type: this is type checking. If no such type is available, we resort to type inferencing. In both cases type matching fills in the gaps represented by type variables.

Let us look at some examples to see how this works out in different contexts. We repeat our initial example:

Example 6.1

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

Checking against specified type signature For id we have specified type signature $\forall a.a \to a$, which will be the expected type of $\lambda x \to x$ in the value declaration for id. Before proceeding with type inference for $\lambda x \to x$ we need to match a fresh type $v_1 \to v_2$ (representing the required type structure of the λ-expression) with the expected type, in order to decompose the expected type into argument and result (for further use lower in the AST):

$$v_1 \to v_2 \leqslant \forall a.a \to a$$

Because the signature for id states that we cannot choose the quantified type variable a freely in the lambda expression $\lambda x \to x$ we need to instantiate "$\forall a.a \to a$" with a fixed type variable f_3 for a:

$$v_1 \to v_2 \leqslant f_3 \to f_3$$

Use of polymorphic function as a function In f's body, function i will be retrieved from the environment Γ for use in application "i 3". At the occurrence of i in "i 3", we know that i's expected type is a function type, but we do not (yet) know what its argument and result type are: "$v_4 \to v_5$". i's type (from the environment) must match the expected type "$v_4 \to v_5$":

$$\forall a.a \to a \leqslant v_4 \to v_5$$

Type "$\forall a.a \to a$" fits in "$v_4 \to v_5$" if we instantiate "$\forall a.a \to a$" with the fresh type variable v_6:

$$v_6 \to v_6 \leqslant v_4 \to v_5$$

HM type inference instantiates a type immediately after retrieval from the environment Γ, our approach postpones instantiation until it can no longer be avoided.

Use of polymorphic value as an argument when the expected argument type is known Function f gets passed id as its argument; id's type must fit in f's argument type:

$$\forall a.a \to a \leqslant \forall a.a \to a$$

This is treated as a combination of the previous two matches.

Use of polymorphic value as an argument when the expected argument type is being inferred The real tricky point arises when the type of f's argument is not known, for example if no type signature is specified for f:

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

The argument type of f then still is a type variable v:

$$\forall a.a \to a \leqslant v$$

Is v to be bound to "$\forall a.a \to a$" (being impredicative) or to the instantiated "$v_1 \to v_1$"? There is no way to tell. Only the context in which the matching takes place can specify how to bind: before or after instantiation.

As a general rule we bind impredicatively (that is, without instantiation). However, for a function application we instantiate the type of the argument before binding because (as a design choice) we want to mimic Haskell's type inferencing behavior. As a consequence of binding non-impredicatively we cannot infer a type for f (from our example), because i (f's argument) is used monomorphically in the body of f. Function i can not be applied

polymorphically. This, of course, can be remedied by putting back the type signature for f as in Example 6.1.

In Chapter 7 we will investigate how we can exploit the presence of quantified types even more.

Soundness and completeness Although we make no (formally proven) claims about the type system(s) described in this thesis, we intend our type systems to be sound and complete in the sense described by the remainder of this section. We present our intent by means of the following definition and theorems.

Definition 6.2 HM typing *types an expression e according to Hindley-Milner type inference. If an expression types according to HM rules, we denote this by the following typing judgement, which types e in context* Γ *with type* σ*:*

$$\Gamma \vdash^{HM} e : \sigma$$

Similarly, System F typing *and* EH typing *respectively type an expression e according to System F with type annotations for all expressions and the EH4 type inference algorithm described in this (and following chapters).*

$$\Gamma \vdash^{F} e : \sigma$$
$$\Gamma \vdash^{EH} e : \sigma \rightsquigarrow \vartheta; \vartheta_a$$

ϑ_a represents the translation of e with System F type annotations; ϑ represents the translation of e without additional System F type annotations.

The annotated translation ϑ_a requires additional abstract syntax, but otherwise its computation only consists of moving types to argument positions of function applications. For this EH version e and ϑ are syntactically equal.

These judgement forms are exclusively used to relate EH's type system to the HM and system F type system. We intend EH's type system to be a conservative extension with respect to HM:

Theorem 6.3 (Completeness with respect to HM, or, conservative extension) *All expressions e which type according to HM typing also type according to EH typing:*

$$\Gamma \vdash^{HM} e : \sigma \quad \Rightarrow \quad \Gamma \vdash^{EH} e : \sigma \rightsquigarrow \vartheta; \vartheta_a$$

The other way around, when restricting EH expressions to those types HM can deal with, we claim:

Theorem 6.4 (Soundness with respect to HM) *If the expression e types according to EH typing, σ and all types participating in type inference are rank-1 types, then its translation ϑ types according to HM typing:*

$$\Gamma \vdash^{EH} e : \sigma \rightsquigarrow \vartheta; \vartheta_a \quad \Rightarrow \quad \Gamma \vdash^{HM} \vartheta : \sigma$$

6. EH 4: Local quantifier propagation

For EH without restrictions we claim:

Theorem 6.5 (Soundness with respect to System F) *If the expression e types according to EH typing then its translation ϑ_a (type annotated e) types according to System F typing:*

$$\Gamma \vdash^{EH} e : \sigma \rightsquigarrow \vartheta; \vartheta_a \quad \Rightarrow \quad \Gamma \vdash^{F} \vartheta_a : \sigma$$

These theorems express the following:

- When no type signatures are specified, or only rank-1 type signatures are specified, EH's type inference is as clever as HM type inference. We do not invent higher-ranked polymorphism.
- When type signatures are specified for all value definitions and anonymous λ-expressions, EH is equivalent to System F.

6.3 It all boils down to fitting

Fitting ($\leqslant$) is the place where all these issues come together. Type matching has to deal with $\forall$ quantifiers, and allows for some control of its behavior by the context in which $\leqslant$ is used. We first look at options we will provide as context to $\leqslant$, next we look at their use in previous and new typing rules. In the implementation of $\leqslant$ (*fitsIn*) this corresponds to an additional parameter.

Fig. 6.3 shows, relative to the previous EH version, an additional o as context for $fitsIn /\!/ \leqslant$. In the implementation this will be represented by a value of type *FIOpts* (**f**its**I**n **opt**ion**s**), a set of boolean flags. A $FIOpts /\!/ o$ uses the flags from Fig. 6.4 for obtaining the previously discussed desired behavior. These options are used in specific combinations throughout the type rules (see Fig. 6.5 for an overview). *True* and *False* values are denoted by an additional + or − respectively, for example for fi_{r-bind} with fi^{+}_{r-bind} and fi^{-}_{r-bind} respectively.

$w^{\mathcal{W}}_{\mathcal{W}}$

We use the named combinations of these flags during type inferencing (Fig. 6.6). The name of a combination also suggests a (intuitive) meaning. For example, o_{str} stands for a strong context where the expected type is fully known. The actual flags associated with o_{str} are used in the rules for matching (Fig. 6.2).

The rules for type matching differ from their previous version in the following additions and modifications:

- Rule M.FORALL.L instantiates with fresh type variables, for further binding during type matching and type inference. Rule M.FORALL.R instantiates with fresh fixed type variables, for further use in type checking. The fixed type variables, once again, simulate unknown types chosen by the user of the value with the quantified type.

$$\boxed{o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{I_1 \equiv I_2}{o \vdash^{\cong} I_1 \cong I_2 : I_2 \rightsquigarrow [\,]}\ \textsc{M.CON}_{I1} \qquad \frac{v_1 \equiv v_2}{o \vdash^{\cong} v_1 \cong v_2 : v_2 \rightsquigarrow [\,]}\ \textsc{M.VAR}_{I1}$$

$$\frac{C \equiv [v \mapsto \sigma] \quad fi^{+}_{l-bind} \in o}{o \vdash^{\cong} v \cong \sigma : \sigma \rightsquigarrow C}\ \textsc{M.VAR.L1}_{I1} \qquad \frac{C \equiv [v \mapsto \sigma] \quad fi^{+}_{r-bind} \in o}{o \vdash^{\cong} \sigma \cong v : \sigma \rightsquigarrow C}\ \textsc{M.VAR.R1}_{I1}$$

$$\frac{\sigma_i \equiv C_\alpha \sigma_1,\ C_\alpha \equiv \overline{\alpha \mapsto v},\ \overline{v} \text{ fresh} \quad o \vdash^{\leqslant} \sigma_i \leqslant \sigma_2 : \sigma \rightsquigarrow C}{o \vdash^{\leqslant} \forall \overline{\alpha}.\sigma_1 \leqslant \sigma_2 : \sigma \rightsquigarrow C}\ \textsc{M.FORALL.L}_{I1}$$

$$\frac{\sigma_i \equiv C_\alpha \sigma_2,\ C_\alpha \equiv \overline{\alpha \mapsto f},\ \overline{f} \text{ fresh} \quad o \vdash^{\leqslant} \sigma_1 \leqslant \sigma_i : \sigma \rightsquigarrow C}{o \vdash^{\leqslant} \sigma_1 \leqslant \forall \overline{\alpha}.\sigma_2 : C\ (\forall \overline{\alpha}.\sigma_2) \rightsquigarrow C}\ \textsc{M.FORALL.R}_{I1} \qquad \frac{C \equiv [v \mapsto \sigma] \quad fi^{-}_{l-bind} \in o}{o \vdash^{\cong} v \cong \sigma : \sigma \rightsquigarrow C}\ \textsc{M.VAR.L2}_{I1}$$

$$\frac{C \equiv [v \mapsto \sigma] \quad fi^{-}_{r-bind} \in o}{o \vdash^{\cong} \sigma \cong v : \sigma \rightsquigarrow C}\ \textsc{M.VAR.R2}_{I1}$$

$$\frac{fi^{+}_{r-bind}, fi^{+}_{l-bind}, o \vdash^{\cong} \sigma^a_2 \cong \sigma^a_1 : \sigma_a \rightsquigarrow C_a \quad o \vdash^{\cong} C_a \sigma^r_1 \cong C_a \sigma^r_2 : \sigma_r \rightsquigarrow C_r}{o \vdash^{\cong} \sigma^a_1 \to \sigma^r_1 \cong \sigma^a_2 \to \sigma^r_2 : C_r \sigma_a \to \sigma_r \rightsquigarrow C_r C_a}\ \textsc{M.ARROW}_{I1}$$

$$\frac{o \vdash^{\cong} \sigma^l_1 \cong \sigma^l_2 : \sigma_l \rightsquigarrow C_l \quad o \vdash^{\cong} C_l \sigma^r_1 \cong C_l \sigma^r_2 : \sigma_r \rightsquigarrow C_r}{o \vdash^{\cong} (\sigma^l_1, \sigma^r_1) \cong (\sigma^l_2, \sigma^r_2) : (C_r \sigma_l, \sigma_r) \rightsquigarrow C_r C_l}\ \textsc{M.PROD}_{I1}$$

σ_l matches σ_r under constraints C, $\sigma \equiv C\ \sigma_r$
C : Additional constraints under which matching succeeds
o : Options to steer $\cong$, encodes matching variants as well
σ_l : Type to match
σ_r : Type to match
σ : Result type

Figure 6.2.: Type matching (related to ∀) (I1)

$$\boxed{o \vdash^{\leqslant} \sigma_l \leqslant \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}{o \vdash^{\leqslant} \sigma_l \leqslant \sigma_r : \sigma \rightsquigarrow C}\ \text{FIT}_{I1}$$

$\leqslant$ delegates to $\cong$.
o : Options to $\cong$
σ_l : Type to fit in σ_r
σ_r : Type in which σ_l must fit
σ : $\sigma \equiv C\ \sigma_r$

Figure 6.3.: Fitting of types (I1)

Option	meaning	default
fi_{r-bind}	prefer binding of a rhs tvar over instantiating	fi^{+}_{r-bind}
fi_{l-bind}	prefer binding of a lhs tvar over instantiating	fi^{+}_{l-bind}

Figure 6.4.: Options to *fitsIn*$/\!/\ \leqslant$

- The rules for binding type variables are split into two groups to emphasize the order in which the rules are to be used: rule M.VAR.L1 and rule M.VAR.R1, textually precede the rules for quantified types; Rule M.VAR.L2 and rule M.VAR.R2 are positioned after the rules for quantified types. These rules only differ in the value of fi_{r-bind}. The order in which the rules are textually ordered now is important because they overlap. The idea is that fi^{+}_{r-bind} (in rule M.VAR.R1) triggers binding before instantiation (in the quantifier related rules), and fi^{-}_{r-bind} the other way around.
- Rule M.ARROW for function types matches the argument types with the binding flags set to *True*. In this way higher-ranked type information will be propagated. The binding flags thus only influence rank-1 quantifiers. Only when a higher-ranked type is referred to by means of an identifier (in an expression) with that type, it will be treated (by means of further matching) as a rank-1 type.
- Co- and contravariance now matters. For
  $\sigma^a_1 \to \sigma^r_1 \leqslant \sigma^a_2 \to \sigma^r_2$
  we match the result types σ^r_1 and σ^r_2 in the same direction: $\sigma^r_1 \leqslant \sigma^r_2$. The result type of a function type is called *co-variant* because matching of the complete type and its result part are matched in the same direction. On the other hand, the argument types

Combination	options (relative to the default)	context
o_{str}		strong
$o_{inst-lr}$	fi^-_{l-bind},fi^-_{r-bind}	left and right instantiating

Figure 6.5.: Option combinations

are matched in the opposite direction: $\sigma^a_2 \leqslant \sigma^a_1$. This is called *contra-variance*. For the argument part of a function type this translates to the intuition that $\sigma^a_1 \to \sigma^r_1$ can be used where $\sigma^a_2 \to \sigma^r_2$ is expected, provided that a use of $\sigma^a_2 \to \sigma^r_2$ passes an argument σ^a_2 that can be used where a σ^a_1 is expected. Here, this means that a polymorphic type σ^a_2 can be instantiated to the expected type σ^a_1.

6.4 Type inference

Flags are passed to $\leqslant$ at a limited number of locations in the type rules for expressions (Fig. 6.6, Fig. 6.7). Rule D.VAL specifies that all expressions use o_{str} to do matching, for example in rule E.VAR. The exception is located in rule E.APP. For the argument of a function instantiating takes precedence over binding. Hence $o_{inst-lr}$ is passed to the argument in rule E.APP.

No further changes are required for type inference for expressions. There is no need to adapt inference for pattern expressions: identifiers are bound to the types extracted from the expected types that are passed to pattern expressions.

Option tweaking It is possible to deviate from Haskell at a function application by passing different flags to the argument:

Pass fi^+_{r-bind} (instead of fi^-_{r-bind}) The effect of this modification can best be observed from the following example:

$$
\begin{array}{l}
\textbf{let } g \;::\; (\forall\, a.a \to a) \to \mathit{Int} \\
\quad \mathit{id} = \lambda x \to x \\
\quad f \;\; = \lambda h \to \textbf{let } y \;\; = g\ h \\
\qquad\qquad\qquad\quad\; x_1 = h\ 3 \\
\qquad\qquad\qquad\quad\; x_2 = h\ \texttt{'x'} \\
\qquad\qquad\quad \textbf{in } x_1 \\
\textbf{in } f\ \mathit{id}
\end{array}
$$

First assume that we are still using fi^-_{r-bind}. Then we can infer from the call '$g\ h$':

$h :: f \to f$

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

$$\frac{\iota \mapsto \sigma_g \in \Gamma \qquad o \vdash^{\leqslant} C^k\sigma_g \leqslant C^k\sigma^k : \sigma \rightsquigarrow C}{o; \Gamma; C^k; \sigma^k \vdash^e \iota : \sigma \rightsquigarrow C\ C^k}\ \text{E.VAR}_{I1}$$

$$\frac{\begin{array}{c} v \text{ fresh} \\ o_{str}; \Gamma; C^k; v \to \sigma^k \vdash^e e_1 : \_ \to \sigma \rightsquigarrow C_f \\ o_{inst-lr}; \Gamma; C_f; v \vdash^e e_2 : \_ \rightsquigarrow C_a \end{array}}{o; \Gamma; C^k; \sigma^k \vdash^e e_1\ e_2 : C_a\sigma \rightsquigarrow C_a}\ \text{E.APP}_{I1}$$

$$\frac{\begin{array}{c} v_1, v_2 \text{ fresh} \\ fi^+_{r-bind}, o \vdash^{\leqslant} v_1 \to v_2 \leqslant C^k\sigma^k : \_ \rightsquigarrow C_F \\ o; [\,], \Gamma; C_F C^k; v_1 \vdash^p p : \sigma_p; \Gamma_p \rightsquigarrow C_p; \_ \\ o; \Gamma_p; C_p; v_2 \vdash^e e : \sigma_e \rightsquigarrow C_e \end{array}}{o; \Gamma; C^k; \sigma^k \vdash^e \lambda p \to e : C_e\sigma_p \to \sigma_e \rightsquigarrow C_e}\ \text{E.LAM}_{I1}$$

Within environment Γ and context o, expecting the type of expression e to be $C^k\sigma^k$, e has type σ, under constraints C.

- e : Expression
- o : *fitsIn* options, additional contextual information for $\leqslant$
- σ^k : Expected/known type of expression
- σ : Type of expression
- Δ : Environment $\overline{\iota \mapsto \sigma}$ for type identifiers, cannot be modified (hence treated as a global constant in rule E.ANN)
- Γ : Environment $\overline{\iota \mapsto \sigma}$ for value identifiers
- C^k : Already known constraints
- C : C^k+ new constraints

Figure 6.6.: Expression type rules (I1)

This will lead to errors at the applications 'h 3' and 'h `'x'`'. These errors could have been avoided by concluding at '$g\ h$' that:

$$h :: \forall a.a \to a$$

This is accomplished by using fi^+_{r-bind} instead of fi^-_{r-bind}. This is the desirable behavior because h needs to have this type anyway to be accepted by g. However, we run into problems when we swap the declaration of '$y = g\ h$' with the remaining declarations,

$$\boxed{\Gamma_t^k;\Gamma_p^k;\Gamma;C_p^k;C^k \vdash^d d : \Gamma_t;\Gamma_p \rightsquigarrow C_p;C}$$

$$\frac{\begin{array}{c}\Delta \vdash^t t : \sigma_i \rightsquigarrow \_;\overline{v_t} \\ \overline{v_\Delta} \equiv ftv\ (\Delta) \\ \sigma_q \equiv \forall(ftv\ (\sigma_i)\backslash(\overline{v_t},\overline{v_\Delta})).\sigma_i \\ \Gamma_i \equiv [\,i \mapsto \sigma_q\,]\end{array}}{\_;\Gamma_p;\_;C_p^k;C^k \vdash^d (i :: t) : \Gamma_i;\Gamma_p \rightsquigarrow C_p^k;C^k}\ \text{D.TYSIG}_{I1}$$

$$\frac{\begin{array}{c}v \text{ fresh} \\ p \mapsto \sigma_s \in \Gamma_t^k \\ \sigma_p^k \equiv \sigma_s \wedge \sigma_e^k \equiv \sigma_s \ \vee\ \sigma_p^k \equiv v \wedge \sigma_e^k \equiv \sigma_p \\ o_{str};\Gamma_p^k;C_p^k;\sigma_p^k \vdash^p p : \sigma_p;\Gamma_p \rightsquigarrow C_p;\_ \\ o_{str};\Gamma;C^k;\sigma_e^k \vdash^e e : \_ \rightsquigarrow C_e\end{array}}{\Gamma_t^k;\Gamma_p^k;\Gamma;C_p^k;C^k \vdash^d (p = e) : [\,];\Gamma_p \rightsquigarrow C_p;C_e}\ \text{D.VAL}_{I1}$$

Figure 6.7.: Declaration type rules (I1)

because we infer types in a specific (left to right) order. We then conclude at the application 'h 3':

$h :: Int \to v$

This leads to an error at the application 'h `'x'`'; an error that could have been avoided if we would have known the inferencing results from '$g\ h$'.

We conclude that the order in which we infer (unfortunately) matters. In Chapter 7 we will investigate an approach in which we infer twice: first to extract impredicativeness, and subsequently to do normal type inference.

Pass fi^+_{l-bind} (instead of fi^-_{l-bind}) The effect of this modification can best be observed from the following example:

let *choose* :: $\forall\ a.a \to a \to a$
  id :: $\forall\ a.a \to a$
  v_1 = *choose id*
in v_1

Again, first assume that we are still using fi^-_{l-bind}. At the application '*choose id*', first *id* will be instantiated to $v_1 \to v_1$, and subsequently this type is bound to the instantiated type variable a from *choose*'s type:

choose id :: $(v_1 \to v_1) \to (v_1 \to v_1)$

for which, after generalization, we obtain:

$v_1 :: \forall a.(a \to a) \to (a \to a)$

Alternatively, we might have concluded:

$v_1 :: (\forall a.a \to a) \to (\forall b.b \to b)$

This effect can be achieved by using fi^{+}_{l-bind} instead of fi^{-}_{l-bind}. We then propagate the uninstantiated type. This mechanism can be offered as a mechanism to the programmer. We denote this by a tilde '$\sim$' in front of an argument to indicate System F like propagation of the type of the argument, that is, impredicatively, without instantiation. The use of this notation is restricted to applications where the type of both function and argument are known.

The following rule E.APP.F describes this; the difference with rule E.APP lies in the passing of o_{str}:

$$\frac{\begin{array}{c} v \text{ fresh} \\ o_{str};\Gamma;C^k;v \to \sigma^k \vdash^e e_1 : \_ \to \sigma \rightsquigarrow C_f \\ o_{str};\Gamma;C_f;v \vdash^e e_2 : \_ \rightsquigarrow C_a \end{array}}{o;\Gamma;C^k;\sigma^k \vdash^e e_1\ e_2 : C_a\sigma \rightsquigarrow C_a}\ \text{E.APP.F}_{I1}$$

For example, the following program uses both variants:

```
let choose :: a → a → a
    id     :: a → a
    v1     = choose id
    v2     = choose ~id
in  v1
```

This leads to the following bindings:

```
v1 :: ∀a.(    a → a) → (    a → a)
v2 ::     (∀a.a → a) → (∀b.b → b)
```

Alternatively, we could have provided an explicit type instead, but this is more verbose:

```
let v3 = (choose :: (∀ a.a → a) → (∀ b.b → b) → (∀ c.c → c)) id
    v4 :: (∀ a.a → a) → (∀ b.b → b)
    v4 = choose id
 ...
```

Both v_3 and v_4 have the same type as v_2.

6.5 Conclusion

In this chapter we have described part of the fourth version of EH, that is, the use of type annotations for higher-ranked types. Our approach is to pass these type annotations

downwards through the AST of an expression. Others have also exploited type annotation in a similar way, but we postpone the discussion of related work to Section 7.

In the next chapter we exploit type annotations even further by allowing type information to propagate more globally throughout the AST.

7 EH 4: Global quantifier propagation

In Chapter 6 we added higher-ranked types to EH. If a programmer specifies a type signature, then the system uses this signature for type checking. The idea was to check against a known type signature by distributing such a signature over the AST. We call this *local quantifier propagation* because locally available quantifier related information is used: the expected type is provided by the parent node in the AST. Occasionally we call quantifier propagation *impredicativity inference*, because we allow type variables to be bound to quantified types (called *impredicativity*), and we allow quantified types to participate in the type inference process.

However, we can exploit the presence of type signatures even further by considering function applications as well. The idea is that from the use of a value as an argument for a particular function we can derive type information for that argument based on the (argument) type of the function. Thus we can infer type information, which can be used elsewhere, non-locally, in the AST. The local quantifier propagation from the previous chapter then becomes a special case of what we call *global quantifier propagation*. The following example illustrates this idea:

Example 7.1

$$
\begin{array}{l}
\textbf{let } g \;\; :: (\forall\, a.a \to a) \to \mathit{Int} \\
\quad \mathit{id} = \lambda x \to x \\
\quad f \;\; = \lambda h \to \textbf{let } x_1 = h\ 3 \\
\qquad\qquad\qquad\quad\;\; x_2 = h\ \texttt{'x'} \\
\qquad\qquad\qquad\quad\;\; y \;\; = g\ h \\
\qquad\qquad\qquad\;\; \textbf{in } x_1 \\
\textbf{in } f\ \mathit{id}
\end{array}
$$

From the application '$g\ h$' we can conclude that h certainly must have the following type:

$$h :: \forall a.a \to a$$

A less general type would not be accepted by g. At h's call sites we now can use this inferred type for h to correctly type the applications '$h\ 3$' and '$h\ \texttt{'x'}$', and to infer the higher-ranked type for f. The basic idea behind this approach in this chapter is:

> If a type for an identifier ι has been "touched by", either directly or indirectly, polymorphic type information, then this type information can be used at use sites of ι.

More precisely, "touched by" translates to:

- An identifier occurs in a position where a polymorphic type is expected (direct touching). In particular, argument positions in function applications are used to detect this.
- An identifier has a type which comes from another touched identifier (indirect touching).

So, in our example, h is touched by type "$\forall a.a \to a$". If the application '$g\ h$' would be removed, no touching would take place and both applications 'h 3' and 'h `'x'`' would result in an error: the idea is to propagate polymorphic type information, not invent it.

Choosing the most general type For the following example the same type for h is inferred. It differs from the previous example in that h is expected to be used in two different ways (instead of one), because it is passed to both g_1 and g_2.

Example 7.2

$$
\begin{array}{l}
\textbf{let}\ g_1 :: (\forall\ a.a \to a) \to Int \\
\quad g_2 :: (Int \to Int) \to Int \\
\quad id :: \forall\ a.a \to a \\
\quad f\ \ = \lambda h \to \textbf{let}\ x_1 = g_1\ h \\
\qquad\qquad\qquad\ \ x_2 = g_2\ h \\
\qquad\qquad\ \ \textbf{in}\ x_2 \\
\textbf{in}\ f\ id
\end{array}
$$

Function h is expected to be used as "$\forall a.a \to a$" and "$Int \to Int$". The most general of these types, that is "$\forall a.a \to a$", is bound to h. The relation "more general" is defined in terms of $\leqslant$: "σ_1 is more general than σ_2" is equivalent to $\sigma_1 \leqslant \sigma_2$.

Generality is even further exploited in the following (somewhat contrived) example. It differs from the previous example in that h is not chosen from the set of available expected types, but is the greatest common instance [90] (or least general anti-unification, defined later in this chapter as the meet of two types).

$$
\begin{array}{l}
\textbf{let}\ g_1 :: (\forall\ a.(Int, a) \to (Int, a)) \to Int \\
\quad g_2 :: (\forall\ b.(b, Int) \to (b, Int)) \to Int \\
\quad id\ = \lambda x \to x \\
\quad f\ \ = \lambda h \to \textbf{let}\ y_1 = g_1\ h \\
\qquad\qquad\qquad\ \ y_2 = g_2\ h \\
\qquad\qquad\ \ \textbf{in}\ y_2 \\
\textbf{in}\ f\ id
\end{array}
$$

Here h is expected to be used as "$\forall a.(Int, a) \to (Int, a)$" and "$\forall b.(b, Int) \to (b, Int)$". We choose the type of h (and consequently f) to be:

$h :: \forall a.\forall b.(a, b) \to (a, b)$
$f :: (\forall a.\forall b.(a, b) \to (a, b)) \to Int$

Contravariance Contravariance, that is, the reversal of $\leqslant$ for the arguments of a function type, implies that the "more general" means "less general" for arguments. The following example demonstrates this:

Example 7.3

$$
\begin{array}{l}
\textbf{let } g_1 :: ((\forall\ a.a \to a) \to Int) \to Int \\
\quad g_2 :: ((Int \to Int) \to Int) \to Int \\
\quad id :: \forall\ a.a \to a \\
\quad f \ \ = \lambda h \to \textbf{let } x_1 = g_1\ h \\
\qquad\qquad\qquad\qquad\ \ x_2 = g_2\ h \\
\qquad\qquad\qquad\qquad\ \ h_1 = h\ id \\
\qquad\qquad\qquad\quad \textbf{in } h_1 \\
\quad v \ \ = f\ (\lambda i \to i\ 3) \\
\textbf{in } v
\end{array}
$$

Function h now is expected to be used as "$(\forall a.a \to a) \to Int$" but also as "$(Int \to Int) \to Int$". This means that h is passed a "$\forall a.a \to a$" in g_1's context, so it can use the passed function polymorphically as far as the context is concerned. In g_2's context a "$Int \to Int$" is passed; g_2 expects this function to be used on values of type Int only. Hence we have to choose the least general for the type of the function passed by g_1 and g_2, that is, the argument of h:

$h :: (Int \to Int) \to Int$
$f :: ((Int \to Int) \to Int) \to Int$

Because of the contra-variance of function arguments, the least general type for the function passed by g_1 and g_2 coincides with the most general type for f's argument h.

7.1 Design overview

The design of our solution for the propagation of quantifier related type information is a combination of the following:

- Quantifier propagation, described in this chapter, is the first stage of a two stage process. The second stage consists of the previously described type inference, which exploits expected type information, and determines bindings for type variables. The

stage described in this chapter extracts as much as possible quantifier related type information for type variables, to be used as expected type information by the next stage. Fresh type variables are created once, in the first stage, and retained for use in the following stage, so type variables act as placeholders for inferred types.

- For type variables which represent possibly polymorphic types, we gather all bindings to the types they are expected to have. This is encoded by means of a type holding type alternatives and constraint variants. These types and constraints are computed by a variation of normal HM type inference. Type alternatives resemble intersection types [11]. However, our type alternatives are used only internally and are not available to a programmer as a (type) language construct.
- For each introduced identifier we compute the most (or least, depending on contravariance) general type based on its type alternatives. This results in constraints (for type variables) which are subsequently used by type inference as discussed in earlier versions of EH. For this to work, it is essential that all possible type alternatives are grouped together, including the type information extracted from explicitly specified type signatures.
- The computation of most/least general types is based on the lattice induced by $\leqslant$. Fig. 7.1 shows an example of such a lattice for the examples presented so far. We propagate the result of this computation if the type alternatives used to compute the most/least general type contains a type with a quantifier. Otherwise there is no quantifier related information to propagate. Although we do not discuss existential types in this chapter yet, existential types are included for reasons of symmetry in Fig. 7.1.

We call the resulting strategy *global quantifier propagation*.

7.2 Finding possible quantifiers

The first step in our strategy for global quantifier propagation is to find for a type variable not just one type, but all types it can be matched with. Remember that the reason for this chapter's problem is a too early binding of a type variable to a type. We need to delay that decision by gathering all possible bindings, and extract a polymorphic type from it, if any. Actually, we also need to find out if polymorphism needs to be inhibited. This is a consequence of the contravariance of function arguments.

For instance, in Example 7.2, page 102 we conclude:

$h :: \forall a.a \to a$

This is based on the following type matches:

$h :: v_1$
$v_1 \leqslant \forall a.a \to a$

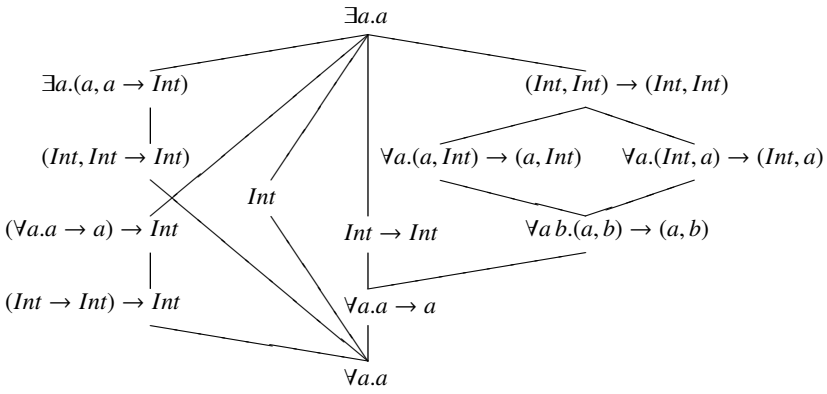

Figure 7.1.: Type lattice (∃ is discussed in Chapter 8)

$v_1 \leqslant Int \to Int$

Our previous approach was to bind v_1 to one of the righthand sides of $\leqslant$. Here we delay this binding by binding the type variable v_1 to both v_1 and its binding alternatives. We use a *type alternatives* to represent this (see Fig. 7.3 and Fig. 7.2):

$$\begin{array}{lll} \sigma ::= & ... & \\ & \mid \mathfrak{o} & \text{type alternatives} \\ \mathfrak{o} ::= & v\ [\overline{\varphi}] & \text{type alternatives} \end{array}$$

We denote types σ which contain type alternatives by $\mathfrak{o}$. Types $\mathfrak{o}$ only participate in quantifier propagation.

Each type alternative φ corresponds to an alternative type σ, together with additional information about the context in which this type is used. We need to know this context when type alternatives are reduced to a most/least general type. First, we need to know at which side of $\leqslant$ a type occurred. For example, in Example 7.3, page 103 we conclude:

$h :: (Int \to Int) \to Int$

This is based on the following type matches:

$$\begin{array}{ll} h & :: v_2 \to Int \\ \forall a.a \to a & \leqslant v_2 \\ Int \to Int & \leqslant v_2 \end{array}$$

Here two types have to fit in v_2. In the previous example, a type variable v_1 had to fit the other way around, in two types. We call this fitting direction the *type alternative need*, denoted by $\mathbb{N}$. The direction of the current example is marked as a *type alternative offering*,

Notation	Meaning
$\mathfrak{o}$	σ for quantifier propagation
σ_Q	σ with a quantifier
$\sigma_{\neg Q}$	σ without a quantifier
$\mathbb{C}$	C for quantifier propagation
$\vartriangle$	meet of two types
$\triangledown$	join of two types
$\cong$	$\leqslant$, $\vartriangle$ or $\triangledown$
$\mathbb{H}$	type alternative hardness (hard or soft)
$\mathbb{H}_h$	hard type alternative
$\mathbb{H}_s$	soft type alternative
$\mathbb{N}$	type alternative need/context (offered or required)
$\mathbb{N}_o$	offered type alternative
$\mathbb{N}_r$	required type alternative
φ	type alternative

Figure 7.2.: Notation for quantifier propagation

denoted by $\mathbb{N}_o$, because the two types are offered to be fit in the type variable. The direction of the previous example is marked as a *type alternative requirement*, denoted by $\mathbb{N}_r$. We encode this information in a type alternative:

$\varphi ::= \sigma :: \mathbb{H} \ / \ \mathbb{N}$	type alternative
$\mathbb{N} ::= \mathbb{N}_o$	'offered' context
$\mid \ \mathbb{N}_r$	'required' context
$\mathbb{H} ::= \mathbb{H}_h$	'hard' constraint
$\mid \ \mathbb{H}_s$	'soft' constraint

A type alternative also has to remember the *type alternative hardness*, denoted by $\mathbb{H}$. Hardness may be hard, denoted by $\mathbb{H}_h$, or soft, denoted by $\mathbb{H}_s$. By default every type alternative is soft. $\mathbb{H}_h$ is used internally by our quantifier propagation algorithm to mark types without a quantifier to be propagated; this is necessary to inhibit propagation of quantified types. For example, for Example 7.3 we have to conclude the constraint "$v_2 \mapsto Int \to Int$" on v_2, the least general type, and inhibit the propagtion of "$\forall a.a \to a$" as a possible binding for v_2.

For our respective examples we find the following constraints (on v_1 from Example 7.2, v_2 from Example 7.3):

$v_1 \mapsto v_1 \ [\forall a.a \to a :: \mathbb{H}_s \ / \ \mathbb{N}_r, Int \to Int :: \mathbb{H}_s \ / \ \mathbb{N}_r]$
$v_2 \mapsto v_2 \ [\forall a.a \to a :: \mathbb{H}_s \ / \ \mathbb{N}_o, Int \to Int :: \mathbb{H}_s \ / \ \mathbb{N}_o]$

Collecting these constraints is relatively straightforward: if a type variable is to be bound to a type during type matching, we bind it to a type alternative. This behavior is enabled

Types:		
σ	::= *Int* | *Char*	literals
	| v	variable
	| $\sigma \to \sigma$	abstraction
	| $\sigma\ \sigma$	type application
	| $\forall v.\sigma$	universally quantified type
	| f	(fresh) type constant (a.k.a. fixed type variable)
Types for quantifier propagation:		
σ	::= ...	
	| $\mathfrak{o}$	type alternatives
$\mathfrak{o}$	::= $v\ [\overline{\varphi}]$	type alternatives
Types for computing meet/join:		
σ	::= ...	
	| $v{=}\sigma$	both
	| $\square$	absence of type information
Type alternative:		
φ	::= $\sigma :: \mathbb{H} \,/\, \mathbb{N}$	type alternative
$\mathbb{N}$	::= $\mathbb{N}_o$	'offered' context
	| $\mathbb{N}_r$	'required' context
$\mathbb{H}$	::= $\mathbb{H}_h$	'hard' constraint
	| $\mathbb{H}_s$	'soft' constraint

Figure 7.3.: Type language

by an additional flag fi_{alt} to type matching (see Fig. 7.4 and Fig. 7.5). For example, binding to a type alternative is disabled in rule M.VAR.L1 (Fig. 7.6, used previously), and enabled in rule M.VAR.L3 (a new rule). New bindings for type alternatives are combined, for example in rule M.ALT and rule M.ALT.L1.

This mechanism is used by quantifier propagation, preceding normal type inference. We next discuss the computation of most/least general types, and postpone the use of these mechanisms (in Fig. 7.15 and Fig. 7.16) until later.

7.3 Computing actual quantifiers

After the gathering of type alternatives, we compute most/least general types based on these type alternatives. The result of this computation are constraints on type variables.

Option	meaning	default
fi_{r-bind}	prefer binding of a rhs tvar over instantiating	fi^{+}_{r-bind}
fi_{l-bind}	prefer binding of a lhs tvar over instantiating	fi^{+}_{l-bind}
fi_{alt}	bind as type alternative	fi^{-}_{alt}
$fi_{\leqslant}$	fit	$fi^{+}_{\leqslant}$
$fi_{\triangle}$	meet	$fi^{-}_{\triangle}$
$fi_{\triangledown}$	join	$fi^{-}_{\triangledown}$

Figure 7.4.: Options to fit

Combination	options (relative to the default)	context
o_{str}		strong
o_{inst-l}	fi^{-}_{r-bind}	left instantiating
$o_{inst-lr}$	$fi^{-}_{l-bind}, fi^{-}_{r-bind}$	left and right instantiating
o_{im}	fi^{+}_{alt}	impredicative inference
o_{meet}	$fi^{+}_{\triangle}$	meet
o_{join}	$fi^{+}_{\triangledown}$	join

Figure 7.5.: Option combinations

We compute either a most general (polymorphic) type or a least general (usually non-polymorphic) type. These constraints are used by type checking and inferencing, representing additional assumptions for some types.

We need the combination of the following mechanisms:

- The computation of *type meet*'s and *type join*'s for types, using the ordering on types defined by $\leqslant$ and its induced lattice (Fig. 7.1) [18].
- The elimination of type alternatives in a type, and the simultaneous extraction of bindings for type variables to quantified types.

These mechanisms are mutually recursive, because type alternative elimination uses meet/join computation to find (and combine) quantifier information, and meet/join computation may combine (deeper nested) type alternatives.

Meet and join of types The *type meet*, denoted by $\triangle$, and *type join*, denoted by $\triangledown$, of two types σ_1 and σ_2 are defined by [18]:

$$\sigma_1 \triangle \sigma_2 \equiv max\{\sigma \mid \sigma \leqslant \sigma_1 \wedge \sigma \leqslant \sigma_2\}$$
$$\sigma_1 \triangledown \sigma_2 \equiv min\{\sigma \mid \sigma_1 \leqslant \sigma \wedge \sigma_2 \leqslant \sigma\}$$

$$\boxed{o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{C \equiv [v \mapsto \sigma] \quad fi^{-}_{alt}, fi^{+}_{l-bind} \in o}{o \vdash^{\cong} v \cong \sigma : \sigma \rightsquigarrow C}\ \text{M.VAR.L1}_{I2} \qquad \frac{\sigma \equiv v_1[\sigma_2 :: \mathbb{H}_s \,/\, \mathbb{N}_r] \quad C \equiv [v_1 \mapsto \sigma] \quad \sigma_2 \not\equiv \_\,[\_] \quad fi^{+}_{alt}, fi^{+}_{l-bind} \in o}{o \vdash^{\cong} v_1 \cong \sigma_2 : \sigma \rightsquigarrow C}\ \text{M.VAR.L3}_{I2}$$

$$\frac{C \equiv [v \mapsto \sigma] \quad fi^{-}_{alt}, fi^{+}_{r-bind} \in o}{o \vdash^{\cong} \sigma \cong v : \sigma \rightsquigarrow C}\ \text{M.VAR.R1}_{I2} \qquad \frac{\sigma \equiv v_2[\sigma_1 :: \mathbb{H}_s \,/\, \mathbb{N}_o] \quad C \equiv [v_2 \mapsto \sigma] \quad \sigma_1 \not\equiv \_\,[\_] \quad fi^{+}_{alt}, fi^{+}_{r-bind} \in o}{o \vdash^{\cong} \sigma_1 \cong v_2 : \sigma \rightsquigarrow C}\ \text{M.VAR.R3}_{I2}$$

$$\frac{\sigma \equiv v_2[\overline{\varphi_1}, \overline{\varphi_2}] \quad C \equiv [v_1 \mapsto \sigma, v_2 \mapsto \sigma]}{o \vdash^{\leqslant} v_1[\overline{\varphi_1}] \leqslant v_2[\overline{\varphi_2}] : \sigma \rightsquigarrow C}\ \text{M.ALT}_{I2} \qquad \frac{\sigma \equiv v_1[\sigma_2 :: \mathbb{H}_s \,/\, \mathbb{N}_r, \overline{\varphi_1}] \quad C \equiv [v_1 \mapsto \sigma]}{o \vdash^{\leqslant} v_1[\overline{\varphi_1}] \leqslant \sigma_2 : \sigma \rightsquigarrow C}\ \text{M.ALT.L1}_{I2}$$

$$\frac{\sigma \equiv v_2[\sigma_1 :: \mathbb{H}_s \,/\, \mathbb{N}_o, \overline{\varphi_2}] \quad C \equiv [v_2 \mapsto \sigma]}{o \vdash^{\leqslant} \sigma_1 \leqslant v_2[\overline{\varphi_2}] : \sigma \rightsquigarrow C}\ \text{M.ALT.R1}_{I2}$$

Figure 7.6.: Type alternative related matching (finding possible quantified types) (I2)

The relation $\leqslant$ on types is assymetrical due to the presence of a universal quantifier $\forall$ in a type. We have $\forall v.\sigma_1 \leqslant \sigma_2$ if we can instantiate v to some type for which $\sigma_1 \leqslant \sigma_2$. In case of absence of a quantifier in $\sigma_1 \leqslant \sigma_2$, both types must match: $\sigma_1 \cong \sigma_2$. Therefore $\sigma_1 \vartriangle \sigma_2$ represents the type which can be instantiated to both σ_1 and σ_2; $\sigma_1 \triangledown \sigma_2$ represents the type which is an instantiation of both σ_1 and σ_2.

The following use of meet and join constitutes a key part of our algorithm. The type meet $\vartriangle$ is used to extract "$\forall a.a \to a$" from the following example constraint:

$$v_1 \mapsto v_1\ [\forall a.a \to a :: \mathbb{H}_s \,/\, \mathbb{N}_r, \mathit{Int} \to \mathit{Int} :: \mathbb{H}_s \,/\, \mathbb{N}_r]$$

The type variable v_1 represents a type which must fit (because tagged by $\mathbb{N}_r$) into both "$\forall a.a \to a$" and "$\mathit{Int} \to \mathit{Int}$". The type for v_1 (from Example 7.2, page 102) must be the most general of these two types so it can be instantiated to both the required types. This type for v_1 is defined as follows:

$$\forall a.a \to a \equiv \forall a.a \to a \vartriangle \mathit{Int} \to \mathit{Int}$$

$$\boxed{o \vdash^{\leqslant} \sigma_l \leqslant \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{fi^{+}_{\leqslant}, o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}{o \vdash^{\leqslant} \sigma_l \leqslant \sigma_r : \sigma \rightsquigarrow C}\ \text{FIT}_{I2}$$

Figure 7.7.: Fitting of types (I2)

$$\boxed{o \vdash^{\triangle} \sigma_l \triangle \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{fi^{+}_{\triangle}, o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}{o \vdash^{\triangle} \sigma_l \triangle \sigma_r : \sigma \rightsquigarrow C}\ \text{MEET}_{I2}$$

$\triangle$ delegates to $\cong$.
σ_l : Type to meet
σ_r : Type to meet
σ : Result type: $\sigma \leqslant \sigma_l \wedge \sigma \leqslant \sigma_r$

Figure 7.8.: Join of types (I2)

On the other hand, for v_2 (from Example 7.3, page 103) we know it represents a type of a value in which both a value with type "$\forall a.a \to a$" and "$Int \to Int$" will flow.

$$v_2 \mapsto v_2\ [\forall a.a \to a :: \mathbb{H}_s \,/\, \mathbb{N}_o, Int \to Int :: \mathbb{H}_s \,/\, \mathbb{N}_o]$$

The type for v_2 must be the least general of these two types so both contexts can coerce their value to a value of type v_2:

$$Int \to Int \equiv \forall a.a \to a \triangledown Int \to Int$$

The implementation of fit $\leqslant$, meet $\triangle$, and join $\triangledown$ are much alike, so we define their implementation as variations on type matching $\cong$. The rules in Fig. 7.7, Fig. 7.8, and Fig. 7.9 dispatch to $\cong$, and pass the variant by means of additional (mutually exclusive) flags: $fi^{+}_{\leqslant}$, $fi^{+}_{\triangle}$, and $fi^{+}_{\triangledown}$. When the rules for $\cong$ are meant to be used only by a particular variant we either require the presence of the corresponding flag or we use the corresponding denotation ($\leqslant$, $\triangle$, or $\triangledown$) in the rules, as is done in the rules dealing with the meet and join of $\forall$ quantified types in Fig. 7.12.

$$\boxed{o \vdash^{\triangledown} \sigma_l \triangledown \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{fi^{+}_{\triangledown}, o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}{o \vdash^{\triangledown} \sigma_l \triangledown \sigma_r : \sigma \rightsquigarrow C}\ \text{JOIN}I2$$

$\triangledown$ delegates to $\cong$.
σ_l : Type to join
σ_r : Type to join
σ : Result type: $\sigma_l \leqslant \sigma \wedge \sigma_r \leqslant \sigma$

Figure 7.9.: Join of types (I2)

Type alternative elimination The computation of the most/least general type from type alternatives, presented in Fig. 7.10, may look overwhelming at first, but basically selects specific subsets from a set of type alternatives and combines their types by meeting or joining, where the choice between meet and join depends on the (contra)variance. The computation is described by rule TY.AE.ALTS; the remaining rules deal with default cases. In rule TY.AE.ALTS we slightly stretch the notation for matching ($\cong$) by allowing a vector of types to be matched: $\overline{\sigma} \cong \sigma^r$. This means "*foldr* ($\cong$) σ^r $\overline{\sigma}$".

Rule TY.AE.ALTS starts with extracting type alternatives: type alternatives with a quantifier ($\overline{\sigma_Q}$), without a quantifier ($\overline{\sigma_{\mathbb{H}_s}}$), and those marked as hard ($\overline{\sigma_{\mathbb{H}_h}}$). These sets are further restricted by their need $\mathbb{N}$, selecting $\mathbb{N}_r$ in a meet context (flag $fi^{+}_{\triangle}$), selecting $\mathbb{N}_o$ otherwise. Only when quantified or hard types are present we first compute their meet (or join), so we obtain all quantifier related information. Then we combine the result with the remaining types. The result may still contain type alternatives, because we only eliminate the top level type alternatives. We recursively eliminate these nested type alternatives and finally bind the result to the type variable for this set of type alternatives.

We walk through our initial example (Example 7.1), which we repeat here:

```
let g  :: (∀ a.a → a) → Int
    id = λx → x
    f  = λh → let x1 = h 3
                  x2 = h 'x'
                  y  = g h
              in  x1
in f id
```

Our implementation finds the following information for h (the fragments are edited bits of internal administration):

$$\boxed{o; \mathbb{C}^k; \overline{v_g}^{\ \varphi\ elim} \vdash \mathfrak{o} : \sigma \rightsquigarrow \mathbb{C}}$$

$$\frac{\begin{array}{c}\mathbb{N} \equiv \textbf{if } fi^+_\Delta \in o \textbf{ then } \mathbb{N}_r \textbf{ else } \mathbb{N}_o \\ v\ [\overline{\varphi}] \equiv \mathfrak{o} \\ \overline{\sigma_{\mathcal{Q}}} \equiv [\sigma_{\mathcal{Q}} \mid (\sigma_{\mathcal{Q}} :: \mathbb{H}_s \,/\, \mathbb{N}) \leftarrow \overline{\varphi}] \\ \overline{\sigma_{\mathbb{H}_h}} \equiv [\sigma \mid (\sigma :: \mathbb{H}_h \,/\, \mathbb{N}) \leftarrow \overline{\varphi}] \\ o \vdash^{\cong} (\overline{\sigma_{\mathbb{H}_h}}, \overline{\sigma_{\mathcal{Q}}}) \cong \square : \mathfrak{o}_{\mathbb{H}_h} \rightsquigarrow \mathbb{C}_h \\ \overline{\sigma_{\mathbb{H}_s}} \equiv [\sigma_{\neg\mathcal{Q}} \mid (\sigma_{\neg\mathcal{Q}} :: \mathbb{H}_s \,/\, \mathbb{N}) \leftarrow \overline{\varphi}] \\ o \vdash^{\cong} \mathbb{C}_h \overline{\sigma_{\mathbb{H}_s}} \cong \mathfrak{o}_{\mathbb{H}_h} : \mathfrak{o}_{\mathbb{H}_s} \rightsquigarrow \_ \\ o; \mathbb{C}^k; \overline{v_g}^{\ \varphi\ elim} \vdash \mathfrak{o}_{\mathbb{H}_s} : \sigma \rightsquigarrow C_e \\ C \equiv [v \mapsto \sigma] \\ \mid \overline{\sigma_{\mathbb{H}_h}}, \overline{\sigma_{\mathcal{Q}}} \mid > 0 \\ v \notin \overline{v_g}\end{array}}{o; \mathbb{C}^k; \overline{v_g}^{\ \varphi\ elim} \vdash \mathfrak{o} : \sigma \rightsquigarrow C\ C_e}\ \text{TY.AE.ALTS}_{I2}$$

$$\frac{\begin{array}{c}v\ [\_] \equiv \mathfrak{o} \\ v \notin \overline{v_g}\end{array}}{o; \mathbb{C}^k; \overline{v_g}^{\ \varphi\ elim} \vdash \mathfrak{o} : v \rightsquigarrow [\,]}\ \text{TY.AE.VAR}_{I2} \qquad \frac{}{o; \mathbb{C}^k; \overline{v_g}^{\ \varphi\ elim} \vdash \sigma : \sigma \rightsquigarrow [\,]}\ \text{TY.AE.TY}_{I2}$$

$$\frac{\begin{array}{c}o_a \equiv \text{ toggle } fi^+_\Delta \text{ and } fi^+_\nabla \text{ in } o \\ o_a; \mathbb{C}^k; \overline{v_g}^{\ \varphi\ elim} \vdash \mathfrak{o}_a : \sigma_a \rightsquigarrow C_a \\ o; \mathbb{C}^k; \overline{v_g}^{\ \varphi\ elim} \vdash \mathfrak{o}_r : \sigma_r \rightsquigarrow C_r\end{array}}{o; \mathbb{C}^k; \overline{v_g}^{\ \varphi\ elim} \vdash \mathfrak{o}_a \rightarrow \mathfrak{o}_r : \sigma_a \rightarrow \sigma_r \rightsquigarrow C_a C_r}\ \text{TY.AE.ARROW}_{I2}$$

Within a meet/ join context (indicated by o), known constraints $\mathbb{C}^k$ for $\mathfrak{o}, \sigma$ equals $\mathfrak{o}$ in which all type alternatives (except for global type variables $\overline{v_g}$) are eliminated, under $\mathbb{C}$ constraining the type alternative variables to their type alternive eliminated type.

- o : Options to matching, in particular indicating meet/ join
- $\mathfrak{o}$: Type with type alternatives φ
- $\mathbb{C}$: New constraints, constraining the type alternative variables to their type alternive eliminated type
- σ : Result type, without type alternatives φ (for non global type variables)
- $\mathbb{C}^k$: Known constraints for type variables in $\mathfrak{o}$
- $\overline{v_g}$: Global variables (which are not eliminated)

Figure 7.10.: Type alternative elimination (I2)

$$\boxed{o; \mathbb{C}^k; \overline{v_g}^{\ \Gamma\ \varphi\ elim} \vdash \Gamma : \mathbb{C}}$$

$$\frac{o; \mathbb{C}^k; \overline{v_g}^{\ \varphi\ elim} \vdash \mathbb{C}^k\mathfrak{o} : \_ \rightsquigarrow \mathbb{C}_a \qquad o; \mathbb{C}^k; \overline{v_g}^{\ \Gamma\ \varphi\ elim} \vdash \mathbb{C}_a\Gamma : \mathbb{C}_g}{o; \mathbb{C}^k; \overline{v_g}^{\ \Gamma\ \varphi\ elim} \vdash (\_ \mapsto \mathfrak{o}, \Gamma) : \mathbb{C}_g\mathbb{C}_a}\ \text{G.AE.CONS}_{I2}$$

Within a meet/ join context (indicated by o), known constraints $\mathbb{C}^k$ for types in Γ, all type alternatives (except for global type variables $\overline{v_g}$) are eliminated, under constraints $\mathbb{C}$.

Figure 7.11.: Type alternative elimination (for a Γ) (I2)

$$\begin{array}{lll} h :: v\_23\_0 & & \\ v\_23\_0 \mapsto v\_23\_0\ [\forall a.a & \rightarrow a & :: \mathbb{H}_s \,/\, \mathbb{N}_r \\ \quad , \quad (v\_38\_0\ [Int :: \mathbb{H}_s \,/\, \mathbb{N}_o]) & \rightarrow v\_35\_0 & :: \mathbb{H}_s \,/\, \mathbb{N}_r \\ \quad , \quad (v\_47\_0\ [Char :: \mathbb{H}_s \,/\, \mathbb{N}_o]) & \rightarrow v\_44\_0 & :: \mathbb{H}_s \,/\, \mathbb{N}_r \\ \quad] & & \end{array}$$

Function h is used in three different contexts, of which one requires h to be a polymorphic type, and the remaining two require h to be a function which can accept an *Int* and a *Char* argument respectively. Because h must be the most general type we eliminate type alternatives in a fi^+_Δ context. Rule TY.AE.ALTS then extracts type alternative subsets:

$$\begin{array}{lll} \overline{\sigma_Q} \equiv [\forall a.a & \rightarrow a] & \\ \overline{\sigma_{\neg Q}} \equiv [\quad (v\_38\_0\ [Int :: \mathbb{H}_s \,/\, \mathbb{N}_o]) & \rightarrow v\_35\_0 & \\ \quad , \quad (v\_47\_0\ [Char :: \mathbb{H}_s \,/\, \mathbb{N}_o]) & \rightarrow v\_44\_0 & \\ \quad] & & \\ \overline{\sigma_{\mathbb{H}_h}} \equiv [\,] & & \end{array}$$

The solution $\forall a.a \rightarrow a$ does not contain nested type alternatives, so we end with the constraint:

$$v\_23\_0 \mapsto \forall a.a \rightarrow a$$

In the remainder of the type inference process we can now use h polymorphically.

Meet/join computation The computation of the meet Δ and join ∇ of two types is similar to the introduction and elimination of type alternatives:

- Quantified type variables are instantiated with type variables v which remember both

$$\boxed{o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{\begin{array}{c}\overline{v_{\rightleftharpoons}} \overset{\rightleftharpoons elim}{\vdash} \sigma_m : \sigma \rightsquigarrow \_ ; C_e \\ \sigma_i \equiv C_\alpha \sigma_1,\ C_\alpha \equiv \overline{\alpha \mapsto (v_{\rightleftharpoons} \rightleftharpoons \square)},\ \overline{v_{\rightleftharpoons}} \text{ fresh} \\ o \vdash^{\triangle} \sigma_i \triangle \sigma_2 : \sigma_m \rightsquigarrow C_m\end{array}}{o \vdash^{\triangle} \forall\overline{\alpha}.\sigma_1 \triangle \sigma_2 : \forall\overline{v_{\rightleftharpoons}}.\sigma \rightsquigarrow C_e C_m}\ \text{M.FORALL.L2}_{I2}$$

$$\frac{\begin{array}{c}\overline{v_{\rightleftharpoons}} \overset{\rightleftharpoons elim}{\vdash} \sigma_m : \sigma \rightsquigarrow \_ ; C_e \\ \sigma_i \equiv C_\alpha \sigma_1,\ C_\alpha \equiv \overline{\alpha \mapsto (v_{\rightleftharpoons} \rightleftharpoons \square)},\ \overline{v_{\rightleftharpoons}} \text{ fresh} \\ o \vdash^{\triangledown} \sigma_i \triangledown \sigma_2 : \sigma_m \rightsquigarrow C_m\end{array}}{o \vdash^{\triangledown} \forall\overline{\alpha}.\sigma_1 \triangledown \sigma_2 : \forall\overline{v_{\rightleftharpoons}}.C_e\sigma \rightsquigarrow C_e C_m}\ \text{M.FORALL.L3}_{I2}$$

$$\frac{\begin{array}{c}\sigma \equiv v_1[\sigma_2 :: \mathbb{H}_h \,/\, \mathbb{N}_r, \overline{\varphi_1}] \\ C \equiv [v_1 \mapsto \sigma]\end{array}}{o \vdash^{\triangle} v_1[\overline{\varphi_1}] \triangle \sigma_2 : \sigma \rightsquigarrow C}\ \text{M.ALT.L2}_{I2} \qquad \frac{\begin{array}{c}\sigma \equiv v_1[\sigma_2 :: \mathbb{H}_h \,/\, \mathbb{N}_o, \overline{\varphi_1}] \\ C \equiv [v_1 \mapsto \sigma]\end{array}}{o \vdash^{\triangledown} v_1[\overline{\varphi_1}] \triangledown \sigma_2 : \sigma \rightsquigarrow C}\ \text{M.ALT.L3}_{I2}$$

Figure 7.12.: Type meet/join (I2)

the type variable and the type σ (if any) bound (by matching) to the type variable:

$\sigma ::= ...$
| $v \rightleftharpoons \sigma$ both
| $\square$ absence of type information

The instantiation with these types is (for example) done as part of Rule M.FORALL.L2 (Fig. 7.12).

- After instantation and further matching (Fig. 7.13) we end with a type which encodes both a type variable and its binding. We then either forget or use these bindings, depending on the context (meet or join).

For example, in rule M.FORALL.L2 (Fig. 7.12) the meet of

$\forall a.a \to a$
$Int \to Int$

gives σ_m:

$a \rightleftharpoons Int \ \to \ a \rightleftharpoons Int$

The rules in Fig. 7.14 then split this type into a type with type variables, and constraints for those type variables:

$$\boxed{o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{o \vdash^{\Delta\nabla} \sigma_1 \ \Delta\nabla \ \sigma_2 : \sigma \rightsquigarrow C_m \qquad C \equiv [v_1, v_2 \mapsto v_2 \barwedge \sigma]}{o \vdash^{\Delta\nabla} v_1 \barwedge \sigma_1 \ \Delta\nabla \ v_2 \barwedge \sigma_2 : v_2 \barwedge \sigma \rightsquigarrow C \ C_m} \text{M.BOTH}_{I2}$$

$$\frac{C \equiv [v \mapsto v \barwedge \sigma]}{o \vdash^{\Delta\nabla} v \barwedge \square \ \Delta\nabla \ \sigma : v \barwedge \sigma \rightsquigarrow C} \text{M.BOTH.L1}_{I2}$$

$$\frac{o \vdash^{\Delta\nabla} \sigma_1 \ \Delta\nabla \ \sigma_2 : \sigma \rightsquigarrow C_m \qquad C \equiv [v \mapsto v \barwedge \sigma]}{o \vdash^{\Delta\nabla} v \barwedge \sigma_1 \ \Delta\nabla \ \sigma_2 : v \barwedge \sigma \rightsquigarrow C \ C_m} \text{M.BOTH.L2}_{I2}$$

Figure 7.13.: Type matching ($\cong$ on $\barwedge$) (I2)

$\sigma \equiv a \to a$
$C_e \equiv a \mapsto Int$

In case of a meet Δ the constraints C_e are forgotten for the result type. The constraints C_e are still propagated, because other type variables may still be further constrained as a 'side effect' of the meet Δ. For a join ∇ (rule M.FORALL.L3) the constraints are not forgotten but applied to σ_m.

Finally, rule M.ALT.L2 and rule M.ALT.L3 (Fig. 7.12) add a type computed by a meet or join as a hard $\mathbb{H}_h$ type to type alternatives. For types with quantifiers this does not make a difference, but for types without (like $Int \to Int$) it does. Being marked as hard $\mathbb{H}_h$, we ensure the triggering of type alternative elimination (rule TY.AE.ALTS) and subsequent propagation of the resulting type. If a type variable is bound by this process to a (non-polymorphic) type we effectively inhibit its further binding to a polymorphic type.

7.4 Impredicativity inference

Impredicativity inference uses type alternatives and their elimination to respectively gather and extract polymorphism, to be used by subsequent normal type inference. The algorithm (Fig. 7.15, Fig. 7.16, and Fig. 7.17) uses two constraint threads. The first constraint thread, denoted by $\mathbb{C}$, gathers type alternatives, and the second, denoted by C, participates in normal type inference. Both inference stages return a type. The type returned by quantifier propagation may contain type alternatives and is therefore denoted by $\mathfrak{o}$; the type returned

$$\boxed{bv \overset{\neq elim}{\vdash} \sigma_{\neq} : \sigma \rightsquigarrow \sigma_e ; C}$$

$$\frac{v \in bv}{bv \overset{\neq elim}{\vdash} v{\neq}\square : v \rightsquigarrow v ; [\,]}\ \textsc{ty.eb.any}_{I2} \qquad \frac{\begin{array}{c} v \in bv \\ bv \overset{\neq elim}{\vdash} \sigma_b : \sigma \rightsquigarrow v_e ; C \end{array}}{bv \overset{\neq elim}{\vdash} v{\neq}\sigma_b : v \rightsquigarrow v ; [v_e \mapsto v]\, C}\ \textsc{ty.eb.var}_{I2}$$

$$\frac{\begin{array}{c} v \in bv \\ bv \overset{\neq elim}{\vdash} \sigma_b : \sigma \rightsquigarrow \sigma_e ; C \end{array}}{bv \overset{\neq elim}{\vdash} v{\neq}\sigma_b : v \rightsquigarrow \sigma_e ; [v \mapsto \sigma_e]\, C}\ \textsc{ty.eb.ty}_{I2}$$

Split $\sigma_{\neq}$ holding $\neq$ types into σ holding type variables (of $\neq$ types) and C holding constraints on those type variables.

bv : Already bound variables for which no elimination takes place

σ : Result type, $\neq$ types replaced by their original type variable

C : Constraints for $\neq$ type variables, mapping to their $\neq$ type

σ_e : Type where $\neq$ types are replaced by their $\neq$ type (if not $\square$), only used internally

$\sigma_{\neq}$: Type to be $\neq$ type eliminated

Figure 7.14.: Type 'both' elimination (I2)

by normal inference is denoted by σ. We focus on quantifier propagation and its integration with normal type inference (and postpone the discussion of the judgements for constraints superscripted with *ex* required for existential types):

- The complete inference process is split in two stages: quantifier propagation and (normal) type inference.
- Bindings for value identifiers are gathered and propagated via environments. Each binding binds to a type variable, a placeholder for type information, about which specific type information is stored in constraints C. We separate placeholders and actual type information because the two inference stages infer different types for a type variable.
- Constraints for the first stage are denoted by $\mathbb{C}$, for the second stage by C.
- Only the result of type alternative elimination is propagated to the second stage.

Impredicativity inference in isolation follows a similar strategy as type inference for previous versions of EH, in that we gather and match type information, partially bottom-up, partially top-down:

- Expected types are still used, but their matching now is done at those places in the AST where we expect the need for type alternatives: rule E.APP and rule E.LAM.
- At some places in the AST we 'fix' type alternatives by extracting polymorphism; we use the elimination of type alternatives.

For example, in rule E.APP we match the impredicative function type ϖ_f with $v \to \sigma^k$, with the flag fi^+_{alt} passed to $\leqslant$. Any known information about the function's argument is thus bound as a type alternative to v. The argument type is matched similarly, so we end up with all information about the argument bound to v as a set of type alternatives.

Fixating type information is done at two places: at the introduction of identifiers in **let**-bindings and λ-bindings. Similar to the generalisation of HM type inference, these places limit the scope of an identifier. If a type variable is not accessed outside this boundary, we can close the reasoning about such a type by eliminating type alternatives (or quantify, in the case of HM type inference). The restriction on eliminating type alternatives for a pattern, to be used as the known type for the pattern, arises from our combination of type inference and type checking. We hope to remove this restriction in a future version of our algorithm as it complicates rule E.LAM; we will come back to this later with some examples.

The intricacy of rule E.LAM is caused by the combination of the following:

- Type variables act as placeholders for (future) type information. Hence we must take care to avoid inconsistencies between constraints. Inconsistencies arise as the result of double instantiation (during each inference stage), and instantiated type variables are not constrained to be equal when the semantics require this.
- We assume that all known type information is available during the first inference stage, so we can include this information into type alternatives.
- For patterns, only a single 'pass' is used to extract type information. As a consequence we require its types and constraints, used in the first stage, to remain consistent with results from the second stage.

Rule E.LAM first extracts possibly polymorphic information from the known type σ^k, which may contain type alternatives (introduced as part of rule E.APP). The resulting type σ^k_e is used to extract the possible polymorphic (higher ranked) type of the argument. We need this type to ensure the invariant that all available known type information is used as part of the first stage, and becomes bound in a type alternative. After being combined with pattern constraints and being threaded through the body, emerging as $\mathbb{C}_e$, the set of constraints is used to eliminate type alternatives for each introduced identifier.

The tricky part is the combination with the next stage. We need to match with the known type a second time as we may have found new polymorphic types for arguments. However, $w^{\mathcal{W}}_W$

$$\boxed{o; \Gamma; \mathbb{C}^k; C^k; \sigma^k \vdash^e e : \mathfrak{o}; \sigma \rightsquigarrow \mathbb{C}; C}$$

$$\frac{o \vdash^{\leqslant} Int \leqslant C^k\sigma^k : \sigma \rightsquigarrow C}{o; \Gamma; \mathbb{C}^k; C^k; \sigma^k \vdash^e int : Int; \sigma \rightsquigarrow \mathbb{C}^k; C\ C^k}\ \text{E.INT}_{I2}$$

$$\frac{\begin{array}{c}\iota \mapsto \sigma_g \in \Gamma \\ o \vdash^{\leqslant} C^k\sigma_g \leqslant C^k\sigma^k : \sigma \rightsquigarrow C\end{array}}{o; \Gamma; \mathbb{C}^k; C^k; \sigma^k \vdash^e \iota : \mathbb{C}^k\sigma_g; \sigma \rightsquigarrow \mathbb{C}^k; C\ C^k}\ \text{E.VAR}_{I2}$$

$$\frac{\begin{array}{c}v \text{ fresh} \\ o_{str}; \Gamma; \mathbb{C}^k; C^k; v \to \sigma^k \vdash^e e_1 : \mathfrak{o}_f; \_ \to \sigma \rightsquigarrow \mathbb{C}_f; C_f \\ o_{im} \vdash^{\leqslant} \mathfrak{o}_f \leqslant \mathbb{C}_f(v \to \sigma^k) : \_ \rightsquigarrow \mathbb{C}_F \\ o_{inst-lr}; \Gamma; \mathbb{C}_F\mathbb{C}_f; C_f; v \vdash^e e_2 : \mathfrak{o}_a; \_ \rightsquigarrow \mathbb{C}_a; C_a \\ fi^+_{alt}, o_{inst-l} \vdash^{\leqslant} \mathfrak{o}_a \leqslant \mathbb{C}_a v : \_ \rightsquigarrow \mathbb{C}_A \\ \mathbb{C}_1 \equiv \mathbb{C}_A\mathbb{C}_a\end{array}}{o; \Gamma; \mathbb{C}^k; C^k; \sigma^k \vdash^e e_1\ e_2 : \mathbb{C}_1\sigma^k; C_a\sigma \rightsquigarrow \mathbb{C}_1; C_a}\ \text{E.APP}_{I2}$$

Within environment Γ and context o, expecting the types of expression e to be $\mathbb{C}^k\sigma^k$ (and $C^k\sigma^k$), e has type $\mathfrak{o}$ (and σ), under constraints $\mathbb{C}$ (and C).

- e : Expression
- o : *fitsIn* options, additional contextual information for $\leqslant$
- $\mathfrak{o}$: Type (with type alternatives φ) of expression (for quantifier propagation)
- σ^k : Expected/known type of expression
- σ : Type of expression
- Δ : Environment $\overline{\iota \mapsto \sigma}$ for type identifiers, cannot be modified (hence treated as a global constant in rule E.ANN)
- Γ : Environment $\overline{\iota \mapsto \sigma}$ for value identifiers
- $\mathbb{C}^k$: Already known constraints (for quantifier propagation)
- $\mathbb{C}$: $\mathbb{C}^k$+ new constraints (for quantifier propagation)
- C^k : Already known constraints
- C : C^k+ new constraints

Figure 7.15.: Expression type rules, part I (I2)

$$\boxed{o;\Gamma;\mathbb{C}^k;C^k;\sigma^k \vdash^e e : \mathfrak{o};\sigma \rightsquigarrow \mathbb{C};C}$$

$$\frac{\begin{array}{c} v_1, v_2 \text{ fresh} \\ \overline{v_g} \equiv ftv\ (\Gamma) \\ o_{meet};\mathbb{C}^k;\overline{v_g} \overset{\varphi\ elim}{\vdash} \sigma^k : \sigma^k_e \rightsquigarrow \_ \\ fi^+_{r-bind}, o \vdash^{\leqslant} v_1 \to v_2 \leqslant \sigma^k_e : \_ \rightsquigarrow \mathbb{C}_F \\ o;[\,],\Gamma;[\,];\mathbb{C}_F v_1 \vdash^p p : \sigma_p;\Gamma_p \rightsquigarrow C_p;\_ \\ \Gamma_l, \_ \equiv \Gamma_p \\ fi^+_{r-bind}, fi^-_{l,r}, o \vdash^{\leqslant} C^k(v_1 \to v_2) \leqslant C^k\sigma^k : \_ \rightsquigarrow C_F \\ C_2 \equiv C_F \backslash ftv\ (C^k(v_1 \to v_2)) \\ o;\Gamma_p;C_p\mathbb{C}_F\mathbb{C}^k;C_3;v_2 \vdash^e e : \mathfrak{o}_e;\sigma_e \rightsquigarrow \mathbb{C}_e;C_e \\ o_{meet};\mathbb{C}_e;\overline{v_g} \overset{\Gamma\ \varphi\ elim}{\vdash} \Gamma_l : C_\Gamma \\ C_3 \equiv C_\Gamma C_p C_2 (C^k \circledcirc \mathbb{C}_F)\ C^k \\ \mathbb{C}_1 \equiv C_\Gamma \mathbb{C}_e \end{array}}{o;\Gamma;\mathbb{C}^k;C^k;\sigma^k \vdash^e \lambda p \to e : \mathbb{C}_1\sigma_p \to C_\Gamma\mathfrak{o}_e;C_e\sigma_p \to \sigma_e \rightsquigarrow \mathbb{C}_1;C_e}\ \text{E.LAM}_{I2}$$

$$\frac{\begin{array}{c} \overline{v_g} \equiv ftv\ (\Gamma) \\ o_{join};\mathbb{C}_d;\overline{v_g} \overset{\Gamma\ \varphi\ elim}{\vdash} \Gamma_l : C_\Gamma \\ C_t\Gamma_t;\Gamma_t,\Gamma;\Gamma_p;[\,];C^{ex}_t C_t;C_p\mathbb{C}^k;C_\Gamma C_p C^k \vdash^d d : \Gamma_t;\Gamma_p \rightsquigarrow C_t;C_p;\mathbb{C}_d;C_d \\ C^{ex}_t \equiv [\,v_g \mapsto C\ \sigma \mid (i \mapsto v_g) \leftarrow \Gamma_t, \exists\overline{v}.\sigma \equiv C_t v_g, C \equiv \overline{v \mapsto f}, f \text{ fresh}\,] \\ \Gamma_l, \Gamma_g \equiv \Gamma_p \\ C_q \equiv [\,v_g \mapsto \forall\overline{\alpha}.\sigma \mid (i \mapsto v_g) \leftarrow \Gamma_l, \sigma \equiv C_d v_g, \overline{\alpha} \equiv ftv\ (\sigma)\backslash ftv\ (C_d\Gamma_g)\,] \\ C^{ex}_l \equiv [\,v_g \mapsto C\ \sigma \mid (i \mapsto v_g) \leftarrow \Gamma_l, \exists\overline{v}.\sigma \equiv C_q C_d v_g, C \equiv \overline{v \mapsto f}, f \text{ fresh}\,] \\ o;\Gamma_p;C_\Gamma\mathbb{C}_d;C^{ex}_l C_q C_d;\sigma^k \vdash^e b : \mathfrak{o};\sigma \rightsquigarrow \mathbb{C}_e;C_e \end{array}}{o;\Gamma;\mathbb{C}^k;C^k;\sigma^k \vdash^e \textbf{let}\ d\ \textbf{in}\ b : \mathfrak{o};\sigma \rightsquigarrow \mathbb{C}_e;C_e}\ \text{E.LET}_{I2}$$

Figure 7.16.: Expression type rules, part II (I2)

this match may result in fresh instantiated type variables or fixed type variables. Constraint C_3 requires some careful constraint manipulation. New constraints for v_1 (and v_2) are avoided; old bindings for v_1 (and v_2) are updated with new constraints.

In a **let**-expression type alternatives are eliminated for locally introduced bindings. Rule E.LET shows how this is done. Although the propagation of Γ's and constraints specified by rule E.LET is complete it also has become complex. This is mainly the consequence of the use of multiple Γ's and constraints being threaded through all declarations, and being tied together at rule E.LET. Fig. 7.18 therefore provides a graphical summary.

$$\boxed{\Gamma_t^k;\Gamma_p^k;\Gamma;C_t^k;C_p^k;\mathbb{C}^k;C^k \vdash^d d : \Gamma_t;\Gamma_p \rightsquigarrow C_t;C_p;\mathbb{C};C}$$

$$\frac{\begin{array}{c}\Delta \vdash^t t : \sigma_i \rightsquigarrow \_;\overline{v_t} \\ \overline{v_\Delta} \equiv ftv\ (\Delta) \\ \sigma_q \equiv \forall(ftv\ (\sigma_i)\backslash(\overline{v_t},\overline{v_\Delta})).\sigma_i \\ \sigma_v \equiv v_v \ \wedge\ C_v \equiv [\,v_v \mapsto \sigma_q\,] \ \wedge\ v_v \text{ fresh} \\ \Gamma_i \equiv [\,i \mapsto \sigma_v\,]\end{array}}{\_;\Gamma_p;\_;C_t^k;C_p^k;\mathbb{C}^k;C^k \vdash^d (i :: t) : \Gamma_i;\Gamma_p \rightsquigarrow C_vC_t^k;C_p^k;\mathbb{C}^k;C^k}\ \text{D.TYSIG}_{I2}$$

$$\frac{\begin{array}{c}v \text{ fresh} \\ p \mapsto \sigma_s \in \Gamma_t^k \\ \sigma_p^k \equiv \sigma_s \wedge \sigma_e^k \equiv \sigma_s \ \vee\ \sigma_p^k \equiv v \wedge \sigma_e^k \equiv \sigma_p \\ o_{str};\Gamma_p^k;C_p^k;\sigma_p^k \vdash^p p : \sigma_p;\Gamma_p \rightsquigarrow C_p;\_ \\ o_{str};\Gamma;\mathbb{C}^k;C^k;\sigma_e^k \vdash^e e : \mathfrak{o}_e;\_ \rightsquigarrow \mathbb{C}_e;C_e \\ o_{im} \vdash^{\leqslant} \mathfrak{o}_e \leqslant \mathbb{C}_e\sigma_p : \_ \rightsquigarrow \mathbb{C}_E\end{array}}{\Gamma_t^k;\Gamma_p^k;\Gamma;C_t^k;C_p^k;\mathbb{C}^k;C^k \vdash^d (p = e) : [\,];\Gamma_p \rightsquigarrow C_t^k;C_p;\mathbb{C}_E\mathbb{C}_e;C_e}\ \text{D.VAL}_{I2}$$

Declaration d has explicit type bindings Γ_t, within explicit bindings Γ_t^k and implicit type bindings $C_p^k\Gamma_p^k$, and type checks within $C^k\Gamma$, yielding additional bindings Γ_p, under constraints C_p (for Γ_p) and C (for Γ).

- d : Declaration
- Γ_t : Environment with new type signature bindings
- Δ : Environment $\overline{\iota \mapsto \sigma}$ for type identifiers, cannot be modified (hence treated as a global constant in rule E.ANN)
- Γ_t^k : Collected Γ_t, used by patterns to extract bindings for pattern variables
- Γ : Environment with known bindings
- $\mathbb{C}^k$: Known/gathered constraints during quantifier propagation
- $\mathbb{C}$: $\mathbb{C}^k$+ new constraints
- C_p^k : Known/gathered constraints during type inference of patterns (i. e. use of type signatures and pattern structure)
- C_p : C_p^k+ new constraints
- Γ_p^k : Known/gathered pattern variable bindings
- Γ_p : Γ_p^k+ new bindings
- C^k : Known/gathered constraints during type inference of expressions bound to patterns
- C : C^k+ new constraints
- C_t^k : Type signature information represented as constraint for binding to type variable in Γ_t
- C_t : C_t^k+ new constraints

Figure 7.17.: Declaration type rules (I2)

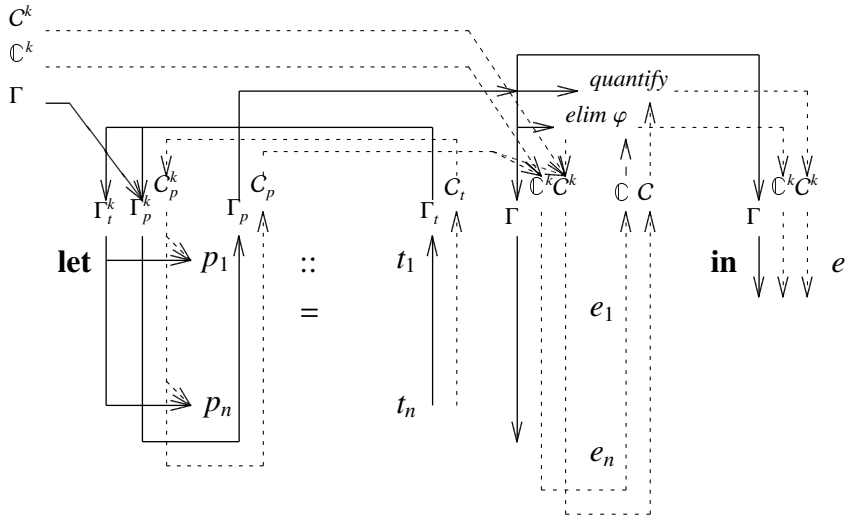

Figure 7.18.: Constraint flow for let expression

Additional complexity arises from the presence of existential types, which we will discuss in Chapter 8. Existential types are part of this version of EH.

Fig. 7.18 shows how rule E.LET first gathers bindings for value identifiers, in parallel with constraints for type variables bound to identifiers. Type signatures are gathered in Γ_t, bindings from patterns are gathered in Γ_p. The corresponding constraints (C_t, and C_p) are propagated to the quantifier propagation constraint thread $\mathbb{C}$. Similar to rule E.LAM these constraints are used to eliminate type alternatives. The result of elimination is propagated to normal type inference.

7.5 Related work, discussion

Literature Higher-ranked types have received a fair amount of attention. Type inference for higher-ranked types in general is undecidable [116]; type inference for rank-2 types is possible, but complex [53]. The combination of intersection types [11] and higher-rankedness [54, 52] appears to be implementable [56].

In practice, requiring a programmer to provide type annotations for higher-ranked types for use by a compiler turns out to be a feasible approach [79] with many practical applications [99, 64, 45]. Some form of distribution of known type information is usually

employed [92, 81, 87]. Our implementation distributes type information in a top-down manner (Chapter 6), and, additionally, distributes type information non-locally (in this chapter).

Quantifier propagation Our approach is to rely on explicitly provided type annotations, and the propagation of this type information. Internally, our implementation uses type alternatives, similar to intersection types. We rely on 'classical' style type inference, with types which can incorporate constraints, and are applied as greedily as possible.

The quantifier propagation described in this chapter is algorithmic of nature. Recent work by Pottier and Rémy [93, 95] takes a similar approach (although in a constraint based setting), calling the propagations process elaboration. Theirs and our approach share the two-pass nature in which the first pass infers missing type annotations. Although we make no claims about the correctness of our algorithm, quantifier propagation only propagates that which is already available in the first place, thus being true to our conservative "don't invent polymorphism" design starting point. We are confident that this approach does not 'break' normal type inference and checking.

Constraint-based approaches provide an alternative point of view where the 'difficult' part of a type is encoded as a constraint, treated separately from type inference [80]. Botlan's extension to ML [15] uses (explicitly specified) constraints to allow polymorphic type information to be propagated impredicatively. Both approaches also allow the integration of qualified types [103, 65].

Whatever the approach taken, the availability of higher-ranked types in a programming language complicates the implementation; this is the price to pay for a bit of System F functionality. Our approach provides such an implementation and, additionally, stretches the exploitation of type annotations even further by propagating impredicativity globally throughout and expression.

For this thesis we have chosen the 'classical' approach as a starting point to keep matters (relatively) simple. Only recently new extensions are expressed using a constraint approach. We expect to use a constraint based approach, because of this and the prospect of better error message [35].

Finally, this chapter reflects an experiment which has not (yet) been integrated into the final of our series of compilers. The combination with a class system (Chapter 9) and partial type signatures (Chapter 10) requires further investigation.

8 EH 4: Existential types

In Chapter 6 universal quantification of types was introduced. A universally quantified type expresses that a value of such a type can be used with any type subsituted for the universally quantified part. In this chapter we extend EH with its counterpart: the *existentially quantified type*, (or *existential type*) [78, 63]. First, we look at examples, then we look at the implementation issues.

The difference between a universally and existentially quantified type can be characterized by the following observation:

- The *use* of a value with a $\forall$ quantified type determines the type to choose for the instantiation of the quantified type variable. For example, the caller of the identity function "$id :: \forall a.a \rightarrow a$" determines the type to choose for the type variable a for this particular application of id. For the function application "id 3" this type equals Int.
- The *creation* of a value with a $\exists$ quantified type determines, and hides, the type of the quantified type variable. For example, a creator of a "$\exists a.(a, a \rightarrow Int)$" may have constructed a value of that type from "$(3, \lambda x \rightarrow x)$"; another creator has constructed a value with the same type from "$(\texttt{'x'}, \lambda x \rightarrow ord\ x)$". From a users point of view both values have the same type and are thus interchangeable. The value has a specific type chosen for type variable a, but we do not know which type, so this information can no longer be exploited. This value specific type information has been 'forgotten'; we only know it exists.

Existential types are available in Haskell [75], be it in a slightly disguised form. If type variables occur in a constructor of a data type, but not in the type itself, they are assumed to be existentially quantified. The keyword `forall` (confusingly) is used to specify this explicitly:

data $Hide = \forall a.Hide\ a$

In EH we prefer to denote this as $\exists a.a$. We do not restrict the occurrences of $\exists$ quantifiers to data declarations.

As pointed out in Section 6.1 the univeral quantifier is also used in Haskell for encapsulation, we repeat the example:

$$runST :: \forall a.(\forall s.ST\ s\ a) \rightarrow a$$

This is also slightly confusing because a universal quantifier has a useful meaning when used for functions. A function can be passed and return values without knowing their type. For "$\forall s.ST\ s\ a$", the body of *runST* can choose s, but this is a rather useless choice because no value can be created by the caller of *runST* that still allows the body of *runST* to choose the type s. The effect therefore is that the type of s is hidden. In EH we would encode this directly:

$$runST :: \forall a.(\exists s.ST\ s\ a) \rightarrow a$$

We summarize the use of quantifiers in EH:

- A universal quantifier $\forall$ is used for functions which (polymorphically) accept an unknown type and return a value of this same unknown type.
- An existential quantifier $\exists$ is used for values for which type information has been forgotten.

In Chapter 10 we will exploit this use further.

8.1 Motivating examples

Existential types are a necessary ingredient for encapsulation, abstract data types, and modules, because existential types allow us to hide type information. The following example uses a minimal abstract data type "$\exists a.(a, a \rightarrow Int)$": a value tupled with an observer function for that value. Note that for all practical purposes this type is isomorphic to *Int*.

Example 8.1

```
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'
```

Value *xy* holds an "$\exists a.(a, a \rightarrow Int)$". An "$(Int, Int \rightarrow Int)$" has been bound to in *xy*, but the signature for *xy* only reflects that the value and function argument have the same type,

so we can apply this function to the value (via *ixy*). Value *pq* is similarly typed, but the assignment of a value is erroneous.

Opening an existential type When we create a value by an existential type, we forget (part of) its type and represent this with an existentially quantified type variable. We call this the *closing* of a type, as opposed to the *opening* of an existential type. The use of an existential type requires a concrete type instead of a (existentially quantified) type variable. The creation of such a concrete type is called *opening*. Ideally, opening would give us back the original type, but this requires some form of dependent types. In EH, we merely create a fresh type constant. For example, the type of *xy* from Example 8.1 is the following (instead of $\exists a.(a, a \to Int)$):

$xy :: (C\_0\_2\_0, C\_0\_2\_0 \to Int)$

The opening of an existential type is often tied up to special syntax, usually a variation of a **let**-expression. In EH, the opening is associated with the binding of a type to a value identifier. This (design decision) follows the intuition that a value is a concrete object with a concrete type.

Opening an existential type by binding also means that the following example does not type check[1]:

```
let ord                :: Char → Int
    id                 :: ∀ a.a → a
    f                  :: Int → ∃ a.(a, a → Int)
    f                  = λx → case x of
                                 2 → (x   , id  )
                                 3 → ('a', ord)
    ((xx,fx),(yy,fy))  = (f 2,f 3)
    x                  = fx yy
in  x
```

Function *f* returns $(2, id)$ when passed 2 and $(\texttt{'a'}, ord)$ otherwise. EH creates the following bindings; the creation of type constants guarantees that *fx* cannot be applied to *yy*:

$fy :: C\_35\_1\_0 \to Int$
$yy :: C\_35\_1\_0$
$fx :: C\_31\_1\_0 \to Int$
$xx :: C\_31\_1\_0$

The opening of an existential for a value binding is only done for a top-level existential quantifier. If an existential quantifier is nested inside a composite type, then the opening is not done:

```
let v1 :: (∃ a.a, ∃ b.b)
    v1 = (2, 'x')
```

[1] Case expressions are introduced together with data types, not included in this thesis.

$v_2 = v_1$
$(a, b) = v_1$
$(c, d) = v_1$
in v_2

The opening is delayed until the binding of v_1's components:

$v_1 :: (\exists\, a.a, \exists\, b.b)$
$v_2 :: (\exists\, a.a, \exists\, b.b)$
$a :: C\_1\_2\_0$
$b :: C\_1\_3\_0$
$c :: C\_1\_4\_0$
$d :: C\_1\_5\_0$

These types are pessimistic. We know (for example) that a and c refer to the same value.

EH is also pessimistic when an value with an existential type is passed through a function. For example, the following extends our simplistic abstract data type with an additional observer function:

let $chr :: Int \to Char$
$f :: (\exists\, a.(a, a \to Int)) \to \exists\, a.(a, a \to Int, a \to Char)$
$f = \lambda(v, i) \to (v, i, \lambda x \to chr\ (i\ x))$
$x = (3, \lambda x \to x)$
$y = f\ x$
in y

We do not preserve type equality through f; additional information about f's implementation would be required to say something about this.

8.2 Design overview

Relative to the type language for the previous EH version, the type language has to be extended with existential quantification, which is similar to universal quantification, and type constants c:

$\sigma ::= Int \mid Char \mid c$
$\mid\ (\sigma, ..., \sigma) \mid \sigma \to \sigma$
$\mid\ v \mid f$
$\mid\ Q\ v.\sigma, Q \in \{\forall, \exists\}$

Universal and existential types are each at their end of an extreme: A $\forall a.a$ can be instantiated to any desired type, whereas a $\exists a.a$ can be obtained from any type by forgetting (its type). In terms of the type lattice (Fig. 7.1, page 105) induced by $\leqslant$, $\forall a.a$ represents the bottom $\bot$, and $\exists a.a$ represents the top $\top$.

$$\boxed{o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{\sigma_i \equiv C_\alpha \sigma_1,\ C_\alpha \equiv \overline{\alpha \mapsto c},\ \overline{c} \text{ fresh} \quad o \vdash^{\leqslant} \sigma_i \leqslant \sigma_2 : \sigma \rightsquigarrow C}{o \vdash^{\leqslant} \exists \overline{\alpha}.\sigma_1 \leqslant \sigma_2 : \sigma \rightsquigarrow C}\ \text{M.EXISTS.L}_{I1}$$

$$\frac{\sigma_i \equiv C_\alpha \sigma_2,\ C_\alpha \equiv \overline{\alpha \mapsto v},\ \overline{v} \text{ fresh} \quad o \vdash^{\leqslant} \sigma_1 \leqslant \sigma_i : \sigma \rightsquigarrow C_f \quad C \equiv C_f \backslash_{\overline{v}}^{dom}}{o \vdash^{\leqslant} \sigma_1 \leqslant \exists \overline{\alpha}.\sigma_2 : C\ (\exists \overline{\alpha}.\sigma_2) \rightsquigarrow C}\ \text{M.EXISTS.R}_{I1}$$

Figure 8.1.: Type matching (related to ∃) (I1)

8.3 Type matching

Universal and existential types are dual when used in type matching. For

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

we can freely choose a, whereas for

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

we can not: in case of universal type, a is chosen by the context, whereas a is chosen by the creator of the existential type. a is chosen by the context of the expected universal type. In both case we emulate this "choice from outside" by instantiating a to a fixed type variable during type matching (Fig. 6.2, page 93).

The type matching rules (Fig. 8.1) for existential types therefore resemble the rules for universal types; they differ in the instantiation with (fixed) type variables.

Type matching required for quantifier propagation requires additional rules for the meet and join of two types. The forgetting, respectively propagation, of found constraints is swapped (Section 8.2); this is a consequence of the dualistic relationship between universal and existential types (and meet and join).

The effect of this duality can be seen in the example type lattice (Fig. 7.1, page 105), and in the following example:

$$\begin{aligned} &\textbf{let } g_1 :: (\exists\ a.(a, a \to Int)) \to Int \\ &\quad\ g_2 :: (Int, Int \to Int) \to Int \\ &\quad\ f\ \ = \lambda h \to \textbf{let } x_1 = g_1\ h \end{aligned}$$

$$\boxed{o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{\begin{array}{c}\overline{v_{\rightleftharpoons}} \vdash^{\rightleftharpoons elim} \sigma_m : \sigma \rightsquigarrow \_; C_e \\ \sigma_i \equiv C_\alpha \sigma_1,\ C_\alpha \equiv \overline{\alpha \mapsto (v_{\rightleftharpoons} \rightleftharpoons \square)},\ \overline{v_{\rightleftharpoons}} \text{ fresh} \\ o \vdash^{\triangle} \sigma_i \triangle \sigma_2 : \sigma_m \rightsquigarrow C_m\end{array}}{o \vdash^{\triangle} \exists\overline{\alpha}.\sigma_1 \triangle \sigma_2 : \exists\overline{v_{\rightleftharpoons}}.C_e\sigma \rightsquigarrow C_e C_m}\ \text{M.EXISTS.L2}_{I2}$$

$$\frac{\begin{array}{c}\overline{v_{\rightleftharpoons}} \vdash^{\rightleftharpoons elim} \sigma_m : \sigma \rightsquigarrow \_; C_e \\ \sigma_i \equiv C_\alpha \sigma_1,\ C_\alpha \equiv \overline{\alpha \mapsto (v_{\rightleftharpoons} \rightleftharpoons \square)},\ \overline{v_{\rightleftharpoons}} \text{ fresh} \\ o \vdash^{\triangledown} \sigma_i \triangledown \sigma_2 : \sigma_m \rightsquigarrow C_m\end{array}}{o \vdash^{\triangledown} \exists\overline{\alpha}.\sigma_1 \triangledown \sigma_2 : \exists\overline{v_{\rightleftharpoons}}.\sigma \rightsquigarrow C_e C_m}\ \text{M.EXISTS.L3}_{I2}$$

Figure 8.2.: Type meet/join (∃ specific) (I2)

```
        x2 = g2 h
    in  3
in  3
```

h is expected to be used as "$\exists a.(a, a \to Int)$" and as "$(Int, Int \to Int)$". The most general of these two is "$(Int, Int \to Int)$", reflected by the following signature for f:

$f :: (Int, Int \to Int) \to Int$

During quantifier propagation we find for h the following type alternatives:

$h :: v\_13\_0$
$v\_13\_0 \mapsto v\_13\_0\ [\exists\ a.(a, a \to Int) :: \mathbb{H}_s\ /\ \mathbb{N}_r, (Int, Int \to Int) :: \mathbb{H}_s\ /\ \mathbb{N}_r]$

From this, we compute "$v\_13\_0 \mapsto (Int, Int \to Int)$".

Again, a contravariant position requires us to compute the least general type (instead of the most general):

```
let g1 :: ((∃ a.(a, a → Int)) → Int) → Int
    g2 :: ((Int, Int → Int) → Int) → Int
    id :: a → a
    ci :: Char → Int
    f  = λh → let x1 = g1 h
                  x2 = g2 h
                  h1 = (3, id)
                  h2 = ('x', ci)
                  y1 = h h1
```

$$\boxed{o; \Gamma^k; C^k; \sigma^k \vdash^p p : \sigma; \Gamma \rightsquigarrow C; \sigma_{pf}}$$

$$\frac{\begin{array}{c}\sigma_v \equiv \sigma^k, \quad \sigma_v \not\equiv \square \\ \sigma_p \equiv C_i \sigma', \ \exists \overline{v}.\sigma' \equiv \sigma_v, \ C_i \equiv \overline{[v \mapsto f_i]}, \ \overline{f_i} \text{ fresh} \\ \Gamma_i \equiv [i \mapsto \sigma_p]\end{array}}{o; \Gamma; C^k; \sigma^k \vdash^p i : \sigma_p; \Gamma_i, \Gamma \rightsquigarrow C^k; \square} \text{ P.VAR}_{I1}$$

$$\frac{\begin{array}{c}\sigma_v \equiv \sigma^k, \quad \sigma_v \not\equiv \square \\ \sigma_p \equiv C_i \sigma', \ \exists \overline{v}.\sigma' \equiv \sigma_v, \ C_i \equiv \overline{[v \mapsto f_i]}, \ \overline{f_i} \text{ fresh} \\ \Gamma_i \equiv [i \mapsto \sigma_p] \\ o; \Gamma; C^k; \sigma_p \vdash^p p : \_; \Gamma_p \rightsquigarrow C_p; \_\end{array}}{o; \Gamma; C^k; \sigma^k \vdash^p i \ @p : C_p \sigma_p; \Gamma_i, \Gamma_p \rightsquigarrow C_p; \square} \text{ P.VARAS}_{I1}$$

Figure 8.3.: Pattern expression type rules (I1)

```
        y2 = h h2
    in  3
in  3
```

Functions g_1 and g_2 provide the context in which h will be used, that is, g_1 only knows h's argument will be an existential, g_2 knows h's argument is "$(Int, Int \rightarrow Int)$". h can only make the least of the assumptions both g_1 and g_2 offer, so the following signature is inferred for f:

$f :: ((\exists a.(a, a \rightarrow Int)) \rightarrow Int) \rightarrow Int$

8.4 Impredicativity inference and type inference

Type matching takes care of most of the implementation of existential types. We only need to ensure the opening of an existential type when bound to an identifier:

- Inside patterns, when an expected type is bound to an identifier (Fig. 8.3).
- In a let expression, for explicitly introduced type signatures, and for inferred type signatures (Fig. 7.16, page 119).

8.5 Related work, discussion

By opening an existential type when bound to a value, we deviate from most treatments of existential types [73, 78, 63], which leave existential types closed, to be opened by special language constructs when the need arises. We can see the following benefits and drawbacks of both approaches, in which the scope of the identity of the hidden type plays a crucial role:

- Opening an existential type by need creates a problem with the following example, using some fantasy syntax for opening:

  $$\begin{aligned}&\textbf{let } v \;\; :: \exists a.(a, a \to Int)\\ &\quad fst :: \forall a.\forall b.(a, b) \to a\\ &\quad v_1 = \textbf{open } v' = v \textbf{ in } fst\ v'\\ &\quad v_2 = \textbf{open } v' = v \textbf{ in } fst\ v'\\ &\textbf{in } \;\; ...\end{aligned}$$

  The field access to v opens v twice. The consequence is that v_1 and v_2 have different types, because each opening introduces a new type.

  A solution to this problem is to treat field access (usually denoted by a dot notation) in a special way by using the same type for the opening of the same value. Laufer (et.al.) [63] observe that existential types, in practical settings, have to be opened for a large scope, losing some of the benefits of abstraction.

  By opening an existential type once when bound to a value identifier, we partially solve this problem. We achieve encapsulation, avoid the clutter of opening, but only do so for toplevel existential quantification. Existentials which are nested in a composite type only will be opened when bound to an identifier, so in order to preserve type identity across multiple nested selections, we would have to open all fields of a composite value in this way.

- In our solution we open a type by creating fresh type constants for the existentially quantified type variables. We allow these constants to escape to a larger scope. This is not a problem because only functions accepting such a constant can do something with it that is particular to the type constant. However, as freshness of a type constant is guaranteed by means of uniqueness, we must also ensure uniqueness in the context of separatedly compiled modules; as we do not discuss modules in this thesis we merely point this out.
- If the need arises to (again) forget a type constant, this can be done by an explicit type annotation.

Existential types are a necessary ingredient for abstract data types [78]. However, using existential types to construct a module mechanism requires additional mechanisms for preserving the type identity of modules [66, 67, 70], easily leading to forms of dependent typing [73].

9 Making implicit parameters explicit

Note to the reader: this chapter is a slightly adapted version of a paper. It can be read independently of previous chapters, but also is not yet updated to use Ruler. The original version did include partial type signatures; this has been moved to Chapter 10.

In almost all languages all arguments to functions have to be given explicitly. The Haskell class system however is an exception: functions can have class predicates as part of their type signature, and dictionaries are implicitly constructed and implicitly passed for such predicates, thus relieving the programmer from a lot of clerical work and removing clutter from the program text. Unfortunately Haskell maintains a very strict boundary between the implicit and the explicit world; if the implicit mechanisms fail to construct the hidden dictionaries there is no way the programmer can provide help, nor is it possible to override the choices made by the implicit mechanisms. In this paper we describe, in the context of Haskell, a mechanism that allows a programmer to explicitly construct implicit arguments. This extension blends well with existing resolution mechanisms, since it only overrides the default behavior. Finally we show how the system can easily be extended to deal with higher-order predicates, thus enabling the elegant formulation of some forms of generic programming.

9.1 Introduction

The Haskell class system, originally introduced by both Wadler and Blott [114] and Kaes [50], offers a powerful abstraction mechanism for dealing with overloading (ad-hoc polymorphism). The basic idea is to restrict the polymorphism of a parameter by specifying that some predicates have to be satisfied when the function is called:

$f :: Eq\ a \Rightarrow a \to a \to Int$
$f = \lambda \quad x \quad y \to \mathbf{if}\ x == y\ \mathbf{then}\ 3\ \mathbf{else}\ 4$

In this example the type signature for f specifies that values of any type a can be passed as arguments, as long as the predicate $Eq\ a$ is satisfied. Such predicates are introduced by *class declaration*s, as in the following version of Haskell's *Eq* class declaration:

```
class Eq a where
  (==) :: a → a → Bool
```

The presence of such a class predicate in a type requires the availability of a collection of functions and values which can only be used on a type a for which the class predicate holds. For brevity, the given definition for class *Eq* omits the declaration for `/=`. A class declaration alone is not sufficient: *instance declarations* specify for which types the predicate actually can be satisfied, simultaneously providing an implementation for the functions and values as a witness for this:

```
instance Eq Int where
  x == y = primEqInt x y
instance Eq Char where
  x == y = primEqChar x y
```

Here the equality functions for *Int* and *Char* are implemented by the primitives *primEqInt* and *primEqChar*. The compiler turns such instance declarations into records (dictionaries) containing the functions as fields, and thus an explicit version of this internal machinery reads:

```
data EqD a = EqD {eqEqD :: a → a → Bool}   -- class Eq
eqDInt     = EqD primEqInt                 -- Eq Int
eqDChar    = EqD primEqChar                -- Eq Char
```

Inside a function the elements of the predicate's dictionaries are available, as if they were defined as top-level variables. This is accomplished by implicitly passing a dictionary for each predicate occurring in the type of the function. So the actual implementation of f (apart from all kind of optimisations) is:

```
f ::   EqD a → a → a → Int
f = λ dEq      x     y → if (eqEqD dEq) x y then 3 else 4
```

At the call site of the function f the dictionary that corresponds to the actual type of the polymorphic argument must be passed. Thus the expression $f\ 3\ 4$ can be seen as an abbreviation for the semantically more complete $f\ eqDInt\ 3\ 4$.

Motivating examples The translation from $f\ 3\ 4$ to $f\ eqDInt\ 3\ 4$ is done implicitly; a programmer has little or no control over the passing of dictionaries. This becomes problematic as soon as a programmer desires to express something which the language definition cannot infer automatically. For example, we may we want to call f with an alternate instance for *Eq Int*, which implements a different equality on integers:

```
instance Eq Int where
  x == y = primEqInt (x `mod` 2) (y `mod` 2)
```

Unfortunately this extra instance declaration would introduce an ambiguity, and is thus forbidden by the language definition; the instances are said to overlap. However, a programmer could resolve the issue if he was only able to explicitly specify which of these two possible instances should be passed to f.

As a second example we briefly discuss the use Kiselyov and Chan [57] make of the type class system to configure programs. In their modular arithmetic example integer arithmetic is configured by a modulus: all integer arithmetic is done modulo this modulus. The modulus is implemented by a class function *modulus*:

```
class Modular s a | s → a where modulus :: s → a

newtype M s a = M a

normalize :: (Modular s a, Integral a) ⇒ a → M s a
normalize a :: M s a = M (mod a (modulus (⊥ :: s)))

instance (Modular s a, Integral a) ⇒ Num (M s a) where
  M a + M b = normalize (a + b)
  ...    -- remaining definitions omitted
```

The problem now is to create for a value m of type a an instance of *Modular s a* for which *modulus* returns this m. Some ingenious type hackery is involved where phantom type s (evidenced by $\bot$'s) uniquely represents the value m, and as such is used as an index into the available instances for *Modular s a*. This is packaged in the following function which constructs both the type s and the corresponding dictionary (for which *modulus* returns m) for use by k:

```
withModulus :: a → (∀s.Modular s a ⇒ s → w) →  w
withModulus   m    k                          =  ...
```

They point out that this could have been done more directly if local type class instances would have been available:

```
data Label
withModulus :: a → (∀s.Modular s a ⇒ s → w) → w
withModulus   m    k
  = let instance Modular Label a where modulus _ = m
    in  k (⊥ :: Label)
```

The use of explicit parameter passing for an implicit argument proposed by us in this chapter would have even further simplified the example, as we can avoid the phantom type *Label* and related type hackery altogether and instead create and pass the instance directly.

As we may infer from the above the Haskell class system, which was originally only introduced to describe simple overloading, has become almost a programming language of its own, used (and abused as some may claim) for unforeseen purposes.

Haskell's point of view Haskell's class system has turned out to be theoretically sound and complete [44], although some language constructs prevent Haskell from having prin-

cipal types [27]. The class system is flexible enough to incorporate many useful extensions [43, 47]. Its role in Haskell has been described in terms of an implementation [46] as well as its semantics [33, 26]. Many language constructs do their work automatically and implicitly, to the point of excluding the programmer from exercising influence. Here we feel there is room for improvement, in particular in dealing with implicit parameters.

The compiler is fully in control of which dictionary to pass for a predicate, determined as part of the resolution of overloading. This behavior is the result of the combination of the following list of design choices:

- A class definition introduces a record type (for the dictionary) associated with a predicate over type variables.
- Instance definitions describe how to construct a value for the record type for the class predicate specialized for a specific type (or combination of types in the case of multiparameter type classes).
- The type of a function specifies the predicates for which dictionaries have to be passed at the call site of the function.
- Which dictionary is to be passed at the call site of a function is determined by:
  - required dictionaries at the call site of a function; this is determined by the predicates in the instantiated type of the called function.
  - the available dictionaries introduced by instance definitions.

  Internally the compiler uses a predicate proving machinery and heuristics [48, 85, 26] to compute the proper dictionaries.
- Which dictionaries are to be passed is fully fixed by the language definition.
- The language definition uses a statically determined set of dictionaries introduced by instance definitions and a fixed algorithm for determining which dictionaries are to be passed.

The result of this is both a blessing and a curse. A blessing because it silently solves a problem (i.e. overloading), a curse because as a programmer we cannot easily override the choices made in the design of the language (i.e. via Haskell's default mechanism), and worse, we can in no way assist the compiler if no unique solution according to the language semantics exists. For example, overlapping instances occur when more than one choice for a dictionary can be made. Smarter, more elaborate versions of the decision making algorithms can and do help [36], but in the end it is only the programmer who can fully express his intentions. The system at best can only make a guess.

The issue central to this paper is that Haskell demands from a program that all choices about which dictionaries to pass can be made automatically and uniquely, whereas we also want to be able to specify this ourselves explicitly. If the choice made (by Haskell) does not correspond to the intention of the programmer, the only solution is to convert

all involved implicit arguments into explicit ones, thus necessitating changes all over the program. Especially for (shared) libraries this may not always be feasible.

Our contribution Our approach takes explicitness as a design starting point, as opposed to the described implicitness featured by the Haskell language definition. To make the distinction between our and Haskell's approach clear in the remainder of this chapter, we call our explicit language and its implementation Explicit Haskell (EH) whereas we refer to Haskell language and its implementations by just Haskell.

- In principle, all aspects of an EH program can be explicitly specified, in particular the types of functions, types of other values, and the manipulation of dictionaries, without making use of or referring to the class system.
- The programmer is allowed to omit explicit specification of some program aspects; EH then does its best to infer the missing information.

Our approach allows the programmer and the EH system to jointly construct the completely explicit version of a program, whereas an implicit approach inhibits all explicit programs which the type inferencer cannot infer but would otherwise be valid. If the type inferencer cannot infer what a programmer expects it to infer, then the programmer can provide the required information. In this sense we get the best of two worlds: the simplicity of systems like system F [30, 96] and Haskell's ease of programming.

In this chapter explicitness takes the following form:

- Dictionaries introduced by instance definitions can be named; the dictionary can be accessed by name as a record value.
- The set of class instances and associated dictionaries to be used by the proof machinery can be used as normal values, and normal (record) values can be used as dictionaries for predicates as well.
- The automatic choice for a dictionary at the call site of a function can be overruled.
- Types can be composed of the usual base types, predicates and quantifiers (both universal and existential) in arbitrary combinations.

We will focus on all but the last items of the above list: the explicit passing of values for implicit parameters. Although explicit typing forms the foundation on which we build [22], we discuss it only as much as is required.

Related to programming languages in general, our contribution, though inspired by and executed in the context of Haskell, offers language designers a mechanism for more sophisticated control over parameter passing, by allowing a mixture of explicit and implicit parameter passing.

Outline of this chapter In this chapter we focus on the exploration of explicitly specified implicit parameters, to be presented in the context of EH, a Haskell variant [19, 21, 22] in which all features described in this chapter have been implemented. In Section 9.2 we start with preliminaries required for understanding the remainder of this chapter. In Section 9.3 we present examples of what we can express in EH. The use of partial type signatures and their interaction with predicates is demonstrated in Chapter 10. In Section 9.4 we give some insight in our implementation, highlighting the distinguishing aspects as compared to traditional implementations. In Section 9.5 we discuss some remaining design issues and related work. We conclude in Section 9.6.

Limitations of this chapter Our work is made possible by using some of the features already available in EH, for example higher ranked types and the combination of type checking and inferencing. We feel that our realistic setting contributes to a discussion surrounding the issues of combining explicitly specified and inferred program aspects [110] as it offers a starting point for practical experience. For reasons of space we have made the following choices:

- We present examples and part of our implementation, so the reader gets an impression of what can be done and how it ties in with other parts of the implementation [19].
- We do *not* present all the context required to make our examples work. This context can be found elsewhere [22, 21].
- We focus on prototypical implementation before considering formally proving properties of EH.
- We do not prove properties like soundness, completeness and principality. In Section 9.5 we will address the reasons why we have chosen not to deal with those issues here.
- Our type rules therefore describe an algorithm which has been implemented using an attribute grammer system [41, 9]. An attribute grammar provides better separation of implementation aspects whereas type rules are more concise in their presentation; we therefore have chosen to incorporate typing rules in this chapter. We describe the similarities between typing rules and their attribute grammar counterpart in Chapter 11.

9.2 Preliminaries

Intended as a platform for both education and research, EH offers advanced features like higher ranked types, existential types, partial type signatures and records. Syntactic sugar has been kept to a minimum in order to ease experimentation with and understanding of

Values (expressions, terms):	
$e ::= int \mid char$	literals
$\mid i$	program variable
$\mid e\ e$	application
$\mid$ **let** $\overline{d}$ **in** e	local definitions
$\mid \lambda i \rightarrow e$	abstraction
$\mid (l = e, ...)$	record
$\mid (e \mid l := e, ...)$	record update
$\mid e.l$	record selection
$\mid e\ (!e \leftsquigarrow \pi!)$	*explicit implicit application*
$\mid \lambda(!i \leftsquigarrow \pi!) \rightarrow e$	*explicit implicit abstraction*
Declarations of bindings:	
$d ::= i :: \sigma$	value type signature
$\mid i = e$	value binding
$\mid$ **data** $\overline{\sigma} = \overline{I\ \overline{\sigma}}$	data type
$\mid$ **class** $\overline{\pi} \Rightarrow \pi$ **where** $\overline{d}$	class
$\mid$ **instance** $\overline{\pi} \Rightarrow \pi$ **where** $\overline{d}$	introduced instance
$\mid$ **instance** $i \leftsquigarrow \overline{\pi} \Rightarrow \pi$ **where** $\overline{d}$	*named introduced instance*
$\mid$ **instance** $i :: \overline{\pi} \Rightarrow \pi$ **where** $\overline{d}$	*named instance*
$\mid$ **instance** $e \leftsquigarrow \pi$	*value introduced instance*
Identifiers:	
$\iota ::= i$	lowercase: (type) variables
$\mid I$	uppercase: (type) constructors
$\mid l$	field labels

Figure 9.1.: EH terms (emphasized ones explained throughout the text)

the implementation; other mechanisms like syntax macro's [10] provide the means for including additional syntax into the language without having to change the compiler.

Fig. 9.1 and Fig. 9.2 show the terms and types featured in EH. Throughout this chapter all language constructs will be gradually introduced and explained. In general, we designed EH to be as upwards compatible as possible with Haskell. We point out some aspects required for understanding the discussion in the next section:

- An EH program is single stand alone term. All types required in subsequent examples are either silently assumed to be similar to Haskell or will be introduced explicitly.

Types:		
σ	$::= Int \mid Char$	literals
	$\mid v$	variable
	$\mid \sigma \rightarrow \sigma$	abstraction
	$\mid \sigma\ \sigma$	type application
	$\mid \forall v.\sigma$	universally quantified type
	$\mid \pi \Rightarrow \sigma$	implicit abstraction
	$\mid (l :: \sigma, ...)$	record
Types for quantifier propagation:		
σ	$::= ...$	
	$\mid \mathfrak{o}$	type alternatives
$\mathfrak{o}$	$::= v\ [\overline{\varphi}]$	type alternatives

Figure 9.2.: EH types

- All bindings in a **let** expression are analysed together; in Haskell this constitutes a binding group.
- We represent dictionaries by records. Records are denoted as parenthesized comma separated sequences of field definitions. Extensions and updates to a record e are denoted as $(e \mid ...)$, with e in front of the vertical bar '|'. The notation and semantics is based on existing work on extensible records [29, 49]. Record extension and updates are useful for re-using values from a record.

The language of types as used in this chapter is shown in Fig. 9.2. A programmer can specify types using the same syntax. We mention this because often types are categorized based on the presence of (universal) quantifiers and predicates [38, 87]. We however allow quantifiers at higher ranked positions in our types and predicates as well. For example, the following is a valid type expression in EH:

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

Existential types are part of EH, but are omitted here because we will not use them in this chapter. Quantification has lower priority than the other composite types, so in a type expression without parentheses the scope of the quantifier extends to the far right of the type expression.

We make no attempt to infer higher ranked types [53, 54, 40]; instead we propagate explicitly specified types as good as possible to wherever this information is needed. Our strategies here are elaborated in earlier chapters of this thesis.

9.3 Implicit parameters

In this section we give EH example programs, demonstrating most of the features related to implicit parameters. After pointing out these features we continue with exploring the finer details.

Basic explicit implicit parameters Our first demonstration EH program contains the definition of the standard Haskell function *nub* which removes duplicate elements from a list. A definition for *List* has been included; definitions for *Bool*, *filter* and *not* are omitted. In this example the class *Eq* also contains *ne* which we will omit in later examples. Notice that a separate *nubBy*, which is in the Haskell libraries enabling the parameterisation of *nub* with an equality test, is no longer needed:

```
let data List a = Nil | Cons a (List a)
    class Eq a where
      eq :: a → a → Bool
      ne :: a → a → Bool
    instance dEqInt ⇜ Eq Int where     -- (1)
      eq = primEqInt
      ne = λx y → not (eq x y)
    nub :: ∀ a.Eq a ⇒ List a → List a
    nub = λxx → case xx of
                  Nil         → Nil
                  Cons x xs → Cons x (nub (filter (ne x) xs))
    eqMod2 :: Int → Int → Bool
    eqMod2 = λx y → eq (mod x 2) (mod y 2)
    n1 = nub (!dEqInt ⇜ Eq Int!)       -- (2)
             (Cons 3 (Cons 3 (Cons 4 Nil)))
    n2 = nub (!(eq = eqMod2            -- (2)
                ,ne = λx y → not (eqMod2 x y)
                ) ⇜ Eq Int
              !)
             (Cons 3 (Cons 3 (Cons 4 Nil)))
in  ...
```

This example demonstrates the use of the two basic ingredients required for being explicit in the use of implicit parameters (the list items correspond to the commented number in the example):

1. The notation ⇜ binds an identifier, here *dEqInt*, to the dictionary representing the instance. The record *dEqInt* from now on is available as a normal value.
2. Explicitly passing a parameter is syntactically denoted by an expression between (! and !). The predicate after the ⇜ explicitly states the predicate for which the ex-

pression is an instance dictionary (or *evidence*). The dictionary expression for n_1 is formed by using *dEqInt*, for n_2 a new record is created: a dictionary can also be created by updating an already existing one like *dEqInt*; in our discussion (Section 9.5) we will come back to this.

This example demonstrates our view on implicit parameters:

- Program values live in two, possibly overlapping, worlds, *explicit* and *implicit*.
- Parameters are either passed explicitly, by the juxtapositioning of explicit function and argument expressions, or passed implicitly (invisible in the program text) to an explicit function value. In the implicit case the language definition determines which value to take from the implicit world.
- Switching between the explicit and implicit world is accomplished by means of additional notation. We go from implicit to explicit by instance definitions with the naming extension, and in the reverse direction by means of the (! !) construct.

The *Modular* motivating example now can be simplified to (merging our notation into Haskell):

```
class Modular a where modulus :: a
newtype M a = M a
normalize :: (Modular a, Integral a) ⇒ a → M a
normalize a = M (mod a modulus)
instance (Modular a, Integral a) ⇒ Num (M a) where
  M a + M b = normalize (a + b)
  ...    -- remaining definitions omitted
withModulus :: a →     (Modular a ⇒ w) → w
withModulus    (m :: a) k
   = k (!(modulus = m) ↜ Modular a!)
```

Higher order predicates We also allow the use of higher order predicates. Higher order predicates are already available in the form of instance declarations. For example, the following program fragment defines the instance for *Eq* (*List a*) (the code for the body of *eq* has been omitted):

```
instance dEqList ↜ Eq a ⇒ Eq (List a) where
  eq = λx y → ...
```

The important observation is that in order to be able to construct the dictionary for *Eq* (*List a*) we need a dictionary for *Eq a*. This corresponds to interpreting *Eq a* ⇒ *Eq* (*List a*) as stating that *Eq* (*List a*) can be proven from *Eq a*. The implementation for this instance is a function taking the dictionary for *Eq a* and constructing the dictionary for *Eq* (*List a*). Such a function is called a *dictionary transformer*.

We allow higher order predicates to be passed as implicit arguments, provided the need for this is specified explicitly. For example, in f we can abstract from the dictionary transformer for *Eq* (*List a*), which can then be passed either implicitly or explicitly:

$$f :: (\forall\ a.Eq\ a \Rightarrow Eq\ (List\ a)) \Rightarrow Int \rightarrow List\ Int \rightarrow Bool$$
$$f = \lambda p\ q \rightarrow eq\ (Cons\ p\ Nil)\ q$$

The effect is that the dictionary for *Eq* (*List Int*) will be computed inside f as part of its body, using the passed dictionary transformer and a more globally available dictionary for *Eq Int*. Without the use of this construct the dictionary would be computed only once globally by:

let *dEqListInt* = *dEqList dEqInt*

The need for higher order predicates really becomes apparent when genericity is implemented using the class system. The following example is taken from Hinze [39]:

```
let data Bit          = Zero | One
    data GRose f a = GBranch a (f (GRose f a))
in  let class Binary a where
            showBin :: a → List Bit
        instance dBI ↚ Binary Int where
            showBin  = ...
        instance dBL ↚ Binary a ⇒ Binary (List a) where
            showBin  = ...
        instance dBG ↚ (Binary a, (∀ b.Binary b ⇒ Binary (f b)))
                              ⇒ Binary (GRose f a) where
            showBin  = λ(GBranch x ts)
                              → showBin x ++ showBin ts
in  let v1 = showBin (GBranch 3 Nil)
    in  v1
```

The explicit variant of the computation for v_1 using the explicit parameter passing mechanism reads:

$$v_1 = showBin\ (!dBG\ dBI\ dBL \leftsquigarrow Binary\ (GRose\ List\ Int)!)$$
$$(GBranch\ 3\ Nil)$$

The value for *dBG* is defined by the following translation to an explicit variant using records; the identifier *showBin* has been replaced by *sb*, *List* by *L* and *Bit* by *B* in order to keep the programfragment compact:

```
sb   = λd → d.sb
dBG ::     (sb :: a → L B)
        → (∀ b.(sb :: b → L B) → (sb :: f b → L B))
        → (sb :: GRose f a → L B)
dBG = λdBa dBf → d
    where d = (sb = λ(GBranch x ts)
```

$$\rightarrow sb\ dBa\ x \mathbin{+\!\!+} sb\ (dBf\ d)\ ts$$
$$)$$

Hinze's solution essentially relies on the use of the higher order predicate $Binary\ b \Rightarrow Binary\ (f\ b)$ in the context of $Binary\ (GRose\ f\ a)$. The rationale for this particular code fragment falls outside the scope of this chapter, but the essence of its necessity lies in the definition of the *GRose* data type which uses a type constructor f to construct the type $(f\ (GRose\ f\ a))$ of the second member of *GBranch*. When constructing an instance for $Binary\ (GRose\ f\ a)$ an instance for this type is required. Type (variable) f is not fixed, so we cannot provide an instance for $Binary\ (f\ (GRose\ f\ a))$ in the context of the instance. However, given dictionary transformer $dBf \leftsquigarrow Binary\ b \Rightarrow Binary\ (f\ b)$ and the instance $d \leftsquigarrow Binary\ (GRose\ f\ a)$ under construction, we can construct the required instance: $dBf\ d$. The type of v_1 in the example instantiates to *GRose List Int*; the required dictionary for the instance $Binary\ (GRose\ List\ Int)$ can be computed from *dBI* and *dBL*.

The finer details For our discussion we take the following fragment as our starting point:

$$\begin{array}{l}\textbf{let}\ f = \lambda p\ q\ r\ s \rightarrow (eq\ p\ q, eq\ r\ s)\\ \textbf{in}\ f\ 3\ 4\ 5\ 6\end{array}$$

Haskell infers the following type for f:

$$f :: \forall\ a\ b.(Eq\ b, Eq\ a) \Rightarrow a \rightarrow a \rightarrow b \rightarrow b \rightarrow (Bool, Bool)$$

On the other hand, EH infers:

$$f :: \forall\ a.Eq\ a \Rightarrow a \rightarrow a \rightarrow \forall\ b.Eq\ b \Rightarrow b \rightarrow b \rightarrow (Bool, Bool)$$

EH not only inserts quantifiers as close as possible to the place where the quantified type variables occur, but does this for the placement of predicates in a type as well. The idea is to instantiate a quantified type variable or pass an implicit parameter corresponding to a predicate as late as possible, where later is defined as the order in which arguments are passed.

The position of a predicate in a type determines the position in a function application (of a function with that type) where a value for the corresponding implicit parameter may be passed explicitly. For example, for f in the following fragment first we may pass a dictionary for $Eq\ a$, then we must pass two normal arguments, then we may pass a dictionary, and finally we must pass two normal arguments:

$$\begin{array}{l}\textbf{let}\ f :: \forall a.Eq\ a \Rightarrow a \rightarrow a \rightarrow \forall b.Eq\ b \Rightarrow b \rightarrow b \rightarrow (Bool, Bool)\\ \quad f = \lambda p\ q\ r\ s \rightarrow (eq\ p\ q, eq\ r\ s)\\ \textbf{in}\ f \qquad\qquad\qquad\qquad\qquad\qquad 3\ 4\\ \quad\ (!(eq = eqMod2) \leftsquigarrow Eq\ Int!)\ 5\ 6\end{array}$$

The value for the first implicit parameter ($Eq\ a$) is computed automatically, the value (an explicitly constructed dictionary record) for the second ($Eq\ b$) is explicitly passed by

means of (! !). Inside these delimiters we specify both value and the predicate for which it is a witness. The notation ($!e \leftsquigarrow p!$) suggests a combination of "is of type" and "is evidence for". Here "is of type" means that the dictionary e must be of the record type introduced by the class declaration for the predicate p. The phrase "is evidence for" means that the dictionary e is used as the proof of the existence of the implicit argument to the function f.

Explicitly passing a value for an implicit parameter is optional. However, if we explicitly pass a value, all preceding implicit parameters in a consecutive sequence of implicit parameters must be passed as well. In a type expression, a consecutive sequence of implicit parameters corresponds to sequence of predicate arguments delimited by other arguments. For example, if we were to pass a value to f for $Eq\ b$ with the following type, we need to pass a value for $Eq\ a$ as well:

$$f :: \forall\ a\ b.(Eq\ a, Eq\ b) \Rightarrow a \rightarrow a \rightarrow b \rightarrow b \rightarrow (Bool, Bool)$$

We can avoid this by swapping the predicates, as in:

$$f :: \forall\ a\ b.(Eq\ b, Eq\ a) \Rightarrow a \rightarrow a \rightarrow b \rightarrow b \rightarrow (Bool, Bool)$$

For this type we can pass a value explicitly for $Eq\ b$. We may omit a parameter for $Eq\ a$ because dictionaries for the remaining predicates (if any) are automatically passed, just like Haskell.

The above types for f have to be specified explicitly. All types signatures for f are isomorphic, so we always can write wrapper functions for the different varieties.

Overlapping instances By explicitly providing a dictionary the default decision making by EH is overruled. This is useful in situations where no unique choice is possible, for example in the presence of overlapping instances:

```
let instance dEqInt1 ⇜ Eq Int where
      eq = primEqInt
    instance dEqInt2 ⇜ Eq Int where
      eq = eqMod2
    f = ...
in f (!dEqInt1 ⇜ Eq Int!) 3 4
     (!dEqInt2 ⇜ Eq Int!) 5 6
```

The two instances for *Eq Int* overlap, but we still can refer to each associated dictionary individually, because of the names *dEqInt1* and *dEqInt2* that were given to the dictionaries. Thus overlapping instances can be avoided by letting the programmer decide which dictionaries to pass to the call $f\ 3\ 4\ 5\ 6$.

Overlapping instances can also be avoided by not introducing them in the first place. However, this conflicts with our goal of allowing the programmer to use different instances at different places in a program. This problem can be overcome by excluding instances participating in the predicate proving machinery by:

```
instance dEqInt2 :: Eq Int where
  eq = λ_ _ → False
```

The naming of a dictionary by means of ⇜ actually does two things. It binds the name to the dictionary and it specifies to use this dictionary as the default instance for *Eq Int* for use in its proof process. The notation :: only binds the name but does not introduce it into proving predicates. If one at a later point wants to introduce the dictionary nevertheless, possibly overriding an earlier choice, this may done by specifying:

```
instance dEqInt2 ⇜ Eq Int
```

Local instances We allow instances to be declared locally, within the scope of other program variables. A local instance declaration shadows an instance declaration introduced at an outer level:

- If their names are equal, the innermost shadows the outermost.
- In case of having overlapping instances available during the proof of predicates arising inside the **let** expression, the innermost instance takes precedence over the outermost.

This mechanism allows the programmer to fully specify which instances are active at any point in the program text:

```
let instance dEqInt1 ⇜ Eq Int where ...
    instance dEqInt2  ::  Eq Int where ...
    g = λx y → eq x y
in  let v1 = g 3 4
        v2 = let instance dEqInt2 ⇜ Eq Int
             in  g 3 4
in  ...
```

The value for v_1 is computed with *dEqInt1* as evidence for *Eq Int*, whereas v_2 is computed with *dEqInt2* as evidence.

In our discussion we will come back to local instances.

Higher order predicates revisited As we mentioned earlier, the declaration of an instance with a context actually introduces a function taking dictionaries as arguments:

```
let instance dEqInt ⇜ Eq Int where
      eq = primEqInt
    instance dEqList ⇜ Eq a ⇒ Eq (List a) where
      eq = ...
    f :: ∀ a.Eq a ⇒ a → List a → Bool
    f = λp q → eq (Cons p Nil) q
in  f 3 (Cons 4 Nil)
```

In terms of predicates the instance declaration states that given a proof for the context *Eq a*, the predicate *Eq* (*List a*) can be proven. In terms of values this translates to a function which takes the evidence of the proof of *Eq a*, a dictionary record ($eq :: a \to a \to Bool$), to evidence for the proof of *Eq* (*List a*) [44]:

$$\begin{aligned}
&dEqInt \;\; :: (eq :: Int \to Int \to Bool)\\
&dEqList :: \forall\, a.(eq :: a \to a \to Bool)\\
&\qquad\qquad \to (eq :: List\ a \to List\ a \to Bool)\\
&eq \qquad = \lambda dEq\ x\ y \to dEq.eq\ x\ y
\end{aligned}$$

With these values, the body of f is mapped to:

$$f = \lambda dEq\_a\ p\ q \to eq\ (dEqList\ dEq\_a)\ (Cons\ p\ Nil)\ q$$

This translation can now be expressed explicitly as well; a dictionary for *Eq* (*List a*) is explicitly constructed and passed to *eq*:

$$\begin{aligned}
&f :: \forall\, a.Eq\ a \Rightarrow a \to List\ a \to Bool\\
&f = \lambda(!dEq_a \leftsquigarrow Eq\ a!)\\
&\qquad \to \lambda p\ q \to eq\ (!dEqList\ dEq_a \leftsquigarrow Eq\ (List\ a)!)\\
&\qquad\qquad (Cons\ p\ Nil)\ q
\end{aligned}$$

The type variable *a* is introduced as a lexically scoped type variable [86], available for further use in the body of f.

The notation *Eq a* ⇒ *Eq* (*List a*) in the instance declaration for *Eq* (*List a*) introduces both a predicate transformation for a predicate (from *Eq a* to *Eq* (*List a*)), to be used for proving predicates, as well as a corresponding dictionary transformer function. Such transformers can also be made explicit in the following variant:

$$\begin{aligned}
&f :: (\forall a.Eq\ a \Rightarrow Eq\ (List\ a)) \Rightarrow Int \to List\ Int \to Bool\\
&f = \lambda(!dEq_La \leftsquigarrow \forall a.Eq\ a \Rightarrow Eq\ (List\ a)!)\\
&\qquad \to \lambda p\ q \to eq\ (!dEq_La\ dEqInt \leftsquigarrow Eq\ (List\ Int)!)\\
&\qquad\qquad (Cons\ p\ Nil)\ q
\end{aligned}$$

Instead of using *dEqList* by default, an explicitly specified implicit predicate transformer, bound to *dEq_La* is used in the body of f to supply *eq* with a dictionary for *Eq* (*List Int*). This dictionary is explicitly constructed and passed to *eq*; both the construction and binding to *dEq_La* may be omitted. We must either pass a dictionary for *Eq a* ⇒ *Eq* (*List a*) to f ourselves explicitly or let it happen automatically; here in both cases *dEqList* is the only choice possible.

9.4 Implementation

We focus on the distinguishing characteristics of our implementation in the EH compiler [19, 21, 22].

The type system is given in Fig. 9.3 which describes the relationship between types in the type language in Fig. 9.2. Our σ types allow for the specification of the usual base types ($Int, Char$) and type variables (v) as well aggregrate types like normal abstraction ($\sigma \to \sigma$), implicit abstraction ($\pi \Rightarrow \sigma$), (higher ranked) universal quantification ($\forall \alpha.\sigma$), predicates (π) and their transformations ($\pi \Rightarrow \pi$). Translations ϑ represent code resulting from the transformation from implicit parameter passing to explicit parameter passing. An environment Γ binds value identifiers to types ($\iota \mapsto \sigma$). Instance declarations result in bindings of predicates to translations (dictionary evidence) paired with their type ($\pi \rightsquigarrow \vartheta : \sigma$) whereas class declarations bind a predicate to its dictionary type ($\pi \rightsquigarrow \sigma$):

$$bind = \iota \mapsto \sigma \mid \pi \rightsquigarrow \vartheta : \sigma \mid \pi \rightsquigarrow \sigma$$
$$\Gamma \quad = \overline{bind}$$

We use vector notation for any ordered collection, denoted with a horizontal bar on top. Concatenation of vectors and pattern matching on a vector is denoted by a comma ',' .

Basic typing rules Type rules in Fig. 9.3 read like this: given contextual information Γ it can be proven ($\vdash$) that term e has (:) type σ and some additional ($\rightsquigarrow$) results, which in our case is the code ϑ in which passing of all parameters has been made explicit. Later type rules will incorporate more properties; all separated by a semicolon ';'. If some property does not matter or is not used, an underscore '_' is used to indicate this. Rules are labeled with names of the form $x - variant_{version}$ in which x is a single character indicating the syntactic element, *variant* its variant and *version* a particular version of the type rule which also corresponds to a compiler version in the implementation. In this chapter only versions *Ev*, *EvK* and *I* are used, respectively addressing evidence translation, use of expected types and the handling of implicit parameters. We have only included the most relevant type rules and have omitted those dealing with the introduction of classes and instances; these are all standard [26].

The conciseness of the rules suggests that its implementation should not pose much of a problem, but the opposite is true. Unfortunately, in their current form the rules do not fully specify how to combine them in order to build a complete proof tree, and hence are not algorithmic [91]. This is especially true for the last rule E-PRED, since its use is not associated with a syntactic construct of the source language. Algorithmic variants of the rules have two pleasant properties:

- The syntax tree determines how to combine the rules.
- By distributing data over a larger set of variables a computation order becomes apparent.

The first property is taken care of by the parser, and based on the second property we can implement rules straightforwardly using an attribute grammar, mapping variables in rules to attributes. Our situation is complicated due to a combination of several factors:

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

$$\frac{}{\Gamma \overset{expr}{\vdash} int : Int \rightsquigarrow int}\ \text{E-INT}_{Ev}$$

$$\frac{(\iota \mapsto \sigma_\iota) \in \Gamma}{\Gamma \overset{expr}{\vdash} \iota : \rightsquigarrow \iota}\ \text{E-ID}_{Ev}$$

$$\frac{\Gamma \overset{expr}{\vdash} e_2 : \sigma_a \rightsquigarrow \vartheta_2 \qquad \Gamma \overset{expr}{\vdash} e_1 : \sigma_a \to \sigma \rightsquigarrow \vartheta_1}{\Gamma \overset{expr}{\vdash} e_1\ e_2 : \sigma \rightsquigarrow \vartheta_1\ \vartheta_2}\ \text{E-APP}_{Ev}$$

$$\frac{i \mapsto \sigma_i, \Gamma \overset{expr}{\vdash} e : \sigma_e \rightsquigarrow \vartheta_e}{\Gamma \overset{expr}{\vdash} \lambda i \to e : \sigma_i \to \sigma_e \rightsquigarrow \lambda i \to \vartheta_e}\ \text{E-LAM}_{Ev}$$

$$\frac{i \mapsto \sigma_i, \Gamma \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta_e \qquad i \mapsto \sigma_i, \Gamma \overset{expr}{\vdash} e_i : \sigma_i \rightsquigarrow \vartheta_i}{\Gamma \overset{expr}{\vdash} \textbf{let}\ i :: \sigma_i; i = e_i\ \textbf{in}\ e : \sigma \rightsquigarrow \textbf{let}\ i = \vartheta_i\ \textbf{in}\ \vartheta_e}\ \text{E-LET-TYSIG}_{Ev}$$

$$\frac{\Gamma \overset{pred}{\vdash} \pi \rightsquigarrow \vartheta_\pi : \sigma_\pi \qquad \Gamma \overset{expr}{\vdash} e : \pi \Rightarrow \sigma \rightsquigarrow \vartheta_e}{\Gamma \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta_e \vartheta_\pi}\ \text{E-PRED}_{Ev}$$

Figure 9.3.: Type rules for expressions

- The structure of the source language cannot be used to determine where rule E-PRED should be applied: the term e in the premise and the conclusion are the same. Furthermore, the predicate π is not mentioned in the conclusion so discovering whether this rule should be applied depends completely on the typing rule. Thus the necessity to pass an implicit parameter may spontaneously pop up in any expression.

- In the presence of type inferencing nothing may be known yet about e at all, let alone which implicit parameters it may take. This information usually only becomes available after the generalization of the inferred types.
- These problems are usually circumvented by limiting the type language for types that are used during inferencing to predicate-free types. By effectively stripping a type from both its predicates and quantifiers standard Hindley-Milner (HM) type inferencing becomes possible. However, we allow predicated as well as quantified types to participate in type inferencing. As a consequence, predicates as well as quantifiers can be present in any type encountered during type inferencing.

Implicitness made explicit So, the bad news is that we do not know where implicit parameters need to be passed; the good news is that if we represent this lack of knowledge explicitly we can still figure out if and where implicit parameters need to be passed. This is not a new idea, because type variables are usually used to refer to a particular type about which nothing is yet known. The general strategy is to represent an indirection in time by the introduction of a free variable. In a later stage of a type inferencing algorithm such type variables are then replaced by more accurate knowledge, if any. Throughout the remainder of this section we work towards algorithmic versions of the type rules in which the solution to equations between types are computed by means of

- the use of variables representing unkown information
- the use of constraints on type variables representing found information

In our approach we also employ the notion of variables for sets of predicates, called *predicate wildcard variable*s, representing a yet unknown collection of implicit parameters, or, more accurately their corresponding predicates. These predicate wildcard variables are used in a type inferencing/checking algorithm which explicitly deals with expected (or known) types σ^k, as well as extra inferred type information.

Fig. 9.5 provides a summary of the judgement forms we use. The presence of properties in judgements varies with the version of typing rules. Both the most complex and its simpler versions are included.

These key aspects are expressed in the adapted rule for predicates shown in Fig. 9.6. This rule makes two things explicit:

- The context provides the expected (or known) type σ^k of e. Jointly operating, all our rules maintain the invariant that e will get assigned a type σ which is a subtype of σ^k, denoted by $\sigma \leqslant \sigma^k$ (σ is said to be subsumed by σ^k), enforced by a *fit* judgement (see Fig. 9.5 for the form of the more complex variant used later in this chapter). The *fit* judgement also yields a type σ, the result of the subsumption. This type is required because the known type σ^k may only be partially known, and additional type information is to be found in σ.

Notation	Meaning
σ	type
σ^k	expected/known type
$\square$	any type
v	type variable
ι	identifier
i	value identifier
I	(type) constructor identifier, type constant
Γ	assumptions, environment, context
C	constraints, substitution
$C_{k..l}$	constraint composition of $C_k ... C_l$
$\leqslant$	subsumption, "fits in" relation
ϑ	translated code
π	predicate
ϖ	predicate wildcard (collection of predicates)

Figure 9.4.: Legenda of type related notation

- An implicit parameter can be passed anywhere; this is made explicit by stating that the known type of e may start with a sequence of implicit parameters. This is expressed by letting the expected type in the premise be $\varpi \rightarrow \sigma^k$. In this way we require the type of e to have the form $\varpi \rightarrow \sigma^k$ and also assign an identifier ϖ to the implicit part.

A predicate wildcard variable makes explicit that we can expect a (possibly empty) sequence of implicit parameters and at the same time gives an identity to this sequence. The type language for predicates thus is extended with a predicate wildcard variable ϖ, corresponding to the dots '...' in the source language for predicates:

$$\begin{aligned} \pi ::= & \; I\ \overline{\sigma} \\ \mid & \; \pi \Rightarrow \pi \\ \mid & \; \varpi \end{aligned}$$

In algorithmic terms, the expected type σ^k travels top-to-bottom in the abstract syntax tree and is used for type checking, whereas σ travels bottom-to-top and holds the inferred type. If a fully specified expected type σ^k is passed downwards, σ will turn out to be equal to this type. If a partially specified type is passed downwards the unspecified parts may be filled in by the type inferencer.

The adapted typing rule E-PRED in Fig. 9.6 still is not of much a help in deciding when it should be applied. However, as we only have to deal with a limited number of language constructs, we can use case analysis on the source language constructs. In this chapter we only deal with function application, for which the relevant rules are shown in their

Version	Judgement	Read as
I	$\Gamma; \sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow C; \vartheta$	With environment Γ, expected type σ^k, expression e has type σ and translation ϑ (with dictionary passing made explicit), requiring additional constraints C.
EvK	$\Gamma; \sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta$	version for evidence + expected type only
Ev	$\Gamma \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta$	version for evidence only
I	$\Gamma \overset{fit}{\vdash} \sigma^l \leqslant \sigma^r : \sigma \rightsquigarrow C; \delta$	σ^l is subsumed by σ^r, requiring additional constraints C. C is applied to σ^r returned as σ. Proving predicates (using Γ) may be required resulting in coercion δ.
EvK	$\overset{fit}{\vdash} \sigma^l \leqslant \sigma^r : \sigma$	version for evidence + expected type only
I	$\Gamma \overset{pred}{\vdash} \pi \rightsquigarrow \vartheta : \sigma$	Prove π, yielding evidence ϑ and evidence type σ.
I	$\sigma^k \overset{pat}{\vdash} p : \sigma; \Gamma_p \rightsquigarrow C$	Pattern has type σ and variable bindings Γ_p.

Figure 9.5.: Legenda of judgement forms for each version

full glory in Fig. 9.8 and will be explained soon. The rules in Fig. 9.8 look complex. The reader should realize that the implementation is described using an attribute grammar system [21, 9] which allows the independent specification of all aspects which now appear together in a condensed form in Fig. 9.8. The tradeoff is between compact but complex type rules and more lengthy but more understandable attribute grammar notation.

Notation The typing rules in Fig. 9.7 and Fig. 9.8 are directed towards an implementation; additional information flows through the rules to provide extra contextual information. Also, the rule is more explicit in its handling of constraints computed by the rule labeled *fit* for the subsumption $\leqslant$; a standard substitution mechanism constraining the different variable variants is used for this purpose:

$$\begin{aligned} \mathit{bindv} &= v \mapsto \sigma \mid \varpi \mapsto \pi, \varpi \mid \varpi \mapsto \emptyset \\ C &= \overline{\mathit{bindv}} \end{aligned}$$

The mapping from type variables to types $v \mapsto \sigma$ constitutes the usual substitution for

$$\boxed{\Gamma;\sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta}$$

$$\frac{\overset{fit}{\vdash} \sigma_\iota \leqslant \sigma^k : \sigma \quad (\iota \mapsto \sigma_\iota) \in \Gamma}{\Gamma;\sigma^k \overset{expr}{\vdash} \iota : \sigma \rightsquigarrow \iota}\ \text{E-ID}_{EvK}$$

$$\frac{\Gamma \overset{pred}{\vdash} \pi \rightsquigarrow \vartheta_\pi : \sigma_\pi \quad \Gamma;\varpi \Rightarrow \sigma^k \overset{expr}{\vdash} e : \pi \Rightarrow \sigma \rightsquigarrow \vartheta_e}{\Gamma;\sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta_e\vartheta_\pi}\ \text{E-PRED}_{EvK}$$

Figure 9.6.: Implicit parameter passing with expected type

type variables. The remaining alternatives map a predicate wildcard variable to a possibly empty list of predicates.

Not all judgement forms used in Fig. 9.7 and Fig. 9.8 are included in this chapter; in the introduction we indicated we focus here on that part of the implementation in which explicit parameter passing makes a difference relative to the standard [26, 91, 44]. Fig. 9.5 provides a summary of the judgement forms we use.

The judgement **pred** (Fig. 9.5) for proving predicates is standard with respect to context reduction and the discharge of predicates [26, 44, 48], except for the scoping mechanism introduced. We only note that the proof machinery must now take the scoped availability of instances into account and can no longer assume their global existence.

Explicit parameter passing The rules in Fig. 9.7 specify the typing for the explicit parameter passing where an implicit parameter is expected. The rules are similar to those for normal parameter passing; the difference lies in the use of the predicate. For example, when reading through the premises of rule E-IAPP, the function e_1 is typed in a context where it is expected to have type $\pi_2 \to \sigma^k$. We then require a class definition for the actual predicate π_a of the function type to exist, which we allow to be instantiated using the *fit* judgement which matches the class predicate π_d with π_a and returns the dictionary type in σ_a. This dictionary type σ_a is the expected type of the argument.

Because we are explicit in the predicate for which we provide a dictionary value, we need not use any proving machinery. We only need the predicate to be defined so we can use its corresponding dictionary type for further type checking.

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

$$\frac{\begin{array}{c}\Gamma; \sigma_a \overset{expr}{\vdash} e_2 : \_ \rightsquigarrow C_2; \vartheta_2 \\ \_ \overset{fit}{\vdash} \pi_d \Rightarrow \sigma_d \leqslant \pi_a \Rightarrow v : \_ \Rightarrow \sigma_a \rightsquigarrow \_; \_ \\ \pi_d \rightsquigarrow \sigma_d \in \Gamma \\ \Gamma; \pi_2 \Rightarrow \sigma^k \overset{expr}{\vdash} e_1 : \pi_a \Rightarrow \sigma \rightsquigarrow C_1; \vartheta_1 \\ v \text{ fresh}\end{array}}{\Gamma; \sigma^k \overset{expr}{\vdash} e_1\ (!e_2 \leftsquigarrow \pi_2!) : C_2\sigma \rightsquigarrow C_{2..1}; \vartheta_1\ \vartheta_2}\ \text{E-IAPP}_I$$

$$\frac{\begin{array}{c}[\pi_a \rightsquigarrow p : \sigma_a], \Gamma^p, \Gamma; \sigma_r \overset{expr}{\vdash} e : \sigma_e \rightsquigarrow C_3; \vartheta_e \\ \sigma_a \overset{pat}{\vdash} p : \_; \Gamma^p \rightsquigarrow C_2 \\ \_ \overset{fit}{\vdash} \pi_d \Rightarrow \sigma_d \leqslant \pi_a \Rightarrow v_2 : \_ \Rightarrow \sigma_a \rightsquigarrow \_; \_ \\ \pi_d \rightsquigarrow \sigma_d \in \Gamma \\ \Gamma \overset{fit}{\vdash} \pi \Rightarrow v_1 \leqslant \sigma^k : \pi_a \Rightarrow \sigma_r \rightsquigarrow C_1; \_ \\ v_1, v_2 \text{ fresh}\end{array}}{\Gamma; \sigma^k \overset{expr}{\vdash} \lambda(!p \leftsquigarrow \pi!) \rightarrow e : C_{3..2}\pi_a \Rightarrow \sigma_e \rightsquigarrow C_{3..1}; \lambda p \rightarrow \vartheta_e}\ \text{E-ILAM}_I$$

Figure 9.7.: Type rules for explicit implicit parameters

The rule E-ILAM for λ-abstractions follows a similar strategy. The type of the λ-expression is required to have the form of a function taking an implicit parameter. This *fit* judgement states this, yielding a predicate π_a which via the corresponding class definition gives the dictionary type σ_a. The pattern is expected to have this type σ_a. Furthermore, the body e of the λ-expression may use the dictionary (as an instance) for proving other predicates so the environment Γ for e is extended with a binding for the predicate and its dictionary p.

Implicit parameter passing: application From bottom to top, rule E-APP in Fig. 9.8 reads as follows (to keep matters simple we do not mention the handling of constraints C). The result of the application is expected to be of type σ^k, which in general will have the structure $\varpi^k \Rightarrow v^k$. This structure is enforced and checked by the subsumption check described by the rule *fit*; the rule binds ϖ^k and v^k to the matching parts of σ^k similar to pattern matching. We will not look into the *fit* rules for $\leqslant$; for this discussion it is only relevant to know that if a ϖ cannot be matched to a predicate it will be constrained to $\varpi \mapsto \emptyset$. In other words, we start with assuming that implicit parameters may occur

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

$$\frac{\begin{array}{c}\overline{\pi_i^k \rightsquigarrow \vartheta_i^k}, \Gamma \overset{pred}{\vdash} C_3 \overline{\pi_a} \rightsquigarrow \overline{\vartheta_a} : \_ \\ \overline{\pi_i^k \rightsquigarrow \vartheta_i^k}, \Gamma; \sigma_a \overset{expr}{\vdash} e_2 : \_ \rightsquigarrow C_3; \vartheta_2 \\ \overline{\pi_i^k \rightsquigarrow \vartheta_i^k}, \Gamma; \varpi \Rightarrow v \to \sigma_r^k \overset{expr}{\vdash} e_1 : \overline{\pi_a} \Rightarrow \sigma_a \to \sigma \rightsquigarrow C_2; \vartheta_1 \\ \overline{\pi_i^k \rightsquigarrow \vartheta_i^k} \equiv inst_\pi(\overline{\pi_a^k}) \\ \Gamma \overset{fit}{\vdash} \varpi^k \Rightarrow v^k \leqslant \sigma^k : \overline{\pi_a^k} \Rightarrow \sigma_r^k \rightsquigarrow C_1; \_ \\ \varpi, \varpi^k, v^k, v \text{ fresh}\end{array}}{\Gamma; \sigma^k \overset{expr}{\vdash} e_1\ e_2 : C_3\sigma \rightsquigarrow C_{3..1}; \lambda \overline{\vartheta_i^k} \to \vartheta_1\ \overline{\vartheta_a}\ \vartheta_2}\ \text{E-APP}_I$$

$$\frac{\begin{array}{c}\overline{\pi_i^p \rightsquigarrow \vartheta_i^p}, \Gamma_p, \Gamma; \sigma_r \overset{expr}{\vdash} e : \sigma_e \rightsquigarrow C_3; \vartheta_e \\ \overline{\pi_i^p \rightsquigarrow \vartheta_i^p} \equiv inst_\pi(\overline{\pi_a}) \\ \sigma_p \overset{pat}{\vdash} p : \_; \Gamma_p \rightsquigarrow C_2 \\ \Gamma \overset{fit}{\vdash} \varpi \Rightarrow v_1 \to v_2 \leqslant \sigma^k : \overline{\pi_a} \Rightarrow \sigma_p \to \sigma_r \rightsquigarrow C_1; \_ \\ \varpi, v_i \text{ fresh}\end{array}}{\Gamma; \sigma^k \overset{expr}{\vdash} \lambda p \to e : C_{3..2}\overline{\pi_a} \Rightarrow C_3\sigma_p \to \sigma_e \rightsquigarrow C_3; \lambda \overline{\vartheta_i^p} \to \lambda p \to \vartheta_e}\ \text{E-LAM}_I$$

Figure 9.8.: Implicit parameter type rules

everywhere and subsequently we try to prove the contrary. The subsumption check $\leqslant$ gives a possible empty sequence of predicates $\overline{\pi_a^k}$ and the result type σ_r^k. The result type is used to construct the expected type $\varpi \Rightarrow v \to \sigma_r^k$ for e_1. The application $e_1\ e_2$ is expected to return a function which can be passed evidence for $\overline{\pi_a^k}$. We create fresh identifiers $\overline{\vartheta_i^k}$ and bind them to these predicates. Function $inst_\pi$ provides these names bound to the instantiated variants $\overline{\pi_i^k}$ of $\overline{\pi_a^k}$. The names $\overline{\vartheta_i^k}$ are used in the translation, which is a lambda expression accepting $\overline{\pi_a^k}$. The binding $\overline{\pi_i^k \rightsquigarrow \vartheta_i^k}$ is used to extend the type checking environment Γ for e_1 and e_2 which both are allowed to use these predicates in any predicate proving taking place in these expressions. The judgement for e_1 will give us a type $\overline{\pi_a} \Rightarrow \sigma_a \to \sigma$, of which σ_a is used as the expected type for e_2. The predicates $\overline{\pi_a}$ need to be proven and evidence to be computed; the top judgement **pred** takes care of this. Finally, all the translations together with the computed evidence forming the actual implicit parameters $\overline{\pi_a}$ are used to compute a translation for the application, which accepts the implicit parameters it is supposed to accept. The body $\vartheta_1 \overline{\vartheta_a} \vartheta_2$ of this lambda expression contains the actual

application itself, with the implicit parameters are passed before the argument.

Even though the rule for implicitly passing an implicit parameter already provides a fair amount of detail, some issues remain hidden. For example, the typing judgement for e_1 gives a set of predicates π_a for which the corresponding evidence is passed by implicit arguments. The rule suggests that this information is readily available in an actual implementation of the rule. However, assuming e_1 is a **let** bound function for which the type is currently being inferred, this information will only become available when the bindings in a **let** expression are generalized [46], higher in the corresponding abstract syntax tree. Only then the presence and positioning of predicates in the type of e_1 can be determined. This complicates the implementation because this information has to be redistributed over the abstract syntax tree.

Implicit parameter passing: λ-abstraction Rule E-LAM for lambda expressions from Fig. 9.8 follows a similar strategy. At the bottom of the list of premises we start with an expected type σ^k which by definition has to accept a normal parameter and a sequence of implicit parameters. This is enforced by the judgement *fit* which gives us back predicates $\overline{\pi_a}$ used in a similar fashion as in rule E-APP.

9.5 Discussion and related work

Soundness, completeness and principal types EH allows type expressions where quantifiers and predicates may be positioned anywhere in a type, and all terms can be explicitly typed with a type annotation. Thus we obtain the same expressiveness as System F (Theorem 6.5, page 92). What remains are the following questions:

- For a completely explicitly typed program, is our algorithm and implementation sound and complete? Evidence translation replaces predicates by dictionaries, of which the type is fully known. Thus we are confident that Theorem 6.5 (page 92) still holds.
- For a partially explicitly typed program, what is the characterisation of the types that can be inferred for the terms for which no type has been given? If we impose the additional restriction that predicates are absent from all types, we are confident that Theorem 6.4 (page 91) holds.

We have not investigated these questions in the sense of proving their truth or falsehood. However, we repeat our design starting point from Chapter 6:

- Stick to HM type inferencing, except for the following:
- Combine type checking and inferencing. In order to be able to do this, impredicative types are allowed to participate in HM type inferencing.

By design we avoid 'breaking' HM type inferencing. However, Faxen [27] demonstrates the lack of principal types for Haskell due to a combination of language features. EH's quantified class constraints solve one of the problems mentioned by Faxen.

Our choice to allow quantifiers and predicates at any position in a type expression provides the programmer with the means to specify the type signature that is needed, but also breaks principality because the type inferencer will infer only a specific one (with quantifiers and predicates as much as possible to the right) of a set of isomorphic types. We have not investigated this further.

In general it also is an open question what can be said about principal types and other desirable properties when multiple language features are combined into a complete language. In this light we take a pragmatic approach and design starting point: if the system guesses wrong, the programmer can repair it by adding extra (type) information.

Local instances Haskell only allows global instances because the presence of local instances results in the loss of principal types for HM type inference [114]:

let class *Eq a* **where** $eq :: a \rightarrow a \rightarrow Bool$
  instance *Eq Int* **where**
  instance *Eq Char* **where**
in *eq*

With HM the problem arises because *eq* is instantiated without being applied to an argument, hence no choice can be made at which type *Eq a* (arising from *eq*) should be instantiated at. In EH, we circumvent this problem by delaying the instantiation of *eq*'s type until it is necessary, for example when the value is used as part of an application to an argument [22].

Coherence is not a problem either because we do not allow overlapping instances. Although local instances may overlap with global instances, their use in the proving machinerey is dictated by their nesting structure, which is static: local instances take priority over global instances.

How much explicitness is needed Being explicit by means of the (! ... !) language construct very soon becomes cumbersome because our current implementation requires full specification of all predicates involved inside (! ... !). Can we do with less?

- Rule E-IAPP from Fig. 9.7 uses the predicate π_2 in $(!e_2 \leftsquigarrow \pi_2!)$ directly, that is, without any predicate proving, to obtain π_d and its corresponding dictionary type σ_d. Alternatively we could interpret $(!e_2 \leftsquigarrow \pi_2!)$ as an addition of π_2 to the set of predicates used by the predicate proving machinery for finding a predicate whose dictionary matches the type of e_2. However, if insufficient type information is known about e_2 more than one solution may be found. Even if the type of e_2 would be fully known, its type could be coerced in dropping record fields so as to match different dictionary types.

- We could drop the requirement to specify a predicate and write just $(!e_2!)$ instead of $(!e_2 \leftsquigarrow \pi_2!)$. In this case we need a mechanism to find a predicate for the type of the evidence provided by e_2. This is most likely to succeed in the case of a class system as the functions introduced by a class need to have globally unique names. For other types of predicates like those for dynamically scoped values this is less clear. By dropping the predicate in $(!e_2!)$ we also loose our advocated advantage of explicitness because we can no longer specify type related information.
- The syntax rule E-ILAM requires a predicate π in its implicit argument $(!p \leftsquigarrow \pi!)$. It is sufficient to either specify a predicate for this form of a lambda expression or to specify a predicate in a corresponding type annotation.

Whichever of these routes leads to the most useful solution for the programmer, if the need arises our solution always gives the programmer the full power of being explicit in what is required.

Binding time of instances One other topic deserves attention, especially since it deviates from the standard semantics of Haskell. We allow the re-use of dictionaries by means of record extension. Is the other way around allowed as well: can previously defined functions of a dictionary use newly added values? In a variation of the example for *nub*, the following invocation of *nub* is parameterized with an updated record; a new definition for *eq* is provided:

$$\begin{aligned} &\mathit{nub}\ (!(\mathit{dEqInt} \mid \mathit{eq} := \mathit{eqMod2}) \leftsquigarrow \mathit{Eq}\ \mathit{Int}!) \\ &\quad (\mathit{Cons}\ 3\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 4\ \mathit{Nil}))) \end{aligned}$$

In our implementation *Eq*'s function *ne* invokes *eq*, the one provided by means of the explicit parameterization, thus allowing open recursion. This corresponds to a late binding, much in the style employed by object oriented languages. This is a choice out of (at least) three equally expressive alternatives:

- Our current solution, late binding as described. The consequence is that all class functions now take an additional (implicit) parameter, namely the dictionary where this dictionary function has been retrieved from.
- Haskell's solution, where we bind all functions at instance creation time. In our *nub* example this means that *ne* still uses *dEqInt*'s *eq* instead of the *eq* provided in the updated $(\mathit{dEqInt} \mid \mathit{eq} := ...)$.
- A combination of these solutions, such as using late binding for default definitions, and Haskell's binding for instances.

Again, whichever of the solutions is preferred as the default case, especially in the light of the absence of open recursion in Haskell, we notice that the programmer has all the means available to express his differing intentions.

Dynamically scoped variables GHC [75] enables the passing of plain values as dynamically scoped variables (also known as implicit parameters). It is possible to model this effect [42, 69, 75] with the concepts described thus far. For example, the following program uses dynamically scoped variable ?*x*:

```
let f  :: (?x :: Int) ⇒  ...
    f  = λ                ... → ... ? x + 2 ...
     ? x = 3
in f ...
```

The signature of f specifies a predicate $?x :: Int$, meaning that f can refer to the dynamically scoped variable x with type *Int*. Its value is introduced as a binding in a **let** expression and is used in the body of f by means of $?x$. This can be encoded using the class system:

```
let class Has_x a where
      value_x :: a
   f :: (Has_x Int) ⇒  ...
   f = λ                ... → ...value_x + 2 ...
   instance Has_x Int where
      value_x = 3
in f ...
```

We only mention briefly some issues with this approach:

- The type for which an instance without context is defined usually is specified explicitly. This is no longer the case for '?' predicates if an explicit type signature for e.g. **let** $? x = 3$ is omitted.
- GHC [75] forbids dynamically scoped variable predicates in the context of instance declarations because it is unclear which scoped variable instance is to be taken. Scoping for instances as available in EHC may well obviate this restriction.
- Use of records for dictionaries can be optimized away because each class contains a single field only.

Our approach has the additional benefit that we are not obliged to rely on the proving machinery by providing a dictionary directly:

```
let class Has_x a ...
   f :: (Has_x Int) ⇒  ...
   f = λ                ... → ...value_x + 2 ...
in f (!(value_x = 3) ⇜⇝ Has_x Int!) ...
```

Named instances Scheffczyk has explored named instances as well [51, 98]. Our work differs in several aspects:

- Scheffczyk partitions predicates in a type signature into ordered and unordered ones. For ordered predicates one needs to pass an explicit dictionary, unordered ones are

those participating in the normal predicate proving by the system. Instances are split likewise into named and unnamed instances. Named instances are used for explicit passing and do not participate in the predicate proving. For unnamed instances this is the other way around. Our approach allows a programmer to make this partitioning explicitly, by stating which instances should participate in the proof process. In other words, the policy of how to use the implicit parameter passing mechanism is made by the programmer.

- Named instances and modules populate the same name space, separate from the name space occupied by normal values. This is used to implement functors as available in ML [66, 67] and as described by Jones [45] for Haskell. Our approach is solely based on normal values already available.
- Our syntax is less concise than the syntax used by Scheffczyk. This is probably difficult to repair because of the additional notation required to lift normal values to the evidence domain.

Implementation The type inferencing/checking algorithm employed in this chapter is described in greater detail in [21, 22] and its implementation is publicly available [19], where it is part of a work in progress. Similar strategies for coping with the combination of inferencing and checking are described by Pierce [92] and Peyton Jones [87].

9.6 Conclusion

Allowing explicit parameterization for implicit parameters gives the programmer an additional mechanism for reusing existing functions. It also makes explicit what otherwise remains hidden inside the bowels of a compiler. We feel that this a 'good thing': it should be possible to override automatically made decisions.

We have implemented all features described in this chapter in the context of a compiler for EH [22, 21]; in this paper we have presented the relevant part concerning explicit implicit parameters in an as compact form as possible. To our knowledge our implementation is the first combining language features like higher ranked types, existentials, class system, explicit implicit parameters and extensible records into one package together with a description of the implementation.

10 Partial type signatures

Specifying a type signature becomes cumbersome as soon as the specified type becomes complex. This is a reason why a type inferencer is so useful. However, if a type inferencer fails to infer the (intended) signature, we have to specify the signature ourselves. Thus far we have been required to specify the full signature, instead of only that part that cannot be inferred by the type inferencer.

In this chapter we investigate two techniques for relieving a programmer from the limitations of this "all or nothing" approach. Both techniques support a joint specification of a type by programmer and system:

- *Partial type signatures*. Often a complex type has only (relatively) small parts which are too complex to be inferred. With partial type signatures we allow the specification of those parts, leaving the remainder to be inferred by a type inferencer.
- *Quantifier location inference*. Thus far type variables in a type signature are to be explicitly quantified. Often $\forall$ is used for type variables relating function argument and result, and $\exists$ is used for type variables of a tuple (or other product) type. For these kind of uses, we may omit the quantifier, which subsequently is inferred by *quantifier location inference*.

We first examine some examples, followed by a discussion of the typing rules affected by these features. Both features are relatively independent of other language features, which is the reason why they are discussed separately. Examples are based on the EH4 (Chapter 6) and EH9 (Chapter 9). We only discuss the type rules within the context of EH4.

Partial type signatures Partial type signatures are specified by type expressions where three dots "...", called a *type wildcard*, denote unspecified parts. For example, in the following expression the identity function *id* is declared:

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

The type signature for *id* specifies that *id* should be a function, but does not state anything about its argument and result. The argument and result type remain unknown until the type inferencer can infer the argument and result type.

A type wildcard is similar to a type variable, because both represent unknown type information. However, type variables in a type signature represent quantified type variables, whereas type wildcards represent yet to be inferred types. If we want to refer to a type wildcard, we prefix a normal type variable with '%'. This is called a *named type wildcard*. The previous declaration of *id* can now be rewritten to:

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

Omission of '%' yields a type expression "$a \to b$", which is interpreted (by "Quantifier location inference") as "$(\exists a.a) \to \forall b.b$".

Named type wildcards allow the specification of additional constraints for use by the type inferencer. In the following, argument and result type are specified to be equal, and inferred to be of type *Int*. This results in an inferred signature "$ii :: Int \to Int$" for:

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

Used in this way, partial type signatures are similar to lexically scoped type variables [86]. Lexically scoped type variables allow scoped references to the types of parameters:

$$\begin{array}{l}\textbf{let}\ ii = \lambda(x :: a) \to (3 :: a) \\ \textbf{in}\ ii\ 3\end{array}$$

Partial type signatures are most useful in specifying only those parts which the type inferencer cannot infer, in particular higher-ranked type information:

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

For f we only need to specify its rank-2 typed argument; the rest is inferred.

Partial type signatures for implicit parameters Class predicates can be omitted as well. Within the context of the examples from Chapter 9 we start with the following function:

$$f = \lambda p\ q\ r\ s \to (eq\ p\ q, eq\ r\ s)$$

If f's type signature is omitted, we infer the following type:

$$f :: \forall\ a.Eq\ a \Rightarrow a \to a \to \forall\ b.Eq\ b \Rightarrow b \to b \to (Bool, Bool)$$

Variation 1: Now, if we want to make clear that the dictionary for b should be passed before any of the a's we write:

$f :: \forall \quad b.(Eq\ b,\ ...\ \ ) \Rightarrow ... \rightarrow ... \rightarrow b \rightarrow b \rightarrow ...$
-- INFERRED:
$f :: \forall\ a\ b.(Eq\ b, Eq\ a) \Rightarrow a\ \rightarrow a\ \rightarrow b \rightarrow b \rightarrow (Bool, Bool)$

The parts indicated by '...' are inferred.

Variation 2: The dots '...' in the type signature specify parts of the signature to be filled by the type inferencer. The inferred type may be polymorphic if no restrictions on its type are found by the type inferencer, or it may be monomorphic as for $r :: Int$ in:

$f :: \forall\ a.(Eq\ a,\ ...\ ) \Rightarrow a \rightarrow a \rightarrow\ ...$
$f = \lambda \qquad\qquad p \quad q \quad r \quad s \quad \rightarrow (eq\ p\ q, eq\ r\ 3)$
-- INFERRED:
$f :: \forall\ a.\ Eq\ a \qquad \Rightarrow a \rightarrow a \rightarrow Int \rightarrow \forall\ b.b \rightarrow (Bool\ \ , Bool\ \ )$

If '...' occurs in a type position, we call it a *type wildcard*. If '...' occurs in a in predicate position, we call it a *predicate wildcard*.

Although the given examples suggest that a wildcard may be used anywhere in a type, there are some restrictions:

- A named wildcard $\%a$ cannot be used as a predicate wildcard, because $\%a$ then would refer to a set of predicates; it does not make much sense to pass this set twice.
- A type wildcard can occur at an argument or result position of a function type. A type wildcard itself may bind to a polymorphic type with predicates. In other words, impredicativeness is allowed. This is particularly convenient for type wildcards on a function's result position. For example, the type wildcard $\%b$ in

  $f :: \forall\ a.Eq\ a \Rightarrow a \rightarrow a \rightarrow \%b$

  is bound to

  $\forall\ b.Eq\ b \Rightarrow b \rightarrow b \rightarrow (Bool, Bool)$

  after further type inferencing.
- For the non wildcard part of a type signature all occurrences of a type variable in the final type must be given. This is necessary because the type signature will be quantified over explicitly introduced type variables.
- A sequence of explicit predicates may end with a predicate wildcard, standing for an optional collection of additional predicates. Multiple occurrences of a predicate wildcard or between explicit predicates would defeat the purpose of being partially explicit. For example, for the type signature $(Eq\ b, ..., Eq\ c) \Rightarrow$ ... the argument position of $Eq\ c$'s dictionary cannot be predicted by the programmer.
- The absence of a predicate wildcard in front of a type means *no* predicates are allowed. The only exception to this rule is a single type variable or a type wildcard, since these may be bound to a type which itself contains predicates.

We need to impose these restrictions because the partially specified type represents the shape of a type: a combination of fixed and yet to be inferred parts. The fixed part corresponds to the universally quantified part of the partial type. The shape is then passed to an expression as its known type.

Quantifier location inference Quantifiers for a type signature are specified explicitly in its type expression, but may be omitted if their location in the type expression is where we expect them to be. Here, the notion of what we expect is based on the following observations:

- The universal quantifier $\forall$ is used to express the propagation of type information between argument and result type of a function type.
- The existential quantifier $\exists$ is used to express encapsulation of a type which represents data about which we want to hide type information.

For example, the type signature of the identity function *id* can be specified by the following type expression:

$$id :: a \to a$$

The expected location for $\forall$ is in front of the type signature. Similarly, the following type signature also specifies "$a \to a$", but on a rank-2 position:

$$f :: (a \to a) \to Int$$

Because type variable a is not referred to outside the rank-2 position, the expected location of $\forall$ is in front of "$a \to a$" at rank-2, not in front of the full type signature on a rank-1 position, which is Haskell's default.

For tuple types we put an existential quantifier in front. For example for:

$$v :: (a, a \to Int)$$
$$f :: (a, a \to Int) \to Int$$

A tuple type represents data composition. A type variable indicates lack of knowledge about the type of (part of) the data, hence is interpreted as 'to be forgotten' type information, or an existential type:

$$v :: \exists a.(a, a \to Int)$$
$$f :: (\exists a.(a, a \to Int)) \to Int$$

Informally, a quantifier for a type variable is placed in front of a type fragment if the type variable does not occur outside the type fragment. If that position is a function type, $\forall$ is used, if it is a product type, an $\exists$ is used. This algorithm is applied irrespective of the position of the type fragment, so the following type expressions:

$$f :: Int \to (a \to a)$$
$$g :: Int \to (a, a \to Int)$$

yield the following type signatures:

$f :: Int \to \forall a.a \to a$
$g :: Int \to \exists a.(a, a \to Int)$

For f, this resulting type is isomorphic to the placement of $\forall$ in front of the type: "$f :: \forall a.Int \to a \to a$". However, for g, the type has a different meaning if the quantifier $\exists$ is placed in front:

$g :: \exists a.Int \to (a, a \to Int)$

The first signature of g allows two different invocations of g to return two different types for the encapsulated type variable a. The second type signature is opened when bound to g, with a fresh type constant for type variable a:

$g :: Int \to (C\_1, C\_1 \to Int)$

Two different invocations now are required to return the same, but hidden, type.

A single type variable takes a somewhat special place, since there is no corresponding '$\to$' or '$(,)$' type constructur to determine which quantifier should be chosen. What is the interpretation of the following types?

$v :: a$
$f :: a \to b$

We interpret these types as follows:

$v :: \forall a.a$
$f :: (\exists a.a) \to \forall b.b$

v corresponds to Haskell's `undefined`, whereas f's type corresponds (by means of logical equivalence via the Curry-Howard isomorphism [102, 111, 112, 113]) with Haskell's interpretation:

$f :: \forall a.\forall b.a \to b$

Finally, we note that the automatic placement of quantifiers always can be overridden by means of an explicit specification of the quantifier.

10.1 Partial type signatures

Partial type signatures are already meaningful in early versions of EH. EH version 2 (Chapter 4) allows type variables; type wildcards are just type variables as far as the implementation is concerned. Rule T.WILD (Fig. 10.1) shows how a type variable is created. The type variable remains unbound, that is, we cannot refer to this type variable.

The next version of EH, version 3 (Chapter 5), allows references to type variables, via program identifiers. The process of collecting bindings for identifiers to type variables is similar to the collecting of bindings for value identifiers. We thread an extra environment Δ

$$\boxed{\Delta \vdash^t t : \sigma}$$

$$\frac{v \text{ fresh}}{\Delta \vdash^t \ldots : v}\ \text{T.WILD}_C$$

Figure 10.1.: Type expression type rules (C)

$$\boxed{\Delta^k \vdash^t t : \sigma \rightsquigarrow \Delta; \overline{v_w}}$$

$$\frac{I \mapsto \sigma \in \Delta^k}{\Delta^k \vdash^t I : \sigma \rightsquigarrow \Delta^k; [\,]}\ \text{T.CON}_{HM} \qquad \frac{\begin{array}{c}\Delta^k \vdash^t t_1 : \sigma_f \rightsquigarrow \Delta_f; \overline{v_f} \\ \Delta_f \vdash^t t_2 : \sigma_a \rightsquigarrow \Delta_a; \overline{v_a}\end{array}}{\Delta^k \vdash^t t_1\ t_2 : \sigma_f \sigma_a \rightsquigarrow \Delta_a; \overline{v_a}, \overline{v_f}}\ \text{T.APP}_{HM}$$

$$\frac{i \mapsto \sigma \in \Delta^k \ \vee\ \Delta_i \equiv (i \mapsto v) \wedge \sigma \equiv v \wedge v \text{ fresh}}{\Delta^k \vdash^t i : \sigma \rightsquigarrow \Delta_i, \Delta^k; [\,]}\ \text{T.VAR}_{HM}$$

$$\frac{i \mapsto \sigma \in \Delta^k \ \vee\ \Delta_i \equiv (i \mapsto v) \wedge \sigma \equiv v \wedge v \text{ fresh}}{\Delta^k \vdash^t \%i : \sigma \rightsquigarrow \Delta_i, \Delta^k; [v]}\ \text{T.VAR.W}_{HM}$$

Within environment Δ^k, type expression t has a (replica) type signature σ, yielding additional bindings Δ and wild type variables $\overline{v_w}$.
t : Type expression
σ : Type signature
$\overline{v_w}$: Type variables which occur as wildcard
Δ^k : Environment $\overline{i \mapsto \sigma}$ with known bindings for type identifiers
Δ : Environment with Δ^k+ new bindings

Figure 10.2.: Type expression type rules (HM)

through the type expression. At rule T.VAR and its wildcard variation rule T.VAR.W (Fig. 10.2) a binding is added to Δ.

The type expression is used in a declaration of a type signature for a value identifier. In rule D.TYSIG (Fig. 5.2, page 78) the type signature of this type expression is quantified over its free type variables, except those which are introduced as a type wildcard: these are still free to be bound by type inference. Generalisation is done a second time after type

Notation	Meaning
$\mathcal{V}$	co-, contravariant context
$\mathcal{V}^+$	covariant context
$\mathcal{V}^-$	contravariant context

Figure 10.3.: Notation for co- and contravariant context

inference.

10.2 Quantifier location inference

Our algorithm to place quantifiers is based on the rules which are specified in Fig. 10.4:

- If a type variable occurs in two components of a composite type, and the type variable does not occur outside the composite type, the composite type is the quantifier location for the type variable (rule TY.QU.ARROW, rule TY.QU.PROD).
- If a quantifier location is in front of a product type, an $\exists$ is used (rule TY.QU.PROD), if in front of a function type, a $\forall$ is used (rule TY.QU.ARROW).
- For a standalone type variable (does not occur elsewhere in the type), a $\forall$ is used in a co-variant context, $\exists$ otherwise.

For the co- or contravariance context we use some additional notation (Fig. 10.3).

$$\boxed{\overline{v_g}; \mathcal{V} \vdash^{\mathcal{Q}} \sigma : \sigma_q \rightsquigarrow \overline{v_f}}$$

$$\frac{v \notin \overline{v_g} \qquad \mathcal{Q} \equiv \textbf{if } \mathcal{V} \equiv \mathcal{V}^+ \textbf{ then } \forall \textbf{ else } \exists}{\overline{v_g}; \mathcal{V} \vdash^{\mathcal{Q}} v : \mathcal{Q}\ v.v \rightsquigarrow [v]}\ \text{TY.QU.VAR}_{I1}$$

$$\frac{\begin{array}{c} v, \overline{v_g}; \mathcal{V}^+ \vdash^{\mathcal{Q}} \sigma_a : \sigma_a^q \rightsquigarrow \overline{v_a^f} \\ v, \overline{v_g}; \mathcal{V}^- \vdash^{\mathcal{Q}} \sigma_r : \sigma_r^q \rightsquigarrow \overline{v_r^f} \\ v \in (\overline{v_a^f} \cap \overline{v_r^f}) \backslash \overline{v_g} \end{array}}{\overline{v_g}; \_ \vdash^{\mathcal{Q}} \sigma_a \to \sigma_r : \forall v.\sigma_a^q \to \sigma_r^q \rightsquigarrow \overline{v_a^f} \cup \overline{v_r^f}}\ \text{TY.QU.ARROW}_{I1}$$

$$\frac{\begin{array}{c} v, \overline{v_g}; \mathcal{V}^+ \vdash^{\mathcal{Q}} \sigma_l : \sigma_l^q \rightsquigarrow \overline{v_l^f} \\ v, \overline{v_g}; \mathcal{V}^+ \vdash^{\mathcal{Q}} \sigma_r : \sigma_r^q \rightsquigarrow \overline{v_r^f} \\ v \in (\overline{v_l^f} \cap \overline{v_r^f}) \backslash \overline{v_g} \end{array}}{\overline{v_g}; \_ \vdash^{\mathcal{Q}} (\sigma_l, \sigma_r) : \exists v.(\sigma_l^q, \sigma_r^q) \rightsquigarrow \overline{v_l^f} \cup \overline{v_r^f}}\ \text{TY.QU.PROD}_{I1}$$

$$\frac{\begin{array}{c} v, \overline{v_g}; \mathcal{V} \vdash^{\mathcal{Q}} \sigma : \sigma^q \rightsquigarrow \overline{v^f} \\ v \notin \overline{v_g} \end{array}}{\overline{v_g}; \mathcal{V} \vdash^{\mathcal{Q}} \mathcal{Q}\ v.\sigma : \mathcal{Q}\ v.\sigma^q \rightsquigarrow \overline{v^f} \backslash [v]}\ \text{TY.QU.QUANT}_{I1}$$

Type σ_q equals σ, with quantifiers for type variables in σ_q not in $\overline{v_g}$
$\mathcal{V}$: Co/contravariance context, used internally
σ : Type to be quantified
$\overline{v_f}$: Free type variables of σ, used internally
$\overline{v_g}$: Global type variables, are not quantified
σ_q : Quantified type

Figure 10.4.: Type quantification rules (I1)

11 RULER: PROGRAMMING TYPE RULES

Note to the reader: this chapter is a slightly adapted version of a submitted paper. It can be read independently of previous chapters. The original paper includes a short introduction to the AG system; this has been omitted as Chapter 2 can be read instead.

Some type systems are first described formally, to be sometimes followed by an implementation. Other type systems are first implemented as a language extension, to be sometimes retrofitted into a formal description. In neither case it is an easy task to keep both artefacts consistent. In this chapter we present *Ruler*, a domain specific language for type rules. Our prototype compiler for *Ruler* both generates (1) a visual LaTeX rendering, suitable for use in the presentation of formal aspects, and (2) an attribute grammar based implementation. Combining these two aspects in *Ruler* contributes to bridging the gap between theory and practice: mutually consistent representations can be generated for use in both theoretical and practical settings.

11.1 Introduction

Theory and practice of type systems often seem to be miles apart. For example, for the programming language Haskell the following artefacts exist:

- A language definition for the Haskell98 standard [84], which defines Haskell's syntax and its meaning in informal terms. Part of this is specified in the form of a translation to a subset of Haskell.
- A formal description of the static semantics of most of Haskell98 [26].
- Several implementations, of which we mention GHC [75] and Hugs [2].
- Experimental language features of which some have been formally described in isolation, some of them found their way into Haskell, or are available as non-standard features. As an example we mention Haskell's class system [44], and multi-parameter type classes [85, 25] present in extensions [75, 2] to Haskell98.

- A Haskell description of type inferencing for Haskell98 [48], serving at the same time as a description and an implementation.

We can ask ourselves the following questions:

- What is the relationship between all the descriptions (i.e language definition and static semantics) of Haskell and available implementations?
- What is the effect of a change or extension which is first implemented and subsequently described?
- What is the effect of a change or extension which is first described and subsequently implemented?

For example, if we were to extend Haskell with a new feature, we may start by exploring the feature in isolation from its context by creating a minimal type system for the feature, an algorithmic variant of such a type system, a proof of the usual properties (soundness, completeness), or perhaps a prototype. Upto this point the extension process is fairly standard; however when we start to integrate the feature into a working implementation this process and the preservation of proven properties becomes less clear. Whatever route we take, that is, first extend the implementation then give a formal description or the other way around, there is no guarantee that the formal description and the implementation are mutually consistent. Even worse, we cannot be sure that an extension preserves the possibility to prove desirable properties. As a example, it has already been shown that Haskell does not have principal types, due to a combination of language features and seemingly innocent extensions [27].

Based on these observations we can identify the following problems:

Problem 1. It is difficult, if not impossible, to keep separate (formal) descriptions and implementations of a complex modern programming language consistent.

Our approach to this problem is to maintain a single description of the static semantics of a programming language. From this description we generate both the material required for a formal treatment as well as the implementation.

Problem 2. The extension of a language with a new feature means that the interaction between the new and all old features needs to be examined with respect to the preservation of desirable properties, where a property may be formal (e.g. soundness) or practical (e.g. sound implementation).

The *Ruler* language that we introduce in this paper aims to make it easy to describe language features in relative isolation. The separate descriptions for these features however can be combined into a description of the complete language. Note that traditional programming language solutions, like the use of modules and abstract data types, are not

$$\boxed{\Gamma \vdash^e e : \tau}$$

$$\frac{}{\Gamma \vdash^e int : Int}\ \text{E.INT}_E \qquad \frac{i \mapsto \sigma \in \Gamma \quad \tau = inst\ (\sigma)}{\Gamma \vdash^e i : \tau}\ \text{E.VAR}_E \qquad \frac{\Gamma \vdash^e a : \tau_a \quad \Gamma \vdash^e f : \tau_a \to \tau}{\Gamma \vdash^e f\ a : \tau}\ \text{E.APP}_E$$

$$\frac{(i \mapsto \tau_i), \Gamma \vdash^e b : \tau_b}{\Gamma \vdash^e \lambda i \to b : \tau_i \to \tau_b}\ \text{E.LAM}_E \qquad \frac{(i \mapsto \sigma_e), \Gamma \vdash^e b : \tau_b \quad \Gamma \vdash^e e : \tau_e \quad \sigma_e = \forall \overline{v}.\tau_e, \quad \overline{v} \notin ftv\ (\Gamma)}{\Gamma \vdash^e \textbf{let}\ i = e\ \textbf{in}\ b : \tau_b}\ \text{E.LET}_E$$

Figure 11.1.: Expression type rules (E)

sufficient: a language extension often requires the extension of the data types representing the abstract syntax and the required implementation may require changes across multiple modules.

Our approach is similar to, but also different from literate programming. We emphasize that *Ruler* provides a solution for the abovementioned problems; in the conclusion (Section 11.7) we will further discuss additional desirable features of the *Ruler* system.

How our approach contributes to solving the problems We explore these problems and our solution by looking at the final products that are generated by the *Ruler* system as described in this chapter, and which are presented in figures 11.1 through 11.3. We emphasize at this point that a full understanding of these figures is not required nor intended. The focus of this chapter is on the construction of the figures, not on their meaning. Our aim is to look at these figures from a metalevel, to see how type rules can be specified and how their content can be generated using our *Ruler* system. Nevertheless, we have chosen a realistic running example: the Hindley-Milner (HM) type system. Fig. 11.1 gives the equational rules, Fig. 11.2 the algorithmic variant and Fig. 11.3 part of the generated implementation. In later sections we will come back to the technical part of these figures. For now we only use their content to discuss the general idea of our approach.

The need for a system producing these artefacts arose in the context of the Essential Haskell (EH) project [21, 19]. The design goal of EH is to construct a compiler for an extended version of Haskell, and to build (simultaneously) an explanation of its implementation, in which we try to keep both versions consistent by generating corresponding parts from a single source. This approach resembles the one taken by Pierce [91] who explains both non-algorithmic and algorithmic variants of type systems. The EH project starts with the description of a very simple language, and extends it in a sequence of steps, leading to full

$$\boxed{C^k;\Gamma \vdash^e e : \tau \rightsquigarrow C}$$

$$\frac{}{C^k;\Gamma \vdash^e int : Int \rightsquigarrow C^k}\ \text{E.INT}_A \qquad \frac{\begin{array}{c} i \mapsto \sigma \in \Gamma \\ \tau = inst\ (\sigma) \end{array}}{C^k;\Gamma \vdash^e i : \tau \rightsquigarrow C^k}\ \text{E.VAR}_A$$

$$\frac{\begin{array}{c} C^k;\Gamma \vdash^e f : \tau_f \rightsquigarrow C_f \\ C_f;\Gamma \vdash^e a : \tau_a \rightsquigarrow C_a \\ v \text{ fresh} \\ \tau_a \to v \cong C_a\tau_f \rightsquigarrow C \end{array}}{C^k;\Gamma \vdash^e f\ a : C\ C_a v \rightsquigarrow C\ C_a}\ \text{E.APP}_A \qquad \frac{\begin{array}{c} v \text{ fresh} \\ C^k;(i \mapsto v),\Gamma \vdash^e b : \tau_b \rightsquigarrow C_b \end{array}}{C^k;\Gamma \vdash^e \lambda i \to b : C_b v \to \tau_b \rightsquigarrow C_b}\ \text{E.LAM}_A$$

$$\frac{\begin{array}{c} v \text{ fresh} \\ C^k;(i \mapsto v),\Gamma \vdash^e e : \tau_e \rightsquigarrow C_e \\ \sigma_e = \forall\ (ftv\ (\tau_e) \backslash ftv\ (C_e\Gamma)).\tau_e \\ C_e;(i \mapsto \sigma_e),\Gamma \vdash^e b : \tau_b \rightsquigarrow C_b \end{array}}{C^k;\Gamma \vdash^e \textbf{let}\ i = e\ \textbf{in}\ b : \tau_b \rightsquigarrow C_b}\ \text{E.LET}_A$$

Figure 11.2.: Expression type rules (A)

```
data Expr
   | App f   : Expr
         a   : Expr
attr Expr [g : Gam | c : C | ty : Ty]
sem Expr
   | App (f.uniq, loc.uniq1)
                = rulerMk1Uniq @lhs.uniq
         loc.tv_ = Ty_Var @uniq1
         (loc.c_, loc.mtErrs)
                = (@a.ty `Ty_Arr` @tv_) ≅ (@a.c ⊕ @f.ty)
         lhs.c  = @c_ ⊕ @a.c
            .ty = @c_ ⊕ @a.c ⊕ @tv_
```

Figure 11.3.: Part of the generated implementation

Haskell with extensions (including higher ranked polymorphism, mechanisms for explicitly passing implicit parameters [20, 23], extensible records [29, 49], higher order kinds). Each step introduces new features and describes the corresponding compiler.

Both type rules and fragments of corresponding source code are used in the explanation of the compiler. For example, rule E.APP from Fig. 11.2 and the corresponding attribute grammar (AG) implementation from Fig. 11.3 are jointly explained, each strengthening the understanding of the other. However, later versions of EH introduce more features, resulting in the following problems:

- Type rules and AG source code both become quite complex and increasingly difficult to understand.
- A proper understanding may require explanation of a feature both in isolation as well as in its context. These are contradictory requirements.
- With increasing complexity comes increasing likeliness of inconsistencies between type rules and AG source code.

Part of our solution to these problems is the use of the concept of *views* on both the type rules and AG source code. Views are ordered in the sense that later views are built on top of earlier views. Each view is defined in terms of its differences with its ancestor view; the resulting view on the artefact is the accumulation of all these incremental definitions.

This, of course, is not a new idea: version managment systems use similar mechanisms, and object-oriented systems use the notion of inheritance. However, the difference lies in our focus on a whole sequence of versions as well as the changes between versions: in the context of version management only the latest version is of interest, whereas for a class hierarchy we aim for encapsulation of changes. We need simultaneous access to all versions, which we call views, in order to build both the explanation and the sequence of compilers. A version management systems uses versions as a mechanism for evolution, whereas we use views as a mechanism for explaining and maintaining EH's sequence of compilers.

For example, Fig. 11.1 displays view *E* (equational), and Fig. 11.2 displays view *A* (algorithmic) on the set of type rules. View *A* is built on top of view *E* by specifying the differences with view *E*. In the electronic version of this thesis, the incremental definition of these views is exploited by using a color scheme to visualise the differences. The part which has been changed with respect to a previous view is displayed in blue (or black when printed); the unchanged part is displayed in grey (we will come back to this in our discussion). In the paper version of this thesis all rules are typeset (and printed) in black. In this way we address "Problem 2".

Independently from the view concept we exploit the similarity between type rules and AG based implementations. To our knowledge this similarity has never been exploited. We use this similarity by specifying type rules using a single notation, but which contains enough information to generate both the sets of type rules (in Fig. 11.1 and Fig. 11.2) as well as part

of the AG implementation (in Fig. 11.3). Fig. 11.3 shows the generated implementation for rule E.APP. In this way we address "Problem 1".

Our *Ruler* system allows the definition of type rules, views on those rules, and the specification of information directing the generation of a partial implementation. In addition, *Ruler* allows the specification of the structure of type rules: the type of a type rule. This "type of a type rule" is used by *Ruler* to check whether concrete type rules follow the correct pattern.

In the course of the EH project the Ruler system has become indispensable for us:

- *Ruler* is a useful tool for describing type rules and keeping type rules consistent with their implementation. In subsequent sections we will see how this is accomplished.
- It is relatively easy to incorporate the generation of output to be used as input for other targets (besides LaTeX and AG). This makes *Ruler* suitable for other goals while at the same time maintaining a single source for type rules.
- We also feel that it may be a starting point for a discussion about how to deal with the complexities of modern programming languages, and both their formal and practical aspects. In this light, this chapter also is an invitation to the readers to improve on these aspects. In our conclusion (Section 11.7) we will discuss some developments we foresee and directions of further research.

We summarize *Ruler*'s strong points, such that we can refer to these points from the technical part of this chapter:

Single source. Type rules are described by a single notation, all required type rule related artefacts are generated from this.

Consistency. Consistency between the various type rule related artefacts is guaranteed automatically as a consequence of being generated from a single source.

Incrementality. It is easy to incrementally describe type rules.

The remainder of this chapter is organised as follows: in Section 11.2 we present an overview of the *Ruler* system. This overview gives the reader an intuition of what *Ruler* can do and how it interacts with other tools. Preliminaries for the example language and type systems are given in Section 11.3. In Section 11.4 we specify the contents of Fig. 11.1, in Section 11.5 we extend this specification for the contents of Fig. 11.2. In Section 11.6 we extend the example *Ruler* specification so that *Ruler* can generate AG code. Finally we discuss and conclude in Section 11.7.

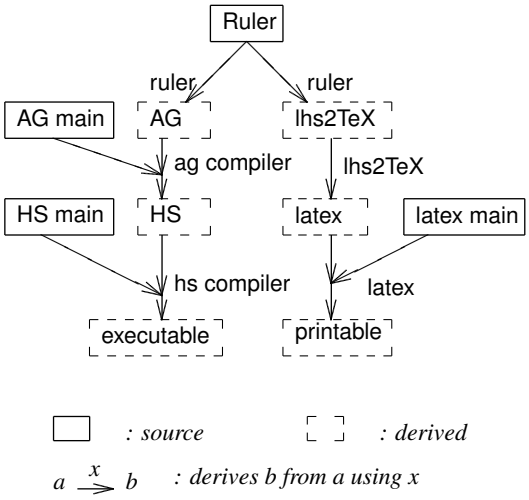

Figure 11.4.: Ruler overview

```
scheme X =
  view A =
    holes ...
    judgespec ...
  view B =
    holes ...
    judgespec ...
ruleset x scheme X =
  rule r =
    view A =
      judge ...    -- premises
       ...
       –
      judge ...    -- conclusion
    view B = ...
  rule s =
    view A = ...
    view B = ...
```

Figure 11.5.: High level structure of Ruler source

11.2 *Ruler* overview

Infrastructure around Ruler Although the *Ruler* system allows us to generate part of an implementation, it is by no means the only tool we use in the construction of our compilers. Fig. 11.4 gives an overview of the tools used to construct the example compiler for the type rules presented in this chapter. In the left branch we generate an executable compiler using the following sources:

- *Ruler* code (in box 'Ruler') for type rules, out of which attribute grammar AG code is generated by *Ruler*.
- AG code (in box 'AG main') for the specification of a pretty printed representation of the input and error handling. The AG compiler generates Haskell.
- Haskell code (in box 'HS main') for the specification of a parser, interaction with the outside world and remaining functionality.

In the right branch we generate LaTeX commands for *Ruler* type rules which can be used in a LaTeX document (in box 'latex main'). The major part of generating LaTeX is delegated to *lhs2TeX* [72].

The use of tools for the EH compilers is slightly more complicated because we need to specify different views on AG and Haskell code as well. A separate fragment management tool, called *shuffle* (part of the EH project [19]), is used to generate AG and Haskell code from code fragments describing the view inheritance chains for AG and Haskell code. Because we do not discuss this any further, this part has been omitted (from Fig. 11.4).

The design of *Ruler* In the remainder of this section we discuss the concepts used in *Ruler* by inspecting elements of figures 11.1, 11.2 and 11.3.

The design of *Ruler* is driven by the need to check the following properties of type rules:

- All judgements match an explicitly specified structure for the judgement. For example, in Fig. 11.1 all judgements for an expression should match the structure of an expression judgement in the box at the top of the same figure.
- If an identifier is used for the generation of an implementation, it must be defined before it can be used (the meaning of this will be explained later).

Other properties can be added to this list, but we limit ourselves to this list and the requirement of output generation for different targets.

In the remainder of this section we give a high-level overview of the concepts manipulated by *Ruler*. Fig. 11.5 gives a schematic *Ruler* specification, showing how these concepts relate syntactically.

The structure of a judgement is described by a *scheme*. On each scheme multiple *views* exist. A view on a scheme consists of named *holes* and a set of templates referring to these holes. Such templates, called *judgeshape*s, come in two varieties:

- A *judgespec*, used to specify the template by which judgements, which together form a rule, are specified.
- A *judgeuse*, used to specify the template which is used to specify how to display a judgement for an output target.

A *rule* consists of a set of judgements (syntactically called a *judge*) for the premises and a judgement for the conclusion. On each rule multiple views exists. The above judgements are defined for each view. Each of these judgements is of a specified scheme, and its definition must comply with the structure defined by the corresponding view on the scheme. A judgement is defined by bindings of hole names to *Ruler* expressions. These bindings are either specified by the use of a template (introduced by a judgespec) or specified for each hole individually.

Rules are grouped into *rulesets*. A ruleset corresponds to a figure like Fig. 11.1, so it consists of a set of rules, the scheme for which the rules specify a conclusion, and additional information like the text for the caption of the figure.

Views are ordered by a *view hierarchy*. A view hierarchy specifies which view inherits from which other (ancestor) view. A view on a scheme inherits the holes and judgeshapes. A view on a rule inherits the hole bindings.

Fig. 11.5 presents a schematic, high-level *Ruler* specification. The syntactic structure of a *Ruler* specification reflects the relationships between the aforementioned concepts. The incremental definition of views on a rule is supported by two different variants of specifying a judgement (using the above mechanisms):

- A judgement in a (view on a) rule can be specified by using a judgespec as a macro where the values of the holes are defined by filling in the corresponding positions in the judgespec. This variant is useful for the first view in a viewhierarchy, because all holes need to be bound to a *Ruler* expression.
- A judgement in a (view on a) rule can be specified by individually specifying values for each hole. This variant is useful for views which are built on top of other views, because only holes for which the value differs relative to the ancestor view need to be given a new value.

The incremental definition of views on a scheme is supported in a similar way: only the holes not present in an ancestor view require a definition.

The *Ruler* system is open-ended in the sense that some judgements can be expressed in a less structured form, for which its implementation is defined externally. For example, the premises of rule E.VAR consist of arbitrary conditions. These arbitrary (i.e. as far as *Ruler* is concerned unstructured) conditions are treated like regular judgements, but their

Values (expressions, terms):	
$e ::= int$	literals
$\mid i$	program variable
$\mid e\ e$	application
$\mid \lambda i \to e$	abstraction
$\mid$ **let** $i = e$ **in** e	local definitions

Figure 11.6.: Terms

Types:	
$\tau ::= Int$	literals
$\mid v$	variable
$\mid \tau \to \tau$	abstraction
$\sigma ::= \forall \overline{v}.\tau$	universally quantified type, $\overline{v}$ possibly empty

Figure 11.7.: Types

implementation has to be specified explicitly. We call the scheme of such a judgement variant a *relation*.

11.3 Preliminaries

In this section we introduce notation used by our running example, that is, the set of type rules to be specified by *Ruler*. There should be no surprises here as we use a standard term language based on the λ-calculus (see Fig. 11.6). A short overview of the type related notation is included in Fig. 11.8. Our example language contains e.g. the following program:

let $id = \lambda x \to x$
in **let** $v_1 = id\ 3$
in **let** $v_2 = id\ id$
in $v_2\ v_1$

The type language for our example term language is given in Fig. 11.7. Types are either monomorphic types τ, called *monotypes*, or universally quantified types σ, called *polymorphic types* or *polytypes*. A monotype either is a type constant *Int*, a function type $\tau \to \tau$, or an unknown type represented as a type variable v. We discuss the use of these types when we introduce the typing rules for our term language in the following sections.

Notation	Meaning
σ	type (possibly polymorphic)
τ	type (monomorphic)
$\overline{x}$	sequence of x (possibly empty)
v	type variable
Γ	$\overline{i \mapsto \sigma}$, assumptions, environment, context
C	$\overline{v \mapsto \tau}$, constraints, substitution
$\cong$	type matching relation, unification

Figure 11.8.: Legenda of type related notation

The typing rules use an environment Γ, holding bindings for program identifiers with their typings:

$$\Gamma ::= \overline{i \mapsto \sigma}$$

During HM type inferencing, type variables will be bound to monotypes:

$$C ::= \overline{v \mapsto \tau}$$

A C represents constraints on type variables, usually called a *substitution*. Its application to a type, denoted by juxtapositioning, has the usual meaning; it replaces type variables with types.

11.4 Describing typing rules using *Ruler* notation

In this section we make the use of *Ruler* more precise. We start by describing how to specify the content of Fig. 11.1 using *Ruler* notation. The full *Ruler* syntax is given in Fig. 11.9 and Fig. 11.10. The rules in Fig. 11.1 specify the non-algorithmic version of the typing rules for our term language. The transition (instantiation) from polytypes to monotypes is performed by *inst*, whereas the transition (generalisation) from monotypes to polytypes is happens in rule E.LET.

Because the rules implicitly state that certain equalities between types (of terms) should hold, we call this the equational view; the subscript E is used throughout this chapter to identify equational views.

The use of an equational version of typing rules usually serves to explain a type system and to prove properties about the type system. An algorithmic version subsequently is introduced to specify an implementation for such a type system. In this chapter we follow the same pattern, but use it to show how *Ruler* can be used to describe both type systems in such a way that its type rule representation can be included in the documentation (read here: this chapter) and its partial implementation can be integrated into a full implementation.

The basics: judgement schemes A typing rule consists of judgements describing the conclusion and premises of the rule. A judgement has a structure of its own, described by a *scheme*. A scheme plays the same role in rules as a type does for an expression in our example term language. In our example, we want to specify a judgement for terms (expressions), so we start a new **scheme** declaration by:

scheme *expr* =

which is immediately followed by the views on this scheme. Each view defines empty slots (**holes**), the judgement shape (*judgeshape*) by which concrete judgements will be specified (**judgespec**) and judgement shapes that will be used for output generation (**judgeuse**). Holes act like parameters to a judgement shape. The view *E* on scheme *expr* is defined by:

view *E* =
  holes [| *e* : *Expr*, *gam* : *Gam*, *ty* : *Ty* |]
  judgespec *gam* ⊢ *e* : *ty*
  judgeuse tex *gam* ⊢ .."e" *e* : *ty*

Here we specify for view *E*, that is the equational view, three empty slots (*e*, *gam*, *ty*), or *holes*, denoted by names (alphanumerical identifiers), which are to be filled in by judgements based on this scheme. Each hole has an associated hole type, so *ty* has type *Ty*; we postpone the discussion of hole types until Section 11.6. Holes can be filled in two different ways:

- A judgespec can be used as a macro by passing arguments at the hole positions.
- Holes are individually assigned a value by referring to their name.

Judgeshapes are introduced by the keyword **judgespec** or **judgeuse**. A **judgespec** judgement shape introduces the template which is to be used to specify a concrete judgement. A **judgeuse** judgement shape introduces the template which is used for the generation of output. A **judgeuse** specifies the kind of output, called a *target*, as well. The target **tex** indicates that the shape is to be used to generate LaTeX; later we will use the target **ag** to indicate that the shape is to be used for AG generation. We will refer to these three shapes as the **spec**, **tex** and **ag** judgement shapes.

A *Ruler* expression (*rexpr*), is used to specify the shape of a judgespec. The text for a *Ruler* expression already appears in pretty printed form throughout this chapter, but in the original source code the **spec** judgement shape appears as:

```
judgespec gam :- e : ty
```

A *Ruler* expression consists of a distfix operator with simple expressions as its operands. A distfix operator consists of operator symbols, which are denoted by combinations of operator like characters such as ':' and '-'. A simple expression may be the (possibly empty) juxtapositioning of a mixture of identifiers, parenthesized expressions or one of the other ⟨*rexpr_base*⟩ alternatives in Fig. 11.10.

The identifiers of a judgeshape should refer to the introduced hole names. When using a judgespec, the expression is matched against its associated judgespec, thus binding the hole identifiers occurring in the judgespec.

The dot character '.' has a special role in *Ruler* expressions and names for the **tex** target output generation. It is used to specify subscripts, superscripts and stacking on top of each other. For example, x.1.2.3 pretty prints as:

$$\overset{3}{x_1^2}$$

The part after the first dot is used as a subscript, the part after the second dot is used as a superscript, and the part after the third dot is stacked on top. In this context the underscore character '_' denotes a horizontal line for use in vector like notations, so `v..._` pretty prints as $\overline{v}$. Additional dots are ignored.

Names, rexpr's and operators all may be immediately followed by this dot notation. For names however, the dots and their related information form part of the name.

Since the judgespec and an associated **judgeuse tex** are usually quite similar, we have decided to make the latter default to the first. For this reason we allow the dot notatation to be used in the judgespec too, although it only will play a role in the defaulted use.

The basics: rulesets Rules are grouped in rulesets to be displayed together in a figure. So the description of Fig. 11.1 starts with:

ruleset *expr.base* **scheme** *expr* `"Expression type rules"` =

specifying the name *expr.base* of the ruleset, the scheme *expr* for which it defines rules, and text to be displayed as part of the caption of the figure. The judgespec of (a view on) the scheme is used to provide the boxed scheme representation in Fig. 11.1. LaTeX commands are generated for all the individual rules as well as for the figure for the full ruleset, for all defined views. The ruleset name *expr.base* is used to uniquely label the names of these LaTeX commands. We do not discuss this further; we only note that part of the LaTeX formatting (e.g. for a single rule) is delegated to external LaTeX commands.

The ruleset heading is immediately followed by a list of rules, of which only one is shown here (*e.int* is pretty printed in small caps as E.INT):

rule *e.int* =
  view *E* =
    –
      judge *R* : *expr* = *gam* ⊢ *int* : *Ty_Int*

Before discussing its components, we repeat its LaTeX rendering from Fig. 11.1 to emphasize the similarities between the rule specification and its visual appearance:

$$\frac{}{\Gamma \vdash^{e} \mathit{int} : \mathit{Int}}\ \textsc{e.int}_{E}$$

```
⟨ruler_prog⟩      ::= ( ⟨scheme_def⟩ | ⟨format_def⟩ | ⟨rewrite_def⟩
                      | ⟨rules_def⟩ | ⟨viewhierarchy_def⟩
                      | ⟨external_def⟩
                      ) *
⟨scheme_def⟩      ::= (scheme | relation)⟨nm⟩[⟨ag_nm⟩]
                        '='⟨scm_view_def⟩ *
⟨scm_view_def⟩    ::= view⟨vw_nm⟩'='⟨hole_def⟩⟨shape_def⟩ *
⟨hole_def⟩        ::= hole '['⟨hole_defs⟩
                           '|'⟨hole_defs⟩
                           '|'⟨hole_defs⟩
                           ']'
⟨shape_def⟩       ::= judgeuse [⟨target⟩]⟨rexpr⟩
                    | judgespec⟨rexpr⟩
⟨target⟩          ::= tex | ag | ...
⟨hole_defs⟩       ::= [thread]⟨hole_nm⟩':'⟨hole_type⟩
⟨hole_type⟩       ::= ⟨nm⟩
⟨rules_def⟩       ::= rules⟨nm⟩scheme⟨scm_nm⟩"info"
                        '='⟨rule_def⟩ *
⟨rule_def⟩        ::= rule⟨nm⟩[⟨ag_nm⟩] = ⟨rl_view_def⟩ *
⟨rl_view_def⟩     ::= view⟨vw_nm⟩
                        '='⟨judge_rexpr⟩ *
                           '-'
                           ⟨judge_rexpr⟩
```

Figure 11.9.: Syntax of ruler notation (part I)

All views of a rule are jointly defined, although we present the various views separately throughout this chapter. We will come back to this in our discussion.

Each view for a rule specifies premises and a conclusion, separated by a '-'. The rule E.INT for integer constants only has a single judgement for the conclusion. The judgement has name *R*, is of scheme *expr*, and is specified using the **spec** judgement shape for this view. The name of the judgement is used to refer to the judgement from later views, either to overwrite it completely or to adapt the values of the holes individually. In the latter case the hole values of the previous view which are not adapted are kept. Later, when we introduce subsequent views we will see examples of this.

The rule for integer constants refers to *Ty_Int*. This is an identifier which is not introduced as part of the rule. and its occurrence generates an error message unless we specify it to be external:

external *Ty_Int*

Additionally we also have to specify the way *Ty_Int* will be typeset as *Ruler* does not make any assumptions here. *Ruler* outputs identifiers as they are and delegates formatting to

```
⟨judge_rexpr⟩   ::= judge [⟨nm⟩':'](⟨scm_nm⟩
                      ( '='⟨rexpr⟩
                       | ('|'⟨hole_nm⟩'='⟨rexpr⟩) *
                      )
⟨rexpr⟩         ::= ⟨rexpr_app⟩⟨op⟩⟨rexpr⟩ | ⟨rexpr_app⟩
⟨rexpr_app⟩     ::= ⟨rexpr_app⟩⟨rexpr_base⟩ | ⟨rexpr_base⟩ | ε
⟨rexpr_base⟩    ::= ⟨nm⟩ | ⟨rexpr_parens⟩ | unique
                  | `=` | `|` | `.` | `-`
                  | int | "string"
⟨rexpr_parens⟩  ::= '(' ( ⟨rexpr⟩
                          | ⟨rexpr⟩'|'⟨hole_type⟩
                          | node int = ⟨rexpr⟩
                          | text "string"
                          | ('|' | '.' | '=' | '-' | ⟨keyword⟩) *
                          )
                    ')' ('.'⟨rexpr_base⟩) *
⟨op⟩            ::= ⟨op_base⟩('.'⟨rexpr_base⟩) *
⟨op_base⟩       ::= ('!#$%&*+/<=>?@\^|-:;,[]{}~') *
                      − ('|' | '.' | '=' | '-')
⟨viewhierarchy_def⟩
                ::= viewhierarchy⟨vw_nm⟩('<'⟨vw_nm⟩) *
⟨format_def⟩    ::= format [⟨target⟩]
                      ⟨nm⟩'='⟨rexpr⟩
⟨rewrite_def⟩   ::= rewrite [⟨target⟩] [def | use]
                      ⟨rexpr⟩'='⟨rexpr⟩
⟨ag_nm⟩, ⟨scm_nm⟩, ⟨vw_nm⟩, ⟨hole_nm⟩
                ::= ⟨nm⟩
⟨nm⟩            ::= ⟨nm_base⟩('.' (⟨nm_base⟩ | int)) *
⟨nm_base⟩       ::= 'a-zA-Z_' 'a-zA-Z_0-9' *
⟨keyword⟩       ::= (scheme | ...) − (unique)
⟨external_def⟩  ::= external⟨nm > *
```

Figure 11.10.: Syntax of ruler notation (part II)

lhs2TeX [72]. A simple renaming facility however is available as some renaming may be necessary, depending on the kind of output generated. Formatting declarations introduce such renamings:

format tex *Ty_Int = Int*

Here the keyword **tex** specifies that this renaming is only used when LaTeX (i.e. the **tex** target) is generated. The formatting for the names *gam* and *ty* are treated similarly.

The rule E.APP for the application of a function to an argument is defined similarly to

rule E.INT. Premises now relate the type of the function and its argument:

```
rule e.app =
  view E =
    judge A : expr = gam ⊢ a : ty.a
    judge F : expr = gam ⊢ f : (ty.a → ty)
    –
    judge R : expr = gam ⊢ (f a) : ty
```

which results in (from Fig. 11.1):

$$\frac{\Gamma \vdash^e a : \tau_a \quad \Gamma \vdash^e f : \tau_a \to \tau}{\Gamma \vdash^e f\ a : \tau}\ \text{E.APP}_E$$

The dot notation allows us to treat `ty.a` as a single identifier, which is at the same time rendered as the subscripted representation τ_a. Also note that we parenthesize $(ty.a \to ty)$ such that *Ruler* treats it as a single expression. The outermost layer of parentheses are stripped when an expression is matched against a judgement shape.

Relations: external schemes The rule E.VAR for variables is less straightforward as it requires premises which do not follow an introduced scheme:

$$\frac{i \mapsto \sigma \in \Gamma \quad \tau = inst\ (\sigma)}{\Gamma \vdash^e i : \tau}\ \text{E.VAR}_E$$

This rule requires a binding of the variable i with type σ to be present in Γ; the instantiation τ of σ then is the type of the occurrence of i. These premises are specified by judgements G and I respectively:

```
rule e.var =
  view E =
    judge G : gamLookupIdTy = i ↦ pty ∈ gam
    judge I : tyInst = ty '=' inst (pty)
    –
    judge R : expr = gam ⊢ i : ty
```

Judgements G and I use a variation of a scheme, called a *relation*. For example, the judgement G must match the template for relation *gamLookupIdTy* representing the truth of the existence of an identifier i with type ty in a *gam*:

```
relation gamLookupIdTy =
  view E =
```

holes [| $nm : Nm, gam : Gam, ty : Ty$ |]
judgespec $nm \mapsto ty \in gam$

A relation differs only from a scheme in that we will not define rules for it. It acts as the boundary of our type rule specification. As such it has the same role as the foreign function interface in Haskell (or any other programming language interfacing with an outside world). As a consequence we have to specify an implementation for it elsewhere. The relation *tyInst* is defined similarly:

relation *tyInst* =
  view E =
    holes [| $ty : Ty, ty.i : Ty$ |]
    judgespec $ty.i$ `=` $inst\ (ty)$

11.5 Extending to an algorithm

In this section we demonstrate the usefulness of views and incremental extension by adapting the equational rules from Fig. 11.1 to the algorithmic variant in Fig. 11.2. We call this the A view. We only need to specify the differences between two views. This minimises our specification work; *Ruler* emphasises the differences using color. The resulting type rules are shown in Fig. 11.2.

Fig. 11.2 not only shows the adapted rules but also shows the differences with the previous view by using colors. In the electronic version of this thesis the unchanged parts of the previous view (E) are shown in grey, whereas the changed parts are shown in black (blue, if seen in color). The paper version typesets the rules in black for better readability. In our opinion, clearly indicating differences while still maintaining an overview of the complete picture, contributes to the understandability of the type rules when the complexity of the rules increases.

For this to work, we specify which view is built on top of which other view:

viewhierarchy = $E\langle A\langle AG$

The view hierarchy declaration defines the A view to be built on top of view E, and AG again on top of A. We can also specify branches, for example $E\langle X$ specifies X to be built on top of E, independently of other views; because we do not use this feature, will will not discuss it further. A view inherits the hole structure and the judgement shapes from its predecessor. Similarly, for each rule the bindings of hole names to their values are preserved as well. As a consequence we only have to define the differences.

In order to turn the equational specification into an algorithmic one based on HM type inference, we need to:

- Specify the direction in which values in the holes flow through a rule. This specifies the computation order.

- Represent yet unknown types by type variables and knowledge about those type variables by constraints.

Both modifications deserve some attention, because they are both instances of a more general phenomenon which occurs when we shift from the equational to the algorithmic realm: we need to specify a computation order.

From relationships to functions In an equational view we simply relate two values. In an algorithmic view this relation is replaced by a function mapping input values to output values. For example, rule E.APP from Fig. 11.1 specifies that the type of a and the argument part of the type of f must be equal. The use of the same identifier τ_a expresses this equality. To compute τ_a however we either need to:

- compute information about a's type first and use it to construct f's type,
- compute information about f's type first and use it to deconstruct and extract a's type,
- compute information about both and then try to find out whether they are equal (or remember they should be equal).

The last approach is taken for hole *ty*, because it allows us to compute types compositionally in terms of the types of the children of an *Expr*.

Using yet unknown information In an equational view we simply use values without bothering about how they are to be computed. However, computation order and reference to a value may conflict if we to refer to a value before its value is computed. For example, rule E.LET allows reference to the type of i (in e) before its type has been computed. In rule E.LET the type of i is available only after HM's generalisation of the type of a let-bound variable. The standard solution to this problem is to introduce an extra indirection by letting the type of i be a placeholder, called a type variable. Later, if and when we find more information about this type variable, we gather this information in the form of constraints, which is the information then used to replace the content of the placeholder.

Adding direction to holes In *Ruler* notation, we specify the direction of computation order as follows for view A on scheme *expr*:

view A =
  holes [$e : Expr, gam : Gam$ | **thread** $cnstr : C$ | $ty : Ty$]
  judgespec $cnstr.inh; gam \vdash$ ..`"e"` $e : ty \rightsquigarrow cnstr.syn$
  judgeuse – tex

The holes for *expr* are split into three groups, separated by vertical bars '|'. Holes in the first group are called *inherited*, holes in the third group are called *synthesized* and the

holes in the middle group are both. The type rules now translate to a syntax directed computation over an abstract syntax tree (AST). Values for inherited holes are computed in the direction from the root to the leaves of the AST providing contextual information; values for synthesized holes are computed in the reverse order providing a result. We will come back to this in following sections.

In our *A* view on scheme *expr* both *e* and *gam* are inherited, whereas *ty* is the result. This, by convention, corresponds to the standard visualisation of a judgement in which contextual information is positioned at the left of the turnstyle '⊢' and results are placed after a colon ':'. As we will see, the hole *e* plays a special role because it corresponds to the AST.

Besides being declared as both an inherited and a synthesized hole, *cnstr* is also declared to be *threaded*, indicated by the keyword **thread**. For a threaded hole its computation proceeds in a specific order over the AST, thus simulating a global variable. For now it suffices to know that for a threaded hole *h* two other holes are introduced instead: *h.inh* for the inherited value, *h.syn* for the synthesized value. Because *cnstr* is declared threaded, *cnstr.inh* refers to the already gathered information about type variables, whereas this and newly gathered information is returned in *cnstr.syn*. For example, view *A* on rule E.INT fills *cnstr.syn* with *cnstr.inh*.

```
view A =
  –
  judge R : expr
    | cnstr.syn = cnstr.inh
    | cnstr.inh = cnstr.inh
```

Although a definition for *cnstr.inh* is included, we may omit the hole binding for *cnstr.inh*, that is *cnstr.inh* = *cnstr.inh* (we will do this in the remainder of this chapter). If a binding for a new hole is omitted, the hole name itself is used as its value.

Instead of using a shape to specify the rule, we may bind individual hole names to their values. In this way we only need to define the holes which are new or need a different value. The *Ruler* system also uses this to highlight the new or changed parts and grey out the unchanged parts. This can be seen from the corresponding rule from Fig. 11.2 (value *cnstr.inh* shows as C^k by means of additional formatting information):

$$\frac{}{C^k; \Gamma \vdash^e \mathit{int} : \mathit{Int} \rightsquigarrow C^k}\ \text{E.INT}_A$$

For rule E.APP both the handling of the type (hole *ty*) and the constraints need to be adapted. The type *ty.a* of the argument is used to construct $ty.a \rightarrow tv$ which is matched against the type *ty.f* of the function. Constraints are threaded through the rules. For example constraints *cnstr.f* constructed by the judgement for the function *f* are given to the judgement *a* in the following fragment (which follows view *E* of rule E.APP in the *Ruler* source text):

```
view A =
```

```
judge V : tvFresh = tv
judge M : match  = (ty.a → tv) ≅ (cnstr.a ty.f)
                        ⇝ cnstr
judge F : expr
    | ty       = ty.f
    | cnstr.syn = cnstr.f
judge A : expr
    | cnstr.inh = cnstr.f
    | cnstr.syn = cnstr.a
 –
judge R : expr
    | ty       = cnstr cnstr.a tv
    | cnstr.syn = cnstr cnstr.a
```

The rule E.APP also requires two additional judgements: a *tvFresh* relation stating that *tv* should be a fresh type variable and a *match* relation performing unification of two types, resulting in additional constraints under which the two types are equal. The resulting rule (from Fig. 11.2) thus becomes:

$$\frac{\begin{array}{c} C^k;\Gamma \vdash^e f : \tau_f \rightsquigarrow C_f \\ C_f;\Gamma \vdash^e a : \tau_a \rightsquigarrow C_a \\ v \text{ fresh} \\ \tau_a \to v \cong C_a\tau_f \rightsquigarrow C \end{array}}{C^k;\Gamma \vdash^e f\ a : C\ C_a v \rightsquigarrow C\ C_a}\ \text{E.APP}_A$$

The way this rule is displayed also demonstrates the use of the inherited or synthesized direction associated with a hole for ordering judgements. The value of a hole in a judgement is either in a position where the identifiers of the value are introduced for use elsewhere or in a position where the identifiers of a value are used:

- A synthesized hole corresponds to a result of a judgement. Its value specifies how this value can be used; it specifies the pattern it must match. This may be a single identifier or a more complex expression describing the decomposition into the identifiers of the hole value. For example, *cnstr.f* in the premise judgement *F* for function *f* is in a so called *defining* position because it serves as the value of a hole which is defined as synthesized.
- For an inherited hole the reverse holds: the hole corresponds to the context of, or parameters for, a judgement. Its value describes the composition in terms of other identifiers introduced by values at defining positions. For example, *cnstr.f* in the judgement *A* for argument *a* is in a so called *use* position because its hole is inherited.
- For the concluding judgement the reverse of the previous two bullets hold. For example, *cnstr.inh* of the conclusion judgement *R*, implicitly defined as *cnstr.inh* =

cnstr.inh, is on a defining position although its hole is inherited. This is because it is given by the context of the type rule itself, for use in premise judgements.

Ruler uses this information to order the premise judgements from top to bottom such that values in holes are defined before used. Because judgements may be mutually dependent this is done in the same way as the binding group mechanism of Haskell: the order in a group of mutually dependent judgements cannot be determined and therefore is arbitrary.

Relation *match* represents the unification of two types; it is standard. Relation *tvFresh* simply states the existence of a fresh type variable; we discuss its implementation in Section 11.6.

11.6 Extensions for AG code generation

In this section we discuss the modifications to our type rule specification required for the generation of a partial implementation, and the additional infrastructure required for a working compiler. The end result of this section is a translation of type rules to AG code. For example, the following is generated for rule E.APP; the required additional *Ruler* specification and supporting code is discussed in this section:

```
attr Expr [g : Gam | c : C | ty : Ty]
sem Expr
  | App (f.uniq, loc.uniq1)
                = rulerMk1Uniq @lhs.uniq
        loc.tv_ = Ty_Var @uniq1
        (loc.c_, loc.mtErrs)
                = (@a.ty `Ty_Arr` @tv_) ≅ (@a.c ⊕ @f.ty)
        lhs.c   = @c_ ⊕ @a.c
           .ty  = @c_ ⊕ @a.c ⊕ @tv_
```

We need to deal with the following issues:

- Type rules need to be translated to AG code that describes the computation of hole values. We exploit the similarity between type rules and attribute grammars to do this.
- Fresh type variables require a mechanism for generating unique values.
- Type rules are positive specifications, but do not specify what needs to be done in case of errors.
- Of course we also need to specify parsing to an AST as well as output generation, but we won't treat this here.

Type rule structure and AST structure The structure of type rules and an abstract syntax tree are often very similar. This should come as no surprise, because type rules are usually syntax directed in their algorithmic form so the choice which type rule to apply can be made deterministically. We need to tell *Ruler*:

- Which hole of a scheme acts as a node from the AST, the *primary hole*.
- Which values in this primary hole in the conclusion of a rule are children in the AST.
- To which AG **data** a scheme maps, and for each rule to which alternative.

The AST is defined externally relative to *Ruler* (this may change in future versions of *Ruler*). For example, the part of the AST for expression application is defined as:

```
data Expr
  | App f : Expr
        a : Expr
```

The keyword **node** is used to mark the primary hole that corresponds to the AST node for scheme *expr* in the AST:

```
view AG =
  holes [node e : Expr ||]
```

For each rule with children we mark the children and simultaneously specify the order of the children as they appear in the AST. For example, for rule E.APP we mark f to be the first and a to be the second child (the ordering is required for AG code generation taking into account AG's copy rules):

```
view AG =
  –
  judge R : expr
    | e = ((node 1 = f) (node 2 = a))
```

The scheme *expr* is mapped to the AST node type *Expr* by adapting the scheme definition to:

```
scheme expr "Expr" =
```

Similarly we adapt the header for rule E.APP to include the name *App* as the name of the alternative in the AST:

```
rule e.app "App" =
```

Ruler **expressions and AG expressions** Expressions in judgements are defined using a notation to which *Ruler* attaches no meaning. In principle, the *Ruler* expression defined for a hole is straightforwardly copied to the generated AG code. For example, for rule E.APP the expression $ty.a \rightarrow tv$ would be copied, including the arrow $\rightarrow$. Because AG attribute definitions are expressed in Haskell, the resulting program would be incorrect without any further measures taken.

Ruler uses rewrite rules to rewrite ruler expressions to Haskell expressions. For example, $ty.a \rightarrow tv$ must be rewritten to a Haskell expression representing the meaning of the *Ruler* expression. We define additional Haskell datatypes and functions to support the intended meaning; unique identifiers *UID* are explained later:

```
type TvId = UID
data Ty   = Ty_Any | Ty_Int | Ty_Var TvId
          | Ty_Arr Ty Ty
          | Ty_All [TvId] Ty
            deriving (Eq, Ord)
```

A *Ty_All* represents universal quantification $\forall$, *Ty_Arr* represents the function type $\rightarrow$, *Ty_Var* represents a type variable and *Ty_Any* is used internally after an error has been found (we come back to this later). We define a rewrite rule to rewrite $ty.a \rightarrow tv$ to $ty.a$ `` `Ty_Arr` `` tv:

```
rewrite ag def a → r = (a) `Ty_Arr` (r)
```

A rewrite declaration specifies a pattern (here: $a \rightarrow r$) for an expression containing variables which are bound to the actual values of the matching expression. These bindings are used to construct the replacement expression (here: (a) `` `Ty_Arr` `` (r)). The target **ag** limits the use of the rewrite rule to code generation for AG. The flag **def** limits the use of the rule to defining positions, where a *defining position* is defined as a position in a value for an inherited hole in a premise judgement or a synthesized hole in a conclusion judgement. This is a position where we construct a value opposed to a position where we deconstruct a value into its constituents. Although no example of deconstructing a value is included in this chapter, we mention that in such a situation a different rewrite rule expressing the required pattern matching (using AG language constructs) is required. The flag **use** is used to mark those rewrite rules.

The rewrite rule used for rewriting $ty.a \rightarrow tv$ actually is limited further by specifying the required type of the value for both pattern and the type of the replacement pattern:

```
rewrite ag def (a | Ty) → (r | Ty) = ((a) `Ty_Arr` (r) | Ty)
```

The notion of a type for values in *Ruler* is simple: a type is just a name. The type of an expression is deduced from the types specified for a hole or the result expression of a rewrite rule. This admittedly crude mechanism for checking consistency appears to work quite well in practice.

Limiting rewrite rules based on *Ruler* type information is useful in situations where we encounter overloading of a notation; this allows the use of juxtapositioning of expressions to keep the resulting expression compact. We can then specify different rewrite rules based on the types of the arguments. The meaning of such an expression usually is evident from its context or the choice of identifiers. For example, *cnstr cnstr.a tv* (rule E.APP, Fig. 11.2) means the application of constraints *cnstr* and *cnstr.a* as a substitution to type *tv*. Constraints can be applied to constraints as well, similar to Haskell's overloading. To allow for this flexibility a pattern of a rewrite rule may use (*Ruler*) type variables to

propagate an actual type. For example, the rewrite rule required to rewrite *cnstr cnstr.a tv* is defined as:

```
rewrite ag def (c1 | C) (c2 | C) (v | a)
                  = (c1 ⊕ c2 ⊕ (v) | a)
```

Rewrite rules are only applied to saturated juxtapositionings or applications of operators. Rewrite rules are non-recursively applied in a bottom-up strategy.

The rule assumes the definition of additional Haskell types and class instances defined elsewhere:

```
type C = [(TvId, Ty)]

class Substitutable a where
  (⊕) :: C → a → a
  ftv  :: a → [TvId]

instance Substitutable Ty where
  s ⊕ t @(Ty_Var v) = maybe t id (lookup v s)
  s ⊕ Ty_Arr t1 t2  = Ty_Arr (s ⊕ t1) (s ⊕ t2)
  _ ⊕ t             = t
  ftv (Ty_Var v)    = [v]
  ftv (Ty_Arr t1 t2) = ftv t1 ∪ ftv t2
  ftv _             = [ ]
```

Unique values Our implementation of "freshness", required for fresh type variables, is to simulate a global seed for unique values. The global seed is implemented by a threaded attribute *uniq*; we have omitted its declaration and initialisation. *Ruler* assumes that such an implementation is provided externally. From within *Ruler* we use the keyword **unique** to obtain a unique value. For example, the relation *tvFresh* has a **ag** judgement shape for the generation of AG which contains a reference to **unique**:

```
relation tvFresh =
  view A =
    holes [|| tv : Ty]
    judgespec tv
    judgeuse tex tv (text "fresh")
    judgeuse ag tv `=` Ty_Var unique
```

AG code generation inlines the judgement using the **ag** judgement shape:

```
sem Expr
  | App (f.uniq, loc.uniq1)
              = rulerMk1Uniq @lhs.uniq
        loc.tv_ = Ty_Var @uniq1
        (loc.c_, loc.mtErrs)
              = (@a.ty `Ty_Arr` @tv_) ≅ (@a.c ⊕ @f.ty)
```

$$\begin{aligned}\textbf{lhs}.c &= @c\_ \oplus @a.c \\ .ty &= @c\_ \oplus @a.c \oplus @tv\_\end{aligned}$$

The presence of **unique** in a judgement for a rule triggers the insertion of additional AG code to create an unique value and to update the unique seed value. *Ruler* automatically translates the reference to **unique** to *uniq1* and inserts a call to *rulerMk1Uniq*. The function *rulerMk1Uniq* is assumed to be defined externally. It must have the following type:

$$\begin{aligned}&rulerMk1Uniq :: \langle X\rangle \to (\langle X\rangle, \langle Y\rangle) \\ &rulerMk1Uniq = \ ...\end{aligned}$$

For $\langle X\rangle$ and $\langle Y\rangle$ any suitable type may be chosen, where $\langle X\rangle$ is restricted to match the type of the seed for unique values, and $\langle Y\rangle$ matches the type of the unique value. Our default implementation is a nested counter which allows a unique value itself to also act as a seed for an unlimited series of unique values. This is required for the instantiation of a quantified type where the number of fresh type variables depends on the type (we do not discuss this further):

$$\begin{aligned}&\textbf{newtype}\ UID = UID\ [Int]\ \textbf{deriving}\ (Eq, Ord) \\ &uidStart = UID\ [0] \\ &rulerMk1Uniq :: UID \to (UID, UID) \\ &rulerMk1Uniq\ u\ @(UID\ ls) = (uidNext\ u, UID\ (0 : ls)) \\ &uidNext :: UID \to UID \\ &uidNext\ (UID\ (l : ls)) = UID\ (l + 1 : ls)\end{aligned}$$

When a rule contains multiple occurrences of **unique**, *Ruler* assumes the presence of $rulerMk\langle n\rangle Uniq$ which returns $\langle n\rangle$ unique values; $\langle n\rangle$ is the number of **unique** occurrences.

The *Ruler* code for relation *tvFresh* also demonstrates how the **ag** judgement shape for *tvFresh* is inlined as an attribute definition. The **ag** shape for a relation must have the form $\langle attrs\rangle$ '='$\langle expr\rangle$.

Handling errors The generated code for rule E.APP also shows how the implementation deals with errors. This aspect of an implementation usually is omitted from type rules, but it cannot be avoided when building an implementation for those type rules. Our approach is to ignore the details related to error handling in the LaTeX rendering of the type rules, but to let the generated AG code return two values at locations where an error may occur:

- The value as defined by the type rules. If an error occurs, this is a "does not harm" value. For example, for types this is *Ty_Any*, for lists this is an empty list.
- A list of errors. If no error occurs, this list is empty.

For example, the AG code for relation *match* as it is inlined in the translation for rule E.APP is defined as:

relation *match* =

```
view A =
  holes [ty.l : Ty, ty.r : Ty || cnstr : C]
  judgespec ty.l ≅ ty.r ⇝ cnstr
  judgeuse ag (cnstr, mtErrs) '='(ty.l) ≅ (ty.r)
```

The operator ≅ implementing the matching returns constraints as well as errors. The errors are bound to a local attribute which is used by additional AG code for error reporting.

11.7 Discussion, related work, conclusion

Experiences with *Ruler* *Ruler* solves the problem of maintaining consistency and managing type rules; it is a relief to avoid writing LaTeX for type rules by hand and to know that the formatted rules correspond directly to their implementation.

Ruler enforces all views on a type rule to be specified together. This is a consequence of our design paradigm in which we both isolate parts of the type rules specification (by using views), and need to know the context of these isolated parts (by rendering parts together with their context). As a developer of a specification all views can best be developed together, to allow for a understandable partitioning into different views while at the same time keeping an overview.

Literate programming Literate programming [6, 59] is a style of programming where the program source text and its documentation are combined into one document. So called *tangling* and *weaving* tools extract the program source and documentation. Our *Ruler* system is different:

- Within a literate programming document program source and documentation are recognizable and identifiable artefacts. In *Ruler* there is no such distinction.
- *Ruler* does not generate documentation; instead it generates fragments for use in documentation.

We think *Ruler* is mature enough to be used by others, and we are sure such use will be a source of new requirements. Since *Ruler* itself has been produced using the AG system new extensions can be relatively easily incorporated.

Emphasizing differences We use colors to emphasize differences in type rules. For black-and-white print this is hardly a good way to convey information to the reader. We believe however that in order understand more complex material, more technical means (like colors, hypertext, collapsable/expandable text) must be used to express and explain the complexity.

Future research We foresee the following directions of further research and development of *Ruler*(see Chapter 12 for a further elaboration):

- The additional specification required to shift from equational to algorithmic type rules is currently done by hand. However, our algorithmic version of the type rules uses a heuristic for dealing with yet unknown information and finding this unknown information. We expect that this (and other) heuristics can be applied to similar problems as an automated strategy.
- *Ruler* currently generates output for two targets: LaTeX and AG. We expect the *Ruler* to be useful in many different situations, requiring different kinds of output, such as material for use in theorem provers.

Related work TinkerType [68], used for Pierce's book [91], comes closest to *Ruler*. Type system features can be combined into type systems. The system provides checks for valid combinations, and allows the specification of (implementing) code fragments. However, no automatic code generation is supported, nor checks on the structure of judgements.

The theorem proving environment Twelf [100] is used to describe and proof properties for programming languages [34], thus answering the POPLmark challenge [8]. Although we intend to generate descriptions for use in such theorem proving tools, we emphasize that *Ruler* is meant as a lightweight tool for the construction of compilers.

Various AST based compiler construction tools exist [1, 4, 109, 31], among which our AG system. Such tools have in common that they only allow programming on the level of AST's, whereas *Ruler* allows a higher level programming. Furthermore, in our experience, stepwise AG descriptions became too complex (hence the use of *Ruler*), and we expect this the case for similar formalisms as well.

Finally, we also mention the Programmatica project, which provides mechanisms and tools for proving properties of Haskell programs.

12 Conclusion and future work

In the introduction (Section 1.1, page 2) we listed the following research goals:

- **EH, explanation and presentation:** A compiler for Haskell, plus extensions, fully described for use in education and research.
- **EH, use of explicit and implicit type information:** A better cooperation between (type) information specified by a (EH) programmer and (type) information inferred by the system.
- **Partitioning and complexity:** A stepwise approach to explaining a compiler (or similar large programs).
- **Consistency:** An organisation of material which guarantees consistency by construction (a priori), instead of by comparison (a posteriori).

We discuss these goals in the following sections. Although not listed as a goal, we also look into the following list of topics, as these constitute a large part of this thesis, or are related otherwise:

- **EH, formal properties:** EH (formal) properties.
- **EH, relation to Haskell:** EH and Haskell.
- **AG experience:** Our experience with the AG system.

A high level conclusion is that the key to an understandable and maintainable description of a working compiler lies in the use of domain specific languages for dealing with the complexity of such a compiler. Although traditionally many compilers are built without the use of tools like the AG system, we feel that traditional techniques have reached their limit, especially in dealing with mature implementations (like GHC (Glasgow Haskell Compiler)) of complex languages such as Haskell. In the following sections we detail this conclusion to more specific aspects.

12.1 EH, explanation and presentation

Discussion Our approach to the explanation of EH shares the focus on implementation with the approach taken by Jones [48], and shares the focus on the combination of formalities and implementation with Pierce [91].

However, there are also differences:

- Our focus is on the implementation, which is described by means of various formalisms like type rules and Attribute Grammar notation.
- Our compilers are complete, in the sense that all aspects of a compiler are implemented. For example, parsing, error reporting and code generation (in later EH versions) are included.
- We use type rules as a specification language for the implementation of a compiler; type rules directly specify the implementation.

Conciseness suffers when features are combined, because a short, to the point, presentation of an issue can only be constructed thanks to a simplification of that issue, and the reduction to its bare essentials. On the other hand, a treatment of feature interaction usually suffers, because of the simplifications required to understand a feature.

The problem is that the understanding of a language feature requires simplicity and isolation, whereas the practical application of that feature requires its co-existence with other features: these are contradictory constraints. This is a problem that will never be resolved, because language features are to be used as a solution to a real programmer's need (that is, a wider context); a need that can only be understood if stripped of unrelated issues (that is, a restricted context).

Although the problem of understanding language features, both in isolation and context, cannot be resolved, we can make it more manageable:

- Specify and explain in terms of differences. This is the approach taken by this thesis. The AG system allows separation into attributes, the *Ruler* system and fragment manipulation tool *Shuffle* specify in terms of views. All tools provide a simple mechanism to combine the separated parts: redefinition of values associated with an attribute (or a similar identification mechanism).
- If separated parts influence each other in combination, then we specify their combination. This is the place where an ordering of separate parts becomes important. Our tools can be parameterized with an ordering (of views). If separate parts do not influence each other, they can co-exist as (conceptually) separate computations.
- Emphasize differences in their context. Visual clues as to what has changed and what remained the same, help to focus on the issue at hand, while not forgetting its context. The *Ruler* tool uses colors to accomplish this. However, this has its limits,

for example Fig. 7.16, page 119 and Fig. 7.17, page 120 are likely to require a bit of study.

- Hide 'irrelevant' material. This thesis hides a fair amount of code, making it only indirectly available. This is not a satisfactory solution, because the need to expose hidden information depends on the reader's frame of reference, understanding and knowledge.
- Render non-linearly. A paper version of an explanation is one-dimensional, understanding often is non-linear and associative.

Most of the abovementioned measures facilitate understanding. However, their realisation requires alternate ways of browsing through material, supported by proper tool(s). Existing mechanisms for abstraction and encapsulation, like module systems, can help a great deal, but in our experience a single feature often requires changes crossing traditional encapsulation boundaries. Crucial to the success of such a tool would be, in our opinion, the handling of feature isolation and feature combination.

This thesis and future work We summarize our contribution and future work:

- This thesis: type system specification and implementation are jointly presented.
- Future work: the type system is algorithmically specified; a declarative specification would benefit understanding but its relationship with the implementation would be less clear. Our intended approach is to incorporate strategies into *Ruler* to be able to automate the translation from declarative to algorithmic specification (see also Section 12.4).
- This thesis: a stepwise description and implementation.
- Future work: a step is described relative to a previous step; splitting such a step into (1) an isolated feature description which is (2) explicitly combined with a particular step would allow for further modularisation of features.

12.2 EH, use of explicit and implicit type information

Discussion EH supports a flexible use of explicitly specified type information:

- Type signatures are exploited as much as possible by employing several strategies for the propagation to the place (in the AST) where they needed.
- Type signatures may be partially specified. This gives the "all or nothing" type specification requirement of Haskell a more (user-)friendly face, by a allowing a gradual shift between explicit and implicit.

Furthermore, EH supports a flexible use of implicit parameters by allowing explicit parameter passing for implicit parameters and by allowing explicitly introduced program values to be used implicitly for an implicit parameter.

Our approach to the interaction between explicit (higher-ranked) types and type inference is algorithmic. This is similar to the algorithmic approach taken by Rémy [95], but also lacks a clear type theoretical characterisation of the types which we can infer. However, our informal claim is that the notion of "touched by" a quantified type provides an intuitive invariant of our algorithm. In practice, we expect this to be sufficiently predictive for a programmer.

The central idea is to allow a programmer to gradually shift between full explicit type signature specification to implicitly inferred types. We can apply the 'gradual shift' design starting point to other parts of the language as well. For example, the instantiation of type variables with types now is done implicitly or indirectly. We may well allow explicit notation a la System F, instead of the more indirect means by specifying (separate) type signatures and/or type annotations.

We have also implemented explicit kind signatures, but did not discuss these in this thesis.

This thesis and future work We summarize our contribution and future work:

- This thesis: allow a more flexible use and integration (into the type inferencer) of explicit (type) information.
- Future work: allow explicit specification of other implicit elements of the language (e.g. System F like notation).
- This thesis: exploitation of type annotations by means of (global) quantifier propagation.
- Future work: formalise quantifier propagation.

12.3 Partitioning and complexity

Discussion In this thesis we have chosen a particular partitioning of the full EH system into smaller and ordered steps. Although we feel that the chosen partitioning serves the explanation well, others are possible and perhaps better within different contexts.

Fig. 12.1 intuitively illustrates where we have to make choices when we partition. We can partition along two dimensions, one for the language constructs (horizontally), and one for the semantic aspects (vertically). For example, in Chapter 3 we fill in the light grey zone, ending in Chapter 9 by filling in the dark grey zone. The horizontal dimension corresponds to AST extensions, the vertical dimension corresponds to attributes and their (re)definition. Each step describes one or more polygon shaped areas from the two-dimensional space. Partitioning means selecting which squares are picked for extension and description.

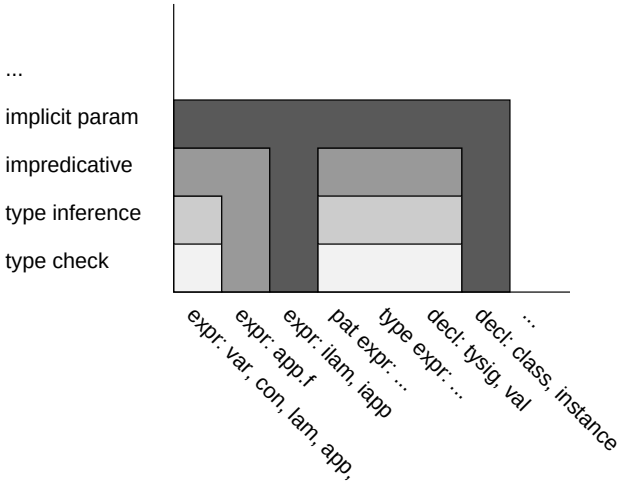

Figure 12.1.: AST and aspects

Fig. 12.1 also shows that complexity increases with the addition of features. Not only do we have to describe new features, but we also have to adapt the semantics of previous features as well, hopefully minimizing their interdependencies. Most likely the partitioning has the least interdependencies; this is similar to proper module design.

Extending EH is not simply a matter of adding a new module. An EH extension requires changes to the AST, attributes, semantics for attributes, supporting Haskell functions and type rules. It is unclear how to capture such a group of changes by a module-like mechanism.

This thesis and future work We summarize our contribution and future work:

- This thesis: a partitioning of EH into steps, with tool support (either already existing (AG) or new (*Shuffle*, *Ruler*)).
- Future work: what is a good partitioning, and which mechanisms support the partitioning required for the description and construction of a compiler.

12.4 Consistency

Discussion Our approach to maintaining consistency between material used for explanation and implementation, is to avoid inconsistencies in the first place. Inconsistencies are introduced when two (or more) artefacts represent derived information, but are treated as independent, non-related pieces of information. A change in such an artefact, or the artefact derived from, not accompanied by a corresponding (correct) change in the remaining related artefacts, results in an inconsistency.

The dangers of inconsistency are therefore twofold:

- Changes are not propagated.
- Changes are made in related artefacts, but the changes are incorrect, that is, inconsistent.

This is a general problem. In our particular case we consider it a problem for:

- Implementation (program code) and its incorporation into explanation.
- Formal representation of type rules, and their implementation (AG code).

The first combination often is handled by "copy and paste" of code. For a one-shot product this usually is not a large problem, but for long-lived products it is. A solution to this problem consists of mechanisms for sharing text fragments. Both our tool *Shuffle* and similar literate programming tools [6] have in common that a shared text fragment is associated with some identification by which the shared text fragment can be included at different places.

The second combination suffers the fate of many specifications: they are forgotten after being implemented. This is also a general problem, because non-automated translation between artefacts is involved: human translations are seldom flawless. This, of course, also holds for the artefacts themselves, but translation at least can be automated, provided a well-defined semantics of the artefacts and their relation.

Type rules and AG implementation correspond to such a degree that it is possible to translate from a common type rule description to an implementation and a visual rendering, which can be included in text dealing with the formal aspects of type rules. This is the responsibility of the *Ruler* tool, which already in its first version turned out to be indispensable for the construction of this thesis. Actually, the *Ruler* tool started out to reduce the amount of work involved in typesetting type rules, soon to be extended to generate AG code when we were confronted with the amount of work required to keep those type rules consistent with their AG implementation.

The strong point of a tool like *Ruler* is that it acts, like any compiler, as a substitute for the proof that implementation and type rules describe the same semantics. And, like any compiler, optimisations can be performed. We foresee that a tool like *Ruler* can deal with

aspects of the translation from type rule specification to implementation, some of which are done manually in this thesis:

- In this thesis, equational type rules are implemented by algorithmic ones, which easily map to AG rules. The transition from equation to algorithm involves a certain strategy. In this thesis we use HM inference, a greedy resolution of constraints. Alternate strategies exist [37, 36]; *Ruler* (or similar tools) can provide abstractions of such strategies.
- Strategies can be user defined. This thesis uses a representation of yet unknown information (type variable), a representation of found information (constraints) and combinatorial behavior (application of constraints as a substitution, type matching, AST top-down/bottom-up propagation). These, and similar aspects, may well form the building blocks of strategies.
- This thesis often uses multiple passes over an AST. For example, Chapter 7 describes a two-step inference for finding polymorphic type information. A more general purpose variant of this strategy would allow categorisation of found information, where each pass would find information from a particular category.
- *Ruler* exploits the syntax-directed nature of type rules. This means that the structure of an AST determines which rule has to be used. The choice of the right rule may also depend on other conditions (than the structure of the AST), or a choice may be non-deterministic. The consequence of this observation is that *Ruler* has to deal with multiple levels of rules, transformed into eachother, with the lowest level corresponding to an AST based target language.
- *Ruler* uses AG as its target language. In thesis, the rules for type matching (e.g. Fig. 3.8, page 43), are also syntax-directed, but base their choice on two AST's (for types), instead of one. This is a special, but useful, case of the previous item.
- *Ruler* extends views on type rules by adding new computations, expressed in terms of holes in a judgement. The final version combines all descriptions; this easily becomes too complex (Fig. 7.15, page 118, Fig. 7.17, page 120) to be helpful. Instead, mechanisms for separation of feature description (say, a feature module or trait) and feature combination would better serve understanding.
- *Ruler* compiles to target languages (AG, TEX), but does not proof anything about the described rules. A plugin architecture would allow the translation to different targets, in particular, a description suitable for further use by theorem provers, or other tools.

This thesis and future work We summarize our contribution and future work:

- This thesis: consistency between parts of this thesis and EH implementation by generating material from shared sources. This is done for (1) type rules and their

AG implementation (*Ruler*), and (2) all source code used for the construction of EH compilers and their inclusion in this thesis.

- Future work: other guarantees and derived information related to consistency, for example "explain before use" of (program) identifiers occuring in this thesis, or (automatically generated) indices of used (type system) symbols.
- This thesis: automic generation of type rule implementation.
- Future work: high level (declarative) type rule specification with various mechanisms to automate the translation to an implementation for multiple targets.

12.5 EH, formal properties

Discussion In this thesis we do not make claims about the usual formal properties of a type system: soundness, completeness, and principality (of type inference). However, we still can make the following observations:

- EH3 implements Hindley-Milner type inference, which is standard [38, 17], combined with the use of explicit type signatures, which also has been explored [92, 81].
- From EH4 onwards we allow the same expressiveness as System F by means of type annotations which allow quantifiers at arbitrary positions (in the type annotation). The key question is what kind of types can be inferred when type annotations are omitted. We informally argue the following:
  - We rely on (classical) Hindley-Milner type inference, hence we inherit its properties in case no type signatures are specified.
  - We propagate known type signatures (specified or generalised by means of HM inference) to wherever these signatures are needed, so these signatures can be used as if specified by type annotations.
  - We allow polymorphic types to propagate impredicatively, but we do not invent polymorphism other than via HM generalisation. Propagation is based on the relatively simple notion of "touched by another polymorphic type" (Chapter 7)). This notion can be seen as a characterisation of what we can infer.

These observations are formulated more precisely in Section 6.2 (Theorem 6.4 and Theorem 6.5).

This thesis and future work We summarize our contribution and future work:

- This thesis: we have engineered (but not proven) a type system for exploiting type annotations.

- Future work: further investigate the formal properties of our type inference, in particular the quantifier propagation.

12.6 EH, relation to Haskell

Discussion EH includes essential features of Haskell; EH is closer to the core language internally used by GHC [107, 75], in that it allows System F expressiveness [30, 96]. EH also resembles the core used for the description of the static semantics of Haskell [26] or used by the language definition [84], in the assumption that syntactic desugaring and dependency analysis (of identifiers) has been done.

The strong point, however, of EH, is the lifting of restrictions with respect to explicitly specified information and implicitly inferred information. With a Haskell frontend for syntactic sugar, and a module mechanism, these strong points are easily made available as Haskell with extensions.

This thesis and future work We summarize our contribution and future work:

- This thesis: Haskell extensions on top of EH, a simplified Haskell, or, in the form of acronym: Haskell--++.
- Future work: make EH available via a Haskell frontend. This requires additional preprocessing with respect to dependency analysis and desugaring, a module system and completion of code generation (and much more).

12.7 AG experience

Discussion The AG system is heavily used for the description of all EH implementations. For the description of the EH compilers, the following features of the AG system proved to be essential:

- The AG notation (in essence) offers a domain specific language for the specification of tree based computations (catamorphisms).
- The AG system offers mechanisms to split a description into smaller fragments, and later combine those fragments.
- The AG system allows focussing on the places where something unusual needs to be done, similar to other approaches [62]. In particular, copy rules allow us to forget about a large amount of plumbing.

- A collection of attribute computations can be wrapped into a Haskell function. Although the AG system does not provide notation for higher-order AG [105], this mechanism can be used to simulate higher-order AG's as well as use AG for describing transformations on an AST.

Although the AG system is a simple system it turned out to be a surprisingly useful system, of which some of the features found their way into the *Ruler* system as well. However, the simplicity of our AG system also has its drawbacks:

- Type checking is delegated to AG's target language: Haskell. As a consequence errors are difficult to read because AG's translation is exposed.
- Performance is expected to give problems for large systems. This seems to be primarily caused by the simple translation scheme in which all attributes together live in a tuple just until the program completes. This inhibits garbage collection of intermediate attributes that are no longer required. It also stops GHC from performing optimizations. Work to improve this is in progress, based on AG dependency analysis [97].

Attribute grammar systems have been around for a while [58]. We refer to Parigot's AG www page [82] and Saraiva's work [97] for further reading. Alternative systems are FNC-2 [83], Eli [5, 32], and JastAdd (Java based, with some rewriting like Stratego [109, 108])

BIBLIOGRAPHY

[1] Projet CROAP. Design and Implementaiton of Programming Tools. http://www-sop.inria.fr/croap/, 1999.

[2] Hugs 98.http://www.haskell.org/hugs/, 2003.

[3] C–.http://www.cminusminus.org/, 2004.

[4] ASF+SDF.http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ASF+SDF, 2005.

[5] Eli: An Integrated Toolset for Compiler Construction. http://eli-project.sourceforge.net/, 2005.

[6] Literate Programming.http://www.literateprogramming.com/, 2005.

[7] Martin Abadi and Luca Cardelli.*A Theory of Objects*.Springer, 1996.

[8] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, and Benjamin C. Pierce.Mechanized metatheory for the masses: The POPLmark challenge.In *The 18th International Conference on Theorem Proving in Higher Order Logics*, 2005.

[9] Arthur Baars.Attribute Grammar System. http://www.cs.uu.nl/groups/ST/Center/AttributeGrammarSystem, 2004.

[10] Arthur Baars and S. Doaitse Swierstra.Syntax Macros (Unfinished draft). http://www.cs.uu.nl/people/arthurb/macros.html, 2002.

[11] Stef van Bakel.*Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*.PhD thesis, Mathematisch Centrum, Amsterdam, 1993.

[12] Richard S. Bird.Using Circular Programs to Eliminate Multiple Traversals of Data.*Acta Informatica*, 21:239–250, 1984.

[13] Urban Boquist.*Code Optimisation Techniques for Lazy Functional Languages, PhD Thesis*. Chalmers University of Technology, 1999.

[14] Urban Boquist and Thomas Johnsson.The GRIN Project: A Highly Optimising Back End For Lazy Functional Languages.In *Selected papers from the 8th International Workshop on Implementation of Functional Languages*, 1996.

[15] Didier Botlan, Le and Didier Rémy.ML-F, Raising ML to the Power of System F.In *ICFP*, 2003.

[16] 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.

[17] Luis Damas and Robin Milner.Principal type-schemes for functional programs.In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[18] B.A. Davey and H.A. Priestley.*Introduction to Lattices and Order*.Cambridge Univ. Press, 2nd edition edition, 2002.

[19] Atze Dijkstra.EHC Web.`http://www.cs.uu.nl/groups/ST/Ehc/WebHome`, 2004.

[20] Atze Dijkstra and S. Doaitse Swierstra.Explicit implicit parameters.Technical Report UU-CS-2004-059, Institute of Information and Computing Science, 2004.

[21] Atze Dijkstra and S. Doaitse Swierstra.Typing Haskell with an Attribute Grammar.In *Advanced Functional Programming Summerschool*, number 3622 in LNCS. Springer-Verlag, 2004.

[22] Atze Dijkstra and S. Doaitse Swierstra.Typing Haskell with an Attribute Grammar (Part I). Technical Report UU-CS-2004-037, Department of Computer Science, Utrecht University, 2004.

[23] Atze Dijkstra and S. Doaitse Swierstra.Making Implicit Parameters Explicit.Technical report, Utrecht University, 2005.

[24] Atze Dijkstra and S. Doaitse Swierstra.Ruler: Programming Type Rules.Technical report, Utrecht University, 2005.

[25] Dominic Duggan and John Ophel.Type-Checking Multi-Parameter Type Classes.*Journal of Functional Programming*, 2002.

[26] Karl-Filip Faxen.A Static Semantics for Haskell.*Journal of Functional Programming*, 12(4):295, 2002.

[27] Karl-Filip Faxen.Haskell and Principal Types.In *Haskell Workshop*, pages 88–97, 2003.

[28] Leonidas Fegaras and Tim Sheard.Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space).In *Principles of Programming Languages*, 1996.

[29] Benedict R. Gaster and Mark P. Jones.A Polymorphic Type System for Extensible Records and Variants.Technical Report NOTTCS-TR-96-3, Languages and Programming Group, Department of Computer Science, Nottingham, November 1996.

[30] Jean-Yves Girard.*Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*.PhD thesis, Université Paris VII, 1972.

[31] GrammaTech.Synthesizer Generator. `http://www.grammatech.com/products/sg/overview.html`, 2005.

[32] Rober W. Gray, Simon P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite.Eli: a complete, flexible compiler construction system.*Communications of the ACM*, 35(2):121–130, 1992.

[33] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler.Type Classes in Haskell.*ACM TOPLAS*, 18(2):109–138, March 1996.

[34] Robert Harper.Mechanizing Language Definitions (invited lecture at ICFP05). `http://www.cs.cmu.edu/ rwh/`, 2005.

[35] Bastiaan Heeren.*Top Quality Type Error Messages*.PhD thesis, Utrecht University, Institute of Information and Computing Sciences, 2005.

[36] Bastiaan Heeren and Jurriaan Hage.Type Class Directives.In *Seventh International Symposium on Practical Aspects of Declarative Languages*, pages 253 – 267. Springer-Verlag, 2005.

[37] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra.Generalizing Hindley-Milner Type Inference Algorithms.Technical Report UU-CS-2002-031, Institute of Information and Computing Science, University Utrecht, Netherlands, 2002.

[38] J.R. Hindley.The principal type-scheme of an object in combinatory logic.*Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[39] Ralf Hinze and Simon Peyton Jones.Derivable Type Classes.In *Haskell Workshop*, 2000.

[40] Trevor Jim.Rank 2 type systems and recursive definitions.Technical Report MIT/LCS TM-531, MIT, 1995.

[41] Thomas Johnsson.Attribute grammars as a functional programming paradigm.In *Functional Programming Languages and Computer Architecture*, pages 154–173, 1987.

[42] Mark Jones.Exploring the design space for typebased implicit parameterization.Technical report, Oregon Graduate Institute, 1999.

[43] Mark P. Jones.A system of constructor classes: overloading and implicit higher-order polymorphism.In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, 1993.

[44] Mark P. Jones.*Qualified Types, Theory and Practice*.Cambridge Univ. Press, 1994.

[45] Mark P. Jones.Using Parameterized Signatures to Express Modular Structure.In *Proceedings of the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[46] Mark P. Jones.Typing Haskell in Haskell.In *Haskell Workshop*, 1999.

[47] Mark P. Jones.Type Classes with Functional Dependencies.In *Proceedings of the 9th European Symposium on Programming, ESOP 2000,*, March 2000.

[48] Mark P. Jones.Typing Haskell in Haskell.`http://www.cse.ogi.edu/ mpj/thih/`, 2000.

[49] Mark P. Jones and Simon Peyton Jones.Lightweight Extensible Records for Haskell.In *Haskell Workshop*. Utrecht University, Institute of Information and Computing Sciences, 1999.

[50] Stefan Kaes.Parametric overloading in polymorphic programming languages .In *Proc. 2nd European Symposium on Programming*, 1988.

[51] Wolfram Kahl and Jan Scheffczyk.Named Instances for Haskell Type Classes.In *Haskell Workshop*, 2001.

[52] A. Kfoury and J. Wells.Principality and type inference for intersection types using expansion variables.`http://citeseer.ist.psu.edu/kfoury03principality.html`, 2003.

[53] 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.

[54] 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.

[55] A.J. Kfoury and J.B. Wells.Principality and Type Inference for Intersection Types Using Expansion Variables.*Theoretical Computer Science*, 311(1-3):1–70, 2003.

[56] Assaf Kfoury, Hongwei Xi, and Santiago M. Pericas.The Church Project. `http://www.church-project.org/`, 2005.

[57] Oleg Kiselyov and Chung-chieh Shan.Implict configuration - or, type classes reflect the value of types.In *Haskell Workshop*, 2004.

[58] D.E. Knuth.Semantics of context-free languages.*Mathematical Systems Theory*, 2(2):127–145, 1968.

[59] D.E. Knuth.Literate Programming.*Journal of the ACM*, (42):97–111, 1984.

[60] Donald E. Knuth.*Computers and Typesetting, Volume B, TeX: The Program*.Addison-Wesley, 1986.

[61] M.F. Kuiper and S. Doaitse Swierstra.Using Attribute Grammars to Derive Efficient Functional Programs.In *Computing Science in the Netherlands CSN'87*, November 1987.

[62] Ralf Lämmel and Simon Peyton Jones.Scrap your boilerplate: a practical design pattern for generic programming.In *Types In Languages Design And Implementation*, pages 26–37, 2003.

[63] Konstantin Laufer and Martin Odersky.Polymorphic Type Inference and Abstract Data Types.Technical Report LUC-001, Loyola University of Chicago, 1994.

[64] J. Launchbury and SL. Peyton Jones.State in Haskell. `http://citeseer.nj.nec.com/details/launchbury96state.html`, 1996.

[65] Daan Leijen and Andres Löh.Qualified types for MLF.In *ICFP*, 2005.

[66] Xavier Leroy.Manifest types, modules, and separate compilation.In *Principles of Programming Languages*, pages 109–122, 1994.

[67] Xavier Leroy.Applicative Functors and Fully Transparent Higher-Order Modules.In *Principles of Programming Languages*, pages 142–153, 1995.

[68] Michael Y. Levin and Benjamin C. Pierce.TinkerType: A Language for Playing with Formal Systems.`http://www.cis.upenn.edu/ milevin/tt.html`, 1999.

[69] Jeffrey R. Lewis, Mark B. Shields, Erik Meijer, and John Launchbury.Implicit Parameters: Dynamic Scoping with Static Types.In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts*, pages 108–118, January 2000.

[70] Mark Lillibridge.*Translucent Sums: A Foundation for Higher-Order Module Systems*.PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[71] Andres Löh.*Exploring Generic Haskell*.PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2004.

[72] Andres Löh.lhs2TeX.`http://www.cs.uu.nl/people/andres/lhs2tex/`, 2004.

[73] David B. MacQueen.Using dependent types to express modular structure.In *Principles of Programming Languages*, pages 277–286, 1986.

[74] G. Malcolm.Homomorphisms and promotability.In J.L.A. van Snepscheut, editor, *Mathematics of Program Construction*, number 375 in LNCS, pages 335–347, 1989.

[75] Simon Marlow. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2004.

[76] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[77] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[78] John C. Mitchell and Gordon D. Plotkin. Abstract Types Have Existential Type. *ACM TOPLAS*, 10(3):470–502, July 1988.

[79] Martin Odersky and Konstantin Laufer. Putting Type Annotations to Work. In *Principles of Programming Languages*, pages 54–67, 1996.

[80] 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.

[81] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored Local Type Inference. In *Principles of Programming Languages*, number 3, pages 41–53, March 2001.

[82] Didier Parigot. Attribute Grammars Home Page. `http://www-rocq.inria.fr/oscar/www/fnc2/attribute-grammar-people.html`, 1998.

[83] Didier Parigot. The Fnc-2 Attribute Grammar System. `http://www-rocq.inria.fr/oscar/www/fnc2/littlefnc2.html`, 1998.

[84] Simon Peyton Jones. *Haskell 98, Language and Libraries, The Revised Report*. Cambridge Univ. Press, 2003.

[85] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997.

[86] Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. In *ICFP*, 2003.

[87] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. `http://research.microsoft.com/Users/simonpj/papers/putting/index.htm`, 2004.

[88] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types, 2004.

[89] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[90] Frank Pfenning. Unification and Anti-Unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, 1991.

[91] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[92] Benjamin C. Pierce and David N. Turner. Local Type Inference. *ACM TOPLAS*, 22(1):1–44, January 2000.

[93] Francois Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types (submitted). `http://pauillac.inria.fr/ fpottier/biblio/pottier.html`, 2005.

[94] Francois Pottier and Didier Rémy. *The essence of ML type inference*, chapter 10, pages 389–489. MIT Press, 2005.

[95] Didier Rémy. Simple, partial type-inference for System F based on type-containment. In *ICFP*, 2005.

[96] J.C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation*, number 19 in LNCS, pages 408–425, 1974.

[97] Joao Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University, 1999.

[98] Jan Scheffzcyk. Named Instances for Haskell Type Classes. Master's thesis, Universitat der Bundeswehr München, 2001.

[99] Chung-chieh Shan. Sexy types in action. *ACM SIGPLAN Notices*, 39(5):15–22, May 2004.

[100] Rob Simmons. The Twelf Project (Wiki Home). `http://fp.logosphere.cs.cmu.edu/twelf/`, 2005.

[101] Utrecht University Software Technology Group. UUST library. `http://cvs.cs.uu.nl/cgi-bin/cvsweb.cgi/uust/`, 2004.

[102] M. Srensen and P. Urzyczyn. Lectures on the Curry-Howard isomorphism. Technical Report TOPPS D-368, Univ. of Copenhagen, 1998.

[103] Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. Technical Report TR2002/2, Dept. of Computer Science and Software Engineering, The University of Melbourne, Parkville 3052, Australia, June 2002.

[104] S. Doaitse Swierstra, P.R. Azero Alocer, 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*, number 1608 in LNCS, pages 150–206. Springer-Verlag, 1999.

[105] S. Doaitse Swierstra and H.H. Vogt. Higher order attribute grammars, Lecture notes of the International Summer School on Attribute Grammars, applications and systems. Technical Report RUU-CS-91-14, Utrecht University, 1991.

[106] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[107] Andrew Tolmach. An External Representation for the GHC Core Language (Draft for GHC5.02). `http://www.haskell.org/ghc/documentation.html`, 2001.

[108] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, number 2051 in LNCS, pages 357–361. Springer-Verlag, 2001.

[109] Eelco Visser. Stratego Home Page. `http://www.program-transformation.org/Stratego/WebHome`, 2005.

[110] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy type inference for higher-rank types and impredicativity (submitted to ICFP2005), 2005.

[111] Phil Wadler. Theorems for free! In *4'th International Conference on Functional Programming and Computer Architecture*, September 1989.

[112] Phil Wadler. Proofs are Programs: 19th Century Logic and 21st Century Computing. `http://www.research.avayalabs.com/user/wadler/papers/frege/frege.pdf`, November 2000.

[113] Phil Wadler.The Girard-Reynolds isomorphism .In *Theoretical Aspects of Computer Software*, October 2001.

[114] Phil Wadler and Stephen Blott.How to make ad-hoc polymorphism less ad-hoc.In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, 1988.

[115] Philip Wadler.Deforestation: transforming programs to eliminate trees.In *Theoretical Computer Science, (Special issue of selected papers from 2'nd European Symposium on Programming)*, pages 231–248, 1990.

[116] J.B. Wells.Typability and Type Checking in System F Are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1998.

Samenvatting van "Stapsgewijs door Haskell"

Computerprogramma's worden geschreven met behulp van computerprogramma's, in het bijzonder programma's die een specificatie van een programma vertalen naar een werkend programma. Zo'n specificatie wordt beschreven in een programmeertaal. De huidige trend is dat een programmeertaal, en de daarbij horende implementatie (dwz de realisatie) van de vertaler voor zo'n programeertaal, steeds meer van het werk van een programmeur uit handen neemt. Programmeertaal en implementatie zelf worden daardoor ingewikkelder en steeds moeilijker te implementeren. Het proefschrift "Stapsgewijs door Haskell" is een experiment in het stapsgewijs beschrijven van een implementatie van de (functionele) programmeertaal Haskell met als doel een begrijpelijke en consistente uitleg van die implementatie. Als onderdeel van deze beschrijving worden eveneens enkele uitbreidingen op Haskell beschreven. Het proefschrift representeert een tussenstadium in de ontwikkeling van een complete implementatie met bijbehorende beschrijving, en zal als project hierna voortgezet worden. Essentieel voor het welslagen van dit experiment en de voortzetting ervan is het gebruik van geautomatiseerde oplossingen voor de problemen die handmatig niet te garanderen zijn: consistentie en opsplitsing in stappen.

De reden om deze reis te beginnen is de observatie dat Haskell [84] langzamerhand een gecompliceerde programmeertaal is geworden. Haskell incorporeert veel (experimentele) programmeerconstructies die het programmeren zowel vergemakkelijken als versnellen. Vertalers voor Haskell (e.g. GHC [75]) die ook daadwerkelijk gebruikt worden zijn in de loop van de jaren zo omvangrijk geworden dat het begrijpen van en experimenteren met zulke vertalers ten behoeve van onderzoek erg ingewikkeld geworden is. Daarnaast blijft het experimenteren beperkt tot kleinere implementaties [46], waarbij de aandacht ligt bij een facet van de complete taal en/of implementatie.

Een tweede reden is dat experimenten met Haskell, of het ontwerpen van programmeertalen in het algemeen, meestal plaats vinden in de theoretische setting. De focus ligt dan op het bewijzen van formele eigenschappen. Op het praktische vlak wordt ook geëxperimenteerd, maar vaak is het dan niet duidelijk hoe de theorie en praktijk van een experiment zich onderling verhouden.

De bijdrage van dit proefschrift is allereerst het samenbrengen van een implementatie van Haskell, de beschrijving van deze implementatie, en de bijbehorende formele vastlegging

in de vorm van typeregels. De opzet van het proefschrift is zodanig dat de onderlinge consistentie van deze aspecten wordt gegarandeerd. Hierdoor wordt een verbinding gelegd tussen theorie en praktijk.

ACKNOWLEDGEMENTS

Without my promotor Doaitse Swierstra, this thesis would not have been created. Not only did he give me the opportunity to embark on this thesis journey, during lively discussions he also sparked the ideas implemented in the EHC project described by this thesis.

Johan Jeuring, Oege de Moor, and Lex Augusteijn provided, in their role as reading committee member, valuable feedback. The comments of anonymous reviewers of the included papers were also very helpful.

The Software Technology group is a great place to be. Although occasionally educational obligations blocked my work on this thesis, my colleagues have relieved me of many other educational and orginisational tasks.

Piet van Oostrum and Andres Löh have been helpful with providing some tweaking for respectively TEX and *lhs2TeX*.

Ineke has taken upon her much of the care and worry our mother nowadays requires. She also knows me well enough to ignore my complaining during the last months of thesis writing.

Gerard Legeland, with whom I share the stress relieving pleasure of making music and sitting in his garden.

Finally, but not the least, I thank those who walk with me on the Buddhist path. The (meditation) training, which is part of Buddhism, forms the foundation upon which any life, and thus my life, can be lived in peace. A certain amount of peace of mind turned out to be an essential ingredient for the making of this thesis.

A Notation

A.1 Legenda of notation

Notation	Meaning
σ	type
σ^k	expected/known type
$\square$	any type
v	type variable
ι	identifier
i	value identifier
I	(type) constructor identifier, type constant
Γ	assumptions, environment, context
C	constraints, substitution
$C_{k..l}$	constraint composition of $C_k ... C_l$
$\leqslant$	subsumption, "fits in" relation
σ_Q	σ with a quantifier
$\sigma_{\neg Q}$	σ without a quantifier
f	fixed type variable (a.k.a. skolem type)
o	options to $\cong$

Notation	Meaning
$\mathfrak{o}$	σ for quantifier propagation
σ_Q	σ with a quantifier
$\sigma_{\neg Q}$	σ without a quantifier
$\mathbb{C}$	C for quantifier propagation
$\vartriangle$	meet of two types
$\triangledown$	join of two types
$\cong$	$\leqslant$, $\vartriangle$ or $\triangledown$
$\mathbb{H}$	type alternative hardness (hard or soft)
$\mathbb{H}_h$	hard type alternative
$\mathbb{H}_s$	soft type alternative
$\mathbb{N}$	type alternative need/context (offered or required)
$\mathbb{N}_o$	offered type alternative
$\mathbb{N}_r$	required type alternative
φ	type alternative
ϑ	translated code
$\mathcal{V}$	co-, contravariant context
$\mathcal{V}^+$	covariant context
$\mathcal{V}^-$	contravariant context
π	predicate
ϖ	predicate wildcard (collection of predicates)

A.2 Term language

Values (expressions, terms):

$e ::= int \mid char$	literals
$\mid i$	program variable
$\mid e\ e$	application
$\mid$ **let** $\overline{d}$ **in** e	local definitions
$\mid (e, ..., e)$	tuple
$\mid \lambda p \to e$	abstraction
$\mid e :: t$	type annotated expression
$\mid$ **case** e **of** $\overline{p \to e}$	case expression
$\mid (l = e, ...)$	record
$\mid (e \mid l := e, ...)$	record update
$\mid e.l$	record selection
$\mid (e \mid l = e, ...)$	record extension
$\mid e \sim e$	impredicative application
$\mid e\ (!e \leftsquigarrow \pi!)$	explicit implicit application
$\mid \lambda(!i \leftsquigarrow \pi!) \to e$	explicit implicit abstraction

Declarations of bindings:

$d ::= i :: t$	value type signature
$\mid p = e$	value binding
$\mid$ **data** $\bar{t} = \overline{I\ \bar{t}}$	data type
$\mid$ **class** $\overline{pr} \Rightarrow pr$ **where** $\overline{d}$	class
$\mid$ **instance** $\overline{pr} \Rightarrow pr$ **where** $\overline{d}$	introduced instance
$\mid$ **instance** $i \leftsquigarrow \overline{pr} \Rightarrow pr$ **where** $\overline{d}$	named introduced instance
$\mid$ **instance** $i :: \overline{pr} \Rightarrow pr$ **where** $\overline{d}$	named instance
$\mid$ **instance** $e \leftsquigarrow pr$	value introduced instance

Pattern expressions:

$p ::= int \mid char$	literals
$\mid i$	pattern variable
$\mid i\ @p$	pattern variable, with subpattern
$\mid (p, ..., p)$	tuple pattern
$\mid p :: t$	type annotated pattern
$\mid (r \mid l = p)$	record pattern

Type expressions:

t	$::= Int \mid Char$	type constants
	$\mid t \to t$	function type
	$\mid (t, ..., t)$	tuple type
	$\mid i$	type variable
	$\mid \forall i.t$	universal quantification
	$\mid \exists i.t$	existential quantification
	$\mid (l :: \sigma, ...)$	record

Predicate expressions:

pr	$::= I\ \bar{t}$	class predicate
	$\mid pr \Rightarrow pr$	predicate transformer/abstraction
	$\mid t \backslash l$	record lacks label predicate

Identifiers:

ι	$::= i$	lowercase: (type) variables
	$\mid I$	uppercase: (type) constructors
	$\mid l$	field labels

A.3 Type language

Types:

σ	$::= Int \mid Char$	literals
	$\mid v$	variable
	$\mid \sigma \to \sigma$	abstraction
	$\mid \sigma\ \sigma$	type application
	$\mid \forall v.\sigma$	universally quantified type
	$\mid f$	(fresh) type constant (a.k.a. fixed type variable)
	$\mid \exists \alpha.\sigma$	existentially quantified type
	$\mid (l :: \sigma, ...)$	record
	$\mid \pi \Rightarrow \sigma$	implicit abstraction

Predicates:

π	$::= I\ \overline{\sigma}$	class predicate
	$\mid \pi \Rightarrow \pi$	predicate transformer/abstraction
	$\mid \sigma \backslash l$	record lacks label predicate

Types for quantifier propagation:

σ	$::= ...$	
	$\mid \mathfrak{o}$	type alternatives
$\mathfrak{o}$	$::= v\ [\overline{\varphi}]$	type alternatives

Types for computing meet/join:

σ	$::= ...$	
	$\mid v \barwedge \sigma$	both
	$\mid \square$	absence of type information

Type alternative:

φ	$::= \sigma :: \mathbb{H} \,/\, \mathbb{N}$	type alternative
$\mathbb{N}$	$::= \mathbb{N}_o$	'offered' context
	$\mid \mathbb{N}_r$	'required' context
$\mathbb{H}$	$::= \mathbb{H}_h$	'hard' constraint
	$\mid \mathbb{H}_s$	'soft' constraint

B Rules generated by *Ruler*

The following overview is automatically generated.

EH version	Ruler view	rules
1	*K*	D.TYSIG D.VAL E.ANN E.APP E.APP.F E.APPTOP E.CHAR E.CON E.INT E.LAM E.LET E.VAR P.ANN P.APP P.APPTOP P.CHAR P.CON P.INT P.VAR P.VARAS T.APP T.CON T.QUANT T.VAR T.VAR.W T.WILD
2	*C*	D.TYSIG D.VAL E.ANN E.APP E.APP.F E.APPTOP E.CHAR E.CON E.INT E.LAM E.LET E.VAR P.ANN P.APP P.APPTOP P.CHAR P.CON P.INT P.VAR P.VARAS T.APP T.CON T.QUANT T.VAR T.VAR.W T.WILD
3	*HM*	D.TYSIG D.VAL E.ANN E.APP E.APP.F E.APPTOP E.CHAR E.CON E.INT E.LAM E.LET E.VAR P.ANN P.APP P.APPTOP P.CHAR P.CON P.INT P.VAR P.VARAS T.APP T.CON T.QUANT T.VAR T.VAR.W T.WILD
4	*EX*	D.TYSIG D.VAL E.ANN E.APP E.APP.F E.APPTOP E.CHAR E.CON E.INT E.LAM E.LET E.VAR P.ANN P.APP P.APPTOP P.CHAR P.CON P.INT P.VAR P.VARAS T.APP T.CON T.QUANT T.VAR T.VAR.W T.WILD
5	*DT*	E.CHAR E.CON E.INT E.VAR
6	*DT*	E.CHAR E.CON E.INT E.VAR
7	*DT*	E.CHAR E.CON E.INT E.VAR
8	*CG*	E.CHAR E.CON E.INT E.VAR
9	*P*	E.CHAR E.CON E.INT E.VAR
10	*P*	E.CHAR E.CON E.INT E.VAR
11	*P*	
4_2	*I2*	D.TYSIG D.VAL E.ANN E.APP E.APP.F E.APPTOP E.CHAR E.CON E.INT E.LAM E.LET E.VAR P.ANN P.APP P.APPTOP P.CHAR P.CON P.INT P.VAR P.VARAS T.APP T.CON T.QUANT T.VAR T.VAR.W T.WILD
6_4		

B. Rules generated by *Ruler*

C USED LIBRARIES

C.1 Parser combinators

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 ε	*r*
$f \langle \$ \rangle p$	$\equiv$ *pSucceed f* $\langle * \rangle p$	
pKey `"x"`	symbol/keyword x	`"x"`
$p \langle ** \rangle q$	p followed by q	result of q applied to result of p
p ‘*opt*‘ *r*	$\equiv p \langle | \rangle$ *pSucceed r*	
$p \langle ?? \rangle q$	$\equiv p \langle ** \rangle q$ ‘*opt*‘ *id*	
$p \langle * q, p * \rangle q, f \langle \$ p$	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

C.2 Pretty printing combinators

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

INDEX

TITLES IN THE IPA DISSERTATION SERIES

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21