

# Ruler: programming type rules

Atze Dijkstra and S. Doaitse Swierstra

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

**Abstract.** Some type systems are first described formally, to be sometimes followed by an implementation. Other type systems are first implemented as language extensions, to be sometimes retrofitted with a formal description. In neither case it is an easy task to keep both artefacts consistent. In this paper we introduce *Ruler*, a domain specific language for describing type rules. Type rules can be incrementally described, thus providing a means for building complex type systems on top of simpler ones. Besides checking well-formedness of *Ruler* programs we use them to generate (1) a visual  $\text{\LaTeX}$  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.

## 1 Introduction

Theory and practice of type systems often seem to be miles apart. For example, for the programming language Haskell there exists a language definition [20], a formal description of most of the static semantics [11], a Haskell description of type inferencing [17], several implementations [19, 2], and, on top of this, experimental language features like the class system [16, 21, 10]. However, the relationship between these artefacts is unclear with respect to their mutual consistency and the mutual effect of a change or extension.

### 1.1 The problem

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, or an algorithmic variant of such a type system, or 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, either retrofitting an implementation with a description or the other way around, there is little help in guaranteeing 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.

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 which we generate both the material which can be used as a starting point 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 new and 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 makes 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 to factor code, are not sufficient: a language extension often requires changes in the data types representing the abstract syntax and the required implementation may require changes across multiple modules. Usually, an additional language feature requires textual changes to numerous parts of the language description and implementation.

The feature interactions seem to be inescapable, no matter how much we try to keep language definitions orthogonal. Unfortunately we cannot give simple descriptions of complicated languages, but at least we can try to describe the various aspects in relative isolation. We do this by providing tools that help in building incremental, modular and stratified language descriptions.

The need for the *Ruler* system arose in the context of the Essential Haskell (EH) project [6, 7]. The design goal of EH is to construct a compiler for an extended version of Haskell, and to (simultaneously) build an explanation of its implementation, while keeping both versions consistent by generating corresponding parts from a single source. This approach resembles the one taken by Pierce [22, 18] who explains both non-algorithmic and algorithmic variants of type systems. The EH project starts with the description and implementation of a very simple language, and extends it in a sequence of steps by adding features like higher ranked polymorphism, mechanisms for explicitly passing implicit parameters [9], and higher order kinds. Each step corresponds to a working compiler. In this context, *Ruler* allows the description of a type system to be partitioned as a sequence of steps. In this paper we demonstrate our approach on a very small example, stepping from the declarative view on a type system towards its implementation view. Of these steps, the final one corresponds to a working implementation.

## 1.2 Running example

We explore the above problems and our solution by looking at the final products that are generated by the *Ruler* system, in figures 1 through 3. We emphasize at this point that a

---


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

$$\frac{}{\Gamma \vdash^e \text{int} : \text{Int}} \text{E.INT}_E \quad \frac{i \mapsto \sigma \in \Gamma \quad \tau = \text{inst}(\sigma)}{\Gamma \vdash^e i : \tau} \text{E.VAR}_E \quad \frac{\Gamma \vdash^e a : \tau_a \quad \Gamma \vdash^e f : \tau_a \rightarrow \tau}{\Gamma \vdash^e f a : \tau} \text{E.APP}_E$$

$$\frac{(i \mapsto \tau_i), \Gamma \vdash^e b : \tau_b}{\Gamma \vdash^e \lambda i \rightarrow b : \tau_i \rightarrow \tau_b} \text{E.LAM}_E \quad \frac{(i \mapsto \sigma_e), \Gamma \vdash^e b : \tau_b \quad \Gamma \vdash^e e : \tau_e \quad \sigma_e = \forall \bar{v}. \tau_e, \quad \bar{v} \notin \text{fv}(\Gamma)}{\Gamma \vdash^e \text{let } i = e \text{ in } b : \tau_b} \text{E.LET}_E$$


---

**Fig. 1.** Expression type rules (E)

---


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

$$\frac{}{C^k; \Gamma \vdash^e \text{int} : \text{Int} \rightsquigarrow C^k} \text{E.INT}_A \quad \frac{i \mapsto \sigma \in \Gamma \quad \tau = \text{inst}(\sigma)}{C^k; \Gamma \vdash^e i : \tau \rightsquigarrow C^k} \text{E.VAR}_A$$

$$\frac{C^k; \Gamma \vdash^e f : \tau_f \rightsquigarrow C_f \quad C_f; \Gamma \vdash^e a : \tau_a \rightsquigarrow C_a \quad v \text{ fresh} \quad \tau_a \rightarrow v \cong C_a \tau_f \rightsquigarrow C}{C^k; \Gamma \vdash^e f a : C C_a v \rightsquigarrow C C_a} \text{E.APP}_A \quad \frac{v \text{ fresh} \quad C^k; (i \mapsto v), \Gamma \vdash^e b : \tau_b \rightsquigarrow C_b}{C^k; \Gamma \vdash^e \lambda i \rightarrow b : C_b v \rightarrow \tau_b \rightsquigarrow C_b} \text{E.LAM}_A$$

$$\frac{v \text{ fresh} \quad C^k; (i \mapsto v), \Gamma \vdash^e e : \tau_e \rightsquigarrow C_e \quad \sigma_e = \forall (\text{fv}(\tau_e) \setminus \text{fv}(C_e \Gamma)). \tau_e \quad C_e; (i \mapsto \sigma_e), \Gamma \vdash^e b : \tau_b \rightsquigarrow C_b}{C^k; \Gamma \vdash^e \text{let } i = e \text{ in } b : \tau_b \rightsquigarrow C_b} \text{E.LET}_A$$


---

**Fig. 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.uniqI)
    = rulerMk1Uniq @lhs.uniq
    loc.tv_ = Ty_Var @uniqI
    (loc.c_, loc.mtErrs)
      = (@a.ty ‘Ty_Arr’ @tv_)  $\cong$  (@a.c  $\oplus$  @f.ty)
    lhs.c   = @c_  $\oplus$  @a.c
    .ty     = @c_  $\oplus$  @a.c  $\oplus$  @tv_

```

---

**Fig. 3.** Part of the generated implementation for rule E.APP

full understanding of these figures is not required nor intended. The focus of this paper is on the construction of the figures, not on their meaning: *Ruler* only tackles the problem of *consistency* between such figures and the corresponding implementation. Our aim is to look at these figures from a metalevel and to see how type rules can be specified and how their content can be generated using our *Ruler* system. Nevertheless, we have chosen a small, well known, and realistic example: the Hindley-Milner (HM) type system. Fig. 1 gives the well-known equational rules, Fig. 2 the algorithmic variant and Fig. 3 part of the generated implementation. In this section these figures demonstrate the general idea of *Ruler* and the artefacts generated by *Ruler*; later sections discuss the involved technicalities.

Both type rules and their implementation can be used to explain a type system. This is what we have done within the context of the EH project. For example, rules similar to rule E.APP from Fig. 2 and the corresponding attribute grammar (AG) implementation from Fig. 3 are jointly explained, each strengthening the understanding of the other. However, when features are combined, this inevitably leads to 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 and extensions to 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 management 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. We may e.g. change the very first view, and have these changes automatically included in all subsequent views.

For example, Fig. 1 displays view *E* (equational), and Fig. 2 displays view *A* (algorithmic) on the set of type rules, where each rule consists of judgements (premises and a single conclusion). View *A* is built on top of view *E* by specifying the differences with respect to view *E*. In the electronic version of this paper, 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, the unchanged part is displayed in grey (however, in the printed version all is 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 crystallized into a working system. We use this similarity by specifying type rules using a single notation, which contains enough information to generate both the sets of type rules (in Fig. 1 and Fig. 2) as well as part of the AG implementation (in Fig. 3). Fig. 3 shows the generated implementation for rule *E.APP*. In this way we address “Problem 1”.

The main goal of our *Ruler* system is to have one integrated definition of type rules, views on those rules, and the specification of information directing the generation of a partial implementation. As an additional benefit, *Ruler* allows one to describe the “type of the judgements” and checks whether the rules are “well-typed”.

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 extend the system to generate output to be used as input for other targets (besides  $\text{\LaTeX}$  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: both their formal and practical aspects. In this light, this paper also is an invitation to the readers to improve on these aspects. In our conclusion (Section 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 paper:

**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 automatically guaranteed as a consequence of being generated from a single source.

**Incrementality.** It is (relatively) easy to incrementally describe type rules.

**Well-formedness checks.** Judgements are checked against the ‘type’ of a judgement.

The remainder of this paper is organised as follows: in Section 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 3. In Section 4 we specify the contents of Fig. 1. In Section 5 we continue with the extension towards an algorithmic version from which attribute grammar (AG) code can be generated. Finally, we discuss related work in Section 6, and experiences and future work in Section 7.

## 2 *Ruler* overview

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. 1 all judgements for an expression should match the structure of the 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.

In the remainder of this section we give a high-level overview of the concepts manipulated by *Ruler*. Fig. 4 gives a schematic *Ruler* specification, showing how the concepts relate syntactically.

A *Ruler* specification consists of rules organized into rulesets. Each rule consists of judgements which must comply with a scheme which describes the structure of a judgement. We make this more precise in the remainder of this section.

The structure of a judgement is described by a (judgement) *scheme*. A scheme is the signature of a judgement. For each scheme we specify multiple *views*. A view on a scheme consists of named *holes* and judgeshapes, which come in the following varieties:

- One *judgespec*, used to specify the template for judgements.
- For each output target a *judgeuse*, used to specify how to map a judgement to an output format.

Holes are the parameters of a scheme and are given concrete values when instantiated as a judgement by means of a *judgespec* template.

Rules are grouped into *rulesets*. From a ruleset a figure like Fig. 1 is generated, so it specifies 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.

A *rule* consists of a set of judgement instances (syntactically denoted by keyword **judge**) for the premises, and a judgement for the conclusion. Just as we specify views for schemes, we specify views for rules. For each view, each of the judgements of a rule should comply with the corresponding view of the scheme of the judgement. A judgement is defined by binding hole names to *Ruler* expressions.

---

```

scheme  $X$  =
  view  $A$  =
    holes ...
    judgespec ...
    judgeuse ...
    ...
  view  $B$  =
    holes ...
    judgespec ...
    judgeuse ...
    ...
ruleset  $x$  scheme  $X$  =
  rule  $r$  =
    view  $A$  =
      judge ... -- premises
      ...
      -
      judge ... -- conclusion
    view  $B$  = ...
  rule  $s$  =
    view  $A$  = ...
    view  $B$  = ...

```

---

**Fig. 4.** High level structure of Ruler source

Views are ordered by a *view hierarchy*. A view hierarchy specifies which view inherits from which other (ancestor) view. Both schemes and rules contain a collection of views. A view on a scheme inherits the holes and judgespecs. A view on a rule inherits the hole bindings to *Ruler* expressions. Only new holes have to be given a binding to a *Ruler* expression; existing holes may be updated. This is how *Ruler* supports the incremental definition of views.

The incremental definition of views on a rule is supported by two different variants of specifying a judgement:

- 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 view hierarchy, 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 *Ruler* expressions 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 *Ruler* system is open-ended in the sense that some judgements can be expressed in a less structured form, for which an implementation is defined externally. For example, the premises of rule `E.VAR` consist of conditions specified elsewhere. These arbitrary (i.e. as far as *Ruler* is concerned unstructured) conditions are treated like regular judge-

Value expressions:		Types:	
$e ::= int$	literals	$\tau ::= Int$	literals
$i$	program variable	$v$	variable
$e e$	application	$\tau \rightarrow \tau$	abstraction
$\lambda i \rightarrow e$	abstraction	$\sigma ::= \forall \bar{v}. \tau$	universally quantified type, $\bar{v}$ possibly empty
<b>let</b> $i = e$ <b>in</b> $e$	local definitions		

**Fig. 5.** Terms and types

ments, but their implementation has to be specified explicitly. We call the scheme of such a judgement variant a *relation*.

### 3 Our example language

In this section we explain notation in our example, that is, the set of type rules to be specified with *Ruler*. There should be no surprises here as we use a standard term language based on the  $\lambda$ -calculus (see Fig. 5). Our example language contains the following program:

```

let  $id = \lambda x \rightarrow x$ 
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. 5. Types are either monomorphic types  $\tau$ , called *monotypes*, or universally quantified types  $\sigma$ , called *polymorphic types* or *polytypes*. A monotype either is a type constant  $Int$ , a function type  $\tau \rightarrow \tau$ , or an unknown type represented by a type variable  $v$ . We discuss the use of these types when we introduce the type rules for our term language in the following sections.

The type rules use an environment  $\Gamma$ , holding bindings for program identifiers with their typings:

$$\Gamma ::= \overline{i \mapsto \sigma}$$

### 4 Describing type rules using *Ruler* notation

In this section we make the use of *Ruler* more precise. We describe *Ruler* notation, and explain how to specify the content of Fig. 1. The transition (instantiation) from polytypes to monotypes is performed by *inst*, whereas the transition (generalisation) from monotypes to polytypes is described in rule E.LET.

The use of an equational version of type rules usually serves to explain a type system and to prove properties about the type system. An algorithmic version is introduced subsequently to specify an implementation for such a type system. In this paper we follow the same pattern, but use it to show how *Ruler* can be used to describe the



equational version in such a way that its type rule representation can be included in the documentation (read here: this paper). In the extended version of this paper [7] we also describe how to extend the equational version to an algorithmic one, and a version from which a partial implementation can be generated.

*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). We introduce such a judgement by a **scheme** declaration, which is immediately followed by the views on this scheme<sup>1</sup>:

```

scheme expr =
  view E =
    holes [| e : Expr, gam : Gam, ty : Ty |]
    judgespec gam ⊢ e : ty
    judgeuse tex gam ⊢ .."e" e : ty
  ruleset expr.base scheme expr "Expression type rules" =
    rule e.int =
      view E = -- no premises
      -
      judge R : expr = gam ⊢ int : Ty_Int

```

We first examine the **scheme** definition in detail, and then the **ruleset** definition for the type rules themselves. The operator  $\vdash \text{..} \text{"e"} \text{"}$  forms a single operator in which the dot notation expresses subscripting and superscripting when pretty printed; the part after the first dot is used as a subscript, the part after the second dot as a superscript. Here the turnstyle symbol is superscripted with *e*. We refer to the extended version of this paper [7] for further explanation about the dot notation.

Here we have only a single, equational, view *E*, which defines three holes: *e*, *gam*, and *ty*. This scheme is instantiated in the conclusion of rule *E.INT* by using the **judgespec** of the scheme. In later views, we will (re)define individual holes. Each hole has an associated hole type, for instance *e* has type *Expr*; we will not discuss this further.

Judgeshapes are introduced by the keyword **judgespec** or **judgeuse**. A **judgespec** introduces a distfix operator template (here:  $\vdash \dots : \dots$ ) used for specifying instances of this judgement in a **rule**. A **judgeuse** judgement shape introduces the expression to be used for the generation for a *target*. The target **tex** indicates that the shape is to be used to generate L<sup>A</sup>T<sub>E</sub>X; the **ag** target is meant for attribute grammar generation. We will refer to these three shapes as the **spec**, **tex** and **ag** judgement shapes. The **spec** shape is used as the input template, the **tex** and **ag** shapes are used as output templates.

The identifiers in *Ruler* expressions should refer to the introduced hole names or externally defined identifiers. The **spec** shape of the scheme of a judgement is used to

---

<sup>1</sup> The text for *Ruler* program fragments already appears in pretty printed form throughout this paper, but in the original source code the **judgespec** appears as: “judgespec *gam* :- *e* : *ty*”

extract *Ruler* expressions from a judgement instance (defined by **judge**) and bind these to the corresponding hole identifiers.

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 notation to be used in the judgespec too, although it only will play a role in its defaulted uses.

*The basics: rulesets.* Rules are grouped in rulesets to be displayed together in a figure: the description of Fig. 1 starts with the **ruleset**. A ruleset specifies the name *expr.base* of the ruleset, the scheme *expr* for the conclusion of its contained 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. 1. The ruleset name *expr.base* is used to uniquely identify this figure, so it can be included in text such as this paper. We do not discuss this further; we only note that part of the  $\text{\LaTeX}$  formatting is delegated to external  $\text{\LaTeX}$  commands.

Before discussing its components, we repeat the  $\text{\LaTeX}$  rendering of rule  $\text{E.INT}$  together with its *Ruler* definition to emphasize the similarities between the rule specification and its visual appearance:

$$\begin{array}{l} \text{rule } e.int = \\ \text{view } E = \quad \text{-- no premises} \\ \text{---} \\ \text{judge } R : expr = gam \vdash int : Ty\_Int \end{array} \qquad \frac{}{\Gamma \vdash^e int : Int} \text{E.INT}_E$$

All views of a rule are jointly defined, although we present the various views separately throughout this paper. For an individual figure and a particular view, the *Ruler* system extracts the relevant view of the rules present in a ruleset. We will come back to this in our discussion.

Each view for a rule specifies premises and a conclusion, separated by a ‘-’. The rule  $\text{E.INT}$  for integer constants only has a single judgement for the conclusion. The judgement is named *R*, follows scheme *expr*, and is specified using the **spec** judgement shape for view *E*. The name of the judgement is used to refer to the judgement from later views, either to overwrite it completely or to redefine the values of the individual holes.

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 (we do not discuss this nor its proper  $\text{\LaTeX}$  formatting further).

The rule  $\text{E.APP}$  for the application of a function to an argument is defined similar to rule  $\text{E.INT}$ . Premises now relate the type of the function and its argument:

$$\begin{array}{l} \text{rule } e.app = \\ \text{view } E = \\ \text{judge } A : expr = gam \vdash a : ty.a \\ \text{judge } F : expr = gam \vdash f : (ty.a \rightarrow ty) \\ \text{---} \\ \text{judge } R : expr = gam \vdash (f\ a) : ty \end{array} \qquad \frac{\Gamma \vdash^e a : \tau_a \quad \Gamma \vdash^e f : \tau_a \rightarrow \tau}{\Gamma \vdash^e f\ a : \tau} \text{E.APP}_E$$

## 5 Extending the initial version

The above demonstrates the basic features of *Ruler*. We continue with highlighting *Ruler* features. We construct an algorithmic view  $A$ , which is built upon the equational view  $E$ . The final attribute grammar view  $AG$  adds further implementation details required for the attribute grammar translation. We assume familiarity with the terminology of attribute grammar systems.

*Algorithmic version.* The following extends the scheme definition:

**view**  $A =$   
**holes** [ $e : Expr, gam : Gam \mid$  **thread**  $cnstr : C \mid ty : Ty$ ]  
**judgespec**  $cnstr.inh; gam \vdash \dots e : ty \rightsquigarrow cnstr.syn$

The algorithmic view introduces computation order. *Ruler* assumes that a derivation tree corresponds to an abstract syntax tree (AST): rule conclusions correspond to parent (tree) nodes, rule premises to children nodes. An attribute grammar for such an AST specifies attributes traveling down the tree, called *inherited attributes*, and attributes traveling upwards, called *synthesized attributes*. Inherited attributes correspond to the assumptions of the conclusion of a type rule, and are usually propagated from the conclusion towards the premises of a rule. Synthesized attributes correspond to results. The **holes** declaration specifies its holes as inherited, inherited + synthesized, and synthesized, indicated by their respective positions between vertical bars ‘|’.

For example,  $gam$  is inherited because it provides the assumptions under which a type rule is satisfied. In our example we treat  $gam$  as given, not as something to be computed. In terms of a computation,  $gam$  provides an argument which may be used to compute synthesized values such as the type  $ty$  of the type rule. In addition to these attributes, a new hole  $cnstr$  is declared as inherited + synthesized, and is threaded through the type rule as indicated by the optional keyword **thread**. The inherited  $cnstr$  (referred to by  $cnstr.inh$ ) represents already known assumptions about types, in particular about type variables; the synthesized  $cnstr$  (referred to by  $cnstr.syn$ ) represents the combination of newfound and known information about type variables. These holes generate the *attr* declarations in Fig. 3.

For rule  $E.INT$  we extend its definition with values for  $cnstr.inh$  and  $cnstr.syn$ :

**view**  $A =$   
 $-$   
**judge**  $R : expr$   
 $\mid cnstr.syn = cnstr..k$   $\overline{C^k; \Gamma \vdash^e int : Int \rightsquigarrow C^k}^{E.INT_A}$   
 $\mid cnstr.inh = cnstr..k$

The difference between holes and the values bound to holes lies in the computation order assigned to the holes and its use in the mapping onto an attribute grammar. Both  $cnstr.inh$  and  $cnstr.syn$  are bound to  $cnstr..k$ , which appears as  $C^k$  in rule  $E.INT$ . However, within the context of a type rule the values bound to inherited holes of the conclusion and synthesized holes of the premises correspond to input values, that is, values which may be used further to define the synthesized holes of the conclusion and

the inherited holes of the premises. For example,  $cnstr.k$  is the value bound to the inherited hole  $cnstr.inh$  and the synthesized hole  $cnstr.syn$  of the conclusion. In terms of the type rule this means that both must be equal; in terms of the computation  $cnstr.inh$  is passed to  $cnstr.syn$  via  $cnstr.k$ . The identifier  $cnstr.k$  is an argument to the judgements of the type rule, whereas for the computation it is the name (or pattern when more complicated) to which the value of the attribute associated with  $cnstr.inh$  is bound. It is subsequently used for the computation of  $cnstr.syn$ . This leads to the following AG code for rule  $E.INT^2$ :

```
sem Expr
| Int lhs.c = @lhs.c
  .ty = Ty_Int
```

*Ruler* supports a renaming mechanism which is used to rename identifiers when generating output. Strictly, this is not necessary, but in practice it is convenient to have such a mechanism to glue *Ruler* generated code non-*Ruler* implementation fragments more easily. For instance, in the full *Ruler* specification for the example *cnstr* is renamed to *c*; this shows in the attribute declaration in Fig. 3. The keyword **lhs** refers to the parent node of a AST fragment, **@lhs.c** to its inherited *c* attribute; the parent corresponds to the conclusion of a rule. The translation also assumes an infrastructure of AG and Haskell definitions, for example for the constant *Ty\_Int* representing the *Int* type. We will not discuss such details further, and assume the reader is familiar enough with attribute grammars and Haskell to guess the semantics of the provided translation. A more detailed treatment of these features can be found elsewhere [7, 6].

More complex *Ruler* expressions can be bound to holes, as shown by the following extension to view *A* for rule  $E.APP$ :

```
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
```

$$\frac{\begin{array}{l} C^k; \Gamma \vdash^e f : \tau_f \leadsto C_f \\ C_f; \Gamma \vdash^e a : \tau_a \leadsto C_a \\ v \text{ fresh} \\ \tau_a \rightarrow v \cong C_a \tau_f \leadsto C \end{array}}{C^k; \Gamma \vdash^e f a : C C_a v \leadsto C C_a} E.APP_A$$

The hole  $cnstr.syn$  for the conclusion (judgement *R*) is defined in terms of *cnstr* and *cnstr.a*. In the type rule this shows as a juxtaposition, which one may read as the

<sup>2</sup> The actual AG code is optimized w.r.t. the elimination of straightforward copies like **lhs.c = @lhs.c**: the AG system provides a copy rule mechanism which automatically inserts copy rules for attributes when explicit attribute rules are absent.

first constraint *cnstr* applied to the second *cnstr.a*. However, for the AG code additional rewriting (specified by means of rewrite rules) is required to a form which is more explicit in how the juxtaposition is to be computed. Again, we omit further discussion of this feature, and show the translation to AG code instead:

```
sem Expr
| App (f.uniq, loc.uniq1)
    = rulerMk1Uniq @lhs.uniq
  f .c = @lhs.c
  .g = @lhs.g
  a .c = @f.c
  .g = @lhs.g
  loc.tv_ = Ty_Var @uniq1
  (loc.c_, loc.mtErrs)
    = (@a.ty 'Ty_Arr' @tv_)  $\cong$  (@a.c  $\oplus$  @f.ty)
  lhs.c = @c_  $\oplus$  @a.c
  .ty = @c_  $\oplus$  @a.c  $\oplus$  @tv_
```

This translation is not yet optimized in order to show the correspondence with the type rule. Fig. 3 shows the optimized version. The overloaded operator  $\oplus$  applies a constraint by substituting type variables with types in the usual way. The dataconstructors, of which the name starts with *Ty* represent the various type encodings.

Of the remaining judgements *V* and *M* we discuss judgement *M* in the next paragraph. Both judgements use the same mechanism for specifying arbitrary judgements.

*External schemes: relations.* Rule E.APP also demonstrates the use of judgements which are not directly related to the structure of the AST. For example, the freshness of a type variable (judgement *V*) and type unification (judgement *M*) are expressed in terms of attribute values, not children nodes of a parent node in an AST. *Ruler* automatically derives an implementation for judgements related to children nodes, but for the remaining judgements the *Ruler* programmer has to provide an implementation.

For example, type unification (or matching), is declared as a *relation*, which is a judgement scheme for which we have to define the AG implementation ourselves. Such an implementation is defined using **judgeuse** for target **ag**:

```
relation match =
  view A =
    holes [ty.l : Ty, ty.r : Ty || cnstr : C]
    judgespec ty.l  $\cong$  ty.r  $\rightsquigarrow$  cnstr
    judgeuse ag (cnstr, mtErrs) '='(ty.l)  $\cong$  (ty.r)
```

The scheme of *match* specifies how the judgement is to be translated to attribute grammar code by means of a **judgeuse ag**. It is the responsibility of the programmer to provide the correct form (an attribute rule) and translation, so the attribute grammar translation of *match* is expressed in terms of the Haskell function  $\cong$ , returning both additional constraints as well as (possibly zero) errors. It is the responsibility of the surrounding infrastructure to do something useful with reported errors as these are not

part of the result of the relation. As such, a relation in *Ruler* has the same role as a foreign function in Haskell.

*The rest.* The expanded version of this paper [7] describes view *A* and the following additional aspects:

- The third (AG) view, which extends the specification of view *A* on rule *E.INT* and rule *E.APP* with information binding the type rules to the abstract syntax.
- Additional datastructures (e.g. substitutions/constraints *C*) required for an algorithmic version of the type rules, further surrounding infrastructure (error handling, parsing, commandline invocation)
- The creation and handling of fresh (or unique) values, as required by judgement *V* of scheme *tvFresh*.
- The use of rewrite rules and identifier formatting.

## 6 Related work

*Literate programming.* Literate programming [4] is a style of programming where the program source text and its documentation are combined into a single 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.

*TinkerType.* TinkerType [18], used for Pierce’s book [22], 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 code fragments. The structure of judgements is not checked.

*Twelf.* The theorem proving environment Twelf [23] is used to describe and prove properties for programming languages [13], thus answering the POPLmark challenge [5]. 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 well-documented compilers.

*AST based tools.* Various abstract syntax tree (AST) based compiler construction tools exist [1, 3, 24, 12], 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 form of programming. Furthermore, in our experience, stepwise AG descriptions [8] became too complex (which inspired us to design *Ruler* in the first place), and we expect this to be 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.

## 7 Discussion and future work

*Experiences with Ruler.* *Ruler* solves the problem of maintaining consistency and managing type rules. Within the context of constructing a documented Haskell compiler (EH project, [6, 7]) this was an acute problem. It is a relief not to have to write  $\text{\LaTeX}$  for type rules by hand, to know that the formatted rules correspond directly to their implementation, and to know several checks have been performed.

*Lightweight solution.* *Ruler* is a lightweight solution to the general problem of maintaining consistency, tailored towards the needs of the EH project. This is reflected in the following:

- *Ruler* uses a few operator symbols for its own purposes, leaving available as much as possible to the user of *Ruler*. Similarly, judgements have minimal structure, using combinations of operator symbols as distfix operators. Both design choices give maximum typographical freedom and avoid a fixed meaning of judgements.
- The structure provided by holes and judgement schemes allows exactly the translation to AG. Although not formally defined, the correspondence between type rules and AG is relatively self-evident and often observed in literature.

These design decisions ensure that *Ruler* is what it is meant to be: a relatively simple solution for keeping type rules and their implementation consistent. However, the desire to check and generate more from a single description surely will motivate further evolution of the *Ruler* formalism (e.g.) to be more restrictive (thus allowing more checks) and more expressive (thus allowing more aspects to be specified). A more restrictive syntax for *Ruler* expressions would also enable a more concise, but implicit, notation, avoiding the need for keywords like *judge*.

*Views as organizational structure.* Rules are organized as groups of views; *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). In practice it works well to develop all views together, to allow for a understandable partitioning into different views, while at the same time keeping an overview.

*Emphasizing differences.* We use colors in the electronic version of this paper to emphasize differences in type rules. For black-and-white print this is hardly a good way to convey information to the reader. However, we believe that to understand more complex material, more technical means (like colors, hypertext, collapsable/expandable text) must be used to manage the complexity of descriptions.

*Future research.* We foresee the following directions of further research and development of *Ruler*:

- 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.
- In this paper, 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 paper we use HM inference, a greedy resolution of constraints. Alternate strategies exist [15, 14]; *Ruler* (or similar tools) can provide abstractions of such strategies.
- *Ruler* exploits the syntax-directed nature of type rules. This implies 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 each other, with the lowest level corresponding to an AST based target language.
- *Ruler* compiles to target languages (AG,  $\text{T}_{\text{E}}\text{X}$ ), but does not prove anything about the described rules. A plugin architecture would allow the translation to different targets, and in particular, into a description suitable for further use by theorem provers etc..
- Views are built on top of each other by introducing new holes and adapting rules. Currently we do not deal with possible conflicts between views. However, we expect that a combination with the more aspect oriented approach taken by Tinker-Type [18] eventually will lead to a more modular approach where type system aspects together with implementation fragments can be combined and checked for inconsistencies.

## References

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. ASF+SDF. <http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ASF+SDF>, 2005.
4. Literate Programming. <http://www.literateprogramming.com/>, 2005.
5. 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.
6. Atze Dijkstra. EHC Web. <http://www.cs.uu.nl/groups/ST/Ehc/WebHome>, 2004.
7. Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.
8. Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an Attribute Grammar. In *Advanced Functional Programming Summerschool*, number 3622 in LNCS. Springer-Verlag, 2004.
9. Atze Dijkstra and S. Doaitse Swierstra. Making Implicit Parameters Explicit. Technical report, Utrecht University, 2005.



10. Dominic Duggan and John Ophel. Type-Checking Multi-Parameter Type Classes. *Journal of Functional Programming*, 2002.
11. Karl-Filip Faxen. A Static Semantics for Haskell. *Journal of Functional Programming*, 12(4):295, 2002.
12. GrammaTech. Synthesizer Generator.  
<http://www.grammatech.com/products/sg/overview.html>, 2005.
13. Robert Harper. Mechanizing Language Definitions (invited lecture at ICFP05).  
<http://www.cs.cmu.edu/~rwh/>, 2005.
14. Bastiaan Heeren and Jurriaan Hage. Type Class Directives. In *Seventh International Symposium on Practical Aspects of Declarative Languages*, pages 253 – 267. Springer-Verlag, 2005.
15. 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.
16. Mark P. Jones. *Qualified Types, Theory and Practice*. Cambridge Univ. Press, 1994.
17. Mark P. Jones. Typing Haskell in Haskell. <http://www.cse.ogi.edu/~mpj/thih/>, 2000.
18. Michael Y. Levin and Benjamin C. Pierce. TinkerType: A Language for Playing with Formal Systems. <http://www.cis.upenn.edu/~milevin/tt.html>, 1999.
19. Simon Marlow. The Glasgow Haskell Compiler. <http://www.haskell.org/ghc/>, 2004.
20. Simon Peyton Jones. *Haskell 98, Language and Libraries, The Revised Report*. Cambridge Univ. Press, 2003.
21. Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997.
22. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
23. Rob Simmons. The Twelf Project (Wiki Home).  
<http://fp.logosphere.cs.cmu.edu/twelf/>, 2005.
24. Eelco Visser. Stratego Home Page.  
<http://www.program-transformation.org/Stratego/WebHome>, 2005.