Ruler: Programming Type Rules

Atze Dijkstra
S. Doaitse Swierstra

institute of information and computing sciences,
utrecht university

technical report UU-CS-2005-033

www.cs.uu.nl

Ruler: programming type rules

Atze Dijkstra and S. Doaitse Swierstra

August 8, 2005

Abstract

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 paper 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. Uniting 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 the following artefacts exist:

- A language definition for the Haskell98 standard [24], 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 [15].
- Several implementations, of which we mention GHC [2] and Hugs [1].
- 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 [19], and multi-parameter type classes [25, 14] present in extensions [2, 1] to Haskell98.
- A Haskell description of type inferencing for Haskell98 [20], 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 [16].

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 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.

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 paper, and which are presented in figures 1 through 3. The reader does not need to understand the content of these figures. The focus of this paper is on the construction of the figures, not on their meaning. Our aim is to look at these figures

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

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

$$\frac{(i \mapsto \tau_i), \Gamma \vdash^e b : \tau_b}{\Gamma \vdash^e \lambda i \rightarrow b : \tau_i \rightarrow \tau_b}\ \text{E.LAM}_E \qquad \frac{\begin{array}{c} (i \mapsto \sigma_e), \Gamma \vdash^e b : \tau_b \\ \Gamma \vdash^e e : \tau_e \\ \sigma_e = \forall \overline{v}.\tau_e, \quad \overline{v} \notin ftv\ (\Gamma) \end{array}}{\Gamma \vdash^e \textbf{let}\ i = e\ \textbf{in}\ b : \tau_b}\ \text{E.LET}_E$$

Figure 1: Expression type rules (E)

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

$$\frac{}{C^k; \Gamma \vdash^e int : Int \leadsto C^k}\ \text{E.INT}_A \qquad \frac{\begin{array}{c} i \mapsto \sigma \in \Gamma \\ \tau = inst\ (\sigma) \end{array}}{C^k; \Gamma \vdash^e i : \tau \leadsto C^k}\ \text{E.VAR}_A$$

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

$$\frac{\begin{array}{c} v\ \text{fresh} \\ C^k; (i \mapsto v), \Gamma \vdash^e e : \tau_e \leadsto C_e \\ \sigma_e = \forall\ (ftv\ (\tau_e) \backslash ftv\ (C_e \Gamma)).\tau_e \\ C_e; (i \mapsto \sigma_e), \Gamma \vdash^e b : \tau_b \leadsto C_b \end{array}}{C^k; \Gamma \vdash^e \textbf{let}\ i = e\ \textbf{in}\ b : \tau_b \leadsto C_b}\ \text{E.LET}_A$$

Figure 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 3: Part of the generated implementation

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. 1 gives the equational rules, Fig. 2 the algorithmic variant and Fig. 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 [12, 8, 9]. The design goal of EH is to build 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, which in his book [26] explains both non-alogoirthmic and algorithmic variant of type systems. The EH projects start with the description of a very simple language, and extend it in a sequence of steps, leading to full Haskell with extensions (including higher ranked polymorphism, mechanisms for explicitly passing implicit parameters [11, 13], extensible records [17, 21], higher order kinds). Each step introduces new features and describes the associated 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. 2 and the corresponding attribute grammar (AG) implementation from Fig. 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. The difference is that a version management system stores delta's between versions in order to save space, whereas for us the changes themselves are object of discussion. A version management system allows access to one version at a time, usually the latest, whereas 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. 1 view *E* (equational), and Fig. 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*. 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 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. 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".

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 paper also is an invitation to the readers to improve on these aspects. In our conclusion (Section 8) 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 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 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 extend this specification for the contents of Fig. 2. In Section 6 we explain the AG (attribute grammar) system. In Section 7 we extend the example *Ruler* specification so that *Ruler* can generate AG code. Finally we discuss and conclude in Section 8.

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. 4 gives an overview of the tools used to construct the example compiler for the type rules presented in this paper. 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* [23].

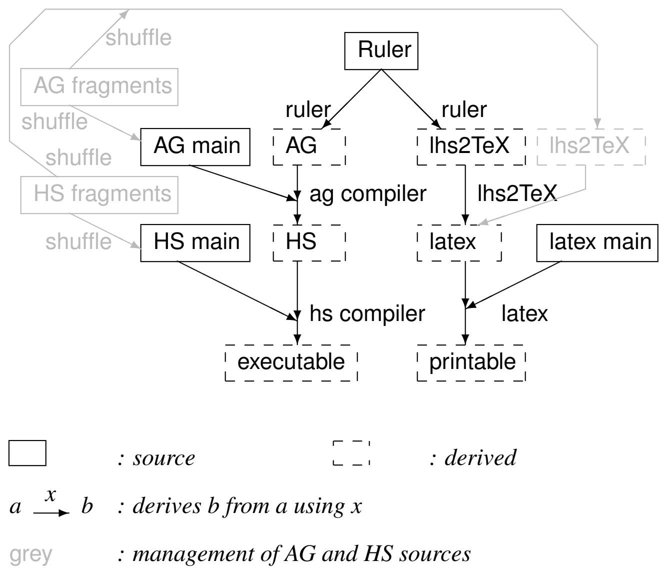

Figure 4: Ruler overview

```
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 5: High level structure of Ruler source

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 [8]), 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 displayed in grey (in Fig. 4). Our approach is similar to, but also different from literate programming; we will discuss this in Section 8.

The design of ***Ruler*** In the remainder of this section we discuss the concepts used in *Ruler* by inspecting elements of figures 1, 2 and 3.

The design of *Ruler* is driven by the need to check the following properties of type rules:

- All judgements match the structure it should have. For example, in Fig. 1 all judgements for an expression should match the structure of an expression judgement in the box at the top of the same figure.
- All identifiers used in a type rule are defined.
- For all the rules in a set of rules displayed together in a figure the conclusion should be of the structure in the box at the top of the figure.

Other properties can be added to this list, but we limit ourselves to this list and the requirement of output generation for different targets.

The structure of a judgement is described by a *scheme*. Each scheme consists of a set of *views* on the scheme. A view on a scheme consists of named *holes* and a set of templates referring to these holes. Such templatse, called *judgeshape*s, come in two varieties:

- A *judgespec*, used to specify a judgement.
- A *judgeuse*, used to display a judgement for an output target.

Rules are grouped into *rulesets*. A ruleset corresponds to a figure like Fig. 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. Each rule in a ruleset consists of views on the rule. Each view on a rule consists of a set of judgements for the premises and a judgement for the conclusion. Each of these judgements, called *judge*, follows of a particular scheme, and consists of bindings of hole names (of its scheme) to *Ruler* expressions.

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. 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:

- A judgement in 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 given a value.
- A judgement in 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 need 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 (*relation*) are treated like regular judgements, but their implementation has to be specified explicitly.

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. 6). A short overview of the type related notation is included in Fig. 8. Our example language contains e.g. the following program:

$$\begin{array}{l}\textbf{let}\ id = \lambda x \rightarrow x \\ \textbf{in}\ \ \textbf{let}\ v_1 = id\ 3 \\ \qquad \textbf{in}\ \ \textbf{let}\ v_2 = id\ id \\ \qquad\qquad \textbf{in}\ \ v_2\ v_1\end{array}$$

Values (expressions, terms):

$e ::=$	int	literals
$\mid$	i	program variable
$\mid$	$e\ e$	application
$\mid$	$\lambda i \rightarrow e$	abstraction
$\mid$	$\textbf{let}\ i = e\ \textbf{in}\ e$	local definitions

Figure 6: Terms

The type language for our example term language is given in Fig. 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.

Types:		
τ ::=	Int	literals
|	v	variable
|	$\tau \to \tau$	abstraction
σ ::=	$\forall \overline{v}.\tau$	universally quantified type, $\overline{v}$ possibly empty

Figure 7: Types

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* which can be seen as a representation for the more specific type information with which a type variable (representing unknown type information) will be substituted during HM type inferencing. This operation is denoted by the juxtapositioning of a C and a type σ:

$$\begin{aligned} C\ v &= \sigma, (v \mapsto \sigma) \in C \\ & v,\ \textit{otherwise} \\ C\ (\sigma_1 \to \sigma_2) &= C\ \sigma_1 \to C\ \sigma_2 \\ C\ (\forall v.\sigma) &= \forall v.(C \backslash v)\ \sigma \end{aligned}$$

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 8: Legenda of type related notation

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. 1 using *Ruler* notation. The full *Ruler* syntax is given in Fig. 9 and Fig. 10. The rules in Fig. 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 combined with 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 paper 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 paper 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 paper) 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. Judgements have a structure of their own, described *schemes*. A scheme plays the same role in rules as types do for expressions 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**). The view E on scheme *expr* is defined by:

view E =
  holes $[\,|\; e : Expr, gam : Gam, ty : Ty \;|\,]$
  judgespec $gam \vdash e : ty$
  judgeuse tex $gam \vdash ..$`"e"` $e : ty$

Here we specified 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 7. 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. The text for a *Ruler* expression already appears in pretty printed form throughout this paper, but in the original source code (included in the appendix) 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. 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. 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. 1. LaTeX

```
⟨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 9: Syntax of ruler notation (part I)

```
⟨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 10: Syntax of ruler notation (part II)

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); for a complete description see appendix A :

rule *e.int* =

view *E* =
  –
  judge *R* : *expr* = *gam* ⊢ *int* : *Ty_Int*

Before discussing its components, we repeat its LaTeX rendering from Fig. 1 to emphasize the similarities between the rule specification and its visual appearance:

$$\frac{}{\Gamma \vdash^{e} int : Int}\ \textsc{e.int}_{E}$$

All views of a rule are jointly defined, although we present the various views separately throughout this paper. 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 *lhs2TeX* [23]. 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. 1):

$$\frac{\Gamma \vdash^{e} a : \tau_a \quad \Gamma \vdash^{e} f : \tau_a \rightarrow \tau}{\Gamma \vdash^{e} f\ a : \tau}\ \textsc{e.app}_E$$

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* → *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:

$$\frac{i \mapsto \sigma \in \Gamma \quad \tau = inst\ (\sigma)}{\Gamma \vdash^{e} i : \tau}\ \textsc{e.var}_E$$

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* ↦ *ty* ∈ *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)
```

5 Extending to an algorithm

In this section we demonstrate the usefulness of views and incremental extension by adapting the equational rules from Fig. 1 to the algorithmic variant in Fig. 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. 2.

Fig. 2 not only shows the adapted rules but also shows the differences with the previous view by using colors. 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). 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 = $E < A < AG$

The view hierarchy declaration defines the *A* view to be built on top of view *E*, and *AG* again on top of *A*. 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 instance 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. 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 ⊢ .."e" e : ty ⤳ 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 paper). 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. 2 (value *cnstr.inh* shows as C^k by means of additional formatting information):

$$\frac{}{C^k; \Gamma \vdash^e int : Int \leadsto C^k}\ \text{E.INT}_A$$

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. 2) thus becomes:

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

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 7.

6 Target language: attribute grammar

In this section we give a brief overview of the AG system used as the target language for *Ruler* to generate code for. *Ruler* generates a partial implementation expressed as an attribute grammar (AG); we discuss this in the next section. We present as much as is required for an understanding of the next section; more can be found elsewhere [6, 12]. This section can safely be skipped by those who are familiar with our AG system.

An attribute grammar describes computations over an AST by means of attributes. An AST is a data structure similar to data types in Haskell. For example, part of the AST required for our type rule implemention is defined as follows:

```
data Expr
  | App f   : Expr
        a   : Expr

  | Int int : {Int}
  | Var i   : {String}
```

This AST for an *Expr* node defines *alternatives* (or *variants*, *productions*) for application, integer constants and use of variables respectively. The application alternative *App* has two *Expr children*, whereas *Int* and *Var* have a *field* holding the integer and identifier respectively. In the context of an alternative the node itself is called the *parent*.

An *attribute* holds the value of a computation; it has a name, a type and is defined as inherited (before the first vertical '|'), synthesized (after the second vertical '|') or both (in between both '|'). In AG code we define attributes for a node, for example for *Expr*:

```
attr Expr [g : Gam | c : C | ty : Ty]
```

Our AG system and *Ruler* use similar notation for attribute and hole definitions. For example, attribute *ty* is synthesized and has type *Ty*.

We define the value of an attribute for each alternative of a node by specifying an *attribute equation* for the synthesized attributes of the parent and the inherited attributes of all children. For example, for the *Int* alternative of *Expr* we define the value of the *ty* attribute of the parent (referred to by **lhs**) to be *Ty_Int*:

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

Each attribute equation is of the form

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

A ⟨*node*⟩ may be:

- **lhs**: reference to parent.
- ⟨*child*⟩: reference to child.

- **loc**: reference to a local attribute. The scope of a local attribute is the alternative it is declared in.

Our implementation (based on [7, 18]) uses Haskell expressions to define values for an attribute. From within these Haskell expressions we refer to attributes by means of the notation @$\langle \mathit{node} \rangle$.$\langle \mathit{attr} \rangle$:

- @**lhs**.$\langle \mathit{attr} \rangle$: reference to (inherited) $\langle \mathit{attr} \rangle$ of parent.
- @$\langle \mathit{child} \rangle$.$\langle \mathit{attr} \rangle$: reference (synthesized) $\langle \mathit{attr} \rangle$ of $\langle \mathit{child} \rangle$.
- @$\langle \mathit{attr} \rangle$: reference to a local attribute $\langle \mathit{attr} \rangle$.

For example, the following combines this notation (where ≻ is a Haskell operator for applying constraints as a substitution):

```
sem Expr
   | App (f.uniq, loc.uniq1)
                 = rulerMk1Uniq @lhs.uniq
         loc.tv_ = Ty_Var @uniq1
         f   .c  = @lhs.c    -- may be omitted
         f   .g  = @lhs.g    -- may be omitted
         a   .c  = @f.c      -- may be omitted
             .g  = @lhs.g    -- may be omitted
         (loc.c_, loc.mtErrs)
                 = ((@a.ty) `Ty_Arr` (@tv_)) ≅ (@a.c ≻ (@f.ty))
         lhs.c   = @c_ ≻ (@a.c)
             .ty = @c_ ≻  @a.c ≻ (@tv_)
```

In this fragment

- @**lhs**.*c* refers to the *c* attribute of the parent which is passed on to the *c* attribute of child *f*.
- *tv_* is defined locally and is referred to by @*tv_* in an expression for @**lhs**.*ty*.

Additionally, the AG notation allows the following notational variations:

- For a sequence of attribute equations defining a value for the same $\langle \mathit{node} \rangle$, only the first one needs to mention $\langle \mathit{node} \rangle$. In the example this has been done for *a* but not for *f*.
- The rules for *f* and *a* may be omitted anyway as the AG system uses built-in copy rules for attributes for which no equation has been given. We omit the details; the intuition is that values of attributes with the same name are copied top-to-bottom (if inherited), bottom-to-top (if synthesized) and left-to-right (if inherited + synthesized).

- AG allows pattern matching for tuples. For example, **loc**.*c_* and **loc**.*mtErrs* are defined via AG's pattern matching notation.

- AG fragments may be specified at textually different locations. The AG system gathers all fragments.

7 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 throughout 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.

- We mention in passing the need to parse input into an AST as well as to produce output from the AST and to make the result of type analysis available.

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 choosing 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 ∀, *Ty_Arr* represents the function type →, *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 $ty.a \to tv$ to $ty.a$ ‘*Ty_Arr*‘ tv:

rewrite ag def $a \to r = (a)$ ‘*Ty_Arr*‘ (r)

A rewrite declaration specifies a pattern (here: $a \to r$) 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 paper, 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 $ty.a \to tv$ 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 $(a \mid Ty) \to (r \mid Ty) = ((a)$ ‘*Ty_Arr*‘ $(r) \mid Ty)$

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. 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 $(c1 \mid C)\ (c2 \mid C)\ (v \mid a)$
$\qquad = (c1 \succ c2 \succ (v) \mid a)$

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" that is required for fresh type variables is to simulate a global seed for unique values. *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
```

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. We repeat the translation of rule E.APP as an example:

```
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_
```

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:

```
rulerMk1Uniq :: ⟨X⟩ → (⟨X⟩, ⟨Y⟩)
rulerMk1Uniq = ...
```

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):

```
newtype UID = UID [Int] deriving (Eq, Ord)
uidStart = UID [0]
rulerMk1Uniq :: UID → (UID, UID)
rulerMk1Uniq u @(UID ls) = (uidNext u, UID (0 : ls))
uidNext :: UID → UID
uidNext (UID (l : ls)) = UID (l + 1 : ls)
```

When a rule contains multiple occurrences of **unique**, *Ruler* assumes the presence of *rulerMk⟨n⟩Uniq* which returns ⟨*n*⟩ unique values; ⟨*n*⟩ is the number **unique** occurrences. We have omitted the declaration and initialisation of attribute `uniq`.

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 ⟨*attrs*⟩ ‘=‘⟨*expr*⟩.

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.

8 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 [3, 22] 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 jardly 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*:

- 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. We already have started to develop a plugin architecture for this kind of extensibility.

Related work We are not aware of other work which aims at consistency and understandability of compiler artefacts. However, we feel that the POPLmark challenge [5] and the Programmatica project [4] are relevant to *Ruler*:

- The POPLmark challenge aims at accompanying papers on programming languages by machine-checked proofs. This overlaps with our *Ruler* system which can be easily extended to generate input for external verification systems.
- The Programmatica project provides mechanisms and tools for proving properties of Haskell programs.

As mentioned in our future research, in both cases we envision many useful extensions to *Ruler* for the generation of material for theorem provers. We want to stress however that, already in its current state, we have found it an indispensable tool in keeping the formal description of a whole series of compilers and the associated implementations consistent. Ruler can be downloaded as part of EHC [10].

References

[1] Hugs 98. `http://www.haskell.org/hugs/`, 2003.

[2] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2004.

[3] Literate Programming. `http://www.literateprogramming.com/`, 2005.

[4] Programmatica Project (WWW site). `http://www.cse.ogi.edu/PacSoft/projects/-programatica/default.htm`, 2005.

[5] The POPLmark Challenge. `http://www.cis.upenn.edu/group/proj/plclub/mmm/`, 2005.

[6] Arthur Baars. Attribute Grammar System. `http://www.cs.uu.nl/groups/ST/Center/AttributeGrammarSystem`, 2004.

[7] Richard S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21:239–250, 1984.

[8] Atze Dijkstra. EHC Web. `http://www.cs.uu.nl/groups/ST/Ehc/WebHome`, 2004.

[9] Atze Dijkstra. *Haskell, Step by Step (to be published).* PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.

[10] Atze Dijkstra. Ruler. `http://www.cs.uu.nl/wiki/Ehc/Ruler`, 2005.

[11] Atze Dijkstra and S. Doaitse Swierstra. Explicit implicit parameters. Technical Report UU-CS-2004-059, Institute of Information and Computing Science, 2004.

[12] Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an Attribute Grammar (to be published). In *Advanced Functional Programming Summerschool*, LNCS. Springer-Verlag, 2004.

[13] Atze Dijkstra and S. Doaitse Swierstra. Making Implicit Parameters Explicit (submitted to POPL06), 2005.

[14] Dominic Duggan and John Ophel. Type-Checking Multi-Parameter Type Classes. *Journal of Functional Programming*, 2002.

[15] Karl-Filip Faxen. A Static Semantics for Haskell. *Journal of Functional Programming*, 12(4):295, 2002.

[16] Karl-Filip Faxen. Haskell and Principal Types. In *Haskell Workshop*, pages 88–97, 2003.

[17] 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.

[18] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, pages 154–173, 1987.

[19] Mark P. Jones. *Qualified Types, Theory and Practice*. Cambridge Univ. Press, 1994.

[20] Mark P. Jones. Typing Haskell in Haskell. `http://www.cse.ogi.edu/ mpj/thih/`, 2000.

[21] Mark P. Jones and Simon Peyton Jones. Lightweight Extensible Records for Haskell. In *Haskell Workshop*, number UU-CS-1999-28. Utrecht University, Institute of Information and Computing Sciences, 1999.

[22] D.E. Knuth. Literate Programming. *Journal of the ACM*, (42):97–111, 1984.

[23] Andres Loh. lhs2TeX. `http://www.cs.uu.nl/people/andres/lhs2tex/`, 2004.

[24] Simon Peyton Jones. *Haskell 98, Language and Libraries, The Revised Report*. Cambridge Univ. Press, 2003.

[25] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997.

[26] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

A Source code using *Ruler* notation

The preambles are specific to the generation for this paper because of the need for additional preprocessing to include pretty printed fragments of the *Ruler* source code in this paper.

```
preamble tex "%include lhs2TeX.fmt\n%include afp.fmt\n"
preamble ag "%%[0\n%include lhs2TeX.fmt\n%include afp.fmt\n%%]\n"
external Ty_Int
format tex Ty_Int = Int
format tex Gam = Gamma
format tex gam = Gamma

format tex ty = tau
format tex pty = sigma
format tex mty = tau
format tex tv = v

format tex cnstr.inh = Cnstr..k
format tex cnstr.syn = Cnstr
format tex cnstr     = Cnstr
format ag cnstr     = c
format ag gam       = g
rewrite ag def  (a | Ty) -> (r | Ty) = ((a) `Ty_Arr` (r) | Ty)
rewrite ag def  (c1 | Cnstr) (c2 | Cnstr) (v | a)
                  = (c1 |=> c2 |=> (v) | a)
rewrite ag def  (c | Cnstr) (v | a) = (c |=> (v) | a)
rewrite ag def  (i :-> (t|Ty)) = ([(i,t)] | Gam)
rewrite ag def  (g1 | Gam), (g2 | Gam) = ((g1) ++ (g2) | Gam)
viewhierarchy = E < A < AG < 3
scheme expr "Expr" =
  view E =
    holes [ | e: Expr, gam: Gam, ty: Ty | ]
    judgespec gam :- e : ty
    judgeuse tex gam :-.."e" e : ty
  view A =
    holes [ e: Expr, gam: Gam | thread cnstr: Cnstr | ty: Ty ]
    judgespec cnstr.inh ; gam :-.."e" e : ty ~> cnstr.syn
    judgeuse - tex
  view AG =
    holes [ node e: Expr | | ]
ruleset expr.base scheme expr "Expression type rules" =
  rule e.int "Int" =
    view E =
      -
      judge R : expr = gam :- int : Ty_Int
    view A =
      -
      judge R : expr
```

```
        | cnstr.syn  = cnstr.inh
        | cnstr.inh  = cnstr.inh
rule e.var "Var" =
  view E =
    judge G : gamLookupIdTy = i :-> pty `elem` gam
    judge I : tyInst = ty `=` inst(pty)
    -
    judge R : expr = gam :- i : ty
  view A =
    -
    judge R : expr
        | cnstr.syn = cnstr.inh
rule e.app "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
  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
  view AG =
    -
    judge R : expr
        | e = ((node 1 = f) (node 2 = a))
rule e.lam "Lam" =
  view E =
    judge B : expr = ((i :-> ty.i) , gam) :- b : ty.b
    -
    judge R : expr = gam :- (\i -> b) : (ty.i -> ty.b)
  view A =
    judge V : tvFresh = tv
    judge B : expr
        | cnstr.syn = cnstr.b
        | gam = (i :-> tv) , gam
    -
    judge R : expr
        | ty = cnstr.b tv -> ty.b
        | cnstr.syn = cnstr.b
    view AG =
      -
      judge R : expr
          | e = \i -> (node 1 = b)
  rule e.let "Let" =
    view E =
      judge D : expr = gam :- e : ty.e
      judge B : expr = ((i :-> pty.e), gam) :- b : ty.b
      judge G : tyGen = pty.e `=` ty.e \\ gam
      -
      judge R : expr = gam :- (let i `=` e in b) : ty.b
    view A =
      judge V : tvFresh = tv
      judge D : expr
          | cnstr.syn = cnstr.e
          | gam = (i :-> tv) , gam
      judge B : expr
          | cnstr.inh = cnstr.e
          | cnstr.syn = cnstr.b
      judge G : tyGen
          | gam = cnstr.e gam
      -
      judge R : expr
          | cnstr.syn = cnstr.b
    view AG =
      -
      judge R : expr
          | e = let i `=` (node 1 = e) in (node 2 = b)
relation match =
  view A =
    holes [ ty.l: Ty, ty.r: Ty | | cnstr: Cnstr ]
    judgespec ty.l <=> ty.r ~> cnstr
    judgeuse ag (cnstr,mtErrs) `=` (ty.l) <=> (ty.r)
relation gamLookupIdTy =
  view E =
    holes [ | nm: Nm, gam: Gam, ty: Ty | ]
    judgespec nm :-> ty `elem` gam
  view AG =
    holes [ nm: Nm, gam: Gam | | ty: Ty ]
    judgeuse ag (ty,nmErrs) `=` gamLookup nm gam
relation tvFresh =
  view A =
    holes [ | | tv: Ty ]
    judgespec tv
    judgeuse tex tv (text "fresh")
    judgeuse ag tv `=` Ty_Var unique
relation tyInst =
  view E =
    holes [ | ty: Ty, ty.i: Ty | ]
    judgespec ty.i `=` inst(ty)
  view AG =
    holes [ ty: Ty | | ty.i: Ty ]
    judgeuse ag ty.i ‘=‘ tyInst unique (ty)
relation tyGen =
  view E =
    holes [ ty: Ty, gam: Gam | | pty: Ty ]
    judgespec pty ‘=‘ ty \\ gam
    judgeuse tex pty ‘=‘ forall v..._ ‘.‘ ty, ^^^ v..._ ‘notElem‘ ftv(gam)
  view A =
    judgeuse tex pty ‘=‘ forall (ftv(ty) \\ ftv(gam)) ‘.‘ ty
  view AG =
    judgeuse ag  pty ‘=‘ mkTyAll (ftv(ty) \\ ftv(gam)) (ty)
```

B Supporting AG

```
{
module Main
where
import IO
import Data.List
import UU.Pretty
import UU.Parsing
import UU.Scanner
import UU.Scanner.Position (initPos,Pos)
import DemoUtils

}
{
}
INCLUDE "demo.ag"
WRAPPER AGItf
{
-- main
main :: IO ()
main
  = do { txt <- hGetContents stdin
       ; let tokens = scan ["let", "in"] ["->", "=", "\\"]
                           "()" "\\->=" (initPos "") txt
       ; pres <- parseIOMessage show pAGItf tokens
       ; let res = wrap_AGItf pres Inh_AGItf
       ; putStrLn (disp (pp_Syn_AGItf res) 200 "")
       }

-- Parser
pAGItf :: (IsParser p Token) => p T_AGItf
pAGItf
  = pAGItf
  where pAGItf    =   sem_AGItf_AGItf <$> pExpr
        pExprBase =   pParens pExpr
                  <|> sem_Expr_Var          <$> pVarid
                  <|> (sem_Expr_Int . read) <$> pInteger
        pExprApp  =   foldl1 sem_Expr_App   <$> pList1 pExprBase
        pExprPre  =   sem_Expr_Let
```

```
                      <$ pKey "let" <*> pVarid
                      <* pKey "="   <*> pExpr <* pKey "in"
                  <|> sem_Expr_Lam
                      <$ pKey "\\"  <*> pVarid
                      <* pKey "->"
        pExpr     =   pExprPre <*> pExpr
                  <|> pExprApp
}
-- AST
DATA AGItf
  | AGItf  e  : Expr
DATA Expr
  | App  f    : Expr
         a    : Expr
  | Int  int  : {Int}
  | Var  i    : {String}
  | Lam  i    : {String}
         b    : Expr
  | Let  i    : {String}
         e    : Expr
         b    : Expr
-- Initialisation
SEM AGItf
  | AGItf  e  .   g  =   []
              .   c  =   []
-- Pretty printing
ATTR AGItf Expr [ | | pp: PP_Doc ]

SEM Expr
  | App  lhs  .   pp   =   @f.pp >#< pp_parens @a.pp
                           >-< mkErr @mtErrs
  | Int  lhs  .   pp   =   pp @int
  | Var  lhs  .   pp   =   pp @i >-< mkErr @nmErrs
  | Lam  lhs  .   pp   =   "\\" >|< pp @i >#< "->" >#< @b.pp
  | Let  lhs  .   pp   =   "let"     >#< pp @i
                           >#< "::"  >#< show @pty_e_
                           >#< "="   >#< @e.pp
                           >-< "in " >#< @b.pp
-- Uniq
ATTR Expr [ | uniq: UID | ]

SEM AGItf
  | AGItf  e  .   uniq  =   uidStart
```

C Supporting Haskell

Type, unification

```
module DemoUtils
where
import Data.List
import UU.Pretty

-- Error
mkErr :: [PP_Doc] -> PP_Doc
mkErr [] = empty
mkErr p  = "<ERR:" >#< vlist p >|< ">"
-- Unique identifier
newtype UID = UID [Int] deriving (Eq,Ord)
uidStart = UID [0]

rulerMk1Uniq :: UID -> (UID,UID)
rulerMk1Uniq u@(UID ls) = (uidNext u,UID (0:ls))

uidNext :: UID -> UID
uidNext (UID (l:ls)) = UID (l+1:ls)
mkUIDs :: UID -> [UID]
mkUIDs = iterate uidNext

instance Show UID where
  show (UID l)
    = concat .  intersperse "_" . map show . reverse $ l
-- Type
type TvId  = UID
data Ty    =  Ty_Any | Ty_Int | Ty_Var TvId
           |  Ty_Arr  Ty Ty
           |  Ty_All  [TvId] Ty
              deriving (Eq,Ord)
mkTyAll tvs t = if null tvs then t else Ty_All tvs t

instance Show Ty where
  show Ty_Any         = "?"
  show Ty_Int         = "Int"
  show (Ty_Var v)     = "v" ++ show v
  show (Ty_All vs t)  = "forall" ++ concat (map ((' ':) . show) vs)
                         ++ " . " ++ show t
  show (Ty_Arr t1 t2) = "(" ++ show t1 ++ ") -> " ++ show t2
-- Gam
type Gam = [(String,Ty)]

gamLookup :: String -> Gam -> (Ty,[PP_Doc])
gamLookup n g
  = maybe (Ty_Any,[n >#< "undefined"]) (\t -> (t,[]))
  $ lookup n g
-- Constraints
type Cnstr = [(TvId,Ty)]

class Substitutable a where
  (|=>)  ::  Cnstr -> 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 `union` ftv t2
  ftv  _                =  []
instance Substitutable Cnstr where
  s1 |=> s2 = s1 ++ map (\(v,t) -> (v,s1 |=> t)) s2
  ftv       = foldr union [] . map (ftv . snd)
```

```
instance Substitutable Gam where
  s |=> g  = map (\(i,t) -> (i,s |=> t)) g
  ftv      = foldr union [] . map (ftv . snd)
-- Type matching (unification)
(<=>) :: Ty -> Ty -> (Cnstr,[PP_Doc])
Ty_Any          <=> t2          = ([],[])
t1              <=> Ty_Any      = ([],[])
Ty_Int          <=> Ty_Int      = ([],[])
Ty_Var v1       <=> Ty_Var v2
    | v1 == v2                  = ([],[])
Ty_Var v1       <=> t2
    | v1 `notElem` ftv t2       = ([(v1,t2)],[])
t1              <=> Ty_Var v2
    | v2 `notElem` ftv t1       = ([(v2,t1)],[])
Ty_Arr a1 r1    <=> Ty_Arr a2 r2
  = (sr |=> sa,ea ++ er)
  where (sa,ea) = a1 <=> a2
        (sr,er) = (sa |=> r1) <=> (sa |=> r2)
t1              <=> t2          = ([],["could not match"
                                       >#< show t1 >#< "with"
                                       >#< show t2]
                                  )
-- Type instantiation
tyInst :: UID -> Ty -> Ty
tyInst u (Ty_All vs t) = c |=> t
                       where c = zipWith (\v u -> (v,Ty_Var u))
                                         vs (mkUIDs u)
tyInst _ t             = t
```

D Generated AG

```
ATTR Expr [g: Gam | c: Cnstr | ty: Ty]
SEM Expr
  | Int  lhs  .  ty  =  Ty_Int
SEM Expr
  | Var  (lhs.uniq,loc.uniq1)
                        =  rulerMk1Uniq @lhs.uniq
         (loc.pty_,loc.nmErrs)
                        =  gamLookup @i @lhs.g
         lhs  .  ty     =  tyInst @uniq1 @pty_
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_
SEM Expr
  | Lam  (b.uniq,loc.uniq1)
                       =  rulerMk1Uniq @lhs.uniq
         loc  .  tv_   =  Ty_Var @uniq1
         b    .  g     =  ([ (@i , @tv_) ]) ++ @lhs.g
         lhs  .  ty    =  (@b.c |=> @tv_) `Ty_Arr` @b.ty
SEM Expr
  | Let  (e.uniq,loc.uniq1)
                       =  rulerMk1Uniq @lhs.uniq
         e    .  g     =  ([ (@i , Ty_Var @uniq1) ]) ++ @lhs.g
         b    .  g     =  ([ (@i , mkTyAll (ftv @e.ty \\ ftv (@e.c |=> @lhs.g)) @e.ty) ]) ++ @lhs.g
```