Thesis Proposal

Tamar Christina - 3019721
Graduate Student
University of Utrecht
`tnfchris@students.cs.uu.nl`

June 21, 2010

1 Thesis

The Utrecht Haskell Compiler (UHC[1]) implements most of it is type checking code using Attribute Grammars, more specifically using the tool **uuagc**[2]. However two important parts are not written using Attribute Grammars: unification and context reduction, this is due to the limitations of attribute grammar systems. With the development of **ruler-front**[3] it is now possible to implement a higher-rank polymorphic type inferencing algorithm using only Attribute Grammars. The big question that we aim to provide an answer to is

> *Can a higher-rank polymorphic type inferencing algorithm be implemented using only Attribute Grammars and if so can it be done in such a way that it would be simpler to understand and correspond closer to typing rules.*

The primary focus of this thesis is on the tools and not the type system itself.

2 Motivation

Type systems for languages such as Haskell are generally hard to prove correct and implement. Most of the literature use typing rules to describe the type system. Unfortunately these typing rules for a type system (including syntax directed typing rules) do not give you an idea on how to implement the associated inference algorithms needed, because these typing rules usually contain a large amount of non-determinism and implicit assumptions that need to be taken into account. Once implemented it is also much harder to prove the correctness of the algorithm. The implementation usually does not resemble the original typing rules. This unfortunately means we can no longer use the original proofs that were made for the typing rules to prove the inference algorithm correct.

We expect by implementing the UHC type system in the new **ruler-front** system that it would be

1. Easier to understand because we can use the machinery provided by Attribute Grammars to hide most of the implementation details such as tree traversals and threading of values around the tree.

2. Easier to prove by having the implementation coincide to the typing rules for the system by using the expressiveness of **ruler-front**.

3. Easier to generate documentation for, by virtue of having a simpler implementation.

3 Related Research

There have been multiple attempts in the past to simplify the implementations of type systems, these include but not limited to:

- Using monads[6] to provide some level of abstraction (most notable Read,Writer and Logger monads)

- Using Guesses[4] to deal with non-determinism

- The Ruler[5] system used in UHC in the past mixed with Attribute Grammars

However most of these attempts do not try and do this using Attribute Grammars.

The current UHC implementation however is the only([1]) project that uses Attribute Grammars but that specific implementation of Attribute Grammars is as mentioned before not flexible enough to describe the entire type inference algorithm with.

Classic **uuagc** provides us a mechanism to define attributes for every node in the AST to be traversed and then subsequently fill them in and use them to calculate our result values without having to worry about the order of the tree walks or the tree walks themselves. It also provides a mechanism that allows us to not have to specify certain rules, like threading of values around in the tree walks.

ruler-front inherits this flexibility but extends it with a way to reason about a specific visit (or tree walk) and to also make decisions based on information gathered so far. This is the extra expressiveness we want to leverage in order to re-implement the type system.

Since the aim is to implement a higher-rank polymorphic type system there are three main approaches to look at for inspiration and guidance:

1. Flexible Types: Robust type inference for first-class polymorphism - *Daan Leijen*
2. FPH: First-class Polymorphism for Haskell - *Dimitrios Vytiniotis, Stephanie Weirich and Simon Peyton Jones*
3. Practical type inference for arbitrary-rank types - *Simon Peyton Jo Jones, Dimitrios Vytiniotis, Stephanie Welrich and Mark Shields*

These are all extensions of the popular Hindley-Milner which itself can be described as a restrictive version of System F. Restrictive because It can type less expressions than System F but has the benefit that inference algorithm is complete, sound and decidable.

3.1 The basics

As mentioned before these systems are all extensions of Hindley Milner (some more conservative than others) and they all try to attain the some of the expressiveness of System F (namely higher-rank types) while still keeping the algorithms decidable. For that reason a short introduction into Hindley Milner and System F might be useful.

3.1.1 System F

System F or otherwise known as the polymorphic lambda calculus is an extension of the simply typed lambda calculus with constructs for

Terms
- Type abstraction ($\Lambda x.t$)
- Type application (t [T])

Values Type abstraction values ($\Lambda X.t$ as opposed to the simply typed lambda calculus in which only abstractions were values)

Types
- Type variable (X)

- Universal types ($\forall X.T$)

Environment It also provides an environment for the mapping between type variable and their binding.

These addition together with their typing rules allows one to do type reconstruction on polymorphic expressions.

It allows the abstraction of types out of terms and subsequently fill them in later, for example:

$$\text{id} = \Lambda X.\lambda x : X.x$$

which states that the type of id is $X \rightarrow X$ where X is the type variable bound by the newly introduced type abstraction.

If we were to want to construct the type for "id 3" (assuming the type of 3 is Int) we would get

$$\text{id [Int]} = [\text{X}\rightarrow\text{Int}](\lambda x : X.x)$$

Applying the substitution gives us that the type of id when applied to an Int should be $Int \rightarrow Int$.

Universal types are used to capture the dependency that the type of the resulting type actually depends on the type we pass it, for instance the type of the *id* above is $\forall X.X \rightarrow X$

System F thus introduces and formalizes parametric polymorphism by adding mechanisms to reason about type variables, unfortunately its type inferencing while very powerful is undecidable.

3.1.2 Hindley-Milner

Another type system which supports polymorphic functions is the Hindley-Milner. Its inference algorithm Damas-Milner is decidable primarily due to Hindley-Milner being more restrictive than System F. System F allows more types than Hindley-Milner but in order to type check it requires annotations by the programmer.

The Hindley-Milner system works on the bases of *principle types.* Each well-types term has a unique "best" type such that all other types possible for the term can be constructed from the *principle type.*

For example the principle type of the *id* function mentioned before is $a \rightarrow a$.

Damas-Milner works by collecting type variable constrains (or substitutions) from how the terms are used in the expression and subsequently solving this constraint set by means of unification. If this is not possible then the term is considered to be ill-typed.

consider the following (admittedly simple) example where we wish to type bar:

```
foo :: String → Int
bar x y = foo x + y
```

The first step is to annotate bar with fresh type variables

```
bar x::X y::Y = foo x + y :: Z
```

There is now already have some extra information about bar, it is determined that it has the form $X \rightarrow Y \rightarrow Z$ how ever it is still do not known what X, Y and Z are. In order to find out what these are the body of bar is examined for more information and a constraint set is constructed.

> Notice that x is passed as an argument to foo, which suggests that x has to be of type String, a constraint $[x \rightarrow String]$ is added to the constrains set.
>
> The next observation is that the addition operator + is used to add the result of foo x and y. This follows from the type of + and foo x that y should be of type Int, another constraint $[y \rightarrow Int]$ is added to the set. It is now also known what Z is, as it is the result of the addition which is also an Int so $[z \rightarrow Int]$ is added to the set.
>
> Using the constraint set $[X \rightarrow String,\ Y \rightarrow Int,\ Z \rightarrow Int]$ the type for bar can now be constructed which is $String \rightarrow Int \rightarrow Int$.

In the case of functions with polymorphic types, the resulting type will still contain variables for which the constraint set had no bindings to concrete types. In these cases the type of the fresh variables are perceived to be polymorphic variables in the type of the term.

3.2 FPH[10]

This is an extension to the Dama-Milner type inference algorithm to provide first class polymorphism for Haskell which relies on System F types and (see above) and does so while maintaining decidability.

The following example does not type check in standard Hindley-Milner:

```
f push = (push 3, push "hello")
```

because it requires the type of the variable *push* to be able to accept arguments of both *String* and *Int*. This is a typical example of a higher-rank function.

This system allows two distinct forms of first class polymorphism which can be seen by the more involved example from the paper[10]:

```
($)   :: forall a. b (a→b) → a → b
runST :: forall r. (forall s. ST s r) → r
foo   :: forall s. Int → ST s Int
```

Every type variable is explicitly bound by an outward universal quantification ∀ (forall) quantification as per System F (remember that FPH uses System F types) this is called a rank-1 type. With higher rank types we mean types or rank > 1. the *runST* example above has a rank-2 argument which can be seen by the universal quantification inside the parenthesis. In order for *...(runST $ foo 4)...* to type check this system thus needs to allow for two constructs:

1. That functions can have higher-rank types and thus able to take a polymorphic type argument

2. That type arguments can be instantiated to a quantified polymorphic type. This is called *impredicativity*.

Impredicative Allows a polymorphic function to be instantiated with a polytype.

FPH requires little help from the programmer in the form of annotations. It only requires type annotations on *let*-bindings or λ-abstractions. Function calls may require a lot of impredicative instantiations but never requires and explicit type annotation.

The type system distinguishes between Damas-Milner types (types permitting up to rank-1 types) and *rich* types (higher rank types).

For instance $Int \rightarrow Int$ and $\forall a.a \rightarrow a$ are Damas-Milner types, but $\forall b.(\forall a.a \rightarrow a) \rightarrow b$ is a *rich* type.

FPH as mentioned before is an extension of Damas-Milner but due to allowing higher-rank polymorphism and impredicativity we lose the ability to type an expression with a single best fitting type (principle type). This brings with it some added complications in that the algorithm can no longer determine one single type and use it throughout the program where the function is in scope.

This is where the programmer is required to help the type system out by for instance providing annotations, for example the earlier defined function f f push = (push 3, push ”hello”)

would pose a problem, since there is no one best type that describes f. both the types $(\forall a.a \rightarrow a) \rightarrow (Int, String)$ and $(\forall a.a \rightarrow Int) \rightarrow (Int, Int)$ are valid and so there is no one type that describes them all. If the programmer would annotate the function with $f :: (\forall a.a \rightarrow a) \rightarrow (Int, String)$ then there would no longer be any ambiguity and the type system can check against the provided type. The burden on the programmer can also be eased by allowing partial type annotations, this would for instance only require the programmer to annotate f with $(\forall a.a \rightarrow Int) \rightarrow ...$ since ”push” was the cause of the ambiguity. Another method is to allow annotations on abstractions, in this case only the ”push” variable above would have to be annotated.

FPH allows impredicative instantiations in a controlled fashion since when a binding is ambiguous (as in, can be instantiated to two or more incomparable types) this causes a problem since we do not know which type to use. The idea presented in FPH is then to explicitly mark the types of bindings that can lead to such ambiguity with *boxes*. The allowed types are

$$\begin{aligned} \tau' &::= \alpha \mid \tau_1' \rightarrow \tau_2' \mid \mathrm{T}\ \tau' \mid \boxed{\mathrm{o}} \\ o &::= \forall \alpha.o \mid \alpha \mid o \rightarrow o \mid \mathrm{T}\ o \end{aligned}$$

The idea is to allow quantifications (and so polymorphic types) but only inside the box $\Box$ and govern their use with a set of 3 *Ideas*[10]

1. A polymorphic function is instantiated with boxy monotypes which indicate places in the type where ”guessing” is required to fill in a type in order to type check.

2. When comparing types discard all boxes. This means that there is no need to annotate function applications, since even though one of the argument may emit a boxy type, we can compare it with an unboxed type.

3. No boxy type can enter the environment. This means any quantified type in the environment is a programmer supplied type and so there is no ambiguity anymore so this is fine. This also means that when a function can be instantiated to both a Damas-Milner type and a System F type that the Damas-Milner type is chosen because the boxy System F type cannot enter the environment.

These are the three central ideas that are used to modify the original Hindley-Milner rules in order to support higher-rank types, along with rules for boxy instantiations, generalizations and subsumption.

3.3 Flexible Types[11]

This paper introduces HML which is another type system that supports first class polymorphism. In contrast to FPH which needed annotations on *Let*-bindings and λ-abstractions, HML requires only arguments with polymorphic types to be annotated. It is a simplification of another type system MLF. MLF is a very expressive inference system and only requires annotations on function parameters that are used at two or more sites.

HML in contrast to MLF uses only the so called *flexible* types, which are a more elaborate take on principle typing (more on this later).

It also has the same notational requirement as MLF namely only on function parameters. HML is a extension to Hindley-Milner and because it uses only flexible types puts most of the burden of the implementation on the unification algorithm.

Annotation is only needed thus when defining a function with functional parameters

$$\text{f} = \lambda(\forall \text{push} :: \alpha \to \alpha).\ (\text{push } 3, \text{push "hello"})$$

Functions and data structures can be used freely without any further explicit annotation. Another property of HML is that it is not sensitive to the order of applications (neither is FPH).

Consider the following example from the paper[10]:

$$\begin{array}{ll} \textit{inc} & :: \text{Int} \to \text{Int} \\ \textit{single} & :: \forall \alpha.\ \alpha \to \text{List } \alpha \\ \textit{append} & :: \forall \alpha.\ \text{List } \alpha \to \text{List } \alpha \to \text{List } \alpha \\ \textit{map} & :: \forall \alpha\beta.\ (\alpha \to \beta) \to \text{List } \alpha \to \text{List } \beta \end{array}$$

The following expression type checks just fine under HML

```
let ids = single id
in (map f ids, append (single inc) ids)
```

But would require *ids* to be two incomparable System F types at the same time, namely List (Int $\to$ Int) and List ($\forall \alpha.\ \alpha \to$ List α). This is where *flexible* types come in, the type of ids can now be described as $\forall(\beta \geq \forall \alpha \to \alpha)$. List β, which can be interpreted as "any β that is an instance of $\forall \alpha \to \alpha$ the type List β is valid". This can be considered as the most "principle" or general type of that expression.

These *flexible* types are the key HML, but flexible types are only allowed on *Let*-bindings (see f) and values of a flexible type cannot be passed as arguments.

A couple of new types are introduced by HML, the most important ones are:

Types These are the normal System F types called σ in HML except that the function constructor $(\rightarrow)$ is not treated specially but instead is just used in the general constructor type "c $o_1 \ldots o_n$". A consequence of this general treatment of constructors is that HML does not do subtyping.

Mono types Same as in Hindley-Milner except with the same non-special treatment of $(\rightarrow)$ as mentioned above.

Type schemes Type scheme are the core part of *flexible* types. In the example of *ids* above the type scheme was "$\forall(\beta \geq \forall\alpha \rightarrow \alpha)$. List β".

The structure of a type scheme φ is:

1. $\forall(\alpha \geq \varphi_1).\varphi_2$
2. σ
3. $\bot$

The φ_1 above is called the *quantification bounds* and by varying this we can create two interesting schemes of which the first is the *trivially quantified* scheme where φ_1 is a monotype, e.g. $(\forall\alpha \geq Int).\alpha \rightarrow \alpha)$ which is equivalent to just $Int \rightarrow Int$. Schemes of this kind cannot be entered into the *Prefix*

The second variation is when the *bounds* φ_1 is $\bot$. This is called the *unconstrained bound* since it means that α can be instantiated to any type. This is usually shortened to just $\forall\alpha$ since the *bounds* no longer has any use.

Prefix The prefix Q is defined as a list of quantifiers from a type scheme. $\alpha_1 \geq \hat{\varphi}_1, \ldots, \alpha_n \geq \hat{\varphi}_n$ the $\hat{\varphi}$ here refers to the restriction mentioned before that a *trivial* type cannot be entered into the Prefix. All the types variables $\alpha_1 \ldots \alpha_n$ are assumed to be distinct since a scheme should not bind multiple types.

Quantified schemes The types $\hat{\varphi}$ allowed in Q.

$\hat{\varphi} ::= \forall(\alpha \geq \hat{\varphi}_1).\hat{\varphi}_2$ with $\alpha \in ftv(\hat{\varphi}_2) | \bot$

The type rules in HML are almost the same as the ones in Hindley-Milner, except that in the presence of *flexible* types the Instantiation rule is slightly different and every rule now gets an explicit *Prefix.*

$$\text{Var:}\quad \frac{x : \varphi \in \Gamma}{Q, \Gamma \vdash x : \varphi}$$

The var rule is still very straight forward, if there is a binding for x in the environment Γ with the type φ then return the type of φ for x under the same environment Γ and prefix Q.

$$\text{Inst:}\quad \frac{Q, \Gamma \vdash e : \varphi_1 \quad Q \vdash \varphi_1 \sqsubset \varphi_2}{Q, \Gamma \vdash e : \varphi_2}$$

The instantiation rule states that we can always use a instance of a type under the prefix Q as the type of the expression. This instance of relation is denote by the $\sqsubset$ relation above which states that φ_2 is an instance of φ_1 under the prefix Q

$$\text{Gen:}\quad \frac{(Q, \alpha \geq \hat{\varphi}_1), \Gamma \vdash e : \varphi_2 \quad \alpha \notin ftv(\Gamma)}{Q, \Gamma \vdash e : \forall(\alpha \geq \hat{\varphi}_1).\varphi_2}$$

The generalization rule generalizes a type by moving it's bounds out from the prefix Q into the type φ.

$$\text{App:}\quad \frac{Q, \Gamma \vdash e_1 : \sigma_2 \to \sigma \quad Q, \Gamma \vdash e_2 : \sigma_2}{Q, \Gamma \vdash e_1\ e_2 : \sigma}$$

The application rule is the standard application rule over types. It requires that the type of e_2 be equal to the type of the argument of e_1. Note that this ranges over type and not typeschemes.

$$\text{Let:}\quad \frac{Q, \Gamma \vdash e_1 : \varphi_1 \quad Q, (\Gamma, x : \varphi_1) \vdash e_2 : \varphi_2}{Q, \Gamma \vdash \textbf{Let}\ x = e_1\ \textbf{in}\ e_2 : \varphi_2}$$

The Let rule is pretty straight forward, given an expression e_1 with type φ_1 and a variable x which in the environment Γ has the same type φ_1 and an expression e_2 with type φ_2 we can create a let binding **Let** x = e_1 **in** e_2.

$$\text{Fun:}\quad \frac{Q, (\Gamma, x : \tau) \vdash e : \sigma}{Q, \Gamma \vdash \lambda x.e : \tau \to \sigma}$$

The Fun rule restricts the type of the parameter to a monomorphic type τ in order to avoid having to guess polymorphic types. This does not mean that a polymorphic type can not be infered since the type schemes are hidden in the prefix Q.

$$\text{Fun-Ann:} \quad \frac{Q,(\Gamma, x:\sigma_1) \vdash e : \sigma_2}{Q, \Gamma \vdash \lambda(x :: \sigma_1).e : \sigma_1 \to \sigma_2}$$

The Function Annotation rule is used to provide a mechanism to explicitly annotate abstraction variables with a (possibly) polymorphic type. Like annotation of the *push* example above with the needed higher-rank type.

In particular the *FUN* rule for lambda expressions restrict the type of the parameters to be a mono type τ since any scheme on the mono type is in the Prefix Q, this is the secrete behind HML.

3.4 Practical type inference for arbitrary-rank types[9]

This approach is a restricted version of a Type system for arbitrary rank types from Odersky & Laufer[12] and uses element from the work of Pierce & Turner[13]. The first is about annotating terms and functions with *type schemes* which are the usual type annotations we know today in Haskell. This enabled annotation of arguments with a higher-rank type. The latter describes a mechanism for *bidirectional typing* which allows types of parameters of annonymous functions (lambdas) and *local synthesis of type arguments* which finds the principe (best) type of polymorphic applications when one exists.

The restrictions in this paper are used to limit the amount of annotation needed in order to type a higher-rank program since the original Odersky & Laufer[12] requires alot of programmer annotation.

By combining the two approaches it is possible to annotate just one clauses of a function and have the type information propagated along so that you do not have to explicitly annotate every argument in every clause.

Comparable to the previously discussed systems, because this system is a conservative extension of Hindley-Milner means that any program typeable in Hindley-Milner is still typeable without any additional annotation. Annotation is only needed on higher-rank types.

It also uses an idea of *subsumption* to compare types and see if one is an instance (or more polymorphic) of the other type. An example is

$$\begin{aligned}
&\text{g} :: ((\forall \beta.[\beta] \to [\beta]) \to \mathit{Int}) \to \mathit{Int} \\
&\text{k1} :: (\forall \alpha.\alpha \to \alpha) \to \mathit{Int} \\
&\text{k2} :: ([\mathit{Int}] \to [\mathit{Int}]) \to \mathit{Int}
\end{aligned}$$

The expression *(g k1)* is ill-typed since *k1* expects a more polymorphic type ($\forall \alpha.\alpha \to \alpha$) than g can pass to it ($\forall \beta.[\beta] \to [\beta]$).

Subsumption if σ_1 is *less* polymorphic than σ_2 then $\sigma_1 \to \tau$ is *more* polymorphic than $\sigma_2 \to \tau$

To compare types that are isomorphic but not directly comparable due to the location of the foralls. Such as

$$\forall \alpha\beta.\alpha \rightarrow \beta \rightarrow \beta \leq \forall \alpha.\alpha \rightarrow (\forall \beta.\beta \rightarrow \beta)$$

deep skolemisation is done. The idea of *deep skolemisation* is that we preprocess the second type in the comparison and float all the $\forall$s that appear to the right of a top level arrow to the left.

Such directly incomparible types arise from the use of *eager generalisation* which is defined as the generalisation of every lambda body in the abstraction tules so that the $\forall$'s in the resulting type appear as far right as possible in the type.

In terms of the types the only difference between this system and Hindley-Milner is in the intermediate type ρ, which in the system discussed here adds notation to allow arbitrary ranked types.

The typing rules have also been slightly modified to handle this but this system is not impredicative.

4 Approach

Since the goal is to see whether we can implement a first class polymorphic type system using only Attribute Grammars we choose to use Haskell as the source language and would like to compare the resulting implementation with that of the one currently implemented in UHC which also uses Attribute Grammars as stated above. The internal structures of UHC will be used, most notably EH(Essential Haskell) which is the desugared and simplified internal representation of Haskell used inside UHC for type inferencing and checking.

In terms of which algorithms to use the choice would be between *FPH* and HML^F a restriction on *HML* where only let bindings and lambda abstractions with higher-rank types may require annotation.

HML^F is chosen as the algorithm to implement because of it is expressiveness when compared to *FPH*, but also because unification in HML^F is quite a lot more involved. Unifications was one of those parts that couldn't be implemented in pure AGs before, so having a more complicated example would be a good test.

We propose to make use of the existing UHC Parsers and EH abstract syntax. This has a couple of advantages:

1. Input can be entered using the concrete syntax of Haskell which is much easier to type and scale than if we were to write it directly in the

abstract syntax EH. It provides an easier way to enter much larger examples later on while still leaving the possibility to enter EH manually should we choose to.

2. It would be easier later on to swap the existing type system with the proposed one should it prove beneficial.

3. The existing UHC test files can be re-used to directly compare the two implementations using the same programs.

4. Since the abstract syntax of the to be implemented system would be a derivation, the comparison will for the most part be between how the type checking and inferencing was implemented in **uuagc** and **ruler-front**.

We plan to first implement the typesystem using the supplied unification and inference algorithms. The reason for this is that the typing rules alone are not enough to get to a working implementation since they contain alot of subtle and implicit assumptions. By first implementing the unification algorithm we can get a much better understanding of how th system is supposed to work and we get a reference point for comparison later on. Since the unification algorithm has already been proven correct by the author, we can use it as a comparison baseline for the version we will later implement using only the typing rules.

5 Deliverables

At the end we expect to have a working type checker implementation in **ruler-front** for at least the aforementioned features, proof of correctness and soundness for the implementation, and a comparison between the current UHC type checker and the newly implemented one.

6 Evaluation

Since the focus is on the tools and not the type system, evaluation should be done by comparing the current UHC type inference algorithm with the proposed one in terms of:

- How easy it is to prove correct
- Readability and understandability
- Size of implementation

6.1 Correctness

The use of this would be two folds: It would be trivial to make a non-correct type system, but in order for us to be able to compare the implementations we must be sure that the implementation is correct.

And it would also be a nice way to see how far removed the implementation is from the type system specification and how much of the original proofs we can reuse.

6.2 Readability

We compare how easy the actual implementation code is to understand having understood the typing rules associated with it. The primary focus of this comparison is to see how far removed the specification and implementation are.

6.3 Size

This is a bit of a rudimentary measurement, but it is useful because it gives a notion of the ease of implementation. In general it gives a measure of how much we had to specify and how much the copy rules automatically did for us.

7 Planning

The planning is pretty basic and shows only the main activities involved

Due by	**Description**
June 22	Setting up environment and create the needed new types
Sept 22	Unification implementation
Sept 29	verify correctness of unification
Okt 16	type inferencing implementation
Nov 20	implementation based on typing rules
Dec 4	verifying implementation
Dec 11	Partial type signatures
Jan 8	Evaluate implementation
Feb 8	Thesis written

Table 1:

Because EH is more elaborate than the system presented in HML^F it is a good idea to first inspect the differences a bit more closely and decide if any new rules need to be made or if most of the difference are syntactic sugar.

The most difficult part of this would be unification, which is also one of the parts that couldn't be done in AGs before. In addition some difficulty is also expected in solemnization, generalization and instantiation because like unifications these all work based on the unknown structure of a type and unknown information in the Schemes Q.

8 Possible Extras

As future work or possible amendments to the current planned feature sets is to implement Type classes , GADTs and dependent types. A possible approach for doing so is using work derived form the OutsideIn(X)[8] algorithm which uses a local constraint solver to implement the above mentioned features. It would be interesting to see if using **ruler-front** would be convenient to express and implement this kind of algorithm in.

References

[1] Utrecht Haskell Compiler Structure *Structure Documentation* `http://www.cs.uu.nl/wiki/bin/view/Ehc/EhcStructureDocumentation`

[2] UUAGC *"Utrecht Attribute Grammar System"* `http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem`

[3] Arie Middelkoop, Atze Dijkstra, S.Doatse Swierstra *Iterative Type Inference with Attribute Grammars* Utrecht University

[4] Arie Middelkoop, Atze Dijkstra, S.Doatse Swierstra, Lucilia Camarao de Figueiredo *Controlling Non-Determinism in Type rules using First-Class Guessing* Utrecht University, Universidade Federal de Ouro Preto

[5] Atze Dijkstra, S. Doaitse Swierstra *Ruler: Programming Type Rules* LNCS FLOPS 2006 proceedings of FLOPS 2006

[6] Arie Middelkoop, Atze Dijkstra, S.Doatse Swierstra *Factoring Type Rules Into Aspects* Utrecht University

[7] Atze Dijkstra *Stepping through Haskell* ISBN 90-393-4070-6

[8] Dimitrios Vytiniotis, Simon Peyton-Jones, Tom Schrijvers, Martin Sulzann *Modular type inference with local assumptions* Microsoft Research, Katholieke Universiteit Leuven, Informatik Consulting Systems AG. Submitted to Journal of Functional Programming.

[9] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Welrich and Mark Shields *Practical type inference for arbitrary-rank types* Microsoft Research, University of Pennsylvania, Microsoft

[10] Dimitrios Vytiniotis, Stephanie Weirich and Simon Peyton Jones *FPH: First-class Polymorphism for Haskell* University of Pennsylvania and Microsoft Research

[11] Daan Leijen *Robust type inference for first-class polymorphism* Microsoft Research

[12] Martin Odersky and Konstantin Laufer *Putting Type Annotations to Work* University of Karlsruhe and Loyola University Chicago

[13] Benjamin C. Pierce and David N. Turner *Local Type Inference* Indiana University and Technology Transfer Center