Controlling Non-Determinism in Type Rules using First-Class Guessing

Arie Middelkoop Atze Dijkstra
S. Doaitse Swierstra
Universiteit Utrecht
{ariem, atze, doaitse}@cs.uu.nl

Lucília Camarão de Figueiredo
Universidade Federal de Ouro Preto
lucilia@dcc.ufmg.br

Abstract

Given a type system written as a collection of type rules, we investigate the automatic derivation of inference algorithms from these rules. A minor challenge are the side effects of a rule, which need to be expressed algorithmically. A major challenge are non-deterministic aspects of rules that cannot be directly mapped to an algorithm.

We present Ruler, a language for type inferencers, to meet these challenges. An inferencer is written as a collection of rules with side conditions explicitly expressed in Haskell, and with annotations for the scheduling of the rules.

This paper includes an extensive case study of an inferencer for the "First Class Polymorphism for Haskell" type system (Vytiniotis et al. 2008).

1. Introduction

A type system is "a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of value they compute" (Pierce 2002). Given a type system, it is often not immediately clear whether there exists an algorithm that can automatically infer a valid type for a (type correct) program. More specifically, if the type system has principal types, is there an algorithm that can infer the most general type of each expression?

Most type systems have a declarative specification in the form of a collection of type rules. How to effectively and efficiently use these rules for building type correctness proofs is a separate issue, and having a systematic way in building such type inferencers from such a collection of rules is still an open issue. The benefits of having such a method are:

Consistency. A strong coupling between formal description and implementation makes it easier to show how certain properties proved for the type system carry over to the inferencer.

Rapid prototyping. Experimenting with type systems leads to a deeper understanding. However, language developers are currently discouraged to do so as it is cumbersome to write inferencers from scratch. A framework will relieve programmers from this burden and hence support rapid prototyping.

Abstraction. Interacting language features obscure and complicate language semantics. In many inference algorithms, the unification procedure makes the essential decisions about non-deterministic aspects. This requires context-information to be carried to and into the place where unification is performed, and complicates the inferencer. Instead, we would like to be able to deal with the decision making process at the places where the non-deterministic aspects occur in the type rules.

Documentation. In comparison with the type rules, the type inference algorithms are often not completely documented and explained. Often they are specified by a concrete (and sometimes obscure) implementation.

This paper shows that it is possible to semi-automatically obtain inference algorithms from type rules. We do not get them entirely for free. A major problem is that type rules generally contain non-deterministic aspects. For example, more than a single rule may be applicable at the same time. Even when the rules are syntax directed, they may state demands in the form of side conditions about a type (or other value), of which concrete information is not easily available from the context.

A solution to this challenge are *annotations* which control the scheduling of the resolution of non-deterministic aspects by manipulating guesses. A *guess* is an opaque value representing a derivation that has not been constructed yet. It also serves as a place holder for a concrete value. We can pass such a guesses around, observe them, impose requirements on them. When a sufficient number of requirements have been accumulated, the actual value of the guess is revealed and we can attempt to construct the derivation.

Therefore, we contribute the following:

- We present a typed domain specific language for type inferencers called Ruler. One of its distinguishing features is the possibility to provide annotation for type rules. Also, side expressions are expressed using conventional Haskell code.
- We give examples of increasing complexity of inferencers for type systems with non-deterministic aspects, and show how manipulating guesses leads to their resolution (Section 2). We demonstrate the power of these annotations by providing an inferencer for the type system of FPH (Vytiniotis et al. 2008) that is directly based on FPH's collection of declarative type rules (Section 2.4).
- We formalize the notation (Section 4), the operational semantics (Section 5) and the static semantics (Appendix A) of Ruler.
- We discuss the rationale of our design compared to prior work on the construction of type inferencers (Section 3).

- We have proof-of-concept Haskell-based implementation for a meta-typed front-end in which inferencer rules with custom syntax can be encoded. Furthermore we provide an executable version of the operational semantics which interprets the inference rules and produces a derivation in terms of the original type rules for all expressions it manages to type.

The reason that we have chosen Haskell as the target language for our generated inferencers are:

- We can use the expressiveness of Haskell for writing the semantics of side conditions in type rules.
- There are many libraries available for Haskell that provide efficient data structures and external constraint solvers.
- We can integrate the inferencer with other Haskell projects, in particular the Utrecht Haskell Compiler (Dijkstra et al. 2007). We have compiler technology readily available (parsers, tree-walk generators, pretty printers, etc.) to facilitate rapid prototyping.

2. Examples

In this section we show how to use Ruler in describing a series of type inferencers of increasing complexity. We took the examples such that each example builds on the previous one. We start from an inferencer for the explicitly typed lambda calculus in Section 2.1. Admittedly, the inferencer in this case does only type checking, but we use it to informally introduce the Ruler inferencer language (formally in Section 4) and informally describe its evaluation model (formally in Section 5). Then, in Section 2.2, we move on to an inferencer for implicitly typed System F, in which several cases of non-determinism arise. Finally, we show the inferencer for FPH in Section 2.4, which demonstrates the expressive power of the annotations.

For each example we show the type rules and the actual inferencer code. As they have a tight resemblance, be warned not to confuse the two!

Syntax:

$$\begin{array}{rcl} e & = & x \mid f\ a \mid \lambda(x :: \tau).\, e \mid \mathbf{let}\ x = e\ \mathbf{in}\ b \mid \mathbf{fix}\ f \\ \tau & = & \alpha \mid \tau_1 \to \tau_2 \end{array}$$

Rules:

$$\frac{(x,\tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{var} \qquad \frac{\Gamma \vdash f : \tau_f \qquad \Gamma \vdash a : \tau_a \qquad \tau_f \equiv \tau_a \to \tau_r}{\Gamma \vdash f\ a : \tau_r}\ \textsc{app}$$

$$\frac{x :: \tau_x, \Gamma \vdash e : \tau}{\Gamma \vdash \lambda(x :: \tau_x).e : \tau_x \to \tau}\ \textsc{lam.expl} \qquad \frac{\Gamma \vdash e : \tau_x \qquad x :: \tau_x, \Gamma \vdash b : \tau}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ b : \tau}\ \textsc{let}$$

$$\frac{\Gamma \vdash f : \tau_f \qquad \tau_f \equiv \tau \to \tau}{\Gamma \vdash \mathbf{fix}\ f : \tau}\ \textsc{fix}$$

Figure 1: Type system for explicitly typed lambda calculus.

2.1 Explicitly Typed Lambda Calculus

Figure 1 gives the type system for the explicitly typed lambda calculus (Church 1940; Pierce 2002). The rules define a *relation* between an environment Γ, expression e, and type τ.

For the inferencer, this corresponds to a *function* (we call it a *scheme*) that takes an environment Γ and expression e as inputs and produces a valid type τ, if there exist such a type according to the rules. The Ruler code of the inferencer of this type system is given in Figure 2. We discuss each part further below.

Scheme declarations:

$$\begin{array}{lll} \mathsf{tc}: & & (\Gamma \lhd \mathsf{Map\ String\ Ty}) \vdash (e \lhd \mathsf{Expr}) \ :\ (\tau \rhd_{\mathsf{d}} \mathsf{Ty}) \\ \mathsf{fr}: & \forall\alpha. & (v \rhd_{\mathsf{d}} \alpha)\ \text{fresh} \\ \mathsf{lk}: & \forall\alpha\beta. & (\,(k \lhd \alpha)\,,\,(v \rhd \beta)\,) \in (\Gamma \lhd \mathsf{Map}\ \alpha\ \beta) \end{array}$$

Inferencer rules:

$$\frac{(x,\tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{var} \qquad \frac{v \leftarrow \mathit{lookup}\ x\ \Gamma}{(x,v) \in \Gamma}\ \textsc{lookup}$$

$$\frac{\Gamma \vdash f : \tau_f \qquad \Gamma \vdash a : \tau_a \qquad \tau_r\ \text{fresh} \qquad \tau_f \equiv \tau_a \to \tau_r}{\Gamma \vdash f\ a : \tau_r}\ \textsc{app} \qquad \frac{\mathsf{defer}_v\ [\,\emptyset\,]}{v\ \text{fresh}}\ \textsc{fresh}$$

$$\frac{x :: \tau_x, \Gamma \vdash e : \tau}{\Gamma \vdash \lambda(x :: \tau_x).e : \tau_x \to \tau}\ \textsc{lam.expl}$$

$$\frac{\Gamma \vdash e : \tau_x \qquad x :: \tau_x, \Gamma \vdash b : \tau}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ b : \tau}\ \textsc{let} \qquad \frac{\tau\ \text{fresh} \qquad \Gamma \vdash f : \tau_f \qquad \tau_f \equiv \tau \to \tau}{\Gamma \vdash \mathbf{fix}\ f : \tau}\ \textsc{fix}$$

Syntax and semantics:

```
data Ty = TGuess GuessVar   data Expr = EVar String
        | TConst GuessVar             | EApp Expr Expr
        | TArr   Ty  Ty               | ...
instance Container Ty where
  appSubst rec (TArr f a) = rec f ≫ rec a
  deferVars (TConst _)    = empty
  deferVars (TGuess v)    = single v
  deferVars (TArr a r)    = deferVars a ‘union‘ deferVars r
instance Unifyable Ty where
  unify rec (TArr f a) (TArr g b) = rec f g ≫ rec a b
  unify _   _          _          = fail "type error"
instance Deferrable Ty where
  mkDeferValue                = TGuess
  mkFixedValue                = TConst
  matchDeferValue (TGuess v)  = Just v
  matchFixedValue (TConst v)  = Just v
pattern Map String Ty where  x :: τ,Γ input   insert x τ Γ
pattern Expr          where  λx . e  output ELam x e
  -- other patterns omitted.
```

Figure 2: Inferencer for explicitly typed lambda calculus.

Scheme declarations. To obtain an inferencer in Haskell, we actually want a Haskell function *tc* with the type: *Map String Ty* → *Expr* → *I Ty* where *I* is some monad encapsulating failure and state. Thus, concerning the meta variables, we need to know whether they serve as inputs or outputs, and what their meta type is. This is information is given in the scheme declaration. It defines the name of the function (i.e. *tc*), the syntax of the function call in the inferencer rules (scheme instantiation), the names and types of the meta variables, whether a meta variable is an input (◁) or an output (▷), and a optional property d or u of a meta variable. In this case, the d-property requires that we supply an instance of *Deferrable* for the Haskell type *Ty*, and allows us to use the defer statement on types (to be explained later).

Inferencer rules. The inferencer rules provide the actual definition of the scheme. They consist of an ordered sequence of *statements*, related to the *premises* of the type rules, and a concluding statement. Such a statement can be:

- A scheme invocation, i.e. $(x, \tau) \in \Gamma$, which executes the corresponding function with the given parameters when evaluated.
- Haskell code in the *I* monad. This code is used to express side-conditions of type rules as statements in the inferencer rules. For example, the type system in Figure 2 implicitly mentions a lookup-relation in the VAR rule. This is explicitly defined in our inferencer code by means of some Haskell code in the LOOKUP rule.
- An equality statement, i.e. $\tau_f \equiv \tau_a \to \tau_r$, stating that its two inputs will be the same after type inference has finished.
- Non-determinism annotations, such as defer (explained later).

The rules represent the actual definition of cases for functions *tc*, *lk*, and *fr*. For example, the APP, LOOKUP, and FRESH inferencer rules are projected to concrete Haskell code as follows:

```
tc_app Γ e                         lk_lookup x Γ
    = do let (EApp f a) = e          = do v ← lookup x Γ
         τ_f ← tc Γ f                     return v
         τ_a ← tc Γ a                 lk = lk_lookup
         τ_r ← fr                     fr_fresh = do (v, ()) ← defer f
         unif τ_f (TArr τ_a τ_r)                   return v
         return τ_r                   where f v′ = return ()
tc = tc_var ⊕ tc_app ⊕ ...         fr = fr_fresh
```

The equivalence statement gets translated to a monadic expression **unif** which is an API function provided by Ruler.

The statements are executed in the order of appearance. Each statement may fail, causing the entire rule to fail. Rules that fail due to pattern matches or equality statements at the very beginning of the statement sequence allow other applicable rules to be applied (offering a limited form of backtracking). Otherwise, the failure is turned into an abort of the entire inference, with a type error as result.

Non-determinism. A major recurring problem is that not all relations are functions. Sometimes a meta variable is required to be both an input and an output. For example, in the inferencer rule APP, the value τ_r needs to be produced before it can be passed to the equality statement. It is also an output of the rule, not an input. This means that there is no indication how to obtain it. Therefore, we conceptually *guess* the value of τ_r. This value is kept hidden behind an opaque guess-value, and is only revealed when we actually discover what the value must be. The *fr*-scheme gives a function that produces these values.

Ruler accomplishes this as follows. The rule FRESH has a defer-statement, which is a non-determinism annotation. It is parametrized with a list of statement sequences, and produces an guess *v*. The statement sequences are not executed immediately, but a closure is created for them which is triggered once we discover concrete information about guess *v*. At that point, one of the statement sequences is required to execute successfully with *v* as an input and the current knowledge about guesses. The defer-statement in FRESH has only one statement sequence, the empty sequence, which always succeeds. In later examples we have non-trivial sequences of statements that allow us to defer and control decision making.

So, a guess needs to get produced for *v*. Ruler requires help in the form of a *Deferrable* instance on the type of *v* to construct this guess. Operationally, defer produces an opaque guess variable, which is wrapped into the domain of *v* by means of *mkDeferValue*. This guess is thus first class, and can be passed around and end up in other data structures. Ruler maintains information about these guess variables, such as the closures produced by defer. When a concrete value is discovered about a guess, all occurrences of this guess are replaced with this concrete value (thus revealing the guess). Again, Ruler requires help by means of a *Container* instance in order to deal with values holding guesses. Furthermore, the inferencer rules may check if certain values are still opaque variables and act on that. This is also something we exploit later.

Concrete values for a guess are discovered by executing equality statements. When comparing the two input values, if one value contains a guess and the other a concrete value, then we *commit* that concrete value to the guess. This leads to the execution of the deferrable statements. We call this commit because it is an irreversible action: a guess can be opaque for a while, a commit conceptually only uncovers it. If both values are guesses, the guesses are merged. In case both values are concrete values, Ruler requires help in the form of a *Unifyable* instance, of which *unify* is required to traverse one level through the values and check that their heads are the same. Another requirement on guesses is that the value committed to a guess may not contain the guess itself (the infamous *occur check*). Therefore, the function *deferVars* needs to be defined to tell Ruler which guesses are contained in a value.

Data semantics. The last part of the Ruler code consists of a definition of the data structures involved. One may also define custom syntax to be used in the rules, for which translations to either Haskell patterns (for inputs) or Haskell expressions (for outputs) need to be given. This custom syntax may be ambiguous as long as it can be resolved based on the meta types of the meta variables. Finally, we remark that these instances for data types are likely to be automatically generated from the structure of the data types, or readily available in a library with support code.

Before we continue, consider the addition of an extra rule to the inferencer:

$$\frac{\tau_x \text{ fresh} \qquad x :: \tau_x, \Gamma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau_x \to \tau}\ \text{LAM.IMPL}$$

With this addition, we obtain an inferencer for the simply typed lambda calculus. However, with this rule, there can be unresolved guesses remaining after we finish inferencing. For example, consider inferring the type of the expression $\lambda x\ .\ x$. We obtain $\tau \to \tau$, where τ is a guess. However, at the end of the inference, Ruler forces all remaining guesses to get evaluated. Those guesses that remain are mapped to a *fixed* value. A fixed value is an opaque value like a guess (created with *mkFixedValue*), except that it is only equal to itself and cannot be committed on.

2.2 Implicitly Typed System F

We give a sound but incomplete inferencer for implicitly typed System F (Reynolds 1974), using a relaxation of Milner's algorithm (Milner 1978) and exploiting type annotations to deal with higher-ranked types. Compilers such as GHC and UHC utilize inference algorithms based on this type system, which makes it an interesting case study. The inferencer algorithm described here is clearly inferior versus other algorithms in terms of completeness and predictability, but is powerful and simple enough to serve as a basis for the inference algorithm of the next section.

Implicitly typed System F (or polymorphic lambda calculus) extends the simply typed lambda calculus with two rules, and a more expressive type language (Figure 4). This change adds a lot of expressive power. Consider the following expressions (assuming for the moment that we have *Int*s in the type language):

Extra syntax:

$$\tau = \ldots \mid \forall\overline{\alpha}.\tau$$

Extra rules:

$$\frac{\Gamma \vdash e : \forall\overline{\alpha}.\tau_2}{\Gamma \vdash e : [\overline{\alpha := \tau_1}]\,\tau_2}\ \textsc{inst} \qquad \frac{\Gamma \vdash e : \tau_1 \qquad \overline{\alpha} \notin \mathit{ftv}\,\Gamma}{\Gamma \vdash e : \forall\overline{\alpha}.\tau_1}\ \textsc{gen}$$

Figure 3: Type system of implicitly typed System F.

$$\begin{array}{ll} f = \lambda(k :: (\mathit{Int} \to \mathit{Int}) \to \mathit{Int})\,. & k\,(\lambda x\,.\ x) \\ g = \lambda(k :: \forall\,\alpha\,.\ (\alpha \to \alpha) \to \mathit{Int})\,. & k\,(\lambda x\,.\ x) \end{array}$$

The definition of f can be typed within the simply typed lambda calculus, but g cannot. In g's case, the GEN rule is needed after typing $\lambda x\,.\ x$.

There is no simple way to translate these rules to the type inferencer rules. The problem lies in the decision when to apply these rules, because this is not specified by the syntax. They could be applied any time, even an arbitrary number of times. However, we choose to only apply instantiation once directly after the VAR rule, and generalization once for each let-binding, and once for the argument of each application. Figure 4 lists the inferencer rules. We partitioned the rules such that they belong either to scheme $\vdash$, $\vdash_x$, $\vdash_g$, or $\vdash_l$, and adapted the recursive invocations accordingly. This solves the problem of when to apply the rules. Also note that we have two versions of the GEN rule, one for the let-binding (GEN.LET), and one for the argument of an application (GEN.LAZY).

Inferencer rules:

$$\frac{(x,\tau) \in \Gamma}{\Gamma \vdash_x x : \tau}\ \textsc{var} \qquad \frac{\Gamma \vdash_x x : \forall\overline{\alpha}.\tau_2 \qquad \overline{\tau_1}\ \text{fresh}}{\Gamma \vdash x : [\overline{\alpha := \tau_1}]\,\tau_2}\ \textsc{inst.v}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1 \\ \mathsf{defer}_{\tau_2}\ [[\ \mathsf{let}\,(\forall\overline{\alpha}.\tau_2') = \tau_2,\ \tau_1 \equiv \tau_2',\ \overline{\alpha} \notin \mathit{ftv}\,\Gamma\ ]]\end{array}}{\Gamma \vdash_g e : \tau_2}\ \textsc{gen.lazy}$$

$$\frac{\begin{array}{c}\tau_r\ \text{fresh} \\ \Gamma \vdash f : \tau_f \qquad \Gamma \vdash_g a : \tau_a \\ \tau_f \equiv \tau_a \to \tau_r\end{array}}{\Gamma \vdash f\,a : \tau_r}\ \textsc{app} \qquad \frac{\begin{array}{l}\mathsf{fixate}_\tau\,[\ \Gamma \vdash e : \tau\ ] \\ \mathsf{let}\,\overline{\alpha} = \mathit{ftv}\,\tau - \mathit{ftv}\,\Gamma\end{array}}{\Gamma \vdash_l e : \forall\overline{\alpha}.\tau}\ \textsc{gen.let}$$

$$\frac{\begin{array}{c}\Gamma \vdash_l e : \tau_x \\ x :: \tau_x, \Gamma \vdash b : \tau\end{array}}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ b : \tau}\ \textsc{let}$$

Syntax and semantics:

$$\mathbf{data}\ \mathit{Ty} = \ldots \quad \mid \quad \mathit{TAll}\,[\mathit{GuessVar}]\ \mathit{Ty}$$

$$\begin{array}{ll} \mathit{ftv}\ (\mathit{TConst}\ v)\ = \mathit{single}\ v & \mathit{ftv}\ (\mathit{TArr}\ a\ r) = \mathit{ftv}\ a\ \text{`}\mathit{union}\text{`}\ \mathit{ftv}\ r \\ \mathit{ftv}\ (\mathit{TGuess}\ \_) = \mathit{empty} & \mathit{ftv}\ (\mathit{TAll}\ \mathit{vs}\ t) = \mathit{ftv}\ t\ \text{`}\mathit{difference}\text{`}\ \mathit{vs} \end{array}$$

Figure 4: Inferencer for implicitly typed System F.

However, this leads us back to the non-determinism problems that we encountered before. The INST.V rule requires us to choose which bound variables to instantiate, and what type to instantiate them to. Similarly, for both GEN rules, a decision needs to be made what variables to generalize over. These are all examples of non-deterministic aspects. We use the following tricks to resolve them:

- Instantiation (rule INST.V) is greedy and instantiates all bound variables that are know at the time when instantiation is applied. We use the *fr*-relation to guess the types to which they are instantiated.
- Generalization of the argument of an application (rule GEN.LAZY) is done on-demand. The result type τ_2 is guessed. At some point the head (or more) of τ_2 is discovered. In case of our example: for f we discover at some point that τ_2 is $\mathit{Int} \to \mathit{Int}$, and for g that it is $\forall\alpha.\alpha \to \alpha$. At that moment the deferred statements are triggered.

  When these statements trigger, the requirement is that enough information about the outermost quantifiers of τ_2 is known. Furthermore, with the greedy assumption about instantiation, assume that τ_1 does not have any outermost quantifiers. With this knowledge in mind, consider the GEN type rule again in Figure 3. The type rule tells us to take the portion of τ_1 without outermost quantifiers, this should then be equal to τ_2. In that case, the variables $\overline{\alpha}$ are not allowed to be in the environment. This is exactly what the deferred statements of GEN.LAZY establish.
- Generalization just before the let-binding is also greedy. It generalizes over all unbound variables in the type that are not in the environment. However, since a guess can represent an arbitrary type, we cannot generalize over them. Therefore, we introduce the fixate-statement. It is parametrized with a sequence of statements, and executes those. The guesses which are introduced during the execution and remain are forced to be evaluated. Those for which no concrete value is discovered are mapped to fixed types (*TConst* values). The order of this forcing is undefined. These *TConst* values are real type variables and can be generalized over (if free in the environment).

$$\frac{\begin{array}{c} q, \tau_x\ \text{fresh} \\ \mathsf{fixate}\left[\begin{array}{l}\mathsf{defer\_}\ [\tau_x \equiv \mathit{pick}\,q] \\ (x,\tau_x,q), \Gamma \vdash e : \tau\end{array}\right]\end{array}}{\Gamma \vdash \lambda x.e : \tau_x \to \tau} \qquad \frac{\begin{array}{c}(x,\tau_1,q) \in \Gamma \\ \mathsf{defer}_{\tau_2}[q'\ \text{fresh}, \mathsf{commit}_{(\mathit{last}\,q)}\,(\tau_2,q')] \\ \mathsf{defer}_{\tau_1'}\ [\ \tau_1' \leqslant \tau_2\ ] \qquad \tau_1 \equiv \tau_1'\end{array}}{\Gamma \vdash x : \tau_2}$$

We defer the instantiation $\leqslant$ until we have more information about all the types we want to instantiate the left-hand side to. Queue q is a nested product, where each left component is a type, and each right component is a queue. This queue is terminated with a guess. The queue stores all encountered values for the type of the lambda parameter. Each time such a value is encountered, it gets appended to the queue. When fixating the lambda term, the deferred statement executes that traverses the queue and picks out the most general type, and matches it with the type of the lambda parameter. This causes all deferred instantiations $\leqslant$ to execute. The $\leqslant$ relation is not affected by this complex scheduling.

Figure 5: Complex example: queuing all expected types.

Many variations on the above rules are possible that result in a more complete inference algorithm (although no complete inference algorithm exists). For example, making instantiation also happen on demand, or queuing up all guesses of the type of lambda parameters (see Figure 5) before making a final choice, thus emulating type propagation algorithms (Peyton Jones et al. 2007; Dijkstra and Swierstra 2006a). Such algorithms are normally very hard to implement, because with conventional approaches the unification algorithm has to deal with it all. However, with Ruler, such algorithms can now be described easily, and locally at the places in the rules where full context-information is available, with only minimal effects on modularity.

2.3 Summary

We have seen several examples of non-determinism that are problematic when writing an inference algorithm. In the end, these problems boil down to deferring decisions and controlling the decision process. We have seen and will see the following annotations to resolve them:

- defer introduces a guess with the promise that a sequence of statements will be executed when the guess is revealed.
- fixate introduces a scope for guesses and forces all guesses introduced in this scope to be resolved when leaving the scope.
- commit unveils the guess, causing the deferred statements to run.
- force is syntactic sugar for a commit with a fresh guess, followed by a commit with a fixed value if the result was still a guess.

The driving force behind propagating the type information that slowly comes available are the equality statements (≡).

2.4 Example: FPH

The FPH type system (Vytiniotis et al. 2008) is a restriction of implicitly typed System F, such that there exist a principle type for each binding, and a complete inference algorithm that finds these types. In this section, we give an alternative inference algorithm. Compare the FPH's declarative rules (Figure 6) with the inference algorithm (Figure 7) and be surprised how close the resemblance is.

Consider the following example where *choose* is instantiated predicatively and impredicatively:

$$\begin{array}{ll} f = \mathit{choose}\ \mathit{id} & \\ f :: \forall\,\alpha\ .\ (\alpha \to \alpha) \to (\alpha \to \alpha) & \text{-- predicative inst} \\ f :: (\forall\,\alpha\ .\ \alpha \to \alpha) \to (\forall\,\alpha\ .\ \alpha \to \alpha) & \text{-- impredicative inst} \end{array}$$

The observation underlying FPH is that impredicative instantiation may result in more than one incomparable most general System F type for a binding. This is undesired for reasons of modularity and predictability. FPH dictates that impredicative instantiation is forbidden if it has influence on the type of a binding.

To formalize this difference, FPH introduces the concept of a box. When a bound variable is instantiated with a polymorphic type in FPH, it is enclosed within a box. A box expresses that the type it encloses may have been obtained through impredicative instantiation. FPH forbids the type of a binding to have a box in the type, thus ensuring that these possible undesired effects have no influence on the type. This absence of boxes can arise due to two reasons:

- The type with the box is simply not part of the type of the binding.
- There is an unboxing relation (⪯⊑) that allows shrinking of boxes over the monomorphic spine of a type. When we discover that the type in the box cannot influence the type of the result, we can remove the box.
- The programmer can give an explicit type signature, which does not have boxes.

A particular invariant maintained by FPH is that there may not be a box within a box (the "monomorphic substitution" operator := takes care of this).

The type rules for FPH contain many non-deterministic aspects, especially due to the interaction between types and boxes. Both the structure of the types, and the demands on the boxes become only gradually available. In some cases, we may discover that the type is not allowed to have any boxes before the actual type becomes known. Alternatively, in case of box-stripping (⌊·⌋), we may know portions of the type structure, but nothing about the boxes yet.

Types with boxes:

$$\tau = \ldots \mid \boxed{\tau}$$

Type rules:

$$\frac{\begin{array}{c}\Gamma \vdash x : \forall\overline{\alpha}.\tau_2 \\ \tau_1 \text{ unboxed iff mono}\end{array}}{\Gamma \vdash x : [\overline{\alpha := \tau_1}]\,\tau_2}\ \textsc{inst} \qquad \frac{\begin{array}{c}\Gamma \vdash e : \tau_1 \\ \tau_1 \preceq\sqsubseteq \tau_2\end{array}}{\Gamma \vdash e : \tau_2}\ \textsc{subs}$$

$$\frac{\begin{array}{cc}\Gamma \vdash f : \tau_f & \Gamma \vdash a : \tau_a \\ \lfloor\tau_a\rfloor \equiv \lfloor\tau'_a\rfloor & \tau_f \equiv \tau'_a \to \tau_r\end{array}}{\Gamma \vdash f\,a : \tau_r}\ \textsc{app} \qquad \frac{\begin{array}{c}\Gamma \vdash e : \tau_1 \\ \lfloor\tau_1\rfloor \equiv \tau_2\end{array}}{\Gamma \vdash (e :: \tau_2) : \tau_2}\ \textsc{ann}$$

$$\frac{\begin{array}{c}\mathit{mono}\,\tau_x \\ x :: \tau_x, \Gamma \vdash e : \tau\end{array}}{\Gamma \vdash \lambda x.e : \tau_x \to \tau}\ \textsc{lam.impl} \qquad \frac{x :: \tau_x, \Gamma \vdash e : \tau}{\Gamma \vdash \lambda(x :: \tau_x).e : \tau_x \to \tau}\ \textsc{lam.expl}$$

$$\frac{\Gamma \vdash e : \tau_x \qquad \mathit{noBoxes}\,\tau_x \qquad x :: \tau_x, \Gamma \vdash b : \tau}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ b : \tau}\ \textsc{let}$$

Boxy instantiation rules:

$$\boxed{\forall\overline{\alpha}.\tau_1} \preceq \boxed{[\overline{\alpha := \tau_2}]\tau_1}\ \textsc{bi} \qquad \tau \preceq \tau\ \textsc{br}$$

Protected unboxing rules:

$$\frac{\mathit{mono}\,\tau}{\boxed{\tau} \sqsubseteq \tau}\ \textsc{tbox} \qquad \tau \sqsubseteq \tau\ \textsc{refl} \qquad \frac{\begin{array}{c}\tau_1 \sqsubseteq \tau_2 \\ \mathit{unboxed}\,\overline{\alpha}\,\tau_1 \\ \mathit{unboxed}\,\overline{\alpha}\,\tau_2\end{array}}{\forall\overline{\alpha}.\tau_1 \sqsubseteq \forall\overline{\alpha}.\tau_2}\ \textsc{poly}$$

$$\frac{\boxed{\tau_1} \sqsubseteq \tau_3 \qquad \boxed{\tau_2} \sqsubseteq \tau_4}{\boxed{\tau_1} \to \boxed{\tau_2} \sqsubseteq \tau_3 \to \tau_4}\ \textsc{conbox} \qquad \frac{\tau_1 \sqsubseteq \tau_3 \qquad \tau_2 \sqsubseteq \tau_4}{\tau_1 \to \tau_2 \sqsubseteq \tau_3 \to \tau_4}\ \textsc{cong}$$

Figure 6: The FPH type system.

Idea. We choose here to be able to guess the type independent of the boxes. Each alternative of a type gets a box annotation. Types τ are of the form $\boxed{\tilde{\tau}}_b$, where $\tilde{\tau}$ is a regular type with types τ as components, and b a box annotation. A box annotation is either concrete (*BYes* or *BNo*), or a guess. Types in the environment have box-annotations *BNo*, and box-annotations *BYes* are introduced by instantiation. The unboxing rules then relate boxes to types, and eliminate boxes as soon as more type information becomes available. This unboxing (see subscript s and Subs) is done for each sub-expression. For an application, only the instantiation of the function type can cause boxes to appear in the result type. Possible boxes in the type of the argument are stripped away (these would otherwise cause boxes inside boxes). Also, annotations are considered safe and causes boxes to disappear. Finally, at a let-binding, we first generalize and fixate the guesses in the types, and only then fixate the boxes. This ensures that when fixating the boxes, we know that choices of the boxes do not influence the types in the local scope anymore.

Note the following notation. A $[\![v]\!]$ represents a guess containing variable v (produced or obtained by means of *mkDeferValue* or *matchDeferValue* respectively). A type $\tilde{\tau}$ (Ty') at a place where a τ (Ty) is expected represents a pair of $\tilde{\tau}$ with a box annotation of *BNo*. A type $\boxed{\tilde{\tau}}$ represents a pair of $\tilde{\tau}$ with a box annotation of *BYes*.

The boxy-instantiation rules allows instantiation inside an outermost box. The application of these rules is controlled by Boxy.Inst, as follows. Decisions are delayed until more is known about the result type. Then we force the decisions to have been made about a potential box surrounding it. We then know which one the two actual instantiation rules is applicable. Note that we are not afraid of instantiation: the Gen.Lazy generalizes again if needed.

Boxy types:

```
type Ty    = (Ty′, Box)
data Box = BYes | BNo | BVar GuessVar
data Ty′   = TArr Ty′ Ty′ | TGuess GuessVar | ...
```

Scheme declarations:

$$(\tau_1 \lhd_{\mathrm{d}} Ty) \sqsubseteq (\tau_2 \rhd_{\mathrm{d}} Ty) \qquad (\tau_1 \lhd_{\mathrm{d}} Ty) \sqsubseteq' (\tau_2 \lhd_{\mathrm{d}} Ty)$$

$$(\tau_1 \lhd_{\mathrm{d}} Ty) \preceq (\tau_2 \rhd_{\mathrm{d}} Ty) \qquad (\tau_1 \lhd_{\mathrm{d}} Ty) \preceq' (\tau_2 \rhd_{\mathrm{d}} Ty)$$

Inferencer rules:

$$\frac{\begin{array}{c}\Gamma \vdash_x x : \forall\overline{\alpha}.\tau_2 \qquad \overline{\tilde{\tau}_1}\text{ fresh}\\ \mathsf{defer}_{b_i}\ \begin{array}{l}[\ [\![v]\!] = b,\ \mathsf{commit}_v\, BYes\ ],\\ [\ b = BNo,\ mono\,\tilde{\tau}\ ],\\ [\ b = BYes\ ]\end{array}\end{array}}{\Gamma \vdash x : [\overline{\alpha := \boxed{\tilde{\tau}_1}_b}]\,\tau_2}\ \textsc{inst.v} \qquad \frac{\begin{array}{c}\Gamma \vdash e : \tau_1\\ \tau_1 \preceq\!\sqsubseteq \tau_2\end{array}}{\Gamma \vdash_s e : \tau_2}\ \textsc{subs}$$

$$\frac{\begin{array}{c}\Gamma \vdash_s e : \tau_1\\ \ldots\end{array}}{\Gamma \vdash_g e : \tau_2}\ \textsc{gen.lazy} \qquad \frac{\begin{array}{cc}\Gamma \vdash_s f : \tau_f & \Gamma \vdash_g a : \tau_a\\ \lfloor\tau_a\rfloor \equiv \lfloor\tau'_a\rfloor & \tau_f \equiv \tau'_a \to \tau_r\end{array}}{\Gamma \vdash f\,a : \tau_r}\ \textsc{app}$$

$$\frac{\begin{array}{c}\Gamma \vdash_g e : \tau_1\\ \lfloor\tau_1\rfloor \equiv \tau_2\end{array}}{\Gamma \vdash (e :: \tau_2) : \tau_2}\ \textsc{ann} \qquad \frac{\begin{array}{c}\mathsf{fixate}_b\,[\ \Gamma \vdash_l e : \tau_x,\ noBoxes\,\tau_x\ ]\\ x :: \tau_x, \Gamma \vdash b : \tau\end{array}}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ b : \tau}\ \textsc{let}$$

Boxy instantiation rules:

$$\frac{\begin{array}{c}b_2\text{ fresh}\\ \mathsf{defer}_{\tilde{\tau}_2}\,[\ b_1 \equiv b_2,\ \mathsf{force}\,b_1,\ \boxed{\tilde{\tau}_1}_{b_1} \preceq' \tau,\ \tau \equiv \boxed{\tilde{\tau}_2}_{b_2}\ ]\end{array}}{\boxed{\tilde{\tau}_1}_{b_1} \preceq \boxed{\tilde{\tau}_2}_{b_2}}\ \textsc{boxy.inst}$$

$$\frac{\overline{\tau_2}\text{ fresh}}{\boxed{\forall\overline{\alpha}.\tau_1} \preceq' \boxed{[\overline{\alpha := \tau_2}]\tau_1}}\ \textsc{bi} \qquad \tilde{\tau} \preceq' \tilde{\tau}\ \textsc{br}$$

Protected unboxing rules:

$$\frac{\begin{array}{c}\tilde{\tau}_2\text{ fresh} \qquad \lfloor\tilde{\tau}_1\rfloor \equiv \lfloor\tilde{\tau}_2\rfloor\\ \mathsf{defer}_{b_2}\ \begin{array}{l}[\ \mathsf{let}\,[\![\_]\!] = b_2,\ b_1 \equiv b_2,\ \mathsf{force}\,b_2,\ \boxed{\tilde{\tau}_1}_{b_1} \sqsubseteq' \boxed{\tilde{\tau}_2}_{b_2}\ ],\\ [\ \mathsf{let}\,[\![v]\!] = b_1,\ \mathsf{commit}_v\,b_2,\ \boxed{\tilde{\tau}_1}_{b_1} \sqsubseteq' \boxed{\tilde{\tau}_2}_{b_2}\ ],\\ [\ \boxed{\tilde{\tau}_1}_{b_1} \sqsubseteq' \boxed{\tilde{\tau}_2}_{b_2}\ ]\end{array}\end{array}}{\boxed{\tilde{\tau}_1}_{b_1} \sqsubseteq \boxed{\tilde{\tau}_2}_{b_2}}$$

$$\frac{\begin{array}{c}\mathsf{let}\,[\![v]\!] = \tilde{\tau}_1\\ \mathsf{defer}_{\tilde{\tau}'_1}\,[\boxed{\tilde{\tau}_1}_b \sqsubseteq' \tau_2]\\ \tilde{\tau}_1 \equiv \tilde{\tau}'_1\end{array}}{\boxed{\tilde{\tau}_1}_b \sqsubseteq' \tau_2}\ \textsc{unbox.ty.defer} \qquad \frac{mono\,\tilde{\tau}}{\boxed{\tilde{\tau}} \sqsubseteq' \tilde{\tau}}\ \textsc{tbox}$$

$$\boxed{\forall\overline{\alpha}.\tau}_b \sqsubseteq' \boxed{\forall\overline{\alpha}.\tau}_b\ \textsc{refl} \qquad \frac{\begin{array}{ccc}\tau_1 \not\equiv \tau_2 & \tau_1 \sqsubseteq \tau'_2 & \tau'_2 \equiv \tau_2\\ \multicolumn{3}{c}{unboxed\,\overline{\alpha}\,\tau_1 \qquad unboxed\,\overline{\alpha}\,\tau_2}\end{array}}{\forall\overline{\alpha}.\tau_1 \sqsubseteq' \forall\overline{\alpha}.\tau_2}\ \textsc{poly}$$

$$\frac{\begin{array}{cc}\boxed{\tilde{\tau}_1} \sqsubseteq \tau_3 & \tau_3 \equiv \tau'_3\\ \boxed{\tilde{\tau}_2} \sqsubseteq \tau_4 & \tau_4 \equiv \tau'_4\end{array}}{\boxed{\tilde{\tau}_1} \to \boxed{\tilde{\tau}_2} \sqsubseteq' \tau'_3 \to \tau'_4}\ \textsc{conbox} \qquad \frac{\begin{array}{cc}\tilde{\tau}_1 \sqsubseteq \tau_3 & \tau_3 \equiv \tau'_3\\ \tilde{\tau}_2 \sqsubseteq \tau_4 & \tau_4 \equiv \tau'_4\end{array}}{\tilde{\tau}_1 \to \tilde{\tau}_2 \sqsubseteq' \tau'_3 \to \tau'_4}\ \textsc{cong}$$

Semantics:

```
instance Unifyable Box where ...
instance Deferrable Box where ... mkFixedValue = const BNo
```

Figure 7: Inferencer for FPH.

For protected unboxing, rule UNBOX controls how the rules are applied. It ensures that the input and output type are matched together, disregarding boxes such that this information flow is independent of the information flow about boxes. Then, applying the actual rules is deferred until the two box-annotations have been resolved. In case b_2 is still unknown, we default it. If b_2 is known, but b_1 is not, we apparently have freedom in the choice and choose b_2 for b_1. Finally, if we are in the situation that we know the annotations but not the type, we delay resolving the unboxing until we know the type by means of UNBOX.TY.DEFER.

Box operations:

```
noBoxes :: Ty → I ()
noBoxes (t,b) = do unif b BNo
                   traverse noBoxes t

unboxed :: GuessVar → Ty → I ()
unboxed a (t,b) | a ∈ ftv t    = do unif b BNo
                                    traverse (unboxed a)
                | otherwise = return ()

⌊τ⌋ = strip τ
strip :: Ty → Ty
strip (t,_) = do t′ ← dtraverse strip t; return (t′,BNo)

dtraverse f = dwrap (traverse f)
traverse f (TArr t1 t2) = do t3 ← f t1; t4 ← f t2
                             return (TArr t3 t4)
traverse f (TAll vs t1)  = do t2 ← f t1; return (TAll vs t2)
traverse f t             = return t

mono t = unif t (dcheck t)
dcheck = dwrap check
check (TArr t1 t2) = check′ t1 ≫ check′ t2
check (TAll _ _)    = fail "not a mono type"
check _             = return ()
check′ (t,_)        = dcheck t
```

Figure 8: FPH monadic Haskell premises

The monadic Haskell expressions used in the premises of the inferencer rules are given in Figure 8. *noBoxes* forces the absence of boxes everywhere in the type, and *unBoxed* only on the spine to each occurrence of type variable a. The most involving, however, is box-stripping. It produces a type with all boxes removed, without affecting the original box annotations. The difficulty is that the type may not be fully known yet. Fortunately, we can use **defer**, *equal* and **commit** in monadic expressions too. In fact, we can write higher-order functions to factor out some patterns. For example, *dwrap* (Figure 9) factors out all the non-determinism of a recursive function where the input is equal to the output modulo some guesses.

Other type systems. The syntax of the defer-statement is actually a bit more general than we presented in these examples. In the examples, the deferred statements did not have outputs, only inputs. We allow the deferred statements to have outputs. For example, another type system for first class polymorphism, HML (Leijen 2009), requires deferred statements that produce a prefix Q (denoted with $\mathsf{defer}^{Q}_{\tau}[\ldots]$).

Although our examples were about inferencers for type systems dealing with polymorphism, we stress that these were chosen in

```
dwrap :: ∀ α . Deferrable α ⇒ (α → I α) → α → I α
dwrap f t = do (t_out, ()) ← defer (λt_in _ →
                    do unifOne t t_in
                       t' ← f t
                       unif t_out t'
                 return t_out
```

Figure 9: Example of evaluation control abstraction.

order to pave the way to a complex example, and that we are not limited to such type systems.

3. Related Work

We present an extension of our previous work on the Ruler system (Dijkstra and Swierstra 2006b). In this system, type rules are required to be written as deterministic functions, and both a typesetted LaTeX document and an efficient Attribute-Grammar based inferencer are derived from them. One of the goals of this system is to close the gap between formal description and implementation. However, non-deterministic aspects cannot be directly described in this system, and are omitted (to be solved externally). Thus, this leaves the gap wide open, and we close it with this work.

Functional Logic Programming. The essential problem with non-deterministic aspects is that the function to resolve it needs to make decisions, but is unable to do so based on what is known about the inputs at that point. Therefore, the idea is to delay the execution until we know more about the output, and let expected output play a role in the decision process. Therefore, at specific places, we turn functions into relations, which has a close resemblance to Functional Logic Programming (Hanus 1994).

With FLP, non-deterministic functions can be written as normal functions. The possible alternatives that these functions can take depends not only on the inputs, but also on the scrutinizing of the result. With evaluation strategies such as narrowing (Antoy 1997), the search space is explored in a demand-driven way. Knowledge of context is pushed inwards, reducing possible alternatives, and causing evaluation to occur that refines the context even more.

With the defer and commit, we offer a poor man's mechanism to FLP. The delaying of choice and the scrutinizing of the choice is explicit. A commit is required to reduce the choice to at most one possibility. Yet, we have good reasons not to support the full generality of FLP. We want to integrate the inferencers specified in our language into mainstream compilers. Our approach makes only little demands on infrastructure. If it can cope with Algorithm W (Milner 1978), then the proposed mechanisms of this paper fit. Furthermore, we want to be able to use constraint solvers in some foreign language, or arbitrary Haskell libraries in our inference rules. This gives rise to problems with narrowing.

A difference with respect to FLP is our fixation and inspection of guesses. Consider an expression like *const x y*. For such an expression, the type of *y* is irrelevant and will not be scrutinized. However, we have several reasons to do so. We produce derivations, so we need a derivation of *y*. Decisions about the inference of *y* need to be made, even when the context does not make strong demands about which one. To make such a decision, we need to inspect which values are still guesses. As a consequence, more type information may be discovered, or even type errors that would otherwise go unnoticed, or which only arise much later (say, when generating code). Also, to deal with rules such as generalization properly, we need to know the difference between an unresolved guess and some fixed but unconstrained information. Furthermore, unresolved guesses retain memory which causes severe memory problems when growing unchecked in mainstream compilers, and when integrating with foreign code, we need invariants about which parts of values are resolved. Finally, examples such as in Figure 5 really require the guessing to be explicit and first class.

Logic Programming In a similar way as with FLP, our approach has strong ties with Logic Programming. One particular difference is that we disallow backtracking and of all possible rules demands that only one rule can succeed. Again, the reasons are related to efficiency and integration. However, there is an even more important reason: to be able to produce sensible type errors, and to prevent infinite searches in the presence of type errors.

Inferencer Frameworks. There are inferencer frameworks such as *HM* (*X*) (Sulzmann and Stuckey 2008), which is based on a fixed set of type rules parametrized over some relation *X*, for which an inference algorithm needs to be given to obtain an inferencer for the full language. Such a framework is in fact orthogonal to Ruler, and Ruler can be used to construct the algorithm for *X*. In fact, the precise relation of Ruler to other constraint-based inference frameworks is that a Ruler specification can be seen as both a specification of constraints, annotated with the algorithm to solve them.

Type rule tools. OTT (Sewell et al. 2007) produces code for proof assistants. SASyLF (Aldrich et al. 2008) is such a proof assistant tailored to proving properties about type systems, as is Twelf (Harper and Licata 2007).

Tinkertype (Levin and Pierce 2003) is a system that can also produce inferencers from type rules. However, the inference algorithms are not derived from the type rules. Instead, it depends on a repository with code for each relation to compose the inferencer.

4. Type Inferencer Syntax

4.1 Core Syntax

The syntax of the type inferencer language (named RulerCore) is given in Figure 10. A type inferencer is a triple $(\Sigma^*, r^*, H_\lambda)$ of schemes Σ^*, rules r^*, and a some Haskell support code in the form of data-type declarations, some instances for them, and utility functions. A scheme Σ represents a function named s with inputs declared by environment Γ_{in}, and outputs by environment Γ_{out}. A scheme can be parametrized over some types α, which provides for a limited form of polymorphism for the inferencer rules. The inferencer rules in r^* with scheme name s define the function s. Each rule r consists of a (possibly empty) sequence of premises (c), and a conclusion (c_s).

It is important to realize that we are not defining type rules here. Schemes are not arbitrary relations, but are functions. The premises are statements, not predicates. Also, the order of the premises matter. Values for all inputs need to be available before a scheme can be instantiated. The rules and statements have a certain operational behavior. A rule evaluates successfully if and only evaluation of all its premises succeeds, and for each statement we give a brief description below. We make this more precise later.

The conclusion r_s of an inferencer rule defines to what names the actual parameters of the scheme s are bound in the context of the rule, and which local results are the outputs of the scheme. Expressed with bindings Δ_{in} and Δ_{out} respectively, where bindings are a mapping from formal name to actual name.

Statements for premises come in different forms. There are a couple of statements that allow algorithms to be described:

- Evaluation of statement c_s instantiates a scheme, which means applying the scheme function to the inputs, and obtaining the outputs if this application is successful. For the input values, the values bound to the actual names are taken for the formal

$$
\begin{array}{lcll}
\Sigma^* & = & \Sigma_1,\ldots,\Sigma_k & \text{(schemes)} \\
\Sigma & = & \forall\overline{\alpha}.\ \Gamma_{\mathrm{in}} \vdash_s \Gamma_{\mathrm{out}} & \text{(scheme)} \\
r^* & = & r_{s_1},\ldots,r_{s_k} & \text{(rules)} \\
r_s & = & c^*\ ;\ c_s & \text{(rule)} \\
c^* & = & c_1,\ldots,c_k & \text{(statements)} \\
c & = & c_s & \text{(instantiate)} \\
 & | & n_1 \equiv n_2 & \text{(unification)} \\
 & | & \mathsf{exec}\ e :: n^* \rightarrow m^* & \text{(execution)} \\
 & | & \mathsf{fixpoint}^{n^*}_{\Delta}\ c_s & \text{(fixpoint)} \\
 & | & \mathsf{defer}^{m^*}_{n}\ c^*_1,\ldots,c^*_k & \text{(defer)} \\
 & | & \mathsf{commit}_{v_\tau}\ n & \text{(commit)} \\
 & | & \mathsf{fixate}_{\tau}\ c^* & \text{(fixate)} \\
c_s & = & \Delta_{\mathrm{in}} \vdash_s \Delta_{\mathrm{out}} & \text{(scheme instance)} \\
\Gamma & = & n_1 ::_\rho \tau_1,\ldots,n_k ::_\rho \tau_k & \text{(environment)} \\
\Delta & = & n_1 \mapsto m_1,\ldots,n_k \mapsto m_k & \text{(bindings)} \\
\rho & = & \mathrm{d} & \text{(deferrable)} \\
 & | & \mathrm{u} & \text{(unifyable)} \\
 & | & \emptyset & \text{(none)}
\end{array}
$$

With scheme names s, identifiers n, m, and v, collection of identifiers n^* and m^*, Haskell types τ, and expressions e.

Figure 10: Syntax of RulerCore.

names (specified by bindings Δ_{in}). Similarly, the values bound to the formal name of the scheme are made available under the actual name (specified by bindings Δ_{out}).

- The unification statement attempts to unify the values bound to the names n_1 and n_2. A unification algorithm based on structural equality needs to be available for the values to which these two types are bound. Successful unification results in a substitution that makes the two values equal. This substitution is implicit and can be assumed to be applied everywhere.
- The execution statement allows monadic Haskell code to be executed. This code is a function taking the values bound to names n^* as parameter and returns a monadic value containing values for outputs m^*. We consider this expression language in more detail later. The purpose of these execution statements is to perform the actual computations needed to produce the values for the inputs of a scheme instantiation, and to inspects the outputs of it by means of pattern matching.
- The fixpoint statement repeatedly instantiates s, as long as the values bound to identifiers n^* change. The bindings Δ specify how the outputs are mapped back to the inputs after each iteration.

And the statements that allow us to algorithmically deal with non-deterministic aspects:

- The defer statement represents one of the sequence of statements c^*_i, except that evaluation of it takes place at a later time. In the mean time, a guess (encoded as a fresh variable) for the output n is produced, and bottom-values for outputs n^*. For each guessable data type, we require that we can encode a variable as a value of this data type (denoted as $[\![v]\!]$). These guesses can be passed around as normal values.
- A commit statement refines a guess bound to v with the value bound to identifier n, and runs deferred statements, which may lead to other refinements of guesses.
- The fixate statement executes statements c^*. All guesses of type τ that were not committed during this execution are resolved by executing the deferred statements. Any remaining guesses are marked as fixed. These now represent opaque values that cannot be refined anymore.

Expressions in an exec-statement are Haskell functions in monad I that get the inputs passed as arguments and are obliged to return a product with the results. Hence the type of an expression e:

$$e :: \tau_{n_1} \rightarrow \ldots \tau_{n_k} \rightarrow I(\tau_{m_1},\ldots,\tau_{m_l})$$

The I monad contains a hidden state, and support failure. In particular, the following operations are available:

$$
\begin{array}{ll}
\textbf{commit} & :: \mathit{Deferrable}\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow I\ () \\
\textbf{defer} & :: (\mathit{Deferrable}\ \alpha, \mathit{Prod}\ \beta) \Rightarrow (\alpha \rightarrow I\ \beta) \rightarrow I\ (\alpha,\beta) \\
\textbf{unif} & :: \mathit{Unifyable}\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow I\ () \\
\textbf{unifOne} & :: \mathit{Unifyable}\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow I\ () \\
\textbf{update} & :: \mathit{Container}\ \alpha \Rightarrow \alpha \rightarrow I\ \alpha \\
\textbf{fail} & :: \mathit{String} \rightarrow I\ ()
\end{array}
$$

We create a deferrable computation with *Defer*. It takes a monadic function that is only executed when a commit is performed on alpha. This monadic function produces the values for the product β. Until this actually happened, the contents of the product may not be touched. The **unif** operation enforces structural equality between values α. In case of **unifOne**, only structural equality on the heads of the values. Finally, **update** brings all guesses in α up to date, and **fail** causes the inference to fail with a type error.

4.2 Syntactic Sugar

The previous section gave the core syntax for the type inferencer. For practical and didactic purposes, the examples in the previous section where given in a somewhat more convenient syntax that can be translated to the core syntax.

First of all, we assume a series of notational conventions involving sequences (lists), environments (maps), or sets:

- A sequence of, for example, statements is denoted by $c^* = \overline{c_i} = c_1 \ldots c_k$, where i $(1 \leqslant i \leqslant k)$ is some index, and k is left implicit as it is clear from the context, i.e. when i ranges also over some list.
- A list of identifiers is denoted by $n^* = \overline{n_i} = n_1,\ldots,n_k$.
- The empty map, empty list, or empty set is written as $\emptyset$.

The syntactic sugar is given in Figure 11. A scheme declaration Σ for a scheme named s, now consists of a sequence of either an input or output declaration, or a keyword. For example, the scheme declaration for a scheme tp_expr:

$$\mathsf{tp\_expr} : (\Gamma \lhd \mathsf{Env}) \vdash (e \lhd \mathsf{Expr}) : (\tau \rhd_{\mathrm{d}} \mathsf{Type})$$

defines two inputs Γ and e with types Env and Expr respectively, and an output named τ with type Type (and the deferrable-property). To instantiate this scheme, the statement has the form: tp_expr : … ⊢ … : …, where at the places of the dots there is a Haskell pattern for an output, and a pure Haskell expression for an input (the reverse for the conclusion statement). The identifiers of a pattern can be referenced by expressions of subsequent statements of a rule. In the examples of Section 2, we left out the scheme names, because it is clear from the context.

The essential differences between Figure 10 and Figure 11 are:

- The syntactic sugar allows for Haskell expressions and patterns at places where originally only identifiers were expected. This syntactic sugar is translated to execution-statements with the appropriate inputs and outputs. Pattern match failures are translated to fail-expressions. We will also assume that pure Haskell expressions are automatically lifted into a monadic expression when needed. Also, an identifier occurring at multiple input-locations is replaced with unique identifiers with the necessary *equiv*-statements added to the front of the statement sequence.

$$
\begin{array}{lcll}
\Sigma^* & = & \Sigma_1,\ldots,\Sigma_k & \text{(schemes)} \\
\Sigma & = & \forall\overline{\alpha}.\ s : d_1,\ldots,d_k & \text{(scheme signature)} \\
d & = & \textbf{kw} & \text{(keyword decl)} \\
 & \mid & n \lhd_\rho \tau & \text{(input decl)} \\
 & \mid & n \rhd_\rho \tau & \text{(output decl)} \\
r^* & = & r_{s_1},\ldots,r_{s_k} & \text{(rules)} \\
r_s & = & \dfrac{c_1 \ldots c_k}{c_s} & \text{(rule)} \\
c & = & c_s & \text{(instantiate)} \\
 & \mid & m_{H_\lambda,1} \equiv m_{H_\lambda,2} & \text{(unification)} \\
 & \mid & \mathsf{let}\ p_{H_\lambda} = e_{H_\lambda} & \text{(pure)} \\
 & \mid & p_{H_\lambda} \leftarrow m_{H_\lambda} & \text{(monadic bind)} \\
 & \mid & m_{H_\lambda} & \text{(monadic exec)} \\
 & \mid & \mathsf{fixpoint}^{n^*}_{\Delta}\ c_s & \text{(fixpoint)} \\
 & \mid & \mathsf{defer}^{m^*}_{n}\ c^*_1,\ldots,c^*_k & \text{(defer)} \\
 & \mid & \mathsf{commit}_{v_\tau}\ e_{H_\lambda} & \text{(commit)} \\
 & \mid & \mathsf{fixate}_{\tau}\ c^* & \text{(fixate)} \\
 & \mid & \mathsf{force}\ m_{H_\lambda} & \text{(force)} \\
c_s & = & s : i_1,\ldots,i_k & \text{(scheme instance)} \\
i & = & \textbf{kw} & \text{(keyword)} \\
 & \mid & p_{H_\lambda} & \text{(value deconstruction)} \\
 & \mid & e_{H_\lambda} & \text{(value construction)}
\end{array}
$$

With keywords **kw**, Haskell patterns p_{H_λ}, Haskell expressions e_{H_λ}, and monadic Haskell expressions m_{H_λ}.

Figure 11: Syntax of RulerBase.

- No special rules about the structure of monadic functions. These are normal Haskell monadic expressions in some monad *I*, and the commit, unify, and update-operations are functions that act in this monad.
- For the core language, only one deferred statement-sequence is allowed to succeed when triggered to evaluate. Here, we assume that more than one is allowed to succeed, but the first one from the left is taken.
- force is syntactic sugar for executing f m_{H_λ}. If the result is a guess, then a commit is done with a fresh value as parameter. If the result is still this fresh value, a commit is done with a fixed value. Otherwise, the result is ignored.

Identifiers occurring in expressions are brought up-to-date with respect to guesses just before evaluating the expressions.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{pattern}\ \mathit{Map\ String\ Ty}\ \textbf{where}} \\
\quad x :: \tau, \Gamma & \textbf{input}\quad \mathit{insert}\ x\ \tau\ \Gamma \\
\multicolumn{2}{l}{\textbf{pattern}\ \mathit{Expr}\ \textbf{where}} \\
\quad x & \textbf{output}\ \mathit{EVar}\ x \\
\quad f\ a & \textbf{output}\ \mathit{EApp}\ f\ a \\
\quad \lambda x\ .\ e & \textbf{output}\ \mathit{ELam}\ x\ e \\
\multicolumn{2}{l}{\textbf{pattern}\ \mathit{Ty}\ \textbf{where}} \\
\multicolumn{2}{l}{\quad [\,\alpha := \tau_1\,]\ \tau_2\ \textbf{output}\ \mathit{singleSubst}\ \alpha\ \tau_1 \Mapsto ty} \\
\multicolumn{2}{l}{\textbf{pattern}\ \mathit{Ty}\ \textbf{where}} \\
\quad \boxed{t}_b & \textbf{input}\quad (t, b) \\
\quad \boxed{t} & \textbf{input}\quad (t, \mathit{BYes}) \\
\quad t & \textbf{input}\quad (t, \mathit{BNo})
\end{array}
$$

Figure 12: Examples of pattern declarations.

Finally, we assume a number of pattern declarations, of which an example is given in Figure 12. These define special syntax for Haskell expressions (indicated with **input**) and patterns (indicated with **output**) for certain specific types, and the translation to Haskell. To disambiguate, the types of identifiers play a role. The pattern *x* can stand for the identifier x (say, if the type is *String*), or for the expression *EVar x* if type is *Expr*.

5. Operational Semantics

In this section we give a big-step operational semantics of the inferencer language introduced in Section 4. We first discuss some notation, then explain the evaluation rules. Evaluation of the inferencer rules involves data manipulation. Some demands are made about the data in question. In particular, we require structural equality to be defined for data types. We finish this section with a discussion of these demands.

5.1 Notation

For the operational semantics, heaps $\mathcal{H}$, substitutions θ and derivations π are used for bookkeeping. Their syntax is given in Figure 13. Heaps are a mapping of locations (in our case, plain identifiers) to Haskell values. Substitutions keep track of information about guesses (identified by a variable v). Either, a guess is resolved and represents some concrete value w of type τ, or will be resolved through a commit on another variable and is for the moment mapped to $\bot$, or represents a closure of the deferred statements. In the latter case, we store in a heap $\mathcal{H}$ entries for each identifier referenced by the deferred statements, store a scope identifier ζ representing the deepest scope in which the deferred statement is introduced (encoded as a number equal to the nesting-depth), and a rule identifier *r*. Each defer-statement introduces a unique rule identifier, which is a placeholder for a derivation. Derivations represents a partial derivation in an abstract way. The conclusions of each rule make up the nodes of the derivation-tree (with the values of their instantiation in heap $\mathcal{H}$). Statements that cannot be represented by this are represented with an opaque-leaf $\diamond$. These derivations may be partial and refer to sub-derivations named ι with $!\iota$.

$$
\begin{array}{lcll}
\mathcal{H} & = & n_1 := w_{1,\tau_1},\ldots,n_k := w_{k,\tau_k} & \text{(heap)} \\
\theta & = & v_1 \mapsto q_1,\ldots,v_k \mapsto q_k & \text{(substitution)} \\
q & = & w_\tau & \text{(subst value)} \\
 & \mid & \bot & \text{(subst bot)} \\
 & \mid & \textbf{deferred}^{\zeta,\iota}_{\mathcal{H}}\ c^*_1,\ldots,c^*_k & \text{(deferred)} \\
\pi & = & \dfrac{\pi}{\mathcal{H} \vdash c_s} & \text{(node)} \\
 & \mid & !\iota & \text{(reference)} \\
 & \mid & \diamond & \text{(opaque)} \\
 & \mid & \pi_1\ \pi_2 & \text{(and-cons)} \\
\Pi & = & \iota_1 \mapsto \pi_1,\ldots,\iota_k \mapsto \pi_k & \text{(named derivations)}
\end{array}
$$

With Haskell value w, rule identifier ι, and scope identifier ζ.

Figure 13: Syntax of heaps, substitutions, and derivations.

Substitutions satisfy the usual substitution properties. Juxtaposition of substitutions $\theta_1\theta_2$ represents the left-biased union of the two substitutions, with θ_1 applied to all entries q of θ_2. Applying a substitution θ_1 to a value w, denoted with θw, replaces each guess $[\![v]\!]$ with either w if $v \mapsto w_\tau \in \theta$ or itself otherwise. Application to a $\bot$-entry is the identity, and to a **deferred**-entry means applying it to the heap. Substitution application is lifted to environments, heaps, and derivations as well.

We also use some notation concerning heaps and bindings. The lookup of a value for an identifier is written as $\mathcal{H}(n)$. With bindings we can take and rename entries in a heap: $\mathcal{H}(\Delta)$ is a heap which for each binding $n \mapsto n'$ has the value w_τ for n taken from $\mathcal{H}$ as n', i.e. $\mathcal{H}(\Delta)(n) = \mathcal{H}(n')$. We also use the reverse: $\Delta(\mathcal{H})(n') = \mathcal{H}(n)$. Juxtaposition of heaps stands for the left-biased union of the two.

5.2 Evaluation Rules

Overview. We can now give the evaluation rules of our big-step operational semantics. Figure 14 lists the structure of the evaluation rules. Given a statement c, the reduction relation gives a transition from a substitution θ_0 and heap $\mathcal{H}_0$ with values for the inputs of c, to an heap $\mathcal{H}_1$ containing values for the outputs of c and an updated substitution θ_1. The transition is labeled with a derivation π which can be considered a trace of the steps that were taken in order to make the transition. Similarly, Π contains (at least) a binding for each reference in derivation π and any reference of any derivation in Π itself. There are some variations of this reduction relation on the level of statements and rules. An important invariant is that the resulting heap is up to date with respect to the resulting substitution.

The semantics of substitution refinement by defaulting the guesses of a certain scope, starts with an initial substitution θ_0, and the current scope identifier ζ, and ends in a state θ_1. The purpose of this relation is to force the evaluation of deferred statements created in ζ of type τ, such that none of these remain in θ_1.

For the evaluation of (monadic) Haskell expressions, we construct an expression e and evaluate it in an execution environment H_λ, containing data type definitions, Haskell support code, augmented with bindings for inputs to the expression, including the substitution.

$\theta_0 ; \mathcal{H}_0 ; \zeta ; c \rightarrow^{\Pi}_{\pi} \mathcal{H}_1 ; \theta_1$	(statement reduction)
$\theta_0 ; \mathcal{H}_0 ; \zeta ; c_1, \ldots, c_k \rightarrow^{\Pi}_{\pi} \mathcal{H}_1 ; \theta_1$	(statements reduction)
$\theta_0 ; \mathcal{H}_0 ; \zeta ; r_s \rightarrow^{\Pi}_{\pi} \mathcal{H}_1 ; \theta_1$	(rule reduction)
$\tau ; \theta_0 ; \zeta \rightarrow^{\Pi}_{*} \theta_1$	(scope defaulting)
$H_\lambda \vdash e_{H_\lambda} \rightarrow w$	(Haskell evaluation)

Figure 14: Structure of the evaluation rules.

Given a type inferencer, a triple $(\Sigma^*, r^*, H_\lambda)$, and an instantiation of scheme Σ by means of statement c_s with a heap $\mathcal{H}_0$, evaluation of this statement with the inferencer rules is the transition

$$\emptyset ; \mathcal{H}_0 ; 0 ; \mathsf{fixate}_* \, c_s \rightarrow^{\Pi}_{\pi} \mathcal{H}_1 ; \theta_1$$

according to the smallest reduction relations satisfying the evaluation rules of Figure 15, Figure 16, Figure 17, and Figure 18. We explain these rules in more detail. Furthermore, we assume that the components of the inferencer-triple are available in the rules as a constant.

Conventional statements. In Figure 16 are the rule for what we call the conventional statements. These are the statements in which type checking algorithms can be expressed. Type inference is not possible with these rules yet since this requires guessing.

The Scheme-rule represents instantiation of a scheme named s. An inferencer rule r_s is chosen and evaluated. The bindings dictate which values to take from the heap to use as inputs to the rule. Similarly, the bindings dictate under which name the outputs after evaluation are to be stored. These inferencer rules must be syntax directed. There should be only one r_s that can be applied.

For the unify-rule, the values bound to n_1 and n_2 are checked for equality with the *unify* function defined on the type of these values. When the types involve guesses, this may lead to discovery of more type information about guesses and an updated substitution.

In the exec-rule, the monadic code is executed with the values of $n_1, \ldots, n_k$ as parameter. The monadic code may update the substitution, or cause the statement to fail. If the execution succeeds, the returned product of the monadic code contains the values for in the output-heap.

$$\frac{r_s \in r^* \qquad \mathcal{H}_{\mathrm{in}'} = \mathcal{H}_{\mathrm{in}}(\Delta_{\mathrm{in}}) \qquad \mathcal{H}_{\mathrm{out}} = \Delta_{\mathrm{out}}(\mathcal{H}_{\mathrm{out}'}) \qquad \theta ; \mathcal{H}_{\mathrm{in}'} ; \zeta ; r_s \rightarrow^{\Pi}_{\pi} \mathcal{H}_{\mathrm{out}'} ; \theta'}{\theta_1 ; \mathcal{H}_{\mathrm{in}} ; \zeta ; \Delta_{\mathrm{in}} \vdash_s \Delta_{\mathrm{out}} \rightarrow^{\Pi}_{\pi} \mathcal{H}_{\mathrm{out}} ; \theta'} \ \textsc{Scheme}$$

$$\frac{\theta' = \mathit{fst}\ (\mathit{run}\ (\mathbf{unif}\ \mathcal{H}(n_1)\ \mathcal{H}(n_2))\ \theta\ \zeta)}{\theta ; \mathcal{H} ; \zeta ; n_1 \equiv n_2 \rightarrow^{\emptyset}_{\diamond} \emptyset ; \theta'} \ \textsc{Unify}$$

$$\frac{H_\lambda \vdash \mathit{run}\ (e\ \mathcal{H}(n_1) \ldots \mathcal{H}(n_k))\ \theta\ \zeta \rightarrow (\theta', (w_1, \ldots, w_l)) \qquad \mathcal{H}' = m_1 \mapsto w_1, \ldots m_k \mapsto w_k}{\theta ; \mathcal{H} ; \zeta ; \mathsf{exec}e :: n_1, \ldots, n_k \rightarrow m_1, \ldots, m_l \rightarrow^{\emptyset}_{\diamond} \mathcal{H}' ; \theta'} \ \textsc{Exec}$$

$$\frac{r_s \in r^* \qquad \theta_1 ; \mathcal{H}_1 ; \zeta ; r_s \rightarrow^{\Pi}_{\pi} \mathcal{H}_2 ; \theta_2 \qquad \mathcal{H}_1(n^*) \neq \mathcal{H}_2(n^*) \qquad \theta_2 ; \mathcal{H}_2(\Delta)\,\mathcal{H}_2 ; \zeta ; \mathsf{fixpoint}^{n^*}_{\Delta}\ c_s \rightarrow^{\Pi}_{\pi'} \mathcal{H}_3 ; \theta_3}{\theta_1 ; \mathcal{H}_1 ; \zeta ; \mathsf{fixpoint}^{n^*}_{\Delta}\ c_s \rightarrow^{\Pi}_{\pi\,\pi'} \mathcal{H}_3 ; \theta_3} \ \textsc{Fixstep}$$

$$\theta ; \mathcal{H} ; \zeta ; \mathsf{fixpoint}^{n^*}_{\Delta}\ c_s \rightarrow^{\Pi}_{\pi} \mathcal{H}(\Delta)\,\mathcal{H} ; \theta \ \textsc{Fixskip}$$

Figure 15: Evaluation rules for conventional statements.

Finally, with the fixpoint-rule a scheme-statement can be repeatedly executed as long as it causes one of the values of n^* to change. For each repetition, the bindings Δ dictate which outputs are the inputs of the next iteration. In this case there may be more than one applicable rule r_s. However, to make a step, evaluation of the inferencer rule must cause a change of value n.

$$\frac{\theta_{\mathrm{in}} ; \Delta_{\mathrm{in}}(\mathcal{H}_{\mathrm{in}}) ; \zeta ; c_1, \ldots, c_k \rightarrow^{\Pi}_{\pi_1, \ldots, \pi_k} \mathcal{H}' ; \theta_{\mathrm{out}} \qquad \pi = \dfrac{\pi_1 \ldots \pi_k}{\mathcal{H}' \vdash (\Delta_{\mathrm{in}} \vdash_s \Delta_{\mathrm{out}})} \qquad \mathcal{H}_{\mathrm{out}} = \mathcal{H}'(\Delta_{\mathrm{out}})}{\theta_{\mathrm{in}} ; \mathcal{H}_{\mathrm{in}} ; \zeta ; c_1, \ldots, c_k ; \Delta_{\mathrm{in}} \vdash_s \Delta_{\mathrm{out}} \rightarrow^{\Pi}_{\pi} \mathcal{H}_{\mathrm{out}} ; \theta_{\mathrm{out}}} \ \textsc{Rule}$$

$$\frac{\theta_i ; (\theta_i \mathcal{H}_i \ldots \mathcal{H}_1) ; \zeta ; c_i \rightarrow^{\Pi}_{\pi_i} \mathcal{H}_{i+1} ; \theta_{i+1},\ 1 \leqslant i \leqslant k}{\theta_1 ; \mathcal{H}_1 ; \zeta ; c_1, \ldots, c_k \rightarrow^{\Pi}_{\pi_1 \ldots \pi_k} \theta_{k+1} \mathcal{H}_{k+1} \ldots \mathcal{H}_1 ; \theta_{k+1}} \ \textsc{Statements}$$

Figure 16: The apply rules.

In Figure 16 are evaluation rules for a chosen inferencer rule r_s. The bindings of the conclusion specifies under what names the inputs to the rule need to be presented to the statements. Likewise, the bindings also specify under what names the values of the outputs of the rule are available. Successful evaluation of a rule means that a derivation π has been produced, of which the current rule forms the root, and the derivations of the premises are its immediate children.

Evaluation of a sequence of statements causes the heap to accumulate the outputs of the statements already executed so far. The outputs of predecessors of a statement in this sequence are also available as input to the statement. Since each evaluation of a statement potentially causes more information to be known about guesses in such outputs, the most recent substitution is applied to these predecessor-outputs first.

Non-deterministic statements To deal with guesses, there are the statements that deal with non-determinism, which we will call the non-deterministic statements. Their semantics is made precise in Figure 17.

For a Defer-statement, guesses are produced as outputs for $n, m_1, \ldots, m_l$. A closure for the statements $c_1, \ldots, c_k$ is stored as

$$\frac{\begin{array}{c} v, v_1, \ldots, v_l, \iota \text{ fresh} \\ \mathcal{H} = n \mapsto \llbracket v \rrbracket, m_1 \mapsto \llbracket v_1 \rrbracket, \ldots, m_l \mapsto \llbracket v_l \rrbracket \\ \mathcal{H}_{\text{out}} = \mathcal{H}\mathcal{H}_{\text{in}} \qquad \theta_1 = v_1 \mapsto \bot, \ldots, v_l \mapsto \bot \\ \theta_2 = v \mapsto \textbf{deferred}^{\zeta,\iota}_{\mathcal{H}_{\text{out}}}\ c^*_1, \ldots, c^*_k \end{array}}{\theta\, ;\, \mathcal{H}_{\text{in}}\, ;\, \zeta\, ;\, \mathsf{defer}^{m_1,\ldots,m_l}_{n}\ c^*_1, \ldots, c^*_k \ \xrightarrow{\emptyset}_{!\iota}\ \mathcal{H}_{\text{out}}\, ;\, \theta_1\theta_2\theta} \ \textsc{Defer}$$

$$\frac{\begin{array}{c} v \mapsto \textbf{deferred}^{\zeta,\iota}_{\mathcal{H}}\ c^{**} \in \theta \\ c_1, \ldots, c_k \in c^{**} \qquad \theta_1 = v \mapsto \mathcal{H}_{\text{in}}(n), \theta_{\text{in}} \\ \theta_1\, ;\, \theta_1\mathcal{H}\, ;\, \zeta\, ;\, c_1, \ldots, c_k \ \xrightarrow{\Pi}_{\pi}\ \mathcal{H}'\, ;\, \theta_2 \\ \theta_{\text{out}} = \{ v \mapsto w \mid n \mapsto \llbracket v \rrbracket \in \mathcal{H}, n \mapsto w \in \mathcal{H}' \}, \theta_2 \\ \iota \mapsto \pi \in \Pi \end{array}}{\theta_{\text{in}}\, ;\, \mathcal{H}_{\text{in}}\, ;\, \zeta\, ;\, \mathsf{commit}_{v_\tau}\ n \ \xrightarrow{\Pi}_{\diamond}\ \emptyset\, ;\, \theta_{\text{out}}} \ \textsc{CommitVar}$$

$$\frac{\begin{array}{c} v \mapsto w \in \theta \qquad n' \text{ fresh} \qquad \mathcal{H} = n' \mapsto w, n \mapsto \mathcal{H}_{\text{in}}(n) \\ \theta_{\text{in}}\, ;\, \mathcal{H}\, ;\, \zeta\, ;\, n' \equiv n \ \xrightarrow{\Pi}_{\diamond}\ \mathcal{H}'\, ;\, \theta_{\text{out}} \end{array}}{\theta_{\text{in}}\, ;\, \mathcal{H}_{\text{in}}\, ;\, \zeta\, ;\, \mathsf{commit}_{v_\tau}\ n \ \xrightarrow{\Pi}_{\diamond}\ \emptyset\, ;\, \theta_{\text{out}}} \ \textsc{CommitVal}$$

$$\frac{\begin{array}{c} \theta_{\text{in}}\, ;\, \mathcal{H}_{\text{in}}\, ;\, \zeta\, ;\, c^* \ \xrightarrow{\Pi}_{\pi}\ \mathcal{H}_{\text{out}}\, ;\, \theta_1 \\ \tau\, ;\, \theta_1\, ;\, \zeta \rightarrow^{\Pi}_{*} \theta_{\text{out}} \qquad \text{deferred}(\zeta, \theta_{\text{out}}) = \emptyset \end{array}}{\theta_{\text{in}}\, ;\, \mathcal{H}_{\text{in}}\, ;\, \zeta - 1\, ;\, \mathsf{fixate}_{\tau}\ c^* \ \xrightarrow{\Pi}_{\pi}\ \theta_{\text{out}}\mathcal{H}_{\text{out}}\, ;\, \theta_{\text{out}}} \ \textsc{Fixate}$$

Figure 17: Evaluation rules for non-determinism annotations.

substitution for the guess of n as closure. A commit on this guess leads to the execution of these statements, and to the production of values for the guesses $m_1, \ldots, m_l$. Committing on these latter guesses is not possible, since the substitution for these guesses is mapped to $\bot$. This closure is introduced in scope ζ and contains a unique identifier ι which is the name of the derivation that is produced later. A reference to this derivation is returned as the derivation of the Defer-statement.

Evaluation of the Commit-statement leads to the evaluation of the deferred statements. The heap stored in the closure is updated to the current substitution, and the substitution reflects the newly found information about the guess. Then, one of the sequences of statements is chosen and evaluated. The evaluation has caused outputs to be produced for values that were before represented as a guess to a $\bot$-substitution. Subsequently, the substitution is updated such that these substitutions do not map to $\bot$ anymore, but to their newly produced value.

Finally, there is the Fixate-statement. Its statement is evaluated in a subscope ζ. After evaluation, the remaining guesses are defaulted such that no deferred statement is left for scope ζ. The rules for defaulting are specified in Figure 18.

A deferred guess is committed to with either a *flexible* guess as value, or with a *fixed* unknown value. In both cases, of the deferred statement-sequences must be able to evaluate with this newly found information. In the first case, such deferred statement observes a guess as value for its input, and is allowed to refine it. In the later case, the value may not be touched.

Since committing to a flexible guess results in the introduction of a guess in the scope, in order to end up with no guesses in the scope in the end, each commit to a flexible guess must lead to refinements of guesses. All guesses that are essentially unconstrained will then end up with fixed unknowns.

Haskell semantics. The semantics of the monadic functions are defined in terms of Haskell. The type of the monad is given in Figure 19. It is a conventional combination between the error monad and the state monad. The *run* function is the interface between the semantic world and the monad world. If the monadic evaluation is successful, the premise with *run* succeeds and there has been a state transition into the monad and back. If the evaluation results in an *Err*, the premise with *run* does not hold.

$$\frac{\begin{array}{c} n, v' \text{ fresh} \qquad v_\tau \in \text{dom}(\theta_{\text{in}}) \\ \mathcal{H} = n \mapsto \llbracket v' \rrbracket \qquad \theta = v' \mapsto \textbf{deferred}^{\zeta,\iota}_{\emptyset}\ \{\emptyset\} \\ \theta\theta_{\text{in}}\, ;\, \mathcal{H}\, ;\, \zeta\, ;\, \mathsf{commit}_{v_\tau}\ n \ \xrightarrow{\Pi}_{\diamond}\ \mathcal{H}'\, ;\, \theta_{\text{out}} \end{array}}{\tau\, ;\, \theta_{\text{in}}\, ;\, \zeta \rightarrow^{\Pi}_{*} \theta_{\text{out}}} \ \textsc{Flex}$$

$$\frac{\begin{array}{c} n \text{ fresh} \qquad v_\tau \in \text{dom}(\theta_{\text{in}}) \qquad \mathcal{H} = n \mapsto \llbracket v \rrbracket_F \\ \theta\theta_{\text{in}}\, ;\, \mathcal{H}\, ;\, \zeta\, ;\, \mathsf{commit}_{v_\tau}\ n \ \xrightarrow{\Pi}_{\diamond}\ \mathcal{H}'\, ;\, \theta_{\text{out}} \end{array}}{\tau\, ;\, \theta_{\text{in}}\, ;\, \zeta \rightarrow^{\Pi}_{*} \theta_{\text{out}}} \ \textsc{Rigid}$$

$$\frac{\tau\, ;\, \theta_{\text{in}}\, ;\, \zeta \rightarrow^{\Pi}_{*} \theta \qquad \tau\, ;\, \theta\, ;\, \zeta \rightarrow^{\Pi}_{*} \theta_{\text{out}}}{\tau\, ;\, \theta_{\text{in}}\, ;\, \zeta \rightarrow^{\Pi}_{*} \theta_{\text{out}}} \ \textsc{Trans}$$

$$\tau\, ;\, \theta\, ;\, \zeta \rightarrow^{\Pi}_{*} \theta \ \textsc{Finish}$$

Figure 18: Defaulting rules.

```
type I α = ErrorT Err (State (θ,ζ)) α
run :: I α → θ → ζ → (θ,α)
```

Figure 19: The *run* function.

5.3 Data-type Semantics

The semantics of the previous section makes some assumptions about the types of identifiers occurring in de inferencer rules. This functionality needs to be available in terms of instances for the type classes listed in Figure 20. This functionality does not have to be available for all types. Most of this functionality can be generically derived from the structure of the types.

```
class Container α where
  appSubst :: (∀β . Container β ⇒ β → β) → α → α
class Unifyable α where
  unify :: (∀β . Unifyable β ⇒ β → β → I ())
        → α → α → I ()
class Deferrable α where
  deferVars        :: α → {GuessVar}
  mkDeferValue     :: GuessVar → α
  mkFixedValue     :: GuessVar → α
  matchDeferValue :: α → Maybe GuessVar
  matchFixedValue :: α → Maybe GuessVar
```

Figure 20: Semantics on data.

A *Container* instance is required to be defined for all types containing guesses. Substitution application $\theta \Rrightarrow w$ replaces all occurrences of guess $\llbracket v \rrbracket$ with w', given $v \mapsto w'_\tau \in \theta$. It generically handles the substitution of guesses, and uses *appSubst* to traverse the type.

For the type of the identifiers of the unification rule a *Unifyable* instance needs to be defined, which asks for a definitely of the *unify*

```
unif w₁ w₂ = unif′ unif w₁ w₂
unifOne w₁ w₂ = unif′ (\_ _ → return ()) w₁ w₂

unif′ r w₁ w₂ = do w₃ ← update w₁; w₄ ← update w₂
                   unif″ r w₁ w₂

unif″ _ w₁    w₂    | w₁ ≡ w₂             = return ()
unif″ _ ⟦v₁⟧  ⟦v₂⟧                        = compose v₁ v₂
unif″ _ ⟦v⟧   w     | v ∉ deferVars w     = commit v w
                    | otherwise           = fail "occur check"
unif″ _ w     ⟦v⟧   | v ∉ deferVars w     = commit v w
                    | otherwise           = fail "occur check"
unif″ r w₁    w₂                          = unify r w₁ w₂
```

Figure 21: Unification and guess specialization.

function. Its sole purpose is to succeed if and only if the heads of the two inputs are structurally equal. For the rest, it should delegate to its recursion parameter. This function does not have to deal with guesses, since those are handled generically by the **unif** function (Figure 21).

The **unif** function deals with guesses by committing concrete type information to the guess, unless both values are guesses. In that case, it takes the composition of the two. This means that the two guesses are substituted with a single guess, such that when information is committed to this single guess (in the minimal scope of the two), the deferred statements of the original guesses are sequenced after each other. Since we require confluence with respect to the order information about guesses is found, the execution order of these guesses is allowed to be arbitrary. Also note that not all *Unifyable* types have to be *Deferrable*, depending on the properties of the type. We omitted this detail here, as it is only a minor detail, and the code for **unif** would be considerably more complicated.

6. Conclusion

Type rules of declarative type systems contain non-deterministic aspects. These aspects are problematic when writing an inference algorithm. We presented a domain specific language for inferencers that has special syntax to formulate algorithms to resolve these non-deterministic aspects. The main concept is a first-class guess, which acts as a remote control to a deferred derivation. By manipulating guesses, the deferred derivations can be scheduled such that decisions are made at the moment sufficient information has become available. The result is that we can write inference algorithms by means of annotating the declarative rules of a type system, describing the global scheduling locally, without breaking the relative isolation of the type rules, and without breaking soundness with respect to the original rules.

Future Work We intend to formalize the type system of UHC (Dijkstra et al. 2007), and generate portions of the inference algorithm from this description. We made design decisions that the generated algorithm to be reasonably efficient. Although we conceptually explained semantics of the language in monadic terms, we actually generate code for multi-pass higher-order attribute grammars. An open question is still if we can keep the complexity of some of UHC's efficient data structures hidden from the type rules.

Acknowledgments

This work was supported by Microsoft Research through its European PhD Scholarship Programme. Thanks to the LerNet Alpha-project to have made it possible to carry out the research for this paper at the Universidade Federal de Minas Gerais in Brazil. Finally, Arie wants to thank Annette Bieniusa for her extensive proof reading, and the Universit at Freiburg for providing a friendly and hospitable environment to work on the actual writing of the paper.

References

Jonathan Aldrich, Robert J. Simmons, and Key Shin. Sasylf: an educational proof assistant for language theory. In *FDPE '08: Proceedings of the 2008 international workshop on Functional and declarative programming in education*, pages 31–40, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-068-5. doi: http://doi.acm.org/10.1145/1411260.1411266.

Sergio Antoy. Optimal non-deterministic functional logic computations. In Michael Hanus, Jan Heering, and Karl Meinke, editors, *ALP/HOA*, volume 1298 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1997. ISBN 3-540-63459-2.

Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

Atze Dijkstra and Doaitse S. Swierstra. Exploiting type annotations. Technical Report UU-CS-2006-051, Department of Information and Computing Sciences, Utrecht University, 2006a.

Atze Dijkstra and S. Doaitse Swierstra. Ruler: Programming type rules. In Masami Hagiya and Philip Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2006b. ISBN 3-540-33438-6.

Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The structure of the essential haskell compiler, or coping with compiler complexity. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *IFL*, volume 5083 of *Lecture Notes in Computer Science*, pages 57–74. Springer, 2007. ISBN 978-3-540-85372-5.

Michael Hanus. The integration of functions into logic programming: From theory to practice. *J. Log. Program.*, 19/20:583–628, 1994.

Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007.

Daan Leijen. Flexible types: robust type inference for first-class polymorphism. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 66–77. ACM, 2009. ISBN 978-1-60558-379-2.

Michael Y. Levin and Benjamin C. Pierce. Tinkertype: a language for playing with formal systems. *J. Funct. Program.*, 13(2):295–316, 2003.

Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, 2007.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag. ISBN 3-540-06859-7.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. Ott: effective tool support for the working semanticist. In Ralf Hinze and Norman Ramsey, editors, *ICFP*, pages 1–12. ACM, 2007. ISBN 978-1-59593-815-2.

Martin Sulzmann and Peter J. Stuckey. HM(X) type inference is CLP(X) solving. *J. Funct. Program.*, 18(2):251–283, 2008.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. FPH: first-class polymorphism for haskell. In James Hook and Peter Thiemann, editors, *ICFP*, pages 295–306. ACM, 2008. ISBN 978-1-59593-919-7.

Appendices to be moved to a technical report.

A. Static Semantics

In this section we define the static semantics of the type inference language. This semantics expresses when a type inferencer $(\Sigma^*, r^*, H_\lambda)$ is correctly typed. Concretely, this means that all identifiers are defined before used, all used schemes are defined, identifiers participating in equality, defer and commit statements have the required properties defined for their types, and Haskell fragments have a type corresponding to the type of its inputs and outputs. When this is the case, then compiling the inferencer rules to an algorithm in Haskell gives a type correct Haskell program, and during the evaluation of the inferencer rules according to the operational semantics of Section 5, the values in the heap have the type of the identifiers if this was the case for the initial heap.

$$\begin{array}{ll}
\vdash r_s & \text{(rule judgment)} \\
\overline{\alpha}\,;\Gamma_{\text{in}} \vdash c : \Gamma_{\text{out}} & \text{(statement judgment)} \\
H_\lambda \vdash e : \tau & \text{(Haskell judgment)}
\end{array}$$

Figure 22: Structure of static semantics judgments.

The structure of the typing judgments is given in Figure 22. The rules may refer to the set of schemes Σ^*, which is assumed as a constant. All schemes are explicitly typed. Local identifiers of an inferencer rule have implicit types which are directly related to the explicit types of schemes due to bindings, or to a type of a Haskell fragment due to inputs and outputs.

We give type rules for two typing judgments, one for a rule and one for a statement. The other judgments are rather trivial and left out. For the validity of schemes we want to remark that the names of identifiers in the input environment must be disjoined to those of the output environment, and that all types in the schemes must have a correct kind with respect to the types in H_λ. About the typing judgment for statements, we remark that it mentions types $\overline{\alpha}$. These are the types over which the scheme, rule, and statements are polymorphic. Technically, the types of identifiers in the environments may have free variables, but only if their explicitly occur in $\overline{\alpha}$. Also, the output environment contains exactly the types for the outputs of the premise, whereas the input environment contains at least the types for the identifiers that are input to the premise.

$$\frac{\begin{array}{c}\forall\overline{\alpha}.\,\Gamma_{\text{in}} \vdash_s \Gamma_{\text{out}} \in \Sigma^*(s) \\ \text{dom}(\Gamma_{\text{in}}) = \text{dom}(\Delta_{\text{in}}) \qquad \text{dom}(\Gamma_{\text{out}}) = \text{dom}(\Delta_{\text{out}}) \\ \overline{\alpha}\,;\Gamma_{\text{in}}(\Delta_{\text{in}})\,\Gamma'_{j<i} \vdash c_i : \Gamma'_i,\ 1 \leqslant i \leqslant k \\ \Gamma_{\text{out}}(\Delta_{\text{out}}) \subseteq \Gamma_{\text{in}}(\Delta_{\text{in}})\,\Gamma'_{j\leqslant k}\end{array}}{\vdash c_1,\ldots,c_k;(\Delta_{\text{in}} \vdash_s \Delta_{\text{out}})}\ \textsc{Rule}$$

Figure 23: Rule typing rule.

To type check an inferencer rule, we check that a scheme has been defined for it, and verify the define-before-use requirement on the premises. Outputs of these premises are accumulated, and the next premise in the sequence may use any of these outputs. The local identifiers that are connected to the types of the scheme due to bindings, must have types that agree with the types of the scheme.

The typing rules for statements are listed in Figure 24. The typing judgement states that given some types $\overline{\alpha}$, and an environment Γ_{in} stating which identifiers are in scope and what type and properties these have, that the statement produces outputs with types Γ_{out}. We now focus at some aspects of these rules.

$$\frac{\forall\overline{\beta}.\,\Gamma'_{\text{in}} \vdash_s \Gamma'_{\text{out}} \in \Sigma^*(s) \qquad [\overline{\beta} := \overline{\tau}]\,\Gamma'_{\text{in}} \subseteq \Gamma_{\text{in}}(\Delta_{\text{in}})}{\overline{\alpha}\,;\Gamma_{\text{in}} \vdash (\Delta_{\text{in}} \vdash \Delta_{\text{out}}) : \Delta_{\text{out}}([\overline{\beta} := \overline{\tau}]\,\Gamma'_{\text{out}})}\ \textsc{Scheme}$$

$$\frac{n_1 ::_{\rho_1} \tau \in \Gamma \qquad n_2 ::_{\rho_2} \tau \in \Gamma \qquad \rho_1 \neq \emptyset \qquad \rho_2 \neq \emptyset}{\overline{\alpha}\,;\Gamma \vdash n_1 \equiv n_2 : \emptyset}\ \textsc{Unify}$$

$$\frac{\begin{array}{c}\tau_{\text{in}} = \Gamma_{\text{in}}(n_1) \to \ldots \to \Gamma_{\text{in}}(n_k) \\ \tau_{\text{out}} = (\tau_{1,\rho_1},\ldots,\tau_{l,\rho_l}) \qquad H_\lambda \vdash e : \forall\overline{\alpha}.\,\tau_{\text{in}} \to I\,\tau_{\text{out}} \\ \Gamma_{\text{out}} = m_1 ::_{\rho_1} \tau_1,\ldots,m_l ::_{\rho_l} \tau_l\end{array}}{\overline{\alpha}\,;\Gamma_{\text{in}} \vdash \textsf{exec}\,e :: (n_1 \ldots n_k) \to (m_1 \ldots m_l) : \Gamma_{\text{out}}}\ \textsc{Execution}$$

$$\frac{\begin{array}{c}\overline{\alpha}\,;\Gamma_{\text{in}} \vdash c_s : \Gamma_{\text{out}} \qquad \Gamma_{\text{in}}(m_i) = \Gamma_{\text{out}}(m'_i),\ 1 \leqslant i \leqslant k \\ n^* \subseteq \text{dom}(\Gamma_{\text{in}}) \qquad n^* \subseteq \text{dom}(\Gamma_{\text{out}})\end{array}}{\overline{\alpha}\,;\Gamma_{\text{in}} \vdash \textsf{fixpoint}^{n^*}_{m_1 \mapsto m'_1,\ldots,m_k \mapsto m'_k}\ c_s : \Gamma_{\text{out}}}\ \textsc{Fixpoint}$$

$$\frac{\begin{array}{c}\Gamma_{\text{in}}\,\Gamma'_{j<i} \vdash c_i : \Gamma'_i,\ 1 \leqslant i \leqslant k \\ \Gamma_{\text{out}} = \Gamma'_{j\leqslant k}\ \{n, m^*\} \qquad n ::_{\text{d}} \tau \in \Gamma_{\text{out}}\end{array}}{\overline{\alpha}\,;\Gamma_{\text{in}} \vdash \textsf{defer}^{m^*}_n\ c^*_1,\ldots,c^*_k : \Gamma_{\text{out}}}\ \textsc{Defer}$$

$$\frac{n ::_{\text{d}} \tau \in \Gamma_{\text{in}}}{\overline{\alpha}\,;\Gamma_{\text{in}} \vdash \textsf{commit}_{v_\tau}\ n : \emptyset}\ \textsc{Commit}$$

$$\frac{\overline{\alpha}\,;Env'_{j<i},\Gamma_{\text{in}} \vdash c_i : \Gamma'_i,\ 1 \leqslant i \leqslant k}{\overline{\alpha}\,;\Gamma_{\text{in}} \vdash \textsf{fixate}_\tau\,c_1,\ldots,c_k : Env'_{j\leqslant k}}\ \textsc{Fixate}$$

Figure 24: Statement typing rule.

Polymorphism. A limited form of polymorphism is allowed for the types of the inferencer rules, by allowing the types to be parametrized over type variables $\overline{\alpha}$. This is useful when in order to be able to reuse some of the inferencer rules, or when the syntax of the language we are writing an inferencer for is itself polymorphic (for example, parametrized over the types of variables). In fact, we will silently also allow ad-hoc polymorphism by having a set of type class constraints over these variables $\overline{\alpha}$, for example to be able to show values of such a polymorphic type, to test for equality, or to use such values in maps.

This polymorphism is visible at two places. In the scheme-rule, when instantiating a scheme polymorphic in $\overline{\beta}$, we can choose the types for these variables. And in the execution-rule, the type of the monadic function must be polymorphic over the variables the scheme itself is polymorphic.

Haskell. To type monadic functions, we use Haskell's typing relation with an initial environment H_λ (containing several utility functions, data types, etc.). The monadic function is a function of taking some of the inputs of the execution-statement, and returning the outputs in monad I.

Properties. Some statements can require additional semantics defines on the values they operate on. For example, in order to test two values for equality in the unify-rule, we require a *Unify*-instance to be defined on the type. This is encoded in the language as a property ρ of an identifier. There are three properties: none, unifyable, and deferrable. When an identifier has the deferrable property, there are instances of both *Unifyable* as *Deferrable* for its type. The commit and defer statements require this deferrable property to be defined for the identifiers they act on.

B. Soundness Almost For Free

In general, it is hard to prove that a concrete type inference algorithm is sound with respect to the type system it is based on. Formally this means that if the inferencer manages to infer a type for some program value, that this is indeed a correct type for the program value according to the type system. However, when the inference algorithm is described with the inferencer rules, we almost get the soundness almost for free.

Almost free. The almost-part is due to some assumptions that we need to make:

- The inferencer rules contains concrete algorithms in the form of Haskell fragments at the places where type rules has (non-judgement) premises. For example, when a type rule has a premise $\overline{\alpha} \# \Gamma$, the the type inferencer rule has a premise

  $$\textbf{let}\ \overline{\alpha} = \mathit{ftv}\ \mathit{ty} - \mathit{ftv}\ \Gamma$$

  . In the first case, we only state a demand on $\overline{\alpha}$, in the later case we precisely define what $\overline{\alpha}$ is. For soundness, we need the guarantee that the code fragment ensures that the constraint on $\overline{\alpha}$ is satisfied.
- The inferencer rules may be more fine-grained than the type rules in order to deal with syntax-directness and explicit scheduling or pipelining of certain rules. For example, for the HML type system, we have a master rule for instantiation with the sole purpose to orchestrate the specific rules for instantiation. Therefore, we require an erasure function $\lfloor \cdot \rfloor :: \pi \rightarrow \pi$ that eliminates the extra structure from the derivation, such that a derivation is left that matches the structure of the type rules.

Notation. Let $D_T(s, \mathcal{H})$ stand for the set of all derivations π that are valid according to type system T for an instantiation of scheme s of T, with the bindings for the identifiers of the scheme in $\mathcal{H}$.

Let $\Pi(\pi)$ stand for the substitution and merging of a derivations reference $!\iota$ in π with the derivations named ι in Π. Let $\Pi_*(\pi)$ stand for the fixpoint. If Π is complete, then there is no reference left in the result.

Soundness.

Theorem B.1. *Suppose that* $(\Sigma^*, r^*, H_\lambda)$ *is an inferencer for some type system* T*,* $\mathcal{H}_0$ *and* $\mathcal{H}_1$ *are some heap, and* c_s *is an instantiation of one of the schemes of* T*. Now suppose that there is some substitution* θ *such that:*

$$\emptyset \,;\, \mathcal{H}_{in} \,;\, 0 \,;\, \mathsf{fixate}\, c_s \ \xrightarrow[\pi]{\Pi}\ \mathcal{H}_{out} \,;\, \theta$$

Then:

$$\lfloor \Pi_*(\pi) \rfloor \in D_T(s, \mathcal{H}_{out}\mathcal{H}_{in})$$

Proof. We give a sketch. First note that only fixate-statements introduce a scope, and also guarantee that there are no deferred statements remaining in this scope. More concretely, θ does not have any deferred statements. Derivations references are only introduced by a deferred-statement, which also brings equally named deferred statements in scope. Since these have all been resolved, it means that Π is complete, and thus $\Pi_*(\pi)$ is a full derivation without any references.

By taking the erasure of this derivation, we obtain a derivation π'. In order to show that $\pi' \in D_T(s, \mathcal{H}_{\text{out}}\mathcal{H}_{\text{in}})$, we need to prove for each node in π', the bindings in its heap $\mathcal{H}$, satisfy the premises of the rule in T corresponding with the node.

If there is no guessing involved, then we know that Haskell fragments have successfully run and ensured that these premises did hold. Now, when guessing was involved, this means that a certain order of evaluation was taken in order to produce the values. Some values where inspected before the full values where known. We now need to show that the same values would be observed when this evaluation would have occurred at the very end of the inference process.

We use an important property: all values are constant up to the guesses. Once a commit has been done on a guess, it is never rolled back. This has as consequence that once we observe that a value has a certain structure, this value can be considered to always keep having this structure. Therefore, a Haskell fragment executing too late does not matter, but what if it executed too early, and thus saw only a partial value?

There are two cases to consider. The first case is that enough of the value was known to complete evaluation. These portions of the value cannot have changed until the end of the inference process, and thus that evaluation would still be valid at a later time. In the second case, not enough of the value was known, and some of the evaluation was deferred. In that case, since all deferred statements have executed, this means that the evaluation was done successfully at a later time. □

Remarks. This section talked about soundness only. Dual to soundness is completeness. Unfortunately, completeness cannot be proved in general, because we can encode type systems for which no inference algorithm exists (for example, implicitly typed System F). However, we can ask ourselves the question what constraints we need on the type system and the Haskell fragments in the inferencer rules, in order to be able to prove completeness. As for now, we do not have an answer to this, otherwise, interesting question. Also, we note that one wants to experiment with inference algorithms. Soundness with respect to the type system is immediately wanted, but completeness is something we expect only to achieve after playing around sufficiently and making the right implementation decisions.