Iterative Type Inference with Attribute Grammars

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

Abstract

Type inference is the process of constructing a typing derivation while gradually discovering type information. During this process, inference algorithms typically make subtle decisions based on the derivation constructed so far.

Because a typing derivation is a decorated tree we aim to use attribute grammars as the main implementation tool. Unfortunately, we can neither express iteration, nor express decisions based on intermediate derivations in such grammars.

We present the language RULER-FRONT, a conservative extension to ordered attribute grammars, that deals with the aforementioned problems. We show why this extension is suitable for the description of constraint-based inference algorithms.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Languages

1. Introduction

Attribute grammars (AGs) are traditionally used to specify the static semantics of programming languages [Knuth 1968]. Moreover, when semantic rules of an AG are written in a general purpose programming language, the AG can be compiled into an (efficient) multi-visit tree walk algorithms that implements the specification [Kastens 1980; Kennedy and Warren 1976].

We implemented a large portion of the Utrecht Haskell Compiler (UHC) [Dijkstra and Swierstra 2004; Fokker and Swierstra 2009] with attribute grammars using the UUAG system [Universiteit Utrecht 1998]. Haskell [Hudak et al. 1992] is a purely functional programming language, with an elaborate and expressive type system. We also compile our attribute grammars to Haskell. The ideas presented in this paper, however, are not restricted to any particular language.

Attribute grammars offer us two general benefits. Dijkstra et al. [2009] give a detailed description why these advantages are important for the UHC project. Firstly, the evaluation order of semantic rules is determined automatically, unrelated to the order of appearance. Rules may be written separately from each other, and grouped by aspect [Saraiva 2002; Viera et al. 2009]. Secondly, semantic rules for idiomatic tree traversals (such as: topdown, bottom-up, and in-order) can be inferred automatically, thus allowing for concise specifications.

Two main components of UHC's type inferencer, polymorphic unification and context reduction, would benefit from an AG-based implementation. However, we implemented them directly in Haskell, because it is not straightforward to express them as an attribute grammar.

These two components exhibit two challenges to attribute grammars. Firstly, the grammar needs to produce typing derivations. The structure of such a derivation depends on what is known about types, and this information gradually becomes available during inferencing, e.g. due to unifications. This requires a mixture of tree construction and attribute evaluation, which are normally separate tasks if one takes an AG view. Secondly, construction of a subtree needs to be postponed when it depends on a type that is not known yet. Later, after we constructed more of the tree, this type may become known and the postponed construction can continue, or it becomes never known and we need to default it to some type.

An evaluator for an attribute grammar starts from a given tree (usually constructed by the parser), and has a fixed algorithm to evaluate attributes. We present an AG extension to open up these algorithms to deal with the above challenges, without loosing the advantages that AGs offer. More precisely, our contributions are:

- We present the language RULER-FRONT, a conservative extension of ordered attribute grammars. It has three concepts to deal with the above challenges.
  - We exploit the notion of visits to the tree. In each visit some attributes are computed. Visits can be done iteratively. The number of iterations can be specified based on the values of attributes.
  - Productions are chosen based on the values of attributes. A production can be chosen per visit.
  - (Attributed) Derivation trees are first class values. They can be passed around in attributes, and inspected by visiting them.

  In Section 3, we define a denotational semantics via a translation to Haskell.
- We present RULER-FRONT by example in Section 2. The example illustrates the challenges and motivates the need for the above concepts, but is necessarily dense. In related work [Middelkoop et al. 2010a,b], we give more gentle introductions to variations on this theme.
- An implementation of RULER-FRONT including several examples is available at: `https://subversion.cs.uu.nl/repos/project.ruler.systems/ruler-core/`. The implementation supports both greedy and ondemand evaluation of attributes.

GPCE'10, October 10–13, 2010, Eindhoven, The Netherlands.
Copyright © 2010 ACM 978-1-4503-0154-1/10/10. . . $10.00

- We compare our approach with other attribute grammar approaches (Section 4) as a further motivation for the need for RULER-FRONT's extensions.

2. Motivation

In this section, we show how to implement a small compiler for an example language we named SCHADOW, written with attribute grammars using RULER-FRONT. The implementation of SCHADOW poses exactly those challenges mentioned in the previous section, while being small enough to fit in this paper.

We assume that the reader is familiar with attribute grammars [Knuth 1968] and their terminology, as well as type systems and their implementation. For a more gentle introduction to both subjects including RULER-FRONT, consider [Middelkoop et al. 2010a,b].

2.1 Example to implement: the language SCHADOW

We take for SCHADOW the simply-typed lambda calculus, with two small modifications: we annotate bindings with annotations (i.e. $\lambda x^u.$), and an identifier may refer to a *shadowed* binding. For example, in the term $\lambda x^{u_1}.\lambda x^{u_2}.f\ x$, the expression x normally refers to the binding annotated with u_2. However, if the expression cannot be typed with that binding, we allow x to refer to the shadowed binding annotated with u_1 instead, if this interpretation would be well-typed[1]. The interpretation of this expression is thus by default $\lambda u_1.\lambda u_2.f\ \ u_2$, but under some conditions (defined more precisely later) it may be $\lambda u_1.\lambda u_2.f\ u_1$.

We write a compiler (i.e. a function *compile* :: *Env* → *ExprS* → *ExprT*) that takes a SCHADOW expression (of type *ExprS*), type checks it with respect to the environment *Env*, and maps it to an expression (of type *ExprT*) in the simply-typed lambda calculus.

```
-- concrete syntax and its abstract syntax in Haskell
e_S ::= x | e_S e_S |      data ExprS = VarS Ident | AppS ExprS ExprS
        λx^u.e_S           |            LamS Ident Ident ExprS
e_T ::= u | e_T e_T |      data ExprT = VarT Ident | AppT ExprT ExprT
        λu.e_T             | LamT Ident ExprT
τ   ::= α | Int | τ → τ    data Ty = TyVar Var | TyInt | TyArr Ty Ty
```

Before delving into the actual implementation, we first give a specification of the type system, together with translation rules.

$$\boxed{\Gamma \vdash e_S : \tau \leadsto e_T}$$

$$\frac{\text{innermost } x^u \text{ of all: } \quad x^u : \tau \in \Gamma}{\Gamma \vdash x : \tau \leadsto u}\ \text{VAR} \qquad \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \leadsto f' \quad \Gamma \vdash a : \tau_1 \leadsto a'}{\Gamma \vdash f\ a : \tau_2 \leadsto f'\ a'}\ \text{APP}$$

$$\frac{\Gamma, x^u : \tau_1 \vdash e : \tau_2 \leadsto e'}{\Gamma \vdash \lambda x^u.e : \tau_1 \to \tau_2 \leadsto \lambda u.e'}\ \text{LAM}$$

Each lambda is assumed to be annotated with a unique u. Rule VAR is rather informal[2]. Of all the bindings for x with the right type τ, the innermost one is to be chosen. Its annotation u is used as name in the translation. The rule APP is standard. In rule LAM, the type of a binding is appended to the environment. The annotation of the binding is used as the name of the binding in the translation.

Given an (empty) environment, a SCHADOW expression, and optionally a type, we can manually construct a derivation tree using these translation rules. The lookup of a binding poses a challenge due to context sensitivity. For example, for $\lambda x^{u_1}.\lambda x^{u_2}.f\ x$, the choice between translations $\lambda u_1.\lambda u_2.f\ \ u_1$ and $\lambda u_1.\lambda u_2.f\ \ u_2$ depends on what the program, where this expression occurs in, states about the type of f and the type of the entire expression by itself. When the context imposes insufficient restrictions to find a unique solution, the VAR states that we should default to the innermost possibility[3].

2.2 Relation to attribute grammars

We focus on writing an implementation for the above translation rules with attribute grammars using RULER-FRONT. For each relation in the above specification, we introduce a nonterminal in the RULER-FRONT code. The parameters of the relations become attributes of these nonterminals, thus also the expression part which is normally implicit in an AG based description. Derivation rules become productions, and their contents we map to semantic rules. The productions contain no terminals: only the values of attributes determine the structure of the derivation tree. Thus, the attribute grammar defines the language of derivation trees for the translation rules. Note that this differs from the normal AG approach, *where a single specific parameter of the relations fully determines the shape of the derivation tree*.

Operationally, the algorithm which constructs derivation trees picks a production, recursively builds the derivations for the children of the production, computes attributes, and if there is a mismatch between the value of an attribute and what is expected of it, backtracks to the next production.

We necessarily treat productions a bit differently in order to capture the gradual process of type inference. Final decisions about what productions are chosen to make up the derivation tree cannot be made until sufficient information is available. Therefore, we construct the derivation tree in one or more sequential passes called visits. As key feature of RULER-FRONT, *a nonterminal has productions per visit*. To make the distinction clearer, we call these clauses. During the construction of the part of a derivation for a visit, we try to apply the available clauses to build the portion of the derivation tree for that visit. When successful, we finalize the choice for that clause (similar to the cut in Prolog). The next visit can thus assume that those parts of the derivation tree constructed in previous visits is final. Moreover, we often wish to repeat a visit when type information was discovered that sheds new light upon decisions taken earlier during the visit. The upcoming example code shows this behavior several times.

In a normal AG, for each non-terminal in the tree, we have exactly one production. In our new model we can at each next visit choose again how to refine the production. So, we can regard our approach as having our productions organized as the leafs of a tree of clauses, and at each next visit we refine the choice a bit by going down one of the paths in the tree. Once we have payed all visits and made all choices we know the production precisely.

A RULER-FRONT program is a Haskell program augmented with attribute grammar code. We generate vanilla Haskell code from such a program. For each nonterminal in a RULER-FRONT program we generate a Haskell coroutine, encoded using continuations. The following is a sketch of RULER-FRONT program (we explain the various forms of syntax later):

```
module MyCompiler where
data ExprS = ...    -- Haskell data declaration
itf TransExpr ...   -- RULER-FRONT nonterm and attr declaration
itf TransExpr ...   -- additional attr declarations for nonterm
```

[1] SCHADOW is an example language that we take as a given. It can be used to model typed disambiguation of duplicately imported identifiers from modules. However, its design rationale is out of the scope of this section.

[2] Actually, the specification itself is incomplete and informal. We stress that our goal is not to rigorously discuss and prove properties about SCHADOW. Instead, we show RULER-FRONT and its concepts. The translation for SCHADOW acts as illustration.

[3] The reader may observe that this example has a strong connection to context reduction in Haskell. The challenge is that we deal with type-directed inference rules, which may be overlapping or whose selection may remain ambiguous.

```
ones = 1 : ones      -- Haskell binding
translate = sem transExpr :: TransExpr    -- embedded nonterminal
    ... rules ...                         -- (RULER-FRONT)
sem transExpr ... rules ...    -- additional rules for nonterm
sem transExpr ... rules ...    -- even more rules
```

The idea is that we define a nonterminal (introduce it, give clauses and rules) inside a Haskell expression using a sem-block. Additional clauses and rules can be given in separate toplevel sem-blocks.

The coroutines we generate from nonterminals are known as visit functions [Swierstra and Alcocer 1998]. Instantiation of a coroutine corresponds to the construction of the root node of a derivation tree. An invocation of such a coroutine corresponds to a visit in our attribute grammar description. Inputs to and outputs from the visits of the coroutine represent inherited and synthesized attributes respectively. Clauses are mapped to function alternatives of the coroutine. The internal state of the coroutine represents the derivation tree. This state contains instances of coroutines that represent the children. In Section 3 we precisely show the translation to Haskell.

2.3 Typing expressions

Figure 1 gives a rudimentary sketch of a derivation tree and some of the nonterminals we introduce. Nonterminal *compile* is the root. It has a child of nonterminal *transExpr*, and clauses for the various forms of syntax of SCHADOW. In locating a variable definition in the environment, three nonterminals play a role. *lookup* has a list of children formed by *lkMany* nonterminals. These are *lookupLam* children, representing a choice for a binding (the dotted line points to that binding). At the end of inference, at most one of these choices remains.

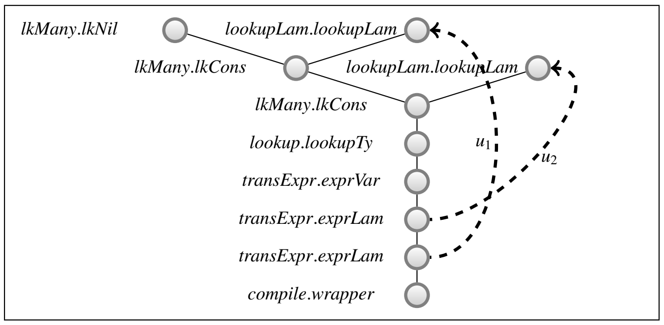

Figure 1: Sketch of derivation tree for $\lambda x^{u_1}.\lambda x^{u_2}.x$

We declare a type *TransExpr* for the nonterminal *transExpr*, which describes the interface (**itf**) of a nonterminal. The interface declares the visits and attributes of a nonterminal.

```
itf TransExpr            -- type of a nonterminal
  inh env :: Env         -- inherited attr (belongs to some visit)
  visit dispatch         -- declares a visit
    inh expr :: ExprS    -- inherited attr (belongs to visit dispatch)
type Env = Map Ident [LookupOne]   -- shown later
```

In this case, a nonterminal with the interface *TransExpr* has a single visit named *dispatch*, and two inherited attributes. The attribute *expr* contains the expression to translate. Later we will define more attributes and visits on *TransExpr*.

The visits are automatically totally ordered if this can be done, based on value-dependencies between attributes derived from all clauses in the code. Attribute *expr* is declared explicitly for visit *dispatch* (note the indenting). The *env* attribute is not: we automatically determine the earliest visit is can be allocated to.

From the above interface, we generate a Haskell type for the coroutines that we generate for nonterminals with this interface, as well as functions to invoke the coroutines and access or provide values for attributes from the Haskell code.

We introduce a nonterminal *transExpr* with interface *TransExpr* using a sem-block, embedded in Haskell code. We translate this sem-block to a vanilla Haskell coroutine expression.

```
    -- embedded toplevel Haskell code:
translate = sem transExpr :: TransExpr
```

We bind it to the Haskell name *translate*, such that we can refer to it from the Haskell code. The nonterminal name *transExpr* is global to the entire program. The Haskell name *translate* follows Haskell's scoping rules.

If we visit a *transExpr*-node for the first time, we have to see what kind of expression we have at hand. We do so by defining a number of alternative ways to deal with the node by introducing a couple of named clauses. For each of the clause we subsequently introduce further sem-rules to detail what has to be done. Unlike most of the other code that we give in this section, the order of appearance does matter for clauses, because operationally they are tried in that order.

```
sem transExpr     -- nonterminal with clauses
  clause exprVar  of dispatch   -- as the name of the first
  clause exprApp  of dispatch   -- visit we typically
  clause exprLam of dispatch    -- use dispatch
sem exprVar       -- clause with rules (or clause of next visit)
    -- semantic rule (i.e. to match against attributes)
    -- semantic rule (i.e. to define a child)
    -- perhaps clause of next visit (in scope of this clause)
```

Clauses provide a means of scoping. For example, we typically declare clauses of the next visit in the scope of the father clause (i.e. clause taken at the previous visit). These inherit all the attributes and children in scope of that clause. Otherwise, they only inherit the common attributes and children. This enforces as well that the embedded clause takes place after the enclosing clause.

With a clause we associate a couple of semantic rules, all of which may fail and cause backtracking, may have an effect on the derivation tree we are constructing, or lead to a runtime error.

-
  ```
  match pattern = code            -- match-rule
  match (VarS loc.nm) = lhs.expr  -- example
  ```

  The Haskell *pattern* must match the value of the right hand side. Evaluation of the rule requires the full pattern match to take place, or causes a backtrack to the next clause.

  Variables in the pattern refer to attributes, and have the form *childname.attrname*. The child name *lhs* is reserved to refer to the attributes of the current node. Furthermore, child name *loc* is a virtual child that conveniently stores local attributes, analogous to local variables. In the example, *lhs.expr* thus refers to the inherited attribute *expr* of the current node.

-
  ```
  pattern = code    -- assert-rule (not prefixed with a keyword)
  ```

  Similar to the above, except that the match is expected to succeed. If not, evaluation aborts with a runtime error.

-
  ```
  child name :: I = code                  -- child-rule
  child fun :: TransExpr = translate      -- example
  ```

  In contrast to a conventional attribute grammar, we construct the tree during attribute evaluation. The rule above creates a child with the given *name*, described by a nonterminal with interface *I*, and defined by the coroutine *code*. For example, *code* could be the expression *translate*, or a more complex expression. Later we show an example where the code for a child is provided by an attribute. Evaluation of the child rule creates a fresh instance of this coroutine. This child will thus have its own set of attributes defined by *I*.

- **default** *name* [= *f*]   -- default rule

  Provides a default definition for all synthesized attributes named *name* of the production and all inherited attributes of the children that are in scope. This default definition applies only to an attribute if no explicit definition is given. We come back to this rule later.

Later, we introduce more forms of rules.

The evaluation order of rules is determined automatically based on their dependencies on attributes. Rules may refer to attributes defined by previous rules, including rules of clauses of previous visits. Similarly, attributes are mapped automatically to visits based on requirements by rules. Cyclic dependencies are considered to be a static error. The rules may be scheduled to a later visit, except for match-rules. These are scheduled in the visit of the clause they appear in. Visits to children are determined automatically based on dependencies of attributes of the children. If a visit to a child fails, which is the case when none of the children's clauses applies, the complete clause backtracks as well.

The following is part of the clause for *exprVar*. It states that the value of attribute *lhs.expr* must match the *VarS* constructor.

```
sem exprVar     -- clause exprVar of nonterminal transExpr
  match (VarS loc.nm) = lhs.expr    -- if succesful, defines loc.nm
```

The names of inherited and synthesized attributes do not have to be distinct. Attribute variables in patterns refer to *synthesized* attributes of *lhs*, or *inherited* attributes of the children. Likewise, attribute variables in the right-hand side of a match refer to *inherited* attributes of *lhs* or *synthesized* attributes of the children. This ensures that attribute occurrences are uniquely identifyable.

The following clause for *exprApp* demonstrates the use of child-rules. It introduces two children *f* and *a* with interface *TransExpr*, represented as instances of the *translate* coroutine.

```
sem exprApp     -- clause exprApp of nonterminal transExpr
  match (AppS loc.f loc.a) = lhs.expr   -- is it an AppS?

  child f :: TransExpr = translate      -- recurse on f
  child a :: TransExpr = translate      -- recurse on a

  f.expr     = loc.f                    -- pass the
  a.expr     = loc.a                    -- expressions

  f.env      = lhs.env                  -- pass the
  a.env      = lhs.env                  -- environments
```

The last two lines express that the environment is passed down unchanged. We may omit these rules, and write the rule **default** *env* instead. When a child has an inherited attribute *env*, but no explicit rule has been given, and the production has *lhs.env*, then that value is automatically passed on.

```
sem transExpr   -- for all clauses of transExpr
  default env
```

There are several variations of default-rules. We see later e.g. a default rule for synthesized attributes.

We skip the clause *exprLam* for now, and consider types and type inference first. As usual with type inference, we introduce type variables for yet unknown types, and compute a substitution that binds types to these variables.

```
itf TransExpr              -- extend interface of TransExpr
  syn ty      :: Ty        -- synthesized attr of some visit
  chn subst₁  :: Subst     -- chained attr auto of some visit

data Subst   -- left implicit: a mapping from variables to types
```

The chained attribute $subst_1$ stands for both an inherited and synthesized attribute with the same name. We can see this as a substitution that goes in, and comes out again updated with new type information that became available during the visit. We get automatic threading of the attribute through all children that have a chained attribute with this name, using the default-rule:

```
sem transExpr   default subst₁
```

To deal with types and substitutions, we define several helper nonterminals.

```
itf Lookup                 -- finds a pair (nm, ty′) ∈ ty
  inh nm    :: Ident       -- such that ty′ matches ty.
  inh ty    :: Ty
  inh env   :: Env

itf Unify                  -- computes a substitution s such
  visit dispatch           -- that s (ty₁) equals s (ty₂), if
    inh ty₁ :: Ty          -- possible. Attributes for the
    inh ty₂ :: Ty          -- substitution and errors added later.

itf Fresh                  -- produces a fresh type
  syn ty    :: Ty
  chn subst :: Subst

lookup = sem lookupTy :: Lookup
unify  = sem unifyTy  :: Unify
fresh  = sem freshTy  :: Fresh
```

The implementation of *fresh* delegates to a library function *varFresh* on substitutions.

```
sem freshTy
  (lhs.subst, loc.var) = varFresh lhs.subst
  lhs.ty = TyVar loc.var
```

We did not explicitly declare any visits for *freshTy*. In that case, it consists of a single anonymous visit. Similarly, when a visit does not have any declared clauses, it consists of a single anonymous clause.

Interesting to note here is that we can wrap any Haskell function (including a data constructors) as a nonterminal, and represent an application of this function as child of the production. This is convenient in case of *fresh*, because we use default rules to automatically deal with the substitution attribute. Overdoing this, however, obscures the code.

Both *fresh* and *lookup* are of use to refine the implementation of *exprVar*. With *fresh* we get a fresh variable to use as the type of the expression. Lookup then ensures that at some point this fresh type is constrained in the substitution to the type of a binding for the variable.

```
sem exprVar     -- repeated sem-block: extends previous one
  child fr :: Fresh    = fresh
  child lk :: Lookup = lookup
  lk.nm = loc.nm    -- pass loc.nm to lk (loc.nm matched earlier)
  lk.ty  = fr.ty    -- pass the fresh type to lk
  lhs.ty = fr.ty    -- also pass it up
```

The substitution we pass to child *fr*. The substitution that comes out, we pass up to the parent.

```
sem exprVar
  sem exprVar   rename subst := subst₁ of fr

  fr.subst    = lhs.subst₁   -- pass down
  lhs.subst₁ = fr.subst      -- pass up
```

Recall that $subst_1$ is a chained attribute, hence there is an inherited $lhs.subst_1$, and a synthesized $lhs.subst_1$. These names are not ambiguous: the right hand side of the rule refers to the inherited attribute, the left hand side to the synthesized. With a rename-rule, we rename attributes of children to choose a more convenient name, for example to benefit from default-rules. The two explicit rules may actually be omitted, because of default-rule mentioned earlier.

In the application clause, we use the *unify* nonterminal to express that various types should match.

```
sem exprApp
  child fr :: Fresh = fresh
child u :: Unify = unify
u.ty_1  = f.ty
u.ty_2  = TyArr a.ty fr.ty
lhs.ty = fr.ty
rename subst := subst_1 of fr   -- for default-rule
```

Rules for the substitution may be omitted. The default-rule threads it properly through the *fr* and *u* children, which (after renaming) both have a $subst_1$ chained attribute.

2.4 Unification

So far the example can be implemented with most attribute grammar systems that operate on a fixed abstract syntax tree [Dijkstra and Swierstra 2004, 2006]. In the above example, the choice of productions solemnly depends on the *expr* inherited attribute. The attribute grammar is directly based on the grammar of expressions. In the remainder of this section, we move beyond such systems. For unification, we allow a selection of production based on two inherited attributes: the attributes ty_1 and ty_2 of interface *Unify* defined above.

The idea behind unification is to recursively compare these types. If one is a variable, then the other type is bound to that variable in the substitution.

```
sem unifyTy
  clause matchEqVars of dispatch  -- the same variables
  clause matchVarL   of dispatch  -- left a variable
  clause matchVarR   of dispatch  -- right a variable
  clause matchArr    of dispatch  -- both an arrow
  clause matchBase   of dispatch  -- both the same constant
  clause matchFail   of dispatch  -- failure
```

To implement these clauses, we need additional infrastructure to obtain the free variables of a type, and bind a type in the substitution. The actual implementations we omit, since these are similar to other examples in this section.

```
itf Ftv inh ty     :: Ty      -- determines free vars of ty
        inh subst  :: Subst   -- after applying the subst
        syn vars   :: [Var]
itf Bind inh var   :: Var     -- appends to subst:
         inh ty    :: Ty      -- [var := ty]
         chn subst :: Subst
ftv  = sem ftv  :: Ftv    -- impl. with RULER-FRONT
bind = sem bind :: Bind   -- wrapper around library fun
```

We define several additional attributes on the *Unify* nonterminal. For the synthesized attributes *success* and *errs*, we give a default definition of the form **default** $\cdot.\cdot = f$. This function f gets as first parameter a list of values of attributes $\cdot.\cdot$ of the children that have this attribute. If the function is not given, we use the Haskell function *last* for f.

```
itf Unify
  visit dispatch
    chn subst_1  :: Subst
    syn success :: Bool   -- True iff unification succeeds
    syn changes :: Bool   -- True iff any variables were bound
  visit outcome
    inh subst_2  :: Subst  -- take subst_2 more recent as
    syn errs    :: Errs   -- subst_1 for better error messages
sem unifyTy
  default success = and    -- and [ ] = True
  default changes = or     -- or [ ] = False
  default errors  = concat

  loc.ty_1 = tyExpand lhs.subst_1 lhs.ty_1   -- apply subst one level
  loc.ty_2 = tyExpand lhs.subst_1 lhs.ty_2   -- apply subst one level
```

The inherited types need to be compared with what is known in the substitution to ensure that we do not bind to a variable twice. Hence we introduce attributes $loc.ty_1$ and $loc.ty_2$ that are computed by applying the substitution to the two inherited types that are to be unified. Their values are shared among all clauses and are computed only once. We match on these values to select a clause.

```
sem matchEqVars   -- applies if we get two equal vars
  match True  = same lhs.ty_1 lhs.ty_2 ∨ same loc.ty_1 loc.ty_2
  -- embedded Haskell code:
same (TyVar v_1) (TyVar v_2) | v_1 ≡ v_2 = True
same _                              = False
sem matchVarL    -- a yet unknown type left
  match (TyVar loc.var) = loc.ty_1
  loc.ty                = loc.ty_2
sem matchVarR    -- a yet unknown type right
  match (TyVar loc.var) = loc.ty_2
  loc.ty                = loc.ty_1
sem matchVarL matchVarR      -- common part of above
  child fr :: Ftv = ftv      -- determine free fr.vars
  fr.ty = loc.ty             -- of loc.ty

  child b :: Bind = bind     -- add substitution
  b.var = loc.var            -- [loc.var := loc.ty]
  b.ty  = loc.ty

  rename subst := subst_1 of fr b

  loc.occurs     = loc.var ∈ fr.vars   -- occur check
  lhs.subst_1     = if loc.occurs then lhs.subst_1 else b.subst_1
  lhs.success    = ¬ loc.occurs
  lhs.changes    = ¬ loc.occurs

  lhs.errs =    if loc.occurs
                then [CyclErr lhs.subst_2 loc.var loc.ty]
                else [ ]
sem matchArr    -- t_1 → t_2 left and t_3 → t_4 right
  match (TyArr t_1 t_2) = loc.ty_1
  match (TyArr t_3 t_4) = loc.ty_2

  child l :: Unify    = unify   -- recurse with argument types
  child r :: Unify    = unify   -- recurse with result types
  l.ty_1 = t_1 ;   l.ty_2 = t_3 ;   r.ty_1 = t_2 ;   r.ty_2 = t_4
sem matchBase    -- applies when e.g. both are TyInt
  match True = loc.ty_1 ≡ loc.ty_2
sem matchFail    -- mismatch between types
  lhs.success    = False
  lhs.errs       = [UnifyErr lhs.subst_2 lhs.ty_1 lhs.ty_2]
```

The clauses of *unifyTy* are total, thus there is always one that applies, with *matchFail* as fallback. The visits to unification thus always succeed. Potential problems that arose during unification can be inspected through attributes *success* and *errs*.

We now have the mechanisms available to deal with the case of a lambda expression. For the type of the binding, we introduce a fresh type *fr.ty*, and add this type together with the name to the environment.

```
sem exprLam
  match (LamS loc.nm loc.u loc.b) = lhs.expr

  child b :: TransExpr = translate   -- recurse
  child fr :: Fresh    = fresh
  rename subst := subst_1 of fr

  b.expr = loc.b
  b.env  = insertWith (++) loc.nm [loc.lk] env   -- append
  lhs.ty = TyArr fr.ty b.ty    -- result type is fr.ty → b.ty

  loc.lk = sem lookupLam :: LookupOne   -- see below
```

Environments are treated a bit differently. Instead of transporting the information needed to construct the lookup-derivation tree in *exprVar*, we transport a coroutine *loc.lk* defined in *exprLam*. We define the nonterminal *lookupLam* belonging to *loc.lk* locally in *exprLam*, such that we access to the attributes of *exprLam*.

```
itf LookupOne     -- interface of nested lookupLam
  visit dispatch inh ty :: Ty
```

The idea is that we instantiate this coroutine at the *exprVar*, then pass it the expected type of the expression, and determine if the expected type matches the inferred type of the binding. The rules for this nested nonterminal (shown later) have access to the local state (i.e. attributes) of the enclosing nonterminal. At the binding-site, we have information such as the type and annotation of the binding, which we need to construct the derivation.

2.5 Lookups in the environment

At *exprVar*, the goal is to prove that there is a binding in the environment with the right type. The overall idea is that we construct all possible derivations of bindings for an identifier, using the *lookupLam* nonterminal mentioned earlier.

When there is only one possibility, we incorporate it in the substitution, and repeat the visits. The extra type information may rule out other derivations, and result in new type information, etc. Eventually a fixpoint is reached. Of all the remaining ambiguous derivations, we pick the deepest ones, and *default* to those, by incorporating their changes into the substitution. We then repeat the process from the beginning, until no ambiguities remain. We run this process on the expression as a whole. In more complex examples that have a let-binding, this process could be repeated per binding group. In the purely functional RULER-FRONT language, we encode this necessarily imperative process using repeated invocation of visits combined with a chained substitution.

We show the implementation of the above algorithm in a step by step fashion. Recall Figure 1. Three nonterminals play an essential role: *Lookup* is invoked from the *exprVar* clause and delegates to *LookupMany* to create all derivations possible. To create one derivation, *LookupMany* creates *LookupOne* derivations, one for each nested nonterminal *lookupLam* that was put for that identifier into the environment at *exprLam*.

```
sem lookupTy   -- invoked from exprVar
  child forest :: LookupMany = lookupMany
  forest.lks    = find [ ] lhs.nm lhs.env   -- all LookupOnes
  forest.ty     = lhs.ty   -- inherited attr of Lookup
```

The *lks* attribute is a list of coroutines. The coroutine lookupMany instantiates each of them, and passes on the *ty* attribute to each.

```
itf LookupMany
  visit dispatch inh lks  :: [LookupOne]
                 inh ty   :: Ty
lookupMany = sem lkMany :: LookupMany
sem lkMany
  clause lkNil of dispatch     -- when lhs.lks is empty
  clause lkCons of dispatch    -- when it has an elem
sem lkNil
  match [ ] = lhs.lks   -- reached end of the list
sem lkCons
  match (loc.hd : loc.tl)  = lhs.lks
  child hd :: LookupOne  = loc.hd           -- taken from list
  child tl :: LookupMany = lookupMany   -- recurse
  tl.lks = loc.tl   -- remainder of the lookups
  default ty   -- pass downwards to hd and tl
```

If all the matching lookupsOnes are reduced to one, we pick that one and return its substitution. Otherwise, we return the substitution belonging to the innermost binding (which has highest *depth*).

```
itf Lookup LookupOne LookupMany
  visit resolve   -- hunt for a derivation
    chn subst :: Subst
    syn status :: Status   -- outcome of the visit
    syn depth :: Int       -- depth of the binding
  visit resolved   -- invoked afterward
data Status = Fails | Succeeds {amb :: Bool, change :: Bool}
isAmbiguous (Succeeds True _) = True
isAmbiguous _                 = False
```

Every visit is invoked at least once, unless it is declared to be hidden. We intend to invoke the *resolve* visit multiple times. We show later how this is done.

The depth information is easily determined at the binding-site for lambda expressions, with an inherited attribute *depth*, starting with 0 at the top, and incrementing it with each lambda.

```
itf TransExpr   inh depth :: Int
sem transExpr default depth = 0
sem exprLam   b.depth = 1 + lhs.depth
```

The default-rule for an inherited attribute optionally takes a Haskell expression (0 in this case), which is only used when there is no parent attribute with the same name.

The rules for *lookupLam* are relatively straightforward.

```
sem lookupLam     -- defined inside exprLam above
  child m :: Unify = unify          -- try match of binding type
  rename subst_1 := subst of m   -- to use-site type lhs.ty
  hide outcome of m   -- declare not to visit outcome
  m.ty_1 = outer.fr.ty   -- of enclosing exprLam
  m.ty_2 = lhs.ty
  lhs.status = if m.success then Succeeds False m.changes else Fails
  lhs.depth = outer.lhs.depth      -- of enclosing exprLam
```

With *hide*, we state not to invoke a visit and the visits that follow. Referencing to attributes of such a visit is considered a static error.

The *lkCons* clause makes a choice. If one derivation remains, it delivers that one's substitution as result. Otherwise, it indicates that an ambiguous choice remains. The lookup with the highest depth is by construction at the beginning of the list.

```
sem lkNil
  lhs.depth  = 0              -- lowest depth
  lhs.subst  = lhs.subst   -- no change to subst
  lhs.status = Fails
sem lkCons
  hd.subst = lhs.subst   -- passed down to
  tl.subst  = lhs.subst   -- both
  (loc.pick, lhs.status, lhs.depth, lhs.subst)
    = case hd.status of
        Fails → (False, tl.status, tl.depth, tl.subst)
        Success _ hdc →
          let status′ = case tl.status of
                          Fails          → hd.status
                          Success _ tlc → Success True (hdc ∨ tlc)
          in (True, status′, hd.depth, hd.subst)
```

When a visit is invoked again, we typically want to access some results of a previous invocation. To retain state between multiple invocations of a visits, we allow visits to take visit-local chained attributes. For example, an attribute *decided* for visit *resolve*.

```
sem lookupTy   visit.resolve.decided = False   -- initial value
```

From inside the visit, we can match on these attributes to select a clause. Furthermore, there is an implicit default rule for them.

```
sem lookupTy
  clause lkRunning of resolve       -- no final choice yet,
    match False = visit.decided   -- try again
    visit.decided = isAmbiguous lk.status
    default status depth subst      -- just pass on
  clause lkFinished of resolve   -- made final choice
    match True = visit.decided
    lhs.status    = Success False False   -- no change
    lhs.depth    = 0
    default subst
```

Children created in a visit are discarded when the visit is repeated. The state of children created by a previous visit is properly maintained if their visits are also repeated. To prevent a created child from being discarded, it is possible to save a child in an attribute. Recall that children are derivations, which are instances of a coroutine, and these are first class values, The detach-rule can exactly be used for this purpose: ⟨ *at* = **detach** *visitname* **of** *childname* ⟩ takes a child *childname* evaluated up to but not including visit *visitname*, and stores it in an attribute *at*. The attach-rule, ⟨ **attach** *visitname* **of** *childname* = *code* ⟩, takes such a child-value (defined by *code*) and attaches it to a child named *childname*. If *childname* already exists as child, the attach-rule overrules the visits starting from *visitname*.

These *resolve* visits on *lookupTy* are invoked from *resolve* visits of *transExpr*. In map *deflMap*, we maintain the substitutions of ambiguous lookups per depth. These have not been incorporated in $subst_2$ yet. Applying the deepest of those, causes a defaulting to the corresponding bindings.

```
itf TransExpr
  visit! resolve
    chn subst_2   :: Subst
    syn changes :: Bool   -- True iff subst_2 was affected
    syn deflMap :: IntMap [Subst]   -- defaulting subst/depth
```

The bang at the resolve visit indicates that all attributes must be scheduled explicitly to this visit. No attribute is automatically assigned to this visit. This gives the visit a predictable interface, which is convenient when invoking the visit explicitly, as we do later.

```
sem transExpr
  default changes = or
  default deflMap = unionsWith (++)
  default subst_2
```

For ambiguous lookups in the *exprVar*, we add to *deflMap*.

```
sem exprVar
  clause varLkAmb of resolve   -- put lk.subst in deflMap
    match (Success True _) = f.status
    lhs.deflMap = singleton lk.depth [lk.subst]
    lhs.subst_2 = lhs.subst_2        -- bypass lk.subst
  clause varLkOther of resolve   -- default rules only
```

To drive the iterations, we introduce a nonterminal *iterInner*, that invokes visit *resolve* one or more times. The iterate-rule ⟨ **iterate** *visitname* **of** *childname* = *e* ⟩ denotes repeated invocation of *visitname* on *childname*. Code *e* defines the coroutine of a special nonterminal (*iterNext*, explained later) that computes the inherited attributes for the visit of the next iteration, out of the synthesized attributes of the previous. The iteration stops when this special nonterminal does not have an applicable clause.

```
iterInner = sem iterInner :: ExprTrans
sem iterInner
  child e :: ExprTrans = translate   -- iterInner is an extra node
  e.expr = lhs.expr                  -- on top of the derivation tree
  default ...   -- omitted: same defaults as transExpr
  iterate resolve of e = next   -- until e.changed is False
  lhs.subst_2 = let pairs  = toAscList e.deflMap ++ [(0, [e.subst_2])]
                    substs = head pairs   -- deepest substitutions
                in foldl substMerge e.subst_2 substs   -- apply them
  lhs.changes = ¬ (null e.deflMap)
```

This special nonterminal has as interface the *contravariant* interface of the visit *resolve* of *ExprTrans*, i.e. the inherited attributes turn to synthesized attributes, and vice versa. The triple instead of dual colons indicate this difference.

```
next = sem iterNext ::: ExprTrans.resolve   -- one anonymous clause
  match True = lhs.changes   -- stops when there are no changes
  default subst_2   -- pass prev subst_2 into the next iter
```

Finally, we introduce a nonterminal *wrapper*, which forms the root of the derivation tree and invokes the visits on the derivation for expressions, including again an iteration of the inner loop.

```
itf Compile inh expr  :: ExprS
            inh env   :: Env
            syn subst :: Subst
            syn ty    :: Ty
compile = sem wrapper :: Compile
sem wrapper
  child e :: TransExpr = iterInner
  default env expr ty
  iterate resolve of e = next   -- repeat the inner loop
  lhs.subst = e.subst_2
```

2.6 Translation to target expression

The code so far computes the information needed to translate the source expression. The shape of the derivation is determined, and after iterations, $subst_2$ contains the substitution for the types. We wrap up with generating the target expression as attribute *trans* and collecting the errors.

```
itf ExprTrans  visit generate
                 inh subst_3 :: Subst
                 syn trans  :: ExprT
                 syn errs   :: Errs
sem exprVar    lhs.trans = VarT lk.nm'   -- lk delivers the name
sem exprApp    lhs.trans = AppT f.trans a.trans
sem exprLam    lhs.trans = LamT loc.u b.trans
sem exprDeriv  default errs = concat
itf Compile    syn trans :: ExprT
sem wrapper  default trans
               e.subst_3 = e.subst_2
```

The lookupTy nonterminal delivers the name for a variable. The alternatives were constructed in iterations of the *resolve* visits, and stored in the *loc.mbDeriv* attribute. We take it out and continue from there. From the derivations of nonterminal *lkMany*, we pick the name for the first one that has *loc.pick* equal to *True*.

```
itf Lookup  visit resolved
              syn nm' :: Ident
              syn errs :: Errs
sem lookupTy
  lhs.errs   = maybe [Err_unresolved lhs.nm] (const [ ]) lk.mbNm
  lhs.nm'    = maybe lhs.nm id lk.mbNm
itf LookupMany
  syn mbNm :: Maybe Ident
sem lkNil    lhs.mbNm = Nothing
sem lkCons lhs.mbNm = if loc.pick then Just hd.nm' else tl.mbNm
itf LookupOne   syn nm' :: Ident           -- use u as name
sem lookupLam lhs.nm' = outer.loc.u   -- defined in exprLam
```

What remains is to invoke the coroutine generated from the *compile* nonterminal, with a *ExprS* expressions, to get a type and an *ExprT* back. We omit these details.

2.7 Discussion

Performance. Clauses introduce backtracking. In the worst case, this leads to a number of traversals exponential in the size of the (longest intermediate) tree. In practice, clause selection is often a function of some inherited attributes (i.e. deterministic), which only requires a constant number of traversals over the tree. For example, this is the case for RULER-FRONT programs expressible in UUAG. We verified that programs generated from RULER-FRONT are comparable to those generated from UUAG, both in time and memory.

Expressiveness. With attributes, we conveniently compute information in one part of the tree and transport it to another part, allowing context-dependent decisions to be made. The notion of visits gives us sufficient control to steer the inference process.

On the other hand, it is not possible to simply plug a type system in RULER-FRONT and automatically obtain an inference algorithm. We provide the building blocks to write inference algorithms for many type systems, but it is up to the programmer to ensure that the result is sound and complete.

Soundness of a RULER-FRONT program is typically easy to prove. Completeness, however, is a different issue. That largely depends on decisions made about unknown types. With RULER-FRONT, we make explicit when choices are made, and when visits are repeated. We believe this helps reasoning about completeness.

Constraint-based inference. We establish the following relation to constraint-based inference. A detached derivation can be seen as a constraint, can be collected in an attribute and solved elsewhere. Solving constraints corresponds invoking visits (such as *resolve* in Section 2.5) on the derivation, potentially multiple times.

Solving a constraint may result in more constraints. We store these either in a node's state, or collect them in attributes.

A constraint is typically parametrized with information from the context that created it. We provide access to this context via nested nonterminals, which have access to the attributes of their outer nonterminals.

3. Semantics

We define RULER-CORE, a small core language for Attribute Grammars. We translate a RULER-FRONT program in two steps to Haskell. We first desugar RULER-FRONT to RULER-CORE, then translate the latter to Haskell. The separately defined attributes of RULER-FRONT are grouped together in RULER-CORE, visits are ordered, attributes allocated to visits, covariant interfaces translated to normal interfaces, rules ordered based on their attribute dependencies, and rules augmented with default rules. We omit description of this step, as it is similar to the frontend of UUAG [Universiteit Utrecht 1998], and of a variation on RULER-CORE [Middelkoop et al. 2010b]. Instead, we focus on the translation to Haskell, which precisely defines the semantics of RULER-CORE, and thus forms the underlying semantics for RULER-FRONT.

3.1 Syntax

The RULER-CORE language is Haskell extended with additional syntax for toplevel interface declarations, semantic expressions, and attribute occurrence expressions. The following grammar lists these syntax extensions.

```
i ::= itf I v̄                       -- interface decl, with visits v
v ::= visit x inh ā₁ syn ā₂          -- visit decl, with atributes a₁ and a₂
a ::= x :: τ                         -- attribute decl, with Haskell type τ

s ::= sem x :: I t                   -- semantics expr, defines nonterm x
t ::= () | visit x₁ chn x̄₂ r̄ c̄       -- visit def, with common rules r
c ::= clause x r̄ t                   -- clause definition, with next visit t

r ::= p ← e                          -- assert-rule, evaluates monadic e
   |  match p ← e                    -- match-rule, backtracking variant
   |  invoke x of c ← e              -- invoke-rule, invokes x on c, while e
   |  attach x of c :: I ← e         -- attach-rule, attaches a part. eval. child
   |  p = detach x of c              -- detach-rule, stores a child in an attr

o ::= x.x                            -- expression, attribute occurrence

x, I, p, e   -- identifiers, patterns, expressions respectively
```

There are some differences in comparison with the examples of the previous section. Invocations of visits to children are made explicit through the invoke-rule, which also represents the iterate-rule. Similarly, the attach rule also takes care of introducing children. A visit definition declares number of visit-local chained attributes $\overline{y}$, and has a number of rules to be evaluated prior to the evaluation of clauses. A clause defines the next visit, if any.

The order of appearance of rules determines the evaluation order, which allows them to be monadic. Non-monadic expressions are lifted with *return*. The monad may be any backtracking-monad, such as *Either String*, *Logic*, and *IO*. We take *IO* as example.

3.2 Example

The following example demonstrates how to to compute sum of a list of integers in two visits in RULER-CORE.

```
itf S visit v₁ inh l :: [Int] syn ∅              -- decompose list l down
      visit v₂ inh ∅          syn s :: Int       -- compute sum s up

sum′ =   sem sum :: S
  visit v₁ chn ∅ ∅
    clause sumNil                                -- when list is empty
      match [ ] ← return lhs.l                   -- match [ ] = l

      visit v₂ chn ∅ ∅                           -- no visit-local attrs
        clause sumNil₂
          lhs.s ← return 0                       -- empty list, zero sum
          ()                                     -- no next visit
    clause sumCons                               -- when list non-empty
      match (loc.x : loc.xs) ← return lhs.l      -- match (x : xs) = l
      attach v₁ of tl :: S ← return sum          -- recursive call
      tl.l ← return loc.xs                       -- l param of call
      invoke v₁ of tl ← noIterations_S           -- visit it to pass l

      visit v₂ chn ∅ ∅
        clause sumCons₂
          invoke v₂ of tl ← noIterations_S       -- visit it to get the sum
          lhs.s ← return (loc.x + tl.x)          -- sum of hd and the tl
          ()                                     -- no next visit
```

We translate a RULER-FRONT nonterminal to a coroutine, in the form of continuations. From the interface, we generate a type signature for these coroutines.

```
type S       = S_v1
newtype S_v1 = S_v1 ([Int] → IO ((), (S_v1, S_v2)))
newtype S_v2 = S_v2 (IO (Int, (S_v2, ())))
```

Inherited attributes become parameters, and synthesized attributes are returned as a tuple of results. Each visit also returns two continuations. The first continuation represents the current visit itself (which may be re-invoked with updated internal state), the second continuation represents the next visit (if any).

The coroutine *nt_sum* has *S* as type. Attributes are encoded as a variable *childIattr* or *childOattr*, depending on whether the attribute is an input or output of the clause. Clause selection relies on backtracking in the monad. When a match-statement doesn't match, a failure is generated in the monad, which we *catch* to switch to the next clause.

```
sum′ = S_v₁ vis_v₁ where
  vis_v₁ lhs_I l = (                         -- first clause of visit v₁
      do [ ] ← return lhs_I l                -- match on lhs.l
         let r = S_v₁ vis_v₁                 -- repetition cont.
             k = S_v₂ vis_v₂ where           -- next visit cont.
               vis_v₂ = (                        -- clause of visit v₂
                 do lhs_O s ← return 0           -- lhs.s computation
                    let r = S_v₂ vis_v₂          -- repetition
                        k = ()                   -- no next visit
                    return (lhs_O s, (r, k))     -- deliver result v₂
                 ) ‘catch‘ (\_ → ⊥)              -- no other clause for v₂
         return ((), (r, k))     -- deliver result of visit v₁
    ) ‘catch‘ (\_ →              -- second clause (when first clause fails)
      do (loc_L x : loc_L xs) ← return lhs_I l      -- match on lhs.l
tl_O l                  ← return loc_L xs    -- inherited attr tl.l
(S_v1 vis_tl_v1) ← return sum′        -- attach child tl
((), (_, S_v2 vis_tl_v2)) ← vis_tl_v1 tl_O l    -- first visit on tl
let r = S_v1 vis_v1                          -- repetition cont.
    k = S_v2 vis_v2 where                    -- next visit cont.
      vis_v2 = (                             -- clause of visit v2
        do (tl_I s, (_, _)) ← vis_tl_v2      -- second visit on tl
           lhs_O s ← return (loc_L x + tl_I s)    -- lhs.s
           let r = S_v2 vis_v2               -- repetition
               k = ()                        -- no next visit
           return (lhs_O s, (r, k))          -- deliver result v2
        ) ‘catch‘ (\_ → ⊥)                   -- no other clause for v2
return ((), (r, k)))    -- deliver result of visit v1
```

The above code is slightly simplified. Below, we show the general translation.

3.3 Translation

We use the following naming conventions from RULER-FRONT names to Haskell names.

```
outp "loc" x = "locL" x        inp "loc" x = "locI" x
outp "lhs" x = "lhsI" x        inp "lhs" x = "lhsS" x
outp c     x = c "S" x         inp c     x = c "I" x
outp y       = "visitS" y      inp       y = "visitI" y
vis c x = "vis_" c "_" x       nt x      = "nt_" x
vis x   = "vis_" x             ity I x = I "_" x
s I x        i I x    -- respectively, inh and syn attrs of x of I
```

From an interface declaration, we generate the types for the coroutines.

```
⟦itf I v̄⟧                 ⤳ type ⟦I⟧ = ⟦ity I x′⟧; ⟦v⟧̄_I     -- x′ next visit,
⟦visit x inh ā syn b̄⟧_I ⤳ newtype ⟦ity I x⟧ =             -- otherwise ()
   ⟦ity I x⟧ (⟦ā⟧ → IO (⟦b̄⟧, (⟦ity I x⟧, ⟦ity I x′⟧)))
```

From these interfaces, we actually also generate wrappers to interface with the coroutines from Haskell code. The translations for them bear a close resemblance to the translation of the attach and invoke rules below.

The clauses of a visit are translated to a function ⟦*vis x*⟧ that tries the clauses one by one. This function takes as parameters the coroutines (⟦*chlds*⟧) of the children in scope prior to invoking the visit, the visit-local attributes $\overline{y}$, and the inherited attributes.

```
⟦sem x :: I t⟧           ⤳ let ⟦nt x⟧ = ⟦t⟧_I in⟦nt x⟧

⟦()⟧_I                   ⤳ ()
⟦visit x chn ȳ r̄ c̄⟧_I ⤳
   let ⟦vis x⟧ ⟦chlds⟧ ⟦inp ȳ⟧ ⟦inp lhs (i I x)⟧
       = catch (do {⟦r̄⟧; ⟦c̄⟧_{I,x,ȳ}}) ⊥ in⟦ity I x⟧ ⟦vis x⟧
```

The clauses themselves translate to a sequence of statements, consisting of the translated statements of the semantic rules, and the construction of the two continuations. We partially parametrize both continuations with the updated children.

```
⟦[ ]⟧_{I,v,ȳ}                ⤳ error "no clause applies"
⟦clause x r̄ t : cs⟧_{I,v,ȳ} ⤳
   catch (do {⟦r̄⟧; let {⟦inp x ȳ = outp ȳ⟧}
             ; let {r = ⟦ity I x⟧ ⟦vis x⟧ ⟦chlds⟧ ⟦outp x ȳ⟧
                   ; k = ⟦t⟧_{I,chlds}}
             ; return (⟦outp lhs (s I x)⟧, (r, k))})
         (\_ → ⟦cs⟧_{I,v,ȳ})

⟦()⟧_{I,ks}                  ⤳ ()
⟦visit x chn ȳ r̄ c̄⟧_{I,ks} ⤳ ⟦visit x chn ȳ r̄ c̄⟧_I ⟦ks⟧ ⟦outp x ȳ⟧
```

Semantic rules translate to monadic statements. For the assert-rule, we match in a let-statement, to ensure that a pattern match failure is considered a runtime error, instead of cause backtracking in the monad.

```
⟦match p ← e⟧         ⤳ ⟦p⟧ ← ⟦e⟧
⟦p ← e⟧               ⤳ x ← ⟦e⟧; let {⟦p⟧ = x}      -- x fresh
⟦attach x of c :: I ← e⟧ ⤳ (⟦ity I x⟧ ⟦vis c x⟧) ← ⟦e⟧
⟦p = detach x of c⟧      ⤳ let {⟦p⟧ = ⟦ity I x⟧ ⟦vis c x⟧}
```

Invoke invokes a visit x (named f in the translation) on child c once, then repeats invoking it, as long as e (named g) succeeds in feeding it new input.

```
⟦invoke x of c ← e⟧ ⤳
   (⟦inp c (s I_c x)⟧, (_, k))
      ← let iter f ⟦outp c (i I_c x)⟧ = do
                  {(⟦coIty I_c x⟧ g) ← ⟦e⟧
                  ; z@(⟦inp c (s I_c x)⟧, (⟦ity I_c x⟧ f′, _))
                        ← f ⟦outp c (i I_c x)⟧
                  ; catch (do {(⟦outp c (i I_c x)⟧, _) ← g ⟦inp c (s I_c x)⟧
                              ; iter f′ ⟦outp c (i I_c x)⟧})
                          (\_ → return z)}
         in iter ⟦vis c x⟧ (outp c (i I_c x))
   ; let (⟦ity I_c x′⟧ ⟦vis c x′⟧) = k    -- x′ is next visit, or line omitted
```

Finally, we add bangs around patterns to enforce evaluation, and replace attribute occurrences with their Haskell names.

```
⟦C p̄⟧         ⤳ !(C ⟦p⟧̄)
⟦(p, . . , q)⟧ ⤳ !(⟦p⟧, ..., ⟦q⟧)
⟦c.x⟧         ⤳ !⟦inp c x⟧
⟦e⟧           ⤳ e [c.x := ⟦outp c x⟧]
```

The translation exhibits a number of properties. If the RULER-FRONT or RULER-CORE program is well typed, then so is the generated Haskell program, and vice versa. Further investigation requires a discussion of RULER-FRONT's type system, which we omit for reasons of space. Furthermore, the translation is not limited to Haskell. A translation similar to above can be given for any language that supports closures.

4. Related Work

Attribute grammars as defined by Knuth [Knuth 1968] are extensions of context free grammars. Typically, the tree is the parse tree determined a priori by a parser. For typing relations that are not syntax directed, the derivation tree is not known beforehand, which conflicts with the attribute grammar model.

Tree manipulations. There are many extensions to attribute grammars to facilitate changing the tree during attribute evaluation. Silver [v. Wyk et al. 2008], JastAdd [Ekman and Hedin 2007] and UUAG [Universiteit Utrecht 1998] support higher-order attribute grammars [Vogt et al. 1989]. These grammars allow the tree to be extended with subtrees computed from attributes, and subsequently decorated. The responsibility of selecting a production of a higher-order child lies with the parent of that child, and the choice is final. In RULER-FRONT, a child itself selects a clause to make a choice, and a choice can be made per visit.

JastAdd and Aster [Kats et al. 2009], support conditional rewrite rules, which allows rigorous changes to be made to the tree. Coordination between rewriting and attribute evaluation is tricky due to mutual influence, especially if the transformations are not confluent. To limit interplay, JastAdd's rewriting of a tree is limited to the first access of that tree, and choices finalized.

Many type inference algorithms, especially for type and effect systems, iteratively traverse the tree. Some algorithms construct additional subtrees during this process. Circular Attribute Grammars [Jones 1990], supported by JastAdd and Aster, iteratively compute circular attributes until a fixpoint is reached. UUAG and Silver can deal with circularity via lazy evaluation with streams. CAGs, however, do not support changes to the tree during these iterations. Stratego's rewrite mechanism that underlies Aster, however, is more general and can change the tree. In RULER-FRONT, a visit may be iterated several times. Each node in the derivation tree can maintain a per-visit state to keep track of newly constructed parts of the tree.

Non-deterministic trees. The attribute grammar systems above have in common that they massage a tree until it has the right form. Alternatively, a tree can be constructed non-deterministically, using e.g. logic programming languages. The grammar produces only the empty string, and the semantic rules disambiguate the choice of productions. Arbab [1986] showed how to translate attribute grammars to Prolog. However, this approach does not allow the inspection of partial *LookupOne* derivations of Section 2.5, nor the defaulting, to be implemented easily. With RULER-FRONT, we offer non-deterministic construction of the tree per visit. The notion of a visit provides an intuitive alternative for the *cut* operator.

Prolog-like approaches also offer unification mechanisms to deal with non-determinism in attribute computations. In contrast, we require the programmer to either program unifications and substitutions manually, or use a logic monad combined with a unification in the translation of Section 3.

Engelfriet and Filé [1989] shows the expressiveness of classes of attribute grammars. Unsurprisingly, deterministic AG evaluators have lower computational complexity bounds compared to non-deterministic ones. With RULER-FRONT, we target heavy compilers (i.e. UHC), that processes large abstract syntax trees, thus we need the control on the non-determinism that visits offer.

Related attribute grammar techniques. Several attribute grammar techniques are important to our work. Kastens [1980] introduces ordered attribute grammars. In OAGs, the evaluation order of attribute computations as well as attribute lifetime can be determined statically, allowing severe optimizations.

Boyland [1996] introduces conditional attribute grammars. In such a grammar, semantic rules may be guarded. Our clauses-per-visit model provides an easier yet less flexible alternative.

Saraiva and Swierstra [1999, chap. 3] describes multiple traversal functions (or visit functions [Swierstra and Alcocer 1998]). These visit functions are one-shot continuations, or coroutines without looping. We improved upon this mechanism to support iterative invocation of visits, thus encoding coroutines with loops.

5. Conclusion

We presented RULER-FRONT, a conservative extension of ordered attribute grammars, intended to describe algorithms for type inferencing, evaluators and code generators. We explained this language by example in Section 2 and described its semantics in Section 3. It has three distinct features.

Firstly, in contrast to most attribute grammar systems, construction of a derivation tree and the evaluation of its attributes is intertwined in RULER-FRONT. This allows us to define a grammar for the language of derivations of some typing relations, instead of being limited to the grammar of expressions or types.

Secondly, we use the notion of explicit visits to capture the gradual, side effectful nature of type inference. Each visit corresponds to a state transition of the derivation tree under construction. These visits may be repeated to form fixpoint iterations.

Thirdly, many inference algorithms reason about what part of the derivation is known, or is still pending, e.g. by means of constraints. In RULER-FRONT, derivation trees are first class and can be inspected by visiting them, which facilitates such reasoning in terms of attributed trees.

As future work, we started a project to describe UHC's type inference algorithm as attribute grammar with RULER-FRONT. With RULER-FRONT, we can express unification and context reduction, which paves the way for a full description of UHC's inference algorithm with attribute grammars.

Acknowledgments

This work was supported by Microsoft Research through its European PhD Scholarship Programme.

References

B. Arbab. Compiling Circular Attribute Grammars Into Prolog. *IBM Journal of Research and Development*, 30(3):294–309, 1986.

J. T. Boyland. Conditional Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):73–108, 1996.

A. Dijkstra and S. D. Swierstra. Typing Haskell with an Attribute Grammar. In *AFP*, pages 1–72, 2004.

A. Dijkstra and S. D. Swierstra. Ruler: Programming Type Rules. In *FLOPS*, pages 30–46, 2006.

A. Dijkstra, J. Fokker, and S. D. Swierstra. The Architecture of the Utrecht Haskell Compiler. In *Haskell*, pages 93–104, 2009.

T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *OOPSLA*, pages 1–18, 2007.

J. Engelfriet and G. Filé. Passes, sweeps, and visits in attribute grammars. *Journal of the ACM*, 36(4):841–869, 1989.

J. Fokker and S. D. Swierstra. Abstract Interpretation of Functional Programs using an Attribute Grammar System. *ENTCS*, 238(5):117–133, 2009.

P. Hudak, S. L. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. *SIGPLAN Notices*, 27(5):1–, 1992.

L. G. Jones. Efficient Evaluation of Circular Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–462, 1990.

U. Kastens. Ordered Attributed Grammars. *Acta Informatica*, 13:229–256, 1980.

L. C. L. Kats, A. M. Sloane, and E. Visser. Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming. In *CC*, pages 142–157, 2009.

K. Kennedy and S. K. Warren. Automatic Generation of Efficient Evaluators for Attribute Grammars. In *POPL*, pages 32–49, 1976.

D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

A. Middelkoop, A. Dijkstra, and S. D. Swierstra. Towards Dependently Typed Attribute Grammars. `http://people.cs.uu.nl/ariem/ifl10-depend.pdf`, 2010a.

A. Middelkoop, A. Dijkstra, and S. D. Swierstra. Iterative Type Inference with Attribute Grammars. `http://people.cs.uu.nl/ariem/wgt10-journal.pdf`, 2010b.

J. Saraiva. Component-Based Programming for Higher-Order Attribute Grammars. In *GPCE*, pages 268–282, 2002.

J. A. B. V. Saraiva and S. D. Swierstra. Purely Functional Implementation of Attribute Grammars. Technical report, Universiteit Utrecht, 1999.

S. D. Swierstra and P. R. A. Alcocer. Attribute grammars in the functional style. In *Systems Implementation 2000*, pages 180–193, 1998.

Universiteit Utrecht. Universiteit Utrecht Attribute Grammar System. `http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem`, 1998.

E. v. Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an Extensible Attribute Grammar System. *ENTCS*, 203(2):103–116, 2008.

M. Viera, S. D. Swierstra, and W. Swierstra. Attribute Grammars Fly First-class: how to do Aspect Oriented Programming in Haskell. In *ICFP*, pages 245–256, 2009.

H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-Order Attribute Grammars. In *PLDI*, pages 131–145, 1989.