Incremental Evaluation of Higher Order Attributes

Jeroen Bransen Atze Dijkstra S. Doaitse Swierstra
Utrecht University, The Netherlands
{J.Bransen,atze,doaitse}@uu.nl

Abstract

Compilers, amongst other programs, often work with data that (slowly) changes over time. When the changes between subsequent runs of the compiler are small, one would hope the compiler to incrementally update its results, resulting in much lower running times. However, the manual construction of an incremental compiler is very hard and error prone and therefore usually not an option.

Attribute grammars provide an attractive way of constructing compilers, as they are compositional in nature and allow for aspect oriented programming. In this work we extend previous work on the automatic generation of incremental attribute grammar evaluators, with the purpose of (semi-)automatically generating an incremental compiler from the regular attribute grammar definition, by adding support for incremental evaluation of higher order attributes, a well known extension to the classical attribute grammars that is used in many ways in compiler construction, for example to model different compiler phases.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Incremental compilers

General Terms Algorithms, Languages, Theory

Keywords incremental evaluation, attribute grammars, change propagation, program transformation, type inference

1. Introduction

Attribute grammars (Knuth 1968) are known to be well-suited for the implementation of the semantics of programming languages. There exist several attribute grammar compilers including UUAGC (Swierstra et al. 1998), JastAdd (Hedin and Magnusson 2003) and Silver (Van Wyk et al. 2010), using which many attribute grammar based compilers have been implemented. The motivating example for this paper is the *Utrecht Haskell Compiler* (Dijkstra et al. 2009). The implementation of the UHC consists of attribute grammar code combined with Haskell expressions (Peyton Jones 2003), which is compiled by the UUAGC to a Haskell program, which is then compiled by the Glasgow Haskell Compiler (GHC) to an executable. In this paper we therefore use Haskell as the basis for the implementation of our techniques.

PEPM '15, January 13–14, 2015, Mumbai, India.

ACM 978-1-4503-3297-2/15/01. . . $15.00.
http://dx.doi.org/10.1145/2678015.2682541

The typical usage of a compiler like the UHC is as follows. The programmer works on a project containing many lines of source code, and compiles and runs the code in order to test. The programmer then changes some lines of code, and compiles and runs the code again. This changing code, compiling and running repeats itself until the project is finished. A potential problem with a traditional compiler is that when the project grows, the compilation time increases, even though the changes to the compiled program are minimal between consecutive runs. To solve that problem we aim to build an *incremental compiler*, that uses results of the previous compilation to efficiently update the results based on the small changes made to its input.

Incremental compilation is not a new idea, but the problem is that building such compilers is very hard and error-prone. In our work we therefore aim to *(semi-)automatically generate* an incremental compiler from the attribute grammar definition of the regular compiler. Because data dependencies are explicit in attribute grammar definitions, we hope to statically use this dependency information to generate attribute grammar evaluators that incrementally update their output based on (small) changes to the input.

In this paper, building upon our earlier work (Bransen et al. 2014), we extend our techniques to support higher order attributes (Vogt et al. 1989). A higher order attribute is an attribute value which itself is a tree structure over which attributes can be computed. Higher order attributes are heavily used in the UHC and support for this in the context of incremental evaluation of attribute grammars is therefore essential to reach our goal. A theoretical limitation is that we rely on attribute grammars to be *linearly ordered* (Engelfriet and Filè 1982); we have reasons to believe that all practical attribute grammars fall in that class (Van Binsbergen et al. 2015).

The outline of the paper is as follows. In Section 2 we informally introduce attribute grammars together with a running example that we use in the rest of the paper to explain our technique. We then give a high-level overview of our approach in Section 3 before describing the details. Section 4 introduces earlier work on the representation of changes made to the input, and in Section 5 we describe the techniques for incremental attribute grammar evaluation without higher order attributes. This is similar to our earlier work (Bransen et al. 2014) but adapted to support the main contribution of this paper in Section 6 where the support for higher order attributes is added. In Section 7 we evaluate the results with a benchmark and we discuss some shortcomings in Section 8. Finally, we describe related work (Section 9) and conclude (Section 10).

2. Attribute grammars and running example

To illustrate the techniques described in this paper we use a simple running example that we introduce in this section. Alongside the example we informally introduce some core attribute grammar concepts and show some syntax from the *Utrecht University Attribute Grammar Compiler* (Swierstra et al. 1998). Although the

UUAGC syntax is not important for the techniques described in this paper, we believe that it might help the reader in understanding the usefulness of attribute grammars.

2.1 Running example

As running example for this paper we look at the very simple task of pretty printing expressions of the untyped lambda calculus, containing variables, lambda abstractions and applications. To illustrate the use of inherited attributes, we make the example slightly less trivial by printing no parentheses around variables or around the child expression of a lambda abstraction. Concretely, a certain term is pretty printed as follows.

```
\f.\x.(\y.y) ((f x) x)
```

This results in a string which can be parsed back unambiguously to the AST from which it was pretty printed, but not necessarily with the minimum number of parentheses: in the above example the parentheses around `f x` could have been left out.

2.2 Attribute grammars

Attribute grammars as introduced by (Knuth 1968) consist of a *context-free grammar*, a set of attribute definitions per nonterminal, and for each attribute defined at a production a semantic rule defining the value of the attribute in terms of other attributes. Attributes can be *inherited*, meaning that the value is passed from parent to child nodes in the derivation tree ("defined from above"), or *synthesized*, meaning that the value is passed from the child node to the parent node ("defined from below").

The UUAGC implements attribute grammars in a slightly different way, by letting the user specify an *abstract syntax tree* (AST) instead of a context-free grammar describing the concrete syntax. As the default back-end of UUAGC is Haskell, the AST corresponds to an algebraic data type with notation similar to that of Haskell, except that the children of a constructor are named.

Data types In UUAGC syntax the data type for representing expressions of the untyped lambda calculus is defined as follows.

$$
\begin{array}{l}
\mathbf{data}\ \mathit{TopLam} \\
\quad |\ \mathit{Top}\ \ e\ :: \mathit{Lam} \\
\mathbf{data}\ \mathit{Lam} \\
\quad |\ \mathit{Var}\ \ x\ :: \mathit{String} \\
\quad |\ \mathit{Lam}\ x\ :: \mathit{String} \\
\qquad\quad\ \ e\ :: \mathit{Lam} \\
\quad |\ \mathit{App}\ \ e_1 :: \mathit{Lam} \\
\qquad\quad\ \ e_2 :: \mathit{Lam}
\end{array}
$$

The $TopLam$ type is convenient as top level wrapper in later stages, but not strictly necessary. Such a wrapper is for example used to initialise an attribute with an empty environment.

Although we do not use the context-free grammars from the original definition (Knuth 1968) in this paper, we stick to the original terminology. Therefore, when talking about a *nonterminal* we refer to an algebraic type and a *production* refers to a constructor of that data type. For example, App is a production of Lam and has two nonterminal children. The Var production has only a terminal child, since $String$ is a built-in Haskell type and has not been defined as a nonterminal in the attribute grammar.

Attribute definitions For the pretty printing of the example we introduce two attributes: a synthesized attribute pp that contains the result of the computation, and an inherited attribute $needp$ indicating whether or not parentheses are necessary. This is defined as follows.

$$
\begin{array}{l}
\mathbf{attr}\ \mathit{Lam} \\
\quad \mathbf{inh}\ \mathit{needp} :: \mathit{Bool} \\
\quad \mathbf{syn}\ \mathit{pp}\ \ \ :: \mathit{String}
\end{array}
$$

Note that both $Bool$ and $String$ are plain Haskell types.

Semantic rules The computation of the value of the attributes is specified by means of semantic rules. In our system the semantic rules can be arbitrary Haskell expressions, in which we may refer to other attributes. The attributes that can be referred to are the inherited attributes of the production itself (coming from the parent) and the synthesized attributes of children (defined within the children). The attributes for which a value needs to be defined are the synthesized attributes of the production and the inherited attributes of the children.

A semantic rule has the form $c.a = e$, where c is the name of the corresponding child or the special name $\mathbf{lhs}$[1] to refer to the parent, a is the name of the attribute, and e is an arbitrary Haskell expression. To refer to other attributes in the expression the notation $@c.a$ is used; $@c$ refers to a terminal value.

For our example we define the semantic rules as follows.

$$
\begin{array}{lll}
\mathbf{sem}\ \mathit{Lam} & & \\
\quad |\ \mathit{Var}\ \ \mathbf{lhs}.\mathit{pp} & = & @x \\
\quad |\ \mathit{Lam}\ \mathbf{lhs}.\mathit{pp} & = & \mathit{pParens}\ @\mathbf{lhs}.\mathit{needp}\ \$ \\
 & & \quad \texttt{"\textbackslash\textbackslash"} +\!\!+\ @x +\!\!+\ \texttt{"."} +\!\!+\ @e.\mathit{pp} \\
\qquad e.\mathit{needp} & = & \mathit{False} \\
\quad |\ \mathit{App}\ \ \mathbf{lhs}.\mathit{pp} & = & \mathit{pParens}\ @\mathbf{lhs}.\mathit{needp}\ \$ \\
 & & \quad @e_1.\mathit{pp} +\!\!+\ \texttt{" "} +\!\!+\ @e_2.\mathit{pp} \\
\qquad e_1.\mathit{needp} & = & \mathit{True} \\
\qquad e_2.\mathit{needp} & = & \mathit{True}
\end{array}
$$

We use the following (Haskell) helper function for conditionally placing parentheses around an expression.

$$
\begin{array}{l}
\mathit{pParens} :: \mathit{Bool} \to \mathit{String} \to \mathit{String} \\
\mathit{pParens}\ \mathit{True}\ \ s = \texttt{"("} +\!\!+\ s +\!\!+\ \texttt{")"} \\
\mathit{pParens}\ \mathit{False}\ s = s
\end{array}
$$

At the top level we set the value of $needp$ to $False$, because parentheses are not needed around the whole expression. To do this we use a wrapper $TopLam$, in which we give $needp$ its initial value and return the pretty printed expression as the only result.

$$
\begin{array}{l}
\mathbf{attr}\ \mathit{TopLam} \\
\quad \mathbf{syn}\ \mathit{pp} :: \mathit{String} \\
\mathbf{sem}\ \mathit{TopLam} \\
\quad |\ \mathit{Top}\ \mathit{e.needp} = \mathit{False} \\
\qquad\quad \mathbf{lhs}.\mathit{pp}\ \ = @e.\mathit{pp}
\end{array}
$$

Trivial attribute equations like the last line can be omitted. The UUAGC will add such equation using default rules, in this case a *copy rule* for simply propagating an attribute value.

2.3 Standard evaluation

The standard non-incremental evaluation of attribute grammars in the UUAGC is done by generating Haskell code that performs the evaluation. The data types are a straightforward translation except that the constructor names are prefixed with the nonterminal name in order to avoid name clashes (note that UUAGC does not require production names to be unique, whereas in Haskell constructor names have to). This results in the following data type to be generated:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Lam} = \mathit{LamVar}\ \ \mathit{String} \\
\qquad\qquad\ \ |\ \mathit{LamLam}\ \mathit{String}\ \mathit{Lam} \\
\qquad\qquad\ \ |\ \mathit{LamApp}\ \ \mathit{Lam}\ \ \ \mathit{Lam}
\end{array}
$$

[1] Left-hand side, terminology coming from context-free grammars

The most straightforward way of generating the actual evaluation code is to build a function that takes a tuple of all inherited attributes as argument and returns a tuple of all synthesized attribute for each nonterminal. Because some inherited attributes may depend on synthesized attributes of that same nonterminal, this relies on Haskell's lazy evaluation to compute the result, and may even loop for cyclic attribute grammars.

In order to get static guarantees about non-cyclicity and to generate code that can also be evaluated in a strict way, we rely on linearly ordered attribute grammars (Engelfriet and Filè 1982). In a linearly ordered attribute grammar the attributes are split up into a linear sequence of *visits*, where each visit takes some inherited attributes as input and produces the values of some synthesized attributes, with the restriction that synthesized attributes can only depend on the inherited attributes from visits up to the visit they are computed in. Although the class of absolutely non-circular attributes is larger than the class of linearly ordered attribute grammar, experience has shown that all practical attribute grammars fall in the latter class. There exist many algorithms for automatically finding such a linear order from the attribute grammar definition.

In the rest of the paper we talk about linearly ordered attribute grammars when we mention attribute grammars, and the techniques described here work for linearly ordered attribute grammars with an arbitrary number of visits. However, because our example contains only a single inherited and a single synthesized attribute, we can simplify some details in the explanation.

The generated function evaluating the attributes for the Lam production of our example is:

```
semLamLam :: String → (Bool → String)
                    → Bool → String
semLamLam _x _e _lhsIneedp =
  let _eOneedp   :: Bool
      _eOneedp   = False
      _eIpp      :: String
      _eIpp      = _e _eOneedp
      _lhsOpp    :: String
      _lhsOpp    = pParens _lhsIneedp $
                     "\\" ++ _x ++ "." ++ _eIpp
  in  _lhsOpp
```

Note that the argument for the child is not the type of the child, but the type of the *evaluator* of the child, which can be invoked to compute the attributes for the child. At top level we construct the complete evaluator with the function $semLam$.

```
semLam :: Lam → Bool → String
semLam (LamVar  x)    = semLamVar  x
semLam (LamLam x  e)  = semLamLam x (semLam e)
semLam (LamApp  e1 e2) = semLamApp (semLam e1)
                                   (semLam e2)
```

For the other nonterminals and productions similar functions are generated.

3. High-level overview

Attribute grammar evaluators can be thought of as tree-walk evaluators '*walking*' up and down the AST. A visit may occur when the values of some further inherited attributes of a node have become available and the evaluator is guaranteed to be able to compute the values of some further synthesized attributes of that node.

One of the steps the evaluator can perform is to directly compute a synthesized attribute, for example when its value is constant or it depends only on available inherited attributes. In many cases other attributes are needed for which the evaluator should first compute some inherited attributes of children of the node, and then recursively visit some of the children before being able to compute some synthesized attributes and returning to the calling node. So, in the overall evaluation of the attributes of the top level node, the evaluator can be thought as walking up and down the tree, possibly visiting subtrees multiple times (multiple visits), before finally returning all synthesized attributes at the top level. In the context of a compiler this is typically a representation of the executable that is generated.

Now when the AST changes, for example in a compiler due to a change in the input source code, we would like to efficiently recompute the changed attribute values. One approach to incremental computation is to use *change propagation* to propagate the changed values through the tree and only evaluate expressions that have changed inputs (Reps et al. 1983). In other words, the evaluator only visits nodes in which something changed, either due to a direct change to that subtree or to a change in the attribute values. However, in many cases there are large parts of the AST in which nothing has changed, so the evaluator needs to perform much fewer steps to compute a new consistent attributed tree.

Implementation wise there are several aspects that need to be addressed. First of all the identification of changes to the AST. In this work we describe how to represent such changes using paths to locations in the AST and inserted values, assume values of this data type are constructed by some external process like a diff tool or a structure editor. Given such a value the incremental evaluation is performed by invalidating the visit results on the path from the root of the AST to the changed subtree. For every visit to a node that has already been evaluated before, it is checked whether the inherited attributes are changed and whether the visit was invalidated. If both are not the case the previous result is returned and thus the whole subtree does not need to be visited for that visit.

To implement this all in a strongly-typed fashion in Haskell we use several advanced Haskell features. For the representation of a node we use the *record syntax*, which produces a standard data type with constructors with named getters and setters for its fields. To carry around type information about the type of children we use *Generalized Algebraic Data Types* (Cheney and Hinze 2003; Xi et al. 2003). In order to associate derived types to their original types we use *type families*. Finally, we use higher-ranked types to pass generic functions as arguments to higher-order functions. Note that our implementation uses only pure functions, even though we like to think about updating the internal state of nodes as a side-effect.

4. Representation of changes

When talking about incremental evaluation in the context of attribute grammars, we assume that our input AST changes slowly over time, and we hope to efficiently recompute the attribute values after a change. In this paper we start from a description of the change to the AST produced by some external tool, for example an structure editor or some diff algorithm keeping track of changes to the source code files.

In this section we describe how to represent these changes, which is described in (Bransen and Magalhães 2013) in more detail. The representation that we describe here is not only used as an external value to alter the original AST, but also inside the evaluation to keep track of changes to constructed values, as described in Section 6.

Informally, we represent a change to the AST by a path describing the location and a new tree that is to be inserted in that location. However, instead of always replacing the full subtree, we allow the newly inserted tree to refer in some places to values of the original tree, thereby reusing existing parts. In that way an insertion of a value v into a list can for example be modelled by a path p and a $Cons\ v\ (Ref\ p)$, indicating that the value at location p should be

replaced by the $Cons$ constructor with a value v and as its tail the value that was originally at location p.

Paths Because ASTs usually consist of a family of mutually recursive data types, we need to carry around some type information in the paths. We therefore use the generalized algebraic data type $Path\ f\ t$ to represent a path in a tree of type f pointing to a node of type t. The constructors of this type are the End constructor for the empty path, and a constructor for each nonterminal child for each production. For our example this leads to:

```
data Path f t where
  End       :: Path f f
  LamPLam   :: Path Lam t → Path Lam t
  LamPAppL  :: Path Lam t → Path Lam t
  LamPAppR  :: Path Lam t → Path Lam t
  TopLamP   :: Path Lam t → Path TopLam t
```

Replacement The values that can be inserted into the tree to replace a subtree, are similar to the the original data for that nonterminal. However, we extend it with a constructor that represents re-usage of existing values, by means of a path. Because this path is relative to the top of the tree which depends on the context, we parametrize over the type of the top level. For our Lam nonterminal we can therefore represent the replacement values as follows.

```
data TopLamR top = TopLamRTop (LamR top)
                 | TopLamR (Path top TopLam)
data LamR top = LamRVar String
              | LamRLam String (LamR top)
              | LamRApp (LamR top) (LamR top)
              | LamR (Path top Lam)
```

Full change Using the paths and replacement values, we can represent a change by a pair of those values. However, as the type of the replacement depends on the type of the node that the path points to, we can not directly specify it as a pair. Instead, we use a *type family* to map each nonterminal type to its corresponding replacement type as follows.

```
type family    ReplType a       :: * → *
type instance  ReplType TopLam = TopLamR
type instance  ReplType Lam    = LamR
```

Using this type we can finally represent changes, which are also parametrized over the type of the top level node.

```
type Change top t = (Path top t, ReplType t top)
```

5. Incremental evaluation

In this section we describe how to write attribute evaluators that can efficiently respond to changes in the AST. Although this section forms an important part of the final solution, the techniques described here do not work as expected for higher order attributes. We postpone the discussion on higher order attributes to Section 6.

The basic idea of our incremental evaluation technique is quite simple: we store the previous input and output of a visit, and whenever the inputs are unchanged the previous output is returned without recomputation. Because visit computations can invoke visits of the children, this can lead to superlinear speedups. However, because the AST can change, we also need to keep track of changes to child nodes for deciding when to recompute, complicating matters a bit.

5.1 Representation

In order to store all information we create a data type for evaluators with a constructor for each production. For the Lam production of the example this data type, written using Haskell record notation, looks as follows.

```
data TLam top = TLamVar {... } | TLamLam {
  tlam_eval      :: ∀ t.TLam top → Path Lam t
                    → SemType t top,
  tlam_change    :: ∀ r.TLam top
                    → (∀ t.Path top t → SemType t top)
                    → Path Lam r
                    → ReplType r top → TLam top,
  tlam_v0        :: TLam top → Bool
                    → (String, TLam top),
  tlam_v0_dirty  :: Bool,
  tlam_e         :: TLam top
  } | TLamApp {... }
```

This data type contains a record for each production representing the state of the evaluator for that node, and has the following fields:

- An $eval$ function to retrieve the evaluator for the node at a given location, which is used when a subtree is inserted in a different part of the AST.
- A $change$ function for pushing a change to the current subtree.
- A vX function for each visit X.
- A $dirty$ flag for each visit function indicating whether or not that visit is *dirty*, i.e. some state changed since last evaluation and the visit should be re-evaluated because it may return a different result.
- A field for the state of the evaluator of each child. These are the only fields that can differ for different productions of the same nonterminal.

The type $SemType$ appearing in the type of the $change$ function is again a type family mapping the nonterminal types to the corresponding evaluator types.

```
type family    SemType t       :: * → *
type instance  SemType TopLam = TTopLam
type instance  SemType Lam    = TLam
```

It is important to notice that $change$ and all visit functions take the current state of the evaluator as argument and return the new state of the evaluator, because the state can be updated. We use this pattern because we work in a purely functional language without global state, which means that all state of an evaluator for a subtree is stored inside the evaluator type of that subtree. The advantage of this is that we do not need any global cache purging strategies but just rely on Haskell's garbage collection for cleaning up unused values, and that whenever a subtree is duplicated and used in multiple places, the states of those subtrees are not shared but diverge from the point when they were split up.

In the next section we introduce the implementation of all these functions and explain how they are used, by showing the implementation of the example.

5.2 Implementation

The semantic functions return an instance of such data type, in which the computation is "remembered". Let us illustrate this with the Lam constructor of the example. The basic function is implemented as follows, with $eval$, $change$ and v_0 bound in the where-clause:

```
semLamLam :: String → TLam top → TLam top
semLamLam x_ e_ = TLamLam {
  tlam_v0      = v0,
  tlam_eval    = eval,
```

```
  tlam_change   = change,
  tlam_v0_dirty = True,
  tlam_e        = e_
  } where
    ...
```

The actual visit code is implemented as follows, where each visit takes the state of the children and the inherited attributes, and returns the synthesized attributes and the new state of the children. These functions are then wrapped to support incremental evaluation in case nothing has changed.

```
realv0 :: TLam top → Bool → (String, TLam top)
realv0 e0 _lhsIneedp = (_lhsOpp, e1) where
  _eOneedp  = False
  (_eIpp, e1) = tlam_v0 e0 e0 _eOneedp
  _lhsOpp   = pParens _lhsIneedp $
                "\\" ++ x_ ++ "." ++ _eIpp
```

There are several things to notice here. The evaluation order is made explicit in the source code, so computations only depend on values that were defined in earlier bindings. For the computation of the pp value of the child e the visit v_0 of the child is invoked, taking the current state of the child as argument (next to the first occurrence of e_0 used to retrieve the visit function from that current state) and returning the new state of the child. Finally, together with the pp value for the current node the new state of the child is returned.

The wrapping code for a visit is then as follows. The visit is performed as usual by calling the $realv_0$ function. However, then the visit function in the evaluator is replaced by a memoizing version that directly returns the synthesized attributes in case nothing has changed.

```
v0 :: TLam top → Bool → (String, TLam top)
v0 cur inh = (syn, res) where
  (syn, e') = realv0 (tlam_e cur) inh
  res = update $ cur {
    tlam_v0        = memv0,
    tlam_v0_dirty = False,
    tlam_e         = e'}
  memv0 cur' inh' = if    inh ≡ inh'
                          ∧ ¬ (tlam_v0_dirty cur')
                    then (syn, cur')
                    else  v0 cur' inh'
```

The $update$ function is a helper function that is used to update the $dirty$ flags after some evaluation has happened. This is where the static dependency graph is represented.

```
update cur = cur {
  tlam_v0_dirty = tlam_v0_dirty cur
                  ∨ tlam_v0_dirty (tlam_e cur)
}
```

To get the evaluator residing at a given path the eval function is used. This function is implemented by simply propagating the request to the given path and then returning the evaluator. Note that on the type level the target type t of the path is already present, so at the end of the path we can return the current evaluator since the End constructor is the witness to the fact that $t{\sim}Lam$ in this case.

```
eval :: TLam top → Path Lam t → SemType t top
eval cur End            = cur
eval cur (LamPLam p) = tlam_eval (tlam_e cur)
                                 (tlam_e cur) p
```

The change function is used to propagate a change to the evaluator. When the current evaluator is changed we replace the full evaluator with the new one, and otherwise we propagate the change to the corresponding child. After propagating the type we update the $dirty$ flags.

```
change :: TLam top
          → (∀ t.Path top t → SemType t top)
          → Path Lam r
          → ReplType r top
          → TLam top
change cur lu End          repl = semLamR lu repl
change cur lu (LamPLam p) repl = update_e p $
  cur {tlam_e = tlam_change (tlam_e cur)
                  (tlam_e cur) lu p repl}
```

The updating of the dirty flags is slightly less trivial; one could think that we need to invalidate all visits in which the child e is used because somewhere in that subtree something has definitely changed. However, it could be the case that no information from that changed node is ever used. Therefore, we adopt the following strategy: whenever the child of a node is replaced all visits in which that child is used are invalidated, and otherwise we use the $update$ function to propagate changes. This is implemented as follows.

```
update_e End cur = cur {tlam_v0_dirty = True}
update_e _   cur = update cur
```

Finally, for the changed child the new evaluators should be constructed or reused. This is done using the following function which is very similar to the $semLam$ function except that is takes a lookup function as first argument to retrieve the evaluators for the reused nodes.

```
semLamR :: (∀ t.Path top t → SemType t top)
           → LamR top
           → TLam top
semLamR lu (LamR p)        = lu p
semLamR _  (LamRVar x)     = semLamVar x
semLamR lu (LamRLam x e)   = semLamLam x
                               (semLamR lu e)
semLamR lu (LamRApp e1 e2) = semLamApp
                               (semLamR lu e1)
                               (semLamR lu e2)
```

5.3 Example invocation

The example lambda term as shown in Section 2.1 is represented as follows.

```
term = LamLam "f" (LamLam "x" (LamApp
         (LamLam "y" (LamVar "y"))
         (LamApp
           (LamApp (LamVar "f") (LamVar "x"))
           (LamVar "x"))))
```

To perform the initial evaluation of the attributes the semantic wrapper function needs to be invoked. This returns the evaluator for which the top level visit can be invoked to retrieve the result and the new state of the evaluator.

```
st1 = semTopLam (TopLamTop term)
(str, st2) = ttoplam_v0 st1 st1
```

The value of str is `\f.\x.(\y.y) ((f x) x)` as expected.

Let us imagine we would like to add a lambda abstraction to the term to let it be `\f.\x.\y.(\y.y) ((f x) x)`. We represent that change by a path and a replacement value. The path needs to point to the subterm under the outermost two lambda abstractions and can thus be represented as follows.

```
path = TopLamP (LamPLam (LamPLam End))
```

The replacement needs to be of type $LamR$ and is a lambda abstraction. However, its body is the value which was at the location of $path$, so we represent the inserted value by the following.

$$repl = LamRLam\ \texttt{"y"}\ (LamR\ path)$$

To push this change to our evaluator we call the $change$ function as follows.

$$st_3 = ttoplam\_change\ st_2\ st_2\ (ttoplam\_eval\ st_2\ st_2)\ path\ repl$$

Finally, we can retrieve the result again by calling the top level visit function.

$$(str_2, st_4) = ttoplam\_v_0\ st_3\ st_3$$

The result is that str_2 has the desired value, but for computing this value the pretty printing result of the shared subtree has been reused.

5.4 Intra-visit attributes

One difficulty that does not occur in the running example of this paper is that of so called *intra-visit* attributes. In linearly ordered attribute grammars the computation of the synthesized attributes may depend on inherited attributes of that visit or earlier visits. However, with the implementation that we propose the inherited attributes of previous visits are not in scope and need to be explicitly passed to the visit in which the inherited attribute is used.

For the standard non-incremental evaluation the *visit-tree* approach (Saraiva et al. 2000) is used by the UUAGC. There it is statically computed which attributes from the first visit are used in subsequent visits, and these are passed as extra arguments to the second visit. For the second visit the attributes used in later visits are passed on to the third visit, and so on.

However, in our implementation we can do better. Since we are already explicitly encoding the current state of a node, the attributes used in later stages can be stored inside this record. Whenever such attribute value is updated due to the recomputation of the visit in which the attribute was declared, the visits in which the attribute is used can be invalidated by setting the *dirty* flag to $True$. The result of this is that only visits that really use the intra-visit attribute are recomputed, and no intermediate visits that only pass on the value to the subsequent visit.

6. Higher order attributes

An important extension to attribute grammars is that of higher order attribute grammars (Vogt et al. 1989). In a typical attribute grammar the attribute values themselves can be trees, and the idea of a higher order attribute grammar is to decorate those trees again with attributes. In the UUAGC this is implemented by allowing the user to give an expression, which can refer to attributes as usual, constructing a new child of the current node for which attributes are also evaluated.

The use of higher order attributes is especially useful in the context of building a compiler, where the result of one compiler phase is again a tree, over which attributes are computed in the next phase. Another use of higher order attributes is to abstract over common patterns like generation of fresh variables.

The problem with the incremental evaluation as discussed in the previous section is that whenever a change to the AST occurs, the value of the higher order attribute changes and therefore the attributes of the corresponding higher order child have to be recomputed. However, when a small change to AST happens, we also expect only a small change the higher order child, and we want to have the same incremental speedups for changes to the higher order child as we have in the case of a change to the AST. In this section we extend the example with a higher order child and show how our technique can be extended to obtain the desired behaviour.

6.1 Extended example

We now extend the running example in the following way. Imagine that our input AST is not a term in the lambda calculus as discussed previously, but in a language that also contains let-bindings. The AST of such language is represented as follows.

```
data Sug
  | Var  x  :: String
  | Lam  x  :: String
         e  :: Sug
  | App  e1 :: Sug
         e2 :: Sug
  | Let  x  :: String
         e1 :: Sug
         e2 :: Sug
```

What usually happens in a compiler is *desugaring*, by rewriting the AST with a rich syntax to an AST containing only simpler constructions. In this case we desugar the Sug data type to the Lam data type by translation an expression like $\mathbf{let}\ x = y\ \mathbf{in}\ z$ to the expression $(\lambda x.z)\ y$. In UUAGC syntax this can be written as follows.

```
attr Sug
  syn desug :: Lam
sem Sug
  | Var  lhs.desug = LamVar  @x
  | Lam  lhs.desug = LamLam @x @e.desug
  | App  lhs.desug = LamApp  @e1.desug @e2.desug
  | Let  lhs.desug = LamApp
                       (LamLam @x @e2.desug)
                       @e1.desug
```

At top level, for which we have also added a $TopSug$ nonterminal as convenient wrapper, we now need to *instantiate* the result of the desugaring as new child. We can then get the result of the pretty printing by simply referring to the pp attribute of the higher order child. For this we use the **inst** syntax that defines the higher order child and instantiates it.

```
attr TopSug
  syn pp :: String
sem TopSug
  | Top inst.des :: TopLam
        inst.des =  TopLamTop @e.desug
        lhs.pp   =  @des.pp
```

As before, the last rule could be omitted because it is a copy rule that can be automatically generated by the UUAGC, but for clarity we have included it here.

Of course one could also get the same behaviour by directly returning a pretty printed version of the Sug language by pretty printing the let-case in the correct way, but that leads to code duplication and is therefore undesired.

6.2 Improved evaluation

The instantiation of the higher order child is evaluated by constructing the value of the higher order child and then calling the $semTopLam$ on it as follows.

```
semTopSugTop = ... where
  ...
  realv0 :: TSug top -> (String, TSug top)
  realv0 e0 = (_lhsOpp, e1) where
```

$$
\begin{aligned}
(\_eIdesug, e_1) &= tsug\_v_0\ e_0\ e_0\\
des\_val\_ &= TopLamTop\ \_eIdesug\\
des\_inst\_ &= semTopLam\ des\_val\_\\
(\_lhsOpp, \_) &= ttoplam\_v_0\ des\_inst\_\ des\_inst\_
\end{aligned}
$$

However, although $des\_inst\_$ itself is also an AST that implements the incremental behaviour, we do not represent changes to that AST and construct a new one every time a (small) change happens to the $TSug$ AST.

To retain incremental behaviour for higher order attributes, we add a new synthesized attribute to $TSug$ that is computed in in a similar fashion as the $desug$ attributes. Such attribute is a representation of the change to the higher order attribute compared to the previous evaluation which we call *derived change*. Whenever a change happens to the $TSug$ AST and a memoized value is used for the $desug$ attribute, a reference is used as value for the derived change.

In order to fill the path in the references of this derived change, we need information about the location where the current value of the tree ends up when it is instantiated. To propagate that information we also add an inherited attribute to $TSug$ containing that information. This path can then be stored when a reference is returned.

To implement this, the type of $tsug\_v_0$ is changed to the following.

$$
\begin{aligned}
&tsug\_v_0 :: TSug\ top\\
&\qquad \rightarrow (\forall\ t.Path\ Lam\ t \rightarrow Path\ TopLam\ t)\\
&\qquad \rightarrow (Lam, LamR\ TopLam, TSug\ top),
\end{aligned}
$$

The second argument is the path where the Lam value ends up when it is instantiated, but instead of (inefficiently) appending to the end of a path, we use the trick of *difference lists*. As a synthesized attribute the $LamR\ TopLam$ is added, which represents the change to the Lam relative to the previous evaluation, in some top level structure $TopLam$ (when the higher order child is instantiated).

The visit function for $semSugApp$ is then changed as follows.

$$
\begin{aligned}
&semSugApp = \ldots\ \mathbf{where}\\
&\quad \ldots\\
&\quad v_0 :: TSug\ top\\
&\qquad \rightarrow (\forall\ t.Path\ Lam\ t \rightarrow Path\ TopLam\ t)\\
&\qquad \rightarrow (Lam, LamR\ TopLam, TSug\ top)\\
&\quad v_0\ cur\ p = (syn, synr, res)\ \mathbf{where}\\
&\qquad (syn, synr, (e_1', e_2')) = realv_0\ (tsug\_e_1\ cur,\\
&\qquad\qquad tsug\_e_2\ cur)\ p\\
&\qquad res = update\ \$\ cur\ \{\\
&\qquad\quad tsug\_v_0 = memv_0,\\
&\qquad\quad tsug\_v_0\_dirty = False,\\
&\qquad\quad tsug\_e_1 = e_1',\\
&\qquad\quad tsug\_e_2 = e_2'\}\\
&\qquad memv_0 :: TSug\ top\\
&\qquad\quad \rightarrow (\forall\ t.Path\ Lam\ t \rightarrow Path\ TopLam\ t)\\
&\qquad\quad \rightarrow (Lam, LamR\ TopLam, TSug\ top)\\
&\qquad memv_0\ cur'\ p' = \mathbf{if}\quad \neg\ (tsug\_v_0\_dirty\ cur')\\
&\qquad\qquad\qquad\quad \mathbf{then}\ (syn, LamR\ (p\ End), cur')\\
&\qquad\qquad\qquad\quad \mathbf{else}\ \ v_0\ cur'\ p'
\end{aligned}
$$

Note the in $memv_0$ the $synr$ attribute is replaced by a reference to the path p, which is completed by passing the End constructor. Due to the smart use of a higher-ranked type together with GADTs and type families this is all strongly typed and also works for families of mutually recursive data types.

In the $realv_0$ function of $semSugApp$ the paths are altered by function composition to construct the correct path for each of the children. Note that these paths do not correspond to constructors of the $TSug$ AST over which these attributes are computed, but are relative to the $desug$ attribute in which the different parts end up.

$$
\begin{aligned}
&realv_0 :: (TSug\ top, TSug\ top)\\
&\qquad \rightarrow (\forall\ t.Path\ Lam\ t \rightarrow Path\ TopLam\ t)\\
&\qquad \rightarrow (Lam, LamR\ TopLam, TSug\ top, TSug\ top)\\
&realv_0\ (e_{10}, e_{20})\ p = (\_lhsOdesug, \_lhsOdesugR, e_{11}, e_{21})\\
&\quad \mathbf{where}\\
&\qquad (\_e_1 Idesug, \_e_1 IdesugR, e_{11}) = tsug\_v_0\ e_{10}\ e_{10}\\
&\qquad\qquad\qquad\qquad\qquad\qquad (p \circ LamPAppL)\\
&\qquad (\_e_2 Idesug, \_e_2 IdesugR, e_{21}) = tsug\_v_0\ e_{20}\ e_{20}\\
&\qquad\qquad\qquad\qquad\qquad\qquad (p \circ LamPAppR)\\
&\qquad \_lhsOdesug = LamApp\ \ \_e_1 Idesug\ \ \_e_2 Idesug\\
&\qquad \_lhsOdesugR = LamRApp\ \_e_1 IdesugR\ \_e_2 IdesugR
\end{aligned}
$$

Finally, at top level the higher order child is instantiated only the first time. The field des_0 is of type $Maybe\ (TLam\ top)$ and is initially set to $Nothing$. In subsequent evaluations in $semTopSugTop$ that field is used for updating the existing evaluator as follows.

$$
\begin{aligned}
&realv_0 :: TSug\ top \rightarrow Maybe\ (TTopLam\ TopLam)\\
&\qquad \rightarrow (String, TSug\ top, TTopLam\ TopLam)\\
&realv_0\ e_0\ des_0 = (\_lhsOpp, e_1, des_1)\ \mathbf{where}\\
&\quad (\_eIdesug, \_eIdesugR, e_1) = tsug\_v_0\ e_0\ e_0\ TopLamP\\
&\quad des\_val\_ = TopLamTop\ \ \_eIdesug\\
&\quad des\_valR\_ = TopLamRTop\ \_eIdesugR\\
&\quad des\_inst\_ = \mathbf{case}\ des_0\ \mathbf{of}\\
&\qquad Nothing \rightarrow semTopLam\ des\_val\_\\
&\qquad Just\ v \ \ \rightarrow ttoplam\_change\ v\ v\ (ttoplam\_eval\ v\ v)\\
&\qquad\qquad\qquad End\ des\_valR\_\\
&\quad (\_lhsOpp, des_1) = ttoplam\_v_0\ des\_inst\_\ des\_inst\_
\end{aligned}
$$

With this transformation in place, we now have restored the incremental behaviour of our code when higher order attributes are used.

7. Evaluation

We have implemented the techniques described into a simple compiler[2]. To simplify the implementation the compiler takes a linearly ordered attribute grammar as input and generates the code as described in the previous sections.

To evaluate the effectiveness of our approach we have implemented a constraint-based type inference algorithm (Heeren et al. 2002) for the lambda calculus with let bindings and let polymorphism. Furthermore, we have added a desugaring step to the algorithm as with the running example in this paper and used a higher order attribute to infer types for the desugared version of the AST. This is exactly the case where our previous work falls short because of the use of the higher order attributes. Furthermore, it is a typical use case in compiler construction with attribute grammars.

7.1 Implementation details

For the inference of the types we use bottom-up type rules as shown in Figure 1 which gather constraints used elsewhere by a constraint solver. Type judgements are of the form $\mathcal{M}, \mathcal{A}, \mathcal{C} \vdash e{:}\tau$ where e is the expression being typed, τ is the type, $\mathcal{M}$ is a set of monomorphic type variables, $\mathcal{A}$ is a set of assumptions and $\mathcal{C}$ is a list of constraints. It is important to notice that in the corresponding implementation $\mathcal{M}$ is a value that is passed top-down and $\mathcal{A}$, $\mathcal{C}$ and τ are passed bottom-up. In contrast to the well-known algorithm $\mathcal{W}$ (Damas and Milner 1982), the bottom-up constraint based type inference algorithm has a bottom-up behaviour that makes it suitable for incremental evaluation.

[2] The code can be found at:
`http://www.staff.science.uu.nl/~brans106/iehoa.zip`

$$\frac{\beta \text{ fresh}}{\mathcal{M}, \{x : \beta\}, [\,] \vdash x : \beta}\ \textsc{Var}$$

$$\frac{\begin{array}{c}\beta \text{ fresh} \\ \mathcal{M}, \mathcal{A}_1, \mathcal{C}_1 \vdash e_1 : \tau_1 \\ \mathcal{M}, \mathcal{A}_2, \mathcal{C}_2 \vdash e_2 : \tau_2 \\ \mathcal{C}_{new} = [\tau_1 \equiv \tau_2 \to \beta]\end{array}}{\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 +\!\!+ \mathcal{C}_2 +\!\!+ \mathcal{C}_{new} \vdash e_1\ e_2 : \beta}\ \textsc{App}$$

$$\frac{\begin{array}{c}\beta_1, \beta_2 \text{ fresh} \\ \mathcal{M} \cup \{\beta_1\}, \mathcal{A}, \mathcal{C} \vdash e : \tau \\ \mathcal{C}_{new} = [\beta_2 \equiv \beta_1 \to \tau] +\!\!+ [\beta_1 \equiv \tau' \mid x : \tau' \in \mathcal{A}]\end{array}}{\mathcal{M}, \mathcal{A} \backslash x, \mathcal{C}_{new} +\!\!+ \mathcal{C} \vdash \lambda x \to e : \beta_2}\ \textsc{Abs}$$

$$\frac{\begin{array}{c}\beta \text{ fresh} \\ \mathcal{M}, \mathcal{A}_1, \mathcal{C}_1 \vdash e_1 : \tau_1 \\ \mathcal{M}, \mathcal{A}_2, \mathcal{C}_2 \vdash e_2 : \tau_2 \\ \mathcal{C}_{new} = [\beta \equiv \tau_2] +\!\!+ [\tau' \leqslant_{\mathcal{M}} \tau_1 \mid x : \tau' \in \mathcal{A}_2]\end{array}}{\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2 \backslash x, \mathcal{C}_1 +\!\!+ \mathcal{C}_{new} +\!\!+ \mathcal{C}_2 \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \beta}\ \textsc{Let}$$

Figure 1. Bottom-up type rules

The type rules can be translated to AGs in an almost straightforward way. We use one inherited attribute for $\mathcal{M}$, and three synthesized attributes corresponding to $\mathcal{A}$, $\mathcal{C}$ and τ. There are however two implementation details that do not follow directly from the type rules.

Fresh variable generation The generation of fresh variables is usually implemented in AGs using a *threaded* attribute, which is both synthesized and inherited. Such an attribute is a simple counter which is increased every time a fresh variable is needed. As described in our previous work (Bransen et al. 2014), such threaded attributes are bad for the effectiveness of the incremental evaluation. In previous work we have suggested several solutions for this, which could also be applied here, but to simplify the toy implementation we have chosen to implement fresh variable generation with a global mutable state instead.

Intermediate constraint solving In the constraint based type inference most time is spent in solving the constraints, not constructing the constraints. The incremental AG machinery is used for making the constraint generation phase more efficient after changes in the AST. With constraint solving being done only at top level we still spend most time there, since the constraint solving is completely redone even after a simple change.

However, many of the generated constraints are equivalent to the constraints generated earlier, so the result of the constraint solving should also be stored. There is a certain order in which constraints should be solved, so it is not possible to solve all constraints as soon as they are generated. For a closed expression, in which no free variables appear, it is possible to solve all constraints and immediately apply all resulting substitutions. In the constraint generation the expression is closed if and only if $\mathcal{A}$ is empty. Based on this observation we have defined our code in such way that whenever $\mathcal{A}$ is empty, all constraints generated so far are solved and the substitutions are applied to τ; no more constraints remain.

7.2 Benchmarking

At first we have run the code on some hand-crafted cases to validate correctness and do preliminary measures. We have measured large speedups (100x) in specific artificial cases, and overall the overhead seemed within acceptable bounds.

To more thoroughly evaluate the effectiveness and measure the overhead of our approach we have run several benchmarks. For the time measurement we used *Criterion* (O'Sullivan 2009) which is a framework for measuring the performance of Haskell programs. It takes care of running benchmarks multiple times for more accurate results, forcing evaluation of the benchmark results, avoiding undesired sharing between runs and generating statistics. In our case we have directed Criterion to use 100 runs for each benchmark.

In order to generate arbitrary sugared lambda expressions and changes to the AST we used *QuickCheck* (Claessen and Hughes 2000) which is a tool for formulating and testing properties of Haskell programs. As a part of this tool there is a set of functions for generating arbitrary instances of data types, which we use to generate the data for our benchmarks.

7.3 Results

In Figure 2 we show the benchmark results. In each sub figure two bar charts are shown: *Base* is the standard evaluation as described in Section 2 and *Incr* is the incremental evaluation as described in Section 6. For all different test cases we have used a set of 100 randomly generated well-typed lambda terms, and five different types of changes for each of them. The unit of measure on the y-axis is in milliseconds, and notice that that last two sub figures have a y-axis spanning twice as much time.

The initial run for both approaches is shown in Figure 2a. The overhead of the incremental evaluation is a 9.3% time increase on average. Figure 2b and Figure 2c show are the corner cases in which the expression is only changed at top level by adding an unused let binding or application of the identity function respectively. Even though it is to be expected that in these cases the code would benefit maximally from incremental evaluation, the speedups are only 28.2% and 32.3%.

Two cases where we would expect speedups but have measured decrease in runtime are the deletion of an arbitrary subtree (Figure 2d) and doing an arbitrary (valid) change (Figure 2e). The increase in runtime is 8.9% and 15.5% respectively. Finally in Figure 2f we show the worst case for incremental computation in which the full AST is replaced by another one, such that no information can be reused. Here the overhead is also 15.5%, which we believe is reasonable if large speedups would be achieved on other cases.

8. Discussion

Benchmarks We have been able to achieve large speedups (100x) in hand-crafted test cases, but we have not measured any speedups in the larger benchmarks. One of the reasons is that the generation of arbitrary well-typed lambda terms is not easy, and our solution is ad-hoc. As a result of this, the types of the generated lambda terms are quite large and evaluating those could take up large parts of the overall computation time.

To improve upon this better benchmarks need to be constructed, for example by generating the lambda terms in a uniformly distributed way (Claessen et al. 2014). Furthermore, the types of changes presented in our benchmarks should be closer to real use, for example by taking data from a structure editor or the revision history of some project.

Overhead One common problem of incremental evaluation is that there is always overhead involved. In our case the initial evaluation is slower because of the extra state that is built up next to the actual evaluation, and in the case of the example the overhead can actually be larger than the actual computation. It is therefore not desirable to apply this technique to all attribute grammars.

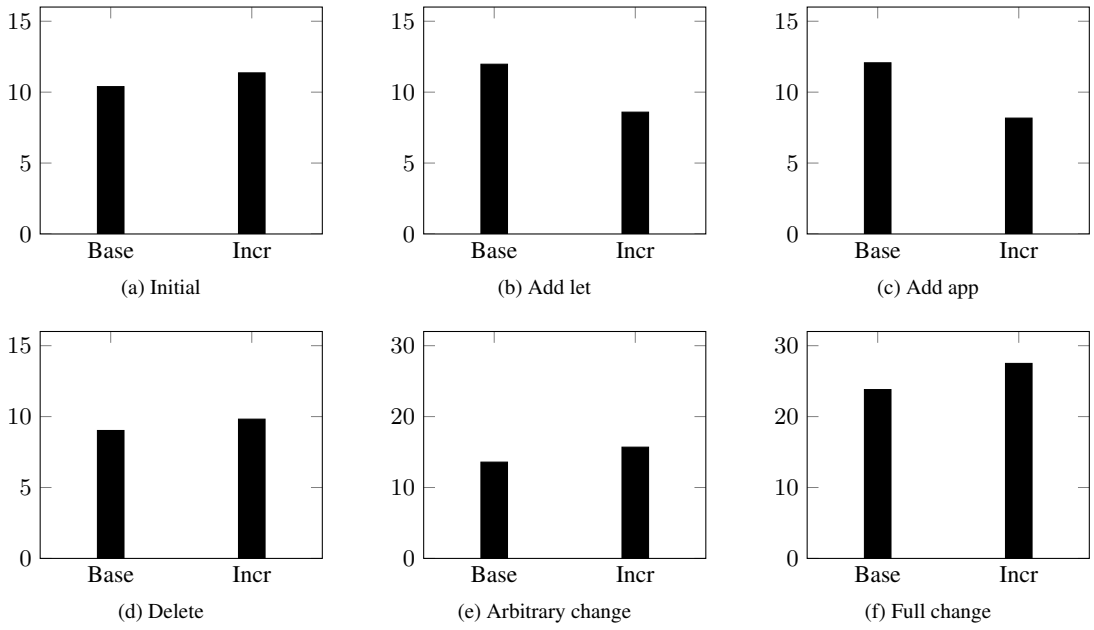

Figure 2. Benchmark results of constraint based type inference

A possible solution for this problem is to rely on Haskell's lazy evaluation mechanism to perform the extra work to achieve incremental speedups at the moment that the incremental step happens, resulting in practically no extra runtime in the first evaluation. In our previous work we found promising results in that direction, but in order to support higher order attributes we need to do more work and evaluate some part of the state in a slightly stricter way.

The reason our current approach does not perform so well on our benchmark is probably because it is too fine-grained. For the type inference example in many nodes only some constraints are gathered, and only in some nodes the constraints are solved. With our incremental evaluation of attributes we check for equality of inherited nodes and propagate the dirty flags for every node, which may take more time than the actual constraint gathering. For our technique to be usable in practice we therefore intend to run the checks only in certain nodes, for example only on nodes where constraints are solved. Although this leads to more actual attribute evaluation, it can be much faster in practice.

Non-observable construction An important limitation of our technique is that it requires knowledge of the construction of the higher order attribute. In particular, when a higher order attribute a is used to construct a new value of a higher order attribute, a path needs to be constructed to indicate in which part of the new AST a ends up. However, because our expressions can be arbitrary Haskell, we can write attribute grammars for which we can not (automatically) find out this information.

In order for our technique to work we therefore require the construction of higher order attributes to be of a restricted form in which only constructors, attribute references, and constants are used. In particular, pattern matching is not allowed as it would highly complicate dependency analysis.

However, in practice there are of course cases where more complicated code is written to construct the higher order tree. Another possibility is to mix observable and non-observable construction of higher order attributes and only get good incremental behaviour when enough attributes can be observed. A concrete example of this could for example be type inference, where the AST describes expressions of the untyped lambda calculus and the result is a term in the typed lambda calculus. In that case, the spine of the result, so the expression itself, can be constructed in an observable way for which incremental behaviour is retained, but the types themselves are considered *black boxes* for which all information is lost after they are reconstructed.

9. Related work

Dynamic dependencies One of the recent developments in incremental computing is that of so-called self-adjusting computation (Chen et al. 2014). In self-adjusting computation all data is labelled with either static or changeable, and special constructs are added to the language in order to handle changeable data in such a way that changes can be efficiently propagated. Based on the types, these special constructs can even be automatically inferred by the compiler.

The important difference to our approach is that in self-adjusting computation all dependencies are gathered dynamically in the first run. While evaluating in the first run a dependency graph is built, and that graph is used in subsequent runs to propagate the changes. Although in essence our technique is doing the same thing, in the attribute grammar world all dependencies are known statically and can therefore be used to generate incremental evaluators that have no runtime dependency tracking overhead.

Function caching Another technique for the incremental evaluation of higher order attribute grammars is to use function caching or memoization (Vogt et al. 1991; Saraiva et al. 2000). Because the subtree and inherited attributes together are used as key for the memoization table, this technique also works for higher order attributes. However, the problem is that such technique requires a global cache which needs to be purged to avoid running out of memory, since otherwise the cache could infinitely grow. However, no purging strategy can perfectly predict future calls to the function, so it can result both in recomputations due to wrong cache items being purged and in cache items that are stored and take up memory but are never used.

In our approach the cache is (implicitly) stored inside the AST, such that the subtree itself is not a parameter of the cache lookup. This does not only make sure that our caches stores exactly the values that are needed, but also can result in faster lookups. In our approach we do however only store one cache item per visit per node, with the immediate previous values, so when a tree is changed to another tree and then is changed back, some recomputation can happen.

Incremental higher order evaluation The work of (Cai et al. 2014) discusses automatically generating incremental evaluators for higher-order languages, by statically constructing derivatives of functions that can handle changes efficiently. However, to get actual incremental speedups the work assumes the existence of certain user-defined change structures that specify for all base-types how changes can be constructed and represented.

Our support for higher order attributes is similar to the construction of derivatives for higher order functions. However, in our case the basic incremental attribute grammar forms the basic change structure, and therefore there is no need for the user to specify any extra information to get incremental speedups. Our work on the other hand only works for the class of attribute grammars, which is of course much more limited than higher order languages in general.

Profiling based caching (Söderberg and Hedin 2011) describe the incremental evaluation of reference attribute grammars based on caching in an imperative setting. However, to improve caching behaviour they use a selective caching mechanism based on profiling, thereby optimising the caching to specific use cases. To avoid cases like our running example where the actual computation time is in some cases smaller than the overhead, such an approach could be viable in our setting too.

10. Conclusion

We have presented a technique for the incremental evaluation of higher order attribute grammars. Our technique is based on earlier work and uses static dependency information to generate efficient evaluators. We have created a toy implementation of our tool and verified that the resulting code is strongly typed and returns correct results. We have measured large runtime speedups on some handcrafted cases indicating that our technique can be effective, but it is future work to evaluate the techniques on a wider scale to find out in which cases attribute grammars can benefit from incremental evaluation.

References

van Binsbergen, L. T., Bransen, J., and Dijkstra, A. (2015). Linearly ordered attribute grammars - with augmenting dependency selection. In *Proceedings of the ACM SIGPLAN 2015 Workshop on Partial Evaluation and Program Manipulation*, PEPM '15, New York, NY, USA. ACM.

Bransen, J., Dijkstra, A., and Swierstra, S. D. (2014). Lazy stateless incremental evaluation machinery for attribute grammars. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 145–156, New York, NY, USA. ACM.

Bransen, J. and Magalhães, J. P. (2013). Generic representations of tree transformations. In *Proceedings of the the 9th ACM SIGPLAN Workshop on Generic Programming (WGP'13)*, WGP '13.

Cai, Y., Giarrusso, P. G., Rendel, T., and Ostermann, K. (2014). A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 145–155, New York, NY, USA. ACM.

Chen, Y., Acar, U. A., and Tangwongsan, K. (2014). Functional programming for dynamic and large data with self-adjusting computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 227–240, New York, NY, USA. ACM.

Cheney, J. and Hinze, R. (2003). First-class phantom types. Technical report, Cornell University.

Claessen, K., Duregård, J., and Pałka, M. (2014). Generating constrained random data with uniform distribution. In Codish, M. and Sumii, E., editors, *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 18–34. Springer International Publishing.

Claessen, K. and Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.*, 35(9):268–279.

Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA. ACM.

Dijkstra, A., Fokker, J., and Swierstra, S. D. (2009). The architecture of the Utrecht Haskell Compiler. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 93–104, New York, NY, USA. ACM.

Engelfriet, J. and Filè, G. (1982). Simple multi-visit attribute grammars. *Journal of computer and system sciences*, 24(3):283–314.

Hedin, G. and Magnusson, E. (2003). Jastadd: An aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58.

Heeren, B., Hage, J., and Swierstra, S. D. (2002). Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Department of Information and Computing Sciences, Utrecht University.

Knuth, D. E. (1968). Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145.

O'Sullivan, B. (2009). Criterion: Robust, reliable performance measurement and analysis. `http://hackage.haskell.org/package/criterion`.

Peyton Jones, S. L. (2003). *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press. Journal of Functional Programming Special Issue 13(1).

Reps, T., Teitelbaum, T., and Demers, A. (1983). Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5:449–477.

Saraiva, J., Swierstra, S. D., and Kuiper, M. F. (2000). Functional incremental attribute evaluation. In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 279–294, London, UK. Springer-Verlag.

Söderberg, E. and Hedin, G. (2011). Automated selective caching for reference attribute grammars. In Malloy, B., Staab, S., and van den Brand, M., editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 2–21. Springer Berlin / Heidelberg.

Swierstra, S. D., Alcocer, P. R. A., and Saraiva, J. (1998). Designing and Implementing Combinator Languages. In *Advanced Functional Programming*, pages 150–206.

Van Wyk, E., Bodin, D., Gao, J., and Krishnan, L. (2010). Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39 – 54. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

Vogt, H. H., Swierstra, S. D., and Kuiper, M. F. (1989). Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, volume 24 of *PLDI '89*, pages 131–145, New York, NY, USA. ACM.

Vogt, H. H., Swierstra, S. D., and Kuiper, M. F. (1991). Efficient incremental evaluation of higher order attribute grammars. In *PLILP*, pages 231–242.

Xi, H., Chen, C., and Chen, G. (2003). Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 224–235, New York, NY, USA. ACM.