Lazy Stateless Incremental Evaluation Machinery for Attribute Grammars

Jeroen Bransen Atze Dijkstra S. Doaitse Swierstra

Utrecht University, The Netherlands

{J.Bransen,atze,doaitse}@uu.nl

Abstract

Many computer programs work with data that changes over time. Computations done over such data usually are repeated completely after a change in the data. For complex computations such repetitive recomputation can become too inefficient. When these recomputations take place on data which has only changed slightly, it often is possible to reformulate the computation to an incremental version which reuses the result of the computation on previous data. Such a situation typically occurs in compilers and editors for structured data (like a program) where program analyses and transformations (for example error checking) are done while editing.

Although rewriting to incremental versions thus offers a solution to this problem, a manual rewrite of an already complex computation to its incremental counterpart is tedious, error prone, and inhibits further development of the original computation. We therefore intend to generate such incremental counterparts (semi)automatically by focusing on computations expressed using Attribute Grammars (AGs).

In this paper we do groundwork for this goal and develop machinery for incremental attribute grammar evaluation based on change propagation and pure functions. We use pretty printing with free variable annotation to explain our techniques. Furthermore, our techniques also expose rules of conduct for a programmer desiring incrementality: the automatic translation of code to an incremental version does not always directly result in efficiency improvements because code often is written in a style unsuitable for automatic incrementalization. We show some common cases in which (small) code changes facilitating incrementality are required. We evaluate the effectiveness of the overall approach using a simple benchmark for the example, and a more extensive benchmark based on constraint-based type inference implemented with AGs.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PEPM '14, January 20–21, 2014, San Diego, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2619-3/14/01... $15.00.
http://dx.doi.org/10.1145/2543728.2543735

1. Introduction

Computer programs that work with data slowly changing over time, like compilers in a development environment, can be reformulated into *incremental* versions that (efficiently) respond to these changes, often resulting in programs that run asymptotically faster compared to a full recomputation. However, the construction of such incremental programs can be a tedious job, so a lot of research has been done on the (semi)automatic construction of incremental computations (Ramalingam and Reps 1993).

One of the existing approaches to this problem is based on *self-adjusting computation* (Acar et al. 2006), where stable data is distinguished from changeable data and language primitives are added to work with changeable data. Because the usage of such primitives requires extensive changes to the code, techniques have been developed to automatically infer these changes based on the types (Chen et al. 2011).

A drawback of self-adjusting computations is that data dependency information is build up *dynamically*, which usually leads to runtime overhead. We therefore propose a solution based on attribute grammars (Knuth 1968), where dependency information is *statically* known. A static approach does not only avoid the runtime overhead of dependency tracking, but also allows for static analyses based on the dependency information that could improve incremental behaviour.

1.1 Attribute grammars

Attribute Grammars (AGs) are known to be suitable for the implementation of the semantics of programming languages and have been extensively studied. AGs consist of a context-free grammar specifying the Abstract Syntax Tree (AST) together with a set of attribute definitions describing how the nodes in the AST are to be decorated with attributes describing properties of that node. Semantic rules describe how attribute values can be computed out of other attributes.

AGs are isomorphic to the class of functional programs known as *catamorphisms*. AGs are of particular interest since they have advantages over functional programs, for example by providing a declarative way of programming in which parts of the program flow can be left implicit. Because AGs are a restricted form of functional programs more optimizations can be applied, for example the one that is described in this paper. Furthermore, because of their compositional nature AGs lend themselves well for implementing program analyses and transformations in an attractive aspect oriented way.

Efficient evaluators can automatically be computed from AG definitions for *ordered* AGs (Kastens 1980). In Ordered Attribute Grammars (OAGs) the dependencies between attributes are used to statically find an evaluation order such that for all possible ASTs the attribute values of the dependencies of an attribute are computed

before the attribute itself is computed. Another approach to the construction of efficient evaluators is the algorithm from (Kennedy and Warren 1976) which works better in certain cases because there are less restrictions on the AG (Bransen et al. 2012). However, we use OAGs in this paper because the linear evaluation order is useful for the incremental evaluation machinery.

A well-known application in which AGs are used to generate syntax-directed editors is the *Synthesizer Generator* (Reps and Teitelbaum 1984). In such interactive programming environments it is not only necessary to evaluate attributes of an AG once, but also to efficiently handle (usually small) changes in the AST. This can be achieved using *incremental evaluation* of AGs (Demers et al. 1981). Note that we use *AG evaluation* as a shortcut for the evaluation of the attributes of an AG.

The main motivation for our work is the Utrecht Haskell Compiler (UHC) (Dijkstra et al. 2009) which is completely built using AGs. It is our intention to use incremental AG evaluation techniques such as those described in this paper to create an incremental version of the UHC that can efficiently respond to such small changes in the input and can efficiently recompile.

Incremental AG evaluation can be done in various ways, all using some form of caching. One approach, which is also used in the Synthesizer Generator, is to store the attribute values in the AST and update them using a tree-walk evaluator after a change in the AST (Demers et al. 1981; Yeh and Kastens 1988). Another approach is taken by Vogt et al. (1991) where the attribute values are not stored in the AST itself, but are computed in visit functions for which *function caching* is used to store previously computed results. This approach is improved upon in (Saraiva et al. 2000).

All such approaches maintain global state. In the case of the approach taken in the Synthesizer Generator this state is the AST itself where the attribute values are stored in the AST. In essence, the AST itself can be seen as a cache where for each possible subtree in the AST for each attribute exactly one value is stored. In the function caching approaches the state is the memoization table, where again for each subtree a value is stored. However, in the latter case the organisation of the cache could be slightly different, for example by not only storing the result of the previous computation, but also keep the memoized value of earlier computations. The advantage is that the chance that a desired value is already in the cache is increased, but the drawback is that purging strategies must be used to avoid infinite growth of the cache.

1.2 Overview

In this paper we present a novel approach for incremental AG evaluation based on a purely functional implementation of AGs. The behaviour of our code is equivalent to that of the stateful solutions with a cache size of one value per attribute per node, but uses no global state. Furthermore, since our solution uses only pure functions we can rely on lazy evaluation to make sure that we only compute values (or retrieve them from some sort of cache) when this is necessary for computing the final result. For our implementation we use Haskell (Peyton Jones 2003).

Our approach is based on *change propagation*, which means that we assume that it is known in what way the AST changes. In the context of an interactive programming environment this is always the case, since the user interactions lead to (explicit) changes to the AST. Another example of such interactive programming environment is *Proxima* (Schrage 2004) which is based on AGs. In other applications however, for example an incremental compiler, it could be the case that a completely new AST is given of which actually a large part corresponds to the original AST. The approaches based on function caching retain their incremental behaviour in such case, but for approaches like ours an explicit representation of the changes to the AST is needed. To find such a representation we use the *diff* algorithm from (Bransen and Magalhães 2013).

All code that we present in this paper is hand-written and in some places simplified for presentation. In actual applications this code is however automatically generated from the corresponding AG definition, such that the programmer does not need to think about the underlying implementation that is described in this paper. We have implemented this in the *Utrecht University Attribute Grammar Compiler* (Swierstra et al. 1998) and the benchmark results are generated using that implementation.

The main contribution of this paper is to show how incremental AG evaluation can be achieved using a purely functional implementation of AGs. The other contributions that we make are the following.

- We describe basic AG machinery which can be seen as an AG way of functional programming.
- We describe some common patterns that can have a negative influence on the incremental behaviour of code and show solutions to these problems.
- We show some benchmarks results for a larger case using our technique which is incremental type inference based on constraint solving.

This paper is organised as follows. In Section 2 we start by briefly introducing the example that is used throughout the paper. Then in Section 3 the existing AG machinery is introduced and the necessary background information on attribute grammars is given. The main contribution of the paper is in Section 4. We describe some extra insights that can facilitate the incremental behaviour in Section 5 and evaluate the different stages of the given approach using both the example and a type inference algorithm as a benchmark in Section 6. Finally, we discuss some shortcomings and future work in Section 7, and in Section 8 we wrap up with some concluding remarks.

2. Example

To illustrate the techniques described in this paper we develop a pretty printer for the untyped lambda calculus in which free variables are annotated. This example is both simple enough to be understood without much explanation, yet complex enough to demonstrate the need for incrementality and usefulness of our techniques.

We define the untyped lambda calculus in the standard way.

$$e := x \mid e\ e \mid \lambda x \rightarrow e$$

The pretty printer should place parentheses around every application and lambda abstraction. Furthermore, each occurrence of a free variable should be annotated with a * character. As an example, a term $\lambda x \rightarrow \lambda y \rightarrow x\ y\ z$ is pretty printed as follows.

```
(\x -> (\y -> ((x y) *z)))
```

The \ is used instead of λ as is done in Haskell.

Although this example is very simple it contains the necessary ingredients to illustrate our techniques. The pretty printing is compositional: the pretty printing of a node does not depend on the result of its siblings. However, the pretty printing does depend on the context (the bound variables) so after a change to the expression a part of the pretty printing might need to be recomputed.

3. Attribute Grammar machinery

In this section we introduce attribute grammars and the existing machinery along with an implementation of the example.

3.1 Attribute grammars

The basis of an AG is a context-free grammar specifying the AST. In a Haskell setting the AST is represented by an *algebraic data type* with a constructor for each production of the context-free grammar. For each nonterminal in the right-hand side of a production there is a constructor (or child in terms of AGs) of the corresponding type in the algebraic data type.

The AST of our lambda expressions is represented in UUAGC syntax as follows.

```
data Lam
  | Var x :: {String}
  | Abs x :: {String}
        e :: Lam
  | App l :: Lam
        r :: Lam
```

This data definition is similar to the record syntax of Haskell. Furthermore, the primitive types of the underlying implementation language (*String* in this case, which is a Haskell type as Haskell is used as the back end language) are wrapped in curly braces.

In our setting the AST is constructed from the concrete syntax by means of parsing using *parser combinators* (Swierstra and Duponcheel 1996). The parsing of the concrete syntax into an AST is not important for the techniques developed in this paper and we therefore assume to have the AST available as input in the rest of this paper.

Attributes The AST is decorated with attributes which can be either *inherited*, *synthesized* or *chained*. An inherited attribute is a value passed from the root to the leaves in the AST. At every production for each inherited attribute of each child an expression must be defined, yielding a value for the attribute at runtime. This is called a semantic rule. Conversely, a synthesized attribute is composed from the leaves of the AST towards its root. For each production for each synthesized attribute of that production an expression must be defined. A chained (or threaded) attribute is a combination of an inherited and synthesized attribute.

In order to pretty print the variables contextual information about all bound variables is needed. We therefore pass a set of all bound variables down, which is an inherited attribute. This attribute is declared as follows using the keywords **attr** and **inh**.

```
attr Lam
  inh boundvars :: {Set String}
```

The result of the pretty printing is a synthesized attribute of type *String*.

```
attr Lam
  syn pp :: {String}
```

Semantic rules Semantic rules specify for each production how the attribute values are computed out of other attribute values. AG systems usually employ AG specific notation for specifying the AST, attributes and semantic rules. Additionally, AG systems, like the *Utrecht University Attribute Grammar Compiler* (UUAGC) (Swierstra et al. 1998), can support features like allowing the user to split up code fragments for aspect oriented programming and AG compile-time combination of those aspects into a single evaluator.

An attribute value can be computed when all attribute values it depends on are already computed. We assume in this paper that we have an evaluation order for the attributes such that for any possible AST the attribute values can be computed in that order. In the OAG setting such an evaluation order is inferred by the AG system at compile time and is guaranteed to exist.

For the implementation of the example, the semantic rules for *boundvars* are defined with the keyword **sem**, followed by an expression for each *Lam* child of each production (constructor). Such expression indicates how the value for the *boundvars* attribute should be computed out of other attribute values. The curly braces in this code fragment are used as a short notation for a set.

```
sem Lam
  | Abs e.boundvars = {@x} ∪ @lhs.boundvars
  | App l.boundvars = @lhs.boundvars
        r.boundvars = @lhs.boundvars
```

The @ notation is used to refer to attribute values. In this case $@x$ is a leaf in the AST representing the variable name, and $@\mathbf{lhs}.boundvars$ refers to the inherited attribute *boundvars* of the current node. In the UUAGC setting the last two rules (called copy-rules) are automatically inferred when their definition is not given.

Finally the expressions for the pretty printing are defined as follows, where *boundvars* is used to decide on the pretty printing of a variable.

```
sem Lam
  | Var lhs.pp = if @x ∈ @lhs.boundvars
                 then @x
                 else "*" ++ @x
  | Abs lhs.pp = "(\\" ++ @x ++ " -> "
                   ++ @e.pp ++ ")"
  | App lhs.pp = "(" ++ @l.pp ++ " "
                   ++ @r.pp ++ ")"
```

3.2 Evaluation machinery

We now show the implementation of the example in Haskell. The data type is a straightforward translation:

```
data Lam = Var String
         | Abs String Lam
         | App Lam    Lam
```

We use an inherited attribute of type *Set String* and a synthesized attribute of type *String*, so the evaluator resulting from each *Lam* alternative therefore has the following type.

```
type TLam = Set String → String
```

This follows the general pattern: the semantics of an AG node (including the root node) is a function from inherited to synthesized attributes.

For each production we define a function that specifies the semantics of that production, given the semantics of its children. The semantic function for variables simply is a direct translation from the semantic rules. For a more concise presentation we use the variable *bv* as name for the *boundvars* attribute.

```
semLamVar :: String → TLam
semLamVar x bv = if   x ∈ bv
                 then x
                 else "*" ++ x
```

For lambda abstraction the evaluator for the child is passed as an argument. We first invoke the child evaluator with the new value for *boundvars* and use the resulting *String* to construct the result for the abstraction.

```
semLamAbs :: String → TLam → TLam
semLamAbs x e bv = "(\\" ++ x ++ " -> " ++ ep ++ ")"
  where ep = e ({x} ∪ bv)
```

Finally for the application case the evaluators for both children are invoked with the set of bound variables passed unchanged, after which the result is constructed.

```
semLamApp :: TLam → TLam → TLam
semLamApp l r bv = "(" ++ lp ++ " " ++ rp ++ ")"
  where lp = l bv
        rp = r bv
```

Using these semantic functions for each constructor, a function that takes an AST and the inherited attributes and returns the synthesized attributes is defined as follows.

```
semLam :: Lam → TLam
semLam (Var x)   = semLamVar x
semLam (Abs x e) = semLamAbs x (semLam e)
semLam (App l r) = semLamApp (semLam l)
                             (semLam r)
```

When we run this code on some example AST this gives the expected result.

```
*Main> semLam (Abs "x" $ Abs "y" $
                Var "f" `App` Var "x") Set.empty
"(\\x -> (\\y -> (*f x)))"
```

Note that we pass an empty set as initial value for the set of bound variables.

3.3 Multiple visits

In many cases there are inherited attributes of a nonterminal that (indirectly) depend on one or more synthesized attributes of that same nonterminal. Passing all inherited attributes at once in order to compute the synthesized attributes is not possible in that case. As a solution for this multiple passes over the AST are used, implemented by multiple *visits*. Each visit is a pass over the AST that takes some (possibly zero) inherited attributes and returns some (non-empty list of) synthesized attributes. We assume we know how to do this, as the specific scheduling is usually provided in an AG front end by OAGs scheduling algorithms.

In the case of multiple visits, the visit functions do not only return the synthesized attributes for that visit, but also a function for a subsequent visit. Values can be shared between visits by (partially) applying the function for the subsequent visit in the semantic functions.

Example Let us extend the example in the following way. Instead of inserting all bound variables in the set, we now add only the variables to the set that are free in the body of that abstraction. For now this change might seem arbitrary, but in Section 5 we show that such a change to the code can improve the efficiency of incremental evaluation.

The gathering of free variables is defined as follows.

```
attr Lam
  syn freevars :: {Set String}
sem Lam
  | Var lhs.freevars = {@x}
  | Abs lhs.freevars = @e.freevars \ {@x}
  | App lhs.freevars = @l.freevars ∪ @r.freevars
```

The rules for propagating the bound variables are changed as follows.

```
sem Lam
  | Abs e.boundvars = if   @x ∈ @e.freevars
                      then {@x} ∪ @lhs.boundvars
                      else @lhs.boundvars
  | App l.boundvars = @lhs.boundvars ∩ @l.freevars
        r.boundvars = @lhs.boundvars ∩ @r.freevars
```

With this change the inherited attribute *boundvars* has become dependent on the synthesized attribute *freevars*, so we need two visits: in the first visit the *freevars* attribute is computed, and the visit we had before can then be done as the second visit.

Implementation The AG automatically infers the two visits and the types of the evaluator in Haskell are the following. The first visit returns the set of free variables together with the evaluator for the second visit, and the second visit takes as argument the set of bound variables and returns the pretty printed expression.

```
type TLam¹ = (Set String, TLam²)
type TLam² = Set String → String
```

For the variable case we simply split up the semantics in two visit functions.

```
semLamVar₂ :: String → TLam¹
semLamVar₂ x = v₁ where
  v₁    = ({x}, v₂)
  v₂ bv = if x ∈ bv
          then x
          else "*" ++ x
```

In the case of lambda abstraction we perform the first visit for the child in v_1 to get the free variables of the child, together with the evaluator for the second visit of the child. Both the free variables of the child and the second visit function for the child are needed in v_2, so we partially apply v_2 to pass the values to be used in the second visit.

```
semLamAbs₂ :: String → TLam¹ → TLam¹
semLamAbs₂ x = v₁ where
  v₁ e           = let (efv, e₂) = e
                   in  (efv \ {x}, v₂ efv e₂)
  v₂ efv e₂ bv = let nbv = if   x ∈ efv
                             then {x} ∪ bv
                             else bv
                       er  = e₂ nbv
                   in  "(\\" ++ x ++ " -> " ++ er ++ ")"
```

Finally in the case for application the free variable sets of the left and right child are used in the second visit so again v_2 is partially applied.

```
semLamApp₂ :: TLam¹ → TLam¹ → TLam¹
semLamApp₂ = v₁ where
  v₁ l r                = let (lfv, l₂) = l
                              (rfv, r₂) = r
                          in  (lfv ∪ rfv, v₂ lfv rfv l₂ r₂)
  v₂ lfv rfv l₂ r₂ bv = let lr = l₂ (bv ∩ lfv)
                              rr = r₂ (bv ∩ rfv)
                          in  "(" ++ lr ++ " " ++ rr ++ ")"
```

Finally, we have to *tie the knot*, where the result of the first visit is used to invoke the second visit. Such a function could be defined as follows.

```
semRootLam₂ :: Lam → (Set String, String)
semRootLam₂ e = (fv, v₂ ∅) where
  (fv, v₂) = semLam₂ e
```

4. Incremental evaluation

As described in the introduction our approach is based on change propagation. The idea is that alongside the first execution of a visit a *change handler* is constructed. A change handler is a function which can (efficiently) handle changes to the AST and inherited attributes. The change handler takes as an argument a description of changes to the AST and the (possibly new) inherited attributes, and returns the synthesized attributes and again a change handler.

In this section we develop an incremental version of the one-visit version of the example.

4.1 Edit operations

There is a rich variety of changes which can be made to an AST. For example:

- Add/delete: The most common type of changes which are insertion, deletion or replacement of nodes.
- Reordering: Swapping two or more subtrees, for example in rebalancing a binary tree.
- Duplication: Insert multiple copies of the same subtree.

All such edit operations are supported by our approach, as long as there is an explicit representation of the change that represents which subtree ends up where. Our approach supports all such operations by using the explicit representation of changes from (Bransen and Magalhães 2013).

In Figure 1 we illustrate insertion, deletion and replacements of nodes. The outermost triangle represents the full AST in which an edit operation is performed. The innermost triangle represents the subtree that can be reused: we call this part R. In the left and right picture R represent the same subtree. The shaded triangle represents the part of the AST that is changed. As the shaded triangle in the left picture does not necessarily correspond to the shaded part in the right picture we call the left one S_1 and the right one S_2.

With this representation an insertion can be implemented by choosing S_1 to be equal to R. This means that in the left picture there is no shaded part, so we reuse the full subtree. In S_2 the shaded part is the part which is inserted, of which R is a child. In a similar way deletion can be modelled by choosing S_2 to be equal to R. In that case there is no shaded part in the right picture, so all nodes in the shaded part in the left picture are removed and the full changed subtree is replaced by the smaller subtree R.

For representing reordering of subtrees or duplication of subtrees, multiple inner triangles are used. For more details on this representation we refer the reader to (Bransen and Magalhães 2013).

This representation is inspired by the *zipper* structure (Huet 1997). In zippers the context in which edit operations happen is explicitly stored together with the subtree that is currently in focus. In our approach the context is not explicitly stored in the data type for representing edit operations, but is implicitly stored by the functions that use this data type for performing the incremental computation.

The information that is stored in our data type for edit operations is the path from the root of the tree towards the changed subtree S_1 (the dashed line in Figure 1), the path from the root of the changed subtree S_1 to the reused tree R (the solid line in Figure 1), and the new nodes that are constructed because of this edit operation (the shaded part in the right picture in Figure 1, using a special marker for R).

Paths in the AST are represented by the $Path$ type below, which simply is a list of integers. The elements in the list are child indices from the top of the AST towards the node that the $Path$. For example, $[0, 1]$ refers to the second child of the first child of the root. This information is enough for our approach as we do not need to store extra contextual information based on the actual constructors on the path.

$$\mathbf{type}\ Path = [Int]$$

The part of the AST that is inserted after a change is represented by the following data type which is similar to Lam. The special marker Lam_{R} indicates a place where a reused subtree must be inserted.

$$\begin{aligned}&\mathbf{data}\ Lam_{\mathrm{R}}\\ &\quad = Lam_{\mathrm{R}}\ Path\\ &\quad \mid Var_{\mathrm{R}}\ \ String\\ &\quad \mid Abs_{\mathrm{R}}\ \ String\ Lam_{\mathrm{R}}\\ &\quad \mid App_{\mathrm{R}}\ \ Lam_{\mathrm{R}}\ \ Lam_{\mathrm{R}}\end{aligned}$$

Using $Path$ and Lam_{R} the type for a full change to the AST is represented as follows.

$$\mathbf{type}\ IncInsert = (Path, Lam_{\mathrm{R}})$$

The first element is the path from the root to the changed subtree, and the second element describes how the new subtree must be constructed.

As an example, let us look at the edit from the top level expression $f\ y$ to $f\ (\lambda x \rightarrow y)$ where f and y are arbitrary expressions. This edit is the insertion of a lambda abstraction in the second child of the root and is represented as follows.

$$\begin{aligned}&addAbs :: IncInsert\\ &addAbs = ([1], Abs_{\mathrm{R}}\ \texttt{"x"}\ (Lam_{\mathrm{R}}\ [1]))\end{aligned}$$

The first $[1]$ here indicates that the insertion should take place in the second child of the root, and the second $[1]$ indicates that the subtree originally at that location should be reused in the inserted tree.

This representation of changes to the AST does not prevent us from constructing invalid edit operations. For example, a path could select the second child of an Abs node, which only has a single child. These values are intended to be generated only, that is, not hand-written by a human programmer, for example by the algorithm from (Bransen and Magalhães 2013). In this paper we therefore assume that all our edit operations apply to the AST under consideration. Another reasonable approach to take is to generate a suitable error message in that case, since in an earlier phase something has gone wrong.

4.2 Change propagation

At first glance one could think that attribute values can only change in the part of the tree that has changed or on the path towards the change. Since inherited attributes can depend on synthesized attributes of other parts of the AST this is not true, and recomputation may be necessary in parts of the AST that have not changed.

Furthermore, during an incremental update our AG machinery should not only be able to respond efficiently to changes, but it should also be possible to select a subtree which is reused. We therefore use another data structure for representing the type of changes in our AST, which will be passed as an argument to a change handler:

$$\begin{aligned}&\mathbf{data}\ IncChange\\ &\quad = NoChange\\ &\quad \mid Change\ (Path \rightarrow TLam_{\mathrm{I}})\ IncInsert\\ &\quad \mid Select\quad Path\end{aligned}$$

The first constructor is used to recompute a part of the AST where only the inherited attributes have changed and not the AST, the second for propagating a change in the AST, and the last for selecting and retrieving the visit function of a given subtree so it can be reused.

Now the type of the visit functions should be changed to also return a change handler. Intuitively attribute evaluation yields a change handler alongside synthesized attribute values:

$$\begin{aligned}&\mathbf{type}\ TLam_{\mathrm{I}} = Set\ String\\ &\qquad\qquad\qquad \rightarrow (String, IncChange \rightarrow TLam_{\mathrm{I}})\end{aligned}$$

To avoid an infinite type in Haskell we reformulate $TLam_{\mathrm{I}}$ by wrapping it in a newtype:

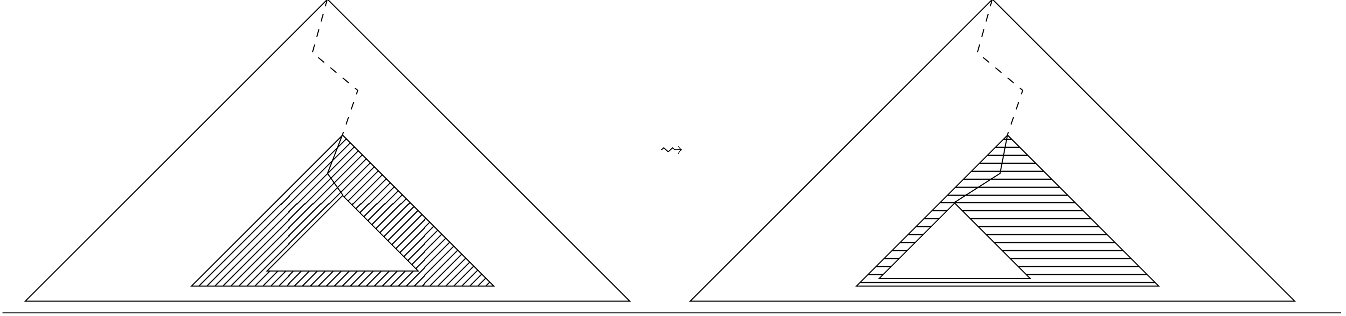

Figure 1. Illustration of an edit operation with the changed part of the tree being shaded.

```
newtype TLam_I = TLam_I {
  runTLam_I :: Set String
                 → (String, IncChange → TLam_I)
  }
```

4.3 Implementation

After a change, the newly inserted part of the AST as described by the Lam_{R} data type should be constructed. We define the following helper function for constructing this new part of the AST, which takes the Lam_{R} describing the shape of the new tree and a selection function for retrieving the visit function for a subtree that is being reused.

```
semRef :: (Path → TLam_I) → Lam_R → TLam_I
semRef sel d = case d of
  Lam_R p    → sel p
  Var_R x    → semLamVar_I x
  Abs_R x e  → semLamAbs_I x (semRef sel e)
  App_R l r  → semLamApp_I   (semRef sel l)
                             (semRef sel r)
```

In Figure 2 the incremental version of *semLamApp* is shown. The computation of the attributes is not changed, only at the child visits an extra argument is returned, which is for each child a function that can efficiently respond to changes in the tree.

The $memv_1$ function is where the incremental behaviour is achieved and it is used when the subtree rooted at this node has not changed. It uses the inherited attribute bv to check whether the new inherited attribute bv' has changed. If this is not the case then the values of the synthesized attributes have not changed either so the previously computed values can immediately be returned. In case of a change in inherited attributes, the current visit is recomputed but for the children the incremental version is used.

Note that a full equality check is done for the inherited attributes. This might seem to be an innocent operation but it is important to realise that this can have a great impact on performance. When the values of the inherited attributes become very large, the equality checking might become computationally expensive. In Section 7 we come back to this issue and describe how we can improve this. Furthermore, the equality check restricts the types of the inherited attributes, since there must be an *Eq* instance for each of them.

Using the helper functions *semRef* and $memv_1$ we construct the change handler *incr*. Its argument c of type *IncChange* allows *incr* to react to changes as follows.

In case of *NoChange* the current subtree has not changed but the synthesized attributes need to be computed from the given inherited attributes. Since this function is constructed in an earlier visit we have already computed a set of attributes once and can therefore return the $memv_1$ function to reuse these previously com-

```
semLamApp_I :: TLam_I → TLam_I → TLam_I
semLamApp_I l r = TLam_I (v_1 l r) where
  v_1 l r bv = (pp, incr) where
    (lp, l') = runTLam_I l bv
    (rp, r') = runTLam_I r bv
    pp       = "(" ++ lp ++ " " ++ rp ++ ")"
    memv_1 =
      TLam_I (λbv' → if   bv ≡ bv'
                     then (pp, incr)
                     else v_1 (l' NoChange)
                              (r' NoChange) bv')
    incr c  = case c of
      NoChange               → memv_1
      (Select [] )           → memv_1
      (Select (0 : is))      → l' (Select is)
      (Select (1 : is))      → r' (Select is)
      (Change sel ( [], r))  → semRef sel r
      (Change sel (0 : is, r))
        → TLam_I (v_1 (l' (Change sel (is, r)))
                      (r'  NoChange))
      (Change sel (1 : is, r))
        → TLam_I (v_1 (l'  NoChange)
                      (r' (Change sel (is, r))))
```

Figure 2. Incremental version of *semLamApp*

puted values if possible. In case that the inherited attributes have changed the $memv_1$ function recomputes the visit and propagates the *NoChange* to its children.

For a *Select* value a part of the already computed subtree should be selected for being reused. In the case of the current subtree being selected again $memv_1$ is returned, otherwise the *Select* is propagated to the corresponding child.

Finally the *Change* value describes a change in the AST. When the current node is changed, the *semRef* helper function is used to construct the new nodes. The part to be reused is found by using the *Select* construct to select the corresponding function. When a child changes, the change is propagated to the corresponding child and the current visit is recomputed with the changed child.

After a change to the AST, recomputation happens at the path from the root to the changed node, and in all nodes where at least one inherited attribute has changed. When multiple changes are propagated, only the immediate previous result is reused. Our mechanism therefore can be seen as an embedded implementation of a cache for attribute values which stores one value per node.

5. Facilitating incrementality

In the previous section we have described the incremental version of the pretty printing example. Although it has some form of incremental behaviour, it is not more efficient in many practical cases. This can however be improved by making small changes to the implementation, which have little effect on the normal evaluation, but have great impact on the incremental behaviour. More generally, use of incremental mechanisms requires a programmer to rethink its program code as to facilitate incremental behaviour; incrementality is not something which totally comes for free.

The issues and their solutions which we describe in this section are not unique to incremental evaluation of AGs. We believe that similar problems appear in other solutions for automatic incremental evaluation, for example that of (Chen et al. 2012).

5.1 Projection of inherited attributes

Our incremental AG evaluation machinery uses equality of inherited attributes for deciding when to do recomputations. However, it may happen that small changes of some inherited attribute have no influence on the value of the synthesized attributes.

As an example, let us change an expression e to $\lambda x \rightarrow e$ where x does not appear anywhere in e. We know that the result of pretty printing e should not change, hence it should be possible to reuse the previous result without further computations. However, since we add the variable x to the set of bound variables, the inherited attributes of e has changed and the pretty printing of e is recomputed.

The solution is to use only the *relevant part* of inherited attributes. The $boundvars$ set is used to decide how to pretty print variables. However, the only variables that we ever look up in this set are the variables that appear in the subexpression. Therefore, if at a lambda abstraction we add a variable to the set that does not appear (free) in the subexpression, this does not change the result of the pretty printing. It does however have negative influence on the incremental behaviour, since the inherited attributes have changed and recomputation happens.

Concretely, to solve this problem we switch to the two-visit approach that we already showed in Section 3.3. In the first visit $freevars$ is constructed in a bottom-up fashion. In the second visit the rest of the attributes are computed based on the inherited set $boundvars$ similar to our earlier approach. However, since $freevars$ is already available for all children at the beginning of the second visit, it can be used to do a *projection* on the set $boundvars$. For Abs, x is only added to $boundvars$ when x appears in the set of free variables of e, and recomputation only happens when there is a relevant change to the bound variables.

Although this solution is specific to the pretty printing example, there is a more general pattern behind this. In Section 5.5 we generalise the insights used here.

5.2 Fresh variable generation

In many program analyses and transformations there is a need for fresh variables. The standard way of implementing fresh variable generation in AGs is to use a chained attribute of type $Integer$, giving the index of the next fresh variable. At each point where a fresh variable is needed, the index is used for generation and the number is increased before being passed to the next node.

This approach is useful in standard AG evaluation since it can be implemented on AG level and no special back end support is needed. With incremental evaluation however this method of generating fresh variables has a negative influence on the effectiveness of the incremental evaluator, because the chained behaviour propagates changes through the whole AST, and chained behaviour in general has a negative influence on incremental evaluation.

For example, imagine a type inference algorithm for the lambda calculus implemented with AGs. Now suppose that an expression e is extended with a let binding at top level, so is converted to $\mathbf{let}\ x = \ldots\ \mathbf{in}\ e$ where x is not free in e. It is clear that the type of e has not changed due to this edit and should be reused. However, when the fresh type variables are first generated for the binding of x, the chained attribute for the variable generation that is passed to e is different from the first evaluation, so all fresh type variables used in e change! As a result of this not only are all types inferred again, but there is also some overhead involved which results in the incremental evaluation being slower even in such a simple case.

The general problem with the fresh variable generation as a changed attribute is that its definition imposes more restrictions than necessary. Instead of defining that we need a unique variable at each node, we have defined a strict order in which the variables should be generated. This results in extra dependencies that in case of incremental evaluation result in unnecessary recomputations.

The solution we propose is to abstract over fresh variable generation at AG level and use a different mechanism for generating fresh variables. This could be handled in the parsing stage by producing a fresh variable generator for each node, for example by using the path from the root to the node as seed.

For now we use a simpler approach and we generate fresh variables in a more imperative way using some form of global state. In a pure setting this can be achieved by running the evaluation in a monadic environment (Wadler 1990). We use a monad $Unique$ with a function $getUnique :: Unique\ Integer$ for fresh variable generation, for which we do not show the implementations. When evaluating our pretty printing example in such monadic environment, the type $TLam$ for examples becomes the following.

$$
\begin{array}{l}
\mathbf{newtype}\ TLam_{\mathrm{I}} = TLam_{\mathrm{I}}\ \{ \\
\quad runTLam_{\mathrm{I}} :: Set\ String \rightarrow \\
\qquad\qquad Unique\ (String, IncChange \rightarrow TLam_{\mathrm{I}}) \\
\quad \}
\end{array}
$$

As a result of this change, the outcome of an incremental evaluation is not necessarily equal to the usual evaluation anymore; they are however equivalent under alpha renaming, which is fine for practical cases where fresh variables are used. In the particular case of type inference a normalisation step could be added which does alpha renaming to some normal form for observational equality.

There is however a problem with this way of generating fresh variables: when a subtree is duplicated due to an edit and no recomputation happens in the resulting subtrees, their fresh variables are shared. In usual applications this is undesirable and could lead to wrong results. This technique does therefore not work when duplication is allowed in the edit operations, but more elaborate techniques for fresh variable generation can work without the need to take special care about this on the AG level.

5.3 Intermediate constraint solving

Many program analyses can be written as a constraint based algorithm. Constraints are usually generated in a bottom up fashion making such algorithms very suitable for incrementalization when written as an AG. However, in such algorithms most time is usually 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. Because in many cases there is a certain order in which constraints should be solved, it is not always

```
semLamApp_I^2 :: TLam_I^1 → TLam_I^1 → TLam_I^1
semLamApp_I^2 l r = TLam_I^1 (v1 l r) where
  v1 l r = do
    (lfv, l')  ← runTLam_I^1 l
    (rfv, r') ← runTLam_I^1 r
    let fv = lfv ∪ rfv
    return (fv, TLam_I^2 (v2 lfv rfv fv l' r'))
  v2 lfv rfv fv l r bv = do
    (lp, li)   ← runTLam_I^2 l (bv ∩ lfv)
    (rp, ri)   ← runTLam_I^2 r (bv ∩ rfv)
    let p = "(" ++ lp ++ " " ++ rp ++ ")"
        memv1 = TLam_I^1 (return (fv, TLam_I^2 memv2))
        memv2 bv' = if      bv' ≡ bv
                    then return res
                    else  do
          (_, l') ← runTLam_I^1 (li NoChange)
          (_, r') ← runTLam_I^1 (ri NoChange)
          v2 lfv rfv fv l' r' bv'
        incr c = case c of
          NoChange                  → memv1
          (Select      []       )   → memv1
          (Select      (0 : is) )   → li (Select is)
          (Select      (1 : is) )   → ri (Select is)
          (Change sel ([]    ,  r)) → semRef_I^2 sel r
          (Change sel (0 : is,  r))
              → TLam_I^1 (v1 (li (Change sel (is, r)))
                             (ri  NoChange))
          (Change sel (1 : is,  r))
              → TLam_I^1 (v1 (li NoChange)
                             (ri (Change sel (is, r))))
        res = (p, incr)
    return res
```

Figure 3. Incremental two-visit version of *semLamApp*

possible to solve all constraints as soon as they are generated. In cases where the constraints can (partially) be solved in intermediate stages, this is beneficial for the incremental evaluation.

For example for constraint based type inference, it is possible to solve all constraints for a closed expression in which no free variables appear. Depending on the application such situation might or might not happen frequently, but when it does it can lead to large speed optimizations. In Section 6 we show some benchmark results related to this example.

5.4 Implementation

With the improvements described in the previous sections the implementation has changed in several ways. When fresh variables are used the visits need to be performed in a monad, and for the projection of the inherited attribute we require two visits.

Although our example does not use fresh variables, we still show the monadic implementation for the sake of example. As with the non-incremental two-visit implementation we use two types to represent the two visits. The types of these two visits are therefore the following.

```
newtype TLam_I^1 = TLam_I^1 {
  runTLam_I^1 :: Unique (Set String, TLam_I^2)
  }
```

```
newtype TLam_I^2 = TLam_I^2 {
  runTLam_I^2 :: Set String
      → Unique (String, IncChange → TLam_I^1)
  }
```

We see that the inherited and synthesized attributes of the visits are similar to the earlier implementation of our example with two visits, but wrapped in a monadic computation.

In Figure 3 we show the corresponding implementation for the *App* constructor. We see that the visit code has become monadic which imposes a restriction on the order in which the attributes are computed, but as mentioned earlier we assume that the evaluation order is statically known. In v_1 the value of *fv* is computed as expected and the visit function for v_2 is returned, which is partially applied to pass values from the first to the second visit, in particular the free variables of the children, the state of the two children and the value of *fv*.

In v_2 the other synthesized attributes are computed as before. The $memv_1$ and $memv_2$ functions have slightly changed to match the new types of the visit functions. Because the first visit does not have any inherited attributes, the $memv_1$ function can always directly return the previously computed synthesized attributes.

5.5 General observations

As we have discussed in Section 5.2 using a chained attribute for generating fresh variables is bad for incremental performance. For the particular case of fresh variable generation we have used special implementation to solve this problem, but from this insight we can draw a more general conclusion: chained attributes are bad for incremental performance, synthesized attributes are good. For this same reason type inferencing with algorithm $\mathcal{W}$ which has a chained behaviour is less suitable for incremental evaluation than the constraint based type inference algorithm.

In Section 5.1 and Section 5.3 we described two optimizations that are specific to certain algorithms. The general observation there is that computations must be done *as early as possible* for better incremental behaviour, where *early* means early in the control flow. In AGs this means that operations on inherited attributes should happen as close to the root of the tree as possible, while operations on synthesized attributes should happen as close to the leaves of the tree as possible.

The motivation behind this idea is that we would like to keep the number of values that need to be changed after an incremental update to the AST as small as possible. When some computation relies on a value that is changed, it has to be recomputed at some point. However, when that recomputation is done earlier this might avoid recomputation of other values that are "in between".

6. Evaluation

To evaluate the effectiveness and to 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 lambda expressions and edit operations 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.

We use the QuickCheck machinery for generating arbitrary instances of *Lam*. To control the size of our benchmarks the func-

```
arbLam :: Int → Gen Lam
arbLam n | n < 2      = genVar
         | otherwise = frequency [(2, genLam n)
                                 ,(3, genApp n)]
genApp :: Int → Gen Lam
genApp n = do
  nl ← choose (1, n − 1)
  l  ← arbLam nl
  r  ← arbLam (n − nl − 1)
  return $ App l r
genLam :: Int → Gen Lam
genLam n = do
  v ← choose ('a', 'j')
  e ← arbLam (n − 1)
  return $ Abs [v] e
genVar :: Gen Lam
genVar = do
  v ← choose ('a', 'j')
  return $ Var [v]
```

Figure 4. The generation of arbitrary *Lam* instances.

tions take a parameter indicating the size of the expression that should be generated, which is our case is the number of nodes in the tree representation. The implementation is showed in Figure 4. For the variable generation we use 10 different variables to balance the number of bound and free variables.

6.1 Versions

In our benchmark we have compared the following four different versions of our example algorithm. Each version has been given a short name which is used to refer to the version in later sections.

Base The base version is the code for the pretty printing example as described in Section 3.2. This version is used as the baseline for the benchmarks.

Incr This is an incremental version of **Base** using the techniques described in Section 4.

Base2 To give an idea of the influence of the improvements from Section 5 on the non-incremental version of the code we have included this version. This code is the non-incremental two-visit approach from Section 3.3 and includes the optimizations described in Section 5.1.

Incr2 This is the incremental version of **Base2**, which includes all improvements described in this paper.

Although there are more versions that could be defined based on the separate changes described in Section 5, we believe that these four versions give a good impression of the effectiveness of the described approach.

6.2 Benchmarks

Our benchmark contains the following eight different benchmarks, which we think gives a good overview of both the extreme cases and the typical cases as found in a syntax-directed editor that infers types during editing.

Initial The pretty printing the initial AST which forms the base of each of the following eight possible changes.

Add lambda top level Adding a lambda abstraction at the top level to the expression as described earlier in this paper.

Add app top level Adding an application of a variable at the top level of the expression.

Full change The full expression is replaced by another expression, to illustrate the overhead in cases where no efficiency gain can be achieved.

Delete The deletion of an arbitrary subexpression.

Add application The addition of an application at an arbitrary place in the expression.

Add lambda The addition of a lambda abstraction at an arbitrary place in the expression.

Replace variable The replacement of an arbitrary variable by another arbitrary variable.

Each benchmark has been applied to a set of 100 lambda expressions, one of each size between one and 100. The arbitrarily generated changes are generated separately for each of the 100 expressions only once, such that all different versions of the code use the same set of changes.

6.3 Results

In Figure 5 we show the result of the benchmarks. The times have been scaled relative to **Base**, so a bar of height 0.5 indicates that a version used half the time of the base version.

The overhead of the incremental evaluation for the initial run is small as we see in (a). The one visit approach is slightly more efficient than the two visit approach so **Base2** takes 1.0008 times the time of **Base**. Due to the construction of change handlers, **Incr** is 1.0214 times slower than **Base**. Because Haskell uses lazy evaluation the construction of the change handlers is the only work that is done, which results in only a very small overhead.

The extreme cases of (b) and (c) illustrate that large speedups can be achieved. In (c) the **Incr2** version takes 0.0077 times the time of **Base**. From (c) it is also clear what the effect of the projection of *boundvars* from Section 5 is, as can be seen in the large difference between **Incr** and **Incr2**.

When the full AST is changed as in (d) we find that **Incr2** is 1.0512 times slower than **Base**. Because no information can be reused in this example an incremental version of the code will never improve the runtime. It is therefore to be expected that an incremental version is slower and the overhead presented here is only minor.

The other four benchmarks show results similar to these. In general **Incr2** takes less time except for extreme cases where no information can be reused. For some other cases the improvement is only minor, which is also to be expected for the type inferencing algorithm; some small changes to an expression can have large impact on all types.

6.4 Constraint based type inference

As a more real world test case we have implemented a type inferencing algorithm with AGs. The algorithm that we use is constraint based type inferencing from (Heeren et al. 2002) on the lambda calculus with let bindings and let polymorphism. The algorithm uses a single inherited attribute which is a set of type variables, and three synthesized attributes of which one is the set of constraints.

The different versions of implementation and the benchmarks are similar to the pretty printing example. In the **Base** version there is a single visit in which all constraints are gathered and solved at top level. The **Base2** version uses two visits and intermediate constraint solving as discussed in Section 5. The **Incr** and **Incr2** are the corresponding incremental versions, and all four versions use the fresh variable generation in a monadic way.

In Figure 6 we show the benchmark results of the type inference algorithm. The sizes of the benchmarks are similar to the pretty

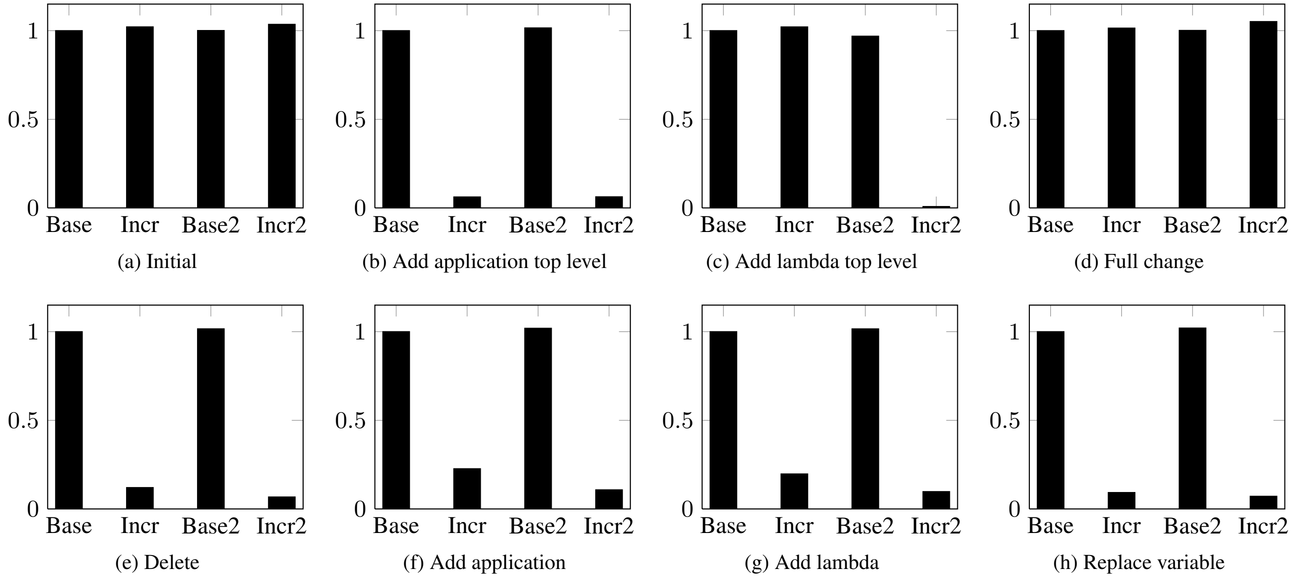

Figure 5. Benchmark results of pretty printing example

printing example and so are the results. All in all we feel that these benchmarks show that our stateless incremental attribute grammar machinery can greatly improve the runtime.

7. Discussion

In this paper we have described our stateless incremental AG machinery using pretty printing with free variable annotation as a running example. Although this example is a good example for showing the core aspects of our approach, it is not a real world use case. In this section we discuss some shortcomings of the example and of our approach in general.

7.1 Pretty printing example

The pretty printing example uses a single data type *Lam* for its AST. However, in many applications a family of mutually recursive data types is used for describing the AST. Our approach fully supports a family of mutually recursive data types for the AST. The visit functions only need minor changes in such case, but for the edit operations a more advanced change is necessary. With multiple data types the inserted nodes om the tree can be of different types. To keep the whole approach type-safe *Generalized Algebraic Data Types* (Cheney and Hinze 2003; Xi et al. 2003) are required to describe the edit operations such that all data types for edit operations have a type parameter indicating the type of the node in which the edit is performed.

In both the single the two-visit approach only the last visit uses an inherited attribute, and it is not straightforward to extend our technique to multiple visits where each visit has both inherited and synthesized attributes. This is a shortcoming of the pretty printing example, but in our implementation we do support such AGs. By storing the intermediate states of the children after each visit, every visit can be recomputed whenever necessary.

7.2 Type inference example

For the type inference implementation we have assumed that both the initial expression and the expression after a change are well-typed. However, in practice during editing the expressions can be temporarily ill-typed. For example, when an argument is added to a function one might first add this argument to the function at the definition site, which changes the whole expression to an ill-typed expression, after which the arguments are added at the call site to make it well-typed again. Support for this has been left out for simplicity, but it is not hard to change our approach to support such behaviour. Note that in such case even though the full expression might be ill-typed, subexpressions that are unchanged can still be well-typed and the incremental behaviour is retained.

For the type inference example we have made the generation of constraints incremental, but the constraint solving procedure itself always performs a full recomputation, even if for example the input is alpha equivalent to an earlier input. This problem can be solved by lifting the constraint solving to the AG level, such that the techniques described in this paper can be applied.

7.3 Drawbacks and shortcomings

The title of this paper states that we develop stateless AG machinery. However, with the fresh variable generation as described in Section 5.2 the evaluation is done inside a monad which essentially adds state. The incremental evaluation is however stateless and the state used for the generation of fresh variables is a unique case; the AG code itself and the incremental evaluation thereof do not use any form of global state.

We have given an alternative to fresh variable generation to avoid the chained behaviour, but we have not solved the problem of avoiding chained behaviour in general. In some cases it is possible to replace the chained attribute by two or more attributes in multiple visits, for example by first gathering the results using a synthesized attribute and then distributing these results over the AST with an inherited attribute. It is however future work to automatically transform chained attributes in an AG to version that does not have the chained behaviour.

Full equality on inherited attributes is used in our incremental evaluation for deciding whether or not do recomputation. This does not only restrict the types of our inherited attributes, but also might be a large bottleneck in case of expensive equality checks for our inherited attributes. However, in a usual attribute grammar system we know more than just the semantic rules. In many cases attribute values are simply copied from the parent to the child

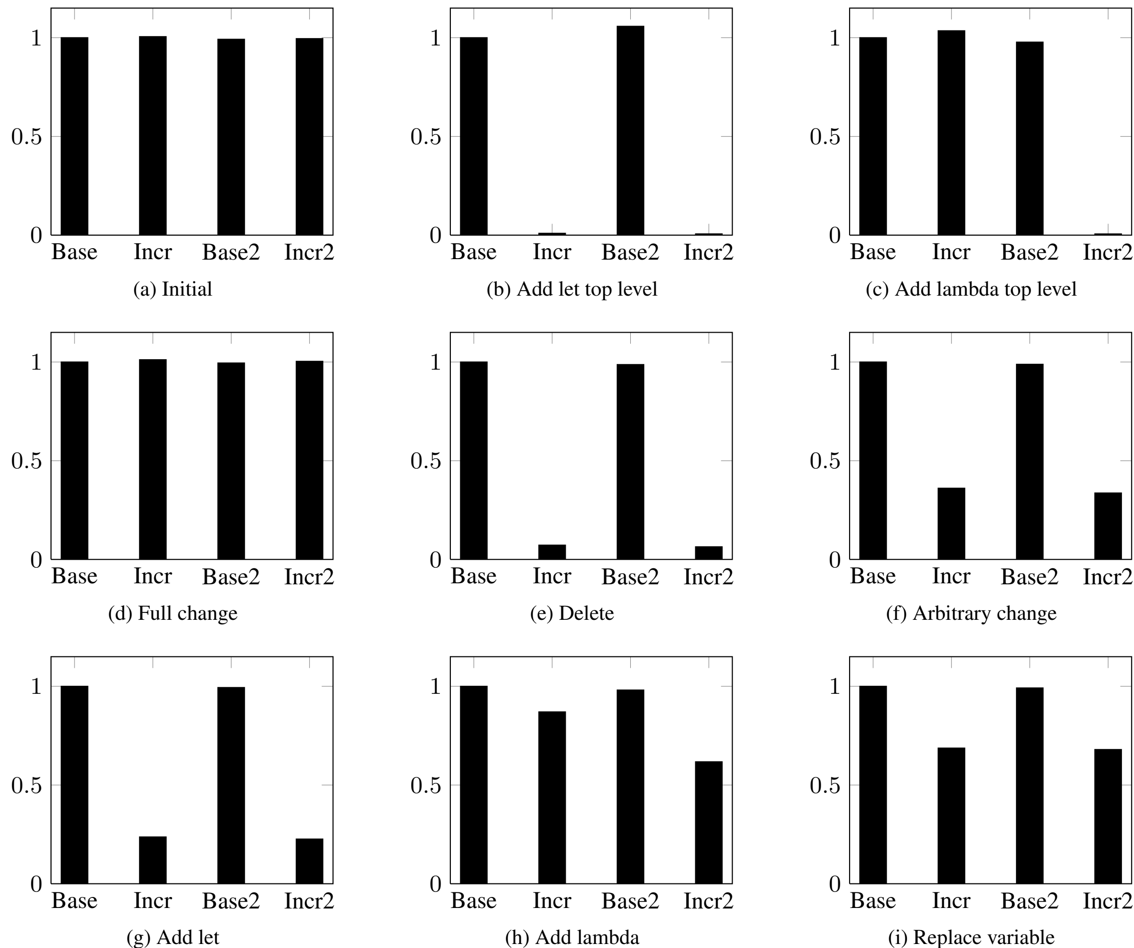

Figure 6. Benchmark results of constraint based type inference

node, which is also the case for the $boundvars$ attribute in the one-visit approach at the App constructor. In such cases we statically know that inserting or deleting such nodes has no influence on the value of $boundvars$, and we thus need no recomputation for their children. It is future work to use that information to avoid equality checks and to test how well such approach would work in practice, because the obvious drawback is that we may not recognise all cases where the value did not change.

The price that is paid for the decrease of runtime in incremental evaluation usually is the increase of memory consumption. As expected the incremental code in our approach also consumes more memory than the non-incremental version, but the memory consumption increases only by a constant amount per node in the AST. Because memory is relatively cheap and the order of magnitude of the total memory consumption for the application does not change with our approach, we think that this increase is not a problem.

In our approach we only efficiently handle cases where the inherited attributes have not changed, but the same trick might be applied for the synthesized attributes. For example, in the type inference example a change in a subtree might result in the same constraints and type being generated, and in the approach as described in this paper the constraint solver is run again to solve the constraints. Since the result of the previous constraint solving phase is still available this should not be necessary. By also requiring equality instances for our synthesized attributes it would be possible to also avoid such recomputations. However, the values of the synthesized attributes of the child are only known after doing the part of a visit in which the inherited attributes of that child are computed and the child visit is invoked. It is therefore not immediately clear how the visit functions should be changed to support this form of incrementality and it is future work to support this.

An important assumption of the incremental evaluation machinery is that an incremental run returns the same result as its non-incremental counterpart. Although we have verified that our techniques work correctly for the examples, we have not formally proved soundness or correctness of our incremental transformation.

7.4 Attribute grammar extensions

In *Higher-Order AGs* (HAGs) (Vogt et al. 1989) attribute values themselves can be trees which can again be decorated with attributes. HAGs can for example be used for modelling multiple phases of a compiler. In the type inference example, we could add a more verbose version of lambda expressions which is first desugared to the simple version on which the type inference is done. Such desugaring step could be described with HAGs by defining a synthesized attribute on the initial AST of type Lam which is the desugared version of the expression. At the top level this attribute is then instantiated as a tree and the type inference is performed. With such higher order behaviour incremental evaluation based on change propagation often breaks because the (implicit)

link between nodes in the initial AST and nodes in the desugared version is lost. It is future work to support HAGs in our approach such that changes in the original AST can be propagated to changes in the higher order children.

Many other AG extensions exist and it is yet an open question if and how these extensions could be incorporated in the presented approach. For example, reference attributes as used in the work of Reference Attributed Grammars (Hedin 2000) are problematic already for the purely functional implementations of non-incremental AGs, and it is therefore not clear if it is feasible to support those in our approach.

8. Conclusion

In this paper we have described the machinery that has been used for the incremental evaluation of AGs. We have described some changes that makes AG code better suitable for incremental evaluation and we have illustrated all of this using the pretty printing example. In the end, the benchmark results for both the pretty printing and the type inferencing example show that the incremental evaluation of AGs with our machinery gives the desired efficiency improvements in many cases and only a minor overhead in cases where no information can be reused. From this we conclude that the given approach works as expected and has the desired properties of incremental evaluators.

In Section 7 we have described some ideas for future work in order to support a wider class of AG programs, in particular higher-order AGs which are used in the Utrecht Haskell Compiler in many places. We believe that the ideas presented in this paper form a solid ground for further work on stateless incremental attribute grammar evaluation.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments.

References

Acar, U. A., Blelloch, G. E., Blume, M., and Tangwongsan, K. (2006). An experimental analysis of self-adjusting computation. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 96–107, 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.

Bransen, J., Middelkoop, A., Dijkstra, A., and Swierstra, S. D. (2012). The Kennedy-Warren algorithm revisited: ordering Attribute Grammars. In Russo, C. and Zhou, N.-F., editors, *Practical Aspects of Declarative Languages*, volume 7149 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin / Heidelberg.

Chen, Y., Dunfield, J., and Acar, U. A. (2012). Type-based automatic incrementalization. In *Programming Language Design and Implementation*.

Chen, Y., Dunfield, J., Hammer, M. A., and Acar, U. A. (2011). Implicit self-adjusting computation for purely functional programs. In *International Conference on Functional Programming*, pages 129–141.

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

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

Demers, A., Reps, T., and Teitelbaum, T. (1981). Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 105–116, 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.

Hedin, G. (2000). Reference attributed grammars. *Informatica (Slovenia)*, 24(3).

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.

Huet, G. (1997). The zipper. *Journal of Functional Programming*, 7(5):549–554.

Kastens, U. (1980). Ordered attributed grammars. *Acta Informatica*, 13:229–256. 10.1007/BF00288644.

Kennedy, K. and Warren, S. K. (1976). Automatic generation of efficient evaluators for attribute grammars. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pages 32–49, New York, NY, USA. ACM.

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).

Ramalingam, G. and Reps, T. (1993). A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 502–510, New York, NY, USA. ACM.

Reps, T. and Teitelbaum, T. (1984). The synthesizer generator. *SIGPLAN Not.*, 19:42–48.

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.

Schrage, M. M. (2004). *Proxima – a presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, The Netherlands.

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

Swierstra, S. D. and Duponcheel, L. (1996). Deterministic, error-correcting combinator parsers. In Launchbury, J., Meijer, E., and Sheard, T., editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag.

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.

Wadler, P. (1990). Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 61–78, New York, NY, USA. ACM.

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.

Yeh, D. and Kastens, U. (1988). Improvements of an incremental evaluation algorithm for ordered attribute grammars. *SIGPLAN Not.*, 23(12):45–50.