CACHET: An interactive, incremental-attribution-based program transformation system for deriving incremental programs

Yanhong A. Liu*
Department of Computer Science, Cornell University, Ithaca, NY 14853

Abstract

This paper describes the design and implementation of an interactive, incremental-attribution-based program transformation system, CACHET, that derives incremental programs from non-incremental programs written in a functional language. CACHET is designed as a programming environment and implemented using a language-based editor generator, the Synthesizer Generator, with extensions that support complex transformations. Transformations directly manipulate the program tree and take into consideration information obtained from program analyses. Program analyses are performed via attribute evaluation, which is done incrementally as transformations change the program tree. The overall approach also explores a general framework for describing dynamic program semantics using annotations, which allows interleaving transformations with external input, such as user input. Designing CACHET as a programming environment also facilitates the integration of program derivation and validation with interactive editing, compiling, debugging, and execution.

1 Introduction

Program transformation systems are important tools that implement various program manipulations that preserve program semantics and improve program performance [3, 11, 12, 15, 24, 26, 28, 31].

Interactive program transformation. Program transformation systems are often required to be interactive, for at least the following two reasons. First, the goal of the system is often so ambitious that no fully automatic transformation can succeed; interaction allows convenient semi-automatic transformation. Second, the study of transformation itself often consists of trying different transformations; interactive invocation of these transformations provides control during experimentation.

The usability of an interactive program transformation system depends greatly on its interface. However, designing and implementing such an interface, especially with various program manipulation functions, is a heavy task.

*The author gratefully acknowledges the support of the Office of Naval Research under contract No. N00014-92-J-1973.
Author's Email: yanhong@cs.cornell.edu

Incremental program analysis. For any non-trivial program transformation, substantial program analysis is needed. The analysis results, such as dependencies, types, and unique identifiers, are used to decide what transformations to apply and where and how to apply them. After each transformation step, the transformed program needs to be analyzed again. Such interleaving of analyses and transformations continues until certain criteria for stopping are satisfied.

Each transformation step typically changes only a small portion of the program; thus one should be able to update the analysis results to reflect the change. This is the problem of incremental program analysis under program transformation. Solving this problem is indispensable for speeding up the program analyses and thus the overall program transformations.

However, until now, all program transformation systems we are aware of use some ad hoc inference engine to do analysis, store the analysis results in some database, and use some rewrite engine to do transformation. The simple storage of the analysis results makes it hard to incrementalize the analyses under the transformations. This creates a performance bottleneck that is more severe for more automated systems [24].

Using an attribute-grammar-based programming environment. Interactive program transformation and incremental program analysis naturally lead one to consider an attribute-grammar-based programming environment. The Synthesizer Generator is a commercially available system that generates such environments [30].

With such a tool, the syntax of programs can be described using a context-free grammar, and properties of programs can be described using attribute equations. In such a *declarative* framework, program analysis is performed by program tree attribution, program transformation directly mutates the program tree, and incremental analysis is conducted by incremental attribute evaluation after each transformation step.

CACHET. CACHET is an interactive, incremental-attribution-based program transformation system for programs written in a first-order functional language. It is implemented using the Synthesizer Generator, with extensions to support complex tree transformations. Incremental program analysis is performed by incremental attribute evaluation, provided automatically by the Synthesizer Generator.

Attribute grammars and incremental attribute evaluation methods have been well addressed in the literature [6] for describing static program semantics. They are not the subject of this paper. Instead, this paper describes how to adopt a traditional programming environment and make it suitable for performing complex tree transformations, interleaving transformations with external inputs, such as user inputs, and making more use of attribution mechanisms.

Deriving Incremental Programs. CACHET has special functionality for deriving incremental programs [19]. Given a program f and an input change $\oplus$, a program f' that computes the result of $f(x \oplus y)$ efficiently by making use of the value of $f(x)$ is called an *incremental version* of f under $\oplus$. Incremental computation has wide applications through computer software, e.g., optimizing compilers [1, 5, 23], transformational program development [3, 24, 31], and interactive systems [2, 22, 30].

Liu and Teitelbaum [19] give a systematic transformational approach for deriving an incremental program f' from a given program f and an input change $\oplus$. The basic idea is to identify in the computation of $f(x \oplus y)$ those subcomputations that are also performed in the computation of $f(x)$ and whose values can be retrieved from the cached result r of $f(x)$. The computation of $f(x \oplus y)$ is transformed symbolically to avoid re-performing these subcomputations by replacing them with corresponding retrievals. This efficient way of computing $f(x \oplus y)$ is captured in the definition of $f'(x, y, r)$.

The transformational approach can be made automatic or semi-automatic depending on the required incrementalization power. CACHET performs *function introduction with generalization* [19] to introduce incremental versions with cached results of the previous computation as parameters, and it uses *auxiliary specialization* [19] to find subcomputations whose values can be retrieved from the cached results of the previous computation.

CACHET has been used to derive numerous incremental programs, including most of the examples in [19, 20, 21]. It has also been of great help in studying transformations for caching intermediate results [20] and discovering auxiliary information [21].

The rest of the paper is organized as follows. Section 2 gives an overview of the desired features, the problems to be solved, and the suggested solutions. Section 3 describes the implementation techniques used for CACHET. Section 4 contains a sample derivation of an incremental program using CACHET. Section 5 discusses related work. Section 6 discusses future work.

2 System design

This section discusses desired features, problems to be solved, and suggested solutions for adopting a traditional programming environment for program transformations.

2.1 Complex tree transformation

To enable program transformations, the major power that needs to be added to a traditional attribute-grammar-based framework is a metalanguage for complex tree transformations.

The mechanisms needed for tree transformations are mainly pattern matching and pattern instantiation. Languages designed specifically for specifying complex tree transformations [14], which may include sophisticated pattern matching languages with, for example, second-order patterns [15], can greatly facilitate programming various transformations.

Transformations should be able to access attributes associated with the tree, since they are often conditioned on the results of program analyses, which are performed by tree attribution. Providing direct access to tree nodes and associated attributes at non-local places can save programming effort as well as run-time space, since otherwise extra attributes are needed to propagate information along the tree.

With pattern matching and instantiation, and with access to attributes, we can program various transformations, such as fold-unfold, simplification, specialization, and transformations enabled by equality analysis. Some transformations require more complicated treatment; in particular, function introduction with generalization [19] involves suspending transformation on the current tree and preparing a new subtree for recursive transformations.

Finally, a metalanguage for complex tree transformations should include a *rewrite* engine that can apply a set of transformations repeatedly to a subtree in a certain traversal order. Higher-order term rewriting [27] offers a framework for defining such rewrites. What needs to be provided is the ability to automatically interleave such repeated rewriting with incremental attribute evaluation.

2.2 External input as annotation

Program transformations are often semi-automatic and involve interaction with users or other external facilities, such as theorem provers. Input from such external sources is called *external input*. External input provides transformations with information that does not depend solely on the program tree or is too inconvenient or too expensive to compute completely from the program tree.

As program transformation generalizes symbolic, and thus normal, executions of programs, external input includes dynamic information that can be set during program executions, e.g., the breakpoints in a debugger or the instruction pointer in an interpreter. Such dynamic semantics [17] is needed for run-time support, symbolic debuggers, interpreters, as well as interactive program transformation systems.

External input scattered in the middle of program execution or transformation is a new concept lacking in traditional declarative attribute-grammar-based environments. We describe a corresponding new notion called *annotation* that fits well into the attribute-grammar-based transformation framework, preserving the declarative nature.

Annotations. Annotations should not be part of the program tree, since they do not represent terms of the subject language; nor should they be conventional attributes, since they are not determined solely by the program tree, as conventional attributes are. However, annotations should be associated with a particular position in the tree. Moreover, they should be accessible for defining attributes, since external input provides information to the analyses that guide the transformations, and the analyses are done by attribution.

To implement annotations, we can put them directly in the tree, and thus attributes can be defined solely on the tree, as before. However, some distinction between annotations and subject language terms is needed to facilitate tree pattern matching on the subject language.

An annotation may be more conveniently implemented as a special kind of attribute whose existence and, perhaps, value are determined by external input, as opposed to completely determined by the term tree or the attribution of the term tree. Of course, a user can define the value of an annotation using anything, including attributes.

The term *annotation* is used by Reiss in his FIELD environment [29] as the primary mechanism for interacting with the source file in an editor; for example, a breakpoint in the debugger is associated with an annotation in the editor. In the current system CACHET, annotations are used to store expensive attributes whose values are computed only upon user request.

2.3 Tree attribution mechanism

In addition to incremental attribute evaluation, as introduced in Section 1, a number of other tree attribution capabilities are desired to facilitate program transformations.

Circular attribute evaluation. Some program analyses are based on fixed point iteration. If such analyses are done by attribute evaluation, then circular attribute evaluation methods are needed. Although solutions to this problem have been proposed [9, 16], they are rarely implemented due to complications and inefficiency. Here, we briefly describe how circular attribute evaluation can be simulated using tree transformations and annotations.

Basically, one defines a circular attribute as a function of the program tree and an annotation, where the annotation stores the value of the circular attribute from the previous iteration, starting from bottom. A user can compare the value of the attribute and the annotation and, if they are not the same, set the annotation to be the value of the attribute, which then triggers the incremental attribution. This can be repeated until the value of the circular attribute and the annotation are equal. This iteration can be automated with rewriting.

Modular attribute evaluation. Heavy program transformations are usually composed of separate phases, where each phase conducts smaller program analyses and performs lighter transformations. Several approaches have been proposed for modular attribute specification [4, 7, 8, 10, 13] and modular attribute evaluation [10, 13].

Modular specification improves the readability of attribute grammars and allows more convenient modular evaluation. Modular evaluation provides the flexibility of turning on and off attribution modules as necessary, which results in evaluators that are speedier and more storage-efficient [10]. They are particularly useful for phase-based transformations. In particular, for phases not involving circular attributes, efficient evaluators for non-circular attribute grammars can be generated.

2.4 Replay

A minimal approach to replaying transformations is to record the *history* or *script* of external input [28]. Powerful metalanguages can help reduce the recording work [11]; for example, with a metalanguage that allows rewrite, we can record one rewrite in place of a sequence of transformations involved in the rewrite. Replay is important not only for helping to understand the whole transformation, but also for incremental transformation under changes to the input program. Whether it is feasible to achieve such incrementality in practice still needs to studied. An alternative to the direct tree manipulation framework for the purpose of replay is discussed in Section 6.

3 Implementation

A prototype system, CACHET, based on the design principles in the previous section, has been implemented. It uses the Synthesizer Generator [30], a system for generating language-based editors, and consists of about 18,000 lines of code written in SSL, the Synthesizer Generator language for specifying editors. CACHET is interactive: its flexible editor interface is generated automatically by the Synthesizer Generator.

3.1 Building in transformation

Source-to-source transformations are operations built in to CACHET. Synthesizer Generator provides the functionality of defining *transforms* with first-order pattern matching, tree mutation, and access to attributes. It has been easily used to implement all but one of the basic program transformations in CACHET, including fold-unfold, simplification, lifting a subexpression, and transformations that may be enabled by equality analyses. Some complex combinations of basic transformations, such as extensive simplification and specialization, have also been coded directly as transforms.

The one basic transformation not so easily implemented is function introduction with generalization, where, in the middle of transforming a subtree, we need to establish a new tree, switch over to transform the new tree recursively, and switch back to the original subtree when the recursion returns. A new *input* facility has been added to the Synthesizer Generator to recursively invoke new program transformation buffers for this purpose.

Currently, derivations are semi-automatic, since they are composed mostly of basic transformations that are invoked manually. Manual invocation is used mainly for two reasons. First, the Synthesizer Generator allows only subtree replacement and does not currently have a rewrite engine for a fully-automatic exhaustive application of basic transformations, in particular, for applicative-order reduction, as specified by the incrementalization approach [19]. Second, we want an interactive environment to study various transformations; thus, manual invocation is suitable most of the time.

Mechanisms for defining combinations of transformations, especially rewrite mechanisms, are being added to the Synthesizer Generator to further automate derivations. We also plan to enrich the transformation language with second-order pattern matching and easy non-local access.

3.2 Simulating annotation

Annotations are currently implemented as special parts in the program tree that are, by default, not displayed together with terms in the subject language; they can be displayed upon request by the user.

One major issue that needs to be addressed is the validity of annotations under program tree transformations and other editing operations like cut and paste. The current implementation provides special transforms to manipulate annotations: annotations whose validities are not limited to the context in the program tree can be elevated to the root of the tree; annotations that are valid only within their contexts can simply be eliminated at the user's request.

We chose this current implementation strategy because it does not require new features in the Synthesizer Generator. But we plan to extend the Synthesizer Generator to implement annotations as special attributes, provide mechanisms to specify the validity conditions of annotations, and automate the elimination, elevation, or possibly other treatments of annotations. This would make the Synthesizer Generator a more suitable tool for describing dynamic semantics with user interaction and other external input.

3.3 Using attribution

Currently, attributes in CACHET are used mainly for propagating global information, collecting context information, analyzing dependencies, and reasoning about equalities. The values of these attributes can be displayed any time, as facilitated by the Synthesizer Generator [30], to help the user understand the derivation. Of course, these attributes are evaluated incrementally after each transformation step.

Methods for circular attribute evaluation still need to be implemented, either by extending the Synthesizer Generator or using the simulation proposed in the previous section. So far, the need for modular attribute specification and implementation is not strong, but we expect that it would be very helpful when we implement the derivation approaches for exploiting intermediate results [20] and auxiliary information [21], since they are strongly phase-oriented.

We believe these implementation issues form a very promising area. Program tree attribution provides a declarative framework for program analyses, which are needed to guide powerful program transformations. Techniques for managing the interactions among tree transformation, external input, and incremental attribution can be used to generate powerful and incremental transformation systems.

4 Example derivation

CACHET is designed specifically for deriving incremental programs. It has been used for most of the examples in [19, 20, 21].

The programs transformed by CACHET are written in a first-order functional language, where expressions in function definitions are composed of variables, data constructions, primitive function applications, user-defined recursive function applications, conditional expressions, and binding expressions. An example is given in the back window of Figure 1. It defines a function `sort`, which does selection sort, and two auxiliary functions `least` and `rest`. The input change operation is adding a new element `i` to an old input `x` to form a new input `cons(i,x)`. We will derive an incremental program `sort'` that sorts `cons(i,x)` by inserting `i` into the sorted list `sort(x)`.

The basic idea for incrementalization is to transform `sort(cons(i,x))` and replace subcomputations whose values are retrievable from the value of `sort(x)` with corresponding retrievals. The derivation introduces incremental functions to compute function applications, puts them in a *definition set*, and uses them to replace the original applications. To obtain the definition of such an introduced function, the derivation unfolds the corresponding function application, collects context information, and simplifies subexpressions. It also finds subcomputations, in their respective contexts, whose values are retrievable from the cached result of the previous computation, puts them in the corresponding *cache sets*, and replaces occurrences of such subcomputations with corresponding retrievals.

Given a set of `FUNCTION DEFINITIONS`, the `FUNCTION TO BE EVALUATED`, the `OLD INPUT TO THE FUNCTION`, and the `NEW INPUT TO THE FUNCTION`, the `EXPRESSION TO BE TRANSFORMED` is initialized by a transform `init-tran-exp`. We select the expression to be transformed, shown by the underlined expression in the back window of Figure 1. Relevant transforms for the current selection are displayed in buttons at the bottom of the window.

Transforms. A transform ending with `*` is a basic transformation that preserves correctness. In particular, a transform ending with `$*` uses the cached result of the previous computation. A transform ending with `!` is a combination of correctness-preserving basic transformations, and one ending with `?` is a basic transformation that may not preserve correctness (enabled for experimentation).

Transforms starting with `.` deal with annotations. In particular, transform `.f_e(fun-intr-repl)*` in-

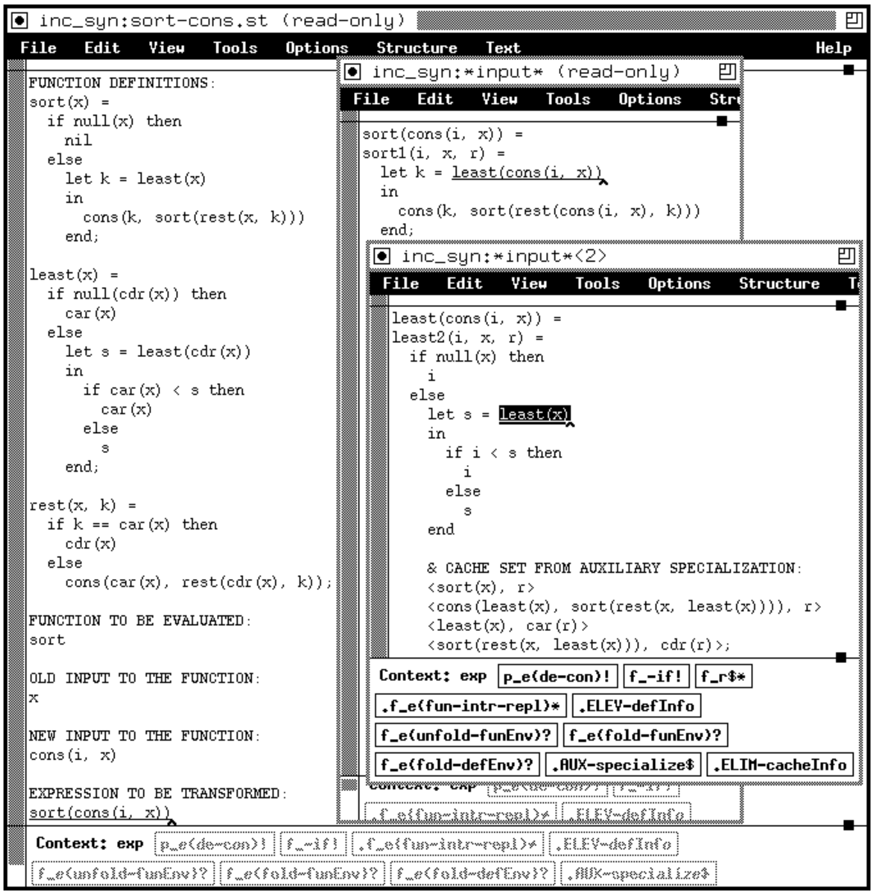

Figure 1: Derivation of the example

troduces a new function to compute the current function application or replaces the current function application with a previously introduced function; any change to the definition set leaves the new set as an annotation at the current selection. Transform $\boxed{\texttt{.AUX-specialize\$}}$ uses an auxiliary specializer to extend, at the current selection, the set of subcomputations whose results are retrievable from the cached result of the previous computation; the resulting cache set is left as an annotation at the current selection. Other transforms starting with `.` but with no special ending symbols manipulate annotations by elevation or elimination.

Names of transforms attempt to be illustrative. Names starting with `c`, `p`, and `f` are for applications of constructors, primitive functions, and user-defined functions, respectively. Names starting with `if` and `let` are for conditional and binding expressions, respectively. Subscript `_e` denotes a transformation like simplification, and `_r` denotes a replacement with a cached result. The strings in parentheses are for further illustration, e.g., `(de-con)` denotes destructing a construction, `(unfold-funEnv)` denotes unfolding a function in the given set of functions, and `(fold-defEnv)` denotes folding into an introduced function in the definition set. Subscript `_n` refers to the nth subexpression; `-if` and `-let` lift the condition and the binding, respectively, of the nth subexpression out of the enclosing expression; omitting a subscript `n` implies lifting all conditions or bindings.

Function introduction. At the underlined selection in the back window of Figure 1, clicking on transform $\boxed{\texttt{.f\_e(fun-intr-repl)*}}$ suspends transformation in the current window, pops up a new window, the middle window of Figure 1, and switches over for recursive transformations that will yield a function `sort1` to replace the application `sort(cons(i,x))`. Note that a fresh identifier `r` is used as parameter for the cached result of `sort(x)`.

Unfolding and simplification. To obtain a definition of `sort1`, in the middle window, we unfold the application `sort(cons(i,x))` and apply simplifications $\boxed{\texttt{p\_e(null-cons)*}}$ (transforming `null(cons(i,x))` to `false`) and $\boxed{\texttt{if\_e(cond-false)*}}$ (transforming `if` `false then e1 else e2` to `e2`). We obtain what is shown in the middle window. Then, recursively, we introduce a function `least2`, in the front window of Figure 1, to compute `least(cons(i,x))`. After unfolding and applying simplification `p_e(de-con)!` (performing all destructions of constructions), we obtain the upper part of the front window.

Auxiliary specialization. The lower part of the front window shows an *annotation*, namely, the cache set at the branch where `null(x)` is false. It is the result of applying auxiliary specialization to `sort(x)` with `null(x)` being false at that branch, and is obtained by clicking `.AUX-specialize$` at the bottom of the window and specializing `sort(x)` in another window (killed upon returning the displayed cache set). The cache set indicates that the value of `sort(x)` is `r`, the value of `cons(least(x),sort(rest(x,least(x))))` (specialized `sort(x)` when `null(x)` is false, by definition of `sort`) is `r`, the value of `least(x)` is `car(r)`, *etc.*

Replacement. With the displayed cache set, when we select the underlined and highlighted expression `least(x)` in the front window, we find that rule `f_r$*` applies. Clicking on it replaces `least(x)` with `car(r)`. Similarly, the boolean expression `null(x)` can be replaced with `null(r)`, essentially because `null(x)` is true if and only if `null(sort(x))` is true.

Function replacement. The above replacements yield the definition of function `least2(i,x,r)` shown on the left of Figure 2. Then, we return to resume the transformation for `sort1`, and we replace the application `least(cons(i,x))` with `least2(i,x,r)`.

```
if null(r) then
  i
else
  let s = car(r)
    in
      if i < s then
        i
      else
        s
    end
```

```
if s == i then
  x
else
  cons(i, rest(x, s))
```

Figure 2: Intermediate function definitions

Application `least2(i,x,r)` is unfolded in place, since `least2` is not recursively defined. We then lift the conditionals and binding in the definition of `least2` out of the enclosing expression, simplify the resulting binding expressions, and obtain the result shown in Figure 3(a).

Then, similarly, we introduce a function to compute `rest(cons(i,x))` in the first branch, replace it, and unfold to obtain just `x`. With this, the enclosing expression becomes `sort(x)` and can be replaced with `r`. The next occurrence of `rest(cons(i,x))` can use the function just introduced and obtain `x` directly. Again, the enclosing `sort(x)` is replaced with `r`. For `rest(cons(i,x),s)` in the last branch, introducing a function, replacing, and unfolding, we obtain the result shown on the right of Figure 2. Lifting the condition out of the enclosing applications of `sort` and `cons`, we obtain the result shown in Figure 3(b).

Next, `sort(x)` is replaced with `r`, and the two branches with conditions `i<s` and `s==i` are merged, yielding `if i<=s then cons(i,r)`. For the last branch, `sort(cons(i,rest(x,s)))` is replaced by recursive call `sort1(i,rest(x,s),sort(rest(x,s)))`. With the knowledge of context information `s = car(r) = least(x)`, and using the cache set displayed in the front window in Figure 1, we see that `sort(rest(x,s)) = sort(rest(x,least(x))) = cdr(r)`. Thus, we obtain the result shown in Figure 3(c).

Dead code elimination. To complete the example, we need to eliminate dead code in the above definition. In particular, the second parameter of `sort1` is dead and should be eliminated. This is not implemented yet. It can be done using traditional compiler technologies, after which we would obtain the result shown in Figure 3(d), which is exactly an insertion. Finally, the original application `sort(cons(i,x))` can be replaced with `sort'(i,r)`, where `r = sort(x)`.

The derivation above is performed by manually selecting the subexpression and clicking on one of the enabled transforms. After we implement the rewrite engine, we will be able to automate the derivation using an applicative-order rewrite [19]. However, to see all the interesting intermediate results during the derivation, manual invocation is appropriate.

5 Related work

Program transformation systems and the approaches and techniques used therein are described in a number of surveys, e.g., [12, 26]. Eminent systems among them and recent systems include APTS [24], KIDS [31], CIP [3], Focus [28], and ZAP [11].

Compared to these systems, the most important and unique characteristic of CACHET is its use of attribute grammars. This has at least two advantages. First, a program transformation system based on program analysis is a complex constraint system; the attribute grammar paradigm provides a declarative framework where constraints can be specified and consistency can be maintained in a clean way. Second, incremental program analysis is important for speeding up the overall program transformation process; using attribute evaluation for program analysis allows us to advantage of known techniques for incremental attribute evaluation. To our knowledge, none of these other systems use incremental attribute evaluation, although incremental semantic analysis is desirable in all of them, especially for the more automatic approaches in APTS [24].

Also, since CACHET is implemented using a language-based editor generator, it has a flexible interactive user interface, as is provided by the existing technologies of program environments. Among the systems above, KIDS [31] seems to be the only one with such a flexible user interface. Of course, im-

(a)

```
  sort1(i, x, r) =
    if null(r) then
      cons(i, sort(rest(cons(i, x), i)))
    else
      let s = car(r)
      in
        if i < s then
          cons(i, sort(rest(cons(i, x), i)))
        else
          cons(s, sort(rest(cons(i, x), s)))
      end;
```

(b)

```
sort1(i, x, r) =
  if null(r) then
    cons(i, r)
  else
    let s = car(r)
    in
      if i < s then
        cons(i, r)
      else
        if s == i then
          cons(s, sort(x))
        else
          cons(s, sort(cons(i, rest(x, s))))
    end;
```

(c)

```
sort1(i, x, r) =
  if null(r) then
    cons(i, r)
  else
    let s = car(r)
    in
      if i <= s then
        cons(i, r)
      else
        cons(s, sort1(i, rest(x, s), cdr(r)))
    end;
```

(d)

```
sort'(i, r) =
  if null(r) then
    cons(i, r)
  else
    let s = car(r)
    in
      if i <= s then
        cons(i, r)
      else
        cons(s, sort'(i, cdr(r)))
    end;
```

Figure 3: Derivation of the example (continued)

plementing CACHET as a programming environment also allows it to be easily integrated with other facilities in programming environments, such as compilers, debuggers, and interpreters.

CACHET can benefit from a stronger metalanguage, such as that pioneered by ZAP [11], and certain replay functionality, such as that in Focus [28]. As discussed in Sections 2 and 3, a metalanguage for powerful tree transformation has been designed and is being implemented for CACHET. How to integrate replay and incremental replay in an attribute grammar framework with annotation is a problem to be studied.

How does CACHET compare to traditional programming environments that do dataflow analysis and code generation? First, if such an environment is viewed as a program transformation system based on program analysis, then CACHET is more general in that it allows interleaving program transformations with annotations of external inputs. Techniques that address such dynamic program semantics have been lacking in traditional attribute-grammar-based programming environments [17]. For example, OPTRAN [18] is an attribute-grammar-based system extended with rewrite mechanisms. However, it is still a batch-oriented system mainly for compiler applications rather than general program transformations.

Another difference is that traditional attribute-grammar-based programming environments perform code generation by attribution [4, 13, 30], while CACHET transforms programs by direct manipulation. How to do program transformation by attribution in the presence of annotation is related to incremental replay, as discussed in Section 6.

Finally, all of the above systems are for transformations from specifications to programs, or from programs to more efficient programs, whereas CACHET has the special functionality of deriving incremental programs. This functionality provides a general solution to the finite differencing problem, which must be addressed in program derivation from specification and program improvement in general [3, 24, 25, 31]

6 Future work

A number of problems need to be further studied. We discuss major ones here.

Annotation. Annotation provides a *declarative* framework for describing dynamic program semantics. However, formal description of annotation in the context of attribute-grammars is needed. An example of a question to be answered is: while attributes are associated with non-terminals and attribute equations with productions, should annotations be associated with productions and/or non-terminals?

Regarding the validity of annotations under tree transformation or other editing operations, we anticipate a number of issues. Should annotations be associated with tree nodes or locations in the tree? When two valid annotations are present at one place as a result of tree rewrite, should one overwrite the other or should they be merged; if the former, which overwrites which; if the latter, how? Should invalid annotation be tagged as invalid or simply removed? One way to answer these questions is to treat annotations like attributes, and thus treat the validity as attribute reevaluation but perhaps simpler.

Finally, we need to study incremental algorithms that combine annotation validation with attribute evaluation. For example, if we treat validity as attribute reevaluation, we need to first check well-definedness of the attribute grammar when repeated tree rewrite is allowed.

Incremental replay. A potential alternative for accommodating replay is to change our basic framework and conduct transformation by attribution instead of direct tree manipulation. Thus, tree attribution is used not only for semantics analysis, but also for recording transformed versions of the program.

Attribution has been traditionally used in programming environments for code generation [30]; it has also been proposed for general program transformation, including phase-based transformation. Approaches include attribute coupled grammars [13], higher-order attribute grammars [32], composable attribute grammars [10], and simple tree attributions [4]. Incremental attribute evaluation algorithms for these frameworks could be used for incremental code generation. However, these frameworks do not address external input. We need to extend them and study how annotations should be incorporated with incremental attribute evaluation for incremental program transformation.

Acknowledgements

The author would like to thank Tim Teitelbaum for numerous discussions and other help in implementing CACHET. Scott Stoller has helped clarify many thoughts. The author also thanks Tim Teitelbaum, Thomas Yan, Aswin van den Berg, and David Pierce for discussions about implementing attribution-based program transformation systems.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The *Pan* language-based editing system. *ACM Trans. on Software Eng. and Methodology*, 1(1):95–127, Jan. 1992.

[3] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations—computer-aided, intuition-guided programming. *IEEE Trans. on Software Eng.*, 15(2):165–180, Feb. 1989.

[4] J. Boyland and S. L. Graham. Composing tree attributions. In *Proc. of the 21th Ann. ACM Symp. on POPL*, pages 375–388, Portland, OR, Jan. 1994.

[5] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, Nov. 1977.

[6] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars: Definitions, Systems, and Bibliography*, vol. 323 of *LNCS*. Springer-Verlag, 1988.

[7] G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *The Computer Journal*, 33(2):164–172, Apr. 1990.

[8] C. Farnum. Pattern-based tree attribution. In *Proc. of the 19th Ann. ACM Symp. on POPL*, pages 211–222, Jan. 1992.

[9] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proc. of the ACM SIGPLAN '86 Symp. on Compiler Construction*, pages 85–98, Jul. 1986.

[10] R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *Proc. of the 19th Ann. ACM Symp. on POPL*, pages 223–234, Jan. 1992.

[11] M. S. Feather. A system for assisting program transformation. *ACM Trans. on Programming Languages and Systems*, 4(1):1–20, Jan. 1982.

[12] M. S. Feather. A survey and classification of some program transformation approaches and techniques. In *Program Specification and Transformation*, pages 165–195. North-Holland, 1987.

[13] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proc. of the ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 157–170, Jun. 1984.

[14] R. Heckmann. A functional language for the specification of complex tree transformations. In *Proc. of the 2nd ESOP*, vol. 300 of *LNCS*, pages 175–190, France, Mar. 1988.

[15] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11(1):31–55, 1978.

[16] L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Trans. on Programming Languages and Systems*, 12(3):429–462, Jul. 1990.

[17] G. E. Kaiser. Incremental dynamic semantics for language-based programming environments. *ACM Trans. on Programming Languages and Systems*, 11(2):168–193, Apr. 1989.

[18] P. Lipps, U. Möncke, and R. Wilhelm. OPTRAN: A language/system for the specification of program transformation—system overview and experiences. In *Proc. of the 2nd Workshop on Compiler Compilers and High Speed Compilation*, vol. 371 of *LNCS*, pages 52–65, Berlin, Oct. 1988.

[19] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, Feb. 1995.

[20] Y. A. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In *Proc. of the ACM SIGPLAN Symp. on PEPM*, pages 190–201, La Jolla, CA, Jun. 1995.

[21] Y. A. Liu and T. Teitelbaum. Incremental computation for transformational software development. TR 95-1499, Cornell University, Mar. 1995.

[22] R. Medina-Mora and P. Feiler. An incremental programming environment. *IEEE Trans. on Software Eng.*, SE-7(5):472–482, Sept. 1981.

[23] S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[24] R. Paige. Viewing a program transformation system at work. In *Proc. of Joint 6th Intl. Conf. on PLILP and 4th Intl. Conf. on ALP*, vol. 844 of *LNCS*, pages 5–24, Sept. 1994.

[25] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. on Programming Languages and Systems*, 4(3):402–454, Jul. 1982.

[26] H. Partsch and R. Steinbrüggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, Sept. 1983.

[27] L. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[28] U. Reddy. Transformational derivation of programs using the Focus system. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments*, pages 163–172, 1988.

[29] S. P. Reiss. Interacting with the FIELD environment. *Software—Practice and Experience*, 20(S1):89–115, Jun. 1990.

[30] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, 1988.

[31] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. on Software Eng.*, 16(9):1024–1043, Sept. 1990.

[32] H. H. Vogt, D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proc. of the ACM SIGPLAN '89 Conf. on PLDI*, pages 131–145, Jun. 1989.