GUSTT: An Amorphous Slicing System which Combines Slicing and Transformation

Mark Harman,
Lin Hu,
Brunel University,
Uxbridge, Middlesex,
UB8 3PH, UK.
Tel: +44 (0)1895 816 237
Fax: +44 (0)1895 251 686
Mark.Harman@brunel.ac.uk

Malcolm Munro,
Xingyuan Zhang,
University of Durham,
South Road, Durham,
DH1 3LE, UK.
+44 (0) 191 374 2634
+44 (0) 191 374 2560
Malcolm.Munro@durham.ac.uk

Keywords: Amorphous Slicing, Transformation, Side Effects, Coq Proof Assistant

Abstract

This paper presents a system for amorphous program slicing which combines slicing and transformation to achieve thinner slices than are possible using conventional syntax-preserving slicing. The approach involves the validation of the transformation and slicing steps using the Coq *proof assistant, thereby guaranteeing the correctness of the amorphous slices produced. The combined application of slicing and transformation is illustrated with a simple case study.*

Several components of the system implement transformation tactics, such as side-effect removal and dependence reduction transformations which have wider applications than the construction of amorphous slices.

1 Introduction

This paper introduces the GUSTT[1] amorphous slicing system. The GUSTT system contains slicing and transformation components which are applied together to implement amorphous slicing. Amorphous slicing is a variation of the traditional slicing theme, which retains the semantic properties of traditional slicing while abandoning the syntactic requirements. This leads to thinner slices and allows for the combination of traditional slicing and traditional transformation.

Traditional approaches to slicing have been *syntax preserving.* That is, they are constructed by the sole transformation of statement deletion. The statements which remain in the slice are therefore a syntactic subset of the original program from which the slice was constructed. Traditional approaches to transformation have been fully meaning preserving. That is, the program's syntax is altered but its behaviour remains invariant during the transformation process.

Transformations which tend to reduce program size are like slices in that they preserve the semantics of the program with respect to the slicing criterion, but are unlike slicing because they do not preserve syntax. On the other hand, slicing is like transformation in the way it preserves the semantics of the program for some projection (onto the variable of interest) of the original semantics. When slicing and transformation are combined, the result is amorphous slicing.

Amorphous slices [4, 14] are constructed using any program transformation which simplifies the program and which preserves the effect of the program with respect to the slicing criterion. This syntactic freedom allows amorphous slicing to perform greater simplification with the result that amorphous slices are never larger than their syntax-preserving counterparts. Often they are considerably smaller.

The idea of improving slicing using transformations on an underlying intermediate representation was first suggested by Tip [27] and Ernst [12]. The idea of using these transformations to produce amorphous slices was suggested by Harman and Danicic [14].

Amorphous slicing can be thought of as a generalization of slicing: the semantic requirement is retained, but the syntactic requirement is relaxed.

[1] GUided Slicing and Targeted Transformation.

Original and Traditional slice	Traditional Transformation	Amorphous slice

`D = 2*r;`
`FaceArea = pi*r*r;`
`C = pi*D;`
`SurfaceArea = 2*FaceArea+h*C;`
	`SurfaceArea=2*pi*r*r+h*pi*2*r;`
	
	`FaceArea = pi*r*r;`
	`C = pi*2*r;`
	`D = 2*r;`
		`SurfaceArea=2*pi*r*r+h*pi*2*r;`

Figure 1: Amorphous Slicing Produces Thinner Slices by Removing Syntactic Restrictions

This makes amorphous slicing more like transformation than traditional slicing. Similarly, amorphous slicing can be thought of as a generalization of transformation: the semantic requirement to preserve the meaning of the program is relaxed to the requirement that the (projected) semantics *with respect to the slicing criterion* is persevered.

Amorphous slicing allows a slicing tool to 'wring out' the semantics of interest, using transformation to reduce the dependencies in the original program and then using traditional syntax preserving slicing to extract the components of the computation which have been isolated by transformation. Consider the simple program fragment in the leftmost Section of Figure 1. This program computes values of interest for a cylinder of radius `r` and height `h`. The computations of each value are 'built up' in terms of previous values creating dependencies between the statements. This inter-dependence inhibits the simplification abilities of traditional, syntax-preserving slicing. For example, the syntax-preserving slice on the final value of the variable `SurfaceArea` contains the whole program.

Consider the version of the original program in the central section of Figure 1. This version of the program has been produced using a simple code motion transformation (Rule 2 of Figure 4). The rule pushes an assignment forward in the program, unfolding the expressions past which the assignment moves. The effect of this transformation is meaning preserving, but it breaks the interdependence which inhibits traditional slicing. For example, in the example in Figure 1, the variable `SurfaceArea` depends upon all four assignment statements; in the 'push transformed' version it depends on only one.

Having removed dependencies in the program, traditional slicing can now be employed to cut out the parts of the program of interest. The result is an amorphous slice, because the assignment to `SurfaceArea` is not present in the original program. The amorphous slice retains the important semantic property of traditional slicing that the behaviour of the original with respect to the slicing criterion remains unaffected.

Dependence reduction is not the only way in which transformation can be used to improve slicing, it is simply one of the most obvious ways in which transformation can improve slicing, once the requirement of syntax-preservation is removed. For example, the amorphous slice on the variable `SurfaceArea` in Figure 1 could be improved by expression simplification, which could simply the expression

```
SurfaceArea=2*pi*r*r+h*pi*2*r;
```

with the expression

```
SurfaceArea=2*pi*r*(r+h);
```

This paper describes the implementation of an amorphous slicing tool that mixes (domain independent) dependence reduction transformation with other (domain specific) transformations, preprocessing transformation and traditional slicing to achieve amorphous slicing. The paper describes the architecture of the overall system, and the dependence reduction transformation component in detail, and illustrates how each phase combines to produce simpler slices using a case study.

Some of the transformation tactics used by the GUSTT system may also find application in other transformation programs. For example the side effect removal tactic removes intraprocedural side effects, while the dependence reduction transformation tactic reduces the inter-dependence between statements. The former may be useful as an enabling step to other transformation-based approaches [2, 6, 23, 29], while the latter may be useful in work on transformation-based automatic parallelization [21, 22, 26, 33].

The rest of the paper is organised as follows: Section 2 introduces the architecture of the GUSTT system. The Side Effect Removal (SER) and Dependence Reduction Transformation (DRT) components are described in sections 3 and 4, respectively. The combined effect of the system's components are illustrated with a simple case study in Section 5. Section 6 provides a brief overview of the application of

Coq to the verification of GUSTT transformations. Section 7 concludes and section 8 provides directions for future work.

2 The Amorphous Slicing System

Figure 2 describes the architecture of the amorphous slicing system developed as part of GUSTT project.

The criteria selection component is currently entirely human based. The ultimate goal of this research is to partly automate the criterion selection component of the system to produce guidance to the human in selecting good slicing criteria.

The system's overall correctness is partly automated using human generalized/system checked proofs in constructive type theory.

The system involves the pre-processing of original programs, for example side-effect removal and control flow restructuring. The goal is to produce a version of the program written in a form which is essentially 'functional with sequencing'. This eases further transformation, as transformation rules are easier to define for a functional language [1, 11, 19, 24].

The central amorphous slicing system is defined for such a 'functional language with sequencing', using a subset of the C language which closely resembles WSL [6, 30]. The pre-processing phase can be thought of as a translator which translates a larger language into the functional subset in a manner reminiscent of that envisaged by Landin [20]. As the capabilities of the pre-processor are enhanced, so the set of programs handled by the system increases, without the requirement to alter the central 'engine of transformation'. The pre-processor includes a side effect removal transformation tactic and the restructuring transformations used in the Maintainers Assistant [31] project and are currently being incorporated.

The central amorphous slicer contains three subcomponents. The three components are iteratively applied until no further reduction in amorphous slice size is possible.

The first component is a traditional syntax-preserving slicer [10].

The second component of the GUSTT system, the DRT tactic, consists of a set of transformations aimed at reducing statement inter-dependence. This phase of the amorphous slicing engine is 'general purpose' because the reduction of inter-dependence between statements will always tend to reduce the size of slices and all applications require slices to be as small as possible. The dependence reduction component of the GUSTT system is described in more detail in Section 4.

The third component of the approach seeks additional transformation rules for domain specific problems. Domain Specific Transformation (DST) exploits knowledge of the application domain in order to improve the chances of achieving slice size reduction. Earlier work [16] has shown considerable improvement in simplification by DST. The DST component is essentially an opportunistic phase of the approach. Different applications will require different 'transformation rule sets' tailored to the problem in hand. These transformations are called 'targeted transformations' because they are targeted at a particular problem.

Underlying all the central phases of the amorphous slicer is the approach to correctness assurance. In the GUSTT system this is a partly automated formal proof process. The Coq proof assistant [9, 25, 7, 8] is used to express and check proofs of correctness for the slicing and transformation components of the system. Work is underway to extend the application of Coq to the pre-processing phases of the system.

3 Side Effect Removal

A side effect is any change in program state that occurs as a by-product of the evaluation of an expression. A side-effect free expression, when evaluated simply returns its value, causing no change in state. Side effects tend to inhibit transformation, and so they are removed in a pre-processing phase of the system. Figure 3 provides several examples of code fragments with side effects and their side-effect free versions.

Side Effect Removal Transformation (SERT) rewrites a program p which may contain side effects into a semantically equivalent program p' which is guaranteed to be side-effect free. Software engineering techniques such as slicing, program transformation and other program analysis techniques can benefit from SERT. It is one of the enabling steps in GUSTT as well.

SERT is achieved using the Linsert system [15]. Linsert uses a very simple symbolic executor to determine a side-effect free version of the expression. Since only expression operators, sequence points and conditionals are involved, the AST of an expression is an acyclic graph. The absence of cycles means that the usual problems which loops present to symbolic executors disappear, and it becomes possible to completely determine symbolic values in all cases.

4 Dependency Reduction Transformation

Amorphous slicing allows the power of program transformation to be used to achieve further simplification over and above that achieved using tra-

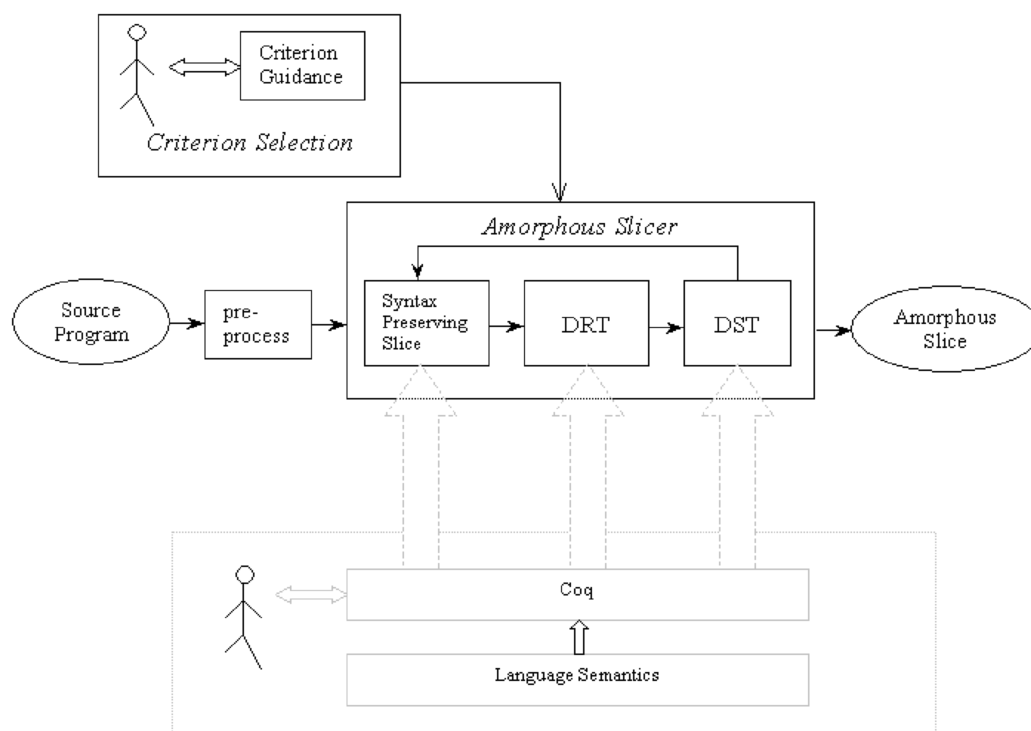

Figure 2: The architecture of the overall system

<table>
<tr><td><code>a = ++x + x;</code></td><td><code>undefined</code></td></tr>
<tr><td><code>b = (++x, ++x);</code></td><td><code>b = x+2; x = x+2;</code></td></tr>
<tr><td><code>c = ++x + ++x;</code></td><td><code>undefined</code></td></tr>
<tr><td><code>d = ++x && ++x;</code></td><td><code>if (x == -1) { d = 0; x = 0; }</code>
<code>else { d = x+2!=0; x = x+2; }</code></td></tr>
<tr><td><code>f = (x=4) + (y=5);</code></td><td><code>f = 9; x = 4; y = 5;</code></td></tr>
<tr><td><code>g = (x=4) + (x=5);</code></td><td><code>undefined</code></td></tr>
<tr><td>Original</td><td>Side-Effect Free Version</td></tr>
</table>

Figure 3: Side-effecting Statements with Side-Effect Free Versions

Axiom 1 **if(e){} else c** $\Rightarrow$ **if(!e) c**

Axiom 2 **if(e){ $\mathbf{c_1}$ c } else{ $\mathbf{c_2}$ c }** $\Rightarrow$ **if(e){ $\mathbf{c_1}$ } else{ $\mathbf{c_2}$ } c**

Axiom 3 **while(e){ if(e){ $\mathbf{c_1}$ } $\mathbf{c_2}$ }** $\Rightarrow$ **while(e){ $\mathbf{c_1}$ $\mathbf{c_2}$ }**

Axiom 4 **if(0){ c }** $\Rightarrow$ **{ ; }**

Axiom 5 **if(0){ $\mathbf{c_1}$ } else{ $\mathbf{c_2}$ }** $\Rightarrow$ **{ $\mathbf{c_2}$ }**

Rule 1 (Unfold Assignment)

$$\frac{e_3 = \mathbf{SUB}(e_2, i, e_1)}{[\![i = e_1;\ i = e_2;]\!] \Rightarrow [\![i = e_3;]\!]}$$

Rule 2 (Push Assignment)

$$\frac{i_1 \neq i_2, i_2 \notin \mathrm{REF}(e_1), e_3 = \mathrm{SUB}(e_2, i_1, e_1)}{[\![i_1 = e_1;\ i_2 = e_2;]\!] \Rightarrow [\![i_2 = e_3;\ i_1 = e_1;]\!]}$$

Rule 3 (Push If)

$$\frac{e_2' = \mathbf{SUB}(e_2, i, e_1), [\![i{=}e_1;\ c]\!] \Rightarrow [\![c'\ i{=}e_1;]\!]}{[\![i{=}e_1;\ \mathtt{if}\ \ (e_2)\ \ c]\!] \Rightarrow [\![\mathtt{if}\ \ (e_2')\ \ c'\ i{=}e_1;]\!]}$$

Figure 4: A subset of DRT rules

ditional syntax-preserving slicing. In producing an amorphous slice, some of the transformation tactics applied will necessarily be domain specific. These are the targeted transformations implemented by the DST component of the system. Before targeted transformation is applied, general transformation rules which reduce dependence are used to improve the chances of size reduction using traditional syntax-preserving slicing[2]. This kind of transformation is called Dependency Reduction Transformation (DRT).

A set of 'push' code motion transformation rules play important role in reducing dependencies. A subset of these DRT transformation rules is presented in Figure 4, where the term $\mathbf{SUB}(e_2, i, e_1)$ returns the expression that results from substituting all occurrences of the variable i in the expression e_2, with the expression e_1 and *REF* is a function which returns the referenced variables of a (side-effect free) expression.

The example in Figure 5 illustrates the application of the push rules. From program p, repeating the push rule transformation, we obtain p', finally p''. In program p, variable `z` is explicitly dependent on `x` and `y` (suppose `a` and `b` are symbolic constants), after transforming, variable `z` is only dependent on the initial value of `x`.

y=x+a;
x=b+1;
z=x+y;
	y=x+a;
	z=b+1+y;
	x=b+1;
		z=b+1+x+a;
		y=x+a;
		x=b+1;

p	p'	p''

Figure 5: Applying push rule to reduce dependence

With respect to variable `z`, the traditional, syntax preserving slice of program p is

```
y=x+a;
x=b+1;
z=x+y;
```

From transformed program p'', an amorphous slice of p is obtained by traditional syntax preserving slicing:

```
z=b+1+x+a;
```

Notice how the push transformations, combined with the application of traditional, syntax preserving

[2]The syntax preserving slice phase precedes the DRT phase in Figure 2 for efficiency reasons. That is, slicing reduces the amount of code to be considered by DRT and DST. The iterative nature of the amorphous slicer means that DRT is also followed by a syntax-preserving slicing phase.

slicing, produces greater simplification than traditional slicing alone. The overall effect is that expressions become more complex, but statements become more interdependent, so slices tend to be smaller (or at least have fewer nodes) and tend to assign to fewer variables.

5 A Simple Case Study

This section illustrates the GUSTT system using the example in Figure 6. The example is supposed to compute the mean values for the numbers in the array `A` and the means for the even and odd numbers of the array. These values are used to compute the difference between the overall mean and the mean for evens and odds.

However, the program contains a bug. The bug is that the assignment operator = has been used in place of the equality test operator == in testing a flag variable `evenflag` which is used to denote the outcome of testing whether or not a particular element of the array is even or not.

It has been previously observed [3] that transformation has a tendency to highlight bugs in programs, *because* the transformation is meaning preserving and yet it renders the program in a different syntax. This worked example shows how the combination of slicing *and* transformation can further improve transformation's bug-revealing tendency.

The rest of this section steps through the application of the various components of the GUSTT system, showing how each affects the example.

After pre-processing, the side-effects in program P_0 are removed. The side-effect free program P_1 (shown in Figure 7) is obtained. Not surprisingly, as the bug involves the mistaken use of a side effect where a side-effect free expression was intended the side-effect removal tactic makes the bug more obvious. That is, the side effect free version of the program contains the tell-tale conditional

```
if (0)
```

This is odd, because it clearly can only evaluate to `false`, making the `then` branch of the conditional redundant. As will be shown, further transformation and slicing, make this bug even more evident.

Slicing (conventionally) program P_1 on the variable `evendifference`, a conventional slice S_0 is computed.

Traditional syntax-preserving slicing [17, 28, 32] does not exploit the fact that `if(0)` can never execute the `then` branch (line 9 and 10 in Figure 8). However, Tip [27] and Ernst [12] show how compiler optimization techniques can be used to transform an intermediate representation that would allow optimized syntax preserving slices to be constructed.

```
total=0;  i=0;
evens=0;  noevens=0;
odds=0;   noodds=0;
scanf("%d",&n);
while (i<=n){
  evenflag = A[i] % 2;
  if(evenflag=0)       /* bug */
    {evens = evens + A[i];
     noevens++; }
  else
    {odds = odds + A[i];
     noodds++; }
  total = total + A[i++];
}
if(noevens!=0) meaneven=evens/noevens;
else meaneven = 0;
if(noodds!=0) meanodd = odds/noodds;
else meanodd = 0;
meanodd = odds/noodds;
mean = total/++n;
evendifference = abs(meaneven-mean);
odddifference = abs(meanodd-mean);
```

Figure 6: Program P_0 to be sliced amorphously with respect to the variable `evendifference` at the end of program

Such optimizations are included in the DRT rule set if they reduce dependence. For example, Axiom 4 and Axiom 5 in Figure 4 are standard redundant computation removal optimizations.

Applying Axiom 4 to the code fragment in Figure 8 causes the code from lines 8 to 10 of S_0 to be deleted. Applying Axiom 5, the code line 17 and 18 of S_0 are transformed into

```
meaneven = 0;
```

The transformed slice S_1 is shown in Figure 9. Note that the dependence on variables `evens` and `noevens` (in line 2) has been broken after the above DRT transformations are applied.

Applying the DRT algorithm to the code fragment from lines 18 to 24, the transformed slice S_2 shown in Figure 10 is obtained.

Slicing S_2 on the variable `evendifference`, the amorphous slice (in Figure 11) is obtained. In the amorphous slice the bug manifests itself as the fact that the assignment to `evendifference` is defined in terms of the number of elements in the array, `n`, and that no mention of the variables `evens` and `noevens` remains. This makes the presence of the bug very evident. In general, amorphous slicing will remove

```
1   total=0;  i=0;
2   evens=0;  noevens=0;
3   odds=0;   noodds=0;
4   scanf("%d",&n);
5   while(i<=n){
6     evenflag = A[i] % 2;
7     evenflag=0;
8     if(0)      /* bug */
9       {evens = evens + A[i];
10       noevens = noevens + 1; }
11    else
12      {odds = odds + A[i];
13       noodds = noodds + 1; }
14    total = total + A[i];
15    i = i + 1;
16  }
17  if(noevens!=0) meaneven=evens/noevens;
18  else meaneven = 0;
19  if(noodds!=0) meanodd = odds/noodds;
20  else meanodd = 0;
21  meanodd = odds/noodds;
22  mean = total/(n+1);
23  n = n+1;
24  evendifference = abs(meaneven-mean);
25  odddifference = abs(meanodd-mean);
```

Figure 7: Side-effect free program P_1

```
1   total=0;  i=0;
2   evens=0;  noevens=0;
3
4   scanf("%d",&n);
5   while(i<=n){
6
7
8     if(0)      /* bug */
9       {evens = evens + A[i];
10       noevens = noevens + 1; }
11
12
13
14    total = total + A[i];
15    i = i + 1;
16  }
17  if(noevens!=0) meaneven=evens/noevens;
18  else meaneven = 0;
19
20
21
22  mean = total/(n+1);
23
24  evendifference = abs(meaneven-mean);
25
```

Figure 8: Conventional slice S_0 w.r.t the variable `evendifference` at the end of program P_1

as much of the semantics of the program as possible, creating the simplest possible program (subject to the usual computability concerns) which retains the effect of the original on the variable of interest. The chances of fault revelation are thereby increases.

6 Verification Using `Coq`

The role of the proof assistant `Coq` [9, 25, 7, 8] plays is to verify the correctness of slicing and transformation steps. The semantics of the programming language is formalized in the style of natural semantics[18]. The semantics of statements is defined as an inductive relation $- \stackrel{-}{\Longrightarrow} -$, where for program state s_1, s_2 and statement st,

$$s_1 \stackrel{st}{\Longrightarrow} s_2$$

means the execution of st starting in state s_1 will end is state s_2. And then the equivalence relation $\approx$ between two statements st_1, st_2 can be defined as:

$$st_1 \approx st_2 \stackrel{\text{def}}{=} \forall s_1, s_2.\ (s_1 \stackrel{st_1}{\Longrightarrow} s_2 \leftrightarrow s_1 \stackrel{st_2}{\Longrightarrow} s_2)$$

Using $\approx$ the correctness of slicing and transformation rules can be expressed. For example, the correctness of Axioms 1, 2 and 3 in Figure 4 can be expressed as:

$$\mathbf{if(e)\{\}\ else\ c} \ \approx\ \mathbf{if(!e)\ c} \qquad (1)$$

$$\mathbf{if(e)\{\ c_1\ c\ \}\ else\{\ c_2\ c\ \}} \ \approx\ \mathbf{if(e)\{\ c_1\ \}\ else\{\ c_2\ \}\ c} \qquad (2)$$

$$\mathbf{while(e)\{\ if(e)\{\ c_1\ \}\ c_2\ \}} \ \approx\ \mathbf{while(e)\{\ c_1\ c_2\ \}} \qquad (3)$$

Similarly, the correctness of the rules in Figure 4 can be expressed as:

$$\forall e_1, e_2, e_3.\ (e_3 = \mathbf{SUB}(e_2, i, e_1)) \rightarrow [\![i = e_1;\ i = e_2;]\!] \approx [\![i = e_3;]\!] \qquad (4)$$

$$\forall i_1, i_2, i_3.\ (i_1 \neq i_2 \wedge i_2 \notin \mathbf{REF}(e_1) \wedge e_3 = \mathbf{SUB}(e_2, i_1, e_1)) \rightarrow [\![i_1 = e_1;\ i_2 = e_2;]\!] \approx [\![i_2 = e_3;\ i_1 = e_1;]\!] \qquad (5)$$

$$\forall e_1, e_2, e_2', c, c'.\ (e_2' = \mathbf{SUB}(e_2, i, e_1) \wedge [\![i{=}e_1;\ c]\!] \Rightarrow [\![c'\ i{=}e_1;]\!] \wedge [\![i{=}e_1;\ c]\!] \approx [\![c'\ i{=}e_1;]\!]) \rightarrow [\![i{=}e_1;\ \texttt{if}\ (e_2)\ c]\!] \approx [\![\texttt{if}\ (e_2')\ c'\ i{=}e_1;]\!] \qquad (6)$$

```
1   total=0;  i=0;
2   evens=0;  noevens=0;
4   scanf("%d",&n);
5   while(i<=n){
14    total = total + A[i];
15    i = i + 1;
16  }
18  meaneven = 0;
22  mean = total/(n+1);
24  evendifference = abs(meaneven-mean);
```

Figure 9: Transformed slice S_1

```
total=0;  i=0;
evens=0;  noevens=0;
scanf("%d",&n);
while(i<=n){
  total = total + A[i];
  i = i + 1;
}
evendifference = total/(n+1);
meaneven = 0;
mean = total/(n+1);
```

Figure 10: Transformed slice S_2

All the statements in (1) – (6) are theorems provable in Coq, once the operational semantics $- \stackrel{-}{\Longrightarrow} -$ is defined.

The definition of $- \stackrel{-}{\Longrightarrow} -$ for three typical statements **if**, =, and **while** is given in Figure 12.

Currently Coq has been used to express the semantics of a simple intra-procedural language, which is based upon a subset of WSL [6], a Wide Spectrum Language used fro expressing transformation. It has also been used to prove some of the basic transformations used in the system. Work is in progress to extend the proofs to cover slicing, domain specific and pre-processing transformations. This work is beyond the scope of the present paper.

```
total=0;  i=0;
scanf("%d",&n);
while(i<=n){
  total = total + A[i];
  i = i + 1;
}
evendifference = total/(n+1);
```

Figure 11: Amorphous slice of program P_0 w.r.t. the variable `evendifference` at the end of P_0

7 Conclusion

This paper has described an amorphous slicing system that mixes (domain independent) dependence reduction transformations with other (domain specific) transformations, pre-processing transformation and traditional slicing to achieve amorphous slicing.

A case study has shown how this combined application of slicing and transformation allows amorphous slicing to achieve thinner slices than traditional syntax-preserving slicing alone and how the combination has beneficial fault-revealing properties.

Some of the transformation tactics used in the system may find application in areas other than the construction of amorphous slices. For example side effect removal may be used as an enabling tactic of other transformation systems which are inapplicable in the presence of side effects. The dependence reduction tactic may be useful in easing automatic parallelization where statement interdependence inhibits the application of parallelizing transformations.

The transformations used in the amorphous slicer are under-pinned by validation which is supported by the Coq proof assistant.

8 Future Work

Future work will consider the following problems:

1. **Criterion guidance component**
   For amorphous slicing technology, the criterion will take on a significant role. Past attempts to exploit slicing technology have been hindered by a lack of expressiveness in slicing-criterion formulation. The criterion guidance component of the GUSTT system described in Figure 2 is currently unimplemented, leaving the human as the sole source of criterion selection. Further work is required to extend the expressibility of the slicing criterion and to provide partially automated assistance to criterion selection.

2. **Domain Specific Transformation component**
   Initial work on domain specific transformation rule sets considered problems in robustness verification [5, 13] and dynamic memory modelling [16]. Further work is required to develop additional targeted transformation rules sets.

3. **Empirical evaluation of amorphous slicing**
   The theory and applications of amorphous slicing to comprehension, reuse, testing and debug-

ging will be evaluated using empirical studies, based upon the use of the system.

Rule 1 (assignment)

$$\frac{[\![e]\!](s) = v}{s \overset{i=e}{\Longrightarrow} s[i \mapsto v]}$$

Rule 2 (if true)

$$\frac{[\![e]\!](s_1) = true,\ s_1 \overset{st}{\Longrightarrow} s_2}{s_1 \overset{\textbf{if}\ (e)\ st}{\Longrightarrow} s_2}$$

Rule 3 (if false)

$$\frac{[\![e]\!](s) = false}{s \overset{\textbf{if}\ (e)\ st}{\Longrightarrow} s}$$

Rule 4 (while true)

$$\frac{[\![e]\!](s_1) = true,\quad s_1 \overset{st}{\Longrightarrow} s_2,\ \ s_2 \overset{\textbf{while}\ (e)\ st}{\Longrightarrow} s_3}{s_1 \overset{\textbf{while}\ (e)\ st}{\Longrightarrow} s_3}$$

Rule 5 (while false)

$$\frac{[\![e]\!](s) = false}{s \overset{\textbf{while}\ (e)\ st}{\Longrightarrow} s}$$

where, $[\![e]\!]$ is the semantics of expression e, which is a function between program states and program values, $s[i \mapsto v]$ is the program state obtained from s by setting the value of variable i to v.

Figure 12: Operational semantics for typical statements

References

[1] Krzsztof R. Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. Alma-O: An imperative language that supports declarative programming. *ACM Transactions on Programming Languages and Systems*, 20(5):1014–1066, September 1998.

[2] Keith Bennett, Tim Bull, Eddie Younger, and Z. Luo. Bylands: reverse engineering safety-critical systems. In *IEEE International Conference on Software Maintenance*, pages 358–366. IEEE Computer Society Press, Los Alamitos, California, USA, 1995.

[3] Keith H. Bennett. Do program transformations help reverse engineering? In *IEEE International Conference on Software Maintenance (ICSM'98)*, pages 247–254, Bethesda, Maryland, USA, November 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[4] David Binkley, Mark Harman, L. Ross Raszewski, and Christopher Smith. An empirical study of amorphous slicing as a program comprehension support tool. In 8th *IEEE International Workshop on Program Comprehension (IWPC 2000)*, pages 161–170, Limerick, Ireland, June 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

[5] David W. Binkley. Computing amorphous program slices using dependence graphs and a data-flow model. In *ACM Symposium on Applied Computing*, pages 519–525, The Menger, San Antonio, Texas, U.S.A., 1999. ACM Press, New York, NY, USA.

[6] Tim Bull. *Software maintenance by program transformation in a wide spectrum language*. PhD thesis, University of Durham, UK, School of Engineering and Computer Science, 1994.

[7] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76:96–120, 1988.

[8] Thierry Coquand and Christine Paulin. Inductively defined types (preliminary version). In P. Martin-Löf and G. Mints, editors, *Proceedings Int. Conf. on Computer Logic, COLOG'88, Tallinn, USSR, 12–16 Dec 1988*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, Berlin, 1990.

[9] Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Gerard Huet, Pascal Manoury, Cesar Munoz, Chetan Murthy, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The coq proof assistant, reference manual, version 5.10. Technical Report RT-0177, Inria, Institut National de Recherche en Informatique et en Automatique, July 1995.

[10] Sebastian Danicic and Mark Harman. Espresso: A slicer generator. In *ACM Symposium on Applied*

Computing, (SAC'00), page To appear, Como, Italy, March 2000.

[11] John Darlington and Rod M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[12] Michael D. Ernst. Practical fine–grained static slicing of optimised code. Technical Report MSR-TR-94-14, Microsoft research, Redmond, WA, July 1994.

[13] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3):143–162, September 1995.

[14] Mark Harman and Sebastian Danicic. Amorphous program slicing. In 5th *IEEE International Workshop on Program Comprehesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[15] Mark Harman, Lin Hu, Xingyuan Zhang, and Malcolm Munro. Side-effect removal transformation. In 9th *IEEE International Workshop on Program Comprehesion (IWPC'01)*, pages 310–319, Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[16] Mark Harman, Yoga Sivagurunathan, and Sebastian Danicic. Analysis of dynamic memory access using amorphous slicing. In *IEEE International Conference on Software Maintenance (ICSM'98)*, pages 336–345, Bethesda, Maryland, USA, November 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[17] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[18] Gilles Kahn. Natural semantics. In *Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*. Springer Verlag, 1987.

[19] Jan Kort, Ralf Lämmel, and Joost Visser. Functional transformation systems. In 9th *International Workshop on Functional and Logic Programming (WFLP'2000)*, Benicassim, Spain, September 2000. Online proceedings at `http://www.dsic.upv.es/~wflp2000/`.

[20] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[21] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Jr., Anwar Ohuloum, and Monica S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In A. Andrew Chien and Marc Snir, editors, *Proceedings of the 1999 ACM Sigplan Symposium on Principles and Practise of Parallel Programming (PPoPP'99)*, volume 34.8 of *ACM Sigplan Notices*, pages 37–48, A.Y., May 4–6 1999. ACM Press.

[22] Vadim Maslov. Lazy array data-flow dependence analysis. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages : papers presented at the Symposium : Portland, Oregon, January 17–21, 1994*, pages 311–325, New York, NY 10036, USA, 1994. ACM Press.

[23] Michael Mehlich and Ira Baxter. Mechanical tool support for high integrity software development. In *High Integrity Systems '97*. IEEE Computer Society Press, Los Alamitos, California, USA, 1997.

[24] Helmut A. Partsch. *The Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer, 1990.

[25] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5–6):607–640, 1993.

[26] Conor Ryan and Paul Walsh. The evolution of provable parallel programs. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 295–302, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[27] Frank Tip. *Generation of Program Analysis Tools*. PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[28] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[29] Martin Ward. Reverse engineering through formal transformation. *The Computer Journal*, 37(5), 1994.

[30] Martin Ward and Keith Bennett. A practical program transformation system. In *Working Conference on Reverse Engineering*, pages 212–221, Baltimore, MD, USA, May 1993. IEEE Computer Society Press, Los Alamitos, California, USA.

[31] Martin Ward, Frank W. Calliss, and Malcolm Munro. The maintainer's assistant. In *Proceedings of the International Conference on Software Maintenance 1989*, page 307. IEEE Computer Society Press, Los Alamitos, California, USA, 1989.

[32] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[33] Kenneth Peter Williams. *Evolutionary Algorithms for Automatic Parallelization*. PhD thesis, University of Reading, UK, Department of Computer Science, September 1998.