Program Repair as Sound Optimization of Broken Programs

Bernd Fischer
School of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, United Kingdom
Email: b.fischer@ecs.soton.ac.uk

Ando Saabas and Tarmo Uustalu
Institute of Cybernetics
Tallinn University of Technology
Akadeemia tee 21, EE-12618 Tallinn, Estonia
Email: {ando|tarmo}@cs.ioc.ee

Abstract

We present a new, semantics-based approach to mechanical program repair where the intended meaning of broken programs (i.e., programs that may abort under a given, error-admitting language semantics) can be defined by a special, error-compensating semantics. Program repair can then become a compile-time, mechanical program transformation based on a program analysis. It turns a given program into one whose evaluations under the error-admitting semantics agree with those of the given program under the error-compensating semantics. We present the analysis and transformation as a type system with a transformation component, following the type-systematic approach to program optimization from our earlier work [12]. The type-systematic method allows for simple soundness proofs of the repairs, based on a relational interpretation of the type system, as well as mechanical transformability of program correctness proofs between the Hoare logics for the error-compensating and error-admitting semantics.

We first demonstrate our approach on the repair of file-handling programs with missing or superfluous open and close statements. Our framework shows that this repair is strikingly similar to partial redundancy elimination optimization commonly used by compilers. In a second example, we demonstrate the repair of programs operating a queue that can over- and underflow, including mechanical transformation of program correctness proofs.

Keywords: program repair, type systems, similarity relations, program optimization soundness, program logics, mechanical transformation of program correctness proofs

1 Introduction

Programmers make mistakes. More often than not, these mistakes manifest themselves in run-time errors that are not caught and lead to abortion. Compilers can be made to detect the possibility of such mistakes by program safety analyses, but they cannot correct them because they cannot know the intended, non-erroneous meaning. Thus it may look as if *program repair* (i.e., transforming programs with abnormal evaluations into programs with meaningful normal evaluations) is inevitably a manual activity.

In this paper, we contest this view by proposing a new, semantics-based approach to program repair. Our central idea is to define the intended meaning of broken programs (i.e., programs that are may abort under a given *error-admitting* language semantics) by a special, *error-compensating* semantics under which programs have no or fewer abnormal evaluations. Program repair can then become a compile-time, mechanical program transformation that turns a given program into one whose evaluations under the error-admitting semantics agree with those of the given program under the error-compensating semantics. We build such repair transformations upon program analyses devised specifically to achieve this form of semantic soundness. In devising error-compensating semantics and repairs we prefer parsimony. This is a justified by a form of Occam's razor: it makes sense to choose a simple explanation for the mistakes in a broken program and thus a simple correction.

Technically, and this is our second key idea, we describe our program analyses and transformations as *type systems* with a transformation component as in our earlier work on program optimizations [12]. The type-systematic method allows for clear separation between the issues of what count as valid analysis and transformation results for a given program (type derivation checking) and how to find the strongest one (principal type inference). As a major practical benefit, this detachment gives simple semantic soundness proofs that also admit a logical counterpart in the form of mechanical transformability of program correctness proofs between the Hoare logics for the error-compensating and error-admitting semantics.

We present our approach on two examples: repair of file-handling programs, which may have missing or superfluous open and close statements, and repair of programs manipulating a queue that can under- and overflow. Both examples show that program repair is similar to program optimization: both are program transformations that are sound and improving in a clear mathematical sense. We can therefore recognize problems and exploit solutions from program optimization, and indeed, file access repair is strikingly similar to partial redundancy elimination à la Paleri et al. [10], which moves expression evaluations.

Our examples are simple, but this is intentional. We have consciously removed all detail that does not contribute to conveying the intuitions we find important. Our previous results on type-systematic program optimization [12, 13, 14, 11] suggest that the techniques of this paper scale smoothly to richer languages and more involved error-compensating semantics and repairs. The hard part, however, is to come up with error-compensating semantics realistically capturing programmer intentions behind mistakes and with matching repairs.

Mechanical program repair is sometimes considered harmful because the programmer may feel uneasy about what the repair does, because one cannot know the "real" intended meaning of broken programs, or because the error-compensating semantics may turn out to be a faulty description of a known intended meaning. While these are valid objections, our position is that manual or heuristic alternatives can only offer less. An error-compensating semantics can indeed be wrong, but the repairs can be passed to the programmer to inspect and approve before they are deployed (cf. refactoring). In the end, such manual testing is the only way to check whether the compensation adopted is right. Moreover, we can define different error-compensating semantics that justify different repair strategies for the same class of errors, and again submit their potentially different outcomes for approval. Importantly, the error-compensating semantics are the only ingredient that can fail in our otherwise mathematically provably sound approach. This isolation of faults is valuable by itself.

2 Repairing file access errors

We introduce our approach on a simple language extending While with statements for opening, closing and reading from (but not writing to) files. Our aim here is to reconstruct the intent of programmers who have made mistakes in the file handling. The arithmetic expressions $a \in \mathbf{AExp}$, boolean expressions $b \in \mathbf{BExp}$ and statements $s \in \mathbf{Stm}$ are defined over supplies of program variables $x \in \mathbf{Var}$ and file names $f \in \mathbf{F}$, and the integer numerals $n \in \mathbb{Z}$:

$$\begin{array}{lll} a & ::= & x \mid n \mid a_0 + a_1 \mid \ldots \\ b & ::= & a_0 = a_1 \mid \ldots \mid \mathrm{tt} \mid \mathrm{ff} \mid \neg b \mid \ldots \\ s & ::= & x := a \mid \mathsf{skip} \mid s_0; s_1 \mid \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \mid \\ & & \mathsf{while}\ b\ \mathsf{do}\ s_t \mid \mathsf{open}(f) \mid \mathsf{close}(f) \mid \mathsf{read}(f, x) \end{array}$$

We consider a simple model for file access. A file must be opened before it can be read and closed before the program terminates; to simplify our presentation, we ignore end-of-file errors. Opening a file sets its pointer (i.e., the line to be read) to zero. Reading a value from the current line increases the file pointer by one.

Error-admitting semantics, safety type system and its soundness The precise semantics of the language is described in terms of states. A state is a pair of a store $\sigma \in \mathbf{Var} \longrightarrow \mathbb{Z}$ and a file status table $\rho \in \mathbf{F} \longrightarrow \{\mathsf{c}\} + \{\mathsf{o}(n) \mid n \in \mathbb{N}\}$ that records, for every file, whether it is closed (c) or open (o) and for open files also the pointer value. The file contents are modeled by a map $\phi \in \mathbf{F} \times \mathbb{N} \longrightarrow \mathbb{Z}$ from file names and pointer values to integers. For programs with an erroneous file access, the normal evaluation rules as given in Figure 1 allow no derivation of a final state; instead, there is a specific judgement form for abnormal evaluations. The semantics is thus *error-admitting*: rather than trying to continue past the error, it aborts.

We can easily construct tools that statically check whether a program is safe (cannot evaluate abnormally), using, e.g., the type system in Figure 2, describing a simple forward analysis. In this type system, a type is a map $d \in \mathbf{F} \to \{\mathsf{c}, \mathsf{o}\}$ judging every file closed or open. Types can be interpreted as properties of states by

$$\frac{}{\mathsf{o}(n) \models \mathsf{o}} \qquad \frac{}{\mathsf{c} \models \mathsf{c}} \qquad \frac{\forall f \in \mathbf{F}.\, \rho(f) \models d(f)}{(\sigma, \rho) \models d}$$

Under this interpretation, the type system is sound in the classical sense that well-typed programs do not go wrong, including safety:

Theorem 1 *If $s : d \longrightarrow d'$ in the safety type system, then (i) if $(\sigma, \rho) \models d$ and $(\sigma, \rho) \succ s \rightarrow (\sigma', \rho')$ in the error-admitting semantics, then $(\sigma', \rho') \models d'$, and (ii) it cannot simultaneously be that $(\sigma, \rho) \models d$ and $(\sigma, \rho) \succ s \rightarrow\!\!\mid$ in the error-admitting semantics.*

This type system is so simple that principal type inference involves no fixed-point computation. But it is also crude and rejects many safe programs; to reason about safety precisely, a dedicated Hoare logic could be used [5].

From safety certification to repair We are interested in more than safety checking, namely in *repairing* broken programs. For file access, it is reasonable to require a repair to satisfy the following conditions. The non-erroneous evaluations of the given program should be preserved. The repaired program should be safe. Two reads in the given program from the same file with no close or open statement in between should be kept in the same session (so no open or close can be inserted between them). In the opposite direction, any close or open statement should be treated as

$$\frac{\rho(f) = \mathsf{c}}{\sigma, \rho \succ \mathsf{open}(f) \rightarrow \sigma, \rho[f \mapsto \mathsf{o}(0)]} \qquad \frac{\rho(f) = \mathsf{o}(n)}{\sigma, \rho \succ \mathsf{close}(f) \rightarrow \sigma, \rho[f \mapsto \mathsf{c}]} \qquad \frac{\rho(f) = \mathsf{o}(n)}{\sigma, \rho \succ \mathsf{read}(f, x) \rightarrow \sigma[x \mapsto \phi(f, \mathsf{o}(n))], \rho[f \mapsto \mathsf{o}(n+1)]}$$

$$\frac{}{\sigma, \rho \succ x := a \rightarrow \sigma[x \mapsto [\![a]\!]\sigma], \rho} \qquad \frac{}{\sigma, \rho \succ \mathsf{skip} \rightarrow \sigma, \rho} \qquad \frac{\sigma, \rho \succ s_0 \rightarrow \sigma'', \rho'' \quad \sigma'', \rho'' \succ s_1 \rightarrow \sigma', \rho'}{\sigma, \rho \succ s_0; s_1 \rightarrow \sigma', \rho'}$$

$$\frac{\sigma \models b \quad \sigma, \rho \succ s_t \rightarrow \sigma', \rho'}{\sigma, \rho \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \sigma', \rho'} \qquad \frac{\sigma \not\models b \quad \sigma, \rho \succ s_f \rightarrow \sigma', \rho'}{\sigma, \rho \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \sigma', \rho'}$$

$$\frac{\sigma \models b \quad \sigma, \rho \succ s_t \rightarrow \sigma'', \rho'' \quad \sigma'', \rho'' \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma', \rho'}{\sigma, \rho \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma', \rho'} \qquad \frac{\sigma \not\models b}{\sigma, \rho \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma, \rho}$$

$$\frac{\rho(f) = \mathsf{o}(n)}{\sigma, \rho \succ \mathsf{open}(f) \rightarrow\!\mid} \qquad \frac{\rho(f) = \mathsf{c}}{\sigma, \rho \succ \mathsf{close}(f) \rightarrow\!\mid} \qquad \frac{\rho(f) = \mathsf{c}}{\sigma, \rho \succ \mathsf{read}(f, x) \rightarrow\!\mid} \qquad \frac{\sigma, \rho \succ s_0 \rightarrow\!\mid}{\sigma, \rho \succ s_0; s_1 \rightarrow\!\mid} \qquad \frac{\sigma, \rho \succ s_0 \rightarrow \sigma'', \rho'' \quad \sigma'', \rho'' \succ s_1 \rightarrow\!\mid}{\sigma, \rho \succ s_0; s_1 \rightarrow\!\mid}$$

$$\frac{\sigma \models b \quad \sigma, \rho \succ s_t \rightarrow\!\mid}{\sigma, \rho \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow\!\mid} \qquad \frac{\sigma \not\models b \quad \sigma, \rho \succ s_f \rightarrow\!\mid}{\sigma, \rho \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow\!\mid}$$

$$\frac{\sigma \models b \quad \sigma, \rho \succ s_t \rightarrow\!\mid}{\sigma, \rho \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow\!\mid} \qquad \frac{\sigma \models b \quad \sigma, \rho \succ s_t \rightarrow \sigma'', \rho'' \quad \sigma'', \rho'' \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow\!\mid}{\sigma, \rho \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow\!\mid}$$

Figure 1. Error-admitting semantics for file access

$$\frac{d(f) = \mathsf{c}}{\mathsf{open}(f) : d \longrightarrow d[f \mapsto \mathsf{o}]} \qquad \frac{d(f) = \mathsf{o}}{\mathsf{close}(f) : d \longrightarrow d[f \mapsto \mathsf{c}]} \qquad \frac{d(f) = \mathsf{o}}{\mathsf{read}(f, x) : d \longrightarrow d}$$

$$\frac{}{x := a : d \longrightarrow d} \qquad \frac{}{\mathsf{skip} : d \longrightarrow d} \qquad \frac{s_0 : d \longrightarrow d'' \quad s_1 : d'' \longrightarrow d}{s_0; s_1 : d \longrightarrow d'} \qquad \frac{s_t : d \longrightarrow d' \quad s_f : d \longrightarrow d'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : d \longrightarrow d'} \qquad \frac{s_t : d \longrightarrow d}{\mathsf{while}\ b\ \mathsf{do}\ s_t : d \longrightarrow d}$$

Figure 2. Safety type system for file access

a session breaker. Finally, for the sake of determinism of the repair, any necessary new opens and closes should be placed so that files are opened as late and closed as early as possible. Different criteria are conceivable too, but we commit to those above.

We will see that a repair meeting our chosen criteria is achievable by first *removing all* open and close statements from the given program and then *inserting some* open and close statements, generally *elsewhere*, to render all read statements safe and belonging appropriately into the same or different sessions, with *minimal session lengths*. Consider, e.g., the broken program on the left and its repaired version on the right:

```
read(f, x);          open(f); read(f, x);
read(f, y);                   read(f, y); close(f) :
open(f);        ↪
read(f, z);          open(f); read(f, z); close(f);
w := x − z;                   w := x − z
close(f)
```

The repair has added the single pair of necessary open and close statements around the first two read statements, which must belong to the same session. The third read is kept in a different session, by reinsertion of the removed original open statement, thus ensuring that z still receives the first value in the file. The removed close is reinserted before the assignment, thus shortening this session (which was already safe in the original program).

Repair of branching programs is more subtle. In general, we must track the possible past and future file reads for all program points. If a file is certainly, i.e., for all incoming paths, unread immediately before a read statement, then the read must be preceded by an open. If there are definitely no further future reads, then the file must be closed immediately after the read. Consider this example:

```
if b then                if b then
    read(f, x)               open(f); read(f, x)
else                ↪    else
    x := x + 1;              x := x + 1;
                             open(f);
read(f, y)               read(f, y); close(f)
```

Opening the file before the if-statement conflicts with the goal of minimizing session lengths. Hence, the repair inserts the open statement as late as possible, right before the read in the then-branch. But it also has to append an open statement to the else-branch, to ensure that the second read is safe, if that branch is taken. This "edge-splitting" insertion is the only option as, at the end of the if-statement, there is a past read on an incoming path and a future read on the outgoing path. Since there is no intervening open or close statement between the two read statements, they belong to the same session and the file must be open at this point.

It is important to realize that sometimes sessions without any reads must be introduced to make a program safe. Consider this variation of the previous example:

```
if $b$ then                  if $b$ then
    read$(f,x)$                  open$(f)$; read$(f,x)$
else                         else
    $x := x+1$;                  $x := x+1$;
                                 open$(f)$;
if $b'$ then           $\hookrightarrow$   if $b'$ then
                                 close$(f)$;
    $y := y+1$                   $y := y+1$
else                         else
    read$(f,y)$                  read$(f,y)$; close$(f)$
```

To ensure that the file is open between the two if-statements (so the two reads can be in the same session), an open is inserted at the end of the false branch of the first if-statement and a close at the beginning of the true branch of the second if-statement. As a consequence, if b is true but b' is false, the file is opened and then closed without reading from it.

We have thus witnessed that our repair will be an analysis-based transformation. Moreover, it not only repairs broken programs, but optimizes safe programs by minimizing session lengths, at no additional cost. The analysis and transformation are similar to partial redundancy elimination in the version of Paleri et al. [10], based on partial availability and partial anticipability made conditional on the disjunction on total availability and anticipability. Inserted open statements play the role of inserted assignments to auxiliary variables, while inserted close statements correspond to inserted discardings of auxiliary variables.

Repair type system Finding possible past reads from the files requires a forward may analysis and finding possible future reads requires a backward may analysis. We describe these two analyses in a single a type system. A type is a pair (d, e), where $d, e \in \mathbf{F} \to \{\mathsf{r}, \mathsf{u}\}$ are maps from file names to tokens r, u. The token r means *possibly read* in the past (and not closed or opened after) resp. in the future (and not opened or closed before), depending on the component d or e. The other token u stands for *certainly unread*. A typing judgement takes the form $s : (d, e) \longrightarrow (d', e')$ and asserts that, if all files read in the past before a run of s are mapped to r by d, then all files read in the past after this run are assigned r by d', and, if all files read in the future after a run of s are r according to e', then before it they are r in e. The subtyping and typing rules are in Figure 3 (ignore the grayed-out part for a moment). The statements changing the r/u values are open, close and read. Since we take both open and close to break sessions, both set the file to u in both the past and future—thus the same typing rules. Not surprisingly, a read statement marks the file as r in both the past and the future. To type if- and while-statements, one may need the subsumption rule (cf. the consequence rule of Hoare logics) to fit the types. In fact, read statements can be treated similarly (together with subsumption, the bracketed simplified rule recovers the more complex read rule).

The grayed-out parts in Figure 3 define the transformation based on the two analyses. A typing judgement with a transformation component has the form $s : (d, e) \longrightarrow (d', e') \hookrightarrow s_*$, stating that, based on its type, the program s can be rewritten to s_*. The open and close rules replace all open and close statements with skips, since the repair inserts new opens and closes. In particular, opens and closes already placed optimally are essentially restored. The transformation rules for read are also straightforward—the first read from a file (i.e., the case when the file is marked unread in the past pretype) in a session is preceded with a file open, and the last read (i.e., the case where the file is unread in the posttype) with a close statement. Edge-splitting insertions of opens and closes at new nodes of the control-flow graph are carried out by the subsumption rule together with the subtyping rules. At a subsumption, the subtypings used dictate sequences of file opens and closes to be inserted before and after the transformed version of the given statement. This is the purpose of having a transformation component also in the subtyping rules: any derivable subtyping defines a sequence of open and close statements. An overarching idea behind the transformation rules is to assume and guarantee that, at the points in the given program where a file f is typed (r, r) (i.e., the point is between two possible reads from f unintervened by closes or opens of f), this file is open in the repaired program; at the points typed otherwise, the repaired program should have the file closed.

An example type and transformation derivation is given in Figure 4. The subsumption rule is applied to the else-branch of the if-statement, where the type of f is weakened from (u, r) to (r, r) at the end. Intuitively, f is now assumed to have been read before, and thus all future reads can assume the file to be open. For this reason, we have to introduce an open(f) statement to guarantee that the file *actually is* open. This is exactly what the subtyping rule gives us.

Soundness wrt. an error-compensating semantics A repair should not only transform a given program into a safe program: the repaired program should inherit the normal evaluations of the given program. But what is more, also the abnormal evaluations of the given program should not be turned into arbitrary normal evaluations of the repaired program. So what is our intent for abnormal evaluations? We have in fact informally specified it by the conditions on the repair we laid out. Rigorously, it can be expressed by the requirement that the repair must respect a non-standard, *error-compensating* semantics of our language. Essentially, we said that the only meaning we give to open and close statements in a broken program is breaking a possibly on-

$$\frac{}{(\mathsf{u},\mathsf{r}) \le (\mathsf{r},\mathsf{r}) \hookrightarrow_f \mathsf{open}(f)} \qquad \frac{}{(\mathsf{r},\mathsf{r}) \le (\mathsf{r},\mathsf{u}) \hookrightarrow_f \mathsf{close}(f)} \qquad \frac{}{(m,m) \le (m,m) \hookrightarrow_f \mathsf{skip}} \qquad \frac{}{(\mathsf{u},m) \le (m',\mathsf{u}) \hookrightarrow_f \mathsf{skip}}$$

$$\frac{\forall f \in \mathbf{F}.\ (d(f),e(f)) \le (d'(f),e'(f)) \hookrightarrow_f s(f)}{(d,e) \le (d',e') \hookrightarrow [s(f) \mid f \in \mathbf{F}]}$$

$$\frac{(d,e) \le (d_0,e_0) \hookrightarrow s_{pre} \qquad s : (d_0,e_0) \longrightarrow (d_0',e_0') \hookrightarrow s_* \qquad (d_0',e_0') \le (d',e') \hookrightarrow s_{post}}{s : (d,e) \longrightarrow (d',e') \hookrightarrow s_{pre}; s_*; s_{post}}$$

$$\frac{}{x := a : (d,e) \longrightarrow (d,e) \hookrightarrow x := a}$$

$$\frac{}{\mathsf{skip} : (d,e) \longrightarrow (d,e) \hookrightarrow \mathsf{skip}} \qquad \frac{s_0 : (d,e) \longrightarrow (d'',e'') \hookrightarrow s_0' \qquad s_1 : (d,e) \longrightarrow (d',e') \hookrightarrow s_1'}{s_0; s_1 : (d,e) \longrightarrow (d',e') \hookrightarrow s_0'; s_1'}$$

$$\frac{s_t : (d,e) \longrightarrow (d',e') \hookrightarrow s_t' \qquad s_f : (d,e) \longrightarrow (d',e') \hookrightarrow s_f'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : (d,e) \longrightarrow (d',e') \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'} \qquad \frac{s_t : (d,e) \longrightarrow (d,e) \hookrightarrow s_t'}{\mathsf{while}\ b\ \mathsf{do}\ s_t : (d,e) \longrightarrow (d,e) \hookrightarrow s_t'}$$

$$\frac{}{\mathsf{open}(f) : (d, e[f \mapsto \mathsf{u}]) \longrightarrow (d[f \mapsto \mathsf{u}], e) \hookrightarrow \mathsf{skip}} \qquad \frac{}{\mathsf{close}(f) : (d, e[f \mapsto \mathsf{u}]) \longrightarrow (d[f \mapsto \mathsf{u}], e) \hookrightarrow \mathsf{skip}}$$

$$\frac{}{\begin{array}{l}\mathsf{read}(f,x) : (d, e[f \mapsto \mathsf{r}]) \longrightarrow (d[f \mapsto \mathsf{r}], e) \\ \hookrightarrow [\mathsf{open}(f) \mid d(f) = \mathsf{u}]; \mathsf{read}(f,x); [\mathsf{close}(f) \mid e(f) = \mathsf{u}]\end{array}} \qquad \left[\ \frac{d(f) = e(f) = \mathsf{r}}{\mathsf{read}(f,x) : (d,e) \longrightarrow (d,e) \hookrightarrow \mathsf{read}(f,x)}\ \right]$$

Figure 3. Repair type system for file access

$$\frac{\dfrac{}{\begin{array}{l}\mathsf{read}(f,x) : (\mathsf{u},\mathsf{r}) \longrightarrow (\mathsf{r},\mathsf{r}) \\ \quad \hookrightarrow \mathsf{open}(f); \mathsf{read}(f,x)\end{array}} \qquad \dfrac{\dfrac{}{x := 0 : (\mathsf{u},\mathsf{r}) \longrightarrow (\mathsf{u},\mathsf{r}) \hookrightarrow x := 0} \quad \dfrac{}{(\mathsf{u},\mathsf{r}) \le (\mathsf{r},\mathsf{r}) \hookrightarrow \mathsf{open}(f)}}{x := 0 : (\mathsf{u},\mathsf{r}) \longrightarrow (\mathsf{r},\mathsf{r}) \hookrightarrow x := 0; \mathsf{open}(f)}}{\begin{array}{l}\mathsf{if}\ b\ \mathsf{then}\ \mathsf{read}(f,x)\ \mathsf{else}\ x := 0 : (\mathsf{u},\mathsf{r}) \longrightarrow (\mathsf{r},\mathsf{r}) \\ \quad \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{open}(f); \mathsf{read}(f,x))\ \mathsf{else}\ (x := 0; \mathsf{open}(f))\end{array}} \qquad \frac{}{\begin{array}{r}\mathsf{read}(f,y) : (\mathsf{r},\mathsf{r}) \longrightarrow (\mathsf{r},\mathsf{u}) \\ \hookrightarrow \mathsf{read}(f,y); \mathsf{close}(f)\end{array}}$$

$$\frac{}{\begin{array}{l}\mathsf{if}\ b\ \mathsf{then}\ \mathsf{read}(f,x)\ \mathsf{else}\ x := 0; \mathsf{read}(f,y) : (\mathsf{u},\mathsf{r}) \longrightarrow (\mathsf{r},\mathsf{u}) \\ \quad \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{open}(f); \mathsf{read}(f,x))\ \mathsf{else}\ (x := 0; \mathsf{open}(f)); \mathsf{read}(f,y); \mathsf{close}(f)\end{array}}$$

Figure 4. Example repair type and transformation derivation for file access

going session, practically amounting to resetting the file pointer. Hence we can take the intended meaning of broken programs to be given by a semantics where all files are always open and both opens and closes just reset the file pointer. Differently from the error-admitting semantics, a file status table in this semantics is thus simply a mapping $\rho : \mathbf{F} \longrightarrow \mathbb{N}$ from file names to natural numbers. The rules for file access statements are in Figure 5 (for the other statements they remain unchanged).

To be able to align the runs of the given and the repaired program in the error-compensating resp. error-admitting semantics, we introduce a repair-type-indexed *similarity relation* between their states by the following rules:

$$\frac{}{n \sim_{(\mathsf{r},\mathsf{r})} \mathsf{o}(n)} \qquad \frac{}{0 \sim_{(\mathsf{u},\mathsf{r})} \mathsf{c}} \qquad \frac{}{n \sim_{(m,\mathsf{u})} \mathsf{c}}$$

$$\frac{\forall f \in \mathbf{F}.\ \rho(f) \sim_{(d,e)} \rho_*(f)}{(\sigma,\rho) \sim_{(d,e)} (\sigma,\rho_*)}$$

In effect, these rules reflect the idea behind the transformation rules. For a point in the given program, the only type corresponding to an open file in the repaired program is (r,r), signifying possible reads in both the past and future in the given program. In this case, the file pointers in both of the related states have to be equal. For all other types, the file should be closed in the repaired program, since files are opened (resp. closed) only *immediately* before reads or control-flow joins (resp. after reads or control-flow forks). For (u,r), in the given program we have that the file is unread in the past, but will be read from in the future (with no prior open or close resetting the file pointer), so the file pointer must already be 0, whereas in the repaired program the file is closed. A type (m,u) indicates that a file is not read in the future before an open or close. In this situation the file pointer can have any value, as it has no chance to affect the final state.

We now are equipped to prove the repair type system mathematically sound by induction on the type derivation.

Theorem 2 *If $s : (d,e) \longrightarrow (d',e') \hookrightarrow s_*$ in the repair type system, then*
(i) if $(\sigma,\rho) \sim_{(d,e)} (\sigma_*,\rho_*)$ and $(\sigma,\rho) \succ\!s\!\rightarrow (\sigma',\rho')$ in the error-compensating semantics, then there must exist

$$\frac{}{\sigma, \rho \succ \mathsf{open}(f) \rightarrow \sigma, \rho[f \mapsto 0]} \qquad \frac{}{\sigma, \rho \succ \mathsf{close}(f) \rightarrow \sigma, \rho[f \mapsto 0]} \qquad \frac{}{\sigma, \rho \succ \mathsf{read}(f, x) \rightarrow \sigma[x \mapsto \phi(f, \rho(f))], \rho[f \mapsto \rho(f) + 1]}$$

Figure 5. Error-compensating semantics for file access

a state (σ'_*, ρ'_*) such that $(\sigma', \rho') \sim_{(d',e')} (\sigma'_*, \rho'_*)$ and $(\sigma_*, \rho_*) \succ s_* \rightarrow (\sigma'_*, \rho'_*)$ in the error-admitting semantics;
(ii) if $(\sigma, \rho) \sim_{(d,e)} (\sigma_*, \rho_*)$ and $(\sigma_*, \rho_*) \succ s_* \rightarrow (\sigma'_*, \rho'_*)$ in the error-admitting semantics, then there must exist a state (σ', ρ') such that $(\sigma', \rho') \sim_{(d',e')} (\sigma'_*, \rho'_*)$ and $(\sigma, \rho) \succ s \rightarrow (\sigma', \rho')$ in the error-compensating semantics;
(iii) it cannot simultaneously be that $(\sigma, \rho) \sim_{(d,e)} (\sigma_*, \rho_*)$ and $(\sigma_*, \rho_*) \succ s_* \rightarrow\!\!\mid$ in the error-admitting semantics;
(iv) $s_* : (d, e)^\# \longrightarrow (d', e')^\#$ where $(\mathsf{r}, \mathsf{r})^\# =_{\mathrm{df}} \mathsf{o}$, $(m, \mathsf{u})^\# =_{\mathrm{df}} \mathsf{c}$, $(\mathsf{u}, m)^\# =_{\mathrm{df}} \mathsf{c}$ and $(d, e)^\#(f) =_{\mathrm{df}} (d(f), e(f))^\#$.

Here (i) and (ii) express semantic equivalence for normal evaluations of the given and repaired program, (iii) states semantic safety of the repaired program and (iv) states that this safety is also detected by our safety type system.

It is also important that the repair does not change the semantics of safe programs. This only requires that the error-admitting and error-compensating semantics agree on the programs typable in the safety type system. To state this, we define a safety-type-indexed similarity relation between the states of the two semantics by

$$\frac{}{\mathsf{o}(n) \approx_{\mathsf{o}} n} \qquad \frac{}{\mathsf{c} \approx_{\mathsf{c}} 0} \qquad \frac{\forall f \in \mathbf{F}.\ \rho(f) \approx_d \rho_*(f)}{(\sigma, \rho) \approx_d (\sigma, \rho_*)}$$

We can now strengthen the soundness of the safety type system (Theorem 1) as follows.

Theorem 3 *If $s : d \longrightarrow d'$ in the safety type system, then*
(i) if $(\sigma, \rho) \approx_d (\sigma_*, \rho_*)$ and $(\sigma, \rho) \succ s \rightarrow (\sigma', \rho')$ in the error-admitting semantics, then there must exist a state (σ'_*, ρ'_*) such that $(\sigma', \rho') \approx_{d'} (\sigma'_*, \rho'_*)$ and $(\sigma_*, \rho_*) \succ s \rightarrow (\sigma'_*, \rho'_*)$ in the error-compensating semantics;
(ii) if $(\sigma, \rho) \approx_d (\sigma_*, \rho_*)$ and $(\sigma_*, \rho_*) \succ s \rightarrow (\sigma'_*, \rho'_*)$ in the error-compensating semantics, then there must exist a state (σ', ρ') such that $(\sigma', \rho') \approx_{d'} (\sigma'_*, \rho'_*)$ and $(\sigma, \rho) \succ s \rightarrow (\sigma', \rho')$ in the error-admitting semantics.

On programs typable in the safety type system, the repair can also be shown improving: it reduces (or, more precisely, does not grow) the number and lengths of file sessions. This can be done with the help of an instrumented error-admitting semantics that keeps track of the number of times each file is opened and how long it stays open.

3 Repairing queue over/underflows

We now look at a different example, over/underflows of a bounded queue. Here the aim of repair is not to reconstruct the intent of programmers that make mistakes, but to *expose* how a particular implementation may deal with them, without warning the programmers. We introduce two boolean expression and two statement forms:

$$\begin{array}{lcl} b & ::= & \ldots \mid \mathsf{full} \mid \mathsf{emp} \\ s & ::= & \ldots \mid \mathsf{enq}(a) \mid \mathsf{deq}(x) \end{array}$$

full and emp test the queue for fullness and emptiness, $\mathsf{enq}(a)$ appends the value of a to the queue, and $\mathsf{deq}(x)$ removes the head value of the queue, assigning it to x.

Error-admitting semantics, safety type system and its soundness A state in the standard, error-admitting semantics is a pair of a store $\sigma \in \mathbf{Var} \longrightarrow \mathbb{Z}$ and a queue content $q \in \mathbb{Z}^*$, with $|q| \leq N$ for some fixed $N \in \mathbb{N}$. We model the queue via the list datatype. We write X^* for lists over a set X, $[]$ for the empty list, $x : xs$ for the list with head x and tail xs, $xs \mathbin{+\!\!+} ys$ for the concatenation of lists xs and ys, and $|q|$ for the length of the list q. The rules for the enqueue and dequeue statements are given in Figure 6 (top); the rules for the other statements are similar to those given in Figure 1, as none of them affect the queue. Under this semantics, a program raises an error when an element is added to a queue that is full or a dequeue is attempted from an empty queue. It is easy to see that safety can be guaranteed by a simple type system for interval analysis of the queue length. The types are pairs of a lower and upper bound on the queue length: $lo, hi \in [0, N]$, $lo \leq hi$. A typing judgement in the form $s : [lo, hi] \longrightarrow [lo', hi']$ states that, if before running s the queue length is between lo and hi, then after running s, it is between lo' and hi'. The rules are in Figure 6 (bottom); the soundness result is analogous to Theorem 1.

Error-compensating semantics, repair type system and its soundness We can imagine an implementation of the queue that skips enqueues to a full queue and dequeues a default value 0 from an empty queue, thus never aborting and defining an *ad-hoc* error-compensating semantics. Repairing a program soundly with respect to this error-compensating semantics makes the program portable from that particular implementation to any implementation that agrees with the normal evaluations of the error-admitting semantics. This is exactly what we will study now.

The states of the error-compensating semantics are the same as those of the error-admitting semantics and the rules are given in Figure 7 (top). The simplest sound repair would guard all enqueue and dequeue statements by fullness and emptiness tests, but we can obviously do better, again by interval analysis. The types of the repair type system are the same as those of the safety type system. The rules appear in

$$\frac{|q| < N}{\sigma, q \succ\mathsf{enq}(a)\rightarrow \sigma, q +\!\!+ [[\![a]\!]\sigma]} \qquad \frac{}{\sigma, v : q \succ\mathsf{deq}(x)\rightarrow \sigma[x \mapsto v], q} \qquad \frac{|q| = N}{\sigma, q \succ\mathsf{enq}(a)\rightarrow\!|} \qquad \frac{}{\sigma, [] \succ\mathsf{deq}(x)\rightarrow\!|}$$

$$\frac{lo' \leq lo \quad hi \leq hi'}{[lo, hi] \leq [lo', hi']} \qquad \frac{[lo, hi] \leq [lo_0, hi_0] \quad s : [lo_0, hi_0] \longrightarrow [lo_0', hi_0'] \quad [lo_0', hi_0'] \leq [lo', hi']}{s : [lo, hi] \longrightarrow [lo', hi']}$$

$$\frac{hi < N}{\mathsf{enq}(a) : [lo, hi] \longrightarrow [lo + 1, hi + 1]} \qquad \frac{0 < lo}{\mathsf{deq}(x) : [lo, hi] \longrightarrow [lo - 1, hi - 1]}$$

Figure 6. Error-admitting semantics (top) and safety type system (bottom) for queue operations

$$\frac{|q| < N}{\sigma, q \succ\mathsf{enq}(a)\rightarrow \sigma, q +\!\!+ [[\![a]\!]\sigma]} \qquad \frac{|q| = N}{\sigma, q \succ\mathsf{enq}(a)\rightarrow \sigma, q} \qquad \frac{}{\sigma, v : q \succ\mathsf{deq}(x)\rightarrow \sigma[x \mapsto v], q} \qquad \frac{}{\sigma, [] \succ\mathsf{deq}(x)\rightarrow \sigma[x \mapsto 0], []}$$

$$\frac{lo' \leq lo \quad hi \leq hi'}{[lo, hi] \leq [lo', hi']} \qquad \frac{[lo_0, hi_0] \leq [lo, hi] \quad s : [lo, hi] \longrightarrow [lo', hi'] \hookrightarrow s_* \quad [lo', hi'] \leq [lo_0', hi_0']}{s : [lo_0, hi_0] \longrightarrow [lo_0', hi_0'] \hookrightarrow s_*}$$

$$\frac{hi < N}{\mathsf{enq}(a) : [lo, hi] \longrightarrow [lo + 1, hi + 1] \hookrightarrow \mathsf{enq}(a)} \qquad \frac{}{\mathsf{enq}(a) : [N, N] \longrightarrow [N, N] \hookrightarrow \mathsf{skip}} \qquad \frac{lo < N}{\mathsf{enq}(a) : [lo, N] \longrightarrow [lo + 1, N] \hookrightarrow \mathsf{if}\ \neg\mathsf{full}\ \mathsf{then}\ \mathsf{enq}(a)\ \mathsf{else}\ \mathsf{skip}}$$

$$\frac{0 < lo}{\mathsf{deq}(x) : [lo, hi] \longrightarrow [lo - 1, hi - 1] \hookrightarrow \mathsf{deq}(x)} \qquad \frac{}{\mathsf{deq}(x) : [0, 0] \longrightarrow [0, 0] \hookrightarrow x := 0} \qquad \frac{0 < hi}{\mathsf{deq}(x) : [0, hi] \longrightarrow [0, hi - 1] \hookrightarrow \mathsf{if}\ \neg\mathsf{emp}\ \mathsf{then}\ \mathsf{deq}(x)\ \mathsf{else}\ x := 0}$$

Figure 7. Error-compensating semantics (top) and repair type system (bottom) for queue operations

Figure 7 (bottom) and should be straightforward. The transformation component in the type system covers three distinct cases for both enqueuing and dequeuing. If we know from the type that the queue is not full, the enqueue statement remains unchanged. If we know for certain that the queue is full (the bounds are $[N, N]$), the enqueue statement is removed. If the queue *may* be full (the case where the upper bound is N, but the lower bound is smaller), the statement is replaced with a conditional checking the queue size. The cases for dequeuing are handled similarly.

Again the repair is sound for a relational interpretation of types. Two states (of the error-compensating and error-admitting semantics) are similar at a type, if they are equal and the queue length is within the given bounds:

$$\frac{lo \leq |q| \leq hi}{(\sigma, q) \sim_{(lo,hi)} (\sigma, q)}.$$

The soundness result is analogous to that for the repair type system for file access (Theorem 2): the given and repaired program agree on the normal evaluations in their respective semantics and the repaired program is safe under the error-admitting semantics.

Hoare logics We now show how the repair type system can be used for mechanical transformation of program correctness proofs—massaging Hoare logic proofs about given programs into Hoare logic proofs about repaired programs and vice versa.

Both the error-admitting and error-compensating semantics have logical counterparts in Hoare logics. In both cases we take the assertion language to have an extralogical constant q to refer to the current queue content. In the Hoare logic for the error-admitting semantics, the rules for enqueue and dequeue are

$$\frac{}{\{|q| < N \wedge Q[q +\!\!+ [a]/q]\}\ \mathsf{enq}(a)\ \{Q\}}$$

$$\frac{}{\{\exists v, u.\ q = v : u \wedge Q[v, u/x, q]\}\ \mathsf{deq}(x)\ \{Q\}}$$

In both rules, the precondition guarantees that the statement does not abort.[1] In the Hoare logic for the error-compensating semantics, the rules for enqueue and dequeue take the form

$$\frac{}{\{(|q| < N \wedge Q[q +\!\!+ [a]/q]) \vee (|q| = N \wedge Q)\}\ \mathsf{enq}(a)\ \{Q\}}$$

$$\frac{}{\left\{\begin{array}{ll} & (\exists v, u.\ q = v : u \wedge Q[v, u/x, q]) \\ \vee & (q = [] \wedge Q[0/x]) \end{array}\right\}\ \mathsf{deq}(x)\ \{Q\}}$$

[1]Note that two flavors of logic are possible, depending on how we wish to treat errors. We consider a triple valid if (i) if the program is run from a state satisfying the precondition and terminates normally, the final state satisfies the postcondition, and (ii) the program cannot abort. Alternatively, validity can be taken to mean the first conjunct only, guaranteeing nothing about broken programs. This leads to a different logic.

They reflect the fact that on a full queue, enqueue is equivalent to skip, and dequeue on an empty queue is equivalent to an assignment. Both logics are sound and relatively complete (relative wrt. the unachievable completeness of arithmetic) wrt. the respective semantics.

Correctness proof transformability Now, corresponding to a similarity relation between two states at a type, we define two *assertion translations* depending on the type. We set $P|_{[lo,hi]} =_{\mathrm{df}} P|^{[lo,hi]} =_{\mathrm{df}} lo \leq |q| \leq hi \wedge P$, so the two translations from the assertions of the Hoare logic for the error-compensating semantics to those of the Hoare logic for the error-admitting semantics and back are in fact identical. This happens because the two semantics rely on the same notion of a state and the similarity relation is symmetric. In general, e.g., in file access repair, this is not the case and there would be two different translations.

The following Hoare logic proof transformability result is provable either from the soundness of the repair type system and soundness and completeness of the Hoare logics or directly by induction on the type derivation.

Theorem 4 *If* $s : [lo, hi] \longrightarrow [lo', hi'] \hookrightarrow s_*$ *in the repair type system, then*
(i) If $\{P\}\, s\, \{Q\}$ *in the Hoare logic for the error-compensating semantics, then* $\{P|_{[lo,hi]}\}\, s_*\, \{Q|_{[lo',hi']}\}$ *in the Hoare logic for the error-admitting semantics,*
(ii) If $\{P\}\, s_*\, \{Q\}$ *in the Hoare logic for the error-admitting semantics,* $\{P|^{[lo,hi]}\}\, s\, \{Q|^{[lo',hi']}\}$ *in the Hoare logic for the error-compensating semantics.*

The direct proof for this theorem gives us mechanical correctness proof transformations: type-derivation directed transformations of proofs between the two Hoare logics.

As a small example, for $N =_{\mathrm{df}} 1$ and the queue initially containing one element, the repair type system derives $s : [1,1] \longrightarrow [0,0] \hookrightarrow s_*$ for $s =_{\mathrm{df}} \mathsf{enq}(x); \mathsf{deq}(y); \mathsf{deq}(z)$ and $s_* =_{\mathrm{df}} \mathsf{deq}(y); z := 0$. The Hoare logic for the error-admitting semantics proves $\{q = u +\!\!+ [v]\}\, s_*\, \{y = v \wedge z = 0\}$, so the Hoare logic for the error-compensating semantics proves $\{|q| = 1 \wedge q = u +\!\!+ [v]\}\, s\, \{|q| = 0 \wedge y = v \wedge z = 0\}$.

4 Related work

The standard approach to error repair is to have programs contain dedicated code for recovering from errors. Often, checkpointing and rollback mechanisms are employed to back up to a prior consistent state. Writing recovery code requires dedicated language constructs such as exception handling or recovery blocks [8]. This extra code can be tailored to the problem at hand, but needs to be crafted manually, so the approach is usually applied only to safety-critical or infrastructure software such as operating systems. It is difficult to give useful guarantees about recoveries and at any rate this requires reasoning about the particular recovery code written; moreover, run-time recovery can cause significant computational overheads. Micro-rebooting [2] eliminates the recovery programs and restarts only those parts of an application that have been affected by a runtime error, but this does not help against the re-occurrence of deterministic errors. In the Bristlecone programming language [3], a system is decomposed into tasks that can be executed and restarted independently as constrained by a system organization description. The recovery technique shares similarities with the assertion-based repair of data structures [4, 6] where the user formulates invariants for the data structures used. When a violation is detected, mutation of the corrupted data structure back into a good state is attempted based on hard-coded heuristics.

Our approach is fundamentally different in that it neither requires manual programming of run-time recovery using dedicated language constructs nor relies on a hard-coded heuristic run-time recovery mechanism. Instead, we use a compile-time program transformation, devised once and for all for a given kind of errors, with repair results rigorously controlled by a semantic guarantee. Our technique compares well to automatic datatype coercion (existing since Algol-68) where the compiler inserts conversion functions to repair wrong-operand-type errors. But this basic repair is driven by standard datatype information rather than by a dedicated repair type system.

The type-systematic approach to program optimization as used here (incl. mechanical transformability of correctness proofs) was introduced in our work [12, 11]. To assess the scalability of the method to complex and subtle optimizations, we applied it in particular to partial redundancy elimination (PRE) [14], an optimization reducing the number of times that each expression is evaluated. The best known modern version of the optimization is due to Knoop et al. [9]. The most advanced and well-motivated are those by Paleri et al. [10] and Xue and Knoop [15]. The file access repair is much inspired by Paleri et al.'s version of PRE relying on partial availability and partial anticipability analyses made conditional on the disjunction of total availability and anticipability (this is also the version we treated in [14]). Similarity-relational notions of validity of analyses and optimizations were pioneered in Benton's work [1] on relational Hoare logics for reasoning about pairs of programs.

Frade, Saabas and Uustalu [7] showed that analyses and optimizations described as type systems should be understood as applied versions of more foundational Hoare logics for reasoning about the same abstract semantics as the type system precisely rather than in terms of approximations or about more concrete computation trace or tree semantics.

Denney and Fischer [5] advocated certification of adherence to a policy in the Hoare logic for an instrumented semantics monitoring this policy.

5 Conclusions and future work

We have demonstrated a novel approach to mechanical, compile-time program repair. Its strengths derive from the firm semantic footing. As soon as the intended non-erroneous meaning of broken programs of a language has been cemented in the form of an error-compensating semantics (which is a psychological engineering issue), building program repair into a compiler reduces to identifying a sound program analysis and transformation (a mathematical problem). The challenge is to find a suitable program analysis with a suitable semantical interpretation. This can be subtle, as the file access example illustrated. As a reward, the type-systematic method, emphasizing general valid analyses over strongest analyses, makes soundness proofs relatively straightforward checks that also enjoy a useful logical counterpart in mechanical transformability of program correctness proofs.

One possible extension of our method beyond literal repair is enforcing of coding conventions. Here one would take the standard semantics of a language to be the error-compensating semantics, but as the error-admitting semantics use an instrumented version that monitors adherence to some coding conventions and aborts as soon as a violation is detected. A strong salient point is that the issue of a possibly wrong error-compensating semantics vanishes: the repair will be an uncontroversial refactoring of a program into a fully equivalent one meeting the coding conventions. As a variation on the theme of portability enforcing (cf. queue over/underflow repair), it can be meaningful to use an idealized semantics (rather than the "intersection" of all sensible implementations) as the error-admitting semantics. We could, e.g., transform programs relying on modular arithmetic for a certain base to programs behaving equivalently under ideal arithmetic within an interval (raising an error when an operation yields a value outside). The repair would augment programs with mod operations, but as few as necessary. Now it is useful to derive correctness proofs for given programs from those of transformed programs: a correctness proof is produced manually for the transformed program, assuming ideal arithmetic; the mechanical proof transformation will complete it with boilerplate modular reasoning.

Acknowledgements We are thankful to our referees for useful remarks. B. Fischer was supported by EPSRC grant no. EP/F052669/1. A. Saabas and T. Uustalu were supported by the Estonian Science Foundation grant no. 6940 and the EU FP6 IST integrated project no. 15905 MOBIUS.

References

[1] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. of 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2004*, pp. 14–25. ACM Press, 2004.

[2] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot—a technique for cheap recovery. In *Proc. of 6th Symp. on Operating System Design and Implementation, OSDI 2004*, pp. 31–44. Usenix Assoc., 2004.

[3] B. Demsky and A. Dash. Bristlecone: A language for robust software systems. In J. Vitek, ed., *Proc. of 22nd Europ. Conf. on Object-Oriented Program., ECOOP 2008*, *Lect. Notes in Comput. Sci.*, v. 5142, pp. 490–515. Springer, 2008

[4] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. on Softw. Engin.*, 32(12):931–951, 2006.

[5] E. Denney and B. Fischer. Correctness of source-level safety policies. In K. Araki, S. Gnesi, and D. Mandrioli, eds., *Proc. of 2003 Symp. of Formal Methods Europe, FME 2003*, *Lect. Notes in Comput. Sci.*, v. 2805, pp. 894–913. Springer, 2003.

[6] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex datastructures. In *Proc. of 22nd IEEE/ACM Int. Conf. on Automated Software Engineering*, pp. 64–73. ACM Press, 2007.

[7] M. J. Frade, A. Saabas, and T. Uustalu. Foundational certification of data-flow analyses. In *Proc. of 1st IEEE and IFIP Int. Symp. on Theoretical Aspects of Software Engineering, TASE 2007*, pp. 107-116. IEEE CS Press, 2007.

[8] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Proc. of Symp. on Operating Systems*, *Lect. Notes in Comput. Sci.*, v. 16, pp. 171–187. Springer, 1974.

[9] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: theory and practice. *ACM Trans. on Program. Lang and Syst.*, 16(4):1117-1155, 1994.

[10] V. K. Paleri, Y. N. Srikant, and P. Shankar. Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. *Sci. of Comput. Program.*, 48(1):1–20, 2003.

[11] A. Saabas. *Logics for low-level code and proof-preserving program transformations* (PhD thesis), *Thesis on Inform. and Syst. Engin.* C43. Tallinn Univ. of Techn., 2008.

[12] A. Saabas and T. Uustalu. Program and proof optimizations with type systems. *J. of Logic and Algebr. Program.*, 77(1–2):131–154, 2008.

[13] A. Saabas and T. Uustalu. Type systems for optimizing stack-based code. In M. Huisman and F. Spoto, eds., *Proc. of 2nd Wksh. on Bytecode Semantics, Verification, Analysis and Transformation, Bytecode 2007*, *Electron. Notes in Theor. Comput. Sci.*, v. 190(1), pp. 103–119. Elsevier, 2007.

[14] A. Saabas and T. Uustalu. Proof optimization for partial redundancy elimination. *J. of Logic and Algebr. Program.*, to appear. Conf. version in *Proc. of 2008 ACM SIGPLAN Wksh. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2008* , pp. 91–101. ACM Press, 2008.

[15] J. Xue and J. Knoop. A fresh look at PRE as a maximum flow problem. In A. Mycroft and A. Zeller, eds., *Proc. of 15th Int. Conf. on Compiler Construction, CC 2006*, *Lect. Notes in Comput. Sci.*, v. 3923, pp. 139–154. Springer, 2006.