Program Transformation in the Presence of Errors

Alexander Aiken John H. Williams Edward L. Wimmers

IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
email: *lastname*@ibm.com

1 Introduction

A laudable trend of the past two decades has been the increased use of denotational semantics to guide the design and implementation of programming languages. Semantics-driven language design has produced cleaner and simpler languages and provided more precise standards for testing the correctness of language implementations.

An apparent exception to this trend is the treatment of error handling. All too often, errors are considered to be outside the pale of the denotational semantics; if anything is specified about error behavior, it is usually through some *ad hoc* mechanism. Some language features—such as strong typing—reduce the negative impact of such a design but cannot avoid it completely; runtime errors exist in every language and must be handled in an implementation.

Many languages simply omit errors altogether from their formal semantic specification. For example, early Fortran implementations were free to report errors as the implementor saw fit, and transformations to improve performance could change the behavior of error-producing programs. This approach is still taken with some modern languages, e.g., Haskell [HWA*88] and the languages of Turner [Tur85], where strong typing and lazy evaluation lessen the need for error recovery and exception handling.

Other languages include errors in the domain of values and provide some mechanism for computing with (or recovering from) errors, but the formal specification allows considerable variation in the behavior of implementations. For example, in [Ste84] Steele writes:

> "The definition of Common Lisp ...explicitly requires the interpreter and compiler to impose identical semantics on *correct* programs *so far as possible*." (emphasis added)

Indeed, in one Common Lisp implementation, taking the `car` of an atom produces a run time error when interpreted but returns the current package when compiled!

One study of program transformations carefully accounts for the fact that many common optimizations do not preserve error behavior by giving a precise, denotational treatment of such transformations [CF89]. In this approach, errors are considered to approximate other values, and program transformations are lifting operations that can (possibly) make programs more defined. Another approach [Hen81]

> "...examines the optimization difficulties imposed by common exception handling facilities [and] proposes restrictions on these mechanisms that make the optimization of programs possible."

Why have language designers and implementors avoided specifying and preserving the meaning of errors? The answer appears to be that not preserving error behavior increases the power and effectiveness of transforming correct programs, i.e., the "important" programs. For instance, substituting `s2` (which selects the second element of a sequence) for `s1` $\circ$ `tl` (which selects the first element of the tail of a sequence) may improve a program's running time, but a programmer who hadn't used the primitive `s2` and was unaware of its existence would be confused by the run-time error message "`s2` incorrectly applied to a non-sequence argument."

If the language designer opts for language clarity and ease of debugging by making a semantic distinction between `tl` errors and `s2` errors, then the general transformation becomes invalid, and some weaker version must be substituted, perhaps in the form of a set of rules identifying particular contexts in which the replacement is valid. This can be a significant loss, since identifying such contexts in general requires knowledge of the entire program. Thus, including errors as semantic objects in order to make a language easier to use appears to weaken the generality and power of the language's program transformations.

This paper presents a technique for preserving the power of general program transformations in the presence of a rich collection of distinguishable error values. This is accomplished by introducing an annotation, "`Safe`", to mark occurrences of functions that cannot produce errors. Succinct and general algebraic laws can be expressed using `Safe`, thereby giving program transformations in a language with many error values the same power and generality as program transformations in a language with only a single error value (such as FP [Bac78]). In fact, the Safe mechanism accomplishes much more. It actually strengthens equational reasoning by providing a sufficient condition on a program context $\mathtt{E}(\cdot)$ and functions `f` and `g`, such that $\mathtt{E}(\mathtt{f}) \equiv \mathtt{E}(\mathtt{g})$ even if $\mathtt{f} \not\equiv \mathtt{g}$.

The Safe mechanism is presented in the context of the functional language FL [BWW86], but it should be applicable to any language whose program transformations can be expressed equationally. Section 2 describes enough of FL to illustrate the technique and prove its soundness. Section 3 introduces the Safe mechanism and gives a simple example illustrating that having just two distinguishable errors causes as much loss of algebraic generality as having arbitrarily many different kinds of errors. This shows that it is not possible through careful language design to make a gradual tradeoff between the expressiveness of error reporting and algebraic generality. Section 3 also contains the Substitutability Theorem, which provides a criterion for proving the soundness of transformations involving `Safe`. Some examples of optimization using `Safe` are given in Section 4. Section 6 concludes with suggestions for further work.

$$
\begin{array}{rcl}
\mathtt{f{:}x} & & \text{denotes function application} \\
\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle & & \text{denotes sequence construction} \\
(\mathtt{f} \circ \mathtt{g}){:}\mathtt{x} & = & \mathtt{f}{:}(\mathtt{g}{:}\mathtt{x}) \\
[\mathtt{f}_1, \ldots, \mathtt{f}_n]{:}\mathtt{x} & = & \langle \mathtt{f}_1{:}\mathtt{x}, \ldots, \mathtt{f}_n{:}\mathtt{x} \rangle \\
(\mathtt{p} \rightarrow \mathtt{q}; \mathtt{r}){:}\mathtt{x} & = & \begin{cases} \mathtt{p}{:}\mathtt{x} & \text{if } \mathtt{p}{:}\mathtt{x} \in \mathcal{E}_{\mathrm{FL}} \\ \mathtt{r}{:}\mathtt{x} & \text{if } \mathtt{p}{:}\mathtt{x} = \underline{\mathtt{false}} \\ \mathtt{q}{:}\mathtt{x} & \text{otherwise} \end{cases} \\
\tilde{}\mathtt{x}{:}\mathtt{y} & = & \mathtt{x} \\
\alpha{:}\mathtt{f}{:}\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle & = & \langle \mathtt{f}{:}\mathtt{x}_1, \ldots, \mathtt{f}{:}\mathtt{x}_n \rangle \\
+{:}\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle & = & \mathtt{x}_1 + \ldots + \mathtt{x}_n \\
\mathtt{si}{:}\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle & = & \mathtt{x}_i \\
\mathtt{tl}{:}\langle \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \rangle & = & \langle \mathtt{x}_2, \ldots, \mathtt{x}_n \rangle \\
\mathtt{rev}{:}\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle & = & \langle \mathtt{x}_n, \ldots, \mathtt{x}_1 \rangle \\
\mathtt{al}{:}\langle \mathtt{x}, \langle \mathtt{y}_1, \ldots, \mathtt{y}_n \rangle \rangle & = & \langle \mathtt{x}, \mathtt{y}_1, \ldots, \mathtt{y}_n \rangle \\
\mathtt{distl}{:}\langle \mathtt{x}, \langle \mathtt{y}_1, \ldots, \mathtt{y}_n \rangle \rangle & = & \langle \langle \mathtt{x}, \mathtt{y}_1 \rangle, \ldots, \langle \mathtt{x}, \mathtt{y}_n \rangle \rangle \\
\mathtt{id}{:}\mathtt{x} & = & \mathtt{x} \\
\mathtt{dom}{:}\mathtt{f} & = & \mathtt{s1} \circ [\mathtt{id}, \mathtt{f}] \\
\mathtt{mkerr}{:}\mathtt{"string"} & = & \mathtt{string}_{\mathtt{err}} \\
\mathtt{catch}{:}\langle \mathtt{f}, \mathtt{g} \rangle{:}\mathtt{x} & = & \begin{cases} \mathtt{g}{:}\langle \mathtt{x}, \mathtt{y} \rangle & \text{if } \mathtt{f}{:}\mathtt{x} = \mathit{Err}(\mathtt{y}) \\ \mathtt{f}{:}\mathtt{x} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 1: A subset of FL.

2 An Overview of FL

FL [BWW86] is the result of an effort to design a practical functional language based on FP [Bac78]. Figure 1 gives the subset of FL needed to understand the laws and examples that follow. Some features of the language are ignored altogether; in particular, input/output functions and syntactic sugar are omitted. A definition of each function is given only for some arguments; for all other arguments, the function $\mathtt{f}$ returns an error value $\mathtt{f}_{\mathtt{err}}$ (e.g., $\mathtt{s1{:}0} \equiv \mathtt{s1}_{\mathtt{err}}$).[1] The evaluation order is leftmost-innermost; thus, in $[\mathtt{f}, \mathtt{g}]{:}\mathtt{x}$, $\mathtt{f{:}x}$ is evaluated and then $\mathtt{g{:}x}$ is evaluated.

In designing FL, it was recognized that one of the deficiencies of FP is that it has a single error message $\perp$ (or *Wrong!*) for all exceptional circumstances. Error messages and exception handling are an integral part of FL; as in the current version of Standard ML [HMT89], errors are first class values rather than the special results of functions that fail to produce values. Semantically, error values in FL are treated differently than ordinary values. All functions are strict with respect to errors; that is, $\mathtt{f{:}x}_{\mathtt{err}} \equiv \mathtt{x}_{\mathtt{err}}$ for any function $\mathtt{f}$ and error value $\mathtt{x}_{\mathtt{err}}$. Sequence construction is also strict with respect to errors; a sequence collapses to the leftmost error it contains. This behavior is justified by

[1] In fact, FL functions produce more informative errors than just the name of the function, but this countable set of errors is sufficient to illustrate the technique.

the intended use of errors in FL: errors represent a situation in which something extraordinary has happened, and therefore an error should persist until caught or until it escapes from (and becomes the result of) the program. Some of the semantic treatment of errors can be seen in the recursive domain equations for FL:

$$\begin{aligned}
\mathcal{D}_{\mathrm{FL}} &= \mathcal{D}^{+}_{\mathrm{FL}} \cup \mathcal{E}_{\mathrm{FL}} \\
\mathcal{D}^{+}_{\mathrm{FL}} &= A \cup \mathit{Seqs}(\mathcal{D}^{+}_{\mathrm{FL}}) \cup (\mathcal{D}^{+}_{\mathrm{FL}} \to \mathcal{D}_{\mathrm{FL}}) \\
&\quad \textit{(the ordinary values)} \\
\mathcal{E}_{\mathrm{FL}} &= \mathit{Err}(\mathcal{D}^{+}_{\mathrm{FL}}) \cup \{\bot\} \\
&\quad \textit{(the error values)}
\end{aligned}$$

In these equations, A is the set of atoms, *Seqs* is sequence construction, and *Err* is error construction. The ordering on $\mathcal{D}^{+}_{\mathrm{FL}}$ is the standard one; in $\mathcal{E}_{\mathrm{FL}}$, $\mathit{Err}(x) \leq \mathit{Err}(y) \Leftrightarrow x \leq y$ and $\bot \leq x$ for all x.

3 The Safe Mechanism

One of the principles underlying FL is that a programming language should have a rich algebra useful for reasoning about and optimizing programs. Errors have a great impact on the algebra; for example, if two expressions can produce distinct errors, then the order of evaluation of the expressions usually cannot be changed without changing the error produced. Even with errors, however, there are many general identities that hold between FL programs; as usual, $\mathtt{f} \equiv \mathtt{g}$ means that f and g denote the same semantic value.

$$\begin{aligned}
\mathtt{f} \circ \mathtt{id} &\equiv \mathtt{f} && (1)\\
\mathtt{id} \circ \mathtt{f} &\equiv \mathtt{f} && (2)\\
[\mathtt{f}_1, \ldots, \mathtt{f}_n] \circ \mathtt{g} &\equiv [\mathtt{f}_1 \circ \mathtt{g}, \ldots, \mathtt{f}_n \circ \mathtt{g}] && (3)\\
\mathtt{f} \circ (\mathtt{p} \to \mathtt{q}; \mathtt{r}) &\equiv \mathtt{p} \to \mathtt{f} \circ \mathtt{q}; \mathtt{f} \circ \mathtt{r} && (4)\\
(\mathtt{p} \to \mathtt{q}; \mathtt{r}) \circ \mathtt{f} &\equiv \mathtt{p} \circ \mathtt{f} \to \mathtt{q} \circ \mathtt{f}; \mathtt{r} \circ \mathtt{f} && (5)
\end{aligned}$$

These laws hold because the order of application of the component functions is unchanged. However,

$$[\mathtt{g}, \mathtt{f}] \equiv \mathtt{rev} \circ [\mathtt{f}, \mathtt{g}] \qquad (6)$$

which is a law in FP, is not valid for all FL functions f and g; if f produces $\bot$ and g produces $\mathtt{tl}_{\mathtt{err}}$, then $[\mathtt{g}, \mathtt{f}]$ produces $\mathtt{tl}_{\mathtt{err}}$, but $\mathtt{rev} \circ [\mathtt{f}, \mathtt{g}]$ produces $\bot$. This example shows there cannot be a gradual trade-off of expressiveness of error reporting for generality of program transformations. Having two errors is as limiting as having arbitrarily many, since the existence of just one error distinguishable from $\bot$ is sufficient to invalidate any "law" that does not preserve the order of evaluation of its constituent functions. However, it is true that

$$[\mathtt{g}, \mathtt{f}] \equiv \mathtt{rev} \circ [\mathtt{f}, \mathtt{g}] \text{ if neither side makes an error} \qquad (7)$$

That is, there are *contexts* in which $[\mathtt{g},\mathtt{f}]$ can be substituted for $\mathtt{rev}\circ[\mathtt{f},\mathtt{g}]$. There are many other examples of rewrite rules that are correct provided neither side produces an error, and including them greatly enhances the power of a program transformation system. The following informal examples illustrate the notion of "rewriting in context", i.e., using rules whose validity depends on the context in which they are applied. In these examples, an occurrence of a function `f` is annotated as being "safe" (written $\mathtt{S}(\mathtt{f})$), if it is known that that occurrence is guaranteed not to produce an error when applied to a non-error value; the notation $\mathtt{f}\mapsto\mathtt{g}$ simply indicates that `f` is rewritten to `g`. (N.B. For now, $\mathtt{S}(\mathtt{f})$ is an extra-linguistic notion; the phrase "annotating `f` with $\mathtt{S}(\mathtt{f})$ " has no more semantic content than the phrase "painting `f` green".)

Consider the program $\mathtt{rev}\circ[\tilde{}\mathtt{0},\tilde{}\mathtt{1}]$. Since the two constant functions cannot produce an error unless applied to an error, neither can the construction of the two constant functions. Thus the program can be rewritten:

$$\begin{aligned}\mathtt{rev}\circ[\tilde{}\mathtt{0},\tilde{}\mathtt{1}] &\mapsto\\ \mathtt{rev}\circ[\mathtt{S}(\tilde{}\mathtt{0}),\mathtt{S}(\tilde{}\mathtt{1})] &\mapsto\\ \mathtt{rev}\circ\mathtt{S}([\tilde{}\mathtt{0},\tilde{}\mathtt{1}])&\end{aligned}$$

Now note that because the argument to reverse is safe, the order of evaluation of the elements of the sequence must be irrelevant, so the application of `rev` can be eliminated. As a last step, the annotation `S` can be dropped:

$$\begin{aligned}\mathtt{rev}\circ\mathtt{S}([\tilde{}\mathtt{0},\tilde{}\mathtt{1}]) &\mapsto\\ \mathtt{S}([\tilde{}\mathtt{1},\tilde{}\mathtt{0}]) &\mapsto\\ [\tilde{}\mathtt{1},\tilde{}\mathtt{0}]&\end{aligned}$$

This rewriting sequence is correct in the sense that the final program is equivalent to the original program; i.e., $\mathtt{rev}\circ[\tilde{}\mathtt{1},\tilde{}\mathtt{0}]\equiv[\tilde{}\mathtt{1},\tilde{}\mathtt{0}]$. At this point, however, the safe mechanism is informal, and it is easy to make mistakes. Consider the program $\mathtt{rev}\circ[\mathtt{distl},\mathtt{tl}\circ\mathtt{al}]$. Because `al` follows `distl` in the order of evaluation, and because `al` produces an error for exactly the same arguments as `distl`, `al` can be marked safe.

$$\mathtt{rev}\circ[\mathtt{distl},\mathtt{tl}\circ\mathtt{al}]\ \mapsto\ \mathtt{rev}\circ[\mathtt{distl},\mathtt{tl}\circ\mathtt{S}(\mathtt{al})]$$

Next notice that if `al` is error-free, then `tl` always succeeds and returns the second component of the original argument. Therefore $\mathtt{S}(\mathtt{s2})$ can be substituted for $\mathtt{tl}\circ\mathtt{S}(\mathtt{al})$.

$$\mathtt{rev}\circ[\mathtt{distl},\mathtt{tl}\circ\mathtt{S}(\mathtt{al})]\mapsto\mathtt{rev}\circ[\mathtt{distl},\mathtt{S}(\mathtt{s2})]$$

At this point, one might suppose that `rev` can be eliminated as in the previous example, since only one of the functions in the sequence to be reversed can produce an error and therefore the evaluation order of the elements of the sequence is irrelevant:

$$\mathtt{rev}\circ[\mathtt{distl},\mathtt{S}(\mathtt{s2})]\ \stackrel{?}{\mapsto}\ [\mathtt{S}(\mathtt{s2}),\mathtt{distl}]$$

Now, however, something has gone wrong: $\texttt{rev} \circ [\texttt{distl}, \texttt{tl} \circ \texttt{al}] \not\equiv [\texttt{s2}, \texttt{distl}]$, because $\texttt{rev} \circ [\texttt{distl}, \texttt{tl} \circ \texttt{al}] : 3$ produces $\texttt{distl}_{\texttt{err}}$, whereas $\texttt{rev} \circ [\texttt{s2}, \texttt{distl}] : 3$ produces $\texttt{s2}_{\texttt{err}}$ (which isn't even mentioned in the original program). The fact that intuition can fail on such a small and simple example is strong motivation to provide a precise formalism for stating and verifying these transformations.

A first step towards formalizing the rewrite rules is to express qualified laws such as (7) equationally. One way to do this is to force both sides to produce some identical "don't care" value for all arguments not in the domain of interest. In the following definition of Safe, $\perp$ plays the role of the "don't care" value.

Definition 3.1 (Safe) For every FL function f, Safe:f denotes the function:

$$\texttt{Safe:f:x} = \begin{cases} \texttt{x} & \text{if } \texttt{x} \in \mathcal{E}_{\text{FL}} \\ \perp & \text{if } \texttt{f:x} \in \mathcal{E}_{\text{FL}} \\ \texttt{f:x} & \text{otherwise} \end{cases}$$

For convenience, Safe:f is often abbreviated S:f. With this definition of Safe, Law (7) can be expressed as:

$$\texttt{S:}[\texttt{g}, \texttt{f}] \equiv \texttt{S:}(\texttt{rev} \circ [\texttt{f}, \texttt{g}]) \tag{8}$$

Moreover, many other useful laws are expressible:

$$\begin{aligned}
\texttt{f} &\equiv \texttt{S:f} \circ \texttt{dom:f} && (9)\\
\texttt{f} \circ \texttt{S:}(\texttt{dom:f}) &\equiv \texttt{S:f} && (10)\\
\texttt{S:f} \circ \texttt{S:g} &\equiv \texttt{S:}(\texttt{f} \circ \texttt{g}) && (11)\\
\texttt{S:p} \rightarrow \texttt{S:q;S:r} &\equiv \texttt{S:}(\texttt{p} \rightarrow \texttt{q;r}) && (12)\\
[\texttt{f}, \texttt{g}] &\equiv [\texttt{f}, \texttt{g} \circ \texttt{S:}(\texttt{dom:f})] && (13)\\
[\texttt{S:f}_1, \ldots, \texttt{S:f}_n] &\equiv \texttt{S:}[\texttt{f}_1, \ldots, \texttt{f}_n] && (14)\\
\texttt{rev} \circ \texttt{S:}[\texttt{f}_1, \ldots, \texttt{f}_n] &\equiv \texttt{S:}[\texttt{f}_n, \ldots, \texttt{f}_1] && (15)\\
\texttt{S:}(\texttt{s1} \circ \texttt{tl}) &\equiv \texttt{S:s2} && (16)\\
\texttt{catch:}\langle \texttt{S:f}, \texttt{g} \rangle &\equiv \texttt{S:f} && (17)\\
\texttt{catch:}\langle \texttt{f}, \texttt{g} \rangle \circ \texttt{S:}(\texttt{dom:h}) &\equiv \texttt{catch:}\langle \texttt{f} \circ \texttt{S:}(\texttt{dom:h}), \texttt{g} \rangle && (18)
\end{aligned}$$

Note that some of the informal $\mapsto$ steps have been captured as equivalences; e.g., Laws (14) and (15). Unfortunately, others cannot be expressed as equivalences. For example, the rule $\texttt{tl} \circ \texttt{S(al)} \mapsto \texttt{S(s2)}$ cannot be written as $\texttt{tl} \circ \texttt{S:al} \equiv \texttt{s2}$, since $(\texttt{tl} \circ \texttt{S:al}) : \langle 1, 2, 3 \rangle \equiv \texttt{al}_{\texttt{err}}$ whereas $\texttt{s2:}\langle 1, 2, 3 \rangle \equiv 2$; nor could it be $\texttt{tl} \circ \texttt{S:al} \equiv \texttt{S:s2}$, for the same reason.

The difficulty is that $\equiv$ is a symmetric relation, whereas the desired property is inherently asymmetric: $\texttt{tl} \circ \texttt{S:al}$ can be rewritten to S:s2, because in any program in which $\texttt{tl} \circ \texttt{S:al}$ could appear, the function S:s2 produces the same result. Thus, at least some of the desired rewrite rules $\texttt{f} \mapsto \texttt{g}$ are valid only when f appears in a "good" context; i.e., in one which enforces the condition that S:f cannot produce $\perp$. The following definitions develop the relation $\triangleright$ (read "rewrites in context to"),

which both captures this asymmetry and defines a set of contexts in which such rewrite rules can be applied.

Definition 3.2

1. The set of *simple expressions* is the smallest set of FL functions such that:
   - `si`, `tl`, `rev`, `al`, `distl`, `id`, `~a` for all atoms `a`, and `mkerr` are simple expressions.
   - If $\mathtt{e}_1, \ldots, \mathtt{e}_n$ are simple expressions, then $\mathtt{e}_1 \circ \mathtt{e}_2$, $[\mathtt{e}_1, \ldots, \mathtt{e}_n]$, $(\mathtt{e}_1 \rightarrow \mathtt{e}_2; \mathtt{e}_3)$, $\mathtt{dom}:\mathtt{e}_1$, $\mathtt{catch}:\langle \mathtt{e}_1, \mathtt{e}_2 \rangle$, $\mathtt{Safe}:\mathtt{e}_1$, and $\alpha:\mathtt{e}_1$ are simple expressions.
2. `E` is a *simple context* iff `E(f)` is a simple expression for every simple expression `f`.

Since only the language processor introduces and manipulates `Safe` expressions, and since transformations preserve the meaning of expressions, the transformations need to work only for expressions that can be written without `Safe`.

Definition 3.3 A simple expression `f` is *user-definable* iff there exists a simple expression `u` with no occurrences of `Safe` such that $\mathtt{f} \equiv \mathtt{u}$.

Definition 3.4 Let `f, g` be simple expressions. $\mathtt{f} \rhd \mathtt{g}$ iff for every simple context `E` such that `E(f)` is user-definable, $\mathtt{E(f)} \equiv \mathtt{E(g)}$.

This definition of $\rhd$ allows the expression of a large number of "in-context" transformations. Note that (22) is a correct version of the incorrect rewriting step discussed above.

$$\mathtt{tl} \circ \mathtt{S}:\mathtt{al} \rhd \mathtt{s2} \tag{19}$$
$$\mathtt{S}:\mathtt{f} \rhd \mathtt{f} \tag{20}$$
$$\mathtt{dom}:(\mathtt{S}:\mathtt{f}) \rhd \mathtt{id} \tag{21}$$
$$\mathtt{rev} \circ [\mathtt{S}:\mathtt{f}, \mathtt{g}] \rhd [\mathtt{g}, \mathtt{S}:\mathtt{f}] \tag{22}$$
$$\alpha:\mathtt{f} \circ \alpha:(\mathtt{S}:\mathtt{g}) \rhd \alpha:(\mathtt{f} \circ \mathtt{S}:\mathtt{g}) \tag{23}$$
$$\mathtt{dom}:(\mathtt{S}:(\alpha:\mathtt{f})) \rhd \alpha:(\mathtt{S}:(\mathtt{dom}:\mathtt{f})) \tag{24}$$

There is one nagging problem. Definition 3.4 provides little assistance in establishing that $\mathtt{f} \rhd \mathtt{g}$, because it requires reasoning about all possible contexts. The purpose of the Substitutability Theorem (given below) is to provide a sufficient condition that is easier to check. The following definition gives this condition; the idea is that `f` should rewrite to `g` if `f` and `g` agree wherever `f` does not return the "don't care" value.

Definition 3.5 $\mathtt{f} \leq_s \mathtt{g}$ iff $\mathtt{f}:\mathtt{x} \equiv \mathtt{g}:\mathtt{x}$ whenever $\mathtt{f}:\mathtt{x} \not\equiv \bot$.

The following two lemmas precisely capture the properties of simple expressions that are needed to make the technique of "in-context substitutions" work.

Lemma 3.6 If E is a simple context and $f \leq_s g$, then $E(f) \leq_s E(g)$.

Lemma 3.7 If e is a user-definable simple expression, then $e{:}x \not\equiv \bot$ whenever $x \not\equiv \bot$.

The Substitutability Theorem reduces the problem of verifying that $f \rhd g$ to the easier problem of checking that $f \leq_s g$ in the case where $E(f)$ is a user-definable simple expression.

Theorem 3.8 (Substitutability Theorem)
If f and g are simple expressions and $f \leq_s g$, then $f \rhd g$.

Proof: Let E be a simple context and assume that $E(f)$ is user-definable. By Lemma 3.6, $E(f) \leq_s E(g)$. By Lemma 3.7, $E(f){:}x \not\equiv \bot$ if $x \not\equiv \bot$. Together these facts imply that $E(f){:}x \equiv E(g){:}x$ if $x \not\equiv \bot$. By strictness, $E(f){:}\bot \equiv \bot \equiv E(g){:}\bot$. Therefore, $E(f) \equiv E(g)$. □

Using the Substitutability Theorem, transformations (19)-(24) are easily verified. Note that the proof of the Substitutability Theorem depends only on Lemmas 3.6 and 3.7; therefore, this approach works with any extension of the simple expressions that preserves these two Lemmas. Also note that pure identities, such as transformations (1)-(17), apply to *all* expressions, not merely the simple expressions.

4 Using Safe

This section shows the usefulness of Safe with a few short examples illustrating the elimination of function calls, the use of Safe in code generation, the optimization of exception handling, and the use of in-context laws. Recall that $dom{:}f{:}x$ is $f{:}x$ if $f{:}x$ is an error value and x otherwise (see Figure 1). The examples use the following laws involving dom:

$$dom{:}id \equiv id \tag{25}$$
$$dom{:}[f_1, \ldots, f_n] \equiv dom{:}f_n \circ \ldots \circ dom{:}f_1 \tag{26}$$
$$dom{:}al \circ dom{:}distl \equiv dom{:}distl \tag{27}$$
$$S{:}(dom{:}tl) \equiv S{:}(dom{:}s1) \tag{28}$$

In the first example, nothing is known about f, but the fact that id is a total function allows the construction of the two functions to be reversed.

$$\begin{array}{lll}
 & rev \circ [f, id] & \\
\equiv & rev \circ S{:}[f, id] \circ dom{:}[f, id] & \text{by 9} \\
\equiv & S{:}[id, f] \circ dom{:}[f, id] & \text{by 15} \\
\equiv & S{:}[id, f] \circ dom{:}id \circ dom{:}f & \text{by 26} \\
\equiv & S{:}[id, f] \circ id \circ dom{:}f & \text{by 25} \\
\equiv & S{:}[id, f] \circ dom{:}f & \text{by 2} \\
\equiv & S{:}[id, f] \circ dom{:}f \circ id & \text{by 1} \\
\equiv & S{:}[id, f] \circ dom{:}f \circ dom{:}id & \text{by 25} \\
\equiv & S{:}[id, f] \circ dom{:}[id, f] & \text{by 26} \\
\equiv & [id, f] & \text{by 9}
\end{array}$$

This transformation is an optimization, because the end result eliminates the application of `rev`. Note, however, that the intermediate steps are not necessarily optimizations, because they involve computing some values twice; in particular, `f` could be arbitrarily expensive to compute. The law $\texttt{f} \equiv \texttt{S:f} \circ \texttt{dom:f}$ is very useful for introducing safe functions, but if the added `dom` cannot be discharged, then the resulting program could be less efficient than the original program. In the worst case, using Law 9 could result in a program that computes `f` once very slowly to preserve errors and then once again to produce the result!

The next example illustrates that there are intermediate cases where `dom` cannot be completely discharged but the result is still a useful optimization. Section 6 contains further discussion of the problem of discharging `dom`.

$$
\begin{array}{lll}
 & \texttt{rev} \circ [\texttt{distl}, \texttt{al}] & \\
\equiv & \texttt{rev} \circ \texttt{S:}[\texttt{distl}, \texttt{al}] \circ \texttt{dom:}[\texttt{distl}, \texttt{al}] & \text{by 9} \\
\equiv & \texttt{S:}[\texttt{al}, \texttt{distl}] \circ \texttt{dom:}[\texttt{distl}, \texttt{al}] & \text{by 15} \\
\equiv & \texttt{S:}[\texttt{al}, \texttt{distl}] \circ \texttt{dom:al} \circ \texttt{dom:distl} & \text{by 26} \\
\equiv & \texttt{S:}[\texttt{al}, \texttt{distl}] \circ \texttt{dom:distl} & \text{by 27} \\
\equiv & [\texttt{S:al}, \texttt{S:distl}] \circ \texttt{dom:distl} & \text{by 14}
\end{array}
$$

In this case, the end result is an optimization not only because the application of `rev` is eliminated, but also because `dom:distl` permits the rest of the program to be executed without checking the arguments of any of the functions. The use of `Safe` makes it easy for a code generator to take advantage of this fact. When a primitive is marked as safe, a code generator can produce a version of the primitive that does not check its argument; in this example, both `al` and `distl` can run unchecked.

The following example presents a more substantial optimization (similar to loop-jamming optimizations for imperative languages [ASU86]) and illustrates the use of in-context laws:

$$
\begin{array}{lll}
 & [\alpha\texttt{:s1}, \alpha\texttt{:+} \circ \alpha\texttt{:tl}] & \\
\equiv & [\alpha\texttt{:s1}, \alpha\texttt{:+} \circ \alpha\texttt{:tl} \circ \texttt{S:}(\texttt{dom:}(\alpha\texttt{:s1}))] & \text{by 13} \\
\equiv & [\alpha\texttt{:s1}, \alpha\texttt{:+} \circ \alpha\texttt{:tl} \circ \alpha\texttt{:}(\texttt{S:}(\texttt{dom:s1}))] & \text{by 24} \\
\equiv & [\alpha\texttt{:s1}, \alpha\texttt{:+} \circ \alpha\texttt{:}(\texttt{tl} \circ \texttt{S:}(\texttt{dom:s1}))] & \text{by 23} \\
\equiv & [\alpha\texttt{:s1}, \alpha\texttt{:+} \circ \alpha\texttt{:}(\texttt{tl} \circ \texttt{S:}(\texttt{dom:tl}))] & \text{by 28} \\
\equiv & [\alpha\texttt{:s1}, \alpha\texttt{:+} \circ \alpha\texttt{:}(\texttt{S:tl})] & \text{by 10} \\
\equiv & [\alpha\texttt{:s1}, \alpha\texttt{:}(\texttt{+} \circ \texttt{S:tl})] & \text{by 23}
\end{array}
$$

Even though the second, third, and last steps use in-context transformations, these steps are actually equivalences, because they occur in simple contexts. Since the first and last lines are equivalent they can be substituted freely one for the other in any program. Note that this shows that it is not necessary that the entire program be simple for an in-context law to apply—it is sufficient that in-context laws be used within simple sub-expressions.

The final example illustrates how `Safe` is used to optimize exception handling. Suppose a programmer defines a function `newtl` that returns the empty sequence whenever `tl` would return an error. A simple definition of `newtl` is `catch:`$\langle$`tl, []`$\rangle$. Now, if `newtl` appears in a context where it always gets an argument in the proper domain of `tl`, `newtl` can be transformed to `S:tl` as follows:

	`newtl` $\circ$ `S:(dom:tl)`	
$\equiv$	`catch:`$\langle$`tl, []`$\rangle$ $\circ$ `S:(dom:tl)`	by def. of `newtl`
$\equiv$	`catch:`$\langle$`tl` $\circ$ `S:(dom:tl), []`$\rangle$	by 18
$\equiv$	`catch:`$\langle$`S:tl, []`$\rangle$	by 10
$\equiv$	`S:tl`	by 17

5 Related Work

Safety analysis bears some resemblance to *projection analysis* [WH87]. Both techniques deal with manipulating "annotations". For projection analysis, these annotations are projections; for safety analysis, the annotation is the function `Safe`. Many of the techniques for manipulating the annotations are also similar; however, safety analysis and projection analysis are addressed at two different problems. Projection analysis is primarily concerned with determining (in a lazy system) whether a function is strict in its arguments, and gives a nice way of addressing that problem. Safety analysis (as presented here for a strict language) is concerned not only with determining when a function is "safe" but also with trying to use that fact to facilitate other source-level optimizations.

6 Conclusions and Future Work

The Safe mechanism resolves the tension between the desire to make functional programs run fast through optimization and the desire to have a language in which it is easy to write and debug programs. This tension is perhaps at a maximum in FL, because no distinction is made between user-generated exceptions and system-generated errors—both are legitimate error values. Thus, it would be disastrous for an FL compiler to fail to preserve the error behavior of a program; on the other hand, preserving errors creates problems for optimization. `Safe` solves this dilemma by providing a way to express the program transformations of a language with a single error value in a language with many error values.

The transformations involving `Safe` are useful only if `Safe` functions can be introduced by a compiler. While it remains to be proven that `Safe` enables large scale optimization in practice, one practical problem is readily apparent from the examples in this paper. As noted in Section 4, it is desirable to introduce `Safe` without using the law `f` $\equiv$ `S:f` $\circ$ `dom:f`, which requires computing `f` twice. In this law, `dom:f` ensures that the *type* of `S:f` is such that `S:f` cannot produce an error. Thus,

discharging `dom:f` is a type inference problem—a type inference algorithm can check whether `f` is safe before Law 9 is applied, and if `f` is safe the `dom` need not be introduced.

It would strengthen the theoretical treatment if the restriction of in-context laws to simple expressions could be removed. While this is not critical from a practical point of view—the class of simple expressions covers many commonly occurring situations—the added generality may help clarify the semantic role of `Safe` and add some power to the algebra.

7 Acknowledgements

The congenial atmosphere of the FL group (John Backus, Thom Linden, Peter Lucas, and Paul Tucker) contributed significantly to this work. It is also a pleasure to thank Luca Cardelli, Robert Cartwright, David Chase, Jim Donahue, Joe Halpern, Paul Hudak, Matthias Felleisen, Phil Wadler, Jennifer Widom, and the members of IFIP WG 2.8 for their helpful comments and suggestions. In particular, Joe Halpern's persistent critiques greatly improved the presentation of Section 3.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[BWW86] J. Backus, J. H. Williams, and E. L. Wimmers. *The FL Language Manual.* Technical Report RJ 5339 (54809), IBM, 1986.

[CF89] R. Cartwright and M. Felleisen. The semantics of program dependence. In *Proceedings of the 1989 Conference on Programming Language Design and Implementation*, June 1989.

[Hen81] J. Hennessy. Program optimization and exception handling. In *Proceedings of the 1981 Symposium on Principles of Programming Languages*, January 1981.

[HMT89] R. Harper, R. Milner, and M. Tofte. *The Definition of Standard ML—Version 3.* Technical Report ECFS-LFCS-89-81, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[HWA*88] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, J. Hughes, T. Johnsson, D. Kieburtz, S. P. Jones, R. Nikhil, M. Reeve, D. Wise, and J. Young. *Report on the Functional Programming Language Haskell.* Technical Report DCS/RR-666, Yale University, December 1988.

[Ste84] G. L. Steele. *Common Lisp: The Language*. Digital Press, 1984.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the IFIP International Conference on Functional Programming and Computer Architecture*, Springer Verlag Lecture Notes in Computer Science no. 201, 1985.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Proceedings of the Symposium on Functional Programming Languages and Computer Architecture*, pages 385–407, Springer Verlag Lecture Notes in Computer Science no. 274, September 1987.