Safe—A Semantic Technique for Transforming Programs in the Presence of Errors

ALEXANDER AIKEN
University of California, Berkeley
and
JOHN H. WILLIAMS and EDWARD L. WIMMERS
IBM Almaden Research Center

Language designers and implementors have avoided specifying and preserving the meaning of programs that produce errors. This is apparently because being forced to preserve error behavior severely limits the scope of program optimization, even for correct programs. However, preserving error behavior is desirable for debugging, and error behavior must be preserved in any language that permits user-generated errors (i.e., exceptions).

This paper presents a technique for expressing general program transformations for languages that possess a rich collection of distinguishable error values. This is accomplished by defining a higher-order function called "`Safe`", which can be used to annotate those portions of a program that are guaranteed not to produce errors. It is shown that this facilitates the expression of very general program transformations, effectively 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.

Using the semantic properties of `Safe`, it is possible to provide some useful sufficient conditions for establishing the correctness of transformations in the presence of errors. In particular, a Substitutability Theorem is proven, which can be used to justify "in-context" optimizations—transformations that alter the meanings of subexpressions without changing the meaning of the whole program. Finally, the effectiveness of the technique is demonstrated by some examples of its use in an optimizing compiler.

Categories and Subject Descriptors: D.3.4 [**Processors**]: Optimization

General Terms: Languages,Theory

Additional Key Words and Phrases: equational reasoning,exceptions, program optimization, program transformation

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

---

Address for first author: Computer Science Division, 773 Soda Hall, University of California, Berkeley, 94720-1776, email: aiken@cs.berkeley.edu. Address for second and third authors: IBM Almaden Research Center, 650 Harry Rd., San Jose, CA, 95120-6077, email: {williams,wimmers}@almaden.ibm.com. A preliminary version of this paper was presented at the 1990 POPL Conference [Aiken et al. 1990].

often, errors are considered to be outside the scope 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 [Hudak et al. 1988] and the languages of Turner [Turner 1985], 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 [Steele 1984] 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 [Cartwright and Felleisen 1989]. 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 [Hennessy 1981]

> "...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 [Backus 1978]). 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 $\mathtt{f}$ and $\mathtt{g}$, 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 [Backus et al. 1986], but it should be applicable in other source-to-source program transformation systems. In fact, the only requirement for using these techniques in other contexts is that a suitable definition of the function `Safe` can be found for the programming language under consideration. 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 trade-off 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 5 discusses some pragmatic concerns, while Section 6 reports on experience with the use of these techniques in an optimizing compiler for FL. Section 7 concludes with a discussion of related work and suggestions for further work.

2. AN OVERVIEW OF FL

FL [Backus et al. 1986] is the result of an effort to design a practical functional language based on FP [Backus 1978]. FL has many of the features expected in modern functional languages, including higher-order functions, recursive definitions, user-defined functions, and datatypes. FL also differs from other functional languages in important ways. In particular, FL is combinator-based, has no static type system, and has first-class exceptions. Only the last feature is important to the results of this paper.

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, function definitions, input/output functions, syntactic sugar are omitted. The evaluation order of FL expressions is leftmost-innermost; thus, in $[\mathtt{f}:\mathtt{x}, \mathtt{g}:\mathtt{x}]$, first $\mathtt{f}:\mathtt{x}$ is evaluated and then $\mathtt{g}{:}\mathtt{x}$ is evaluated.

In Figure 1, 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., $\mathtt{s1}{:}0 \equiv \mathtt{s1}_{\mathtt{err}}$). FL functions produce more informative errors than just the name of the function, but this countable set of errors is sufficient for the purposes of this paper. In fact, the methods presented in subsequent sections are actually independent of the descriptiveness of the error values and work even with, e.g., error values that encode

$$
\begin{array}{rcl}
\texttt{f:x} & & \text{denotes function application} \\
\langle \texttt{x}_1, \ldots, \texttt{x}_n \rangle & & \text{denotes sequence construction, } \texttt{n} \geq 0 \\
(\texttt{f} \circ \texttt{g})\texttt{:x} & = & \texttt{f:(g:x)} \\
[\texttt{f}_1, \ldots, \texttt{f}_n]\texttt{:x} & = & \langle \texttt{f}_1\texttt{:x}, \ldots, \texttt{f}_n\texttt{:x} \rangle \\
(\texttt{p} \rightarrow \texttt{q};\texttt{r})\texttt{:x} & = & \begin{cases} \texttt{p:x} \text{ if } \texttt{p:x} \in \mathcal{E}_{\text{FL}} \\ \texttt{r:x} \text{ if } \texttt{p:x} = \underline{\texttt{false}} \\ \texttt{q:x} \text{ otherwise} \end{cases} \\
\tilde{}\texttt{x:y} & = & \texttt{x} \\
\alpha\texttt{:f:}\langle \texttt{x}_1, \ldots, \texttt{x}_n \rangle & = & \langle \texttt{f:x}_1, \ldots, \texttt{f:x}_n \rangle \\
+\texttt{:}\langle \texttt{x}_1, \ldots, \texttt{x}_n \rangle & = & \texttt{x}_1 + \ldots + \texttt{x}_n \\
\texttt{si:}\langle \texttt{x}_1, \ldots, \texttt{x}_n \rangle & = & \texttt{x}_i \\
\texttt{tl:}\langle \texttt{x}_1, \texttt{x}_2, \ldots, \texttt{x}_n \rangle & = & \langle \texttt{x}_2, \ldots, \texttt{x}_n \rangle \\
\texttt{rev:}\langle \texttt{x}_1, \ldots, \texttt{x}_n \rangle & = & \langle \texttt{x}_n, \ldots, \texttt{x}_1 \rangle \\
\texttt{al:}\langle \texttt{x}, \langle \texttt{y}_1, \ldots, \texttt{y}_n \rangle \rangle & = & \langle \texttt{x}, \texttt{y}_1, \ldots, \texttt{y}_n \rangle \\
\texttt{distl:}\langle \texttt{x}, \langle \texttt{y}_1, \ldots, \texttt{y}_n \rangle \rangle & = & \langle \langle \texttt{x}, \texttt{y}_1 \rangle, \ldots, \langle \texttt{x}, \texttt{y}_n \rangle \rangle \\
\texttt{id:x} & = & \texttt{x} \\
\texttt{dom:f} & = & \texttt{s1} \circ [\texttt{id}, \texttt{f}] \\
\texttt{catch:}\langle \texttt{f}, \texttt{g} \rangle\texttt{:x} & = & \begin{cases} \texttt{g:}\langle \texttt{x}, \texttt{y} \rangle & \text{if } \texttt{f:x} = \texttt{y}_{\text{err}} \\ \texttt{f:x} & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 1. A subset of FL.

the position in the program where the error arose.

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 [Harper et al. 1989], 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, so that $\texttt{f:x}_{\texttt{err}} \equiv \texttt{x}_{\texttt{err}}$ for any function $\texttt{f}$ and error value $\texttt{x}_{\texttt{err}}$. Sequence construction is also strict with respect to errors; a sequence collapses to the leftmost error it contains. This behavior is justified by 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:

$$
\begin{array}{rcl}
\mathcal{D}_{\text{FL}} & = & \mathcal{D}^+_{\text{FL}} \cup \mathcal{E}_{\text{FL}} \\
\mathcal{D}^+_{\text{FL}} & = & A \cup Seqs\,(\mathcal{D}^+_{\text{FL}}) \cup (\mathcal{D}^+_{\text{FL}} \rightarrow \mathcal{D}_{\text{FL}}) \\
 & & \textit{(the ordinary values)} \\
\mathcal{E}_{\text{FL}} & = & Err(\mathcal{D}^+_{\text{FL}}) \cup \{\perp\} \\
 & & \textit{(the error values)}
\end{array}
$$

In these equations, A is the set of atoms, *Seqs* is sequence construction, and *Err* is error construction. The ordering on $\mathcal{D}^+_{\text{FL}}$ is the standard one; in $\mathcal{E}_{\text{FL}}$, $x_{\text{err}} \leq y_{\text{err}} \Leftrightarrow x \leq y$ and $\perp \leq x$ for all x. Note that because other kinds of errors are given distinct values, the value $\perp$ is used only to denote non-terminating computations.

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 $\mathtt{f}$ and $\mathtt{g}$ denote the same semantic value.

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

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 $\mathtt{f}$ and $\mathtt{g}$; if $\mathtt{f}$ produces $\perp$ and $\mathtt{g}$ produces $\mathtt{tl}_{\mathtt{err}}$, then $[\mathtt{g}, \mathtt{f}]$ produces $\mathtt{tl}_{\mathtt{err}}$, but $\mathtt{rev} \circ [\mathtt{f}, \mathtt{g}]$ produces $\perp$. 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 $\perp$ is sufficient to invalidate any "law" that does not preserve the order of evaluation of its constituent functions.

3.1 An Informal Treatment of Safe

Although (6) is not always true, it is the case 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 $\mathtt{f}$ is annotated as being "safe" (written $\mathtt{S(f)}$), 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 $\mathtt{f}$ is rewritten to $\mathtt{g}$. (N.B. For now, $\mathtt{S(f)}$ is an extra-linguistic notion; the phrase "annotating $\mathtt{f}$ with $\mathtt{S(f)}$ " has no more semantic content than the phrase "painting $\mathtt{f}$ green".)

Consider the program $\texttt{rev} \circ [\texttt{\~{}0}, \texttt{\~{}1}]$. 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:

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

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:

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

Intuitively, each of the above steps preserved the meaning of the program, so this appears to be a proper 'optimization'. Indeed, in this case the final program is equivalent to the original program; i.e., $\texttt{rev} \circ [\texttt{\~{}0}, \texttt{\~{}1}] \equiv [\texttt{\~{}1}, \texttt{\~{}0}]$.

So far, however, the safe mechanism is informal, and it is easy to make mistakes. Consider the program $\texttt{rev} \circ [\texttt{distl}, \texttt{tl} \circ \texttt{al}]$. 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.

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

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

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

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:

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

Now, however, something has gone wrong: $\texttt{rev} \circ [\texttt{distl}, \texttt{tl} \circ \texttt{al}] \not\equiv [\texttt{s2}, \texttt{distl}]$, because $\texttt{rev} \circ [\texttt{distl}, \texttt{tl} \circ \texttt{al}]\texttt{:3}$ produces $\texttt{distl}_{\texttt{err}}$, whereas $\texttt{rev} \circ [\texttt{s2}, \texttt{distl}]\texttt{:3}$ 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.

3.2 The Formal Development of Safe

A first step towards formalizing the rewrite rules is to express qualified laws such as (7) equationally. Recall the intuition that a function `f` is safe if `f` cannot produce errors in the context in which it appears. In other words, the fact that `f:x` is an error is irrelevant if the context of `f` guarantees that `f` cannot be applied to `x`. One way to make this intuition explicit is to map all error values to a single "don't care" value. In the following definition of Safe, $\bot$ is chosen as the "don't care" value in addition to its role representing non-termination.[1]

DEFINITION 3.1. For every FL function $\mathtt{f}$, $\mathtt{Safe{:}f}$ denotes the function:

$$\mathtt{Safe{:}f{:}x} = \begin{cases} \mathtt{x} & \text{if } \mathtt{x} \in \mathcal{E}_{\mathrm{FL}} \\ \bot & \text{if } \mathtt{x} \notin \mathcal{E}_{\mathrm{FL}} \text{ and } \mathtt{f{:}x} \in \mathcal{E}_{\mathrm{FL}} \\ \mathtt{f{:}x} & \text{if } \mathtt{x} \notin \mathcal{E}_{\mathrm{FL}} \text{ and } \mathtt{f{:}x} \notin \mathcal{E}_{\mathrm{FL}} \end{cases}$$

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

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

Moreover, many other useful laws are expressible:

$$\begin{aligned}
\mathtt{f} &\equiv \mathtt{S{:}f} \circ \mathtt{dom{:}f} && (9)\\
\mathtt{f} \circ \mathtt{S{:}(dom{:}f)} &\equiv \mathtt{S{:}f} && (10)\\
\mathtt{S{:}f} \circ \mathtt{S{:}g} &\equiv \mathtt{S{:}(f} \circ \mathtt{g)} && (11)\\
\mathtt{S{:}p} \rightarrow \mathtt{S{:}q}; \mathtt{S{:}r} &\equiv \mathtt{S{:}(p} \rightarrow \mathtt{q}; \mathtt{r)} && (12)\\
\mathtt{[f,g]} &\equiv \mathtt{[f,g} \circ \mathtt{S{:}(dom{:}f)]} && (13)\\
\mathtt{[S{:}f_1,\ldots,S{:}f_n]} &\equiv \mathtt{S{:}[f_1,\ldots,f_n]} && (14)\\
\mathtt{rev} \circ \mathtt{S{:}[f_1,\ldots,f_n]} &\equiv \mathtt{S{:}[f_n,\ldots,f_1]} && (15)\\
\mathtt{S{:}(s1} \circ \mathtt{tl)} &\equiv \mathtt{S{:}s2} && (16)\\
\mathtt{catch{:}\langle S{:}f,g\rangle} &\equiv \mathtt{S{:}f} && (17)\\
\mathtt{catch{:}\langle f,g\rangle} \circ \mathtt{S{:}(dom{:}h)} &\equiv \mathtt{catch{:}\langle f} \circ \mathtt{S{:}(dom{:}h),g\rangle} && (18)
\end{aligned}$$

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 $\mathtt{tl} \circ \mathtt{S(al)} \mapsto \mathtt{S(s2)}$ cannot be written as $\mathtt{tl} \circ \mathtt{S{:}al} \equiv \mathtt{s2}$, since $(\mathtt{tl} \circ \mathtt{S{:}al}){:}\langle 1,2,3\rangle \equiv \bot$ whereas $\mathtt{s2}{:}\langle 1,2,3\rangle \equiv 2$; nor could it be $\mathtt{tl} \circ \mathtt{S{:}al} \equiv \mathtt{S{:}s2}$, for the same reason.

The difficulty is that $\equiv$ is a symmetric relation, whereas the desired property is inherently asymmetric: $\mathtt{tl} \circ \mathtt{S{:}al}$ can be rewritten to $\mathtt{S{:}s2}$, because in any program in which $\mathtt{tl} \circ \mathtt{S{:}al}$ could appear, the function $\mathtt{S{:}s2}$ produces the same result. Thus, at least some of the desired rewrite rules $\mathtt{f} \mapsto \mathtt{g}$ are valid only when $\mathtt{f}$ appears in a "good" context; i.e., in one which enforces the condition that $\mathtt{S{:}f}$ cannot produce $\bot$. The following definitions develop the relation $\trianglerighteq$ (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:
- $\mathtt{si}$, $\mathtt{tl}$, $\mathtt{rev}$, $\mathtt{al}$, $\mathtt{distl}$, $\mathtt{id}$, and $\mathtt{\tilde{}a}$ for all atoms $\mathtt{a}$ are simple expressions.
- If $\mathtt{e_1},\ldots,\mathtt{e_n}$ are simple expressions, then $\mathtt{e_1} \circ \mathtt{e_2}$, $\mathtt{[e_1,\ldots,e_n]}$, $(\mathtt{e_1} \rightarrow \mathtt{e_2}; \mathtt{e_3})$, $\mathtt{dom{:}e_1}$, $\mathtt{catch{:}\langle e_1,e_2\rangle}$, $\mathtt{Safe{:}e_1}$, and $\alpha{:}\mathtt{e_1}$ are simple expressions.

[1] The term "don't care" is meant to suggest it doesn't matter what this value is, because the context guarantees that it cannot arise. However, the choice of $\bot$ is not arbitrary—see Section 3.3.

(2) $\mathtt{E}$ is a *simple context* iff $\mathtt{E(f)}$ is a simple expression for every simple expression $\mathtt{f}$.

The simple expressions are first-order functions that can be built from the subset of FL given in Figure 1. Because the simple expressions are first-order, a great deal can be shown about their termination behavior; it is this property that will be exploited below.

It can be assumed that all expressions written by users have no occurrences of `Safe`, because only the language processor introduces and manipulates `Safe` expressions. Because all transformations preserve the meaning of expressions, the transformations are required to be correct only for expressions that also can be written without any occurrences of `Safe`.

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

DEFINITION 3.4. Let $\mathtt{f}, \mathtt{g}$ be simple expressions. $\mathtt{f} \rhd \mathtt{g}$ iff for every simple context $\mathtt{E}$ such that $\mathtt{E(f)}$ 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.

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

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 $\mathtt{f}$ should rewrite to $\mathtt{g}$ if $\mathtt{f}$ and $\mathtt{g}$ agree wherever $\mathtt{f}$ does not return the "don't care" value.

DEFINITION 3.5. $\mathtt{f} \leq_{\mathtt{S}} \mathtt{g}$ iff $\mathtt{f{:}x} \equiv \mathtt{g{:}x}$ whenever $\mathtt{f{:}x} \not\equiv \perp$.

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 $\mathtt{E}$ is a simple context and $\mathtt{f} \leq_{\mathtt{S}} \mathtt{g}$, then $\mathtt{E(f)} \leq_{\mathtt{S}} \mathtt{E(g)}$.

PROOF. First note that if $\mathtt{E(f)}$ does not contain $\mathtt{f}$, then $\mathtt{E(f)} = \mathtt{E(g)}$. The rest of the proof is by induction on the structure of $\mathtt{E}$. For the base case, if $\mathtt{E(f)} = \mathtt{f}$ and $\mathtt{E(g)} = \mathtt{g}$ then the result is immediate since $\mathtt{f} \leq_{\mathtt{S}} \mathtt{g}$. In other base cases $\mathtt{E}$ is a single primitive function not containing $\mathtt{f}$ (Definition 3.2). The inductive step is proven only for function composition; the other cases are similar. Let $\mathtt{E(f)} = \mathtt{E_1(f)} \circ \mathtt{E_2(f)}$ and assume that $\mathtt{E_1(f){:}(E_2(f){:}x)} \not\equiv \perp$. Then $\mathtt{E_2(f){:}x} \not\equiv \perp$ so $\mathtt{E_2(f){:}x} \equiv \mathtt{E_2(g){:}x}$ by induction, and therefore $\mathtt{E_1(f){:}(E_2(f){:}x)} \equiv \mathtt{E_1(g){:}(E_2(g){:}x)}$, also by induction. □

Note that Lemma 3.6 is false in general for non-simple contexts. For example, let f and g be functions such that $\mathtt{f} \leq_{\mathtt{S}} \mathtt{g}$ and $\mathtt{f} \neq \mathtt{g}$. Then $\tilde{}\mathtt{f} \leq_{\mathtt{S}} \tilde{}\mathtt{g}$ does not hold.

LEMMA 3.7. If e is a user-definable simple expression, then $\mathtt{e}\colon\mathtt{x} \not\equiv \perp$ whenever $\mathtt{x} \not\equiv \perp$.

PROOF. Let $\mathtt{e} \equiv \mathtt{u}$ where u contains no occurrence of Safe. The proof is a simple induction on the structure of u using part (1) of Definition 3.2. □

Lemma 3.7 provides the reason for working with simple expressions. Informally, user-definable simple expressions do not return $\perp$ unless applied to $\perp$. Now, program transformation may introduce a function $\mathtt{S}:\mathtt{f}$ into a user-definable simple expression $\mathtt{E}(\mathtt{S}:\mathtt{f})$, and it may be the case that $(\mathtt{S}:\mathtt{f}):\mathtt{x} = \perp$. Because program transformation must preserve the meaning of the simple expression and because expressions are strict in $\perp$, we know that the function S:f can never be applied to x in the context $\mathtt{E}(\mathtt{S}:\mathtt{f})$. In short, a use of Safe:f within a user-definable simple expression indicates that f can never produce an exception in that context. This intuitive discussion is formalized by the Substitutability Theorem, which reduces the problem of verifying that $\mathtt{f} \rhd \mathtt{g}$ to the easier problem of checking that $\mathtt{f} \leq_{\mathtt{S}} \mathtt{g}$ in the case that f appears in a simple context.

THEOREM 3.8. If f and g are simple expressions and $\mathtt{f} \leq_{\mathtt{S}} \mathtt{g}$, then $\mathtt{f} \rhd \mathtt{g}$.

PROOF. Let E be a simple context and assume that $\mathtt{E}(\mathtt{f})$ is user-definable. By Lemma 3.6, $\mathtt{E}(\mathtt{f}) \leq_{\mathtt{S}} \mathtt{E}(\mathtt{g})$. By Lemma 3.7, $\mathtt{E}(\mathtt{f})\colon\mathtt{x} \not\equiv \perp$ if $\mathtt{x} \not\equiv \perp$. Together these facts imply that $\mathtt{E}(\mathtt{f})\colon\mathtt{x} \equiv \mathtt{E}(\mathtt{g})\colon\mathtt{x}$ if $\mathtt{x} \not\equiv \perp$. By strictness, $\mathtt{E}(\mathtt{f})\colon\perp \equiv \perp \equiv \mathtt{E}(\mathtt{g})\colon\perp$. Therefore, $\mathtt{E}(\mathtt{f}) \equiv \mathtt{E}(\mathtt{g})$. □

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)-(5) and (8)-(17), apply to *all* expressions, not merely the simple expressions.

3.3 Beyond Simple Expressions

At this point, it is natural to ask why the set of expressions to which the Substitutability Theorem can be applied must be restricted at all. We conjecture that no restriction is needed but, in fact, achieving an extension that works for all expressions appears to be a difficult technical challenge. This section explains the difficulty.

For the theory presented in Section 3.2 to work, it is necessary to distinguish between $\perp$'s role as the "don't care" value and $\perp$'s role representing non-termination. This is accomplished by restricting attention to the simple expressions, which are guaranteed to terminate. Thus, the straightforward approach to generalizing the Safe mechanism to all expressions is to use something other than $\perp$ as the value to which Safe maps all errors. Even better, this value should be a new element of the domain, not available to programmers. It would then be easy to distinguish "don't care" values from values that a programmer can define.

Let † be a new value and let Safe be redefined as follows:

DEFINITION 3.9. For every FL function $\mathtt{f}$, $\mathtt{Safe}\colon\mathtt{f}$ denotes the function:

$$\mathtt{Safe}\colon\mathtt{f}:\mathtt{x} = \begin{cases} \mathtt{x} & \text{if } \mathtt{x} \in \mathcal{E}_{\mathrm{FL}} \\ \dagger & \text{if } \mathtt{x} \notin \mathcal{E}_{\mathrm{FL}} \text{ and } \mathtt{f}:\mathtt{x} \in \mathcal{E}_{\mathrm{FL}} \\ \mathtt{f}:\mathtt{x} & \text{if } \mathtt{x} \notin \mathcal{E}_{\mathrm{FL}} \text{ and } \mathtt{f}:\mathtt{x} \notin \mathcal{E}_{\mathrm{FL}} \end{cases}$$

Clearly, $\dagger$ must be included somewhere in the domain ordering. Herein lies the problem—it appears that the only possibility is that $\dagger = \bot$! The reasoning goes as follows:

The proof of Lemma 3.6 and Law 10 (among others) depend on the fact that $\dagger$, once produced, cannot be "changed" by the surrounding context. Thus, all expressions must be strict in $\dagger$.

Let $\mathtt{f}$ be any function and $\mathtt{x}$ any normal value such that $\mathtt{f}:\mathtt{x}$ is an error value. Then $(\mathtt{S}:\mathtt{f}):\mathtt{x} = \dagger$ by Definition 3.9. Now, $(\mathtt{S}:\mathtt{f}):\bot = \bot$ by strictness, which is also taken into account in Definition 3.9. Clearly, $\bot \leq \mathtt{x}$, so by monotonicity of application $\bot \leq \dagger$.

Let $\mathtt{K(y)}$ be the FL function such that $\mathtt{K(y)}:\mathtt{x} = \mathtt{y}$ for all normal values $\mathtt{x}$. Let $\mathtt{x}$ be any normal value. Then $\dagger \leq \mathtt{x}$ since $\mathtt{K}(\dagger) = \mathtt{S}:\mathtt{K}(\bot) \leq \mathtt{S}:\mathtt{K}(\mathtt{x}) = \mathtt{K}(\mathtt{x})$.

Finally, note that for any normal value $\mathtt{x}$, we have $\mathtt{K}(\bot):\mathtt{x} = \bot$ and $\mathtt{K}(\bot):\dagger = \dagger$ by strictness of $\dagger$. Since $\dagger \leq \mathtt{x}$, it follows by monotonicity that $\dagger \leq \bot$. But $\bot \leq \dagger$ as well. Therefore $\bot = \dagger$.

The argument presented above is not a complete proof that a distinct $\dagger$ cannot be included in the domain, because it rests on several assumptions that could be modified. Unfortunately, there is very little room for modifications that might provide a solution within the framework of standard domain theory. For example, we have used the fact that FL functions are strict in $\bot$. However, the argument doesn't use the fact that all functions are strict, just that there is at least one strict function in the domain. Thus, the problem is not strictness vs. laziness—the argument applies to non-strict languages as well.

After considerable experimentation, we have not discovered any way to extend the Safe mechanism to all user-definable expressions. We conjecture that it cannot be done within standard domain theory; some quite different construction appears to be required. It is surprising (at least to us) that the simple concept of a result that "can't happen" during evaluation is difficult to formalize.

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 $\mathtt{dom}:\mathtt{f}:\mathtt{x}$ is $\mathtt{f}:\mathtt{x}$ if $\mathtt{f}:\mathtt{x}$ is an error value and $\mathtt{x}$ otherwise (see Figure 1). The examples use the following laws involving `dom`:

$$\mathtt{dom}:\mathtt{id} \equiv \mathtt{id} \tag{25}$$

$$\mathtt{dom}:[\mathtt{f}_1,\ldots,\mathtt{f}_n] \equiv \mathtt{dom}:\mathtt{f}_n \circ \ldots \circ \mathtt{dom}:\mathtt{f}_1 \tag{26}$$

$$\mathtt{dom}:\mathtt{al} \circ \mathtt{dom}:\mathtt{distl} \equiv \mathtt{dom}:\mathtt{distl} \tag{27}$$

$$\mathtt{S}:(\mathtt{dom}:\mathtt{tl}) \equiv \mathtt{S}:(\mathtt{dom}:\mathtt{s1}) \tag{28}$$

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

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

This transformation is an optimization, because the end result eliminates the application of $\texttt{rev}$. Note, however, that the intermediate steps are not necessarily optimizations, because they involve computing some values twice; in particular, $\texttt{f}$ could be arbitrarily expensive to compute. The law $\texttt{f} \equiv \mathsf{S}\colon\texttt{f} \circ \texttt{dom}\colon\texttt{f}$ is very useful for introducing safe functions, but if the added $\texttt{dom}$ 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 $\texttt{f}$ once very slowly to preserve errors and then once again to produce the result! Section 6 shows how this is avoided in practice in the optimizing compiler for FL.

The second example illustrates that there are additional advantages to marking functions safe.

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

In this case, the end result is an optimization not only because the application of $\texttt{rev}$ is eliminated, but also because $\texttt{dom}\colon\texttt{distl}$ permits the rest of the program to be executed without checking the arguments of any of the functions. The use of $\texttt{Safe}$ 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 $\texttt{al}$ and $\texttt{distl}$ can run unchecked.

As noted above, it is in general undesirable to use Law 9 because it duplicates computation. In this particular example, it is worth factoring $\texttt{dom} : \texttt{distl}$ out of the computation, because $\texttt{dom}\colon\texttt{distl}$ can be replaced by a function that merely checks whether the argument is a pair of which the second component is a sequence. This single check is more efficient than the two checks performed by the primitive functions $\texttt{distl}$ and $\texttt{al}$ in the original expression.

The third example presents a more substantial optimization (similar to loop-jamming optimizations for imperative languages [Aho et al. 1986]) and illustrates the use of in-context laws:

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

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 $\mathtt{catch}\!: \langle \mathtt{tl}, [] \rangle$. If `newtl` appears in a context where it always gets an argument in the proper domain of `tl`, `newtl` can be transformed to $\mathsf{S}\!: \mathtt{tl}$ as follows:

$$\begin{array}{lll}
& \mathtt{newtl} \circ \mathsf{S}\!: (\mathtt{dom}\!: \mathtt{tl}) & \\
\equiv & \mathtt{catch}\!: \langle \mathtt{tl}, [] \rangle \circ \mathsf{S}\!: (\mathtt{dom}\!: \mathtt{tl}) & \text{by def. of } \mathtt{newtl} \\
\equiv & \mathtt{catch}\!: \langle \mathtt{tl} \circ \mathsf{S}\!: (\mathtt{dom}\!: \mathtt{tl}), [] \rangle & \text{by } 18 \\
\equiv & \mathtt{catch}\!: \langle \mathsf{S}\!: \mathtt{tl}, [] \rangle & \text{by } 10 \\
\equiv & \mathsf{S}\!: \mathtt{tl} & \text{by } 17
\end{array}$$

5. ON SAFE AND DOM

This paper takes an algebraic approach to program optimization, expressing source-to-source program transformations as algebraic laws. There are practical limitations to this approach. This section discusses some of these limitations, particularly the difficulties that arise from the relationship between `Safe` and `dom`.

Because the function `Safe` is not available to programmers, a mechanism is required for introducing `Safe` annotations into a program. Law 9 is a general rule that introduces `Safe`. Recall that the law is:

$$\mathtt{f} \equiv \mathsf{S}\!: \mathtt{f} \circ \mathtt{dom}\!: \mathtt{f}$$

Law 9 expresses a fundamental principle of the algebra of `Safe` and, in principle, it can be used to introduce `Safe` wherever needed. From a pragmatic point of view, however, using Law 9 is a disaster. To understand why, consider how a program optimization system based on source-to-source program transformation works. Very briefly, the compiler repeatedly selects program transformations to apply to the program. Ideally, the compiler is able to judge the change in program cost (say, execution time) that results from a transformation, and the compiler selects transformations that decrease program cost. This high-level description ignores several practical problems, of which one of the most significant is that it is hard to estimate the true value of transformation at the time it is applied. In particular, it is very difficult to include in the estimate of the value of a particular transformation T the value of additional transformations that T enables.

Law 9 aggravates this problem, since it duplicates the computation of an arbitrarily expensive function. Thus, application of Law 9 is purely speculative—at the time it is applied, it is not known whether the program is ultimately improved. In our opinion, it is desirable to guarantee that an optimizing compiler does not, at the very least, make programs worse. A simple way to achieve this guarantee is to ensure that each individual transformation proceeds "downhill" in the direction of improvement. Introducing arbitrary "uphill" steps in the transformation process removes this guarantee both in theory and, very often in our experience, in practice.

Section 6 presents a program analysis that conservatively infers where `Safe` can be inserted in a program without also introducing `dom`. Thus, this analysis avoids the "uphill" step of applying Law 9.

There is another aspect to the relationship between `Safe` and `dom`. By introducing uses of `dom`, it is possible to replace almost all in-context laws by equations. For example, Law 19 can be expressed equationally as:

$$\texttt{tl} \circ \texttt{S:al} \equiv \texttt{s2} \circ \texttt{dom:(S:al)}$$

While this equation has the advantage of avoiding the additional concept of in-context transformations, it has the disadvantage that it is not clearly an optimization, whereas Law 19 clearly is. However, Law 19 can be recovered by further transformations:

$$\begin{array}{lll} & \texttt{s2} \circ \texttt{dom:(S:al)} & \\ \triangleright & \texttt{s2} \circ \texttt{id} & \text{by (21)} \\ \equiv & \texttt{s2} & \text{by (1)} \end{array}$$

Note that this reasoning requires the use of an in-context law to remove the application of `dom`. Thus, it is not possible to eliminate in-context transformations, although in principle their use could be restricted to applications of Law 21. Formulating the laws in this way only makes sequences of transformations longer without reducing the number of rules; for this reason, we prefer the more direct versions of in-context laws given in Section 3.

There are instances where the introduction of `dom` is useful. As shown above, a function of the form `dom:(S:f)` or (equivalently) `S:(dom:f)` has little or no cost, since it always can be eliminated in favor of `id`. It is sometimes useful to introduce such a function to transfer context information between portions of a program (e.g., Law 13). Finally, there are even limited instances where an unsafe `dom` is introduced, which may or may not be eliminated by subsequent program transformation; an example is given on page 11 in Section 4. However, unlike that example, such introductions are tightly controlled by our compiler and limited to special cases where the transformation can still be shown to be "downhill."

6. INTRODUCING SAFE

This section describes a program analysis method that conservatively identifies functions that can be annotated with `Safe`. That is, the analysis proves theorems of the form `E(f)` $\equiv$ `E(S:f)`. The program analysis is a general type inference system, but the details of type inference are not important to understand how `Safe` is introduced. This section states properties satisfied by the type inference system and shows how these properties are sufficient to introduce `Safe`. The interested reader is referred to [Aiken and Wimmers 1993; Aiken et al. 1994] for more information on the type inference algorithm.

Types are certain subsets of the domain $\mathcal{D}_{\mathrm{FL}}$. The full definition of type requires more development [MacQueen et al. 1984], but the only property required here is that a type is a set of values. A *typed function* has the form $\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}$ where $\mathtt{A}$ and $\mathtt{B}$ are types and $\mathtt{A} \subseteq \mathcal{D}^{+}_{\mathrm{FL}}$ and $\mathtt{B} \subseteq \mathcal{D}_{\mathrm{FL}}$. The notation $\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}$ is read " $\mathtt{f}$ has type $\mathtt{A} \rightarrow \mathtt{B}$. " A formal semantics for the phrase $\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}$ is given below. For the moment, the intuition is that $\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}$ is an assertion saying two things:

(1) For any $\mathtt{x} \in \mathtt{A}$, it is the case that $\mathtt{f}{:}\mathtt{x} \in \mathtt{B}$.

(2) The function $\mathtt{f}$ appears in a context where the only normal values it is applied only to are in $\mathtt{A}$.

For example, consider the following:

$$\begin{aligned} \mathtt{NonEmptySeq} &= \{\langle x_1, \ldots, x_n\rangle | n \geq 1,\ x_i \in \mathcal{D}^{+}_{\mathrm{FL}}\} \\ \mathtt{NotNonEmptySeq} &= \mathcal{D}^{+}_{\mathrm{FL}} - \mathtt{NonEmptySeq} \end{aligned}$$

Then $\mathtt{s1} :: \mathtt{NonEmptySeq} \rightarrow \mathcal{D}^{+}_{\mathrm{FL}}$ is interpreted as saying that `s1` appears in a context where it is guaranteed to be applied to a non-empty sequence and (therefore) to produce a non-error value. Similarly, $\mathtt{s1} :: \mathtt{NotNonEmptySeq} \rightarrow \mathcal{E}_{\mathrm{FL}}$ is interpreted as saying that `s1` appears in a context where it is guaranteed to be applied to something that is not a non-empty sequence and (therefore) is guaranteed to produce an error. Finally, $\mathtt{s1} :: \mathcal{D}^{+}_{\mathrm{FL}} \rightarrow \mathcal{D}_{\mathrm{FL}}$ says that nothing is known about the context in which `s1` appears.

The informal part of the description of $\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}$ above is the idea that `f` appears in a context where it is applied to elements of `A`. Projections are a well-known technique for making precise the idea of a "context" [Wadler and Hughes 1987].

Definition 6.1. A *projection* is an idempotent function less than the identity.

The function $\Delta{:}\mathtt{A}$ is a projection for any type `A`:

$$\Delta{:}\mathtt{A}{:}\mathtt{x} = \begin{cases} \mathtt{x} & \text{if } \mathtt{x} \in \mathtt{A} \\ \bot & \text{otherwise} \end{cases}$$

Because $\Delta{:}\mathtt{A}{:}\mathtt{x}$ is either $\mathtt{x}$ or $\bot$ for all $\mathtt{x}$, it is easy to see that $\Delta{:}\mathtt{A} \leq \mathtt{id}$ and that $\Delta{:}\mathtt{A}$ is idempotent. Therefore, $\Delta{:}\mathtt{A}$ is in fact a projection.[2] The formal meaning of $\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}$ is given in terms of projections.

Definition 6.2. The expression $\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}$ is the function $\mathtt{Propagate}{:}(\Delta{:}\mathtt{B} \circ \mathtt{f} \circ \Delta{:}\mathtt{A})$, where $\mathtt{Propagate}{:}\mathtt{f}{:}\mathtt{x_{err}} = \mathtt{x_{err}}$ and $\mathtt{Propagate}{:}\mathtt{f}{:}\mathtt{x} = \mathtt{f}{:}\mathtt{x}$ otherwise.

[2] A technical note: The function $\Delta{:}\mathtt{A}$ is not necessarily continuous if `A` is an arbitrary set of values. For any type `A`, however, $\Delta{:}\mathtt{A}$ is continuous. Note also that we abuse notation by allowing `A` to contain errors even though $\Delta{:}\mathtt{A}$ is not strict in the error values of `A`.

This definition formalizes the intuition given above by defining $\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}$ to be a function that can produce a non-bottom result only when it is applied to some $\mathtt{x} \in \mathtt{A}$ such that $\mathtt{f}{:}\mathtt{x} \in \mathtt{B}$. The functional `Propagate` simply ensures that if the argument to the typed function is an exception that exception is propagated in keeping with the normal semantics of FL. A *typed expression* is an FL expression that may include typed functions as subexpressions. The following definition explains what it means for the types on subexpressions to be correct.

DEFINITION 6.3. Let $\mathtt{E}$ be a typed expression and let $\mathtt{E}'$ be $\mathtt{E}$ with all types erased (i.e., $\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}$ in $\mathtt{E}$ is replaced by $\mathtt{f}$ in $\mathtt{E}'$). Then $\mathtt{E}$ is *well-typed* iff $\mathtt{E} \equiv \mathtt{E}'$.

The FL type inference system takes an FL expression and produces a well-typed expression. For example, consider the program $\mathtt{s1} \circ [\tilde{}\mathtt{1}, \tilde{}\mathtt{2}]$. The inference system produces the well-typed program

$$(\mathtt{s1} :: \mathtt{Pair(Int)} \rightarrow \mathtt{Int}) \circ [\tilde{}\mathtt{1} :: \mathcal{D}^{+}_{\mathrm{FL}} \rightarrow \mathtt{Int}, \tilde{}\mathtt{2} :: \mathcal{D}^{+}_{\mathrm{FL}} \rightarrow \mathtt{Int}].$$

The type `Int` is the set of integers and `Pair(X)` is all sequences of length 2 with elements drawn from `X`. The system also infers types for the higher-functions composition and sequence construction, but these types are omitted for readability. The following lemma states the main result of this section: if $\mathtt{E}(\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B})$ is well-typed and the type `B` contains no error values, then `f` can be replaced by $\mathtt{S}{:}\mathtt{f}$.

LEMMA 6.4. Let $\mathtt{E}(\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B})$ be well-typed and let $\mathtt{B} \cap \mathcal{E}_{\mathrm{FL}} \subseteq \{\perp\}$. Then $\mathtt{E}(\mathtt{f}) \equiv \mathtt{E}(\mathtt{S}{:}\mathtt{f})$.

PROOF. We prove that $\mathtt{S}:\mathtt{f} \geq \mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}$. The result then follows because $\mathtt{E}(\mathtt{f}) \geq \mathtt{E}(\mathtt{S}{:}\,\mathtt{f}) \geq \mathtt{E}(\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B})$ and $\mathtt{E}(\mathtt{f}) \equiv \mathtt{E}(\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B})$ since $\mathtt{E}(\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B})$ is well-typed.

If $(\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}){:}\,\mathtt{x} \equiv \perp$ there is nothing to prove. Assume $(\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}){:}\,\mathtt{x} \not\equiv \perp$. If $\mathtt{x} \in \mathcal{E}_{\mathrm{FL}}$ then $(\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}){:}\,\mathtt{x} \equiv \mathtt{x}$ and $\mathtt{S}{:}\,\mathtt{f}{:}\,\mathtt{x} \equiv \mathtt{x}$. Now assume that $\mathtt{x} \notin \mathcal{E}_{\mathrm{FL}}$. Then $(\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}){:}\,\mathtt{x} = (\Delta{:}\,\mathtt{B} \circ \mathtt{f} \circ \Delta{:}\,\mathtt{A}){:}\,\mathtt{x} \not\equiv \perp$. Therefore, $\Delta{:}\,\mathtt{A}{:}\,\mathtt{x} \equiv \mathtt{x}$ and $\mathtt{f}{:}\,\mathtt{x} \not\equiv \perp$ and $\Delta{:}\mathtt{B}{:}\,(\mathtt{f}{:}\mathtt{x}) \equiv \mathtt{f}{:}\mathtt{x}$. Since $\mathtt{B} \cap \mathcal{E}_{\mathrm{FL}} \subseteq \{\perp\}$ and $\mathtt{f}{:}\,\mathtt{x} \in \mathtt{B} - \{\perp\}$, it follows that $\mathtt{f}{:}\,\mathtt{x} \notin \mathcal{E}_{\mathrm{FL}}$. Therefore $\mathtt{S}{:}\,(\mathtt{f}{:}\,\mathtt{x}) = \mathtt{f}{:}\,\mathtt{x} = (\mathtt{f} :: \mathtt{A} \rightarrow \mathtt{B}){:}\,\mathtt{x}$. $\square$

Returning to the example above, Lemma 6.4 shows that

$$(\mathtt{s1} :: \mathtt{Pair(Int)} \rightarrow \mathtt{Int}) \circ [\tilde{}\mathtt{1} :: \mathcal{D}^{+}_{\mathrm{FL}} \rightarrow \mathtt{Int}, \tilde{}\mathtt{2} :: \mathcal{D}^{+}_{\mathrm{FL}} \rightarrow \mathtt{Int}] \;\equiv\; \mathtt{S}{:}\,\mathtt{s1} \circ [\mathtt{S}{:}\,\tilde{}\mathtt{1}, \mathtt{S}{:}\,\tilde{}\mathtt{2}].$$

Type inference must be very accurate to be useful for proving that functions are safe. For example, in the example with `s1` above, it is necessary to prove not just that `s1` is applied to a sequence but that `s1` is applied to a non-empty sequence. The standard Hindley/Milner type system [Damas and Milner 1982] cannot prove such properties. A type inference system that can infer types accurate enough to be useful in program optimization has been implemented for FL and is described in [Aiken and Wimmers 1993; Aiken et al. 1994].

7. SAFE IN AN OPTIMIZING COMPILER FOR FL

The `Safe` technique is used heavily in an optimizing compiler for FL developed at IBM Almaden. The FL compiler makes use of literally hundreds of laws, most of which involve safety information. This section gives a brief qualitative summary of several years experience, both good and bad, with using `Safe` in the optimization of FL programs.

The overall structure of the FL compiler is simple. After parsing and desugaring, the type inference algorithm described in Section 6 annotates a program with types. Based on the types assigned, functions are then marked as `Safe` using Lemma 6.4. At this point, the system enters a "match-transform" loop in which subexpressions are matched against the left-hand sides of laws. When a match is found, the matched subexpression is replaced by the right-hand side of the law and the entire process repeats. The strategy used to search for matches is not fixed (there are several such strategies in the compiler) but normally the program is scanned repeatedly until no laws match. A few optimization phases that have well-defined algorithms are implemented directly outside of the match-transform paradigm.

The laws of the FL compiler are organized into groups according to purpose. Some laws are aimed at a particular kind of optimization (e.g., simplifying nested conditionals) or are used merely as an adjunct to compiler phases that have a large component implemented in a way other than via laws (e.g., compile-time evaluation). Although the focus of these phases is something other than safety information, almost all compiler phases make some use of safety information.

The primary pass of the compiler that relies on safety information is a set of approximately 200 laws that are best described as "peephole" optimizations. These laws all have the property that (1) they are improvements to the program ("downhill" transformations) and (2) they reduce the size of the program. The laws are applied repeatedly using a greedy strategy until no more laws apply. Most of the rules in this paper are included in this phase of the compiler. Property (1) guarantees that the result is an optimization of the original program; property (2) guarantees that eventually no more rules apply and the process terminates.

There is another interesting property, namely the Church-Rosser property, that these laws do not enjoy. In our experience, this property is neither crucial nor really practical to obtain for large collections of program transformations. There are several reasons for this conclusion. First, experience suggests it is better to include an optimization rather than exclude it on the grounds that it makes a set of laws non-Church-Rosser. After all, the empty set of laws is Church-Rosser. Second, Church-Rosser does not imply termination of a set of laws and in fact these two properties are sometimes in conflict (e.g., adding a law that makes the system Church-Rosser may destroy termination). We regard the paramount concerns to be optimization and termination (properties (1) and (2) above). Finally, it is an enormous task to prove Church-Rosser properties for large sets of transformations. Even if there were some chance that the laws were Church-Rosser (they are not) it would be prohibitively difficult to prove this fact.

The overall experience with `Safe` has been very good. It is easy to write and add very general laws to the compiler. Without `Safe` many of these laws could not be expressed at all or could be expressed only as an enormous number of special cases. The denotational basis for `Safe` has also proven to be crucial. The example in Section 3.1 of how informal reasoning leads to incorrect transformations is not contrived; on several occasions similar incorrect transformations were added by the authors. Having a denotational basis for reasoning about `Safe` proved indispensable in these cases for identifying the incorrect transformations from among the hundreds of laws in the compiler.

On an engineering level, Safe has also been successful. A key concern in the design of the compiler was the speed of the match-transform step. Because the FL compiler relies heavily on matching and transforming expressions, it is very important that matching common constructs be fast. The function Safe was engineered to be fast to match; it is implemented simply as a one-bit flag attached to functions and adds a negligible overhead to the matching process.

Using Safe does have two drawbacks, however, both of which are related to the use of in-context transformations. The first problem is that when writing an in-context law it is easy to write a law that is less accurate than desired. To see this, recall that $\mathtt{S}{:}\mathtt{f} \rhd \mathtt{f}$ (Law 20), so it is always permissible to drop Safe from a function. Thus, for example, $\mathtt{S}{:}(\mathtt{s1} \circ \mathtt{tl}) \equiv \mathtt{S}{:}\mathtt{s2}$ also can be written $\mathtt{S}{:}(\mathtt{s1} \circ \mathtt{tl}) \rhd \mathtt{s2}$. The disadvantage of the second form is that safety information is lost, so that additional laws are less likely to match, which degrades the quality of program optimization. To see how easy it is to write laws with "safety leaks," note that in Law 19 the s2 on the right-hand side is in fact Safe. Even the authors failed to notice this through many revisions of this paper.

Safety leaks are hard to detect. The only hint of a problem is that a program that should optimize well does not, and lost safety information is not the only potential cause. Losing safety information turned out to be a real but not insurmountable problem in the FL compiler, although considerable time was spent inspecting for and removing safety leaks from laws.

The second problem with in-context laws is that they are not invertible. This is, of course, inherent. The whole point of a law $\mathtt{f} \rhd \mathtt{g}$ is that g can replace f but not vice-versa. In-context laws are one-way tickets: if the compiler makes a poor choice in applying an in-context law, there is no way to undo it. This should not be taken as an argument against in-context laws. Such laws are very useful and even necessary for good optimization. The problem is that it may be difficult to decide when to apply in-context laws because other (perhaps better) avenues for optimization may be lost.

The FL experience is that, while this latter problem does occur, the practical effect is small. There are programs for which some optimization is missed because an in-context law is applied that cannot be undone in a later phase. However, these examples are relatively few and the cost in lost optimization is usually small. It was judged not to be worthwhile to make any special effort to selectively apply in-context laws. The FL compiler currently makes no distinction between laws that are equivalences and laws that apply only in-context.

8. RELATED WORK, CONCLUSIONS, AND FUTURE WORK

The first mention of qualified or "in context" laws for FP appeared in Backus' original paper [Backus 1978]. Backus proposed *qualified equational laws*, which consist of an equation between functions $\mathtt{f} \equiv \mathtt{g}$ and a predicate p. The meaning of qualified equational law is that $\mathtt{f}{:}\mathtt{x} = \mathtt{g}{:}\mathtt{x}$ whenever $\mathtt{p}{:}\mathtt{x}$ is true. In the context of this paper, the combinator Safe replaces the predicate so that equations can be expressed without side conditions.

Safety analysis is related to *projection analysis* [Wadler and Hughes 1987]. The idea of projection analysis is to express formally properties of programs using projections (see Section 7). The following lemma shows that projections can be used to characterize safety analysis:

LEMMA 8.1. $\texttt{S:f} \equiv \texttt{f} \circ \texttt{p}$ for some projection $\texttt{p}$.

PROOF. The projection $\texttt{p}$ is $\texttt{dom:(S:f)}$. Since $\texttt{dom:(S:f):x} \equiv \texttt{x}$ or $\texttt{dom:(S:f):x} \equiv \perp$ for all $\texttt{x}$, it follows that $\texttt{dom:(S:f)} \leq \texttt{id}$ and that $\texttt{dom:(S:f)}$ is idempotent. Hence, $\texttt{dom:(S:f)}$ is in fact a projection.

It is easy to check that $\texttt{dom:(S:f)} \equiv \texttt{dom:f} \circ \texttt{dom:(S:f)}$. Then,

$$
\begin{array}{lll}
 & \texttt{S:f} & \\
\equiv & \texttt{S:(S:f)} \circ \texttt{dom:(S:f)} & \text{by Law 9} \\
\equiv & \texttt{S:f} \circ \texttt{dom:(S:f)} & \texttt{S} \text{ is idempotent} \\
\equiv & \texttt{S:f} \circ \texttt{dom:f} \circ \texttt{dom:(S:f)} & \\
\equiv & \texttt{f} \circ \texttt{dom:(S:f)} & \text{by Law 9}
\end{array}
$$

□

Beyond the connection with the theory of projections, there are other similarities in the use of the two techniques. 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 program transformations. For example, to the best of our knowledge the counterpart of in-context laws for projection analysis has not been developed, although it certainly could be using the techniques presented here.

There is another, perhaps deeper, difference between safety analysis and projection analysis. In the basic formulation of projection analysis, a lifted domain is used with a new element † below $\perp$ to which projections map elements outside the domain of interest [Wadler and Hughes 1987]. As discussed in Section 3.3, it appears that no standard domain with `Safe` allows † to be separated from $\perp$. The crux of the difference between the two techniques is that projection analysis is concerned with program analysis only and thus can avoid giving semantics to programs that contain projections. However, `Safe` is intended for use in a program transformation system where programs contain occurrences of `Safe`, and thus it is crucial that programs with `Safe` be given semantics directly.

It would strengthen the theoretical treatment if the restriction to simple expressions of in-context laws could be removed. As discussed in Section 3.3, such a generalization appears to present difficult semantic problems. On a practical level, most of the common uses of `Safe` are covered even with the restriction to simple expressions. However, a model that worked for all expressions would help clarify the semantic role of `Safe` and add some power to the algebra. We leave as future work whether techniques that extend projection analysis to higher-order programs (e.g., [Hunt and Sands 1991]) can be used to generalize safety analysis.

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.

ACKNOWLEDGMENTS

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. Finally, thanks go to the anonymous referees for many useful suggestions.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

AIKEN, A., WILLIAMS, J. H., AND WIMMERS, E. L. 1990. Program transformation in the presence of errors. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pp. 210–217.

AIKEN, A. AND WIMMERS, E. 1993. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, pp. 31–41.

AIKEN, A., WIMMERS, E., AND LAKSHMAN, T. 1994. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, pp. 163–173.

BACKUS, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21,* 8 (Aug.), 613–641.

BACKUS, J., WILLIAMS, J. H., AND WIMMERS, E. L. 1986. The FL language manual. Tech. Rep. RJ 5339 (54809), IBM.

CARTWRIGHT, R. AND FELLEISEN, M. 1989. The semantics of program dependence. In *Proceedings of the 1989 Conference on Programming Language Design and Implementation.*

DAMAS, L. AND MILNER, R. 1982. Principle type-schemes for functional programs. In *Ninth Annual ACM Symposium on Principles of Programming Languages*, pp. 207–212.

HARPER, R., MILNER, R., AND TOFTE, M. 1989. The definition of standard ML—version 3. Tech. Rep. ECFS-LFCS-89-81, Laboratory for Foundations of Computer Science, University of Edinburgh.

HENNESSY, J. 1981. Program optimization and exception handling. In *Proceedings of the 1981 Symposium on Principles of Programming Languages.*

HUDAK, P., WADLER, P., ARVIND, BOUTEL, B., FAIRBAIRN, J., FASEL, J., HUGHES, J., JOHNSSON, T., KIEBURTZ, D., JONES, S. P., NIKHIL, R., REEVE, M., WISE, D., AND YOUNG, J. 1988. Report on the functional programming language Haskell. Tech. Rep. DCS/RR-666 (Dec.), Yale University.

HUNT, S. AND SANDS, D. 1991. Binding time analysis: A new PERspective. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 154–165.

MacQueen, D., Plotkin, G., and Sethi, R. 1984. An ideal model for recursive polymophic types. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 165–174.

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

Turner, D. A. 1985. 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.

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