WPI-CS-TR-99-39 December 1999

S-Expressiveness and the Abstractive Power of Programming Languages

by

John N. Shutt

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

S-Expressiveness and the Abstractive Power of Programming Languages

John N. Shutt
jshutt@cs.wpi.edu
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609

December 1999

Abstract

This paper investigates possible approaches to developing a mathematical model of the process of *abstraction* in programming. In particular, two formal measures are considered for comparing the relative abstractive power of programming languages. A measure called *expressiveness*, proposed elsewhere by M. Felleisen, is found insufficient for the current task because it is concerned with expression only of runtime semantics. A second measure called *S-expressiveness*, defined here, doesn't fully capture abstractive power either, but its shortcomings are different from those of Felleisen's measure. Implications for future research are discussed.

Contents

List of Definitions

List of Theorems

1 Introduction

This paper is the second step in a program to develop a mathematical model of the process of abstraction in programming. An informal model of the mechanical process of abstraction was developed in [Shut99b]. Building on that work, this paper investigates what kind of mathematical model could capture the qualitative phenomenon.

The approach taken is experimental. Two different formal measures are considered for comparing the relative abstractive power of programming languages (that is, for comparing how effectively they support the expression of abstractions); and the merits and demerits of both measures are analyzed.

The first measure considered is *T-expressiveness*, proposed (under the name *macro expressiveness*) by M. Felleisen in [Fell91]. Felleisen's model is intended to capture the relative facility with which programming languages express computations (as opposed to mere *ability* as captured by the criterion of Turing-equivalence). However, abstractive features — facility, or even ability, to express abstractions — are often invisible to Felleisen's measure, because they have no computational consequences of the kind that his measure can distinguish. §2 defines Felleisen's measure, and demonstrates its failure to encompass abstractive power.

A formal measure called *S-expressiveness* is developed in §3. S-expressiveness doesn't fully capture abstractive power either; but it does detect some features that Felleisen's measure overlooks.

Finally, §4 recaps the strengths and weaknesses of the two measures, and suggests how an improved measure of abstractive power might be developed.

The reader is assumed to have passing familiarity with the rudiments of many-sorted algebra.

2 T-Expressiveness

Felleisen called his formal criterion *macro expressiveness* [Fell91]. It is referred to here as *T-expressiveness* (T being mnemonic for *Term-based*, because Felleisen models a program as a term over an algebra.)

2.1 The criterion

Definition T.1 (Programming language) A *programming language* $\mathcal{L}$ consists of the following parts.

- A signature $\Sigma_{\mathcal{L}}$. The domain of all $\Sigma_{\mathcal{L}}$-terms is $T_{\Sigma_{\mathcal{L}}}$. To simplify our explanation, we assume that the signature is one-sorted, and that each operator in it has a unique arity.
- A (usually infinite) set $P_{\mathcal{L}} \subseteq T_{\Sigma_{\mathcal{L}}}$ of $\mathcal{L}$-programs.

- A recursively enumerable predicate $eval_{\mathcal{L}}: P_{\mathcal{L}} \to \mathbb{B}$ called the *semantics* of $\mathcal{L}$. Program $p \in P_{\mathcal{L}}$ *terminates* iff $eval_{\mathcal{L}}(p)$.

□

By "recursively enumerable predicate", presumably Felleisen means that the set of programs $\{p \in P_{\mathcal{L}} \mid eval_{\mathcal{L}}(p)\}$ may be any recursively enumerable subset of $P_{\mathcal{L}}$.

Definition T.2 (Conservative extension) A programming language $\mathcal{L}$ is a *conservative extension* of a programming language $\mathcal{L}'$, and $\mathcal{L}'$ is a *conservative restriction* of $\mathcal{L}$, iff

- $\Sigma_{\mathcal{L}'} \subseteq \Sigma_{\mathcal{L}}$;
- $P_{\mathcal{L}'} = P_{\mathcal{L}} \cap T_{\Sigma_{\mathcal{L}'}}$; and
- for all $p \in P_{\mathcal{L}'}$, $eval_{\mathcal{L}'}(p)$ iff $eval_{\mathcal{L}}(p)$.

If $S = \Sigma_{\mathcal{L}} - \Sigma_{\mathcal{L}'}$ is the difference between the signatures, the conservative restriction may be denoted $\mathcal{L}' = \mathcal{L} \backslash S$, and the conservative extension, $\mathcal{L} = \mathcal{L}' + S$. □

The following definition of T-expressiveness involves the notion of *polynomials* over a signature. Felleisen actually calls them *syntactic abstractions*, and remarks that they are also referred to variously in the literature as *polynomials*, *notational abbreviations*, *macros*, and *derived operators*. In the interest of completeness, here is a formal definition.

Definition T.3 (Polynomial) Suppose Σ is a signature. A *variable set* over Σ is an ordered set of symbols V, disjoint from Σ and having a least element.

Suppose $V = \{v_1, v_2, \cdots\}$ is a variable set over Σ. A *polynomial* over Σ in variables V is a term over the following signature $\Sigma(V)$.

$$\Sigma(V)_j \quad = \quad \begin{cases} \Sigma_j \cup V & \text{if } j = 0 \\ \Sigma_j & \text{otherwise} \end{cases}$$

The *arity* of a polynomial $\pi \in T_{\Sigma(V)}$ is the smallest integer $ar(\pi) \in \mathbb{N}$ such that for all v_k in π, $k \leq ar(\pi)$. Note, in particular, that if $\pi \in T_\Sigma$ then $ar(\pi) = 0$.

Suppose $\pi \in T_{\Sigma(V)}$ is a polynomial, $ar(\pi) \leq n$, and $t_1, \cdots t_n \in T_\Sigma$ are terms over Σ. Then $\pi(t_1, \cdots, t_n)$ denotes a term over Σ, as follows.

- If $\pi = v_k$, then $\pi(t_1, \cdots t_n) = t_k$.
- If $\pi = \sigma(\pi_1, \cdots, \pi_m)$, then $\pi(t_1, \cdots, t_n) = \sigma(\pi_1(t_1, \cdots, t_n), \cdots, \pi_m(t_1, \cdots, t_n))$.

□

Definition T.4 (T-Expressibility) Suppose $\mathcal{L}$ is a programming language, $S \subseteq \Sigma_{\mathcal{L}}$, and $\mathcal{L}' = \mathcal{L} \backslash S$. Then $\mathcal{L}'$ can *T-express* S with respect to $\mathcal{L}$ iff there is a recursive mapping $\varphi: T_{\Sigma_{\mathcal{L}}} \to T_{\Sigma_{\mathcal{L}'}}$ with the following properties **E1**–**E4**.

E1 $e \in P_{\mathcal{L}}$ implies $\varphi(e) \in P_{\mathcal{L}'}$;

E2 φ is homomorphic in all operators of $\Sigma_{\mathcal{L}'}$;

E3 For all $e \in P_{\mathcal{L}}$, $eval_{\mathcal{L}}(e)$ iff $eval_{\mathcal{L}'}(\varphi(e))$; and

E4 φ is polynomial in every operator in S. That is, for each operator $s \in S$ of arity a there exists a polynomial π_s of arity a over $\Sigma_{\mathcal{L}'}$ such that

$$\varphi(s(e_1, \ldots, e_a)) \quad = \quad \pi_s(\varphi(e_1), \ldots, \varphi(e_a))$$

□

Definition T.5 (Weak T-Expressibility) Suppose $\mathcal{L}' = \mathcal{L} \backslash S$. Then $\mathcal{L}'$ can *weakly T-express* S with respect to $\mathcal{L}$ iff there is a recursive mapping $\varphi \colon T_{\Sigma_{\mathcal{L}}} \to T_{\Sigma_{\mathcal{L}'}}$ with properties **E1**, **E2**, and **E4** from Definition T.4, and the following property **E3′**.

E3′ For all $e \in P_{\mathcal{L}}$, $eval_{\mathcal{L}}(e)$ implies $eval_{\mathcal{L}'}(\varphi(e))$.

□

2.2 Properties

Example T.6 As a simple illustration of how these definitions can be applied, consider a variant on the ordinary lambda calculus. (For a vastly more extensive treatment of this example, see [Fell91].) Instead of the usual λ operator, let us have *two* function constructors, λ_n and λ_v. The λ_n operator takes its parameters by name — which is the way the ordinary lambda calculus works — while λ_v takes its parameters by value. The complete variant lambda calculus with both operators will be called Λ; with only call-by-name, $\Lambda_n = \Lambda \backslash \lambda_v$; and with only call-by-value, $\Lambda_v = \Lambda \backslash \lambda_n$.

Theorem T.6 (Call-by-name and call-by-value)

- Λ_v cannot T-express λ_n with respect to Λ.
- Λ_n cannot T-express λ_v with respect to Λ.
- Λ_n can **weakly** T-express λ_v with respect to Λ.

◇

Without getting into gory detail (see, again, [Fell91]), note that λ_v always waits until its argument is completely evaluated, whereas λ_n may sometimes converge when evaluation of its argument would diverge. Neither operator can precisely simulate the behavior of the other. However, when λ_v does converge it always yields the same result as λ_n, so replacing λ_n with λ_v satisfies criterion **E3′** for weak T-expressiveness.

These results might at first appear to contradict the folk theorem that "call-by-value is more powerful than call-by-name". On closer examination, however, the folk theorem has its origin in the problem of exchanging the contents of variables

$$
\begin{array}{rcl}
\langle \mathit{program} \rangle & \rightarrow & \langle \mathit{modules} \rangle \\
\langle \mathit{modules} \rangle & \rightarrow & \langle \mathit{module} \rangle \mid \langle \mathit{module} \rangle ; \langle \mathit{modules} \rangle \\
\langle \mathit{module} \rangle & \rightarrow & \textbf{module } \langle \mathit{identifier} \rangle ;\ \langle \mathit{declarations} \rangle ;\ \textbf{end} \\
\langle \mathit{declarations} \rangle & \rightarrow & \langle \mathit{private} \rangle \mid \langle \mathit{declaration} \rangle \mid \langle \mathit{declarations} \rangle ; \langle \mathit{declarations} \rangle \\
\langle \mathit{private} \rangle & \rightarrow & \textbf{private } \langle \mathit{declaration} \rangle \\
\langle \mathit{declaration} \rangle & \rightarrow & \langle \mathit{module} \rangle \mid \langle \mathit{other} \rangle \\
\langle \mathit{other} \rangle & \rightarrow & \langle \mathit{identifier} \rangle = \langle \mathit{atom} \rangle \\
\langle \mathit{atom} \rangle & \rightarrow & \langle \mathit{literal} \rangle \mid \langle \mathit{identifier} \rangle
\end{array}
$$

Figure T.7: Syntax of a toy language

in ALGOL 60; and that is a uniquely imperative problem, whereas Theorem T.6 concerns a strictly declarative language. (Lambda calculus, no less.) □

Felleisen's approach to expressiveness is based on the familiar notion of *syntactic sugar* [Land66] — a language feature whose removal from the language would require only "syntactically local" transformation of any given program. Felleisen's definition of programming language (Definition T.1) is ideally suited to this approach.

- "Language features" are the operators of a many-sorted signature, providing a conveniently finite description of language syntax, and compatibility with an extensive literature that uses many-sorted algebra to describe abstract data types.
- A "program" is a syntax tree, providing a convenient characterization of polynomial substitution (Definition T.3).

Because he proceeds from the notion of syntactic sugar, he can directly compare two languages only if one is a sublanguage of the other; arbitrary languages are not commensurate. When two languages are fairly similar, this limitation can often be worked around by relating them to a larger language containing all the features of both, as in Theorem T.6.

Felleisen's work elegantly captures the relative capacity of programming language features to express *computational* capabilities. However, it fundamentally fails to capture capacity to express *abstractional* capabilities, as in the following example.

Example T.7 An important part of abstraction support in most modern programming languages is *modular encapsulation*. Let $\mathcal{L}$ be a toy programming language whose concrete syntax is given by Figure T.7. (The syntax productions for $\langle \mathit{identifier} \rangle$ and $\langle \mathit{literal} \rangle$ are omitted, as they are not important for this example.) Let the visibility rules of $\mathcal{L}$ be as follows.

- Within a module, each object is visible anywhere to the right of its declaration.
- Outside a module, if the module is visible then its public (i.e., non-private) objects are visible.
- An identifier on the right side of a declaration must name an object that is visible at that point in the program.
- An identifier on the *left* side of a declaration must *not* name any object visible at that point in the program; thus there is no question of object name shadowing.

We will make the pragmatically obvious (but mathematically important) assumption that deleting a **private** keyword from a valid program doesn't change its semantics.

Consider the language $\mathcal{L}' = \mathcal{L}\backslash private$. In $\mathcal{L}'$, every object of every module is public. From the perspective of abstraction support, something important has been lost. Computationally, however, encapsulation is pure syntactic sugar.

Theorem T.7 (T-expressiveness does not capture encapsulation) $\mathcal{L}'$ can T-express *private* with respect to $\mathcal{L}$. $\diamond$

Proof. Let $\varphi : T_{\Sigma_{\mathcal{L}}} \to T_{\Sigma_{\mathcal{L}'}}$ be the transformation that simply makes everything public by stripping off all the **private** keywords. That is, φ is homomorphic on all operators of $\Sigma_{\mathcal{L}'}$, and $\varphi(private\ v) = v$. By construction, φ satisfies **E2** and **E4**. Making everything public won't invalidate any valid program, so φ satisfies **E1**; and by assumption it won't change the semantics of any program either, so φ satisfies **E3**. By Definition T.4, $\mathcal{L}'$ can T-express *private* with respect to $\mathcal{L}$. $\diamond$

Taken as complete $\mathcal{L}$-programs, p and $\varphi(p)$ can only be distinguished by their semantics; i.e., by their runtime behavior. However, a careful examination of Figure T.7 will show that every $\mathcal{L}$-program p is also a proper prefix of infinitely many other $\mathcal{L}$-programs. This suggests a way to distinguishing p from $\varphi(p)$: Let $\mathit{suffix}(p) = \{\omega \mid p\,\omega \in P_{\mathcal{L}}\}$ be the set of possible suffixes of p. Then for all $p \in P_{\mathcal{L}}$, $\mathit{suffix}(p) \subseteq \mathit{suffix}(\varphi(p))$; moreover, $\mathit{suffix}(p) = \mathit{suffix}(\varphi(p))$ if and only if $p \in P_{\mathcal{L}'}$. This observation forms the basis for the alternative expressiveness criterion considered in §3. $\square$

3 S-expressiveness

T-expressiveness (§2) addresses only the conventionally semantic (i.e., runtime) consequences of a program. Naturally, it is unable to distinguish programs that have the same runtime semantics. From the observations at the end of Example T.7 it is evident that, in order to capture abstraction, a formal criterion must also address the *syntactic* consequences of a program fragment.

This section describes a criterion called *S-expressiveness*, which attempts to capture abstraction through a purely syntactic analysis of the consequences of program texts. Program fragments are treated as arbitrary symbol strings with no context-free structure. (The letter S is mnemonic for *Set-based*, in difference to Felleisen's *Term-based* approach.)

3.1 The criterion

S-expressiveness is essentially a containment relation, similar in the broad organization of its definitions to Felleisen's criterion.

Definition S.1 (Language) An *alphabet* is a finite set of atomic symbols. Suppose Σ is an alphabet. The set of strings over Σ is Σ^*, the empty string is $\lambda \in \Sigma^*$, the set of nonempty strings over Σ is $\Sigma^+ = \Sigma^* - \{\lambda\}$. A *language* over Σ is any subset of Σ^*. $\square$

If L is a language over Σ and $\Sigma \subseteq S$, then L is a language over S.

Definition S.2 (Semantics) Suppose L is a language over Σ, and $x \in \Sigma^*$. The *semantics of x in L*, denoted $[\![x]\!]_L$, is the set of all strings y such that $xy \in L$. That is,

$$[\![x]\!]_L \quad = \quad \{y \mid xy \in L\}$$

$\square$

If L is a language over Σ, and $x \in \Sigma^*$, then $\lambda \in [\![x]\!]_L$ iff $x \in L$. Also, $[\![\lambda]\!]_L = L$; but there may also exist $x \neq \lambda$ with $[\![x]\!]_L = L$.

Definition S.3 (S-expressibility) Suppose L and L' are languages over Σ. L' can *S-express* L iff there exists $x \in \Sigma^*$ such that $[\![x]\!]_{L'} = L$. $\square$

Trivially from the definition, S-expressibility is transitive; that is, if L'' can S-express L' and L' can S-express L, then L'' can S-express L.

3.2 Properties

Recall the languages $\mathcal{L}$ and $\mathcal{L}' = \mathcal{L}\backslash private$ from Example T.7. It was shown that $\mathcal{L}'$ can T-express *private* with respect to $\mathcal{L}$, even though $\mathcal{L}'$ clearly lacks the abstractive power of *private*. S-expressibility resolves this difficulty.

Theorem S.4 (S-expressibility captures encapsulation) Let $\mathcal{L}$ and $\mathcal{L}' = \mathcal{L}\backslash private$ be the languages defined in Example T.7. Then $\mathcal{L}'$ cannot S-express $\mathcal{L}$, and $\mathcal{L}$ cannot S-express $\mathcal{L}'$. $\square$

At first glance, this is not what we had in mind. The *private* operator would seem to make $\mathcal{L}$ abstractionally more powerful than $\mathcal{L}'$; but under S-expressibility, the two languages are simply incomparable.

There is a way around this problem. The reason that $\mathcal{L}$ cannot S-express $\mathcal{L}'$ is that *private*, though used in $\mathcal{L}$ for abstraction, is not itself within the purview of the abstraction facilities of $\mathcal{L}$. That is, there is no way of "turning off" *private*; so for all valid program prefixes x of $\mathcal{L}$, there exist $y \in [\![x]\!]_{\mathcal{L}}$ that use *private*, and thus $y \notin \mathcal{L}'$. So suppose we define a third language, L'', in which the syntax production for $\langle program \rangle$ in Figure T.7 is replaced with

$$\langle \textit{program} \rangle \quad \rightarrow \quad \langle \textit{modules} \rangle \mid \textbf{disable private;}\ \langle \textit{modules} \rangle$$

with the 'visibility' rule that if a program begins with "**disable private;**", then keyword **private** cannot occur at all in the rest of the program. In $\mathcal{L}''$, *private* can itself be hidden. It follows immediately that

$$[\![\lambda]\!]_{\mathcal{L}'} \quad = \quad [\![\textbf{disable private;}]\!]_{\mathcal{L}''}$$

and thus, $\mathcal{L}''$ can S-express $\mathcal{L}'$.

Of course, this example is trivial — it's like having a switch at the top of a program that indicates whether the ensuing code will be in C or Pascal — but it does suggest an interesting general principle. Apparently, S-expressiveness isn't applicable to abstractive operators unless those operators are also *subject* to abstraction. The reader may ponder whether or not this is a reasonable property for a measure of abstractive power.

Unfortunately, S-expressiveness itself is not a reasonable measure of abstractive power, as will now be demonstrated.

Theorem S.5 (Universal S-expressing language) Suppose Σ is an alphabet. Then there exists a recursively enumerable language L that can S-express every recursively enumerable language over Σ. □

Proof. Let Y be the alphabet $\{0, 1, \$\}$; it doesn't matter whether or not Y and Σ are disjoint. Let L be the set of all strings $n\$w$ such that n is the binary representation of the Gödel number of a Turing machine M (under a suitable Gödelization fixed for given Σ) and w is accepted by M. Then L is recursively enumerable.

Suppose A is a recursively enumerable language over Σ. Let M be a Turing machine that accepts exactly A, and let n be the binary representation of the Gödel number of M. Then $[\![\lambda]\!]_A = [\![n\$]\!]_L$, so L can S-express A. □

In itself, Theorem S.5 seems like a favorable development; the existence of universal languages under various criteria is always of interest. However, the proof of the theorem illustrates a serious drawback of S-expressibility as a measure of abstraction support. Language L in the above proof is a universally S-expressive language;

but its use of Gödel numbering is a singularly opaque way to represent programs, so that while L provides *ability* to express arbitrary abstractions, it does not do so with *facility*.

4 Conclusion

T-expressiveness exploited the internal (context-free) structure of programs in order to capture facility of expression, in addition to mere ability. However, it addressed only the expression of computation, i.e., runtime semantics, and so was inapplicable to the expression of abstraction, which is basically a syntactic phenomenon.

S-expressiveness was based on a purely syntactic treatment of programs; even the runtime semantics, if needed, would be modeled using syntax (as for example in the proof of Theorem S.5). Because it took into account the syntactic as well as traditionally semantic consequences of program code, it was able to address the expression of abstractive features that had eluded T-expressiveness. However, because it ignored program structure, it was only able to capture ability to express abstractions; facility of expression was beyond it.

Based on these observations, it appears likely that a more effective criterion for comparing *facility* of expression of abstractions might combine the purely syntactic approach of S-expressiveness, and the explicit use of program structure as in T-expressiveness.

The RAG formalism [Shut99a] is well suited for the task. It affords a finite, purely syntactic description of arbitrary Turing-powerful computations, making essential use of explicit hierarchical ("context-free") phrase structure (as opposed to cosmetic use as in traditional attribute grammars).

The probable next step in the development of a mathematical model of abstraction, therefore, will be to formulate a fresh criterion for abstractive power based on RAGs.

References

[Fell91] Matthias Felleisen, "On the Expressive Power of Programming Languages", *Science of Computer Programming* 17 nos. 1–3 (December 1991) [Selected Papers of ESOP '90, the 3rd European Symposium on Programming], pp. 35–75.

Available on the Web. See URL
`"http://www.cs.rice.edu/CS/PLT/Publications/"`.

An earlier version of the paper appeared in Neil D. Jones, editor, *ESOP '90: 3rd European Symposium on Programming* [Copenhagen, Denmark, May 15–18, 1990, Proceedings] [*Lecture Notes in Computer Science* 432], New York: Springer-Verlag, 1990, pp. 134–151.

[Land66] P. J. Landin, "The Next 700 Programming Languages", *Communications of the ACM* 9 no. 3 (March 1966) [*Proceedings of the ACM Programming Languages and Pragmatics Conference*, San Dimas, California, August 8–12, 1965], pp. 157–166.

[Shut99a] John N. Shutt, "Recursive Adaptable Grammars", technical report WPI-CS-TR-99-03, Worcester Polytechnic Institute, Worcester, MA, January 1999.

[Shut99b] John N. Shutt, "Abstraction in Programming — working definition", technical report WPI-CS-TR-99-38, Worcester Polytechnic Institute, Worcester, MA, December 1999.