WPI-CS-TR-08-01

March 2008
emended, 26 March 2008

Abstractive Power of Programming Languages: Formal Definition

by

John N. Shutt

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

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

Abstractive Power of Programming Languages: Formal Definition

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

March 2008
emended 26 March 2008

Abstract

Abstraction in programming uses the facilities of a given programming language to customize the abstract machine of the language, effectively constructing a new programming language, so that each program may be expressed in a language natural to its intended problem domain. Abstraction is a major element of software engineering strategy. This paper suggests a formal notion of abstraction, on the basis of which the relative power of support for abstraction can be investigated objectively. The theory is applied to a suite of trivial toy languages, confirming that the suggested theory orders them correctly.

Contents

List of definitions

List of theorems

1 Introduction

Abstraction in programming uses the facilities of a given programming language to customize the abstract machine of the language, effectively constructing a new programming language ([Sh99]). It has been advocated since the 1970s (e.g., [DaHo72]) as a way to allow each program to be expressed in a way natural to its intended problem domain. Various language features have been touted as "powerful abstractions"; in this paper, we suggest a theory by which to objectify such claims.

To validate the theory, we apply it to a suite of trivial toy languages, whose relative ordering by abstractive power is intuitively clear. The theory correctly orders this suite. We also discuss why several related formal devices for comparing programming languages, proposed over the past two decades ([Fe91, Mi93, KrFe98]), are not suitable for our purpose.

Section 2 develops the formal notion of *programming language* on which the remainder of the work is founded. Section 3 describes the test suite of toy languages. Sections 4–5 develop the formal notion of *abstractive power*, and show how it handles the example from §3. Section 6 compares and contrasts the formal notion of abstractive power here with some related formal devices in the literature. Section 7 discusses possible future applications of the theory.

2 Programming languages

To see what we will need from a formal definition of *programming language*, consider an arbitrary abstraction. A starting language $\mathcal{L}$ is altered by means of a source text a, resulting in an incrementally different language $\mathcal{L}'$. Diagrammatically,

$$\mathcal{L} \xrightarrow{\quad a \quad} \mathcal{L}' . \tag{1}$$

Our basic understanding of abstraction in programming (§1) says that a is source text *in language* $\mathcal{L}$. Moreover, we can only speak of the abstractive power "of $\mathcal{L}$" if $\mathcal{L}$ regulates the entire transaction: what texts a are permissible, and for each a, what altered language $\mathcal{L}'$ will result. Each possible $\mathcal{L}'$ then determines what further abstraction-inducing texts are possible, and what languages result from them; and so on.

We can also identify two expected characteristics of the text a.

First, a is not an arbitrary prefix of a valid program: it is understood as a coherent, self-contained account of an alteration to $\mathcal{L}$. By allowing $\mathcal{L}'$ to be reached via a single abstractive step, we imply that $\mathcal{L}$ and $\mathcal{L}'$ are at the same level of syntactic structure, and invite our abstractive-power comparison device, whatever form it might take, to assess their relationship on that basis. So a is a natural unit of *modularity* in the language, and the formal presentation of modularity will heavily influence any judgment of abstractive power.

To illustrate the point, consider a simple Java class definition,

```
public class Pair {
  public Object car;
  public Object cdr;
  }.
```
(2)

This is readily understood to specify a change to the entire Java language, consisting of the addition of a certain class ***Pair***; but a fragmentary prefix of it, say

```
public class Pair {
  public Object car;,
```
(3)

isn't naturally thought of as a modification to Java as a whole: what must follow isn't what we would have expected if we were told only that it would be code written in Java; rather, it must be a completion of the fragment, *after* which we will be back to something very close to standard Java (with an added class ***Pair***). It would be plausible to think of the single line

```
  public Object car;
```
(4)

as modifying the local language of declarations within the body of the class declaration; but that language is related to Java in roughly the same way as two different nonterminals of a CFG when one nonterminal occurs as a descendant of the other in syntax trees (as, in a CFG for English, ⟨noun-phrase⟩ might occur as a descendant of ⟨sentence⟩). For the current work —having no prior reason to suppose that a more elaborate approach is required— we choose to consider abstractive power directly at just one level of syntactic structure at a time, disregarding abstractions whose effect is localized within a single text a.

We thus conceive of the abstraction process as traversing a directed graph whose vertices are languages and whose edges are labeled by source texts. This is also a sufficient conception of computation. The more conventional approach to computation by language, in which syntax is mapped to internal state and observables (e.g., [Mi93]), can be readily modeled by our directed graph, using the vertices for internal state, and a selected subset of the edge-labeling texts for observables. (This modeling technique will be demonstrated, in principle, in §3.)

A second expected characteristic of a is that it *has* internal syntactic structure, that impinges on our study of abstraction at a higher syntactic level (so that we still care about internal syntax, even though we've already decided to disregard internal abstraction). The purpose of abstraction is to affect *how* programs are expressed, not merely *whether* they can be expressed. In assessing the expressive significance of a programming language feature, the usual approach is to ask, if the feature were omitted from the language, first, *could* programs using the feature be rewritten to do without it, but then, assuming that they could be rewritten, how much would the rewriting necessarily disrupt the syntactic structure of those programs. Most basically, Landin's notion of syntactic sugar is a feature whose omission would require only local transformations to the context-free structure of programs. Landin's notion was directly formalized in [Fe91]; and structurally well-behaved transformations are also basic to the study of abstraction-preserving reductions in [Mi93]. For a general study of abstractive power, it will be convenient to consider various classes of structural transformations of programs (§4, below); but to define any class of structural transformations of text a, the structure of a must have been provided to begin with.

To capture these expected characteristics, in concept we want a modified form of state machine. The states of the machine are languages; the transition-labeling "symbols" are texts; and the behavior of the machine is anchored by a set of observable texts rather than a set of final states. However, if we set up our formal definition of *programming language* this way, it would eventually become evident, in our treatment of abstractive power, that the states themselves are an unwanted distraction. We will prefer to concern ourselves only with the set of text-sequences on paths from a state, and the set of observable texts. We therefore greatly streamline our formal definition by including only these elements, omitting the machinery of the states themselves.

Definition 2.1 Suppose T is the set of syntax trees freely generated over some context-free grammar.

A *programming language over texts* T (briefly, language over T) is a set $L \subseteq T^*$ such that for all $y \in L$ and x a prefix of y (that is, $\exists z$ with $y = xz$), $x \in L$. Elements of T are called *texts*; elements of L are called *text sequences* (briefly, sequences). The empty sequence is denoted ϵ.

Suppose A, B are sets of strings, and $f\colon A \to B$. f *respects prefixes* if, for all $x, y \in A$, if x is a prefix of y then $f(x)$ is a prefix of $f(y)$. ■

Intuitively, L is the set of possible sequences of texts from some machine state q; and if sequence y is possible from q, and x is a prefix of y, then x is also possible from q; hence the requirement that languages be closed under prefixing.

The set of texts T will usually be understood from context, and we will simply say "L is a language", etc. The set of texts will be explicitly mentioned primarily in certain definitions, to flag out concepts that are relative to T (as Definition 4.1).

When the CFG G that freely generates T is unambiguous —and this is the usual case— we may elide the distinction between a sentence s over G, and the unique syntax tree t freely generated over G whose fringe is s.

Example 2.2 Suppose T is freely generated over the context-free grammar with start symbol S, terminal alphabet $\{a, b\}$, and production rules $\{S \to a,\ S \to b\}$. Since the CFG is unambiguous, we treat sentences as if they were trees, writing $T = \{a, b\}$, $abaa \in T^*$, etc. Regular set $R_1 = a^*$ is a language, since it is closed under prefixes. Regular set $R_2 = (ab)^*$ is not a language, since $abab \in R_2$ but $aba \notin R_2$. The closure of R_2 under prefixing, $R_3 = (ab)^* \cup (ab)^*a$, is a language.

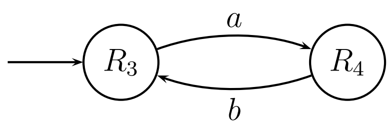

Figure 1: NFA for language $(ab)^* \cup (ab)^*a$.

Function $f\colon T^* \to T^*$ defined by $f(w) = aabbw$ respects prefixes, since for every $xy \in T^*$, $f(xy) = aabbxy = (f(x))y$.

Function $g\colon T^* \to T^*$ defined by $g(w) = ww$ does not respect prefixes, since $g(a) = aa$ is not a prefix of $g(ab) = abab$. ∎

Definition 2.3 Suppose S is a set of strings, and x is a string. Then S *reduced by* x is $S\langle\!\langle x\rangle\!\rangle = \{y \mid xy \in S\}$. (That is, $S\langle\!\langle x\rangle\!\rangle$ is the set of suffixes of x in S.)

Suppose A, B are sets of strings, $x \in A$, $f\colon A \to B$, and f respects prefixes. Then f *reduced by* x is $f\langle\!\langle x\rangle\!\rangle\colon A\langle\!\langle x\rangle\!\rangle \to B\langle\!\langle f(x)\rangle\!\rangle$ such that $\forall y \in A\langle\!\langle x\rangle\!\rangle,\ f(xy) = f(x)f\langle\!\langle x\rangle\!\rangle(y)$. (That is, $f\langle\!\langle x\rangle\!\rangle(y)$ is the suffix of $f(x)$ in $f(xy)$.) ∎

The property of respecting prefixes is exactly what is needed to guarantee the existence of $f\langle\!\langle x\rangle\!\rangle$ for all $x \in A$; it means that f corresponds to a conceptual homomorphism of state machines.[1]

Example 2.4 Recall, from Example 2.2, languages $R_1 = a^*$ and $R_3 = (ab)^* \cup (ab)^*a$, and prefix-respecting function $f(w) = aabbw$.

For every sequence $w \in R_1$, $R_1\langle\!\langle w\rangle\!\rangle = R_1$.

For every sequence $w \in R_3$, if w has even length then $R_3\langle\!\langle w\rangle\!\rangle = R_3$. Let $R_4 = (ba)^* \cup (ba)^*b$. If $w \in R_3$ has odd length, then $R_3\langle\!\langle w\rangle\!\rangle = R_4$. Similarly, if $w \in R_4$ has even length, then $R_4\langle\!\langle w\rangle\!\rangle = R_4$; while if $w \in R_4$ has odd length, then $R_4\langle\!\langle w\rangle\!\rangle = R_3$. In particular, $R_3\langle\!\langle a\rangle\!\rangle = R_4$ and $R_4\langle\!\langle b\rangle\!\rangle = R_3$; thus, R_3 and R_4 are effectively the two states of an NFA (nondeterministic finite automaton), as shown in Figure 1. The set of sentences accepted by the NFA is R_3; if state R_4 were the start state, the set of sentences accepted would be R_4.

For every $x \in T^*$, $f(xy) = (f(x))y$, therefore $f\langle\!\langle x\rangle\!\rangle(y) = y$. In particular, $f\langle\!\langle \epsilon\rangle\!\rangle(y) = y$ (highlighting that $f\langle\!\langle \epsilon\rangle\!\rangle \neq f$). ∎

A language A models the semantics of a sequence $x \in A$ as $A\langle\!\langle x\rangle\!\rangle$. Observables only come into play when one needs to articulate to what extent behavior is preserved by a function from one language to another.

[1] That is, f maps states q of machine A to states $f(q)$ of machine B, and transitions $\delta_A(q,t) = q'$ of A to extended transitions $\widehat{\delta}_B(f(q), f\langle\!\langle q\rangle\!\rangle(t)) = f(q')$ of B.

$$
\begin{array}{rcl}
x_1 & = & (\lambda\mathrm{x}.(\lambda\mathrm{y}.\mathrm{x}))\mathrm{uv} \\
x_2 & = & (\lambda\mathrm{x}.(\lambda\mathrm{y}.\mathrm{x}))\mathrm{uv} \rhd (\lambda\mathrm{y}.\mathrm{u})\mathrm{v} \\
x_3 & = & (\lambda\mathrm{x}.(\lambda\mathrm{y}.\mathrm{x}))\mathrm{uv} \rhd (\lambda\mathrm{y}.\mathrm{u})\mathrm{v} \rhd \mathrm{u} \\
x_4 & = & (\lambda\mathrm{x}.\mathrm{xx})(\lambda\mathrm{x}.\mathrm{xx}) \\
x_5 & = & (\lambda\mathrm{x}.\mathrm{xx})(\lambda\mathrm{x}.\mathrm{xx}) \rhd (\lambda\mathrm{x}.\mathrm{xx})(\lambda\mathrm{x}.\mathrm{xx}) \\
x_6 & = & (\lambda\mathrm{x}.\mathrm{xx})(\lambda\mathrm{x}.\mathrm{xx}) \rhd (\lambda\mathrm{x}.\mathrm{xx})(\lambda\mathrm{x}.\mathrm{xx}) \rhd (\lambda\mathrm{x}.\mathrm{xx})(\lambda\mathrm{x}.\mathrm{xx}) \\
x_7 & = & (\lambda\mathrm{x}.\mathrm{y})((\lambda\mathrm{x}.\mathrm{xx})(\lambda\mathrm{x}.\mathrm{xx})) \\
x_8 & = & (\lambda\mathrm{x}.\mathrm{y})((\lambda\mathrm{x}.\mathrm{xx})(\lambda\mathrm{x}.\mathrm{xx})) \rhd (\lambda\mathrm{x}.\mathrm{y})((\lambda\mathrm{x}.\mathrm{xx})(\lambda\mathrm{x}.\mathrm{xx})) \\
x_9 & = & (\lambda\mathrm{x}.\mathrm{y})((\lambda\mathrm{x}.\mathrm{xx})(\lambda\mathrm{x}.\mathrm{xx})) \rhd (\lambda\mathrm{x}.\mathrm{y})((\lambda\mathrm{x}.\mathrm{xx})(\lambda\mathrm{x}.\mathrm{xx})) \rhd \mathrm{y}
\end{array}
$$

Figure 2: Some sequences in $L_{\lambda n}$.

Definition 2.5 Suppose A, B are languages over T, $O \subseteq T$, and $f\colon A \to B$ respects prefixes.

f weakly respects observables O (briefly, weakly respects O) if

(a) for all $x \in A$, $y \in A\langle\!\langle x\rangle\!\rangle$, and $o \in O$, o is a prefix of y iff o is a prefix of $f\langle\!\langle x\rangle\!\rangle(y)$.

f respects observables O (briefly, respects O) if f weakly respects observables O and

(b) for all $x \in A$, $O \cap B\langle\!\langle f(x)\rangle\!\rangle \subseteq A\langle\!\langle x\rangle\!\rangle$. ■

The intent of both properties, 2.5(a) and 2.5(b), is to treat elements of O not only as observables, but as *halting behaviors.* Under this interpretation, weak Property 2.5(a) says that each halting behavior in $A\langle\!\langle x\rangle\!\rangle$ maps to the same halting behavior in $B\langle\!\langle f(x)\rangle\!\rangle$, and non-halting behavior in $A\langle\!\langle x\rangle\!\rangle$ maps to non-halting behavior in $B\langle\!\langle f(x)\rangle\!\rangle$. Strong Property 2.5(b) says that every halting behavior in $B\langle\!\langle f(x)\rangle\!\rangle$ occurs in $A\langle\!\langle x\rangle\!\rangle$.

Given $f\colon A \to B$ weakly respecting O, Property 2.5(a) already implies $O \cap A\langle\!\langle x\rangle\!\rangle \subseteq B\langle\!\langle f(x)\rangle\!\rangle$, so that Property 2.5(b) is interchangeable with the more symmetric property

(b′) for all $x \in A$, $O \cap B\langle\!\langle f(x)\rangle\!\rangle = O \cap A\langle\!\langle x\rangle\!\rangle$.

Example 2.6 Suppose the texts are just the terms of λ-calculus, $T = \Lambda$ ([Bare84]). Let language $L_{\lambda n}$ consist of sequences in which every consecutive pair of λ-terms in the sequence is related by the compatible reduction relation of λ-calculus, $\longrightarrow_\beta$. To distinguish notationally, in this case, between concatenation of successive texts in a sequence, and concatenation of successive subterms within a single λ-term, we denote the former by infix "$\rhd$"; thus, $(\lambda\mathrm{x}.\mathrm{y})\mathrm{z}$ is a single text with subterms $(\lambda\mathrm{x}.\mathrm{y})$ and z, while $(\lambda\mathrm{x}.\mathrm{y}) \rhd \mathrm{z}$ is a sequence of two texts (though not belonging to $L_{\lambda n}$, since $(\lambda\mathrm{x}.\mathrm{y}) \not\longrightarrow_\beta \mathrm{z}$). Figure 2 shows some sequences of $L_{\lambda n}$. Sequences x_3 and x_9 have no proper suffixes in $L_{\lambda n}$, because the final terms of these sequences are normal forms; that is, $L_{\lambda n}\langle\!\langle x_3\rangle\!\rangle = L_{\lambda n}\langle\!\langle x_9\rangle\!\rangle = \{\epsilon\}$. Sequences x_4, x_5, x_6, x_7, x_8 each have infinitely many proper suffixes in $L_{\lambda n}$, since $(\lambda\text{x.xx})(\lambda\text{x.xx}) \longrightarrow_\beta (\lambda\text{x.xx})(\lambda\text{x.xx})$.

Let language $L_{\lambda v}$ be as $L_{\lambda n}$ except that consecutive texts are related by the compatible reduction relation of the call-by-value λ_v-calculus, $\longrightarrow_v$ ([Plo75]). $\longrightarrow_v$ differs from $\longrightarrow_\beta$ (and, correspondingly, λ_v-calculus differs from λ-calculus) in that a combination $(\lambda x.M)N$ is a redex only if the operand N is not a combination; thus, $\longrightarrow_v \subset \longrightarrow_\beta$, and $L_{\lambda v} \subset L_{\lambda n}$. Of the sequences in Figure 2, all except x_9 belong to $L_{\lambda v}$; but $(\lambda\text{x.y})((\lambda\text{x.xx})(\lambda\text{x.xx})) \not\longrightarrow_v \text{y}$, because operand $(\lambda\text{x.xx})(\lambda\text{x.xx})$ is a combination.

Consider the identity function $f_{\mathbf{id}}\colon L_{\lambda v} \to L_{\lambda n},\ f_{\mathbf{id}}(w) = w$. This is well-defined since $L_{\lambda v} \subset L_{\lambda n}$, and evidently respects prefixes. For every choice of observables $O \subseteq T$, $f_{\mathbf{id}}$ weakly respects O (i.e., satisfies Property 2.5(a)). However, $f_{\mathbf{id}}$ respects O (i.e., satisfies Property 2.5(b)) only if $O = \{\}$. To see this, suppose some λ-term $M \in O$. Let x be some variable that doesn't occur free in M, and let $w = (\lambda x.M)((\lambda\text{x.xx})(\lambda\text{x.xx}))$. Then $w \longrightarrow_\beta M$, but $w \not\longrightarrow_v M$; therefore, $M \in L_{\lambda n}\langle\!\langle w\rangle\!\rangle$ but $M \notin L_{\lambda v}\langle\!\langle w\rangle\!\rangle$. ∎

Theorem 2.7 Suppose texts T, functions f, g between languages over T, and observables $O, O' \subseteq T$.

If f respects O, then f weakly respects O.
If f weakly respects O, and f is surjective, then f respects O.

If f, g respect prefixes, and $g \circ f$ is defined, then $g \circ f$ respects prefixes.
If f, g weakly respect O, and $g \circ f$ is defined, then $g \circ f$ weakly respects O.
If f, g respect O, and $g \circ f$ is defined, then $g \circ f$ respects O.

If f weakly respects O, and $O' \subseteq O$, then f weakly respects O'.
If f respects O, and $O' \subseteq O$, then f respects O'. ∎

All of these results follow immediately from the definitions.

Theorem 2.8 Suppose texts T, functions f, g between languages over T, observables $O \subseteq T$, and f, g respect prefixes.

If g weakly respects O and $g \circ f$ weakly respects O, then f weakly respects O.
If g weakly respects O and $g \circ f$ respects O, then f respects O.

If f is surjective and respects O, and $g \circ f$ weakly respects O, then g weakly respects O.
If f is surjective and respects O, and $g \circ f$ respects O, then g respects O. ∎

Proof. Suppose $f\colon A_1 \to A_2$ respects prefixes, $g\colon A_2 \to A_3$ weakly respects O, and $g \circ f$ weakly respects O. Suppose $x \in A_1,\ y \in A_1\langle\!\langle x \rangle\!\rangle$, and $o \in O$. We must show that o is a prefix of y iff o is a prefix of $f\langle\!\langle x \rangle\!\rangle(y)$.

Since $g \circ f$ weakly respects O, o is a prefix of y iff o is a prefix of $(g \circ f)\langle\!\langle x \rangle\!\rangle(y)$. Since f, g respect prefixes, $(g \circ f)\langle\!\langle x \rangle\!\rangle(y) = g\langle\!\langle f(x) \rangle\!\rangle(f\langle\!\langle x \rangle\!\rangle(y))$. Since g weakly respects O, o is a prefix of $f\langle\!\langle x \rangle\!\rangle(y)$ iff o is a prefix of $g\langle\!\langle f(x) \rangle\!\rangle(f\langle\!\langle x \rangle\!\rangle(y))$. Therefore, o is a prefix of y iff o is a prefix of $f\langle\!\langle x \rangle\!\rangle(y)$.

Suppose $f\colon A_1 \to A_2$ respects prefixes, $g\colon A_2 \to A_3$ weakly respects O, and $g \circ f$ respects O. By the above reasoning, f weakly respects O. We must show that for all $x \in A_1,\ O \cap A_2\langle\!\langle f(x) \rangle\!\rangle \subseteq A_1\langle\!\langle x \rangle\!\rangle$.

Since g weakly respects O, $O \cap A_2\langle\!\langle f(x) \rangle\!\rangle \subseteq O \cap A_3\langle\!\langle (g \circ f)(x) \rangle\!\rangle$. Since $g \circ f$ respects O, $O \cap A_3\langle\!\langle (g \circ f)(x)) \rangle\!\rangle = O \cap A_1\langle\!\langle x \rangle\!\rangle$; so $O \cap A_2\langle\!\langle f(x) \rangle\!\rangle \subseteq O \cap A_1\langle\!\langle x \rangle\!\rangle$.

Suppose $f\colon A_1 \to A_2$ is surjective and respects O, $g\colon A_2 \to A_3$ respects prefixes, and $g \circ f$ weakly respects O. Suppose $x \in A_2,\ y \in A_2\langle\!\langle x \rangle\!\rangle$, and $o \in O$. We must show that o is a prefix of y iff o is a prefix of $g\langle\!\langle x \rangle\!\rangle(y)$.

Since f is surjective and respects prefixes, let $x' \in A_1$ such that $f(x') = x$, and $y' \in A_1\langle\!\langle x' \rangle\!\rangle$ such that $f\langle\!\langle x' \rangle\!\rangle(y') = y$. Since f weakly respects O, o is a prefix of y' iff o is a prefix of y. Since $g \circ f$ weakly respects O, o is a prefix of y' iff o is a prefix of $(g \circ f)\langle\!\langle x' \rangle\!\rangle(y')$, iff o is a prefix of y. But f, g respect prefixes, so $(g \circ f)\langle\!\langle x' \rangle\!\rangle(y') = g\langle\!\langle f(x') \rangle\!\rangle(f\langle\!\langle x' \rangle\!\rangle(y')) = g\langle\!\langle x \rangle\!\rangle(y)$, and o is a prefix of $g\langle\!\langle x \rangle\!\rangle(y)$ iff o is a prefix of y.

Suppose $f\colon A_1 \to A_2$ is surjective and respects O, $g\colon A_2 \to A_3$ respects prefixes, and $g \circ f$ respects O. By the above reasoning, g weakly respects O. We must show that for all $x \in A_2,\ O \cap A_3\langle\!\langle g(x) \rangle\!\rangle \subseteq A_2\langle\!\langle x \rangle\!\rangle$.

Since f is surjective, let $x' \in A_1$ such that $f(x') = x$. Since f and $g \circ f$ respect O, $O \cap A_2\langle\!\langle x \rangle\!\rangle = O \cap A_1\langle\!\langle x' \rangle\!\rangle = O \cap A_3\langle\!\langle (g \circ f)(x') \rangle\!\rangle = O \cap A_3\langle\!\langle g(x) \rangle\!\rangle$. ■

3 Test suite

Two of the simplest and most common techniques for abstraction are (1) give a name to something, so that it can be referred to thereafter by that name without having to know and repeat its entire definition; and (2) hide some local name from being globally visible.

To exemplify these two techniques in a simple, controlled setting, we describe here three toy programming languages, which we call L_0, L_{priv}, and L_{pub}. L_0 embodies the ability to name things, and also sets the stage for modular hiding by providing suitable modular structure: it has modules which can contain integer-valued fields, and interactive sessions that allow field values to be observed. L_{priv} adds to L_0 the ability to make module fields private, so they can't be accessed from outside the

$$
\begin{array}{rcl}
\langle\text{text}\rangle & \longrightarrow & \langle\text{module}\rangle \mid \langle\text{query}\rangle \mid \langle\text{result}\rangle \\[4pt]
\langle\text{module}\rangle & \longrightarrow & \texttt{module}\ \langle\text{name}\rangle\ \texttt{\{}\ \langle\text{field}\rangle^*\ \texttt{\}} \\
\langle\text{field}\rangle & \longrightarrow & \langle\text{name}\rangle\ \texttt{=}\ \langle\text{expr}\rangle\ \texttt{;} \\
\langle\text{expr}\rangle & \longrightarrow & \langle\text{qualified name}\rangle \mid \langle\text{integer}\rangle \\
\langle\text{qualified name}\rangle & \longrightarrow & \langle\text{name}\rangle\ \texttt{.}\ \langle\text{name}\rangle \\[4pt]
\langle\text{query}\rangle & \longrightarrow & \texttt{query}\ \langle\text{expr}\rangle \\[4pt]
\langle\text{result}\rangle & \longrightarrow & \texttt{result}\ \langle\text{integer}\rangle
\end{array}
$$

Figure 3: CFG for L_0.

module. L_{pub} is like L_{priv} except that the privacy designations aren't enforced.

The context-free grammar for texts of L_0 is given in Figure 3. A text sequence is any string of texts that satisfies all of the following.

- The sequence does not contain more than one module with the same name.
- No one module in the sequence contains more than one field with the same name.
- Each qualified name consists of the name of a module and the name of a field within that module, where either the module occurs earlier in the sequence, or the module is currently being declared and the field occurs earlier in the module.
- When a result occurs in the sequence, it is immediately preceded by a query.
- When a query in the sequence is not the last text in the sequence, the text immediately following the query must be a result whose integer is the value of the expression in the query.

When we consider program transformations between these languages, we will choose as observables exactly the results.

Language L_{priv} is similar to L_0, except that there is an additional syntax rule

$$\langle\text{field}\rangle \longrightarrow \texttt{private}\ \langle\text{name}\rangle\ \texttt{=}\ \langle\text{expr}\rangle\ \texttt{;} \tag{5}$$

and the following additional constraint.

- A qualified name can only refer to a `private` field if the qualified name is inside the same module as the `private` field.

Language L_{pub} is similar to L_{priv}, but without the prohibition against non-local references to `private` fields.

We immediately observe some straightforward properties of these languages.

$L_0 \subset L_{priv} \subset L_{pub}$.

Every query in L_0, L_{priv}, or L_{pub} has a unique result. That is, if $w \in L_k$ and w ends with a query, then there is exactly one result r such that $wr \in L_k$. (Note that this statement is possible only because a query and its result are separate texts.)

Removing a `private` keyword from an L_{priv}-program only expands its meaning, that is, allows additional successor sequences without invalidating any previously allowed successor sequence. Likewise, removing a `private` keyword from an L_{pub}-program. (In fact, we made sure of this by insisting that references to a field must always be qualified by module-name.) Let $U\colon L_{pub} \to L_0$ be the syntactic transformation "remove all `private` keywords"; then for all $wx \in L_{pub},\ (U(w))x \in L_{pub}$; and for all $wx \in L_{priv},\ (U(w))x \in L_{priv}$.

Intuitively, L_0 and L_{pub} are about equal in abstractive power, since the `private` keywords in L_{pub} are at best documentation of intent; while L_{priv} has more abstractive powerful, since it allows all L_0-programs to be written and also grants the additional ability to hide local fields.

4 Expressive power

As noted in §2, for language comparison we will be concerned both with *whether* programs of language A can be rewritten as programs of language B and, if so, with *how difficult* the rewriting is, i.e., how much the rewriting disrupts program structure. Macro (a.k.a. polynomial) transformations are notably of interest, as they correspond to Landin's notion of syntactic sugar; but abstraction also sometimes involves more sophisticated transformations (see, e.g., [SteSu76]), so we will parameterize our theory by the kind of transformations permitted.

Definition 4.1 A *morphism* from language A to language B is a function $f\colon A \to B$ that respects prefixes.

For any language A, the *identity morphism* is denoted $\mathbf{id}_A\colon A \to A$. (That is, $\mathbf{id}_A(x) = x$.)

A *category over* T is a family of morphisms between languages over T that is closed under composition and includes the identity morphism of each language over T. ■

Here are some simple and significant categories (relative to T):

Definition 4.2

Category $\mathbf{Any}_T$ consists of all morphisms between languages over T.

Category $\mathbf{Map}_T$ consists of all morphisms $f \in \mathbf{Any}_T$ such that f performs some transformation $\tau_f\colon T \to T^*$ uniformly on each text of the sequence. (That is, $f(t_1t_2 \ldots t_n) = \tau_f(t_1)\tau_f(t_2) \ldots \tau_f(t_n)$.)

Category $\mathbf{Inc}_T$ consists of all morphisms $f \in \mathbf{Any}_T$ such that, for all $x \in \mathrm{dom}(f)$, $f(x) = x$. Elements of $\mathbf{Inc}_T$ are called *inclusion morphisms* (briefly, inclusions).

Category $\mathbf{Obs}_{T,O}$ consists of all morphisms $f \in \mathbf{Any}_T$ such that f respects observables O.

Category $\mathbf{WObs}_{T,O}$ consists of all morphisms $f \in \mathbf{Any}_T$ such that f weakly respects observables O. ■

These five sets of morphisms are indeed categories over T, since every identity morphism over T satisfies all five criteria, and each of the five criteria is composable (i.e., $P(f) \wedge P(g) \Rightarrow P(g \circ f)$). Composability was noted in Theorem 2.7 for three of the five criteria: respects prefixes ($\mathbf{Any}_T$), respects O ($\mathbf{Obs}_{T,O}$), and weakly respects O ($\mathbf{WObs}_{T,O}$).

Having duly acknowledged the dependence of categories on the universe of texts T, we omit subscript T from category names hereafter.

Since *respects* O implies *weakly respects* O, $\mathbf{Obs}_O \subseteq \mathbf{WObs}_O$. Inclusion morphisms satisfy the criteria for three out of four of the other categories: $\mathbf{Inc} \subseteq \mathbf{Any}$ (trivially, since every category has to be a subset of $\mathbf{Any}$ by definition), $\mathbf{Inc} \subseteq \mathbf{Map}$, and $\mathbf{Inc} \subseteq \mathbf{WObs}_O$. $\mathbf{Inc} \not\subseteq \mathbf{Obs}_O$ in general, since the codomain of an inclusion morphism is a superset of its domain and might therefore contain additional sequences that defeat Property 2.5(b).

The intersection of any two categories is a category; so, in particular, for any category C and any observables O one has categories $C \cap \mathbf{Obs}_O$ and $C \cap \mathbf{WObs}_O$.

Felleisen's notion of expressiveness ([Fe91]) was that language B "expresses" language A if every program of A can be rewritten as a program of B with the same semantics by means of a suitably well-behaved transformation $\phi\colon A \to B$. He considered two different classes of transformations (as well as two different degrees of semantic same-ness, corresponding to our distinction between "respects O" and "weakly respects O"). We are now positioned to formulate criteria analogous to Felleisen's expressiveness and weak expressiveness, relative to an arbitrary category of morphisms (rather than to just the two kinds of morphisms in [Fe91]).

Definition 4.3 Suppose category C, observables O, languages A, B.

B *C-expresses A for observables O* (or, B is as C, O expressive as A), denoted $A \sqsubseteq^C_O B$, if there exists a morphism $f\colon A \to B$ in $C \cap \mathbf{Obs}_O$.

B *weakly C-expresses A for observables O* (or, B is weakly as C, O expressive as A), denoted $A \sqsubseteq^C_{\mathrm{wk}\,O} B$, if there exists a morphism $f\colon A \to B$ in $C \cap \mathbf{WObs}_O$. ■

Theorem 4.4

If $A_1 \sqsubseteq^C_O A_2$ and $A_2 \sqsubseteq^C_O A_3$ then $A_1 \sqsubseteq^C_O A_3$.

If $A_1 \sqsubseteq^C_{\mathrm{wk}\,O} A_2$ and $A_2 \sqsubseteq^C_{\mathrm{wk}\,O} A_3$ then $A_1 \sqsubseteq^C_{\mathrm{wk}\,O} A_3$.

If $A \sqsubseteq^C_O B$, $C \subseteq C'$, and $O' \subseteq O$, then $A \sqsubseteq^{C'}_{O'} B$.

If $A \sqsubseteq^C_{\mathrm{wk}\,O} B$, $C \subseteq C'$, and $O' \subseteq O$, then $A \sqsubseteq^{C'}_{\mathrm{wk}\,O'} B$.

If $A \sqsubseteq^C_O B$, then $A \sqsubseteq^C_{\mathrm{wk}\,O} B$. ■

Proof. Transitivity follows from the fact that the intersection of categories is closed under composition; monotonicity in the category, from $(C \subseteq C') \Rightarrow (C \cap X \subseteq C' \cap X)$. Monotonicity in the observables follows from monotonicity of observation (Theorem 2.7), and strong expressiveness implies weak expressiveness because $\mathbf{Obs}_O \subseteq \mathbf{WObs}_O$ (also ultimately from Theorem 2.7). ■

When showing that B cannot C-express A, we will want to use a large category C since this makes non-existence a strong result; while, in showing that B *can* C-express A, we will want to use a *small* category C for a strong result.

Category **Map** is a particularly interesting large category on which one might hope to prove inexpressibility of abstraction facilities, because its morphisms perform arbitrary transformations on individual texts, but are prohibited from the signature property of abstraction: dependence of the later texts in a sequence on the preceding texts.

On the other hand, category **Inc** seems to be the smallest category on which expressibility is at all interesting.

Theorem 4.5 For all languages A, B over texts T, $A \subseteq B$ iff $A \sqsubseteq^{\mathbf{Inc}}_{\mathrm{wk}\,T} B$. ■

From Theorems 4.5 and 4.4, $A \subseteq B$ implies $A \sqsubseteq^{\mathbf{Inc}}_{\mathrm{wk}\,O} B$ for every choice of observables O.

For the current work, we identify just one category intermediate in size between **Map** and **Inc**, that of macro transformations, which were of particular interest to Landin and Felleisen.

Definition 4.6 Category **Macro** consists of all morphisms $f \in \mathbf{Map}$ such that the corresponding text transformation τ_f is polynomial; that is, in term-algebraic notation (where a CFG rule with n nonterminals on its right-hand side appears as an n-ary operator), for each n-ary operator σ that occurs in $\mathrm{dom}(f)$, there exists a polynomial σ_f such that $\tau_f(\sigma(t_1, \ldots, t_n)) = \sigma_f(\tau_f(t_1), \ldots, \tau_f(t_n))$. ■

(Our **Macro** is a larger class of morphisms than Felleisen described, because he imposed the further constraint that if σ is present in both languages then $\sigma_f = \sigma$. The larger class supports stronger non-existence results; and, moreover, when comparing realistic programming languages we will want measures that transcend superficial differences of notation.)

Recall toy languages L_0, L_{priv}, L_{pub} from §3. From Theorem 4.5, $L_0 \sqsubseteq^{\mathbf{Inc}}_{\mathrm{wk}\,O} L_{priv} \sqsubseteq^{\mathbf{Inc}}_{\mathrm{wk}\,O} L_{pub}$ (where observables O are the results). Since queries always get results in all three languages, so that there is no discrepancy in "halting behavior", $L_0 \sqsubseteq^{\mathbf{Inc}}_{O} L_{priv} \sqsubseteq^{\mathbf{Inc}}_{O} L_{pub}$. We can't get very strong inexpressiveness results on these languages, because it really isn't very difficult to rewrite L_{priv}-programs and L_{pub}-programs so that they will work in all three languages; one only needs the "remove all `private` keywords" function U. Thus, for any category C that admits inclusion morphisms and

U (in its manifestations as a morphism $L_{priv} \to L_0$ and as a morphism $L_{pub} \to L_{priv}$), $L_0 \equiv^C_O L_{priv} \equiv^C_O L_{pub}$. In particular, $L_0 \equiv^{\mathbf{Macro}}_O L_{priv} \equiv^{\mathbf{Macro}}_O L_{pub}$.

Yet, our intuitive ordering of these languages by abstractive power is $L_0 = L_{pub} < L_{priv}$; so apparently C, O expressiveness doesn't match our understanding of abstractive power.

5 Abstractive power

To correct this difficulty, we generalize an additional strategem from a different treatment of "expressiveness", by John C. Mitchell ([Mi93]). Mitchell was concerned with whether a transformation between programming languages would preserve the information-hiding properties of program contexts; in terms of our framework, he required of $f\colon A \to B$ that if there is no observable difference between $xy \in A$ and $xy' \in A$ (that is, x hides the distinction between y and y'), then there is no observable difference between $f(xy) \in B$ and $f(xy') \in B$, and vice versa.

We take from Mitchell's treatment the idea that f should preserve observable relationships between programs. However, our treatment is literally deeper than Mitchell's. He modeled a programming language as a mapping from syntactic terms to semantic results: his analog of prefix x was a context (a polynomial of arity one with exactly one occurrence of the variable); of suffix y, a subterm to substitute into the context; and of $A\langle\!\langle xy\rangle\!\rangle$, an element of an arbitrary semantic domain. By requiring that f preserve observational equivalence of subterms y, he was able to define an 'abstractive' ordering of programming languages, in the merely information-hiding sense of *abstraction* — but he could not address any other abstractive techniques than simple information hiding, because his model of *programming language* wouldn't support them. Abstractive power in our general sense concerns what can be expressed in languages $A\langle\!\langle xy\rangle\!\rangle$ and $A\langle\!\langle xy'\rangle\!\rangle$, hence we are concerned with their general linguistic properties, such as their ordering by expressive power; but under Mitchell's model, semantic values $A\langle\!\langle xy\rangle\!\rangle$ and $A\langle\!\langle xy'\rangle\!\rangle$ are ends of computation, not further programming languages, and so they have no linguistic properties to compare. Since our semantic values are programming languages, we can and do require that f preserve expressive ordering of behaviors.

Definition 5.1 A *programming language with expressive structure* (briefly, an expressive language) is a tuple $L = \langle S, C\rangle$ where S is a language and C is a category. For expressive language L, its language component is denoted $\mathrm{seq}(L)$, and its category component, $\mathrm{cat}(L)$; that is, $L = \langle \mathrm{seq}(L), \mathrm{cat}(L)\rangle$.

For expressive language $L = \langle S, C\rangle$ and category D, notation $L \cap D$ signifies expressive language $\langle S, (C \cap D)\rangle$.

For expressive language $L = \langle S, C\rangle$ and sequence x, L *reduced by* x is $L\langle\!\langle x\rangle\!\rangle = \langle S\langle\!\langle x\rangle\!\rangle, C\rangle$. ■

$$
\begin{tikzcd}
\operatorname{seq}(A) \arrow[rr, "f"] \arrow[dr, "x"'] \arrow[ddr, "y"] & & \operatorname{seq}(B) \arrow[dr, "f(x)"] \arrow[ddr, "f(y)"] & \\
& \operatorname{seq}(A)\langle\!\langle x \rangle\!\rangle \arrow[rr, "f\langle\!\langle x \rangle\!\rangle"'] \arrow[d, "g"'] & & \operatorname{seq}(B)\langle\!\langle f(x) \rangle\!\rangle \arrow[d, "h"'] \\
& \operatorname{seq}(A)\langle\!\langle y \rangle\!\rangle \arrow[rr, "f\langle\!\langle y \rangle\!\rangle"'] & & \operatorname{seq}(B)\langle\!\langle f(y) \rangle\!\rangle
\end{tikzcd}
$$

Figure 4: Elements of the criteria for an expressive morphism.

Definition 5.2 Suppose expressive languages A, B.

A morphism $f\colon \operatorname{seq}(A) \to \operatorname{seq}(B)$ is a *morphism from* A *to* B, denoted $f\colon A \to B$, if both of the following conditions hold.

(a) For all $x, y \in \operatorname{seq}(A)$ and $g \in \operatorname{cat}(A)$ with $g\colon \operatorname{seq}(A)\langle\!\langle x \rangle\!\rangle \to \operatorname{seq}(A)\langle\!\langle y \rangle\!\rangle$, there exists $h \in \operatorname{cat}(B)$ with $h\colon \operatorname{seq}(B)\langle\!\langle f(x) \rangle\!\rangle \to \operatorname{seq}(B)\langle\!\langle f(y) \rangle\!\rangle$ such that $f\langle\!\langle y \rangle\!\rangle \circ g = h \circ f\langle\!\langle x \rangle\!\rangle$.

(b) For all $x, y \in \operatorname{seq}(A)$, if there is no $g \in \operatorname{cat}(A)$ with $g\colon \operatorname{seq}(A)\langle\!\langle x \rangle\!\rangle \to \operatorname{seq}(A)\langle\!\langle y \rangle\!\rangle$, then there is no $h \in \operatorname{cat}(B)$ with $h\colon \operatorname{seq}(B)\langle\!\langle f(x) \rangle\!\rangle \to \operatorname{seq}(B)\langle\!\langle f(y) \rangle\!\rangle$. ■

The elements of Conditions 5.2(a) and 5.2(b) are illustrated in Figure 4.

Definition 5.2 is designed so that specific properties of the expressive structure of A are mapped into B. This is why, in Condition 5.2(a), we require not only that for each g there *is* an h, but that there is an h that makes the diagram in Figure 4 commute. Without the commutative diagram, we couldn't deduce properties of h (beyond the identity of its domain and codomain) from properties of g, or vice versa; cf. Theorem 5.4, below. We also need Condition 5.2(b), to prevent f from collapsing the expressive structure of A: if A allows us to create two languages that are distinct from each other, $\operatorname{seq}(A)\langle\!\langle x \rangle\!\rangle$ and $\operatorname{seq}(A)\langle\!\langle y \rangle\!\rangle$ such that the latter can't express the former, then f should map these into distinct languages of B, which is just what Condition 5.2(b) guarantees. We do not ask for the strictly stronger property, symmetric to Condition 5.2(a), that for every h there is a g that makes the diagram commute, because that would impose detailed structure of all $\mathrm{cat}(B)$ morphisms $\mathrm{seq}(B)\langle\!\langle f(x)\rangle\!\rangle \to \mathrm{seq}(B)\langle\!\langle f(y)\rangle\!\rangle$ onto $\mathrm{cat}(A)$, subverting our intent that B can have more expressive structure than A.

Definition 5.3 Suppose category C, observables O, expressive languages A, B.

B *C-expresses A for observables O* (or, B is as C, O abstractive as A), denoted $A \leq^C_O B$, if there exists a morphism $f\colon A \to B$ in $C \cap \mathbf{Obs}_O$.

B *weakly C-expresses A for observables O* (or, B is weakly as C, O abstractive as A), denoted $A \leq^C_{\mathrm{wk}\,O} B$, if there exists a morphism $f\colon A \to B$ in $C \cap \mathbf{WObs}_O$. ∎

Theorem 5.4 Suppose expressive languages A, B, category C, and observables O.

If $A \leq^C_{\mathrm{wk}\,O} B$ and $\mathrm{cat}(B) \subseteq \mathbf{WObs}_O$, then $\mathrm{cat}(A) \subseteq \mathbf{WObs}_O$.
If $A \leq^C_O B$ and $\mathrm{cat}(B) \subseteq \mathbf{Obs}_O$, then $\mathrm{cat}(A) \subseteq \mathbf{Obs}_O$. ∎

These results follow immediately from Theorem 2.8. One can also use Theorem 2.8 to reason back and forth between particular expressiveness relations in B and in A (but we won't formulate any specific theorem to that effect, as it is easier to work directly from the general theorem).

Theorem 5.5 Suppose expressive languages A, A_k, B, B_k, categories C, C', D, D_k, and observables O, O'.

If $A_1 \leq^C_O A_2$ and $A_2 \leq^C_O A_3$, then $A_1 \leq^C_O A_3$.
If $A_1 \leq^C_{\mathrm{wk}\,O} A_2$ and $A_2 \leq^C_{\mathrm{wk}\,O} A_3$, then $A_1 \leq^C_{\mathrm{wk}\,O} A_3$.

If $A \leq^C_O B$, $C \subseteq C'$, and $O' \subseteq O$, then $A \leq^{C'}_{O'} B$.
If $A \leq^C_{\mathrm{wk}\,O} B$, $C \subseteq C'$, and $O' \subseteq O$, then $A \leq^{C'}_{\mathrm{wk}\,O'} B$.

Suppose D is the compositional closure of $D_1 \cup D_2$.
If $A \leq^C_O B \cap D_1$ and $A \leq^C_O B \cap D_2$, then $A \leq^C_O B \cap D$.
If $A \leq^C_{\mathrm{wk}\,O} B \cap D_1$ and $A \leq^C_{\mathrm{wk}\,O} B \cap D_2$, then $A \leq^C_{\mathrm{wk}\,O} B \cap D$.

If $A \leq^C_O B$ then $A \leq^C_{\mathrm{wk}\,O} B$. ∎

All of these results follow immediately from the definitions.

Theorem 5.5 reveals why we introduced *expressive languages*, instead of defining abstractiveness as a relation between sets of sequences. Given relation

$$\langle S_1, (C \cap \mathbf{WObs}_O)\rangle \quad \leq^C_O \quad \langle S_2, (C \cap \mathbf{WObs}_O)\rangle\,, \tag{6}$$

we cannot usually conclude

$$\langle S_1, (C' \cap \mathbf{WObs}_{O'})\rangle \quad \leq^{C'}_{O'} \quad \langle S_2, (C' \cap \mathbf{WObs}_{O'})\rangle \tag{7}$$

for smaller *or* larger C', nor for smaller or larger O', because these variations work differently in the three different positions: they can be weakened generally at $\leq$, and weakened with some care at S_2, and cannot be varied either way at S_1 (unless due to some peculiar convergence of properties of the categories in all three positions).

In practice when investigating $A \leq^C_O B$, our first concern will be the size of cat(A), regardless of what sizes we need for C and cat(B) to obtain our results — because the size of cat(A) addresses *whether* seq(B) can abstractively model particular transformations on seq(A), whereas the sizes of C and cat(B) address merely *how readily* it can do so (which is of only subsequent interest).

Theorem 5.6 Suppose observables O are the results for toy languages L_0, L_{priv}, L_{pub} of §3, and U the syntactic transformation $L_{pub} \to L_0$ of §3.

Then $\langle L_0, \mathbf{Inc}\rangle =^{\mathbf{Inc}\,\cup\,\{U\}}_O \langle L_{pub}, \mathbf{Inc}\rangle <^{\mathbf{Macro}}_{\mathrm{wk}\,O} \langle L_{priv}, \mathbf{Inc}\rangle$ (allowing all meaningful domains and codomains for U in category $\mathbf{Inc} \cup \{U\}$). ■

Proof. Recall that in category **Inc**, for any languages A, B there is at most one morphism $A \to B$, and that morphism exists iff $A \subseteq B$. Thus, $\langle A, \mathbf{Inc}\rangle \leq^C_O \langle B, \mathbf{Inc}\rangle$ iff there exists $f\colon A \to B$ with $f \in C \cap \mathbf{Obs}_O$ such that for all $x, y \in A$,

(a) if $A\langle\!\langle x\rangle\!\rangle \subseteq A\langle\!\langle y\rangle\!\rangle$, then $B\langle\!\langle f(x)\rangle\!\rangle \subseteq B\langle\!\langle f(y)\rangle\!\rangle$ and, for all $z \in A\langle\!\langle x\rangle\!\rangle$, $f\langle\!\langle x\rangle\!\rangle(z) = f\langle\!\langle y\rangle\!\rangle(z)$; and

(b) if $A\langle\!\langle x\rangle\!\rangle \not\subseteq A\langle\!\langle y\rangle\!\rangle$, then $B\langle\!\langle f(x)\rangle\!\rangle \not\subseteq B\langle\!\langle f(y)\rangle\!\rangle$.

Suppose $A \in \{L_0, L_{pub}\}$, $B \in \{L_0, L_{priv}, L_{pub}\}$, and $x, y \in A$. Since A either doesn't have private fields or doesn't enforce them, $A\langle\!\langle x\rangle\!\rangle = A\langle\!\langle U(x)\rangle\!\rangle$ and $A\langle\!\langle y\rangle\!\rangle = A\langle\!\langle U(y)\rangle\!\rangle$. So, $A\langle\!\langle x\rangle\!\rangle \subseteq A\langle\!\langle y\rangle\!\rangle$ iff $A\langle\!\langle U(x)\rangle\!\rangle \subseteq A\langle\!\langle U(y)\rangle\!\rangle$; and both of these mean that $U(y)$ declares at least as much as $U(x)$, consistently with $U(x)$, and therefore, regardless of whether B has or enforces private fields, $B\langle\!\langle U(x)\rangle\!\rangle \subseteq B\langle\!\langle U(y)\rangle\!\rangle$. If $A\langle\!\langle x\rangle\!\rangle \not\subseteq A\langle\!\langle y\rangle\!\rangle$, then $A\langle\!\langle U(x)\rangle\!\rangle \not\subseteq A\langle\!\langle U(y)\rangle\!\rangle$; $U(y)$ either does not declare everything that $U(x)$ does, or declares it inconsistently with $U(x)$; and therefore, $B\langle\!\langle U(x)\rangle\!\rangle \not\subseteq B\langle\!\langle U(y)\rangle\!\rangle$. Also, since $U \in \mathbf{Map}$, $U\langle\!\langle x\rangle\!\rangle(z) = U(z) = U\langle\!\langle y\rangle\!\rangle(z)$. Therefore, since $U \in \mathbf{Obs}_O$, $\langle A, \mathbf{Inc}\rangle \leq^{\mathbf{Inc}\,\cup\,\{U\}}_O \langle B, \mathbf{Inc}\rangle$.

For the remainder of the theorem, it suffices to show that $\langle L_{priv}, \mathbf{Inc}\rangle \not\leq^{\mathbf{Macro}}_{\mathrm{wk}\,O} \langle L_0, \mathbf{Inc}\rangle$. Suppose $f\colon \langle L_{priv}, \mathbf{Inc}\rangle \to \langle L_0, \mathbf{Inc}\rangle$ in **Macro**, and let τ_f be the corresponding polynomial text-transformation.

Consider the behavior of τ_f on L_{priv} fields

$$\begin{array}{lcl} \phi_{pub} & = & \text{“}n\ \texttt{=}\ e\ \texttt{;}\text{”} \\ \phi_{priv} & = & \text{“}\texttt{private}\ n\ \texttt{=}\ e\ \texttt{;}\text{”}\,. \end{array} \tag{8}$$

When ϕ_{pub} is embedded in an L_{priv} sequence, varying e affects what observables may occur later in the sequence, while varying n affects what non-observables may occur later in the sequence. The smallest element of L_0 syntax with parts that affect later observables and parts that don't is the field; so $\tau_f(\phi_{pub})$ must contain one or more fields. Since there are L_{priv} sequences in which ϕ_{priv} can be substituted for ϕ_{pub}, there must be L_0 sequences in which $\tau_f(\phi_{priv})$ can be substituted for $\tau_f(\phi_{pub})$; so $\tau_f(\phi_{priv})$ and $\tau_f(\phi_{pub})$ belong to the same syntactic class in L_0, and, since $\tau_f(\phi_{priv})$ can't be empty (because it has *some* consequences), $\tau_f(\phi_{priv})$ must contain one or more fields.

When ϕ_{priv} is embedded in an L_{priv} sequence σ, it affects what other fields may occur in σ; but it is possible that no other fields in σ depend on it, in which case it could be deleted from σ with no affect on what sequences may follow σ. However, in L_0 there is no such thing as a field embedded in a sequence whose deletion would not affect what further sequences could follow it. Therefore, no choice of one or more fields in $\tau_f(\phi_{priv})$ will satisfy the supposition for f, and by *reductio ad absurdum*, no such f exists. ∎

In comparing realistic programming languages, it may plausibly happen that the abstractive power of language A is not exactly duplicated by language B, but from B one can *abstract* to a language $B\langle\!\langle x\rangle\!\rangle$ that does capture the abstractive power of A (say, $A \not\leq^{\mathbf{Map}}_{\mathrm{wk}\,O} B$ but $A \leq^{\mathbf{Map}}_{\mathrm{wk}\,O} B\langle\!\langle x\rangle\!\rangle$). Formally,

Definition 5.7 For any category C, category $\mathbf{Pfx}(C)$ consists of all morphisms $f \in \mathbf{Any}$ such that there exists $b \in \mathrm{cod}(f)$ and $g \in C$ with, for all $a \in \mathrm{dom}(f)$, $f(a) = b\,g(a)$. ∎

Using this definition, instead of the cumbersome "$\exists x$ such that $A \leq^{\mathbf{Map}}_{\mathrm{wk}\,O} B\langle\!\langle x\rangle\!\rangle$", one can write "$A \leq^{\mathbf{Pfx}(\mathbf{Map})}_{\mathrm{wk}\,O} B$".

6 Related formal devices

Felleisen's formal notion of expressiveness of programming languages was discussed in §4. That device is concerned with ability to express computation, not with ability to express abstraction.

Mitchell's formal notion of abstraction-preserving transformations between programming languages was discussed at the top of §5. That device is concerned with information hiding by particular contexts.

Another related formal device is *extensibility*, proposed by Krishnamurthi and Felleisen ([KrFe98]). Their device has technical similarities both to Mitchell's device and to ours. They divided each program into a prefix and a suffix, mapped each program into a non-linguistic semantic result, and considered when a given suffix with two different prefixes would map to the same result. Thus, their treatment views each prefix as signifying a language, as in our treatment, but views the completed program as signifying a non-language semantic value, as in Mitchell's treatment. Like Mitchell, they are concerned only with a form of information hiding, so no difficulties arise in their treatment from the absence of linguistic properties of semantic values.

Moreover, the particular form of information hiding they address, *black-box reuse*, applies varying prefixes to a fixed set of suffixes, rather than performing some common transformation f on both prefix and suffix — so that, even though they view prefixes as signifying languages, they only consider expressive equivalences of those languages, not any expressive ordering.

Where both Mitchell's and Krishnamurthi and Felleisen's devices reward hiding of program internals, our device rewards flexibility to selectively hide or not hide program internals.

7 Directions for future work

It was remarked in §2 that this formal treatment of abstractive power only directly studies abstraction at a single level of syntax. No exploration of multi-level abstraction is currently contemplated, since the single-level treatment seems to afford a rich theory, and also appears already able to engage deeper levels of syntax through structural constraints on the categories, as via category **Macro** (cf. Theorem 5.6).

Variant simplified forms of Scheme (as e.g. in [Cl98]) are expected to be particularly suitable as early subjects for abstractive-power comparison of nontrivial programming languages, because trivial syntax and simple, orthogonal semantics should facilitate study of specific, clear-cut semantic variations. Interesting comparisons include

- fully encapsulated procedures (as in standard Scheme) versus various procedure de-encapsulating devices (such as MIT Scheme's `procedure-environment`, or, more subtly, `primitive-procedure?` ([MitScheme])).

- Scheme without hygienic macros versus Scheme with hygienic macros.

- Scheme with hygienic macros versus Scheme with Kernel-style fexprs ([Sh07]).

In each case, it seems likely that a contrast of abstractive power can be demonstrated by choosing sufficiently weak categories; consequently, the larger question is not whether a contrast can be demonstrated, but what categories are required for the demonstration. A Turing-powerful programming language may, in fact, support nontrivial results using expressive category **Map**.

A major class of language features that should be assessed for abstractive power is *strong typing* — that is, typing enforced eagerly, at static analysis time, rather than lazily at run-time. Eager enforcement apparently increases encapsulation while decreasing flexibility; it may even be possible to separate these two aspects of strong typing, by factoring the strong typing feature into two (or more) sub-features, with separate and sometimes-conflicting abstractive properties. Analysis of the abstractive properties of strong typing is currently viewed as a long-term, rather than short-term, goal.

References

[Bare84] Hendrik Pieter Barendregt, *The Lambda Calculus: Its Syntax and Semantics* [*Studies in Logic and the Foundations of Mathematics* 103], Revised Edition, Amsterdam: North Holland, 1984.

The bible of lambda calculus.

[Cl98] William D. Clinger, "Proper Tail Recursion and Space Efficiency", *SIGPLAN Notices* 33 no. 5 (May 1998) [*Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, 17–19 June 1998], pp. 174–185. Available (as of February 2008) at URL:
`http://www.ccs.neu.edu/home/will/papers.html`

[DaHo72] Ole-Johan Dahl and C.A.R. Hoare, "Hierarchical Program Structures", in Ole-Johan Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming* [*A.P.I.C. Studies in Data Processing* 8], New York: Academic Press, 1972, pp. 175–220.

[Fe91] 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. A preliminary version appears 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. Available (as of February 2008) at URLs:
`http://www.cs.rice.edu/CS/PLT/Publications/Scheme/`
`http://www.ccs.neu.edu/scheme/pubs/#scp91-felleisen`

[KrFe98] Shriram Krishnamurthi and Matthias Felleisen, "Toward a Formal Theory of Extensible Software", *Software Engineering Notes* 23 no. 6 (November 1998) [*Proceedings of the ACM SIGSOFT Sixth International Conference on the Foundations of Software Engineering*, FSE-6, SIGSOFT '98, Lake Buena Vista, Florida, November 3–5, 1998], pp. 88–98. Available (as of February 2008) at URLs:
`http://www.cs.rice.edu/CS/PLT/Publications/Scheme/#fse98-kf`
`http://www.cs.brown.edu/~sk/Publications/Papers/Published/kf-ext-sw-def/`
`http://www.ccs.neu.edu/scheme/pubs/#fse98-kf`

[Mi93] John C. Mitchell, "On Abstraction and the Expressive Power of Programming Languages", *Science of Computer Programming* 212 (1993) [Special issue of papers from Symposium on Theoretical Aspects of Computer Software, Sendai, Japan, September 24–27, 1991], pp. 141–163. Also, a version

of the paper appeared in: Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software: International Conference TACS'91* [Sendai, Japan, September 24–27, 1991] [*Lecture notes in Computer Science* 526], Springer-Verlag, 1991, pp. 290–310. Available (as of February 2008) at URL:
`http://theory.stanford.edu/people/jcm/publications.htm#typesandsemantics`

[MitScheme] MIT Scheme Reference. Available (as of February 2008) at URL:
`http://www.swiss.ai.mit.edu/projects/scheme/documentation/scheme_toc.html`

[Plo75] Gordon D. Plotkin, "Call-by-name, call-by-value, and the λ-calculus", *Theoretical Computer Science* 1 no. 2 (December 1975), pp. 125–159. Available (as of February 2008) at URL:
`http://homepages.inf.ed.ac.uk/gdp/publications/`

[Sh99] John N. Shutt, "Abstraction in Programming — working definition", technical report WPI-CS-TR-99-38, Worcester Polytechnic Institute, Worcester, MA, December 1999. Available (as of February 2008) at URL:
`http://www.cs.wpi.edu/Resources/Techreports/index.html`

[Sh07] John N. Shutt, "Revised^{-1} Report on the Kernel Programming Language", technical report WPI-CS-TR-05-07, Worcester Polytechnic Institute, Worcester, MA, March 2005, amended 9 September 2007. Available (as of February 2008) at URL:
`http://www.cs.wpi.edu/Resources/Techreports/index.html`

[SteSu76] Guy Lewis Steele Jr. and Gerald Jay Sussman, *LAMBDA: The Ultimate Imperative*, memo 353, MIT AI Lab, March 10, 1976. Available (as of February 2008) at URL:
`http://www.ai.mit.edu/publications/`