# On the Expressive Power of Programming Languages

Matthias Felleisen*
Department of Computer Science
Rice University
Houston, TX 77251-1892

*Abstract*

The literature on programming languages contains an abundance of informal claims on the relative expressive power of programming languages, but there is no framework for formalizing such statements nor for deriving interesting consequences. As a first step in this direction, we develop a formal notion of expressiveness and investigate its properties. To demonstrate the theory's closeness to published intuitions on expressiveness, we analyze the expressive power of several extensions of functional languages. Based on these results, we believe that our system correctly captures many of the informal ideas on expressiveness, and that it constitutes a good basis for further research in this direction.

## 1 Comparing Programming Languages

The literature on programming languages contains an abundance of informal claims on the relative expressive power of programming languages and on the expressibility or non-expressibility of programming constructs with respect to programming languages. Unfortunately, programming language theory does not provide a formal framework for specifying and verifying such statements. This lack makes it impossible to draw any firm conclusions from these claims or to use them for an objective comparison of programming languages.

Landin [10] was the first to propose the development of a formal framework for comparing programming languages. He studied the relationship among programming constructs

*Supported in part by NSF and Darpa

and began to classify some as "essential" and some as "syntactic sugar." Others, most notably Reynolds [17, 18] and Steele and Sussman [19], followed Landin's example. They informally analyzed the expressiveness of imperative extensions of higher-order functional languages and initiated a study of "core" languages. The crucial idea behind the separation of language features is summarized in the remark of Sussman and Steele [19] that there are simple, "syntactically local" translations into applicative languages for many common programming constructs, but that some, notably escape expressions and assignments, involve complex reformulations of large fractions of programs.

The informal approach of Landin and others suggests that a facility is expressible if every usage instance is replaceable by a behaviorally equivalent instantiation of an expression schema. The two key concepts in this idea are *expression schema* and *behavioral equivalence.* These are well-known and widely studied concepts in programming language theory, and are easily adaptable as the basis of a formal definition of expressibility. A first analysis of this definition shows that it supports many informal judgements in the programming language literature. We therefore believe that it correctly formalizes the informal ideas and that it constitutes a basis for further research.

In the following sections, we propose a formal model of expressibility and expressiveness, investigate some of its properties, and analyze the expressive powers of several extensions of functional languages. First, we introduce our formal framework of expressiveness based on the notion of expressibility. Second, we study the expressiveness of an idealized version of Scheme [20] and verify the informal expressibility claims of Steele and Sussman [19] behind the Scheme language design. Finally, we compare our ideas to the study of definability in mathematical logic and put our work in perspective.

## 2 A Formal Theory of Expressiveness

Since Landin's informal ideas about essential and non-essential language features form the basis of our formal framework of expressibility, we begin our investigation with some typical and widely accepted examples of "syntactic sugar." Consider a goto-free, Algol-like language that has a **while**-loop construct but lacks a **repeat**-loop. Clearly, few programmers would consider this a loss since the **repeat**-construct is roughly equivalent to the **while**-construct. More precisely, for all statements $s$ and expressions $e$

$$(\textbf{repeat}\ s\ \textbf{until}\ e)\ \textit{is expressible as}\ (s;\ \textbf{while}\ \neg e\ \textbf{do}\ s).$$

When an instance of the *expressed* (left-hand) side is needed, the appropriate instantiation of the *expressing* (right-hand) side will perform the same operations. Moreover, a *simple* preprocessor could translate a program with **repeat**-statements into a program with the equivalent **while**-programs.

In a dynamically-typed, functional language the **let**-expression is another prototypical example of "syntactic sugar." If a functional language has first-class procedures, the lexical declaration of a variable binding in the form of a **let**-expression is an abbreviation of the immediate application of an anonymous procedure to the initial value:

$$(\textbf{let}\ x\ \textbf{be}\ v\ \textbf{in}\ e)\ \textit{is expressible as}\ (\textbf{apply}\ (\textbf{procedure}\ x\ e)\ v).$$

Similarly, functional languages can also realize **if**-expressions and the truth values through functional combinators [1:133]. In a *lazy* setting, selector procedures can express

$$\textsf{true}\ \textit{as}\ (\textbf{procedure}\ (x\ y)\ x),$$

$$\textsf{false}\ \textit{as}\ (\textbf{procedure}\ (x\ y)\ y),$$

and, based on this, an **if**-construct as an application of the test expression to the two branches:

$$(\textbf{if}\ \textit{tst thn els})\ \textit{is expressible as}\ (\textbf{apply}\ \textit{tst thn els}).$$

Although the **let**- and **if**-examples are similar, they also reveal some of the typical vagueness in informal expressibility claims. For practical purposes, the implementations of **if**, **true**, and **false** suffice. If a program produces an answer, it is possible to replace the *expressed* phrases with the *expressing* construction without effect on the final result. But, if the subexpression *tst* of an **if**-expression does not evaluate to a boolean value, a built-in **if**-construct may signal an error or diverge whereas the *expressing* phrases may return a proper value. In short, the *expressing* phrases may yield results in more situations than the built-in, *expressed* constructs. Still, both expressibility statements are widely accepted claims and deserve consideration.

The essence of simple statements about "syntactic sugar" relationships is a set of three formal properties. First, the *expressing* phrase is only constructed with facilities in a restricted sublanguage. Second, it is constructed without analysis of the subphrases of the *expressed* phrase. Third, replacing the instances of an *expressed* phrase in a program by the corresponding instances of the *expressing* phrases has no effect on the behavior of terminating programs, but may transform a previously diverging program into a converging one. A formal framework of expressibility must account for these ideas with precise definitions.

For our purposes, a programming language is a set of syntactic phrases with a semantics. A program is a phrase whose behavior we can observe by submitting it to an evaluator, which may or may not produce an answer for a given program.

**Definition 2.1.** (*Programming Language*) A *programming language* $\mathcal{L}$ consists of

- a set of $\mathcal{L}$-phrases, which is a set of freely generated abstract syntax trees (or *terms*), based on a possibly infinite number of function symbols $\mathsf{F}, \mathsf{F}_1, \ldots$ with arity $a, a_1, \ldots$;
- a set of $\mathcal{L}$-programs, which is a non-empty subset of the set of phrases; and
- an operational semantics, which is a partial computable function, $eval_{\mathcal{L}}$, from the set of $\mathcal{L}$-programs to an unspecified set of $\mathcal{L}$-answers:

$$eval_{\mathcal{L}} : \mathcal{L}\text{-programs} \multimap \mathcal{L}\text{-answers}.$$

The function symbols, including the 0-ary symbols, are referred to as *programming constructs* or *facilities*.

**Note.** The set of phrases is (the universe of) a *many-sorted*, freely generated term algebra. For simplicity, we ignore the many-sortedness of the abstract syntax. Moreover, in our examples we often use concrete syntax for readability. ∎

Our prototypical example of a programming language is based on the language $\Lambda$ of the pure $\lambda$-calculus [1]. In order to compare the expressiveness of call-by-value and call-by-name procedures later in this section, we extend $\Lambda$ with a new constructor, $\lambda_v$, and rename $\lambda$ to $\lambda_n$. That is, the phrases are generated from a set of variables $\{x, y, z, \ldots\}$ (0-ary constructors) and three binary constructors: $\lambda_v$ : *variable* $\times$ *term* $\longrightarrow$ *term* (call-by-value abstraction), $\lambda_n$ : *variable* $\times$ *term* $\longrightarrow$ *term* (call-by-name abstraction) and $\cdot$ : *term* $\times$ *term* $\longrightarrow$ *term* (juxtaposition):

$$e ::= x \mid (\lambda_v x.e) \mid (\lambda_n x.e) \mid (ee)$$

where $e$ ranges over terms and $x$ over variables. The constructors $\lambda_v$ and $\lambda_n$ bind their variable arguments in their term arguments; if all variables in a $\Lambda$-term are bound, we say the term is closed. The set of $\Lambda$-programs is the set of closed terms. $\Lambda$-answers are $\lambda$-abstractions; we also refer to them as $\Lambda$-*values* and let $v$ range over this set. We adopt the usual $\lambda$-calculus conventions about the concrete syntax of $\Lambda$-terms [1].

The operational semantics of $\Lambda$ is based on the $\beta$- and $\beta_v$-reduction schemas and the standard reduction function [1, 16]. The reduction schemas denote relations on the term language. Their definitions are as follows:[1]

$$(\lambda_n x.e)e' \longrightarrow e[x/e'] \qquad (e' \text{ is arbitrary}) \qquad (\beta)$$

$$(\lambda_v x.e)v \longrightarrow e[x/v] \qquad (v \text{ is a value}). \qquad (\beta_v)$$

An evaluation proceeds by reducing the leftmost-outermost occurrences of reducible expressions (redexes) outside of abstractions until no more such redexes exist. More precisely, if the tree is a redex, the redex is contracted and the evaluation process starts over with the new program. Otherwise, if the left part of the application is a call-by-value abstraction, the search for a redex continues with the right part; if it is not, it concentrates on the left part. The evaluation process terminates after producing an answer, i.e., an abstraction. By summarizing the standard reduction process into a function from $\Lambda$-programs to $\Lambda$-answers, we obtain the operational evaluation function.

A sublanguage is a programming language without certain programming constructs. The programs in the sublanguage must have the same behavior as in the full language.

**Definition 2.2.** (*Sublanguage*) A programming language $\mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$ is a *sublanguage* of a language $\mathcal{L}$ if

- the $\mathsf{F}_i$'s are constructors of $\mathcal{L}$'s phrase language,

[1] $e_1[x/e_2]$ is $e_1$ with all free $x$ substituted by $e_2$, possibly with some of the bound variables in $e_1$ renamed to avoid name clashes. More formally, for the definition of the operational semantics we consider the quotient of $\Lambda$ under $\alpha$-equivalence.

- the set of $\mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$-phrases is the subset of $\mathcal{L}$-phrases that do not contain any constructs in $\{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$,
- the set of $\mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$-programs is the subset of $\mathcal{L}$-programs that do not contain any constructs in $\{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$, and
- the semantics of $\mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$ is a restriction of $\mathcal{L}$'s semantics, i.e., for all $\mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$-programs $e$, $eval_{\mathcal{L}\setminus\{\mathsf{F}_1,\ldots,\mathsf{F}_n\}}(e) = eval_{\mathcal{L}}(e)$.

We sometimes refer to $\mathcal{L}$ as a *language extension* of $\mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$.

Every language is obviously a sublanguage of itself. In our running example, the sublanguage $\Lambda_n$ is $\Lambda$ without $\lambda_v$-abstractions, $\Lambda_v$ is $\Lambda$ without call-by-name abstractions:

$$\Lambda_n = \Lambda \setminus \lambda_v; \quad \Lambda_v = \Lambda \setminus \lambda_n.$$

A restriction of the above evaluation process to $\Lambda_v$-terms and $\Lambda_n$-terms yields call-by-value and call-by-name semantics, respectively.

Next we can turn to the relationship between *expressed* and *expressing* phrases in an "is expressible as"-relation. The syntactic aspect of the relationship is that the abbreviated expression is only generated from a restricted set of syntactic constructors, and that the translations of the subexpressions of the *expressed* phrase occur as subexpressions of the *expressing* phrase. The restriction of a language to a sublanguage takes care of the former condition; the latter condition is satisfiable by considering *parameterized* phrases, i.e., phrases containing meta-variables, and instantiations of such phrases.

**Definition 2.3.** (*Syntactic Abstraction*) The set of *syntactic abstractions* over a programming language $\mathcal{L}$ is the set of freely generated trees based on $\mathcal{L}$'s constructors and an infinite number of additional 0-ary constructors called *meta-variables* $(\alpha, \alpha_1, \ldots)$. We denote syntactic abstractions with $\mathsf{M}(\alpha_1, \ldots, \alpha_n)$. If $\mathsf{M}(\alpha_1, \ldots, \alpha_n)$ is a syntactic abstraction and $e_1, \ldots, e_n$ are phrases in $\mathcal{L}$, then the *instance* $\mathsf{M}(e_1, \ldots, e_n)$ is a phrase in $\mathcal{L}$ that is like $\mathsf{M}(\alpha_1, \ldots, \alpha_n)$ except at occurrences of $\alpha_i$ where it contains the phrase $e_i$:

- if $\mathsf{M}(\alpha_1, \ldots, \alpha_n) = \alpha_i$ then $\mathsf{M}(e_1, \ldots, e_n) = e_i$, and
- if $\mathsf{M}(\alpha_1, \ldots, \alpha_n) = \mathsf{F}(\mathsf{M}_1(\alpha_1, \ldots, \alpha_n), \ldots, \mathsf{M}_a(\alpha_1, \ldots, \alpha_n))$ for some $\mathsf{F}$ with arity $a$ then $\mathsf{M}(e_1, \ldots, e_n) = \mathsf{F}(\mathsf{M}_1(e_1, \ldots, e_n), \ldots, \mathsf{M}_a(e_1, \ldots, e_n))$.

**Note.** Algebraically a syntactic abstraction $\mathsf{M}(\alpha_1, \ldots, \alpha_n)$ is a (sorted) *polynomial* over the free term algebra with the variables $\alpha_1, \ldots, \alpha_n$. An instantiation of a syntactic extension is an application of the function to appropriate arguments. In the terminology of equational algebraic specifications, syntactic abstractions are known as *derived operators* [6]. In Lisp-like languages, syntactic abstractions are realized as *macros* [9]; logical frameworks know them as *notational abbreviations* [7]. ∎

In the framework of our example language $\Lambda$,

$$\mathsf{M}(\alpha_1, \alpha_2, \alpha_3) = (\lambda_n \alpha_1.\alpha_2)\alpha_3$$

is a syntactic abstraction over $\Lambda$. The expression $(\lambda_n x.x)y$ is the instance $\mathsf{M}(x,x,y)$.

The semantic aspect of the relationship between *expressed* and *expressing* phrases is that the replacement of the former by the latter in arbitrary programs has no effect on the program behavior. To capture this idea, we need to formalize an expansion of a program from a language into a sublanguage according to a set of syntactic abstractions.

**Definition 2.4.** (*Syntactic Expansion*) Let $\mathcal{L}' = \mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$ be a sublanguage of $\mathcal{L}$. A *syntactic environment* $\rho$ (over $\mathcal{L}'$ for $\mathcal{L}$) is a finite map from the syntactic constructors $\mathsf{F}_1, \ldots, \mathsf{F}_n$ to syntactic abstractions. A *syntactic expansion* $[\![\cdot]\!]_\rho$ from $\mathcal{L}$ to $\mathcal{L}'$ relative to the syntactic environment $\rho$ maps $\mathcal{L}$-phrases to $\mathcal{L}'$-phrases as follows:

- if $\mathsf{F} \notin Dom(\rho)$ then $[\![\mathsf{F}(e_1,\ldots,e_a)]\!]_\rho = \mathsf{F}([\![e_1]\!]_\rho,\ldots,[\![e_a]\!]_\rho)$
- if $\rho(\mathsf{F}) = \mathsf{M}(\alpha_1,\ldots,\alpha_a)$ then $[\![\mathsf{F}(e_1,\ldots,e_a)]\!]_\rho = \mathsf{M}([\![e_1]\!]_\rho,\ldots,[\![e_a]\!]_\rho)$.

At this point, everything is in place for the definition of a formal notion of expressibility. A language can express a set of constructs if there are syntactic abstractions that can replace occurrences of the old constructs without effect on the program's behavior. Given this, we must only agree on the behavioral characteristics of programs that we would like to observe. In order to avoid overly restrictive assumptions about the set of programming languages, we follow a minimalistic approach and observe the termination behavior of programs.[2]

**Definition 2.5.** (*Expressibility*) Let $\mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$ be a sublanguage of $\mathcal{L}$ and let $\mathcal{L}$ be a sublanguage of $\mathcal{L}'$. The programming language $\mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$ *can express the syntactic facilities* $\{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$ *with respect to* $\mathcal{L}'$ if for every $\mathsf{F}_j$ there is a syntactic abstraction $\mathsf{M}_j$ such that for all $\mathcal{L}$-programs $p$,

$$eval_{\mathcal{L}}(p) \textit{ is defined} \text{ if and only if } eval_{\mathcal{L}}([\![p]\!]_\rho) \textit{ is defined}$$

where $\rho = \{(\mathsf{F}_j, \mathsf{M}_j) \mid 1 \le j \le n\}$.

$\mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$ *can* **weakly** *express the syntactic facilities* $\mathsf{F}_j$ *with respect to* $\mathcal{L}'$ if there are syntactic abstractions $\mathsf{M}_j$ such that for all $\mathcal{L}$-programs $p$,

$$eval_{\mathcal{L}}(p) \textit{ is defined} \text{ implies } eval_{\mathcal{L}}([\![p]\!]_\rho) \textit{ is defined}$$

where $\rho = \{(\mathsf{F}_j, \mathsf{M}_j) \mid 1 \le j \le n\}$.

The qualifying clause "with respect to" is omitted whenever the language universe is obvious from the context. If $\mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$ can express $\mathsf{F}$ because of the syntactic abstraction $\mathsf{M}$, we sometimes say that $\mathsf{M}$ expresses $\mathsf{F}$ when $\mathcal{L} \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$ and $\mathcal{L}'$ are understood.

The terminology "weakly expressible" reflects our belief that *any* differences in behavior should be noted as a failure of complete expressibility. The definition is also consistent with the fact that expressibility implies weak expressibility.

[2]This restriction only excludes *total* programming languages from further consideration. But, by omitting any references to the characteristics of results, it is possible to consider a broad variety of programming languages, e.g., languages with or without basic, observable data.

**Proposition 2.6** *If $\mathcal{L}$ can express* F, *then $\mathcal{L}$ can* **weakly** *express* F.

An alternative understanding of the expressibility relation is that the *expressing* phrase and the *expressed* phrase are interchangeable *in all programs.* This relation between phrases is widely studied in semantics and is known as *operational* or *observational equivalence* [12, 15, 16]. For a formal definition of this relation, we must first establish the auxiliary notion of a context.

**Definition 2.7.** (*Context*) An *$\mathcal{L}$-program context for a phrase* $e$ is a unary syntactic abstraction, $C(\alpha)$, such that $C(e)$ is an $\mathcal{L}$-program.

The syntactic abstraction

$$C_0(\alpha) \;=\; (\lambda_n xy.\alpha)(\lambda_n x.x)\Omega, \text{ where } \Omega = (\lambda_n x.xx)(\lambda_n x.xx)$$

is a program context for all expressions whose free variables are among $x$ and $y$.

In order to establish a relation to the above definition of expressibility, our definition of operational equivalence only compares the termination behavior of programs.

**Definition 2.8.** (*Operational Equivalence*) Let $\mathcal{L}$ be a programming language and let $eval_{\mathcal{L}}$ be its operational semantics. The $\mathcal{L}$-terms $e_1$ and $e_2$ are *operationally equivalent*, $e_1 \cong_{\mathcal{L}} e_2$, if there are $\mathcal{L}$-program contexts for both $e_1$ and $e_2$, and if for all such contexts $C(\alpha)$, $eval_{\mathcal{L}}$ is defined on $C(e_2)$ if and only if it is defined on $C(e_1)$.

With the above program context $C_0$, it is possible, for example, to differentiate the phrases $x$ and $y$. Since $\Omega$ diverges, the program $C_0(x) = (\lambda_n xy.x)(\lambda_n x.x)\Omega$ terminates whereas $C_0(y) = (\lambda_n xy.y)(\lambda_n x.x)\Omega$ diverges.

Based on the idea that an expressing phrase must be operationally equivalent to an expressed phrase, we can now establish our first major meta-theorem on expressibility: If a programming construct fundamentally alters the operational equivalence relation of the extended language, it is impossible to express the additional construct in the restricted sublanguage.

**Theorem 2.9** *Let $\mathcal{L}_0 = \mathcal{L}_1 \setminus \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$ be a sublanguage of $\mathcal{L}_1$, and let $\mathcal{L}_1$ be a sublanguage of $\mathcal{L}$. Let $\cong_0$ and $\cong_1$ be the operational equivalence relations, respectively.*

(*i*) *If the operational equivalence relation of $\mathcal{L}_1$ does not conservatively extend the operational equivalence relation of $\mathcal{L}_0$, i.e., $\cong_0 \;\not\subseteq\; \cong_1$, then $\mathcal{L}_0$ cannot express the facilities $\mathsf{F}_1, \ldots, \mathsf{F}_n$ with respect to $\mathcal{L}$.*

(*ii*) *The converse of* (*i*) *does not hold. That is, there are cases where $\mathcal{L}_0$ cannot express a facility* F *even though the operational equivalence relation of $\mathcal{L}_1$ is a conservative extension of the operational equivalence relation of $\mathcal{L}_0$, i.e. $\cong_0 \;\subseteq\; \cong_1$.*

**Proof.** (*i*) The non-conservativeness of the extension implies that there are two terms $e$ and $e'$ in $\mathcal{L}_0$ such that $e \cong_0 e'$ but $e \not\cong_1 e'$. Given the theorem's assumption, the operational equivalence relation for $\mathcal{L}_1$ can only distinguish $e$ from $e'$ through a context built with some $\mathsf{F} \in \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$. To prove this auxiliary claim, assume the contrary. Then there is a program context $C(\alpha)$ for $e$ and $e'$ over $\mathcal{L}_1$ such that $C(\alpha)$ does not contain any constructors in $\{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$, and such that (without loss of generality) $eval_1$ produces an answer for $C(e)$ but not for $C(e')$. Since neither $e$ nor $e'$ nor $C$ contain any constructor in $\{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$, $C(e)$ and $C(e')$ are $\mathcal{L}_0$-programs. By the definition of a sublanguage, $eval_0(C(e))$ is defined because $eval_1(C(e))$ is defined and $eval_0(C(e'))$ is undefined because $eval_1(C(e'))$ is undefined. Consequently, $eval_0$ is defined on $C(e)$ but not on $C(e')$, which contradicts the assumption that $e \cong_0 e'$. We have thus proved the auxiliary claim.

Now, assume contrary to the claim in the theorem that $\mathcal{L}_0$ can express $\mathsf{F}_1, \ldots, \mathsf{F}_n$. Hence, there is a syntactic environment $\rho$ that maps each $\mathsf{F}_i$ to some syntactic abstraction $\mathsf{M}_i$. Let $C(\alpha)$ be a context over $\mathcal{L}_1$ that can differentiate the two terms $e$ and $e'$ based on some $\mathsf{F}, \ldots \in \{\mathsf{F}_1, \ldots, \mathsf{F}_n\}$. Let us say that $eval_1$ produces an answer for $C(e)$ but not for $C(e')$. By the definition of expressibility, $C(e)$ and $C(e')$ have counterparts in $\mathcal{L}_0$, $p = [\![C(e)]\!]_\rho$ and $p' = [\![C(e')]\!]_\rho$, that have the same termination behavior:

$$eval_1(p) \text{ is defined because } eval_1(C(e)) \text{ is defined}$$

and

$$eval_1(p') \text{ is undefined because } eval_1(C(e')) \text{ is undefined.}$$

By construction, the programs $p$ and $p'$ can only differ in a finite number of occurrences of $e$ and $e'$, respectively. From this fact and the assumption that $e \cong_0 e'$, we can immediately derive that $p$ and $p'$ have the same termination behavior in $\mathcal{L}_0$, i.e.,

$$eval_0(p) \text{ is defined if and only if } eval_0(p') \text{ is defined.}$$

But this implies that

$$eval_1(p) \text{ is defined if and only if } eval_1(p') \text{ is defined}$$

because $\mathcal{L}_0$ is a sublanguage of $\mathcal{L}_1$. Since this conclusion contradicts the above fact that $p$ converges and $p'$ diverges, we have proved our claim.

(*ii*) A simple example can be constructed by merging two disjoint languages such that a program in the combined language is either in one or the other language. The extension of the operational equivalence relation is conservative and the features in the added language are in general not expressible.

One particular example of this kind is the simply typed $\lambda$-calculus, whose types are either base types or arrow types. Because of the type system, it is impossible to define the typical *cons*, *car*, and *cdr* functions for pairs of elements of arbitrary types. Hence this simply-typed language cannot *express* these pairing functions. On the other hand, also due to the type system of the language, the new functions cannot be bound to free variables in phrases of the sublanguage, which implies that the pairing functions cannot be used to distinguish phrases in the simply typed language. It follows that pairing type constructors and functions increase the expressive power without destroying operational equivalences of the underlying language. ∎

Based on this first meta-theorem on expressibility, we can now easily show that the sublanguage $\Lambda_v$ is not strong enough to express call-by-name abstraction, and that $\Lambda_n$ is not strong enough to express call-by-value abstraction. However, $\Lambda_n$ can **weakly** express $\lambda_v$ since call-by-name abstractions are applicable to both proper values and undefined expressions.

**Proposition 2.10** *$\Lambda$ extends both $\Lambda_v$ and $\Lambda_n$.*

(*i*) *$\Lambda_v$ cannot express $\lambda_n$ with respect to $\Lambda$.*

(*ii*) *$\Lambda_n$ cannot express $\lambda_v$ with respect to $\Lambda$.*

(*iii*) *$\Lambda_n$ can* **weakly** *express $\lambda_v$ with respect to $\Lambda$.*

**Proof.** (*i*) Consider the expressions $\lambda_v f.f(\lambda_v x.x)\Omega$ and $\lambda_v f.\Omega$ where $\Omega$ is the call-by-value variant of our prototypical infinite loop. In the pure call-by-value setting, the two are operationally equivalent

$$\lambda_v f.f(\lambda_v x.x)\Omega \cong_{\Lambda_v} \lambda_v f.\Omega.$$

Both abstractions are values; upon application to an arbitrary value, both of them diverge. But, in the extended language $\Lambda$, we can differentiate the two with the context

$$C(\alpha) = \alpha(\lambda_v x.(\lambda_n y.x)).$$

The context applies a phrase to a function that returns the value of the first argument after absorbing the second argument without evaluating it. Hence, $C(\lambda_v f.f(\lambda_v x.x)\Omega)$ terminates while $C(\lambda_v f.\Omega)$ diverges, and the extension of $\Lambda_v$ to $\Lambda$ is non-conservative with respect to operational equivalence. By Theorem 2.9, $\lambda_n$ is not expressible.

(*ii*) There are two proofs for part (*ii*). The first is another application of the preceding meta-theorem and is derived from a closely related theorem by Ong [13: Thm. 4.1.1].[3] The second—related to but discovered independently of the first—provides some operational understanding for the reasons why $\Lambda_n$ cannot express $\lambda_v$-abstractions. We only present the second one.

A simple consequence of the operational semantics for $\Lambda_n$ is that a subterm $e$ of a program $C(e)$ can only affect the evaluation if it occurs in a leftmost-outermost position in some term of the evaluation sequence, *i.e.*,

$$C(e) \longrightarrow^* eq_1 \dots q_n,$$

for some arbitrary $q_1, \dots, q_n$, $n \geq 0$ [15]. When this happens, we say the term becomes *active*.

Suppose a syntactic abstraction $\mathbf{S}(x, e_1)$ over $\Lambda_n$ could express $\lambda_v x.e_1$. To show that this is impossible, we prove that $\mathbf{S}(x, e_1)e_2$ cannot be operationally equivalent to $(\lambda_v x.e_1)e_2$

[3]Gordon Plotkin pointed out Abramsky's and Ong's work on the lazy $\lambda$-calculus, which provided the ideas for the first proof and corrected a mistake in an early draft of this report.

for arbitrary $e_1$ and $e_2$. Clearly, $\mathsf{S}(x, e_1)$ must be a value (or must reduce to a value). Let $\lambda_n y.p$ be this value. It follows that

$$\mathsf{S}(x, e_1)e_2 \longrightarrow^* (\lambda_n y.p)e_2 \longrightarrow p[y/e_2].$$

Since $\mathsf{S}$ is defined without knowledge of $e_1$ and $e_2$, the evaluation of $p[y/e_2]$ proceeds independently of $e_2$ until $e_2$ becomes active. Consequently, we must distinguish two cases. First, the evaluation of $p[y/e_2]$ may never activate $e_2$. This is impossible because $e_2$ in the call-by-value program is evaluated and can thus cause divergence. Second, the evaluation of $p[y/e_2]$ activates $e_2$ for a first time, that is,

$$p[y/e_2] \longrightarrow^* e_2 q_1 \dots q_n.$$

Since the evaluation was thus far independent of $e_2$, $e_2$ is arbitrary and must hold for an arbitrary choice. So, let

$$e_2 = \mathbf{YK} = (\lambda_n f.(\lambda_n x.f(xx))(\lambda_n x.f(xx)))(\lambda_n xy.x).$$

But then the reduction sequence for the expression $\mathsf{S}(x, e_1)e_2$ ends in

$$\mathbf{YK}q_1 \dots q_n \longrightarrow^+ \lambda_n d.\mathbf{YK},$$

which is independent of whether $x$ is free in $e_1$ or not. If $x$ is not free in $e_1$, however, the program $(\lambda_v x.e_1)(\mathbf{YK})$ reduces to the evaluation of $e_1$:

$$(\lambda_v x.e_1)(\mathbf{YK}) \longrightarrow (\lambda_v x.e_1)(\lambda_n d.(\mathbf{YK})) \longrightarrow e_1.$$

Substituting the diverging term $\Omega$ for $e_1$ leads to the desired contradiction. Since there are no other cases, this proves the second claim.

(*iii*) Clearly, $\lambda_v x.e$ operationally approximates $\lambda_n x.e$ and, hence, $\mathsf{M} = \lambda_n \alpha_1.\alpha_2$ can **weakly** express $\lambda_v$ . ∎

Proposition 2.10 provides two examples of pairs of *universal* programming languages that we can differentiate according to our expressiveness criterion. The third part of this proposition also shows that the above meta-theorem on expressibility does *not* carry over to weak expressibility. Even in the case of a language extension that is non-conservative with respect to the operational equivalence or approximation relation, the restricted language may already be able to express the new facilities in a weak sense. Call-by-value procedures and $\Lambda_n$ provide the prototypical example.

**Theorem 2.9 (cont'd)** *Let $\mathcal{L}_0$, $\mathcal{L}_1$, $\cong_0$ and $\cong_1$ be as above.*

(*iii*) *Part (i) does not hold for* **weak** *expressibility. That is, there are cases where $\mathcal{L}_0$ can* **weakly** *express $\mathsf{F}_1, \dots, \mathsf{F}_n$ relative to $\mathcal{L}_1$ even though the extended operational approximation relation does not subsume the restricted one, i.e., $\cong_0 \not\subseteq \cong_1$.*

The definition of an expressibility relationship also leads to a natural measure of expressive power between programming languages. A programming language is less expressive than another if the latter can express all the facilities the former can express in a given language universe.

**Definition 2.11.** (*Expressiveness*) Let $\mathcal{L}_0$ and $\mathcal{L}_1$ be sublanguages of $\mathcal{L}$. The language $\mathcal{L}_0$ *is less expressive than* $\mathcal{L}_1$ *with respect to* $\mathcal{L}$ if $\mathcal{L}_1$ can express all sets of facilities with respect to $\mathcal{L}$ that $\mathcal{L}_0$ can express with respect to $\mathcal{L}$.

Expressiveness is a pre-order on sublanguages in a given language framework.

**Theorem 2.12** *The less-expressive-than relation is a pre-order in its first two arguments.*

(*i*) *$\mathcal{L}_0$ is less expressive than $\mathcal{L}_0$ with respect to to $\mathcal{L}$.*

(*ii*) *If $\mathcal{L}_0$ is less expressive than $\mathcal{L}_1$ with respect to $\mathcal{L}$ and $\mathcal{L}_1$ is less expressive than $\mathcal{L}_2$ with respect to $\mathcal{L}$, then $\mathcal{L}_0$ is less expressive than $\mathcal{L}_2$ with respect to $\mathcal{L}$.*

Based on the definitions of expressibility and expressiveness, we can now analyze the expressiveness hierarchy in more practical languages. In the next section, we study an idealized version of Scheme as a concrete example of this kind, thus providing further insight into our general framework.

Syntax

$$\begin{array}{lcll}
v & ::= & 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots & (\textit{numerals}) \\
 & & \mid \mathsf{zero?} \mid \mathsf{add_1} \mid \mathsf{sub_1} \mid + \mid - & (\textit{numeric functions}) \\
 & & \mid (\textbf{lambda}\ (x \ldots)\ e) & (\textit{abstractions}) \\
e & ::= & v & (\textit{values}) \\
 & & \mid x & (\textit{variables}) \\
 & & \mid (e\ e \ldots) & (\textit{applications})
\end{array}$$

Semantics

$$\begin{array}{rcl}
(f a \ldots) & \longrightarrow & \delta(f, a, \ldots) \quad \text{for } \mathsf{zero?}, \mathsf{add_1}, \ldots \\
((\textbf{lambda}\ (x_1 \ldots x_n)\ e)\ v_1 \ldots v_n) & \longrightarrow & e[x_1/v_1] \ldots [x_n/v_n]
\end{array}$$

Constant Interpretation

$$\begin{array}{rclrclrcl}
\delta(\mathsf{add_1}, n) & = & n+1 & \delta(+, n, m) & = & n+m & \delta(\mathsf{zero?}, 0) & = & (\textbf{lambda}\ (x\ y)\ x) \\
\delta(\mathsf{sub_1}, n) & = & n-1 & \delta(-, n, m) & = & n-m & \delta(\mathsf{zero?}, n) & = & (\textbf{lambda}\ (x\ y)\ y) \\
 & & & & & & & & \text{for } n \neq 0
\end{array}$$

FIGURE 1: *Pure Scheme*

## 3 The Structure of *Idealized Scheme*

*Pure Scheme* [3, 4, 5] is an extension of the simple call-by-value language $\Lambda_v$ that includes multi-ary procedures and algebraic constants. There are basic constants and functional constants; the latter operate on constants and closed abstractions. We assume that the semantics of constants is given through an interpretation $\delta$ from functional constants and closed values to closed values. Typically, the constants include integers, characters, booleans, and some appropriate functions. Initially, we only include integers and a minimal set of functions. Figure 1 contains the complete specification of *Pure Scheme*. The semantics is given via reduction rules, which are applied in the standard reduction order defined in the preceding section; the extended evaluation function is undefined for a program when the evaluation of the program gets stuck because of the application of a constant symbol to a $\lambda$-expression, the application of a numeral to a value, or the application of a constant function to a value for which $\delta$ is undefined.

The main characteristic of *Idealized Scheme* is the extension of the functional core language *Pure Scheme* with imperative facilities and type predicates:

- predicate constants for determining the type of a value,
- branching expressions for the local manipulation of control,
- control expressions for the non-local manipulation of control, and
- assignment statements for the manipulation of state variables.

The extensions are motivated by the belief that imperative facilities and type predicates add to the expressive power of the language [19, 20]. In this section, we demonstrate how to formulate these beliefs in our formal expressiveness framework and how to relate the extensions to the core language.

The addition of type predicates to *Pure Scheme* is simple. For extending *Pure Scheme* with a predicate like int?, it suffices to extend the interpretation function $\delta$ with the clauses

$$\begin{aligned}
\delta(\mathsf{int?}, n) &= (\textbf{lambda}\ (x\ y)\ x) \\
\delta(\mathsf{int?}, (\textbf{lambda}\ (x \ldots)\ e)) &= (\textbf{lambda}\ (x\ y)\ y)
\end{aligned}$$

We refer to the extended language as *PS*(int?). With int?, programs in the extended language can now effectively test the type of a value, which is impossible in *Pure Scheme*.

**Theorem 3.1** *Pure Scheme cannot express* int? *with respect to PS*(int?).

**Proof Sketch.** The claim is another consequence of Theorem 2.9. In *Pure Scheme*, we have:

```
(((p 1) (lambda ()
          (((p (lambda () Ω))
            (lambda () 2)                    (((p 1) (lambda ()
            (lambda ()                                  (((p (lambda () Ω))
              (((p 2)                 ≅_ps                (lambda () 2)
                (lambda () 1)                             (lambda () Ω))))
                (lambda () Ω)))))))                   (lambda () Ω)))
        (lambda () Ω)))
```

In the extended language, however, $p$ could be bound to the predicate int?, which would distinguish the two expressions operationally. ∎

The programming language world knows two types of local branching statements for languages like *Pure Scheme*: the truth-value based **if**-construct and the Lisp-style **if**-construct that distinguishes one special value from all others. The semantics of the former relies on the presence of two distinct values: true and false, or 0 and 1. The following two reduction rules characterize the behavior of truth-value based **if**:

$$(\textbf{if}\ 0\ e_t\ e_f) \longrightarrow e_t \qquad (\textbf{if}.\mathsf{true})$$
$$(\textbf{if}\ 1\ e_t\ e_f) \longrightarrow e_f. \qquad (\textbf{if}.\mathsf{false})$$

If the test value in an **if**-expression is neither 0 nor 1, the evaluation of the program is undefined. We refer to the extended language as *PS*(**if**).

Clearly, *Pure Scheme* can express such a simple **if**.

**Theorem 3.2** *Pure Scheme can express truth-value based* **if** *with respect to PS*(**if**).

**Proof Sketch.** Consider the syntactic abstraction:

$$\begin{aligned}(\textbf{if}\ e\ e_1\ e_2) \;=\; & (((\mathsf{zero?}\ e)\\ & (\textbf{lambda}\ ()\ e_1)\\ & (\textbf{lambda}\ ()\ (((\mathsf{zero?}\ (\mathsf{sub}_1\ e))\ (\textbf{lambda}\ ()\ e_2)\ (\textbf{lambda}\ ()\ \Omega))))))\end{aligned}$$

It is easy to show that this abstraction is operationally equivalent to **if**. ∎

**Note.** (*i*) The trick of *freezing* the evaluation of expressions through vacuous **lambda**-abstraction was already known to Landin and Burge [10], but, clearly, it is impossible to express typed **if** without 0-ary abstraction in a call-by-value framework.

(*ii*) The preceding theorem crucially depends on the specification that values other than 0 and 1 in the test position of the **if** construct render the program undefined. ∎

The Lisp-style **if** assumes that there is *one* distinct value for *false*, in Lisp usually called nil, and *all other values* represent *true*. With 0 serving as nil, the reduction rules differ accordingly from (**if**.true) and (**if**.false):

$$(\textbf{if}\ v\ e_t\ e_f) \longrightarrow e_t \text{ for } v \neq 0 \qquad (\textbf{if}.v)$$
$$(\textbf{if}\ 0\ e_t\ e_f) \longrightarrow e_f. \qquad (\textbf{if}.\mathsf{nil})$$

**Theorem 3.2 (cont'd)** *Pure Scheme cannot express* **if** *with respect to PS*(**Lisp-if**).

**Proof Sketch.** Obviously Lisp-style **if** is equivalent—in our formal sense—to the addition of a **total** functional constant null? for distinguishing 0 from all other values. By Theorem 3.1 it follows that Lisp-if is not expressible. ∎

A more interesting question about expressiveness arises in the context of non-local control abstractions. *Idealized Scheme* has the operation call-with-current-continuation, abbreviated **call/cc**, which applies its sub-expression to an abstraction of the current control state. In analogy to denotational semantics, the Scheme-terminology for this abstraction is *continuation*. Figure 2 specifies the syntax and a simple reduction semantics of *PS*(**call/cc**), an extended version of *Pure Scheme* with this control facility. The reduction semantics forms the basis of a simple equational calculus for **call/cc** and continuations, and permits a simple, algebra-like reasoning in the presence of control operations [4, 5].

Additional Syntax

$$e \ ::= \ \ldots \mid (\textbf{call/cc}\ e) \qquad (\textit{continuation captures})$$

Evaluation Contexts

$$E ::= \alpha \mid (v\ E) \mid (E\ e)$$

Additional Semantics

$$\begin{aligned}
E[(\textbf{call/cc}\ e)] &\longrightarrow (\textbf{call/cc}\ (\textbf{lambda}\ (k) \\
&\qquad E[e\ (\textbf{lambda}\ (x)\ (k\ E[x]))]))\\
(\textbf{call/cc}\ (\textbf{lambda}\ (k)\ E[(k\ e)])) &\longrightarrow (\textbf{call/cc}\ (\textbf{lambda}\ (k)\ e))\\
(\textbf{call/cc}\ (\textbf{lambda}\ (k)\ (\textbf{call/cc}\ e))) &\longrightarrow (\textbf{call/cc}\ (\textbf{lambda}\ (k)\ (e\ k)))\\
(\textbf{call/cc}\ (\textbf{lambda}\ (k)\ e)) &\longrightarrow e, \text{ if } k \notin e
\end{aligned}$$

FIGURE 2: *Pure Scheme* with control

**Theorem 3.3** *Pure Scheme cannot express* **call/cc** *relative to PS*(**call/cc**).

**Proof Sketch.** The theorem is a consequence of Theorem 2.9, i.e., the addition of **call/cc** invalidates operational equivalences in the extended language. A typical example[4] is the operational equivalence

$$(\textbf{lambda}\ (f)\ ((f\ 0)\ \Omega)) \cong_{ps} (\textbf{lambda}\ (f)\ \Omega).$$

As pointed out in the proof of Proposition 2.10($i$), these two procedures are equivalent in a functional setting because both diverge when applied to a value. When we add **call/cc**, however, we can construct the context (**call/cc** $\alpha$), which can differentiate between the two expressions. Whereas the composition of the first expression with this context evaluates to 0:

$$(\textbf{call/cc}\ (\textbf{lambda}\ (f)\ ((f\ 0)\ \Omega))) = (\textbf{call/cc}\ (\textbf{lambda}\ (f)\ 0)) = 0$$

the second expression diverges in the same context:

$$(\textbf{call/cc}\ (\textbf{lambda}\ (f)\ \Omega)) \ = \ \Omega. \qquad \blacksquare$$

The final addition to *Pure Scheme* is the **set!**-construct, Scheme's form of assignment statement. Like in a traditional Algol-like programming language, the **set!**-expression destructively alters a binding of an identifier to a value. A simple reduction semantics for *PS*(**set!**), *Pure Scheme* with **set!** and **letrec** (for recursive declarations of assignable variables with initial values), is given in Figure 3. Again, this semantics is the basis for an equational calculus for reasoning about operational equivalences in *PS*(**set!**) [3, 4].

[4]This example is a folklore example in the theoretical "continuation" community, but it was also used by Meyer and Riecke to argue the "unreasonableness" of continuations [11].

Additional Syntax

$$e \ ::= \ \ldots \mid (\textbf{set!}\ x\ e) \qquad (\textit{assignments})$$
$$\mid (\textbf{letrec}\ ([x\ v]\ldots)\ e) \qquad (\textit{recursive definitions})$$

Evaluation Contexts

$$E ::= \alpha \mid (v\ E) \mid (E\ e) \mid (\textbf{set!}\ x\ E)$$

Additional Semantics

$$\begin{aligned}
((\textbf{lambda}\ (x\ldots)\ e)\ v\ldots) &\longrightarrow (\textbf{letrec}\ ([x\ v]\ldots)\ e), \text{ instead of } \beta_v \\
(\textbf{letrec}\ (\ldots[x\ v]\ldots)\ e) &\longrightarrow (\textbf{letrec}\ (\ldots[x\ v]\ldots)\ e[x/v]) \\
&\quad\ \text{if } x \text{ is not assignable in } e \\
&\quad\ \text{and doesn't occur in the defined values} \\
(\textbf{letrec}\ (\ldots[x\ v]\ldots)\ E[x]) &\longrightarrow (\textbf{letrec}\ (\ldots[x\ v]\ldots)\ E[v]) \\
(\textbf{letrec}\ (\ldots[x\ u]\ldots)\ E[\textbf{set!}\ x\ v]) &\longrightarrow (\textbf{letrec}\ (\ldots[x\ v]\ldots)\ E[v]) \\
(\textbf{letrec}\ ([x\ v]\ldots)\ E[(\textbf{letrec}\ ([y\ u]\ldots)\ e)]) &\longrightarrow (\textbf{letrec}\ ([x\ v]\ldots[y\ u]\ldots)\ E[e])
\end{aligned}$$

FIGURE 3: *Pure Scheme* with state

**Theorem 3.4** *Pure Scheme cannot express* **set!** *with respect to PS*(**set!**).

**Proof Sketch.** Consider the expression

$$((\textbf{lambda}\ (d)\ (f\ 0))\ (f\ 0)),$$

which contains the same subexpression twice. Clearly, in a functional language like *Pure Scheme* the two subexpressions $(f\ 0)$ return the same value, if any, and, given that the value of the first subexpression is discarded, the expression is operationally equivalent to

$$(f\ 0).$$

In the extended language, this is no longer true. Consider the context

$$C(\alpha) = (\textbf{letrec}\ (f\ (\textbf{lambda}\ (x)\ (\textbf{set!}\ f\ (\textbf{lambda}\ (x)\ \Omega))))\ \alpha),$$

which declares a procedure $f$. Upon the first application, the procedure modifies the declaration so that a second invocation leads to divergence. Consequently, an expression with a single use of the function converges, but an expression with two uses diverges. The verification of these claims with the reduction semantics is straightforward. ∎

At this point we do not know whether the above results hold in the framework of weak expressiveness. We conjecture that they carry over but lack a proof of this statement.

## 4 Related Work

Kleene's study of definitional extensions in mathematical logic [8, 21] is the most closely related work. A formal system $S$ is a definitional extension of a formal system $S'$ if, roughly, the (provable) formulas of $S$ can be translated into (provable) formulas of $S'$ with a map that is homomorphic with respect to the logical operations and the constructors in $S'$. This is clearly analogous to our notion of expressibility when we replace provability with termination. Thus, the concept of an expressible programming construct directly corresponds to Kleene's notion of eliminable symbols and postulates [8:§74; 21:I.2].

Williams [22] considers a whole spectrum of formalization techniques for semantic conventions in formal systems. His work starts with ideas of applicative and definitional extensions of logics but also considers techniques that are more relevant in computational settings, e.g. compilation and interpretation. The goal of Williams's research is a comparison of the formalization techniques and not a study of the expressiveness of programming languages. Some of his results may be relevant for future extensions of our work.

Comparative schematology was an early but quite different attempt at measuring the relative expressive power of programming languages: see Chandra and Manna's report [2] for an example. Schematology studies programming languages with a fixed operational semantics for a simple set of control constructs, e.g. **while**-loop programs or recursion equations, and with uninterpreted constant and function symbols. In the absence of arithmetic, it is possible to decide certain questions about such uninterpreted program schemas. Moreover, it makes sense to compare the set of functions that are computable when data structures like stacks, arrays, queues, or general equality are added. In the presence of arithmetic, the approach can no longer compare the expressive power of programming languages since everything can be encoded. Also, given that many modern languages contain higher-order (procedural) data for control structure purposes, we find this approach unsuited for the comparison of realistic languages.

A secondary piece of related work is the study of the full abstraction property of mathematical models [13, 15] and the representability of functions in $\lambda$-calculi [1]. In many cases, the natural denotational model of a programming language contains too many elements so that operationally equivalent phrases have different mathematical meanings. Since it is relatively easy to reverse-engineer a programming language from a model, the equality relation of models without the full abstraction property directly corresponds to the operational equivalence of a language extension. As a consequence, such models naturally lead to the discovery of non-expressible programming constructs. Still, the study of full abstraction does not provide true insight into the expressive power of languages. On one hand, the discovery of new facilities directly depends on the choice of model. For example, whereas a direct model of $\Lambda_n$ requires a facility for exploiting deterministic parallelism, a continuation model leads to operations on the continuation of an expression (in addition to parallel constructs). On the other hand, by Theorem 2.9 we also know that a non-conservative extension of the operational equivalence relation is only a *sufficient* but not a *necessary* condition for the non-expressibility of a programming construct. In short, research on full abstraction is a valuable contribution to—see Proposition 2.10—but not a replacement of the study of expressiveness.

## 5 Towards a Formal Programming Language Design Space

In the preceding sections we have developed a formal framework for comparing the expressive power of programming languages. An analysis of its properties and an application of the framework to an idealized version of Scheme have demonstrated how close the formal notions are to the intuitive ideas in the literature. In particular, we have shown that an increase in expressive power may destroy semantic properties of the core language (Theorem 2.9), but we also conjecture that more expressive languages make programs more concise.

The current framework is only a first step towards a formal programming language design space. The crucial idea behind our development is the restriction of the translation process from an extended language to a smaller language. We believe that there may be an entire spectrum of feasible restrictions that yield interesting, alternative notions of expressiveness, and that these alternatives deserve exploration. Moreover, we have not yet tackled the problem of deriving properties from expressiveness claims but expect to do so in the future. In the long run, we hope that some theory of language expressiveness develops into a formal theory of the programming language design space.

**Acknowledgement.** Dan Friedman directed my attention to the idea of expressiveness by insisting that an understanding of new programming constructs in terms of procedures or macro implementations is superior to an implementation based on interpreters. Conversations with Bruce Duba and Mitchell Wand clarified my understanding of the problem. Bob Harper pointed out the relationship to logic, which ultimately led to the current formalization. Hans Boehm, Robert Cartwright, Dan Friedman, Robert Hieb, John Lamping, and Mitchell Wand suggested many improvements in the presentation of the material. Finally, comments by members of the POPL'88 committee, and by the anonymous referees of ESOP'90 pointed out weaknesses in previous drafts.

## 6 References

1. BARENDREGT, H.P. *The Lambda Calculus: Its Syntax and Semantics.* Revised edition. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, 1984.

2. CHANDRA, A.K. AND Z. MANNA. The power of programming features. *Journal of Computer Languages* (Pergamon Press) **1**, 1975, 219–232.

3. FELLEISEN, M. AND D.P. FRIEDMAN. A syntactic theory of sequential state. *Theor. Comput. Sci.* **69**(3), 1989, 243–287. Preliminary version in: *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987, 314-325.

4. FELLEISEN, M. AND R. HIEB. The revised report on the syntactic theories of sequential control and state. Technical Report 100, Rice University, June 1989.

5. FELLEISEN, M., D.P. FRIEDMAN, E. KOHLBECKER, AND B. DUBA. A syntactic theory of sequential control. *Theor. Comput. Sci.* **52**(3), 1987, 205–237. Preliminary version in: *Proc. Symposium on Logic in Computer Science*, 1986, 131–141.

6. Goguen, J., J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In *Current Trends in Programming Methodology* IV, edited by R. Yeh. Prentice-Hall, Englewood Cliffs, New Jersey, 1979, 80–149.
7. Griffin, T. Notational definition—A formal account. In *Proc. Symposium on Logic in Computer Science*, 1988, 372–383.
8. Kleene, S. C. *Introduction to Metamathematics*, Van Nostrand, New York, 1952.
9. Kohlbecker, E. *Syntactic Extensions in the Programming Language Lisp.* Ph.D. dissertation, Indiana University, 1986.
10. Landin, P.J. The next 700 programming languages. *Commun. ACM* **9**(3), 1966, 157–166.
11. Meyer, A.R. and J.R. Riecke. Continuations may be unreasonable. In *Proc. 1988 Conference on Lisp and Functional Programming*, 1988, 63–71.
12. Morris, J.H. *Lambda-Calculus Models of Programming Languages.* Ph.D. dissertation, MIT, 1968.
13. Ong, L C.-H. Fully abstract models of the lazy lambda-calculus. In *Proc. 29th Symposium on Foundation of Computer Science*, 1988, 368–376.
14. Paterson, M.S. and C.E. Hewitt. Comparative schematology. In *Conf. Rec. ACM Conference on Concurrent Systems and Parallel Computation*, 1970, 119–127.
15. Plotkin, G.D. LCF considered as a programming language. *Theor. Comput. Sci.* **5**, 1977, 223–255.
16. Plotkin, G.D. Call-by-name, call-by-value, and the $\lambda$-calculus. *Theor. Comput. Sci.* **1**, 1975, 125–159.
17. Reynolds, J.C. GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept. *Commun. ACM* **13**(5), 1970, 308–319.
18. Reynolds, J.C. The essence of Algol. In *Algorithmic Languages*, edited by de Bakker and van Vliet. North-Holland, Amsterdam, 1981, 345–372.
19. Steele, G.L., Jr. and G.J. Sussman. Lambda: The ultimate imperative. Memo 353, MIT AI Lab, 1976.
20. Sussman, G.J. and G.L. Steele Jr. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.
21. Troelstra, A. S. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* Lecture Notes in Mathematics 344. Springer-Verlag, Berlin, 1973.
22. Williams, J.G. On the formalization of semantic conventions. Draft version: September 1988. To appear in *Journal of Symbolic logic*, 1990.