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 validate the theory, we analyze some widely held beliefs about 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 foundation for further research in this direction.

1 Comparing Programming Languages

The literature on programming languages contains an abundance of informal claims on the expressive power of programming languages. Arguments in these contexts typically assert the expressibility or non-expressibility of programming constructs relative to a language. Unfortunately, programming language theory does not provide a formal framework for specifying and verifying such statements. Comparing the set of computable functions that a language can represent is useless because the languages in question are usually universal; other measures do not exist. The lack of a comparison relation makes it impossible to draw any firm conclusions from expressiveness claims or to use them for an objective decision about the use of a programming language.

*Supported in part by NSF grant CCR 89-17022 and Darpa/NSF grant CCR 87-20277.

Landin [24] was the first to propose the development of a formal framework for comparing programming languages. He studied the relationship between programming languages and constructs, and began to classify some as "essential" and some as "syntactic sugar." A typical example of an inessential construct in Landin's sense is the **let**-expression in a functional language with first-class procedures. It declares and initializes a new, lexically-scoped variable before evaluating an expression. Whether it is present or absent is inconsequential for a programmer since

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

Similarly, few programmers would consider it a loss if a goto-free, Algol-like language had a **while** but not a **repeat** construct. After all,

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

Others, most notably Reynolds [36, 37] and Steele and Sussman [40], followed Landin's example. They introduced the informal notion of the core of a language and studied the expressiveness of imperative extensions of higher-order functional languages. Steele and Sussman [40:29] summarized the crucial idea behind this kind of classification of language features with the remark that a number of programming constructs are expressible in an applicative notation based on syntactically local, structure- and behavior-preserving translations, but that some, notably control statements and assignments, involve complex reformulations of large fractions of programs.

In the realm of logic, Kleene anticipated the idea of expressible or *eliminable* syntactic symbols in his study of formal systems [21:§74]. Troelstra [42:I.2] resumed this work and introduced further refinements and extensions. Roughly, the additional symbols of a conservative extension of a core logic are eliminable if there is a translation from the extended logic to its core that satisfies a number of conditions. Two of these are important for our purposes. First, the mapping is the identity on the formulae of the core language and is homomorphic in the logical connectors. Second, if a formula is provable in the extension then so is its translation in the core. Clearly, these two conditions imply that this translation preserves the structure of formulae and removes symbols on a local basis.

By adapting the ideas about the relationship among formal systems to programming languages, we obtain a relation that determines whether a programming language can express a programming construct. More precisely, given two universal programming languages that only differ by a set of programming constructs, $\{c_1, \ldots, c_n\}$, the relation holds if the additional constructs make the larger language more expressive than the smaller one. Here "more expressive" means that the translation of a program with occurrences of one of the constructs c_i to the smaller language requires a global reorganization of the entire program. A first analysis shows that this measure of expressiveness supports many informal judgements in the literature. Moreover, we discover that an increase in expressive power comes at the expense of less "intuitive" semantic equivalence relations. We also discuss some attempts at generalizing the measure to a comparison relation for arbitrary programming languages.

The next section briefly reviews the logical notions of eliminable symbols and definitional extensions. In subsequent sections, we propose a formal model of expressibility and expressiveness along the lines of logical expressiveness, investigate some of its properties, and analyze the expressive powers of several extensions of functional languages. More specifically, we introduce our formal framework of expressiveness based on the notion of expressibility. We demonstrate the abstract concepts by proving some sample theorems about λ-calculus-based languages as well as a number of meta-theorems. Next, we study the expressiveness of an idealized version of Scheme and verify the informal expressiveness philosophy behind its design [41]. Following this analysis, we briefly speculate how the use of a more expressive language increases programming convenience. Finally, we compare our ideas to related work and address some open questions.

2 Eliminable Symbols and Definitional Extensions

The theory of comparing formal systems is a peripheral topic in logical studies and finds little or no space in most textbooks. The following short overview summarizes and adapts Troelstra's [42:I.2] descriptions of Kleene's work [21].

A *formal system* is a triple of sets: *expressions*, *formulae*, and *theorems*. The second is a subset of the first, the third a subset of the second. Expressions are freely generated (in the sense of a term algebra) from a number of non-logical and logical operators, e.g., $\wedge$, $\rightarrow$, $\leftrightarrow$, etc. The set of formulae is a recursive subset of the set of expressions and satisfies certain well-formedness criteria. The set of theorems is the subset of the formulae that the formal system defines to be true. If $\mathcal{L}$ is a formal system, then $\mathrm{Exp}(\mathcal{L})$ is its set of expressions, $\mathrm{Fm}(\mathcal{L})$ the set of formulae, and $\mathrm{Thm}(\mathcal{L})$ the set of theorems; $\mathcal{L} \vdash t$ also means t is a theorem of $\mathcal{L}$.

A *conservative extension* $\mathcal{L}$ of a formal system $\mathcal{L}'$ is a formal system whose expressions are a superset of the expressions over $\mathcal{L}'$, generated from a richer set of operators, and whose formulae and theorems restricted to the expressions of $\mathcal{L}'$ are the formulae and theorems of $\mathcal{L}'$:

$$\mathrm{Fm}(\mathcal{L}) \ \cap \ \mathrm{Exp}(\mathcal{L}') \ = \ \mathrm{Fm}(\mathcal{L}'); \qquad \mathrm{Thm}(\mathcal{L}) \ \cap \ \mathrm{Exp}(\mathcal{L}') \ = \ \mathrm{Thm}(\mathcal{L}').$$

A conservative extension $\mathcal{L}$ is a *definitional extension* of $\mathcal{L}'$ if there is a mapping $\varphi : \mathrm{Exp}(\mathcal{L}) \rightarrow \mathrm{Exp}(\mathcal{L}')$ that satisfies the following conditions:

F1 $\varphi(f) \in \mathrm{Fm}(\mathcal{L}')$ for each $f \in \mathrm{Fm}(\mathcal{L})$;

F2 $\varphi(f) = f$ for all $f \in \mathrm{Fm}(\mathcal{L}')$;

F3 φ is homomorphic in all logical operators;

F4 $\mathcal{L} \vdash t$ if and only if $\mathcal{L}' \vdash \varphi(t)$; and

F5 $\mathcal{L} \vdash t \leftrightarrow \varphi(t)$.

Kleene referred to those symbols that generate the additional expressions of the extended formal system as *eliminable.*

Remark 1 (Weak Expressibility). Kleene's original definition contains a weaker version of Condition 3, namely,

F4′ if $\mathcal{L} \vdash t$ then $\mathcal{L}' \vdash \varphi(t)$.

Based on condition F5, it is possible to show that the two definitions are equivalent, assuming the usual axioms for $\leftrightarrow$ [21:§74]. As we shall discuss in several remarks below, this is not the case in the context of programming languages. Instead, condition F4′ leads to a different but related notion of language expressiveness. ∎

3 A Formal Theory of Expressiveness

As a first step towards a formal theory of expressiveness for programming languages, we adapt the logical theory of eliminable symbols to the programming language context. We develop the idea of a programming language as a formal system and reinterpret the concepts of conservative extension and eliminability accordingly. Since many of the examples in the work of Landin, Reynolds, Steele, and Sussman preserve not only the global structure of the program but also the local structure of the transformed phrases, we consider a stricter notion of eliminability as a second step. We refer to this second notion as *macro* expressibility. It satisfies the additional constraint that the transformation of eliminated phrases is always compositional. In the two subsections on the respective topics, we prove theorems about the eliminability and non-eliminability of programming constructs and apply them to a simplistic prototype language based on the λ-calculus. Both notions of expressibility suggest natural comparative measures of the expressive power of programming languages, which we present in the third subsection.

3.1 Expressibility

Like a formal system, a programming language is a system of subsets of a general language. More precisely, a programming language is a set of phrases, a subset of programs, and a semantics that determines some aspects of the behavior of programs.

Definition 3.1. (*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[1] number of function symbols $\mathbb{F}, \mathbb{F}_1, \ldots$ with arities $a, a_1, \ldots$;
- a set of $\mathcal{L}$-programs, which is a non-empty, recursive subset of the set of phrases; and
- a semantics, $eval_{\mathcal{L}}$, which is a recursively enumerable predicate on the set of $\mathcal{L}$-programs. If $eval_{\mathcal{L}}$ holds for a program P, the program *terminates*.

The function symbols are referred to as *programming constructs* or *programming facilities*.

Definition 3.1 is an abstraction of the typical specifications of many realistic programming languages. Most languages have a context-free syntax yet enforce additional context-sensitive constraints by recursive[2] decision procedures. Examples of such constraints are scoping[3] and typing rules, which ensure that names only occur in certain pieces of text and only range over a restricted set of values.

To avoid restrictive assumptions about the set of programming languages, the definition only requires that the semantics observe the termination behavior of programs. By omitting any references to the characteristics of results, it is possible to consider programming languages with and without observable data. For programming languages with simple output data, i.e., constants or opaque representations of procedures, the definition is in many cases equivalent to a definition that refers to the observable output of a program. For a consideration of languages with infinite output, e.g., through imperative output statements or through potentially infinite lists, the definition needs some adjustments.

Finally, the above definition of a programming language also shows that, in a certain sense, a programming language is a formal system. The set of phrases corresponds to the expressions of a formal system, the set of programs plays the role of the set of formulae, and the set of terminating programs is the analog of the set of theorems. In the terminology of universal algebra, the set of expressions is the universe of a free term algebra [5]; instead of relying on the more typical algebraic

[1] We assume that there is enough structure on an infinite set of constructors for specifying the decidability of predicates and the recursiveness of translations on the set of phrases. In the following examples, this is obviously the case.

[2] A notable exception is Scheme as defined in the standard report [35], which only has a recursively enumerable set of programs: An expression is a Scheme program if and only if it has the same result for all possible evaluation orders in its applications. We consider this an unfortunate aberration rather than an interesting extension of our definition.

[3] Although most languages impose lexical scoping, Definition 3.1 only accounts for this fact through the recursive selection of programs from the set of phrases. An explicit inclusion of the lexical scoping structure through a Church encoding [7] of the language in a typed lambda calculus is a feasible and interesting alternative but would probably lead to a slightly different definition of expressibility and expressiveness.

Phrases

$$\begin{array}{lcll} e & ::= & x \mid v \mid (ee) & (\mathit{expressions}) \\ v & ::= & (\lambda_v x.e) \mid (\lambda_n x.e) & (\mathit{values}) \end{array}$$

Programs

$$e \text{ is a program if and only if } \mathrm{fv}(e) = \emptyset$$

where $\mathrm{fv}(e)$ is the set of free variables in e

Semantics

$$\mathit{eval}_\Lambda(e) \text{ holds if and only if } e \longrightarrow^* v \text{ for some } v$$

Evaluation Contexts

$$E ::= \alpha \mid ((\lambda_v x.e)E) \mid (Ee)$$

Reduction Steps

$$E((\lambda_n x.e)e') \longrightarrow E(e[x/e']) \qquad (\beta)$$

$$E((\lambda_v x.e)v) \longrightarrow E(e[x/v]) \qquad (\beta_v)$$

where $e[x/e']$ is the capture-free substituion of e' for all free x in e

Figure 1: The programming language Λ

approach of equational restrictions, the definition uses arbitrary recursive predicates for filtering out the interesting subset of programs. Unlike logic, the programming language world does not know such ubiquitous constructs as the logical connectors.

Our prototypical example of a programming language is a derivative of the language Λ of the pure λ-calculus [3]. Figure 1 summarizes its (concrete) syntax and semantics. In order to compare the expressiveness of call-by-value and call-by-name procedures later in this section, we extend Λ with a new constructor, λ_v, and rename λ to λ_n. More specifically, the Λ-*phrases* are generated from a set of variables (0-ary constructors), $\{x, y, z, \ldots\}$, and two families of unary constructors, one for each variable x: $\lambda_v x : \mathit{term} \longrightarrow \mathit{term}$ (call-by-value abstraction), $\lambda_n x : \mathit{term} \longrightarrow \mathit{term}$ (call-by-name abstraction), and one binary constructor: $\cdot : \mathit{term} \times \mathit{term} \longrightarrow \mathit{term}$ (juxtaposition). Below, λ_v and λ_n denote the sets of *all* λ-constructors. For readability, we use concrete syntax for Λ-terms and adopt the traditional λ-calculus conventions about its use [3].

The constructors $\lambda_v x$ and $\lambda_n x$ bind the variable x in their term arguments. The set of *free* variables in an expression e, $\mathrm{fv}(e)$, is the set of variables that are in e and are not bound. If all variables in a Λ-term are bound, the term is *closed*. The set of Λ-*programs* is the set of closed phrases, i.e., there is only one recursive constraint that distinguishes programs from arbitrary phrases.

The operational semantics of Λ reflects the semantics of realistic programming languages like ISWIM, ML, and Scheme [34].[4] The specification of the semantics in Figure 1 follows the style of extensible operational semantics [8, 10], which is easily adaptable to the imperative extensions of Λ in the following section. An evaluation is a sequence of reduction steps on programs according to the normal-order strategy. If the program is a value (an abstraction), the evaluation stops. Otherwise, the reduction function (uniquely) decomposes the program into an evaluation context, a term with a hole (α), and a redex, the contents of the hole. A redex is either an application of a call-by-name abstraction to an arbitrary expression (β) or an application of a call-by-value abstraction to a value (β_v). In either case, the reduction function replaces the redex, $((\lambda_x.b)a)$, by a new version of the procedure body, $b[x/a]$. Then the evaluation process starts over again.

Summarizing the standard reduction process as a predicate on programs yields the operational semantics of Λ. Some useful examples of phrases are the call-by-name and call-by-value fixed point operators, which facilitate the recursive definition of functions:

$$\mathbf{Y}_n = (\lambda_n f.(\lambda_n x.f(xx))(\lambda_n x.f(xx)))$$

and

$$\mathbf{Y}_v = (\lambda_v fx.(\lambda_v g.gg)(\lambda_v x.f(\lambda_v x.(gg)x))x).$$

Two simple diverging programs are $\Omega_n = \mathbf{Y}_n(\lambda_n x.x)$ and $\Omega_v = \mathbf{Y}_v(\lambda_v xy.xy)(\lambda_v x.x)$.

To illustrate the impact of syntactic constraints on programs, we also define Λ^t, a typed variant of Λ. Λ^t has the same set of phrases as Λ but uses a type checking algorithm for filtering out valid programs. A Λ^t program is not only closed but is also typable as either an integer or a higher-order functional on integers according to the type inference system in Figure 2. It easily follows from Milner's [27] initial work on polymorphism that typability is a recursive predicate for Λ^t. The semantics of Λ^t-programs is the same as that of their untyped counterparts.

Λ^t is a typical example of a *monomorphic* language: all occurrences of a λ-bound variable have the same type. As a consequnce, the typing constraints of Λ^t exclude typical Λ-programs like $\lambda x.(xx)$ or phrases like (xx) in programs. Indeed, the $\mathbf{Y}$ operator for defining recursive functions must be typed explicitly because it would not pass the other type rules.

Based on the interpretation of a programming language as a formal system, it is easy to define the notion of a conservative programming language extension.

Definition 3.2. (*Conservative Extension & Restriction*) A programming language $\mathcal{L}$ is a *conservative extension* of a language $\mathcal{L}'$ if

- the constructors of $\mathcal{L}'$ are a subset of the constructors of $\mathcal{L}$ with the difference being $\{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$, which are not constructors of $\mathcal{L}'$;

[4] As usual, this operational semantics has only remote connections to the equational theory of the λ-calculus.

Programs

$$e \text{ is a program if and only if it is closed and } \emptyset \vdash e : \tau$$

Types

$$\tau ::= \iota \mid \tau \longrightarrow \tau$$

Type Assertion

$$A\text{: } \mathit{Variables} \multimap \mathit{Types} \qquad \text{(finite functions)}$$

$$\text{where } A[x/\tau](x) = \tau \qquad A[x/\tau](y) = A(y) \qquad \text{(functional update)}$$

Type Inference

$$A \vdash x : \tau \text{ if } A(x) = \tau \qquad \frac{A[x/\tau] \vdash e : \tau'}{A \vdash \lambda_{v,n} x.e : \tau \longrightarrow \tau'} \qquad \frac{A \vdash e : \tau' \longrightarrow \tau; \quad A \vdash e' : \tau'}{A \vdash (ee') : \tau}$$

$$A \vdash \mathbf{Y}_n : (\tau \longrightarrow \tau) \longrightarrow \tau \qquad A \vdash \mathbf{Y}_v : ((\tau \longrightarrow \tau') \longrightarrow (\tau \longrightarrow \tau')) \longrightarrow (\tau \longrightarrow \tau')$$

Figure 2: Λ^t

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

Conversely, $\mathcal{L}'$ is a *conservative restriction* of $\mathcal{L}$.

To emphasize the constructors on which the restriction and extension differ, we write $\mathcal{L}' = \mathcal{L} \setminus \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ and $\mathcal{L} = \mathcal{L}' + \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$. We also use the notation to denote the natural restriction and extension that result from subtracting or adding facilities to the syntax (provided the respective languages and, in the latter case, a semantic specification exist).

In our running example, the restricted language Λ_n is Λ without λ_v-abstractions, Λ_v is Λ 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 Λ_v- and Λ_n-phrases yields a call-by-value and a call-by-name semantics, respectively. The corresponding sublanguages of Λ^t are defined similarly.

To enrich the set of examples, we add a **let** construct to Λ. There is one binary **let** constructor for each variable x: $\mathbf{let}_x : \mathit{term} \times \mathit{term} \longrightarrow \mathit{term}$. Like λx, $\mathbf{let}_x$ binds its variable, namely in the second subexpression. The concrete syntax for $\mathbf{let}_x(e, e')$ is

$$(\mathbf{let}\ x\ \mathbf{be}\ e\ \mathbf{in}\ e').$$

The semantics of $\Lambda + \{\mathbf{let}_x\}$, or $\Lambda + \mathbf{let}$ for short, requires an additional clause in the specification of evaluation contexts

$$E ::= \ldots \mid (\mathbf{let}\ x\ \mathbf{be}\ E\ \mathbf{in}\ e)$$

and an additional clause for the reduction function:

$$E(\mathbf{let}\ x\ \mathbf{be}\ v\ \mathbf{in}\ e) \longrightarrow E(e[x/v]). \qquad (\mathbf{let}_v)$$

Otherwise the definition of the semantic predicate in Figure 1 stays the same. It is trivial to check that $\Lambda + \mathbf{let}$ is a conservative extension of Λ.

The extension of Λ^t with **let** additionally requires a type inference rule for the new construct. For greater flexibility, the new rule only requires that, at each occurrence of the abstracted variable, the named subexpression is typable with some type:

$$\frac{A \vdash e : \tau; \quad A \vdash e[x/e'] : \tau'}{A \vdash (\mathbf{let}\ x\ \mathbf{be}\ e\ \mathbf{in}\ e') : \tau'}$$

$\Lambda^t_v + \mathbf{let}$ is polymorphic in the spirit of ML [27, 28, 43:43, 44]. Unlike λ-bound variables, **let**-bound variables can have several types. For example, x in $(\mathbf{let}\ x\ \mathbf{be}\ (\lambda y.y)\ \mathbf{in}\ (xx))$ conceptually assumes two different types: $(\iota \longrightarrow \iota) \longrightarrow (\iota \longrightarrow \iota)$ and $(\iota \longrightarrow \iota)$, which makes the expression a legal program despite the self-application of a variable. Again, it is easy to see that the extension is conservative with respect to Λ^t.

For the re-interpretation of the logical notion of eliminability, we need to be more flexible. The non-existence of ubiquitous programming language features in the sense of logical connectors raises the question whether the mapping from the extended language to the core should be homomorphic and, if so, on which set of features. At this point, we recall the above-mentioned desire that our translations be structure-preserving, and the idea that the homomorphic character of a translation naturally corresponds to this property. To preserve the structure of programs as much as possible, we require that the translations be homomorphic in *all* programming facilities of the core language.

Definition 3.3. (*Eliminability; Expressible Programming Constructs*) Let $\mathcal{L}$ be a programming language and let $\{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ be a subset of its constructors such that $\mathcal{L}' = \mathcal{L} \backslash \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ is a conservative restriction. The programming facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$ *are eliminable* if there is a recursive mapping φ from $\mathcal{L}$-phrases to $\mathcal{L}'$-phrases that satisfies the following conditions:

E1 $\varphi(e)$ is an $\mathcal{L}'$-program for all $\mathcal{L}$-programs e;

E2 $\varphi(\mathbb{F}(e_1,\ldots,e_a)) = \mathbb{F}(\varphi(e_1),\ldots,\varphi(e_a))$ for all facilities $\mathbb{F}$ of $\mathcal{L}'$, i.e., φ is homomorphic in all constructs of $\mathcal{L}'$; and

E3 $eval_{\mathcal{L}}(e)$ holds if and only if $eval_{\mathcal{L}'}(\varphi(e))$ holds for all $\mathcal{L}$-programs e.

We also say that $\mathcal{L}'$ *can express the facilities* $\mathbb{F}_1,\ldots,\mathbb{F}_n,\ldots$ *with respect to* $\mathcal{L}$. We omit the qualification if the language universe is clear from the context. By abuse of notation, we write $\varphi : \mathcal{L} \longrightarrow \mathcal{L}'$.

Condition E2 in this definition implies that the mapping φ is the identity on the language $\mathcal{L}'$. It corresponds to conditions F2 and F3 of the logical notion of eliminability. According to the above interpretation of a programming language as a formal system, an interpretation of condition F4 in Section 2 requires that the translation of a terminating program in the extended language is a terminating program in the restricted language. This is precisely the contents of Condition E3. Finally, the last condition of the logical notion of eliminability, F5, has no counterpart in a programming language context because of the lack of a ubiquitous programming construct.

Remark 2 (Weak Expressibility). By using an adaptation of Kleene's original Condition F4′ (see Remark 1 in the previous section) instead of the third condition in the preceding definition, we get a **weak** notion of expressibility. The revised condition takes on the following shape:

F3′. if $eval_{\mathcal{L}}(e)$ holds then $eval_{\mathcal{L}'}(\varphi(e))$ holds,

The intuition behind this definition is that the translated phrase has at least as many capabilities as the original one. The terminology reflects our belief that *any* differences in behavior should be noted as a failure of complete expressibility. Moreover, the terminology is also consistent with the fact that expressibility implies **weak** expressibility. ∎

An alternative understanding of the above definition is that the translation maps phrases constructed from eliminated symbols to observationally indistinguishable phrases in the smaller language. In other words, replacing the original phrase with its translation does not affect the termination behavior of the surrounding programs. This relation between two phrases of programming languages is widely studied in semantics and is known as *operational* (or *observational*) *equivalence* [25, 29, 33, 34]. After developing the formal definition of operational equivalence, we can characterize sufficient conditions for the eliminability of programming constructs.

A formal definition of the operational equivalence relation relies on the auxiliary notion of a program context.

Definition 3.4. (*Contexts; Program Contexts*) An n-ary *context* over $\mathcal{L}$, $C(\alpha_1,\ldots,\alpha_n)$, is a freely generated tree based on $\mathcal{L}$'s constructors and the additional, 0-ary constructors $\alpha_1,\ldots,\alpha_n$, called *meta-variables*. All subtrees of $C(\alpha_1,\ldots,\alpha_n)$ are also n-ary contexts. If $C(\alpha_1,\ldots,\alpha_n)$ is a context and $e_1,\ldots,e_n$ are phrases in $\mathcal{L}$, then the *instance* $C(e_1,\ldots,e_n)$ is a phrase in $\mathcal{L}$ that is like $C(\alpha_1,\ldots,\alpha_n)$ except at occurrences of α_i where it contains the phrase e_i:

- if $C(\alpha_1,\ldots,\alpha_n)=\alpha_i$ then $C(e_1,\ldots,e_n)=e_i$, and
- if $C(\alpha_1,\ldots,\alpha_n)=\mathbb{F}(C_1(\alpha_1,\ldots,\alpha_n),\ldots,C_a(\alpha_1,\ldots,\alpha_n))$ for some $\mathbb{F}$ with arity a then $C(e_1,\ldots,e_n)=\mathbb{F}(C_1(e_1,\ldots,e_n),\ldots,C_a(e_1,\ldots,e_n))$.

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

For Λ_n, the context $C_0(\alpha)=(\lambda_n xy.\alpha)(\lambda_n x.x)\Omega_n$ is a program context for all expressions whose free variables are among x and y.

Since the semantic predicate of a programming language only tests a program for its termination behavior, our definition of operational equivalence compares the termination behavior of programs.[5]

Definition 3.5. (*Operational Equivalence*) Let $\mathcal{L}$ be a programming language and let $eval_{\mathcal{L}}$ be its operational semantics. The $\mathcal{L}$-phrases e_1 and e_2 are *operationally equivalent*, $e_1\cong_{\mathcal{L}} e_2$, if there are contexts that are program contexts for both e_1 and e_2, and if for all such contexts, $C(\alpha)$, $eval_{\mathcal{L}}(C(e_1))$ holds if and only if $eval_{\mathcal{L}}(C(e_2))$ holds.

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

Figure 3 contains a sequent calculus of operational equivalence on $\Lambda_v+\mathbf{let}$, which is used below in Proposition 3.7; the proof system is similar to Riecke's for a typed version of Λ_v [38]. The calculus proves equations over the language from premisses (Γ) that are finite sets of equations. It is sound but incomplete, i.e., if $\emptyset\vdash e=e'$ then $e\cong e'$ but not vice versa.

Remark 3 (Weak Expressibility). A replacement of the "if-and-only-if" condition with a simple "if" condition yields the notion of operational approximation, which plays the same role for **weak** expressibility as operational equivalence for expressibility. More specifically, the term e_1 *operationally approximates* e_2, $e_1\precapprox_{\mathcal{L}} e_2$, if for all program contexts, $C(\alpha)$, for e_1 and e_2, $eval_{\mathcal{L}}(C(e_2))$ holds if $eval_{\mathcal{L}}(C(e_1))$ holds. — In conjunction with the above context C_0, the program context $C_1(\alpha)=(\lambda_n yx.\alpha)(\lambda_n x.x)\Omega_n$ shows that x and y do not approximate each other. ∎

[5]In many cases, our definition is equivalent to the more traditional definition that compares the termination behavior of programs *and* the results, provided they are among a set of *observable* data.

$$\Gamma \vdash (\lambda_v x.e)v = e[x/v] \qquad \Gamma \vdash \Omega e = \Omega \qquad \Gamma \vdash e\Omega = \Omega$$

$$\frac{\Gamma \vdash e = e'}{\Gamma \vdash \lambda_v x.e = \lambda_v x.e'}\ x \notin \mathrm{fv}(\Gamma) \qquad \frac{\Gamma \vdash e = e'}{\Gamma \vdash ee'' = e'e''} \qquad \frac{\Gamma \vdash e = e'}{\Gamma \vdash e''e = e''e'}$$

$$\Gamma \vdash (\textbf{let}\ x\ \textbf{be}\ v\ \textbf{in}\ e) = e[x/v] \qquad \Gamma \vdash (\textbf{let}\ x\ \textbf{be}\ \Omega\ \textbf{in}\ e) = \Omega$$

$$\frac{\Gamma \vdash e = e'}{\Gamma \vdash (\textbf{let}\ x\ \textbf{be}\ e\ \textbf{in}\ e'') = (\textbf{let}\ x\ \textbf{be}\ e'\ \textbf{in}\ e'')}\ x \notin \mathrm{fv}(\Gamma)$$

$$\Gamma \cup \{e = e'\} \vdash e = e' \qquad \frac{\Gamma \cup \{e = \Omega\} \vdash e_1 = e_2; \quad \Gamma \cup \{e = v\} \vdash e_1 = e_2 \text{ for all } v}{\Gamma \vdash e_1 = e_2}$$

$$\Gamma \vdash e = e \qquad \frac{\Gamma \vdash e = e'}{\Gamma \vdash e' = e} \qquad \frac{\Gamma \vdash e = e'; \quad \Gamma \vdash e' = e''}{\Gamma \vdash e = e''}$$

Figure 3: A Calculus for Operational Equivalence on $\Lambda_v + \textbf{let}$

We now have everything in place to formalize our above idea about the connection between eliminable programming constructs and their translations. The following theorem shows that, at least to some extent, the elimination of expressible programming constructs from a program is a local process and keeps the program structure intact.

Theorem 3.6 *Let $\mathcal{L} = \mathcal{L}' + \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ be a conservative extension of $\mathcal{L}'$. If $\varphi : \mathcal{L} \longrightarrow \mathcal{L}'$ is homomorphic in all facilities of $\mathcal{L}'$ and preserves program-ness, and if $\mathbb{F}_i(e_1, \ldots, e_{a_i}) \cong_{\mathcal{L}} \varphi(\mathbb{F}_i(e_1, \ldots, e_{a_i}))$ for all $\mathbb{F}_i$ and all $\mathcal{L}$-expressions $e_1, \ldots, e_{a_i}$, then $\mathcal{L}'$ can express the facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$.*

Proof. It suffices to show that condition E3 of Definition 3.3 holds for φ. Assume that P is an $\mathcal{L}$-program such that $eval_{\mathcal{L}}(P)$ holds. By the construction of P, there is a context $C(\alpha_1, \ldots, \alpha_n)$ such that

$$P = C(p_1, \ldots, p_n)$$

where $p_1, \ldots, p_n$ are the finite number of outermost occurrences of phrases constructed from some facilities in $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$. Thus, $C(\alpha_1, p_2, \ldots, p_n)$ is a program context for p_1. It follows from the theorem's assumption that

$$eval_{\mathcal{L}}(C(p_1, \ldots, p_n)) \text{ holds if and only if } eval_{\mathcal{L}}(C(\varphi(p_1), \ldots, p_n)) \text{ holds.}$$

Repeating this step n times proves that

$$eval_{\mathcal{L}}(C(p_1, \ldots, p_n)) \text{ holds if and only if } eval_{\mathcal{L}}(C(\varphi(p_1), \ldots, \varphi(p_n))) \text{ holds.}$$

But, since C does not contain any facilities in $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$,

$$C(\varphi(p_1), \ldots, \varphi(p_n)) = \varphi(C(p_1, \ldots, p_n)) = \varphi(P).$$

Moreover, since $\mathcal{L}$ conservatively extends $\mathcal{L}'$,

$$eval_{\mathcal{L}}(P) \text{ holds if and only if } eval_{\mathcal{L}'}(\varphi(P)) \text{ holds.}$$

This completes the proof. ∎

Remark 4 (Weak Expressibility). The theorem holds for **weak** expressibility even if we replace operational equivalence by operational approximation: If $\mathbb{F}_i(e_1, \ldots, e_{a_i}) \sqsubseteq_{\mathcal{L}} \varphi(\mathbb{F}_i(e_1, \ldots, e_{a_i}))$, then the $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$ are **weakly** eliminable. ∎

An application of this theorem shows that **let** is an example of an eliminable construct. For the typed setting, this provides a precise formalization of the folk theorem that ML-style polymorphism is expressible in a monomorphic language.[6] The two examples also reveal a striking difference between the typed and the untyped language variant. Whereas the untyped **let** expression simply abbreviates an application as illustrated in the introduction, its typed counterpart maps to a version of the **let** body in which each occurrence of the abstracted variable is substituted with a copy of the named expression. The reason for this difference is that in the typed case the translation must not only preserve the semantics but also the typability in order to preserve program-ness.

Proposition 3.7 *The constructor* **let** *is eliminable in call-by-value languages.*

(*i*) Λ_v *can express* **let** *with respect to* $\Lambda_v + \textbf{let}$.

(*ii*) Λ_v^t *can express* **let** *with respect to* $\Lambda_v^t + \textbf{let}$.

Proof. (*i*) Set $\varphi(\textbf{let } x \textbf{ be } e \textbf{ in } e') = (\lambda_v x.\varphi(e'))\varphi(e)$. To show that the two phrases are operationally equivalent we proceed by induction on the number of **let**'s in the subexpression. The base case for e, e' in Λ_v proceeds as follows. By the homomorphism constraint, $\varphi(e) = e$ and $\varphi(e') = e'$. By the laws in Figure 3,

$$e = \Omega \vdash (\textbf{let } x \textbf{ be } e \textbf{ in } e') = \Omega = (\lambda x.e')e$$

and, for all values v,

$$e = v \vdash (\textbf{let } x \textbf{ be } e \textbf{ in } e') = e'[x/v] = (\lambda x.e')e,$$

[6] An alternative approach to a formalization of this folk theorem is due to Wand [44].

and therefore,

$$(\textbf{let } x \textbf{ be } e \textbf{ in } e') \cong_{\Lambda+\textbf{let}} (\lambda x.e')e.$$

The induction step proceeds along the same line.

(*ii*) Set $\varphi(\textbf{let } x \textbf{ be } e \textbf{ in } e') = ((\lambda_v d.\varphi(e')[x/\varphi(e)])\varphi(e))$ where d does not occur free in e or e'. To verify that this translation maps programs to programs, it suffices to prove that the translation preserves the implicit type assignment. We proceed by induction on the number of **let**-expressions in the program and show that, given a fixed set of type assumptions, $\varphi(e)$ has the same type as e. Assume e and e' are **let**-free, and consider the typing of an instance of **let**:

$$A \vdash (\textbf{let } x \textbf{ be } e \textbf{ in } e') : \tau'$$

for some A and τ'. By the setup of the inference system, $A \vdash e : \tau$ for some τ and $A \vdash e'[x/e] : \tau'$. Since, by assumption, $\varphi(e) = e$ and $\varphi(e') = e'$, $A \vdash \varphi(e) : \tau$ and $A \vdash \varphi(e')[x/\varphi(e)] : \tau'$. The rest follows easily: $A \vdash (\lambda_v d.\varphi(e)[x/\varphi(e')]) : \tau \longrightarrow \tau'$ and also

$$A \vdash (\lambda_v d.\varphi(e)[x/\varphi(e')])\varphi(e) : \tau'.$$

The induction step requires a lemma that proves $\varphi(e)[x/\varphi(e')]$ has the same type as $e[x/e']$ if $\varphi(e)$ and $\varphi(e')$ have the same type as e and e'.

The proof that φ preserves the semantics of typed **let**-programs follows the same pattern as part (i). Observe that

$$e = \Omega \vdash (\textbf{let } x \textbf{ be } e \textbf{ in } e') = \Omega = (\lambda d.e'[x/e])\Omega = (\lambda d.e'[x/e])e$$

and also, for all v,

$$e = v \vdash (\textbf{let } x \textbf{ be } e \textbf{ in } e') = e'[x/v] = (\lambda d.e'[x/v])v = (\lambda d.e'[x/e])e.$$

The rest is obvious. ∎

The converse of Theorem 3.6 does not hold. That is, the facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$ may still be expressible in $\mathcal{L}'$ even though the translation from $\mathcal{L}$ to $\mathcal{L}'$ maps an eliminable phrase to an observably distinct element. One reason for this is that the set of programs may not contain an element such that $\mathbb{F}_i(e_1, \ldots, e_{a_i})$ for some $\mathbb{F}_i$ occurs in a context over the restricted language, in which case it is irrelevant how the mapping φ translates this phrase. Thus, by imposing an appropriate condition, we can get a theorem on the non-expressibility of programming constructs.

Theorem 3.8 *Let $\mathcal{L} = \mathcal{L}' + \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ be a conservative extension of $\mathcal{L}'$. If for all mappings $\varphi : \mathcal{L} \longrightarrow \mathcal{L}'$ that are homomorphic in all facilities of $\mathcal{L}'$, $\mathbb{F}_i(e_1, \ldots, e_{a_i}) \not\cong_{\mathcal{L}} \varphi(\mathbb{F}_i(e_1, \ldots, e_{a_i}))$ for some $\mathcal{L}$-expressions $e_1, \ldots, e_{a_i}$ and $\mathbb{F}_i$ in $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$, and if there is a context $C(\alpha)$ over $\mathcal{L}'$ that witnesses this inequality, then $\mathcal{L}'$ cannot express the facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$.*

Proof. Let $\varphi : \mathcal{L} \longrightarrow \mathcal{L}'$ be an arbitrary mapping that is homomorphic in all facilities of $\mathcal{L}'$. Suppose $\varphi(\mathbb{F}_i(e_1, \ldots, e_{a_i})) = e$ and let $C(\alpha)$ be the context over $\mathcal{L}'$ that observes the operational difference between e and $\mathbb{F}_i(e_1, \ldots, e_{a_i})$. Since φ is homomorphic over $\mathcal{L}'$,

$$\varphi(C(\mathbb{F}_i(e_1, \ldots, e_{a_i}))) = C(\varphi(\mathbb{F}_i(e_1, \ldots, e_{a_i}))) = C(e).$$

But, by assumption, $eval_{\mathcal{L}}(C(\mathbb{F}_i(e_1, \ldots, e_{a_i})))$ holds while $eval_{\mathcal{L}}(C(e))$ and $eval_{\mathcal{L}'}(C(e))$ do not hold (or vice versa). This implies that no mapping that is homomorphic over $\mathcal{L}'$ can possibly satisfy condition E3 of Definition 3.3 if the antecedent of the theorem holds, and that consequently the programming constructs $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$ cannot be eliminable. ∎

Based on this first non-expressibility theorem, we can now prove that call-by-name Λ cannot express call-by-value abstractions and vice versa.

Proposition 3.9 *Λ extends both Λ_v and Λ_n.*

(i) Λ_n cannot express λ_v with respect to Λ.

(ii) Λ_v cannot express λ_n with respect to Λ.

Proof. [7] (i) According to the semantics for Λ_n, the application of an abstraction $(\lambda_n y.p)$ to an argument can only proceed in one of the following three manners:

1. it may uniformly diverge for all arguments: whenever $(\lambda_n y.p)e \longrightarrow^* e'$ there is always an e'' such that $e' \longrightarrow e''$;

2. it may uniformly converge to a value for all arguments, including Ω_n: for all expressions e there is a value v_e such that $(\lambda_n y.p)e \longrightarrow^* v_e$;

3. it may activate the argument for a first time: $(\lambda_n y.p)e \longrightarrow^* ee_1 \ldots e_k$ for some $e_1, \ldots, e_k$.

The proof of this auxiliary claim is a simple induction on the length of the reduction sequence. Also, it is easy to check that whenever $e \longrightarrow^* e'$ then $E(e) \longrightarrow^* E(e')$.

Let $(\lambda_v x.e)$ be an abstraction in Λ that converges upon application to some values. More specifically, let e and v be in Λ_n and assume that for some value u, $(\lambda_v x.e)v(\lambda_n x.x) \longrightarrow^* u$. Then we claim that $C(\alpha) = (\alpha\ v)(\lambda_n x.x)$ is the context that we are looking for in order to apply Theorem 3.8.

Assume that $\varphi : \Lambda \longrightarrow \Lambda_n$ is a structure-preserving translation. Then $\varphi(\lambda_x v.e)$ is or reduces to a value in Λ_n. Let $\lambda_n y.p$ be this value. Since the original abstraction terminates upon application to some value, the translation of this application must terminate as well. Therefore $\lambda_n y.p$ cannot diverge uniformly. On the other hand, the pre-image, $\lambda_v x.e$, also diverges upon application to Ω_n, which implies that the

[7] Improved by Carolyn Talcott.

translation cannot converge uniformly. Thus, let $e_1, \ldots, e_k$ be the expressions such that

$$C(\varphi(\lambda_v x.e)) = \varphi(\lambda_v x.e)v(\lambda_n x.x) \longrightarrow^* (\lambda_n y.p)v(\lambda_n x.x) \longrightarrow^* ve_1 \ldots e_k(\lambda_n x.x).$$

Setting $e = \lambda_n x.x$ and $v = \lambda_n x.\Omega_n$ (which satisfy the original asumptions), it is trivial to see that this program now diverges. If $n = 0$ the reduction continues with:

$$\ldots \longrightarrow (\lambda_n x.\Omega_n)(\lambda_n x.x) \longrightarrow \Omega_n \longrightarrow \ldots;$$

otherwise, it is:

$$\ldots \longrightarrow (\lambda_n x.\Omega_n)e_1 \ldots e_k(\lambda_n x.x) \longrightarrow \Omega_n e_2 \ldots e_k(\lambda_n x.x) \longrightarrow \ldots$$

That is, whereas $C(\lambda_v x.e)$ converges to $(\lambda_n x.x)$, $C(\varphi(\lambda_v x.e))$ diverges. By the preceding theorem, we have shown our claim.

(*ii*) This part is much simpler. Take $C(\alpha) = (\alpha\Omega_v)$, $e = (\lambda_n x.(\lambda_n x.x))$ and assume that φ is a structure-preserving translation. Clearly, $eval(C(\lambda_n x.(\lambda_n x.x)))$ holds, but, $C(\varphi((\lambda_n x.(\lambda_n x.x))))$ diverges. Hence, a structure-preserving translation cannot preserve operational equivalence, which proves the claim. ∎

Remark 5 (Weak Expressibility). By Remark 4, Λ_n can **weakly** express λ_v because $\lambda_v x.e \lesssim_\Lambda \lambda_n x.e$. ∎

An immediate corollary of Theorem 3.8 is that if for some phrase with an eliminable symbol there is no operationally indistinguishable expression in the restricted language, then the restricted language is less expressive than the full language. We use this corollary instead of the full theorem in Section 4.2.

Corollary 3.10 *Let $\mathcal{L} = \mathcal{L}' + \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ be a conservative extension of $\mathcal{L}'$. If for some $\mathbb{F}_i(e_1, \ldots, e_{a_i})$ there is a program context C over $\mathcal{L}'$ but there is no e in $\mathcal{L}'$ such that $\mathbb{F}_i(e_1, \ldots, e_{a_i}) \cong_{\mathcal{L}} e$, then $\mathcal{L}'$ cannot express the facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$*

3.2 Macro Expressibility

Although the definition of eliminable programming construct is a satisfactory first step towards a better understanding of the formal structure of programming languages, it does not completely account for the idealized notion of "syntactic sugar" of Landin and others [24, 36, 37, 40] as discussed in the introduction. In many cases, the elimination of "syntactic sugar" constructs not only preserves the global program structure but also the structure of the subexpressions of phrases built from eliminable constructs.

Recall the two examples from the introduction:

1. In a functional language with first-class functions, a **let** expression is simply an abbreviation of the immediate application of an anonymous procedure to an argument:

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

2. In a goto-free, Algol-like language,

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

In both examples, the translation of a composite phrase is the (fixed) composition of the translation of its subphrases. More technically, the translation of a phrase is the evaluation of a *term* (in the sense of universal algebra [5]) over the restricted language at the translations of the subphrases. As mentioned above, terms correspond to contexts in our framework; for clarity, we refer to contexts as *syntactic abstractions*[8] in relation to the following definition and its uses.

Definition 3.11. (*Macro Eliminability; Macro Expressibility*) Let $\mathcal{L}$ be a programming language and let $\{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ be a subset of its constructors such that $\mathcal{L}' = \mathcal{L} \setminus \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ is a conservative restriction. The programming facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$ *are macro eliminable* if they are eliminable *and* if the eliminating mapping φ from $\mathcal{L}$ to $\mathcal{L}'$ satisfies the following, additional condition:

E4 For each a-ary construct $\mathbb{F} \in \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ there exists an a-ary syntactic abstraction, A, over $\mathcal{L}'$ such that

$$\varphi(\mathbb{F}(e_1, \ldots, e_a)) = A(\varphi(e_1), \ldots, \varphi(e_a)).$$

We also say that $\mathcal{L}'$ *can macro-express the facilities* $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$ *with respect to* $\mathcal{L}$.

Remark 6 (Weak Expressibility). If some facilities are **weakly** expressible and satisfy the additional condition, we call them **weakly** macro-expressible. ∎

Since macro expressibility is a restriction of simple expressibility, Theorem 3.6 on the eliminability of constructs requires some adaptation.

Theorem 3.12 *Let $\mathcal{L} = \mathcal{L}' + \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ be a conservative extension of $\mathcal{L}'$. If $\varphi : \mathcal{L} \longrightarrow \mathcal{L}'$ is homomorphic in all facilities of $\mathcal{L}'$ and preserves program-ness, and if there is a syntactic abstraction A_i for each $\mathbb{F}_i$ in $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$ such that $\mathbb{F}_i(e_1, \ldots, e_{a_i}) \cong_{\mathcal{L}} A_i(\varphi(e_1), \ldots, \varphi(e_{a_i}))$ for all $\mathcal{L}$-expressions $e_1, \ldots, e_{a_i}$, then $\mathcal{L}'$ can macro-express the facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$.*

[8] In Lisp-like languages, syntactic abstractions are realized as *macros* [22]; logical frameworks know them as *notational abbreviations* [17]. The terminology of equational algebraic specifications [16] refers to syntactic abstractions as *derived operators*.

Proof. It is easy to see that the additional condition in the antecedent is precisely what is needed to adapt the proof of Theorem 3.6 to the stronger conclusion. ∎

Moreover, the additional condition E4 permits a simplification of the theorem to a corollary that no longer makes any reference to the translating map. The corollary is used in Section 4.

Corollary 3.13 *Let $\mathcal{L} = \mathcal{L}' + \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ be a conservative extension of $\mathcal{L}'$. If there is a syntactic abstraction A_i for each $\mathbb{F}_i$ in $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$ so that $\mathbb{F}_i(e_1, \ldots, e_{a_i}) \cong_{\mathcal{L}} A_i(e_1, \ldots, e_{a_i})$ for all $\mathcal{L}$-expressions $e_1, \ldots, e_{a_i}$, then $\mathcal{L}'$ can macro-express the facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$.*

By Proposition 3.7, Λ_v can express **let**. A simple check of the proof reveals that the translation between the two languages satisfies the antecedent of the corollary, and that therefore **let** is also macro-expressible. More importantly, however, the additional condition E4 in Definition 3.11 also leads to a stronger meta-theorem on the *non*-expressibility of facilities. The new theorem shows that new programming constructs add to the expressive power of a language if their addition affects existing operational equivalences.

Theorem 3.14 *Let $\mathcal{L}_1 = \mathcal{L}_0 + \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ be a conservative extension of $\mathcal{L}_0$. Let $\cong_0$ and $\cong_1$ be the operational equivalence relations of $\mathcal{L}_0$ and $\mathcal{L}_1$, respectively.*

- (*i*) *If the operational equivalence relation of $\mathcal{L}_1$ restricted to $\mathcal{L}_0$ expressions is not equal to the operational equivalence relation of $\mathcal{L}_0$, i.e., $\cong_0 \neq (\cong_1 |\mathcal{L}_0)$, then $\mathcal{L}_0$ can***not** *macro-express the facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$.*[9]
- (*ii*) *The converse of (i) does not hold. That is, there are cases where $\mathcal{L}_0$ cannot express some facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$, even though the operational equivalence relation of $\mathcal{L}_1$ restricted to $\mathcal{L}_0$ is identical to the operational equivalence relation of $\mathcal{L}_0$, i.e., $\cong_0 = (\cong_1 |\mathcal{L}_0)$.*

Proof. Let $eval_0$ and $eval_1$ be the respective evaluation predicates for $\mathcal{L}_0$ and $\mathcal{L}_1$.

(*i*) A difference between the restricted operational equivalence relation of $\mathcal{L}_1$ and that of $\mathcal{L}_0$ implies that there are two phrases e and e' in $\mathcal{L}_0$ and $\mathcal{L}_1$ such that either $e \cong_0 e'$ and $e \not\cong_1 e'$ or $e \not\cong_0 e'$ and $e \cong_1 e'$. For the first case, let $C(\alpha)$ be a context over $\mathcal{L}_1$ that can differentiate the two phrases e and e'. Let us say, without loss of generality, that

$$eval_1 \text{ holds for } C(e) \text{ but not for } C(e').$$

Now, assume contrary to the claim in the theorem that $\mathcal{L}_0$ can express the facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$. Then, there is a mapping $\varphi : \mathcal{L}_1 \longrightarrow \mathcal{L}_0$ that satisfies conditions E1

[9] An extension of an equivalence relation to a larger language is also called *conservative* if the restriction to the old syntax yields the original equivalence relation. To avoid confusion, we will not use this terminology here.

through E4 of Definitions 3.3 and 3.11. By conditions E1 and E3, the programs $C(e)$ and $C(e')$ have counterparts in $\mathcal{L}_0$, $\varphi(C(e))$ and $\varphi(C(e'))$, that have the same termination behavior:

$$eval_0(\varphi(C(e))) \text{ holds because } eval_1(C(e)) \text{ holds}$$

and

$$eval_0(\varphi(C(e'))) \text{ does not hold because } eval_1(C(e')) \text{ does not hold.} \qquad (\dagger)$$

By conditions E2 and E4, the programs $\varphi(C(e))$ and $\varphi(C(e'))$ can only differ in a finite number of occurrences of e and e'. In other words, there is a program context $D(\alpha)$ over $\mathcal{L}_0$ such that $\varphi(C(e)) = D(e)$ and $\varphi(C(e')) = D(e')$: for the proof, see the *Translation Lemma* below. Next, from the assumption $e \cong_0 e'$, it follows that $\varphi(C(e)) = D(e)$ and $\varphi(C(e')) = D(e')$ have the same termination behavior in $\mathcal{L}_0$, i.e.,

$$eval_0(D(e)) \text{ holds if and only if } eval_0(D(e')) \text{ holds.}$$

But this contradicts the above fact ($\dagger$) that $\varphi(C(e)) = D(e)$ converges and $\varphi(C(e')) = D(e')$ diverges, which concludes the first case.

For the second case, assume $e \not\cong_0 e'$ and $e \cong_1 e'$. This assumption actually implies that

$$\textit{there are no contexts over } \mathcal{L}_0 \textit{ that complete both } e \textit{ and } e' \textit{ to programs.} \qquad (*)$$

For otherwise, there must be a context $C(\alpha)$ such that $eval_0(C(e))$ holds while $eval_0(C(e'))$ does not. Since $\mathcal{L}_1$ is a conservative extension of $\mathcal{L}_0$, $eval_1(C(e))$ holds, $eval_1(C(e'))$ does not, and therefore contrary to the assumption, $e \not\cong_1 e'$.

Now again, assume contrary to the claim of the theorem that $\mathcal{L}_0$ can express the additional facilities in $\mathcal{L}_1$ via an appropriate translation $\varphi : \mathcal{L}_1 \longrightarrow \mathcal{L}_0$. Since e and e' are operationally equivalent in $\mathcal{L}_1$, there must be a context $C(\alpha)$ over $\mathcal{L}_1$ such that $eval_1(C(e))$ and $eval_1(C(e'))$. By assumption, $eval_0(\varphi(C(e)))$ and $eval_0(\varphi(C(e')))$. Again by the *Translation Lemma*, the two translated programs are instances of the same program context $D(\alpha)$ such that $\varphi(C(e)) = D(e)$ and $\varphi(C(e')) = D(e')$. But by the above fact (*), such a context cannot exist, and we have thereby arrived at a contradiction. This concludes the second case of claim (i).

To finish the proof of claim (i), we must finally show that a homomorphic function preserves the structure of a program.

Translation Lemma. *Let $\varphi : \mathcal{L}_1 \longrightarrow \mathcal{L}_0$ be a translation that satisfies Conditions (0) through (3) in Definitions 3.3 and 3.11. Let $C(\alpha)$ be a context over $\mathcal{L}_1$. Then, there is a context $D(\alpha)$ over $\mathcal{L}_0$ such that $\varphi(C(e)) = D(e)$ and $\varphi(C(e')) = D(e')$.*

Proof. The proof is an induction on the structure of $C(\alpha)$. The only interesting case is the following. Say, $C(\alpha) = \mathbb{F}(C_1(\alpha), \ldots, C_a(\alpha))$ for some $\mathbb{F}$ in $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$. Then,

$$\varphi(\mathbb{F}(C_1(e), \ldots, C_a(e)) = A(\varphi(C_1(e)), \ldots, \varphi(C_a(e)))$$

and

$$\varphi(\mathbb{F}(C_1(e'), \ldots, C_a(e')) = A(\varphi(C_1(e')), \ldots, \varphi(C_a(e')))$$

for some fixed syntactic abstraction A over $\mathcal{L}_0$ by condition E4. By inductive hypothesis, there are contexts $D_i(\alpha)$, for $1 \leq i \leq a$, such that $\varphi(C_i(e)) = D_i(e)$ and $\varphi(C_i(e')) = D_i(e')$. But then $D(\alpha) = A(D_1(\alpha), \ldots, D_a(\alpha))$ is the context corresponding to $C(\alpha)$. The other cases are similar but easier. $\blacksquare_{lemma}$

The proof of the *Translation Lemma* finishes the proof of case (i).

(*ii*) We only sketch the construction of an example that proves claim (ii). For another example that is more interesting and fits more smoothly into this paper, see Subsection 4.2 on the control structure of Idealized Scheme.

For the base language, take the simply typed λ-calculus with a fixed point operator, 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 values of *arbitrary* types. Hence this simply-typed language cannot *express* these 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 additional functions on pairs (of distinctly typed components) cannot be used to distinguish phrases in the simply typed language. It follows that pairing functions and selectors increase the expressive power without destroying operational equivalences of the underlying language. $\blacksquare$

Based on Theorem 3.14, we can show that the sublanguage Λ_v is not strong enough to macro-express call-by-name abstraction, and that Λ_n is not strong enough to macro-express call-by-value abstraction. The proofs utilize the first half of the proof of claim (i).

Proposition 3.15 *Λ extends both Λ_v and Λ_n.*

(*i*) *Λ_n cannot macro-express λ_v with respect to Λ.*

(*ii*) *Λ_v cannot macro-express λ_n with respect to Λ.*

Proof. The claims are obviously consequences of Proposition 3.9.

(*i*) A direct proof for the first claim is derived from a theorem by Ong [31: Thm. 4.1.1],[10] based on the preceding meta-theorem. Consider the phrases $x(\lambda_n y.x(\mathbf{YK})\Omega y)(\mathbf{YK})$ and $x(x(\mathbf{YK})\Omega)(\mathbf{YK})$. The two are equivalent in an adequate model of Λ_n [31] and are therefore operationally equivalent:

$$x(\lambda_n y.x(\mathbf{YK})\Omega y)(\mathbf{YK}) \cong_{\Lambda_n} x(x(\mathbf{YK})\Omega)(\mathbf{YK}).$$

The operational reasoning for a verification of this equivalence is as follows. No matter which argument the procedure x evaluates first, the expression $(\mathbf{YK})$ eventually appears in the hole of the evaluation context, which leads to an immediate termination of the program evaluation.

[10] Gordon Plotkin pointed out Abramsky's [1] and Ong's [31] work on the lazy λ-calculus, which corrected a mistake in an early draft.

In the full language Λ, the above analysis no longer holds: Call-by-value procedures can evaluate and discard the expression $(\mathbf{YK})$ in a way that does not affect the rest of the program. Thus, the context

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

can distinguish between the two phrases: $C(x(\lambda_n y.x(\mathbf{YK})\Omega y)(\mathbf{YK}))$ terminates, but $C(x(x(\mathbf{YK})\Omega)(\mathbf{YK}))$ diverges. By Theorem 3.14, Λ_n cannot express λ_v.

(*ii*) Consider the expressions $\lambda_v f.f(\lambda_v x.x)\Omega$ and $\lambda_v f.\Omega$. 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. A formal proof is straightforward, based on the proof rules in Figure 3. In the extended language Λ, however, 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, which proves that the extension of Λ_v to Λ does not preserve the operational equivalence relation. Again by Theorem 3.14, λ_n is not expressible. ∎

Remark 7 (Weak Expressibility). Remark 5 and the proof of Proposition 3.15 show that Theorem 3.14 does *not* carry over to **weak** expressibility because Λ_n can **weakly** macro-express λ_v and yet $\cong_{\Lambda_n} \not\subseteq \cong_{\Lambda}$. That is, even in the case of a language extension that does not preserve the operational equivalence or approximation relation, the restricted language may already be able to express the new facilities in a weak sense. ∎

For a second application of Theorem 3.14, we show that the polymorphic **let** construct of Λ_v^t is *not macro-expressible* in Λ^t. On one hand, this lemma confirms the folk knowledge that a polymorphic **let** adds expressive power to a monomorphically typed programming language. It does not contradict the above proposition, which only shows that a polymorphic **let** is *expressible* in a monomorphic language. On the other hand, this lemma provides an example of an interesting facility that is expressible but not macro-expressible relative to the same language. The proof relies on the second part of claim (i) in the meta-theorem.

Proposition 3.16 *Λ_v^t cannot macro-express* **let** *with respect to $\Lambda_v^t +$* **let***.*

Proof. Consider the expressions $((gx)(ff)))$ and $(\lambda_v d.((gx)(ff)))(gx)$. Since both contain a self-application of the variable f, there is no Λ_v^t program context for the two expressions, and the two programs cannot be operationally equivalent:

$$((gx)(ff))) \not\cong_{\Lambda^t} (\lambda_v d.((gx)(ff)))(gx)$$

Still their dynamic behavior is the same. The difference between the two programs is that the second computes the application of g to x twice, throwing away the first result via a vacuous abstraction:

$$gx = \Omega \vdash (gx)(ff) = \Omega(ff) = \Omega = (\lambda_v d.((gx)(ff)))\Omega = (\lambda_v d.((gx)(ff)))(gx)$$

and, for all values v,

$$gx = v \vdash (gx)(ff) = v(ff) = (\lambda_v d.((gx)(ff)))v = (\lambda_v d.((gx)(ff)))(gx).$$

Thus, in the extended language, where the variable f can be **let**-bound in an appropriate context, the two program fragments are equivalent:

$$((gx)(ff)) \cong_{\Lambda^t+\mathbf{let}} (\lambda_v d.((gx)(ff)))(gx).$$

Together with the above inequality, this proves the proposition. ∎

Propositions 3.15 and 3.16 provide several examples of pairs of *universal* programming languages that we can differentiate according to our expressiveness criterion. With the full language Λ, it also provides an example of a language that can express *more* than Λ_v and Λ_n. We have come to a point where we can formally distinguish the expressive power of programming languages.

3.3 Expressiveness

The two notions of expressibility are also simple comparison relations for languages and their conservative extensions. For a comparison of *arbitrary* programming languages, these relations are too weak. One solution is to conceive of our abstract programming languages as *signatures* (or *types* in the sense of universal algebra [5]) for classes of real programming languages. It is then possible to compare languages by comparing their signatures if one signature happens to be a conservative extension of the other. Though appealing at first glance, this idea only relaxes syntactic constraints such that the languages under comparion do not have to have the same syntax.

An alternative solution is to consider a common language universe that is a conservative extension of two or more programming languages. Given a common universe that fixes the meaning of a number of interesting programming constructs, there is a natural extension of the notion of expressibility to a notion of relative expressive power. Intuitively, a programming language is less expressive than another if the latter can express all the facilities the former can express in the language universe.

Definition 3.17. (*(Macro) Expressiveness*) Let $\mathcal{L}_0$ and $\mathcal{L}_1$ be conservative language restrictions of $\mathcal{L}$. *$\mathcal{L}_1$ is at least as (macro-) expressive as $\mathcal{L}_0$ with respect to $\mathcal{L}$* if $\mathcal{L}_1$ contains or can (macro-) express a set of *$\mathcal{L}$-constructs* whenever $\mathcal{L}_0$ contains or can (macro-) express the additional facilities.

The expressiveness relation is obviously a pre-order on sublanguages in a given language framework; it is also monotonic in its third argument provided the extension to the universe is conservative.

Theorem 3.18 *Let $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_2$ be conservative restrictions of $\mathcal{L}$, and let $\mathcal{L}$ be a conservative restriction of $\mathcal{L}'$.*

(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 $\mathcal{L}$.

(iii) If $\mathcal{L}_0$ is less expressive than $\mathcal{L}_1$ with respect to $\mathcal{L}$, then $\mathcal{L}_0$ is less expressive than $\mathcal{L}_1$ with respect to $\mathcal{L}'$.

Proof. The proof is an easy calculation, verifying the conditions based on the above definitions. ∎

However, a uniform change to all languages can change expressiveness relations.

Theorem 3.19 *Expressiveness relationships are not invariant under uniform extensions of the languages.*

Proof. For a simple example, consider Λ, Λ_n, and Λ_v, and recall that the two sublanguages are incomparable by Propositions 3.9 and 3.15. To prove the claim, we uniformly add a **begin** construct, (**begin** e e), that evaluates two expressions in sequence and then discards the first value. The formal specification requires an extension of the set of evaluation contexts to

$$E ::= \ldots \mid (\mathbf{begin}\ E\ e)$$

and an additional reduction clause:

$$E(\mathbf{begin}\ v\ e) \longrightarrow E(e).$$

Now, $\Lambda_n + \{\mathbf{begin}\}$ can (macro-) express $\lambda_v x.e$ as $\lambda_n x.(\mathbf{begin}\ x\ e)$, but **begin** does not add anything to the power of Λ_v: after all (**begin** e_1 e_2) is (macro-) expressible as $(\lambda_v xy.y)e_1 e_2$ in Λ_v. Thus, in the extended setting $\Lambda_n + \{\mathbf{begin}\}$ is more expressive than $\Lambda_v + \{\mathbf{begin}\}$.

The claim is still valid if the new facility is already in the language universe. Take the same example and add $\lambda_v xy.y$, i.e., Abramsky's [1] convergence tester C for Λ_n, to both sub-languages, which is equivalent to adding **begin**. ∎

The example in the preceding theorem formalizes Algol 60's definition of call-by-value as an abbreviation of a call-by-name procedure preceded by an additional block or statement [30:12]; i.e., it is not the pure call-by-name subset of Algol that can define call-by-value but an extension thereof that includes a "strict" facility. The theorem thus shows how dangerous it is to use such informal claims as call-by-name can or cannot express call-by-value *etc.* These claims only tend to be true in specific language universes for specific conservative language restrictions: they often have no validity in other contexts!

Syntax

$$
\begin{array}{rcll}
c & ::= & 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots & (\textit{numerals}) \\
f & ::= & 0? \mid 1^{+} \mid 1^{-} \mid + \mid - & (\textit{numeric functions}) \\
v & ::= & c \mid f & (\textit{constants}) \\
 & & \mid (\textbf{lambda}\ (x \ldots)\ e) & (\textit{abstractions}) \\
e & ::= & v & (\textit{values}) \\
 & & \mid x & (\textit{variables}) \\
 & & \mid (e\ e \ldots) & (\textit{applications})
\end{array}
$$

Semantics

$$\mathit{eval}(e) \text{ holds iff } e \longrightarrow^{*} v \text{ for some } v$$

Evaluation Contexts

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

Reduction Steps

$$
\begin{array}{rcll}
E((f\,v \ldots)) & \longrightarrow & E(\delta(f, v, \ldots)) & \text{if } \delta(f, v, \ldots) \text{ is defined} \\
E(((\textbf{lambda}\ (x_1 \ldots x_n)\ e)\ v_1 \ldots v_n)) & \longrightarrow & E(e[x_1/v_1, \ldots, x_n/v_n]) &
\end{array}
$$

Constant Interpretation

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\delta(1^{+}, n) & = & n+1 & \delta(+, n, m) & = & n+m & \delta(0?, 0) & = & (\textbf{lambda}\ (x\ y)\ x) \\
\delta(1^{-}, n) & = & n-1 & \delta(-, n, m) & = & n-m & \delta(0?, n) & = & (\textbf{lambda}\ (x\ y)\ y) \\
 & & & & & & & & \text{for } n \neq 0
\end{array}
$$

Figure 4: *Pure Scheme*

---

4 The Structure of *Idealized Scheme*

Pure Scheme is a simple functional programming language. It has multi-ary, call-by-value procedures and algebraic constants. There are basic constants and functional constants. Following Plotkin, we assume the existence of a partial function (δ) from functional constants and closed values to closed values that specifies the behavior of constants in *Pure Scheme* and its extensions. Typically, the constants include integers, characters, booleans, and some appropriate functions; Figure 4 contains the appropriate definition of δ. In order to gain a complete understanding of *Idealized Scheme*, *Pure Scheme* only contains integers and a minimal set of functions on integers, elements that are expressible in a λ-calculus-based language like Λ_v.

Figure 4 contains the complete specification of *Pure Scheme*, based on a reduction semantics in the style of the previous section. Scheme programs satisfy two context-sensitive definitions: They are closed expressions, and they do not contain **lambda**-abstractions with repeated parameter names. The predicate *eval* holds for a program if the program reduces to an answer, that is, values in the case of *Pure Scheme*. If *eval* does not hold for a program, the program is either in an infinite loop or the reduction process is stuck.[11] In *Pure Scheme*, an evaluation can become *stuck* because of the application of a constant symbol to a λ-expression, the application of a numeral to a value, the application of a constant function to a value for which δ is undefined, or the application of a **lambda**-abstraction to the wrong number of arguments. As before, the reduction rules for *Pure Scheme* constitute the basis for proof systems for the operational equivalence relation in the spirit of Figure 3. For brevity, however, we shall carry out most of the proofs in this section in an informal setting; it should be clear from the proofs, though, how to formalize the various steps. In the following subsections, $\cong_{ps}$ denotes the operational equivalence relation on *Pure Scheme*; other indexes correspond to extensions of *Pure Scheme* and should be self-explanatory.

The main characteristic of *Idealized Scheme* [11, 12, 13] is the extension of the functional core language *Pure Scheme* with type predicates, local branching constructs, and imperative facilities:

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

The extensions reflect the belief that these constructs increase the expressive power of the language [40, 41]. In this section, we demonstrate how to formulate these beliefs in our formal macro-expressiveness framework.

Subsection 1 simultaneously deals with local control and type predicates because the two sets of constructs are closely related. The second subsection is a study of two different control operators, one for stopping the execution of a program and another for handling the general flow of control. The third subsection shows how imperative assignments add expressive power to the core language. Finally, the last

[11] Although this is common practice in semantic considerations, a more realistic specification would have to consider the introduction of an error mechanism. However, an error mechanism actually introduces additional expressive power, which is the reason why we consider it separately.

subsection addresses the unrelated issue of how *Pure Scheme* relates to so-called "lazy" functional languages, or more precisely, to call-by-value languages with call-by-name data constructors. We thus hope to reconcile Proposition 3.15 on the non-expressibility of call-by-name abstractions in Λ_v and *Pure Scheme* with the wide-spread belief that "lazy" data constructions are available in higher-order, by-value languages.

4.1 Local Control and Dynamic Types

The programming language world knows two types of local branching statements: the boolean-value based **if**-construct for distinguishing two values from each other, and the Lisp-style **if**-construct for distinguishing one special value from all others. The semantics of the former relies on the presence of two distinct values: false and true, or $\mathsf{0}$ and $\mathsf{1}$. Assuming an extension of the set of evaluation contexts to

$$E ::= \ldots \mid (\mathbf{Bif}\ E\ e\ e),$$

the following two additional reduction rules characterize the behavior of truth-value based **Bif**:

$$E(\mathbf{Bif}\ \mathsf{1}\ e_t\ e_f) \longrightarrow E(e_t) \qquad (\mathbf{Bif}.\mathsf{true})$$
$$E(\mathbf{Bif}\ \mathsf{0}\ e_t\ e_f) \longrightarrow E(e_f). \qquad (\mathbf{Bif}.\mathsf{false})$$

If the test value in a **Bif**-expression is neither $\mathsf{0}$ nor $\mathsf{1}$, the evaluation of the program is undefined, or equivalently, such a **Bif**-expression is operationally indistinguishable from a diverging expression. The extension is obviously conservative; we refer to it as $PS(\mathbf{Bif})$.

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

Proposition 4.1 *Pure Scheme can macro-express* **Bif***.*

Proof Sketch. The proposition follows from Corollary 3.13 and is basically due to Landin and Burge [23:115], who realized that vacuous **lambda** abstractions could be used to suspend computations. Consider the syntactic abstraction:

$$\begin{aligned}(\mathbf{Bif}\ \alpha\ \alpha_t\ \alpha_f) \ = \ & ((\mathbf{lambda}\ (t\ thn\ els) \\ & \qquad (((\mathsf{0?}\ (\mathsf{1}^-\ t))\ thn\ ((\mathsf{0?}\ t)\ els\ (\mathbf{lambda}\ ()\ \Omega))))) \\ & \quad \alpha\ (\mathbf{lambda}\ ()\ \alpha_t)\ (\mathbf{lambda}\ ()\ \alpha_f))\end{aligned}$$

It is easy to show that this abstraction is operationally equivalent to **Bif**. If the replacement for α is neither $\mathsf{0}$ nor $\mathsf{1}$, then both expressions diverge. Otherwise, both expressions select one of the replacements for α_t or α_f and eliminates the other. The right-hand side accomplishes this by suspending the two expressions in 0-ary procedures and invoking one of them after the selection has been made. ∎

Remark 8 (Weak Expressibility). An extension of *Pure Scheme* with **Bif** and two distinct new values, true and false, would not be macro-eliminable. Otherwise φ would have to map true to a term t in *Pure Scheme*, which implies that the programs

$$(\textbf{Bif}\ \mathsf{true}\ 1\ 2),$$

and

$$(\textbf{Bif}\ t\ 1\ 2),$$

map to the same image, namely

$$A(t, \varphi(e_1), \varphi(e_2))$$

for some fixed syntactic abstraction A. But it is then impossible that the translation preserves program behavior because the first program terminates and the second one diverges.

Clearly, **Bif** is still **weakly** expressible since the translation will only force more programs to terminate. In a typed version of *Pure Scheme*, the problem would disappear. The type system would not admit a program with an ill-typed **Bif**-expression. Put differently, since a typed version of *Pure Scheme* admits fewer programs, expressiveness propositions are stronger. ∎

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 again serving as false, the reduction rules differ accordingly from (**Bif**.true) and (**Bif**.false):

$$\begin{aligned} (\textbf{Lif}\ v\ e_t\ e_f) &\longrightarrow e_t \text{ for } v \neq 0 && (\textbf{Lif}.v) \\ (\textbf{Lif}\ 0\ e_t\ e_f) &\longrightarrow e_f. && (\textbf{Lif}.\mathsf{nil}) \end{aligned}$$

Proposition 4.2 *Pure Scheme cannot macro-express* **Lif**.

Proof Sketch. For readability, we carry out the proof in *PS*(**Bif**). Since **Bif** is macro-expressible by Proposition 4.1, operational equivalences of terms hold in *Pure Scheme* after expanding the **Bif**-expressions.

The proposition is a consequence of Theorem 3.14. The interesting operational equivalence is based on the following context:

$$C(\alpha) = (\textbf{Bif}\ (p\ (\textbf{lambda}\ ()\ \Omega))\ (\textbf{Bif}\ (p\ 0)\ 1\ \alpha)\ \Omega).$$

In *Pure Scheme*, the evaluation of (an instance of) this context cannot reach (the replacement of) α. First, if p is not bound to a procedure, the evaluation process diverges at the first invocation of p on (**lambda** () Ω). Thus assume p is replaced by a procedure. The rest of the proof proceeds by a case analysis on the following property of procedures: a procedure of one argument may (1) ignore its argument and return a constant result, or (2) apply a constant function symbol to its argument, or

(3) use its argument in the procedure position of an application. For the evaluation of $C(\alpha)$ to reach α, the procedure must return two different results: $\mathsf{1}$ on (**lambda** () Ω), and $\mathsf{0}$ on $\mathsf{0}$. Let us then consider the other two alternatives. On one hand, if p applies a functional constant in *Pure Scheme* to its argument, then the application (p (**lambda** () Ω)) diverges. On the other, when p uses its argument as a procedure, the evaluation again diverges in the first test position. In short, either the evaluation of C diverges at the first test position, or the procedure p produces a result that is independent of its argument. Both cases imply that an evaluation cannot reduce a redex in the replacement of α.

It follows from the above that it is inconsequential what α represents. Therefore, $C(\mathsf{1}) \cong_{ps} C(\Omega)$. In the larger setting of *PS*(**Lif**), however, the preceding analysis does not hold. A context over *PS*(**Lif**), could bind the variable p to the procedure

$$(\textbf{lambda}\ (x)\ (\textbf{Lif}\ x\ \mathsf{0}\ \mathsf{1})),$$

which can distinguish the arguments $\mathsf{0}$ and (**lambda** () Ω) in the correct manner: $C(\mathsf{1}) \not\cong_{Lif} C(\Omega)$. ∎

As an alternative to the addition of **Lif**, dynamically typed languages generally include type predicates. For extending *Pure Scheme* with a predicate symbol like $\mathsf{int?}$, it suffices to extend the interpretation function δ with the clauses

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

Again, the extension, *PS*($\mathsf{int?}$), is clearly conservative.

With $\mathsf{int?}$, programs in the extended language can now effectively test the type of a value, and indeed, $\mathsf{int?}$ can express **Lif**. It follows that $\mathsf{int?}$ is not expressible in *Pure Scheme*.

Proposition 4.3 (*i*) *Pure Scheme cannot express* $\mathsf{int?}$.
(*ii*) *PS*($\mathsf{int?}$) *can express* **Lif**.

Proof Sketch. This proof illustrates the use of reduction proofs in the framework of expressiveness. First, *PS*($\mathsf{int?}$) *can macro-express* **Lif**:

$$(\textbf{Lif}\ \alpha\ \alpha_t\ \alpha_f) = (\textbf{Bif}\ (\textbf{cand}\ (\mathsf{int?}\ \alpha)\ (\mathsf{0?}\ (\mathsf{1}^-\ \alpha)))\ \alpha_t\ \alpha_f),$$

where

$$(\textbf{cand}\ \alpha_1\ \alpha_2) = (\textbf{Bif}\ \alpha_1\ \alpha_2\ \mathsf{0})$$

and **Bif** is expressed as in Proposition 4.1 above. Second, since *PS*($\mathsf{int?}$) can express **Lif**, it is stronger than *Pure Scheme* by the preceding proposition. ∎

The converse does not hold: A Lisp-style **Lif** can distinguish between $\mathsf{0}$ and *all* other values but not an arbitrary integer from the class of procedures.

Proposition 4.4 *PS*(**Lif**) *cannot macro-express* int?.

Proof Sketch. The proof proceeds along the lines of the proof of Proposition 4.2. Instead of applying the procedure variable p to 0, the modified context invokes p on 1:

$$C'(\alpha) = (\textbf{Bif}\ (p\ (\textbf{lambda}\ ()\ \Omega))\ (\textbf{Bif}\ (p\ 1)\ 1\ \alpha)\ \Omega).$$

The analysis uses the same reasoning as the above proposition with one exception: a procedure argument may now also appear in the test position of a **Lif**-expression. As above, for the evaluation of (an instance of) C' to reach (the replacement of) α, the procedure may not invoke its argument, may not submit it to a constant function, but can test it with a **Lif**-expression. But this is irrelevant because both 1 and (**lambda** () Ω) cause a **Lif**-expression to take the same branch. Hence, $C'(1) \cong_{Lif} C'(\Omega)$, yet, with int? in the language, this is no longer the case: $C'(1) \not\cong_{int?} C'(\Omega)$. ∎

Putting it all together, we see that *Pure Scheme* can handle some but not all types of local branching decisions. A simple, boolean-valued **if** construct is expressible. The more typical Lisp-style **if** adds the expressive power to distinguish one integer value from all other values, whereas the domain predicate int? permits a distinction between each integer value and the class of all other values.

Additional Syntax

$$\begin{array}{rcll} e & ::= & \ldots \mid (\textbf{call/cc}\ e) & (\textit{continuation captures}) \\ & & \mid (\textbf{abort}\ e) & (\textit{program stops}) \end{array}$$

Additional Reduction Steps

$$\begin{array}{rcl} E(\textbf{call/cc}\ e) & \longrightarrow & E(e\ (\textbf{lambda}\ (x)\ (\textbf{abort}\ E(x)))) \\ E(\textbf{abort}\ e) & \longrightarrow & e \end{array}$$

Figure 5: *Pure Scheme* with control

4.2 Non-Local Control

A more interesting expressiveness constellation arises in the context of non-local control abstractions. *Idealized Scheme* has the operations **abort** and **call/cc**. The former facility abandons the current evaluation context, realizing a simplistic form of error handling. The latter applies its subexpression to an abstraction of the current control state, permitting almost arbitrary manipulations of the flow of control. Its name stands for "call with current continuation" because the Scheme-terminology refers to an abstraction of the control state as a "continuation" in analogy to denotational semantics. Figure 5 specifies the syntax and a simple reduction semantics of *Pure*

Scheme with both control operators. We refer to the entire extension as *PS*(control); $\cong_{c+a}$ denotes its operational equivalence. Two interesting conservative restrictions of *PS*(control) are *PS*(**abort**) = *Pure Scheme* + **abort** and *PS*(**call/cc**) = *Pure Scheme* + **call/cc** with $\cong_a$ and $\cong_c$ as their respective operational equivalences.

With the semantics of Figure 5, it is trivial to verify that the extensions are conservative over *Pure Scheme*. The semantics forms the basis of a simple equational calculus for **abort** and **call/cc**, and permits simple, algebra-like reasoning about programs with control operations [12, 13]. All three languages are more expressive than *Pure Scheme*.

Proposition 4.5 *Pure Scheme cannot macro-express non-local control constructs: Pure Scheme cannot macro-express* **abort** *or* **call/cc** *relative to PS*(**abort**)*, PS*(**call/cc**)*, and PS*(control)*.*

Proof Sketch. The proof relies on Theorem 3.14, i.e., the addition of **abort** and **call/cc** invalidate operational equivalences over *Pure Scheme*. A typical example[12] is the operational equivalence

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

As argued in the proof of Proposition 3.15(i), these two procedures are equivalent in a functional setting: both diverge when applied to a value. It is easy to check that this argument still holds in *Pure Scheme*.

With **abort** and **call/cc**, however, there are contexts that invalidate this equivalence. Two examples are (α (**lambda** (x) (**abort** x))) and (**call/cc** α). Whereas the composition of the first expression with these contexts evaluates to 0, the second expression diverges in the same contexts:

$$(\textbf{lambda}\ (f)\ ((f\ 0)\ \Omega)) \not\cong_x (\textbf{lambda}\ (f)\ \Omega).$$

for x ranging over a, c, and c+a. ∎

The next natural question is whether the two control operations are related or whether they provide distinct facilities. The following proposition shows that in *Idealized Scheme*, the two are actually independent enhancements of the expressive power of the core language.[13]

[12] This example is a folk theorem example in the theoretical "continuation" community, but it was also used by Meyer and Riecke to argue the "unreasonableness" of continuations [26].

[13] The non-expressibility of **abort** appears to be an artifact of our modeling of Scheme. A more realistic model of Scheme *systems* (as opposed to the Scheme semantics [35]) would have to include the interactive loop, which provides a delimiter for control actions [9]. By including an appropriate version of this delimiter in *PS*(control), **abort** becomes macro-expressible as a combination of **call/cc** and the control delimiter [39]. Put differently, interactive programming systems actually add expressive power to the programming language. Peter Lee [personal communication] pointed out another example of this phenomenon: The addition of a read-eval-print loop also introduces true, non-eliminable polymorphism into a language like Λ^t + **let** by providing top-level **let** declarations with an open-ended body expression. The fact that such interactive programming environments add power to their underlying languages suggests that they should be specified as a part of the language standards!

Proposition 4.6 *The control constructs* **abort** *and* **call/cc** *cannot express each other:*

(*i*) *PS*(**abort**) *cannot macro-express* **call/cc** *with respect to PS*(control) [39].

(*ii*) *PS*(**call/cc**) *cannot (macro-) express* **abort** *with respect to PS*(control).

Proof Sketch. (*i*) The proof of the first claim shows that **call/cc** destroys operational equivalences in *PS*(**abort**). A typical example is $C(1) \cong_a C(\Omega)$ where

$$\begin{aligned} C(\alpha) = \; & (\textbf{Bif}\ (f\ (\textbf{lambda}\ (k)\ ((k\ 1)\ \Omega))) \\ & \quad (\textbf{Bif}\ (f\ (\textbf{lambda}\ (k)\ ((k\ \alpha)\ \Omega)))\ 0\ 1) \\ & \quad \Omega) \end{aligned}$$

These two terms could only differ if the procedure f invokes its argument, and if this invocation could return a result. In *PS*(**abort**), this is impossible because expressions can either produce a value, diverge, or abort. Therefore, the body of f's first argument, (**lambda** (k) $((k\ 1)\ \Omega)$), either aborts or diverges, but certainly cannot return a value. After adding **call/cc**, however, a context that binds f to

$$(\textbf{lambda}\ (x)\ (\textbf{call/cc}\ x))$$

can distinguish the term $C(1)$ from $C(\Omega)$: $C(1) \not\cong_{c+a} C(\Omega)$.

(*ii*) The second claim is a consequence of Corollary 3.10, i.e., there is a program with an **abort** expression for which it is impossible to find an operationally equivalent **call/cc** expresion. The program is ((**lambda** (d) Ω) (**abort** 1)); it is the composition of the context ((**lambda** (d) Ω) α) over *PS*(**call/cc**)and an **abort** expression.

The absence of an operationally equivalent expression for (**abort 1**) from *PS*(**call/cc**) follows from the property that expressions in the restricted language cannot eliminate their evaluation context. More technically, if $E(e)$ is a program over *PS*(control) such that all occurrences of **abort** expressions have the form (**abort** $E(e')$) for some e', then either $E(e) \longrightarrow^* E(v)$ or $e \cong_{c+a} \Omega$. The proof of this auxiliary claim is a routine induction on n in the following statement:

> *If* $E(e) \longrightarrow^n E(e')$ *then either (1)* e' *is a value, or (2)* e' *contains a stuck redex, or (3) there is an* e'' *such that* $E(e') \longrightarrow E(e'')$ *and* $E(e'')$ *satisfies the above condition.*

Since an expression over *PS*(**call/cc**) does not contain any **abort** expression, it vacuously satisfies the antecedent of the auxiliary claim. Hence, it either diverges or it returns a value and cannot be interchanged with an **abort** expression without effect on the behavior of a *PS*(control) program.

From the existence of a program that contains (**abort** 1) and the non-existence of an operationally equivalent expression, it follows that *PS*(**call/cc**) cannot express **abort** in *PS*(control). ∎

The preceding proposition not only establishes the formal expressiveness relationship among the control operators of *Idealized Scheme*, but it also provides a concrete example for the second claim in Theorem 3.14.

Theorem 3.14 (restated) *Let $\mathcal{L}_1 = \mathcal{L}_0 + \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ be a conservative restriction of $\mathcal{L}_0$. Let $\cong_0$ and $\cong_1$ be the operational equivalence relations of $\mathcal{L}_0$ and $\mathcal{L}_1$, respectively.*

(*ii*) *The converse of* (*i*) *does not hold. That is, there are cases where $\mathcal{L}_0$ cannot express some facilities $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$, even though the operational equivalence relation of $\mathcal{L}_1$ restricted to $\mathcal{L}_0$ is identical to the operational equivalence relation of $\mathcal{L}_0$, i.e., $\cong_0 = (\cong_1 |\mathcal{L}_0)$.*

Proof. By the preceding proposition we know that *PS*(**call/cc**) cannot express **abort**. To finish the proof, we only need to prove that the operational equivalence relation of *PS*(**call/cc**) is a subset of the operational equivalence relation of *PS*(control): $\cong_c \; \subseteq (\cong_{c+a} |PS(\textbf{call/cc}))$; the other direction is obvious.

Assume that $e \not\cong_{c+a} e'$. We prove that $e \not\cong_c e'$. Suppose there is a context C that can distinguish e and e' in *PS*(control). If the context is also a context over *PS*(**call/cc**), the result is immediate. Otherwise, C contains a number of **abort** expressions, and there exists a context $D(\alpha, \alpha_1, \ldots, \alpha_n)$ such that

$$C(\alpha) = D(\alpha, (\textbf{abort}\ e_1), \ldots, (\textbf{abort}\ e_n)).$$

Now let a be a variable that does not occur in C, and let the context $C'(e)$ be defined as follows:

$$\begin{aligned} C'(\alpha) = \ & ((\textbf{call/cc}\ (\textbf{lambda}\ (a) \\ & \quad (\textbf{lambda}\ () \\ & \quad\quad D(\alpha,\ (a\ (\textbf{lambda}\ ()\ e_1)),\ \ldots,\ (a\ (\textbf{lambda}\ ()\ e_n))))))) \end{aligned}$$

Next, we show that $eval(C(e))$ holds if and only if $eval(C'(e))$ holds. First, the program C' evaluates to an intermediate program with a few administrative steps:

$$\begin{aligned} C'(e) \longrightarrow^+ D(e,\ & ((\textbf{lambda}\ (x)\ (\textbf{abort}\ (x)))\ (\textbf{lambda}\ ()\ e_1)), \ldots, \\ & ((\textbf{lambda}\ (x)\ (\textbf{abort}\ (x)))\ (\textbf{lambda}\ ()\ e_n))) \end{aligned}$$

Second, by a generalized version of the call-by-value β axiom,

$$((\textbf{lambda}\ (x)\ (\textbf{abort}\ (x)))\ (\textbf{lambda}\ ()\ e_i)) \cong_{c+a} (\textbf{abort}\ e_i),$$

and therefore

$$D(e,\ ((\textbf{lambda}\ (x)\ (\textbf{abort}\ (x)))\ (\textbf{lambda}\ ()\ e_1)),\ \ldots,$$
$$((\textbf{lambda}\ (x)\ (\textbf{abort}\ (x)))\ (\textbf{lambda}\ ()\ e_n)))$$

terminates if and only if

$$D(e,\ (\textbf{abort}\ e_1),\ \ldots,\ (\textbf{abort}\ e_n)) = C(e)$$

terminates. The same analysis holds for the program $C(e')$, and we have thus shown that the context $C'(\alpha)$ distinguishes e and e': $e \not\cong_c e'$.[14] ∎

In summary, we have shown that *PS*(control) extends both *PS*(**abort**) and *PS*(**call/cc**) with respect to expressive power, and the latter two individually extend *Pure Scheme* itself. An interesting point is that the extension of *PS*(**abort**) to *PS*(control) is qualitatively different from the extension of *PS*(**call/cc**) to *PS*(control). We expect this point to be a topic of further investigations.

4.3 Assignments

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*(state), *Pure Scheme* with **set!** and **letrec** (for recursive declarations of variables with initial *values*), is given in Figure 6. Clearly, *PS*(state) is a conservative extension of *Pure Scheme*; the new semantics is the basis for an equational calculus for reasoning about operational equivalences in *PS*(state) [11, 12].

Proposition 4.7 *Pure Scheme cannot express* **set!** *and* **letrec**.

Proof Sketch. Consider the expression $((\textbf{lambda}\ (d)\ (f\ 0))\ (f\ 0))$, which contains the same subexpression, $(f\ 0)$, twice. In a functional language like *Pure Scheme*, the two subexpressions return the same value, if any, and, given that the value of the first subexpression is discarded, the expression is operationally equivalent to $(f\ 0)$:

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

The verification of this equivalence in the proof system of Figure 3 is straightforward.

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

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

[14] The transformation of $C(e)$ into $C(e')$ is not a homomorphic translation because it changes the top-level structure of the program. Since such a translation could encode a program as an integer and an interpreter as a function on the integers, a restricted language with all computable functions could express any feature if we allowed such global changes to programs.

Additional Syntax

$$\begin{array}{rcll} e & ::= & \ldots \mid (\textbf{set!}\ x\ e) & (\textit{assignments}) \\ & & \mid (\textbf{letrec}\ ([x\ v]\ldots)\ e) & (\textit{recursive definitions}) \end{array}$$

Extended Semantics

$$\textit{eval}(e) \text{ holds iff } e \longrightarrow^* v \text{ or } (\textbf{letrec}\ ()\ e) \longrightarrow^* (\textbf{letrec}\ (\ldots)\ v) \text{ for some } v$$

Additional Evaluation Contexts

$$E ::= \ldots \mid (\textbf{set!}\ x\ E)$$

Additional Reduction Steps

$$\begin{array}{rcl} (\textbf{letrec}\ (\ldots)\ E((f v \ldots))) & \longrightarrow & (\textbf{letrec}\ (\ldots)\ E(\delta(f, v, \ldots))) \\ & & \quad \text{provided } \delta(f, v, \ldots) \text{ is defined} \\ (\textbf{letrec}\ (\ldots)\ E((\textbf{lambda}\ (x_1 \ldots)\ e)\ v_1 \ldots)) & \longrightarrow & (\textbf{letrec}\ (\ldots[x_i\ v_i]\ldots)\ E(e[x_j/v_j,\ldots])) \\ & & \quad \text{if the } x_i, \ldots \text{ are assignable} \\ & & \quad \text{and the } x_j, \ldots \text{ are } \textit{not} \text{ assignable} \\ (\textbf{letrec}\ (\ldots[x\ v]\ldots)\ E(x)) & \longrightarrow & (\textbf{letrec}\ (\ldots[x\ v]\ldots)\ E(v)) \\ (\textbf{letrec}\ (\ldots[x\ u]\ldots)\ E((\textbf{set!}\ x\ v))) & \longrightarrow & (\textbf{letrec}\ (\ldots[x\ v]\ldots)\ E(v)) \\ (\textbf{letrec}\ ([x\ v]\ldots)\ E((\textbf{letrec}\ ([y\ u]\ldots)\ e))) & \longrightarrow & (\textbf{letrec}\ ([x\ v]\ldots[y\ u]\ldots)\ E(e)) \end{array}$$

Figure 6: *Pure Scheme* with state

which declares a procedure f. Upon the first application, the procedure modifies its 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:

$$((\textbf{lambda}\ (d)\ (f\ \mathsf{0}))\ (f\ \mathsf{0})) \not\cong_{set!} (f\ \mathsf{0}). \qquad \blacksquare$$

Not surprisingly, assignments increase the expressive power of *Idealized Scheme*. Without proof, we add that Scheme's form of assignment is equivalent to cells with a destructive update operation but without domain predicate.

4.4 Non-evaluating Constructors

Functional languages often use the call-by-name parameter-passing protocol instead of *Pure Scheme*'s call-by-value technique. Alternatively, such languages offer data constructors, say `cons`, that do not evaluate their arguments [15]. It is a widely held belief that such provisions are superfluous in the presence of higher-order procedural abstractions.

Evaluation Contexts

$$E ::= \alpha \mid (u \ldots E\ e \ldots) \qquad \text{where } u = v \setminus \{\mathsf{cons\$}\}$$

Additional Reduction Steps

$$\begin{array}{rcl} E((\mathsf{cons\$}\ e_1\ e_2)\ \mathsf{1}) & \longrightarrow & E(e_1) \\ E((\mathsf{cons\$}\ e_1\ e_2)\ \mathsf{2}) & \longrightarrow & E(e_2) \end{array}$$

Figure 7: *Pure Scheme*(cons$)

As shown in the previous section, call-by-value languages cannot express call-by-name abstractions. This result also holds in the extended framework of *Pure Scheme*. However, the introduction of non-evaluating data constructors is a bit more subtle. To study this issue more thoroughly, we consider two different conservative extensions of *Pure Scheme*, each of which incorporates a different form of a call-by-name constructor. The first extension, *PS*(lazy), provides the constructor as a first-class function:

$$\begin{array}{rcll} v & ::= & \ldots \mid \mathsf{cons\$} & (\textit{call-by-name cons}) \\ & & \mid (\mathsf{cons\$}\ e\ e) & (\textit{“lazy” values}) \end{array}$$

For simplicity, “lazy” values are functions, and 1 and 2 serve as selector arguments. Figure 7 contains the corresponding extension of the reduction relation. Though not equivalent to full call-by-name abstractions, this addition of a single call-by-name primitive still introduces new semantic capabilities. A proof of this statement is easily derivable from Proposition 3.15.

The second extension, *PS*(delayed), is a restriction of the first. The non-evaluating constructor is no longer a first-class function but is only available in first-order form:

$$v \ ::= \ \ldots \mid (\mathsf{cons\$}\ e\ e) \qquad (\textit{“lazy” values})$$

The reduction relation remains the same (Figure 7). It is this restricted extension that is expressible in *Pure Scheme*.

Proposition 4.8 *Pure Scheme can macro-express* cons$ *relative to PS*(delayed).

Proof. The desired syntactic abstraction is

$$\begin{array}{rcl} (\mathsf{cons\$}\ \alpha_1\ \alpha_2) & = & (\mathbf{lambda}\ (s) \\ & & \quad (\mathbf{Bif}\ (0?\ (1^-\ s))\ \alpha_1\ (\mathbf{Bif}\ (0?\ (1^-\ (1^-\ s)))\ \alpha_2\ \Omega))) \end{array}$$

It is easy to check that the corresponding translation satisfies the reduction clauses of the original functions. The result follows from Corollary 3.13. ∎

Remark 9 (Weak Expressibility). If the extended language contained selector functions for lazy values, the new values would only be **weakly** expressible for the same reason as **Bif**, true, and false are only **weakly** expressible (see Remark 8). ∎

5 The Conciseness Conjecture

If a programming language can represent all computable functions (on the integers), it contains a functionally equivalent counterpart to each program in a more expressive language. This raises the question as to what advantages there are to programming in the more expressive language when equivalent programs in the simpler language already exist. By the definition of an expressible construct, programs in a less expressive language generally have a globally different structure from functionally equivalent programs in a more expressive language. But, is this really all we can say about programming in more expressive languages?

By studying a number of examples, we have come to the conclusion that programs in less expressive languages exhibit repeated occurrences of programming patterns, and that this pattern-oriented style is detrimental to the programming process. To illustrate our point, we begin by presenting two examples. The first example compares two equivalent programs in variants of full Scheme and Scheme without assignment.[15] Consider the following program fragment:

```
(let (...
      [TransManager (let (TransCounter 0)
                      (lambda (TransType)
                        (if (counter? TransType)
                            TransCounter
                            (begin
                              (set! TransCounter (add1 TransCounter))
                              BODY))))]
      ...)
  ... (TransManager t1) ...)
```

The program first binds the variable *TransManager* to a procedure that handles transactions and simultaneously counts how many transactions it performs. The procedure accomplishes the counting by allocating a local variable, *TransCounter*, in its private scope with initial value 0. For every subsequent proper transaction, the

[15]This comparison is part of the folklore of the expressiveness discussion [24:165]; the particular example is adapted from our previous paper on the equational semantics of assignments [11].

procedure then uses an assignment to increase *TransCounter* by 1. There is a special transaction of appropriate type that can check the number of past transactions.

A program in *Pure Scheme*—or any other functional language without assignments—must realize the counting of transactions in a different way. For example, the above program fragment would have to be rewritten into something like the following code:

```
(let (...
      [TransManager (lambda (TransType TransCounter)
                       (if (counter? TransType)
                           TransCounter
                           (cons (add1 TransCounter)
                                 BODY)))]
      [TransCounter 0]
      ...)
  ... (let (result (TransManager t1 TransCounter))
        (let ([TransCounter (car result)]
              [ProperResult (cdr result)])
          ...)))
```

This functional version of the program declares a variable for transaction counting in the *same* scope as the transaction manager procedure, which now takes the current value of the counter as an additional argument. Upon completion of the transaction, *TransManager* returns a pair whose first component is the increased counter value and whose second component is the proper result of the transaction. All calls to *TransManager* pass the current value of *TransCounter* as an extra argument. Finally, at every call site there is also some additional code to disassemble the result in the desired way.

The functional version offers many opportunities for code simplifications. Specifically, every call site for the transaction procedure could immediately update the counter if the transaction is a proper transaction, and could return the value of the counter if the transaction causes a check on the number of previous calls:

```
(let (...
      [TransManager (lambda (TransType) BODY)]
      [TransCounter 0]
      ...)
  ... (let (TransCounter (add1 TransCounter))
        (TransManager t1)
        ...))
```

But, even after simplifying the functional version as much as possible, it always contains a large number of repeated occurrences of add1 expressions, one per call site for *TransManager*, distributed over the whole program.

The second example concerns the use of control operators. Imagine a large functional program consisting of several modules. The interfaces of these modules have fully formal specifications in the form of (variants of) parameter descriptions. Now suppose that because of some extension of the program's requirements, one of the modules needs the capability to stop the execution of the (revised) program. In a functional setting, this task is accomplished by converting the relevant parts of the program into (simplified) continuation-passing style. Specifically, each function that (transitively) uses the critical module passes a functional abstraction of the rest of the computation to the critical module, and its call sites are in such a position that upon return, no further work needs to be done. It is thus up to the critical module to stop or to continue the execution of the rest of the program. If the former is necessary, the module discards the additional argument; otherwise, it invokes the argument on some intermediate result. This programming style, however, requires fundamental changes to the original, non-abortive program. First, the interface to the critical module must now indicate the possibility that the module could abort the program execution. Second, and more importantly, the code for every call site of a function with connections to the critical module must now satisfy special conditions. Again, as in the above example, there are alternatives, but for each of them, the lack of a non-expressible facility, this time the **abort** operation, causes the occurrence of programming patterns throughout the entire program.

Based on these examples and others with a similar flavor, we have come to believe that the major negative consequence of a lack of expressiveness is the abundance of programming patterns to make up for the missing, non-expressible constructs. Clearly, a more specific conjecture about this issue must address the question of which programs actually benefit from the additional expressive power of larger languages since not all of them do. A relatively naive answer would be that improved programs use non-expressible constructs in a *sensible, observable manner*. An example of a Scheme program that does not use assignments sensibly is a function whose only assignment statement occurs at procedure entry and affects the parameter. A more formal approach to the notion of "observable manner" could be the idea that a program with a sensible use of an additional feature must be transformable into a context that can witness operational distinctions between phrases in the restricted language. Despite the lack of a good definition for "sensible uses of constructs" or even for "programming patterns," we still venture to formulate the following conjecture about the use of expressive programming languages.

Conciseness Conjecture. *Programs in more expressive programming languages that use the additional facilities in a sensible manner contain fewer programming patterns than equivalent programs in less expressive languages.*

The most disturbing consequence of programming patterns is that they are an obstacle to an understanding of programs for both human readers and program-processing programs. In the above *TransManager* example, only a global program analysis can verify that the `add1` expressions really count the number of transactions. Even worse there are two distinct explanations for a continuation-passing style subprogram in a call-by-name functional setting: it may either implement some sophisticated control structure, or it may implement a call-by-value protocol [34]. Only a thorough analysis of the details of the continuation-passing program fragment can reveal the true purpose behind the occurrence of the programming patterns. Thus, the main benefit of the use of expressive languages seems to be the ability to abstract from programming patterns with simple statements and to state the purpose of a program in the concisest possible manner.

6 Related Work

The earliest attempt at defining and comparing the expressive power of programming languages is the work on comparative schematology by Chandra, Hewitt, Manna, Paterson, and others in the early and mid seventies [6, 32]. Schematology studies programming languages with a simple set of control constructs, e.g., **while**-loop programs or recursion equations, and with *uninterpreted* constant and function symbols. As in predicate logic without arithmetic, it is possible to decide certain questions about such uninterpreted program schemas. Moreover, the languages are not universal, and it makes sense to compare the set of functions that are computable based on different sets of control constructs, or based on an interpretation of a subset of the function symbols as operations on data structures like stacks, arrays, queues. In the presence of full arithmetic, i.e., representations of integers with an addition and multiplication function, the approach can no longer compare the expressive power of programming languages since everything can be encoded and all functions become computable.

A second approach is due to Fortune et al [14]. Their basic observation is that statically typed languages without facilities for constructing diverging programs can only encode a subset of the total computable functions. For example, whereas the simply typed λ-calculus-language can define the elementary recursive functions, the second-order version of the calculus comprises the ϵ_0 elementary recursive functions. Like schematology, this approach crucially relies on the fact that the languages under consideration are not universal. While these two approaches illuminate some of the issues about the expressiveness of data and type structures, their applicability to full-fledged programming languages is impossible because an equating of expressiveness with computational power is uninteresting from the programmer's perspective.

Recently, Hoare [20] proposed classifying programming languages according to the equational and inequational laws that their programming constructs satisfy. He illustrates this idea with a collection of examples. The laws are based on denotational semantics, which are generally sound with respect to operational equivalences. Given our theorems that connect expressiveness with the validity of operational equivalences in programming languages, this approach seems to be a related attempt at formalizing or comparing the expressiveness of languages.

Williams [45] looks at a whole spectrum of formalization techniques for semantic conventions in formal systems and, in particular, programming languages. His work starts with ideas of applicative and definitional extensions of formal systems 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 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.

A secondary piece of related work is the study of the full abstraction property of mathematical models [25, 31, 33] and the representability of functions in λ-calculi [3, 4]. 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 conservative extension. As a consequence, such models naturally lead to the discovery of non-expressible programming constructs. In the framework of λ-calculus languages, such facilities are multiple argument functions that do not require the values of all arguments to determine their result [33, 1]. 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 a model. For example, whereas a direct model of Λ_n requires the above-mentioned facility for exploiting deterministic parallelism, a continuation model leads to operations on continuations and to restrictions of such operations [39]. On the other hand, by Theorem 3.14 we also know that a change in 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, but not a replacement for, the study of expressiveness (see Proposition 3.15).

7 Towards a Formal Programming Language Design Space

In the preceding sections we developed several ideas on a formal framework for comparing the expressive power of programming languages. Based on informal claims in the literature, we argued that

- the key to programming language comparisons is a restriction on the set of admissible translations between programming languages.

Specifically, we proposed that

- the translations between languages should preserve as much of a program's structure as possible.

An application of this principle to conservative language extensions produced a number of criteria for deciding whether additional operators increase the expressive power or not. For a concrete example, we considered several language extensions of *Pure Scheme*, a simple functional programming language, and found that our formal expressiveness results are close to the intuitive ideas in the literature.

The most important criterion for comparing programming languages showed that an increase in expressive power may destroy semantic properties of the core language that programmers may have become accustomed to (Theorem 3.14). Among other things, this invalidation of operational laws through language extensions implies that there are now more distinctions to be considered for semantic analyses of expressions in the core language. On the other hand, the use of more expressive languages seems to facilitate the programming process by making programs more concise and abstract (Conciseness Conjecture). Put together, this result says that

- an increase in expressive power is related to a decrease of the set of "natural" (mathematically appealing) operational equivalences.

An interesting challenge is to find expressive extensions of languages whose additional facilities do not invalidate operational laws.[16]

The current framework is only a first step towards a formal programming language design space. On one hand, we must investigate our comparison relation for arbitrary languages in more depth before we can judge its general usefulness. On the other hand, our set of restrictions on language translations is clearly not the only interesting basis for comparing programming languages. There is an entire spectrum of feasible restrictions that yield alternative notions of expressiveness, and these alternatives deserve exploration, too. Finally, 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, and that such a theory can help a programmer in selecting the right set of constructs for solving a problem.

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 provided the motivation for further work in this direction. Bob Harper pointed out the relationship to logic, which ultimately led to the current formalization. Tim Griffin's remark that my original approach focused too much on macro expressiveness, redirected my efforts towards the broader framework of expressiveness of Section 3.1. Hans Boehm, Robert

[16]This is not to be confused with compiler annotations, which also preserve the operational equivalences but do not increase the expressive power of a language, e.g., *futures* for indicating opportunities for parallel evaluations [2, 19] and single-threaded destructive updates in functional languages [18].

Cartwright, Dan Friedman, Robert Hieb, John Lamping, Scott Smith, Rebecca Selke, Carolyn Talcott, Mitchell Wand and numerous of my patient seminar students suggested many improvements in the presentation of the material. Thanks are also due to Carl Gunter (University of Pennsylvania), Peter Lee (Carnegie-Mellon University), and Carolyn Talcott (Stanford University) for giving me opportunities to expose my ideas to a wider audience before writing them up in this form. Finally, comments by members of the POPL'88 committee, and by the anonymous referees of ESOP'90 and of this special issue of Science of Computer Programming exposed weaknesses in early drafts.

8 References

1. ABRAMSKY, S. The lazy λ-calculus. In *Declarative Programming*, D. Turner (ed). Addison Wesley, 1988, 65–116.
2. BAKER, H.G. AND C. HEWITT. The incremental garbage collection of processes. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages. SIGPLAN Notices* **12**(8), 1977, 55-59.
3. 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.
4. BERRY, G. Séquentialité de l'evaluation formelle des λ-expressions. In *Proc. 3rd International Colloquium on Programming*, 1978.
5. BURRIS, S. AND H. P. SANKAPPANARAS. *A Course in Universal Algebra.* Springer Verlag, Berlin, 1981.
6. CHANDRA, A.K. AND Z. MANNA. The power of programming features. *Journal of Computer Languages* (Pergamon Press) **1**, 1975, 219–232.
7. CHURCH, A. A formulation of the simple theory of types. *Journal of Symbolic Logic* **5**(1), 1940, 56–68.
8. FELLEISEN, M. *The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* Ph.D. dissertation, Indiana University, 1987.
9. FELLEISEN, M. The theory and practice of first-class prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, 180–190.
10. FELLEISEN, M. AND D.P. FRIEDMAN. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, edited by M. Wirsing. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986, 193–217.
11. 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.

12. Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report 100, Rice University, June 1989. Accepted at *TCS*.

13. 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.

14. Fortune, S., D. Leivant, and M. O'Donnell. The expressiveness of simple and second-order type structures. *J. ACM* **30**(1), 1980, 151–185.

15. Friedman, D.P. and D.S. Wise. *Cons* should not evaluate its arguments. In *Automata, Languages and Programming*, S. Michaelson and R. Milner (eds). Edinburgh Univ. Press, 257–284, 1976.

16. 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.

17. Griffin, T. Notational definition—A formal account. In *Proc. Symposium on Logic in Computer Science*, 1988, 372–383.

18. Guzmán, J.C. and P. Hudak. Single-threaded polymorphic lambda-calculus. In *Proc. Symposium on Logic in Computer Science*, 1990, 333–345.

19. Halstead, R. Multilisp: A language for concurrent symbolic computataion. *ACM Trans. Program. Lang. Syst.* **7**(4), 1985, 501–538.

20. Hoare, C.A.R. The varieties of programming languages. In *Proc. International Joint Conference on Theory and Practic of Software Development.* Lecture Notes in Computer Science, Springer Verlag, Berlin, 1989, 1–18.

21. Kleene, S. C. *Introduction to Metamathematics*, Van Nostrand, New York, 1952.

22. Kohlbecker, E. *Syntactic Extensions in the Programming Language Lisp.* Ph.D. dissertation, Indiana University, 1986.

23. Landin, P.J. A λ-calculus approach. In *Advances in Programming and Non-numerical Computation*, edited by L. Fox. Pergamon Press, New York, 1966, 97–141.

24. Landin, P.J. The next 700 programming languages. *Commun. ACM* **9**(3), 1966, 157–166.

25. Meyer, A.R. Semantical Paradigms. In *Proc. Symposium on Logic in Computer Science*, 1988, 236–255.

26. Meyer, A.R. and J.R. Riecke. Continuations may be unreasonable. In *Proc. 1988 Conference on Lisp and Functional Programming*, 1988, 63–71.

27. Milner, R. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* **17**, 1978, 348–375.

28. Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts and London, England, 1990.

29. Morris, J.H. *Lambda-Calculus Models of Programming Languages.* Ph.D. dissertation, MIT, 1968.

30. Naur, P. (Ed.). Revised report on the algorithmic language ALGOL 60. *Commun. ACM* **6**(1), 1963, 1–17.

31. Ong, L C.-H. Fully abstract models of the lazy lambda-calculus. In *Proc. 29th Symposium on Foundation of Computer Science*, 1988, 368–376.

32. Paterson, M.S. and C.E. Hewitt. Comparative schematology. In *Conf. Rec. ACM Conference on Concurrent Systems and Parallel Computation*, 1970, 119–127.

33. Plotkin, G.D. LCF considered as a programming language. *Theor. Comput. Sci.* **5**, 1977, 223–255.

34. Plotkin, G.D. Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.* **1**, 1975, 125–159.

35. Rees, J. and W. Clinger (Eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices* **21**(12), 1986, 37–79.

36. 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.

37. Reynolds, J.C. The essence of Algol. In *Algorithmic Languages*, edited by de Bakker and van Vliet. North-Holland, Amsterdam, 1981, 345–372.

38. Riecke, J.G. A complete and decidable proof system for call-by-value equalities. In *Proc. 17th International Conference on Automata, Languages and Programming.* Lecture Notes in Computer Science, 443. Springer Verlag, Berlin, 1990, 20–31.

39. Sitaram, D. and M. Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, 1990, 161–175.

40. Steele, G.L., Jr. and G.J. Sussman. Lambda: The ultimate imperative. Memo 353, MIT AI Lab, 1976.

41. Sussman, G.J. and G.L. Steele Jr. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.

42. Troelstra, A. S. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* Lecture Notes in Mathematics 344. Springer-Verlag, Berlin, 1973.

43. Wand, M. Complete type inference for simple objects. In *Proc. Symposium on Logic in Computer Science*, 1987, 37–44.

44. Wand, M. A types-as-sets semantics for Milner-style polymorphism. In *Proc. 11th Symposium on Principles of Programming Languages*, 1984, 158–164.

45. Williams, J.G. On the formalization of semantic conventions. Draft version: September 1988. To appear in *Journal of Symbolic logic*, 1990.