Tradeoffs in Metaprogramming [*]

[Extended Abstract]

Todd L. Veldhuizen
Open Systems Laboratory
Indiana University Bloomington
Bloomington, IN, USA

tveldhui@acm.org

ABSTRACT

The design of metaprogramming languages requires appreciation of the tradeoffs that exist between important language characteristics such as safety properties, expressive power, and succinctness. Unfortunately, such tradeoffs are little understood, a situation we try to correct by embarking on a study of metaprogramming language tradeoffs using tools from computability theory. Safety properties of metaprograms are in general undecidable; for example, the property that a metaprogram always halts and produces a type-correct instance is Π_2^0-complete. Although such safety properties are undecidable, they may sometimes be *captured* by a restricted language, a notion we adapt from complexity theory. We give some sufficient conditions and negative results on when languages capturing properties can exist: there can be no languages capturing total correctness for metaprograms, and no 'functional' safety properties above Σ_3^0 can be captured. We prove that translating a metaprogram from a general-purpose to a restricted metaprogramming language capturing a property is tantamount to *proving* that property for the metaprogram. Surprisingly, when one shifts perspective from programming to metaprogramming, the corresponding safety questions do not become substantially harder — there is no 'jump' of Turing degree for typical safety properties.

General Terms

metaprogramming, metalanguages, program generators

1. INTRODUCTION AND OVERVIEW

If one starts the clock at Konrad Zuse's insight that a computer could prepare its own instructions, metaprogramming is nearing 65 years old [4]. Happily, it shows no sign of

retiring and instead seems to grow in prominence with each passing decade. As a sort of uninvited *Festschrift* contribution I propose to turn a critical eye to it, investigating its nature both good and bad by characterizing tradeoffs between facets of interest: safety, power, succinctness, and so forth. This paper was motivated in part by ongoing controversy in the program generation community on how metaprogramming tools should approach the tradeoff between safety and power, or even if such a tradeoff exists. Representative of such tradeoffs are the strong safety guarantees of MetaML [25, 28], the unrestrained power (but compromised safety properties) of C++ template metaprogramming [8], and the moderate approach of SafeGen [14]. Similar controversy exists in the design of programming languages, where there is endless contention regarding the 'proper' tradeoff between safety properties and expressive power. Early programming systems of the 1950s often incorporated facilities for syntax extensions and customized code generators (e.g., [26]). The 1971 programming language EL1, possibly the very first to implement generics, allowed arbitrary expressions to appear where a type name was expected. The compiler dealt with such expressions by invoking a built-in interpreter to perform partial evaluation [30, 3, 12]. C++ is very much in the vein of such languages: powerful generics facilities with weak safety properties. Type-safe languages such as ML and Java represent an opposite philosophy, providing restricted forms of generics with strong safety guarantees.

A more thorough understanding of such tradeoffs would be beneficial. In computational complexity there is a well-established tradition of using theory to characterize tradeoffs: between space and time, time and randomness, communication and space, space and reversibility, and so forth. A similarly methodical investigation would be useful for the field of metaprogramming— to map out the lay of the land, so as to have a solid theory of costs and benefits when making design decisions. Part of this we can borrow from the existing literature on tradeoffs in programming languages between power and succinctness. For tradeoffs concerning safety properties we turn to computability theory, which has striking explanatory power for metaprogramming tradeoffs.

It is fruitful to approach the study of metaprogramming as the study of *generalization in software*. We view a metaprogram as a generalization of a set of concrete instances: a parser generator generalizes a class of parsers, for example. To reason about many forms of generalization in a common framework, we will adopt some uniform terminology:

- A *metalanguage* is a language in which we define gen-

eralizations.

- A *generator* is the expression of a generalization in a metalanguage.

- An *instance* is the output produced by a generator when evaluated on some input (e.g., parameters or source program).

The most powerful metalanguage possible is, of course, a general-purpose, Turing-complete programming language, as used in general source-to-source metaprogramming. As we move to more restricted metalanguages, the properties we can guarantee increase, and expressive power decreases. For instance, parametric polymorphism can be usefully viewed as defining a metalanguage for expressing generic functions: the familiar function map—

$$\mathsf{map} \ : \ (\alpha \to \beta, \mathsf{list} \ \alpha) \to \mathsf{list} \ \beta$$

generalizes over the concrete set of map functions obtained by substituting any types for α, β. Parametric polymorphism is highly restrictive, but if correctly implemented, type-safe. In such examples we apply the term 'generator' in a formal sense, representing a translation from a metalanguage to concrete instances, without the requirement that code duplication take place in the implementation. This terminology is incidentally consistent with early papers on generics that called them *type generators* [20].

If one metalanguage can express more generalizations than another, we say it has greater expressive power. By considering the properties of metalanguages of varying power, we can gain insight into the tradeoffs that exist between facets such as power, safety, and succinctness.

Some important themes are nicely illustrated by the simple example of *regular expressions*, which should be familiar to most readers. Regexps have the nice property that they *capture* exactly the computations realizable by deterministic finite automata; this guarantees every regexp can be implemented efficiently and we can check equivalence of their DFA representations in $O(n \log n)$ time [13]. Practical implementations of regexps differ from theory by introducing generalizations of frequently used patterns. For instance one can often write "[a-z]" to mean "[abcdefghijklmnopqrstuvwxyz]". The generalization "$[c_1\text{-}c_2]$" is not definable in the language of regular expressions, i.e., we cannot equate "$[c_1\text{-}c_2]$" with some regular expression "s" that produces the appropriate sequence of characters given arbitrary c_1 and c_2. Instead it acts as a sort of limited escape into a slightly stronger formalism. This escape serves to compress typical expressions by factoring out a commonly occurring motif that is 'incompressible' in the weaker formalism of regular expressions. We can view "[a-z]" as an invocation of a generator that produces the *instance* "[abc...z]". The generator has the desirable property that every instance it produces is a valid regular expression, so the escape into the stronger formalism does not contaminate the 'nice' properties of regexps. Against all this praise we note that when one nears the boundary of what is 'natural' to express in regexps, monsters are quickly encountered. For instance, the expression—

$$((a|ba(aa)^*b)(b(aa)^*b)^*(a|ba(aa)^*b)|b(aa)^*b)^* \cdots$$
$$\cdots (a|ba(aa)^*b)(b(aa)^*b)^*$$

recognizes all strings with an even number of b's and an odd number of a's. This set of strings is more easily recog-

nized by a simple program in a general-purpose language. However, in such a setting we have no guarantee that the language being recognized is, in fact, regular. Moreover, suppose we start with a program in a general-purpose language that recognizes a regular language, and rework it into a regular expression. This reworking is tantamount to *proving* that the set of strings accepted by the program is regular, and is as difficult to carry out as a proof. The resulting loss in succinctness (i.e., explosion in size) cannot be bounded by any computable function; one can say with confidence that there are C programs deciding regular languages that require 10^{100} times more characters to write down when turned into regular expressions. This result is due to pioneering work of Manuel Blum on succinctness tradeoffs [2].

The themes encountered in the above example have analogues in the design of metalanguages. In designing a metalanguage, one may have in mind a safety property it is desirable to guarantee: type-safety or termination, for example. In the ideal case a metalanguage can be found that 'captures' exactly the property: any generator expressed in the metalanguage has the property, and any generator compatible with the property can be expressed in the metalanguage. Such restricted languages generally entail a loss in succinctness compared to a general-purpose language. When it is not possible to capture exactly a safety property with a metalanguage, one is faced with two possible strategies. The first is to devise a metalanguage that guarantees the property but sacrifices expressive power; the second is to sacrifice the safety property in favour of expressive power. These strategies are exemplified by the functional language approach to generics, in which type safety is paramount, and the approach of languages such as C++ and EL1, where one has unlimited expressive power for generics but type safety is compromised. In either strategy there is the possibility of 'chasing' the property by extending the language so as to gradually recoup expressive power or safety. Parametric polymorphism gives way to F-bounded polymorphism gives way to type classes, and so forth. In C++ there is a growing effort to introduce some stronger (but optional) type safety mechanisms for generics. As the parade of language features extends to infinity we may approach arbitrarily closely having both the safety property and unlimited expressive power.

1.1 Contributions

This paper makes three kinds of contributions. First, we initiate a new research programme of characterizing tradeoffs in metalanguages. Second, we gather scattered information about tradeoffs relevant to metaprogramming and make it accessible. Third, and most importantly, we prove an array of results on tradeoffs in metaprogramming, most previously unknown. We prove that multi-stage generators are no more powerful than single-stage generators (Proposition 5.1). In general, deciding safety properties of generators is not possible. For example, the 'partial correctness' property of whether generators written in a general-purpose language always produce instances satisfying a nontrivial safety property is undecidable. The stronger property of *total correctness*, i.e., generators always halt and produce a safe instance is Π_2^0-complete (Proposition 6.3). The more interesting problem is devising metalanguages that *capture* safety properties. We prove that reworking a generator from a general-purpose metalanguage to a restricted metalanguage capturing a property is tantamount to *proving* that prop-

erty (Theorem 6.2). We also give some sufficient conditions and negative results on when languages capturing properties can exist; for example there can be no languages capturing total correctness for generators, nor languages capturing totality, and no 'functional' safety properties above Σ_3^0 can be captured. We review the major results on succinctness in programming languages, and show that bounded succinctness is only plausible for languages capturing properties in Π_2^0. We ask when "going meta" implies a jump in the degree of undecidability of a property; surprisingly, interesting safety properties do not become much harder when we go from programming to metaprogramming (Section 8). Finally, we show the existence of two distinct strategies for 'safe' metaprogramming: one safe, approximating powerful languages conservatively from below, and one powerful, approximating safe languages from above.

2. A CATALOGUE OF TRADEOFFS

We propose to investigate metalanguages and the tradeoffs they represent, restricting ourselves to tradeoffs suited to theoretical investigation.[1] We can compare metalanguages according to how they trade off important characteristics or *facets*:

- The expressive power of the metalanguage, i.e., what generators we can define in it;

- The safety properties we are guaranteed about the instances;

- Succinctness, that is, how long the inputs or parameters must be to produce instances of interest;

- The time and space complexity of "running" the generator to produce an instance.

There are additional facets not investigated in the present paper, but deserving of future research:

- The decidable properties of instances;

- The class of problem domains for which we can write generators that let us make programs shorter (i.e., compress them [29]).

- The difficulty of finding inputs to a generator that will produce a particular instance, i.e., *inversion* of a generator;

- The effort required to devise an appropriate generator, given an instance or class of instances over which we wish to generalize.

Not surprisingly, we are not free to choose the best possible properties among the above facets. Rather, fixing one property constrains our other choices. For example, the following two properties of a metalanguage are at once desirable and irreconcilable:

---

1. The ability to describe any possible generator;

2. The safety property that every generator will produce its output and stop.

The first property implies Turing-completeness; the second property implies a metalanguage that is necessarily sub-recursive (not Turing-complete).

2.1 A tour of tradeoffs

To familiarize ourselves with the nature of these tradeoffs, let us take a brief tour through the above facets. To simplify we shall embark on a one-dimensional tour starting with a universal (Turing-complete) metalanguage and descending down a chain to primitive recursive generators, exponential time generators, polynomial time, and so forth down to very weak formalisms such as CFGs (context-free grammars) and NFAs (nondeterministic finite automata). CFGs and NFAs may be used as generators by providing them 'advice' on which nondeterministic branch to take at each step as input (e.g., [5]). This descent gives us at each stop a certain class of resource-bounded generalizations we can define. How do the other facets behave as we descend from a universal language to increasingly restricted metalanguages?

- With each restriction in the power of the metalanguage, the set of generators we can express becomes, of course, smaller.

- In a universal metalanguage we have no useful guarantees about the behaviour of the generator; as we descend the properties become stronger, e.g., in primitive-recursive we are guaranteed termination.

- As we restrict the language, the program length required to express generators (if they remain expressible) increases. For at least some cases and if our steps are big enough, the increase in program length cannot be bounded by any computable function.

- The time complexity of running a generator in a universal language cannot be bounded by any computable function. As we descend generators are guaranteed to run faster and faster. NFA-based generators can run in nearly linear time.

- As we restrict the language more properties become decidable, and they become easier to decide, for example, in primitive-recursive we are guaranteed termination, for NFAs we can decide whether two generators are equivalent.

- Our ability to use generators to compress patterns is at its peak in a general-purpose language; as we descend there may be commonly occurring patterns and motifs that cannot be captured in the resource-bounded metalanguage. A classic example is Champernowne's number

$$0.12345678910111213141516171819202122\cdots$$

that is easily generated in a universal metalanguage, but becomes incompressible in suitably restricted frameworks. For example, Lempel-Ziv compression is powerless against it since every subsequence occurs equally often, a so-called 'normal' real number [18, §1.9]. Similarly, the failure of simple parametric polymorphism to

capture some useful patterns amongst types in generic programming can be viewed in terms of such patterns being 'incompressible' in the restricted metalanguage.

- Inverting a generator (finding the parameters that will cause it to generate a particular instance) is undecidable at the higher levels, and becomes easier as you descend. For example, the problem of inverting a CFG-based generator is simply parsing. For suitably restricted forms of CFGs the inversion problem is simply unification, for which very efficient algorithms exist.

- The problem of finding a generator that generalizes a given a set of instances starts hard and becomes easier as the metalanguage is restricted. In a universal language, the problem of finding a generator for one instance is closely tied to calculating Kolmogorov complexity, which is undecidable; in restricted metalanguages it becomes possible, for example there are algorithms to approximate the best context-free grammar producing a string (e.g., [5]).

3. PRELIMINARIES

To characterize the nature of these tradeoffs we employ some tools of computability theory as can be found in the introductory chapters of textbooks such as Cooper [6], and summarized here. We adopt the modernized terms for computability theory suggested by Soare [27], such as computably enumerable (c.e.) in place of recursively enumerable, computable instead of recursive, and so forth.

From the vantage point of computability theory, a program represents a function from inputs to outputs, a *partial computable function*. The computability notations for these have a straightforward correspondence to notations from programming language theory. In the perhaps more familiar 'Scott brackets' notation, one writes $[\![p]\!]^L x$ for the value produced by a program p in language L running on an input x. The corresponding notation for partial computable functions is $\varphi_p^L(x)$, but usually some universal machine (equivalently, programming language) is assumed and one writes $\varphi_p(x)$.

In computability theory it is traditional to consider every object as encoded by a unique natural number. The behaviour of a program is viewed as a partial function $\varphi_p : \mathbb{N} \rightharpoonup \mathbb{N}$, from input (coded as a natural) to output (coded as a natural), defined just for those inputs on which the program halts. Programs are likewise coded by naturals, for example by enumerating all valid programs in a language lexicographically and using a program's *index* in this list. (We will equate indices with programs throughout to minimize confusion.) The $\varphi_p(x)$ notation is rather unfortunate for metaprogramming: to represent the behaviour of the program generated by a generator g with parameters x on an input y one would write $\varphi_{\varphi_{g(x)}}(y)$. (The notation $[\![\, [\![g]\!]x \,]\!]y$ seems clearer). For consistency with the computability literature we will use the φ-notation. The smaller inset box summarizes key notations for partial computable functions. Of these the most important are $\downarrow$ (halts) and $\uparrow$ (diverges).

On occasion we shall employ the Church-Turing thesis to, when presented something computable, assert there exists a program i so that φ_i computes it.

The following fact will shortly be useful; it asserts that the partial computable functions are closed under composition.

Notations for partial computable functions	
φ_p	Partial function computed by program p
$\varphi_p(x)$	Output of p on input x
$\varphi_p(x) \downarrow$	p halts on x
$\varphi_p(x) \uparrow$	p diverges on x
$\varphi_{p,k}(x) \downarrow$	p halts in $\leq k$ steps on x
$\varphi_p = \varphi_{p'}$	p and p' compute the same partial function
$\varphi_p(\mathbb{N})$	All possible outputs of p

Fact 3.1. *If φ_x and φ_y are partial computable functions, there is a program p such that $\varphi_x \circ \varphi_y = \varphi_p$.*

In later sections we shall make use of the arithmetical hierarchy classes Σ_n^0, Π_n^0, and Δ_n^0. These are summarized in Figure 1.

4. A UNIVERSAL METALANGUAGE

We wish to reason about metalanguages and the tradeoffs they represent. If generators were expressed in very different languages, this would cause notational confusion. We shall instead fix a universal language, and require that every metalanguage be a restricted subset of the universal language. We can do this without loss of generality by noting that any metalanguage may be embedded in a universal language by means of pasting together an interpreter in the universal language with a generator and its input as a string. This provides a straightforward embedding of any language into the universal language. A typical example of such a construction is:

```
int main()
{
  print(Interpret_Lisp(
    "(lambda (x y) (plus x y))",
    "(1 2)"));
}

string Interpret_Lisp(string prog, string input)
{
  ...
}
```

Such translations are easy to produce; it is easy to extract the original program from its embedded version; and the resulting program is longer by only a constant amount (the size of the interpreter plus a little extra). This is a so-called "two-part code" construction (interpreter plus program) [18, §2.1.1] and it preserves all the properties that interest us. We formalize these claims as follows.

Claim 4.1. *There is a universal language U such that for any metalanguage A, there is a computable function f translating A programs to U so as to satisfy these properties:*

1. *For every A-program p, the translated program $f(p)$ has the same meaning, i.e., $\varphi_p^A = \varphi_{f(p)}^U$;*

2. *We can computably recognize the programs that have been translated from A, i.e., $f(\mathbb{N})$ is also decidable;*

3. *We can computably reverse the mapping translation f;*

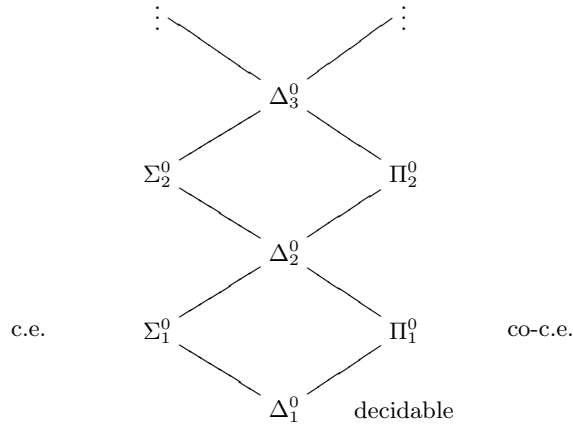4. *The translation adds at most a constant factor to the program size.*

Figure 1: Summary of the arithmetical hierarchy

Proof. (Sketch) Since the programming language A is assumed to be implementable on a computer, we can find an interpreter for A in the universal language U; call this $\mathsf{Int}(p, x)$. Define the translation function to be the equivalent of $f(p) = \lambda x.\mathsf{Int}(p, x)$. Then (1) follows from the use of an interpreter; (2) is guaranteed by the ability to examine the translated version and check that the interpreter is exactly the interpreter for the language A; (3) is ensured by adopting an appropriate "quotation" mechanism for the embedded program; (4) follows from the usual "two-part code" argument [18, §2.1.1] □

We mention also that with a sufficiently powerful machine model we can often devise efficient interpreters that preserve asymptotic time and space complexity up to some small overhead, e.g., logarithmic.

In the remainder of the paper we assume, for tidiness, that metalanguages are all defined by decidable subsets of a fixed universal language.

5. MULTISTAGE GENERATION

We begin our investigation with some simple results concerning the power of having multiple stages, rather than a single stage. In multistage generation one has multiple generators, each producing output taken as input by the next (e.g., [9, 28]). Assuming Church-Turing we consider each stage to be represented by a partial computable function. We may then make use of the fact that by definition, partial computable functions are closed under composition. The following result is then straightforward:

Proposition 5.1. *For every multistage generator there is an equivalent one-stage generator.*

Proof. This is a simple consequence of the class of partial computable functions being closed under composition. Let $\varphi_a, \varphi_b, \ldots, \varphi_n$ be a k-stage generator. Then by repeated application of Fact 3.1, there exists a program α such that $\varphi_\alpha = \varphi_a \circ \varphi_b \circ \ldots \circ \varphi_n$. □

In some situations we may want additional input at each stage. This does not offer any additional *theoretical* power, since we can provide all these inputs up-front to the initial stage and thread unused inputs through to later stages.

Most suitable programming language mechanisms offer function composition mechanisms that are succinct, i.e., the program length of the composed functions is only slightly higher than sum of the lengths of the individual functions. From this we can infer that multistage generation is no more succinct than one-stage generation.

Although multistage generators are no more powerful than single-stage generators, they do have important practical use in distinguishing (say) generation, compilation, load and run-time stages. We are, however, justified in the remainder of this paper to consider only the single-stage case.

6. TRADEOFFS IN SAFETY

A primary concern in contemporary generator research is safety of generators. Some typical questions are:

- Will the generator always halt and produce an instance?

- Will the instance produced by the generator be syntactically well-formed? typable? semantically correct?

- If a generated instance is *not* going to be correct, can we detect this and produce a sensible diagnostic message?

- Can we devise restricted languages that 'capture' useful safety properties, e.g., every generator we write in the language always produces type-safe instances?

As a prelude to tackling deeper questions, in this section we review the well-known phenomena that most functional properties of programs are undecidable, and explore the implications for generators. Readers familiar with Rice's theorem are encouraged to skip ahead.

We can equate a safety property with the set of generators having that property. For example, if we are concerned that the generator always halt, we can consider the set of generators that halt on every input. Some such sets lie in a special class called *index sets* in computability theory [6].

Definition 1 (Index set). An index set is any set of generators $B \subseteq \mathbb{N}$ with the following closure property: if $g \in B$ and g' is some generator with the same behaviour as g, i.e., $\varphi_g = \varphi_{g'}$, then $g' \in B$ also.

We can view an index set as a *functional property* of generators, i.e., a property definable only in terms of input-output behaviour and without reference to (for example) space and time consumption.

The safety property 'halts on every input' can be represented by an index set Tot containing every *total* computable function.

Definition 2 (Total functions). Define Tot be the index set containing all total computable functions, i.e., generators that halt and produce an instance for every input:

$$\text{Tot} = \{g \in \mathbb{N} \mid \forall x \, . \, \varphi_g(x) \downarrow\} \tag{1}$$

The problem "Given generator g, is $g \in$ Tot?", i.e., whether a generator always halts after some time and produces an output, is of course undecidable, as are most problems concerning halting. We might however wonder if simpler properties might be checkable, but this turns out not to be the case because of Rice's theorem.

Theorem 6.1 (Rice [23]). *The only decidable index sets are $\emptyset$ and $\mathbb{N}$.*

It is a quick step from Rice's theorem to proving that partial correctness of generators — if the generator halts, it produces a safe instance — is undecidable. We model whatever safety property of instances we are interested in (type-safety, syntactic correctness) by a set SafeInstance of instances satisfying that property.

Definition 3. A safety property of instances is a set SafeInstance with $\emptyset \subset$ SafeInstance $\subset \mathbb{N}$ so there is at least one safe instance and one unsafe instance— otherwise, the safety property would be trivial, i.e., always true or always false.

Let SafeGenerator be the set of generators that only produce safe instances:

$$\text{SafeGenerator} \equiv \{g \mid \varphi_g(\mathbb{N}) \subseteq \text{SafeInstance}\}$$

(We write $\varphi_g(\mathbb{N})$ for the image of $\mathbb{N}$ under φ_g, i.e., the set of all instances generated by g.)

Proposition 6.1. *The problem "Is $g \in$ SafeGenerator?" is undecidable.*

Proof. This is straightforward: we prove SafeGenerator is an index set not equal to $\emptyset$ or $\mathbb{N}$ and apply Rice's theorem.

Suppose i, j are generators with $\varphi_i = \varphi_j$ and $i \in$ SafeGenerator. Then $\varphi_i(\mathbb{N}) \subseteq$ SafeInstance by definition. Since $\varphi_i = \varphi_j$ we have $\varphi_j(\mathbb{N}) \subseteq$ SafeInstance and hence $j \in$ SafeGenerator also. Therefore SafeGenerator satisfies the closure property of Defn. 1 and is an index set.

From the definition of a safety property there exists a safe instance s and an unsafe instance $\bar{s}$. Consider the following functions:

$$f(x) = s$$
$$f'(x) = \bar{s}$$

Both functions are computable and therefore we can find programs g and g' that compute them. Since $g \in$ SafeGenerator we have SafeGenerator $\neq \emptyset$. From $g' \notin$ SafeGenerator we have SafeGenerator $\neq \mathbb{N}$. By Theorem 6.1, SafeGenerator is undecidable. $\square$ $\square$

So, the general question of whether a generator always produces safe instances is undecidable; this is a simple corollary to Rice's theorem. By finding where this problem lies in the arithmetical hierarchy (Figure 1), we can obtain more precise details of its undecidability. For example, some properties that are undecidable but Σ_1^0 can be approximated nicely— we can write a program that will try to decide the property within some time limit t, and as we let $t \to \infty$ we can get a positive answer if the property is true.[2] We operate under the assumption that the safety of instances is decidable. For example, it is decidable whether the instance is syntactically correct or typable; this reflects current practice.

Proposition 6.2. *If SafeInstance $\in \Delta_1^0$ then SafeGenerator $\in \Pi_1^0$.*

Proof. A generator is safe if and only if there is no input for which it produces an unsafe instance. We can therefore define the safety property by:

$$\text{SafeGenerator}(g) \leftrightarrow \neg(\underbrace{\exists x \, . \, \varphi_g(x) \notin \text{SafeInstance}}_{\Sigma_1^0})$$

This is the negation of a Σ_1^0 formula, and is therefore Π_1^0 or co-computably enumerable. $\square$ $\square$

It is worth noting that when we shift from programming to metaprogramming, the safety problem becomes harder: if instance safety is Δ_1^0, then generator safety is Π_1^0 (and not Δ_1^0). We return to this theme in Section 8, where we ask when "going meta" is accompanied by ratcheting up a level in the arithmetical hierarchy.

That SafeGenerator is Π_1^0 implies we can approximate the property by searching for counterexamples, and if a counterexample exists, we will eventually find it. Unfortunately if we do not find a counterexample within a set amount of time we can conclude nothing about whether our generator is safe.

An even harder problem is deciding whether a generator will halt for any input and also produce an output that is in

---

[2]This is the notion of a Δ_2^0-approximating sequence [6].

SafeInstance. We show this problem is Π_2^0-complete, roughly speaking, as hard as any Π_2^0 relation.[3]

Proposition 6.3. *Suppose* SafeInstance *is decidable and let* $\psi(g)$ *be the property "g halts on every input and outputs a safe instance." Then* $\psi(g)$ *is* Π_2^0-complete.

Proof. First we show $\psi \in \Pi_2^0$. We can define $\psi(g)$ by the formula:

$$\psi(g) \leftrightarrow (\forall x \,.\, \underbrace{\underbrace{\varphi_g(x) \downarrow}_{\Sigma_1^0} \wedge\, \underbrace{\mathsf{SafeGenerator}(g)}_{\Pi_1^0}}_{\Pi_2^0} \qquad (2)$$

We have a universal quantifier afront a Σ_1^0 relation, making it Π_2^0; its conjunction with the Π_1^0 relation is Π_2^0.

The index set Tot is known to be Π_2^0 complete [6]; we reduce deciding Tot to $\psi(g)$. Given a query "Is $f \in$ Tot?" we can construct a function $f'(x)$ that evaluates f on x, and if this halts, returns a safe instance $s \in$ SafeInstance. Then $g \in$ Tot if and only if $\psi(g)$. This is a many-one reduction (see footnote) and therefore ψ is Π_2^0-complete. ☐ ☐

So, this safety property is strictly harder than deciding the "partial correctness" property SafeGenerator.

The safety situation for generators written in a general-purpose language is bleak: no nontrivial safety properties are decidable. This fact has motivated the design of special-purpose languages for generators that *are* able to guarantee some safety property. Of particular interest are languages that *capture* a safety property, which we investigate in the next section.

6.1 Languages capturing properties

As we have seen, interesting safety properties of generators written in a universal language are undecidable. We might conjecture that to ensure a safety property we must sacrifice some classes of computations that *can* be done safely. Perhaps surprisingly, this is not always the case: we can sometimes sidestep undecidability by designing restricted languages that 'capture' the property, in the sense that every restricted program has the property, and conversely, for every unrestricted program with the property there is a functionally equivalent restricted program. Consider for example the property ψ given by "$\varphi_p(x)$ halts for at most a finite number of inputs x." This property is undecidable (in fact Σ_2^0) but has a trivial language capturing it: allow only programs of the form

$$f(x) = \begin{cases} c_1 & \text{when } x = x_1 \\ c_2 & \text{when } x = x_2 \\ \vdots & \vdots \\ c_n & \text{when } x = x_n \\ \uparrow & \text{otherwise} \end{cases} \qquad (3)$$

for all finite n and arbitrary constants c_i, x_i for $i \leq n$. This language clearly captures the undecidable property ψ.

We could capture the property SafeGenerator — "every instance output by the generator is safe" — by a language

---

in which any arbitrary generator can be run, but output is filtered and any unsafe instances are replaced with safe instances. This captures the property in a theoretical sense, but in practice we are fond of diagnostic messages and prefer compilation to always fail if $\neg\psi(p)$ holds.

There are several relevant veins of research in capture of properties by languages:

- Time- and space- complexity classes can be captured by restricted languages, an idea that goes back to the 1960s and has a rich literature (e.g., [21, 24, 16]). There is a kind of 'cheat' method, which involves a *clocked* programming language where every program comes with an attached statement such as "run me for at most $c|x|^k$ steps," with c and k constants; in this case termination in polynomial time is guaranteed. Then there are languages that capture complexity classes in a natural way. For example Neil Jones describes a family of languages with familiar constructs such as while, if and cons that capture complexity classes like PTIME by omitting selected constructs from the language [17].

- *Descriptive* or *Implicit computational complexity* studies restricted logics that capture complexity classes (e.g., [15, 19]). For example, polynomial time queries on ordered relational structures can be captured by first order logic augmented with a least fixpoint operator. Some such results translate easily into programming languages.

- *Program schemes* are restricted forms of recursion for which certain properties (e.g., termination) are decidable [7].

We may hope to design metalanguages that capture safety properties in a similar way— this is essentially the goal of the MetaML research programme [25, 28]. We can use the tools of computability theory to reason about when a language capturing a property might exist and what properties it might have.

In what follows we will consider properties $\psi(g)$ defined by arithmetical formulas as in the previous section. If a particular generator g satisfies a property $\psi(g)$, we say "$\psi(g)$ holds" or "g satisfies ψ".

Definition 4 (Capture). We say a restricted metalanguage $L \subset \mathbb{N}$ *captures* a property ψ when L is a decidable subset of generators, and

1. Every program $g \in L$ in the restricted metalanguage satisfies the property ψ:

$$\forall g \in L \,.\, \psi(g) \qquad (4)$$

2. For every (unrestricted) generator $g' \in \mathbb{N}$ such that g' satisfies ψ, there is some equivalent (restricted) generator $g \in L$ such that $\varphi_g = \varphi_{g'}$:

$$\forall g' \in \mathbb{N} \,.\, \psi(g') \rightarrow (\exists g \in L \,.\, \varphi_g = \varphi_{g'}) \qquad (5)$$

For example, if a metalanguage L captures the property "for all x, $\varphi_g(x)$ runs in $O(|x|^2)$ time", this means not only that every generator in L runs in quadratic time, but also that every computation that *can* run in quadratic time is expressible in L.

First we consider the problem of metalanguages capturing *functional* properties, such as always generating safe instances or always terminating.

Definition 5 (Functional property). A property $\psi(g)$ is *functional* when, equivalently:

1. ψ is defined solely in terms of termination and input-output behaviour;

2. ψ is an index set;

3. $(\varphi_g = \varphi_{g'}) \to (\psi(g) \leftrightarrow \psi(g'))$.

For example, whether a generator always produces safe instances is a functional property; whether it runs in quadratic time is not, and is considered a *non-functional* property. (The term *non-functional* is unfortunate but traditional in software engineering.)

One way to design a language capturing a property is to package programs together with proofs of that property, as in proof-carrying code [22] or Royer and Case's treatment of provably bounded programming systems [24, §4.3.1]. This yields a sufficient condition for the existence of a language capturing a property.

Proposition 6.4. *Let ψ be a property for which there is a sound proof system $\vdash_\psi$ with the following properties:*

1. *Checking whether a deduction $\mathcal{D}$ is a valid proof in the system $\vdash_\psi$ is decidable;*

2. *For every generator g satisfying ψ, there exists an equivalent g' such that $\varphi_g = \varphi_{g'}$ and there is a deduction in $\vdash_\psi$ proving $\psi(g')$. (This is strictly weaker than requiring the proof system to be* complete *for ψ.)*

Then there is a language capturing ψ.

Proof. We follow the proof-carrying code idea, designing a language whose every program is a pair $(g, \mathcal{D})$ where g is the generator and $\mathcal{D}$ is a nonexecutable payload containing a suitably encoded proof of $\psi(g)$ in the system $\vdash_\psi$. We define the language L to be only those $(g, \mathcal{D})$ where $\mathcal{D}$ is a valid deduction proving $\psi(g)$; this is a Δ_1^0 subset because of the premise (1).

We claim this language captures ψ in the sense of Defn. 4: Every generator in L clearly has the property ψ, due to soundness of the proof system; and every generator $g \in \mathbb{N}$ such that $\psi(g)$ holds has an equivalent program in L by the premise (2). □ □

Languages that require programmers to attach proofs suffer from so-called "technology adoption issues." Better perhaps to find a language that *implicitly* captures the property; such languages do not require programmers to explicitly write proofs. However, writing programs in languages that capture functional properties does have an *implicit* relationship to proofs: reworking a generator from a general-purpose language to a restricted language is tantamount to *proving* the property.

Theorem 6.2 (Capture is tantamount to proof). *Let ψ be a* functional *property of generators, and $L \subset \mathbb{N}$ a restricted language capturing ψ. Given a generator $g' \in \mathbb{N}$ satisfying ψ, the problem of transforming it into an equivalent generator $g \in L$ in the restricted language by means of (provably) semantics-preserving steps is at least as hard as finding a proof of $\psi(g')$.*

Proof. We assume we have already a proof that L captures ψ. At each step of transforming g' into g we can, by means of semantics-preserving steps, maintain a proof that the two versions are equivalent, so at the end of the process we have a proof that $\varphi_g = \varphi_{g'}$. Since L is decidable we can readily obtain a proof that $g \in L$ at the end of the process. We then have proofs of

1. $g \in L$ (g is in the restricted language)

2. $\forall g \in L . \psi(g)$ (from capture of ψ by L)

3. $\varphi_g = \varphi_{g'}$ (from semantics-preserving steps)

From (1) and (2) we obtain $\psi(g)$; from (3) and the premise that ψ is a functional property we obtain $\psi(g')$.

Therefore the problem of proving $\psi(g')$ is reducibile to transforming g' into the restricted language L by means of semantics-preserving steps. □ □

Related results on succinctness (Section 6.3) suggest that the reworked program may be as long as a proof of the property $\psi(g)$.

Corollary 6.1. *If ψ is an undecidable property, there can be no automated (computable) process for reworking generators into the restricted language.*

The practical implications of this are that writing generators in certain restricted languages is just as hard as proving safety properties, and may require arbitrary creativity. However, there may be an important social difference: proof construction can be intimidating for programmers, whereas programming in a restricted language can be a source of interesting puzzles requiring ingenious solution. There are intermediate solutions between proof-carrying code and implicit capture, where we design a language with some mix of explicit proof and implicit capture of the property. Type systems are a prime example: the programmer annotates a program with enough type information to make type safety easily provable. From a theoretical perspective programmers are constructing proofs of type safety relative to the decision procedure for the type system in the compiler; but it feels more intuitive than formal proof calculi.

6.2 When is capture possible?

So we may sometimes find languages that capture undecidable properties. In this section we give results on when such languages may or may not exist. There are some niches that can be carved out, though, for example the sufficient conditions of Proposition 6.4. We also know there are languages capturing any deterministic time and space bounds due to the existence of "clocked" programming systems where programs are annotated with resource bounds [24].

Here is a negative result. We show that arbitrarily hard functional properties cannot be captured by programming languages, by turning the tables and characterizing a property in terms of the language capturing it.

Proposition 6.5. *If there is a language capturing a functional property ψ, then $\psi \in \Sigma_3^0$.*

Proof. From the definition of capture (Defn. 4) and the key fact that ψ is functional (Defn. 5), we have the following

correspondence.

$$\psi(g) \leftrightarrow \exists g' \; . \; \underbrace{(\varphi_g = \varphi_{g'})}_{\Pi_2^0} \wedge \underbrace{(g' \in L)}_{\Delta_1^0} \quad (6)$$

The relation $\varphi_g = \varphi_{g'}$ is Π_2^0, so ψ is Σ_3^0. □ □

This implies we cannot capture in languages any functional property not definable in Σ_3^0, i.e., the whole span of the arithmetical hierarchy above Σ_3^0 is off-limits. However, this does not rule out the possibility of arbitrarily hard *non-functional* requirements being captured.

Here is a well-known fact that sets the stage for proving we cannot capture the property "every generator g always halts and produces a safe instance," i.e., total correctness of generators, in a language.

Proposition 6.6. *There is no language capturing the total computable functions, i.e., the property* $\mathrm{Tot}(p) \leftrightarrow \forall x \; . \; \varphi_p(x) \downarrow$.

Proof. Suppose $L \subset \mathbb{N}$ is a language capturing Tot; then every total computable function is expressible in L. We use diagonalization to construct a total computable function obviously not in L. Consider an enumeration $\{p_i\}_{i \in \mathbb{N}}$ of the programs in L; such an enumeration exists since L is decidable by Defn. 4. Consider the function

$$f(n) = 1 + \varphi_{p_n}(n)$$

Since $p_n \in L$, this program halts on every input. It is also total and computable, and is therefore expressible in L. Let k be the L-program that computes it. Then $\varphi_k(k) = 1 + \varphi_k(k)$, a contradiction since $z = 1 + z$ has no solution in the naturals. □ □

Proposition 6.7. *If there are at least two safe instances, there is no metalanguage capturing the property "g halts on every input and outputs a safe instance."*

Proof. Following the same style of diagonalization argument in Proposition 6.6, changing every safe instance on the diagonal. (The diagonalization would fail if there was only one safe instance). □

6.3 Succinctness

It is a well-studied phenomena that often when we move from one language to a more restricted version, some programs have to get larger — a loss of *succinctness*. The intuitive reasons for this are demonstrated by revisiting the language seen earlier capturing the property "halts on only finitely many inputs" by programs of the form:

$$f(x) = \begin{cases} c_1 & \text{when } x = x_1 \\ c_2 & \text{when } x = x_2 \\ \vdots & \vdots \\ c_n & \text{when } x = x_n \\ \uparrow & \text{otherwise} \end{cases}$$

Consider a program written in a general-purpose language that, given input $x < 10^{100}$, outputs $x + 1$, otherwise diverges. This halts only on finitely many inputs, and can be implemented in a few lines of C code. The corresponding "lookup table" program in the above language is too long to fit into the observable universe. (If we scale down the exponent we can get the more practical "too long to fit into

any existing computer.") An underlying cause is that we can pose problems easily solveable in an unrestricted language, but "look random" to the restricted language and one cannot do any better than decomposing it into a large number of cases, as in the above example. Problems that can be decomposed only into an infinite number of cases are inexpressible.

Let us write $|\cdot|$ for the length of a program; such measures are usually required to satisfy the very weak axioms of Blum [2]. Counting bits of the representation is satisfactory.

Definition 6 (Computably succint). Suppose L and L' are two languages. We say L is *computably succinct* relative to L' if for every program $p \in L$ for which there is a functionally equivalent program in L', there is a $p' \in L'$ such that

$$|p'| \leq f(|p|)$$

where f is some computable function.

Saying one language is not computably succinct relative to another is a strong statement; for example, it implies that the loss in succinctness cannot be bounded by your favourite fast-growing computable function, for example the 'power tower':

$$T(k) = \underbrace{10^{10^{10^{10^{\cdots^{10}}}}}}_{k}$$

where $T(1) = 10$, $T(2) = 100$, $T(3) = 10^{100}$, and so on. (Cosmologists suppose the number of atoms in the observable universe is less than $T(3)$.)

All the Turing-complete languages are computably succinct relative to one another, and in fact the interesting ones are all within an additive constant of one another; this follows from a 'two-part code' construction [18, §2.1].

The tradeoff between succinctness and power of languages has been explored rather exhaustively, and we summarize only some highlights here. For details Royer and Case [24] is recommended for the subrecursive languages perspective, and Chapter 7 of Li and Vitányi [18] is recommended for the Kolmogorov complexity viewpoint (concerned primarily with *instance complexity* rather than computable functions, but still interesting).

Results on succinctness of languages fall loosely into three classes.

1. Loss in succinctness when moving from one language to more restrictive language. The general flavour of such results is that if you restrict a language that can compute at least polynomial-time functions in a sufficiently strong way, the resulting loss in succinctness cannot be bounded by any computable function. The first such result was achieved by Blum [2], and similar results are abundant [24].

2. Losses in succinctness between two languages of the same expressive power. For example, most introductory theory classes cover the fact that nondeterministic finite automata (NFAs) can be converted to DFAs with at most an exponential expansion in size. Note though, that both capture the regular languages.

3. Losses in succinctness when moving back and forth between two languages of the same expressive power.

For instance Hartmanis gives an example of two different languages capturing PTIME, neither of which is computably succinct relative to the other [11].

With respect to languages capturing properties, we can make the following observation.

Theorem 6.3. *If a language $L \subset \mathbb{N}$ is computably succinct, then any functional property ψ it captures is Π_2^0.*

Proof. Suppose L captures a property ψ. As before we turn the tables and define ψ in terms of the language:

$$\psi(g) \leftrightarrow \exists g' . (\varphi_g = \varphi_{g'}) \wedge (g' \in L) \tag{7}$$

Since L is computably succinct, there is some computable f such that

$$\psi(g) \leftrightarrow \exists g' \leq f(g) . \underbrace{(\varphi_g = \varphi_{g'}) \wedge (g' \in L)}_{\Pi_2^0} \tag{8}$$

The addition of $\leq f(g)$ turns the existential quantifier into a *bounded* quantifier; therefore $\psi(g) \in \Pi_2^0$. $\square$ $\square$

7. CHASING PROPERTIES

When our attempts to capture a property by a language fail, there remains the possibility of approximating the property. For example, although total correctness of generators cannot be captured by a language, we can choose some restricted language as a starting point and gradually build it up so as to increase its power and succinctness.

We can model this process by a chain of languages $L_0 \subset L_1 \subset L_2 \cdots$ converging towards the desired safety property ψ. Note that given L_i we can always find a set $L_i \subset L_{i+1} \subset \psi$, for example, by adding a finite number of special cases to the test for L_i. But in practice we find a slightly stronger proof system, capture some common patterns, and so forth. We call this *chasing* a property.

Let us write $C(L_i)$ for the length of the shortest program deciding the set L_i.

Proposition 7.1. *Let ψ be an undecidable property of generators, and $L_0 \subset L_1 \subset L_2 \cdots$ a countable (but not c.e.) sequence of languages, each decidable, such that:*

1. *Each language L_i has the property $\psi(p)$ for every $p \in L_i$;*

2. *$\bigcup_{i \in \mathbb{N}} L_i = \psi$, i.e., in the limit we recover exactly the language ψ;*

Then, $\lim_{i \to \infty} C(L_i) = \infty$, i.e., the length of program required to describe the language L_i diverges.

Proof. If $C(L_i)$ did not diverge, we would have a finite program deciding the undecidable property ψ, a contradiction. $\square$

So, in essence the best one can hope for is to capture the perfect metalanguage in the limit by approximating it from below with ever-more-complicated languages. This is the conservative approach.

An optimistic approach is to start with the universal language $L_0 = \mathbb{N}$, which fails the safety property, and find languages $L_0 \supset L_1 \supset L_2 \supset \cdots$ that gradually winnow out the unsafe cases and converge toward ψ, with ever-more-complicated languages.

These two approaches — approximating ψ conservatively from below or optimistically from above — appear to represent irreconcilable approaches to metaprogramming language design with no middle ground. I would propose Meta-ML as representative of the first, and C++ as emblematic of the second, particularly the $C(L_i) \to \infty$ part.

8. WHEN IS "GOING META" A JUMP OF TURING DEGREE?

When we make the shift in perspective from programming to meta-programming — "going meta," as it were, what happens to the difficulty of verification problems? How does the hardness of these problems relate:

1. Does a program p satisfy a property ψ?

2. Does every program generated by a metaprogram g satisfy a property ψ?

These questions suggest a connection to the *jump operator* in computability theory. We define an operator on properties in the following way:

Definition 7 (Meta-jump). Let $\psi(p)$ be a property of programs. The meta-jump of ψ is the property $\psi^\dagger$ given by:

$$\psi^\dagger(g) \Leftrightarrow \forall y . \underbrace{(\varphi_g(y) \downarrow)}_{g \text{ halts}} \rightarrow \underbrace{\psi(\varphi_g(y))}_{\psi \text{ holds for its output}}$$

i.e., we shift from the property of a single program $\psi(p)$ to the property $\psi(p)$ holding for every program p produced by the metaprogram g.

We might expect $\psi^\dagger$ to be much harder to decide than ψ, but this appears not to be the case.

Let us use formula syntax to roughly characterize the action of the meta-jump operator. The meta-jump adds a $\forall$ quantifier at the start of a formula. Based on this syntactic characterization we can immediately get:

Proposition 8.1. *The meta-jump operator satisfies the following rules:*

1. *If $\psi \in \Sigma_k^0$ then $\psi^\dagger \in \Pi_{k+1}^0$.*

2. *If $\psi \in \Pi_k^0$ then $\psi^\dagger \in \Pi_k^0$.*

Note that (2) implies that the meta-jump has no effect on the placement of formulas that are Π_k^0 but not Δ_k^0. The condition (1) states that it is possible for Σ_k^0 properties to become harder under the meta-jump, but this is not necessary because of the inclusion $\Delta_k^0 \subset \Sigma_k^0 \subset \Pi_{k+1}^0$. (Note that this characterization is very rough and does not consider finer degrees of undecidability such as m-degrees).

It turns out that the safety properties usually talked about in connection with metaprogramming and program generation are all in the form Π_k^0: halting, always producing a safe instance, both halting and producing a safe instance, for example. We usually conceive of a safety property as preventing certain some (usually infinite) set of *failure conditions* from occurring. We can enumerate the failure conditions and check them one by one; if a failure condition occurs we know the safety property fails. This style of safety condition is always Π_1^0 relative to some relation, and therefore there is no jump in the level of the arithmetic hierarchy when we 'go meta.'

9. CONCLUSIONS

We set out to characterize tradeoffs in metaprogramming, in retrospect a somewhat presumptuous goal, since the problems turn out to be unexpectedly deep and requiring further investigation. The computability approach is very useful in providing quick and coarse characterizations of tradeoffs.

One result that remains elusive is a characterization of the tradeoff between safety and expressive power involved in metalanguages having the property "every instance is type-safe." The primary challenge is that in a general-purpose metalanguage one can always satisfy the capture properties vacuously by introducing layers of interpretation, and a way to disallow these 'cheats' is not apparent without dropping to a subrecursive metalanguage. Partial correctness of metaprograms is a 'run-time manageable' property, in the sense that one can detect bad outputs before they happen.

Despite these difficulties a rough picture of the major tradeoffs in 'safe metaprogramming' emerges.

1. Total correctness of generators cannot be captured. If termination of the generator is required, one must pick a suitable subrecursive language and try to recoup succinctness and power as needed by building in provably safe 'escapes' back up to more powerful classes of generators.

2. Whether there are languages capturing partial correctness of metaprograms in a useful way is uncertain. However, we can say that if such languages exist, and are not vacuous, then we expect because of Theorem 6.2 that reworking generators into such languages is as hard as *proving* partial correctness. However for social reasons 'capture' is more desirable, if it can be achieved.

3. When we restrict the power of metalanguages, we lose the ability to compress certain kinds of motifs and patterns. Whether this causes a practical (rather than theoretical) loss in succinctness depends on the characteristics of the problem domain (cf. [29]).

To summarize, we can say that there is choice between two strategies when designing a metaprogramming language. One can give safety primacy, start with a restricted language, and try to build up the complexity of the language so as to recapture lost power and succinctness. Or, one can give power primacy, start with a universal language, and try to build up the complexity of the language so as to capture necessary safety properties. These represent fundamentally different attitudes toward metaprogramming.

Acknowledgments

10. REFERENCES

[1] A. F. Blackwell, C. Britton, A. Cox, T. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roes, A. Wong, and R. M. Young. Cognitive dimensions of notations: Design tools for cognitive technology. *Lecture Notes in Computer Science*, 2117:325–341, 2001.

[2] M. Blum. On the size of machines. *Information and Control*, 11(3):257–265, Sept. 1967.

[3] B. M. Brosgol. An implementation of ECL data types. *SIGPLAN Not.*, 6(12):87–95, 1971.

[4] P. E. Ceruzzi. *A History of Modern Computing*. MIT Press, Cambridge, Mass., 1998.

[5] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and abhi shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 792–801, New York, NY, USA, 2002. ACM Press.

[6] B. S. Cooper. *Computability Theory*. Chapman & Hall/CRC mathematics, 2003.

[7] B. Courcelle. Recursive applicative program schemes. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 459–492. 1990.

[8] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[9] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.

[10] T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of the HCI'89 Conference on People and Computers V*, Cognitive Ergonomics, pages 443–460, 1989.

[11] J. Hartmanis. On Gödel speed-up and succinctness of language representations. *Theoretical Computer Science*, 26:335–342, 1983.

[12] G. H. Holloway. Interpreter/compiler integration in ECL. *SIGPLAN Not.*, 6(12):129–134, 1971.

[13] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite-automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[14] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with SafeGen. In *Generative Programming and Component Engineering (GPCE)*, Sept. 2005.

[15] N. Immerman. *Descriptive complexity*. Graduate Texts in Computer Science. Springer-Verlag, New York, 1999.

[16] N. D. Jones. *Computability and Complexity from a Programming Perspective*. Foundations of Computing. MIT Press, Boston, London, 1 edition, 1997.

[17] N. D. Jones. LOGSPACE and PTIME characterized by programming languages. *Theoretical Computer Science*, 228:151–174, 1999.

[18] M. Li and P. Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag, New York, 2nd edition, 1997.

[19] L. Libkin. *Elements of finite model theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin, 2004.

[20] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4):50–59, 1974.

[21] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22nd national conference of the ACM*, pages 465–469. ACM Press, 1967.

[22] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM

Press.

[23] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[24] J. S. Royer and J. Case. *Subrecursive Programming Systems: Complexity and Succinctness.* Birkhauser, 1994.

[25] T. Sheard and J. Hook. Type safe meta-programming. Unpublished manuscript, Oregon Graduate Institute, November 1994.

[26] H. Siegal and J. Painter. The use of generators in TAC. In *ACM '59: Preprints of papers presented at the 14th national meeting of the Association for Computing Machinery*, pages 1–4. ACM Press, 1959.

[27] R. I. Soare. Computability and recursion. *The Bulletin of Symbolic Logic*, 2(3):284–321, Sept. 1996.

[28] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, Oct. 2000.

[29] T. L. Veldhuizen. Software libraries and their reuse: Entropy, Kolmogorov complexity, and Zipf's law. In *OOPSLA 2005 Workshop on Library-Centric Software Design (LCSD'05)*, 2005. arXiv:cs.SE/0508023.

[30] B. Wegbreit. An overview of the ECL programming system. *SIGPLAN Not.*, 6(12):26–28, 1971.