Tradeoffs in Metaprogramming*

Todd Veldhuizen
Open Systems Laboratory
Indiana University Bloomington

January 10, 2006

*This paper on which this talk was based may be found at `http://arxiv.org/abs/cs/0512065`

Two traditions of generics

- Safe (but restricted expressiveness)
  - ⋆ Alphard, CLU, ML, Haskell, Java, etc.
  - ⋆ A parade of language features to increase expressiveness: parametric polymorphism, F-bounded polymorphism, type classes, ...
- Unsafe (but very expressive)
  - ⋆ EL1 [Wegbreit(1974), Holloway(1971)] — arbitrary expressions in type position, compiler had built-in partial evaluator for these.
  - ⋆ C++
  - ⋆ A parade of language features to increase safety: signatures, concepts

Can we have safe and expressive at the same time?

Metalanguages

A metalanguage is a special-purpose language for generating or transforming programs.

Stretch the definition to encompass languages for:

- Metaprogramming: YACC, TXL, Stratego, ...
- Code generation: SafeGen, MetaML, C++ Templates, ...
- Abstraction: Macros, generics, class definition syntax, ... (Programming languages are assemblages of metalanguages.)

Metalanguages

(a) When is it possible to design metalanguages that...

- guarantee well-formedness?
- guarantee type safety?
- preserve semantics of the object language?
- always terminate?

and, (b) can we achieve the above without sacrificing expressive power?

You cannot have it both ways

Can I ...

1. Make C++ templates always halt without sacrificing expressive power?

2. Put a type system on JavaFront so that it only allows semantics-preserving transformations, but without sacrificing expressive power?

3. Design a metalanguage for specifying compiler optimizations that permits any transformation that can be done in polynomial time, without making some transformations ridiculously hard to express?

Answer key: (1) No. (2) No. (3) No.

Tradeoffs

In designing a metalanguage, one must trade off various *facets*:

- Expressive power
- Safety properties
- Succinctness (do trivial metaprograms require vast amounts of code to express?)
- Computational complexity, etc. etc.

My belief: we need to understand these tradeoffs.

Why do we need special metalanguages?

In a universal language (Turing-complete), nontrivial properties are undecidable (Rice's theorem).

Cannot write a procedure that will decide whether a metaprogram

- emits only well-formed or typeable outputs;
- preserves semantics;
- terminates;
- runs in a given time or space bound (e.g., PTIME).

Capture

But sometimes we can find a programming language that "captures" a property.

Example. This is a highly undecidable property:

$$\mathrm{Fin} \equiv \text{The program terminates for at most a finite number of inputs.}$$

Undecidable $\Rightarrow$ you can't write a procedure that decides whether a Java program satisfies the property.

Capture

But we can "capture" the property with a restricted language: only allow programs of the form:

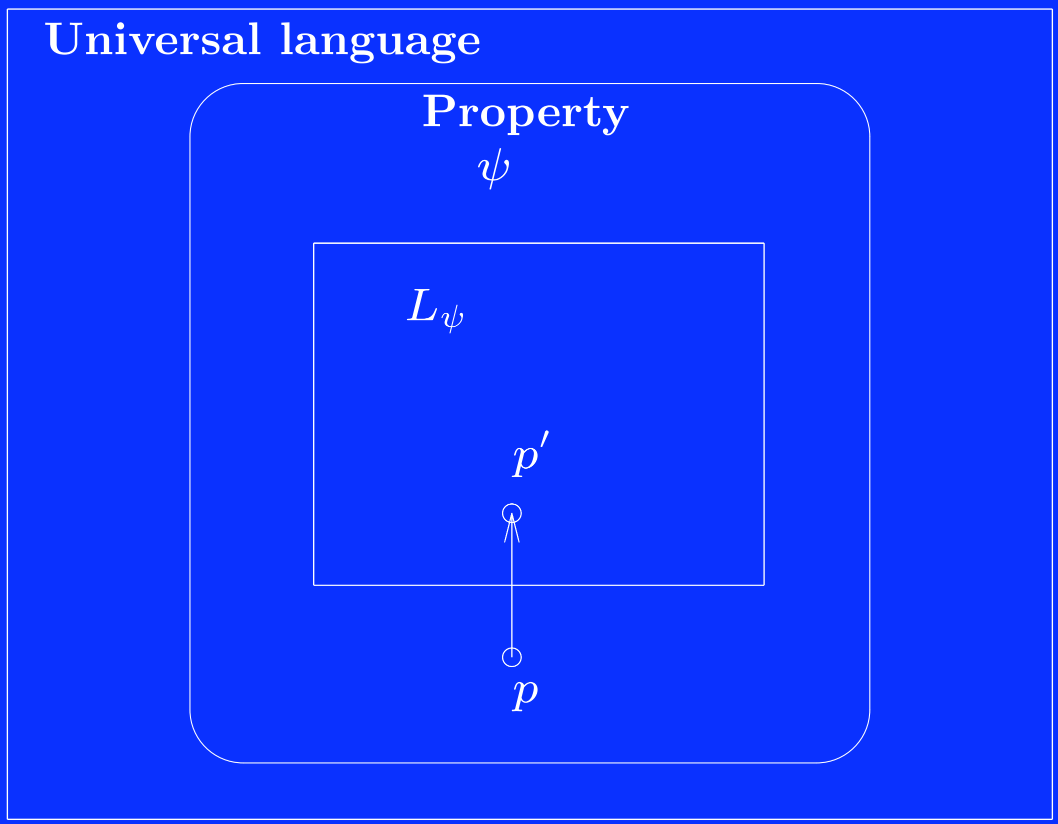
$$f(x) = \begin{cases} c_1 & \text{when } x = x_1 \\ c_2 & \text{when } x = x_2 \\ \vdots & \vdots \\ c_n & \text{when } x = x_n \\ \uparrow & \text{otherwise} \end{cases}$$

This sublanguage is easy to recognize.

Capture

Say a restricted metalanguage *captures* a property ψ when:

1. Every program in the restricted language satisfies ψ;
2. Every program (in a general-purpose language) that has the property ψ is equivalent to some program in the restricted language.



This diagram illustrates the idea of "capture." Imagine this as a kind of Venn-diagram of sets of programs. We have some universal language. Inside this language is a set of programs with some desirable property ψ. *Although* ψ *is undecidable, we can sometimes find a sublanguage* L_ψ *that* is *decidable, such that for any program* p *that models* ψ, *there is an equivalent program* $p' \in L_\psi$.

Languages capturing properties

Classical examples:

- Regular expressions capture computations that can be performed by DFAs.
- Time- and space-complexity classes can be captured by programming languages (e.g., LOGSPACE, PTIME, EXPSPACE).

Metalanguages capturing properties

When can we find metalanguages capturing useful properties (well-formedness, typeable, semantics-preserving, ...)?

Computability theory is good at answering such questions.

The Arithmetical Hierarchy

Introduced by Kleene [Kleene(1950)] to classify noncomputable sets.

$$
\begin{tikzcd}
 & \vdots \arrow[dr, dash] & & \vdots \arrow[dl, dash] & \\
 & & \Delta^0_3 \arrow[dl, dash] \arrow[dr, dash] & & \\
 & \Sigma^0_2 \arrow[dr, dash] & & \Pi^0_2 \arrow[dl, dash] & \\
 & & \Delta^0_2 \arrow[dl, dash] \arrow[dr, dash] & & \\
\text{c.e.} & \Sigma^0_1 \arrow[dr, dash] & & \Pi^0_1 \arrow[dl, dash] & \text{co-c.e.} \\
 & & \Delta^0_1 & \text{decidable} &
\end{tikzcd}
$$

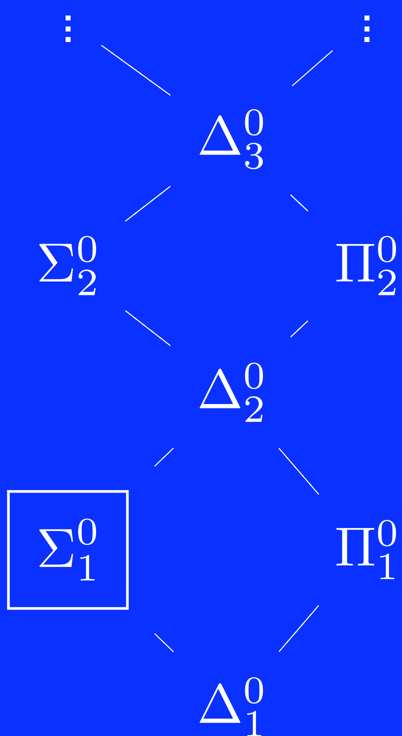
Class Δ^0_1

$$\begin{tikzcd}[arrows=dash]
\vdots \arrow[dr] & & \vdots \arrow[dl] \\
& \Delta^0_3 \arrow[dl] \arrow[dr] & \\
\Sigma^0_2 \arrow[dr] & & \Pi^0_2 \arrow[dl] \\
& \Delta^0_2 \arrow[dl] \arrow[dr] & \\
\Sigma^0_1 \arrow[dr] & & \Pi^0_1 \arrow[dl] \\
& \boxed{\Delta^0_1} &
\end{tikzcd}$$

Decidable sets.

Rice's theorem: there are no nontrivial program properties in this class.

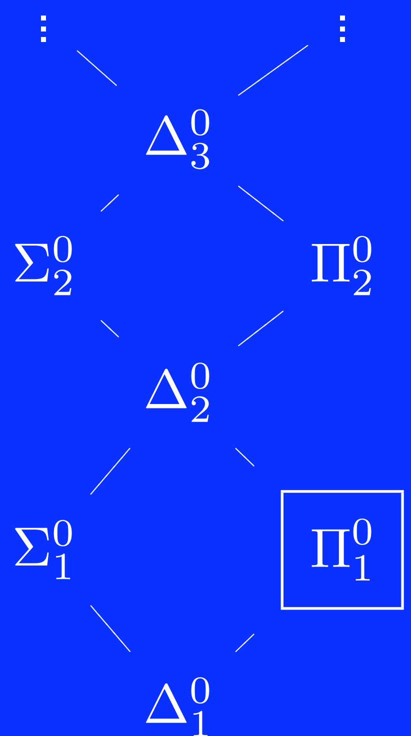
Class Σ_1^0



Computably enumerable (c.e.) sets (aka r.e.)
Sets with effective proof calculi
Sets with an effective inductive definition
Sets with a finite axiomatization

Properties living here: (not very interesting ones)
"Metaprogram halts for at least one input."

Class Π^0_1



Co-Computably enumerable (co-c.e.) sets (aka r.e.)
Sets with an effective coinductive definition

Properties living here:
Partial correctness properties: "If it halts, the metaprogram produces a well-formed/typeable instance."

Class Σ^0_2

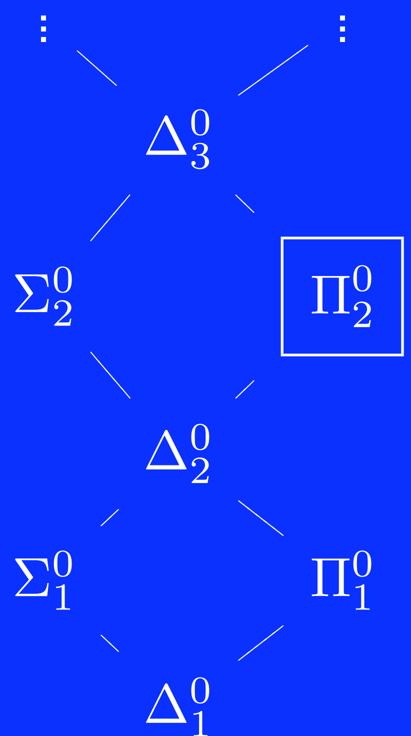
$$
\begin{tikzcd}
\vdots & & \vdots \\
& \Delta^0_3 \arrow[ul, dash] \arrow[ur, dash] & \\
\boxed{\Sigma^0_2} \arrow[ur, dash] & & \Pi^0_2 \arrow[ul, dash] \\
& \Delta^0_2 \arrow[ul, dash] \arrow[ur, dash] & \\
\Sigma^0_1 \arrow[ur, dash] & & \Pi^0_1 \arrow[ul, dash] \\
& \Delta^0_1 \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}
$$

Sets that are c.e. relative to a Π^0_1 oracle

Computational complexity classes live here: "Metaprogram runs in $O(n^2)$ time." Fin is in this class.

Independence issues arise: e.g., there are programs whose running time is independent of the axioms of set theory [Hartmanis and Hopcroft(1976)].

Class Π^0_2



Sets that are co-c.e. relative to a Σ^0_1 oracle

Properties living here:
"Metaprogram always halts and produces a well-formed/type-safe instance."
"Metaprogram performs only semantics-preserving transformations."

Succinctness

Sometimes when we translate programs into a restricted metalanguage, the size explodes. (e.g., DNF for boolean formulas: exponential blowup)

Sometimes this explosion in program length cannot be bounded by any computable function:

- e.g. restricted languages that are total (always terminate)
- Noncomputable blowup $\Rightarrow$ there are programs that require 10^{100} times more code to express in the restricted language.
- Whether we care is another matter (maybe they are not interesting programs).

Succinctness

(Defn) Succinct capture $\equiv$ capturing a property without noncomputable blowup in program size.

(Thm 6.3) It is impossible to succinctly capture properties not in Π^0_2.

$\Rightarrow$ All languages capturing complexity classes (Σ^0_2) have noncomputable blowup.

Silver lining: partial correctness properties are in $\Pi^0_1 \subset \Pi^0_2$.

Negative results on capture

There are no metalanguages capturing:

- (Prop 6.6) Metaprogram always halts.
- (Prop 6.7) Metaprogram always halts and produces a typeable/well-formed instance (total correctness).
- Metaprogram performs only semantics-preserving transformations.

Positive results on capture

There *is* a metalanguage capturing partial correctness:

- If the metaprogram halts, it produces a typesafe/well-formed instance

But we might not like it:

- Run the metaprogram on its input.
- Check the output. If it's bad, replace it with something safe.

i.e. no error messages. Π^0_1 properties seem to be a quagmire.

Capture is tantamount to proof

L = a general-purpose language

L_ψ = a restricted language capturing ψ

(Thm 6.2) Transforming a program p from L into an equivalent program in L_ψ via semantics-preserving steps is equivalent to *proving* that $p \models \psi$.

If the property is nontrivial, there can be no automated process that rewrites programs into the restricted language.

Heisenberg-like effects outside Σ^0_1

Σ^0_1 is the only class where we have finite axiomatizations $\equiv$ complete proof calculi.

Above this class we can only have partial axiomatizations (incomplete proof calculi).

Consequence: If ψ is a property not in Σ^0_1, and L_ψ captures ψ, there will always be programs p that are equivalent to some $p' \in L_\psi$ but we cannot prove that $p \sim p'$ or $p \models \psi$.

Chasing properties with languages

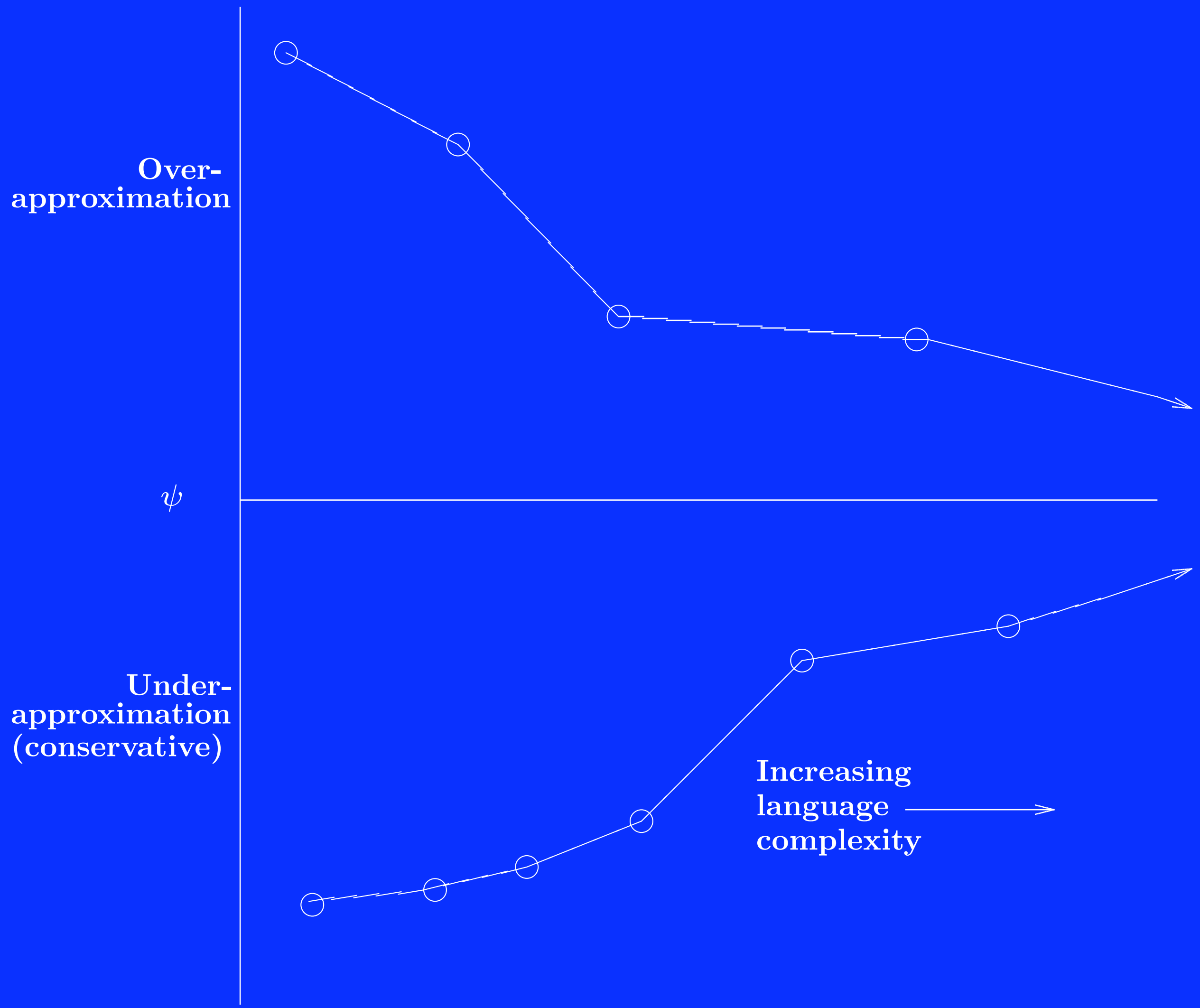
We know that some properties cannot be captured (e.g., total correctness).

But, every 'functional' property is the limit of a sequence of languages with ever-increasing complexity:

$$L_0 \subset L_1 \subset L_2 \subset \cdots$$

with $\lim_{i \to \infty} L_i = \psi$
and L_{i+1} requires a longer interpreter than L_i.

Two fundamentally opposed approaches to language design.


Over-approximation
ψ
Under-approximation (conservative)
Increasing language complexity

Conclusions

- Interesting properties of metaprograms are undecidable.
- But we can sometimes *capture* properties with restricted languages (e.g. partial correctness of metalanguages).
- If capture is not possible (e.g. total correctness), we can *chase* properties: a parade of language features, either
  - ⋆ Giving safety primacy, and recouping expressive power as language complexity $\to \infty$ (e.g., Haskell generics)
  - ⋆ Giving expressive power primacy, and recouping safety as language complexity $\to \infty$ (e.g., C++ generics)

Meta-conclusion

- Computability theory has useful explanatory power for tradeoffs in metalanguage design.

References

[Hartmanis and Hopcroft(1976)] J. Hartmanis and J. E. Hopcroft. Independence results in computer science. *SIGACT News*, 8(4):13–24, 1976. ISSN 0163-5700.

[Holloway(1971)] G. H. Holloway. Interpreter/compiler integration in ECL. *SIGPLAN Not.*, 6(12):129–134, 1971. ISSN 0362-1340.

[Kleene(1950)] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, Princeton, New Jersey, 1950.

[Wegbreit(1974)] B. Wegbreit. The treatment of data types in el1. *Commun. ACM*, 17(5):251–264, 1974. ISSN 0001-0782.