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.)
(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).
But sometimes we can find a programming language that “captures” a property.

Example. This is a highly undecidable property:

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

Undecidable \( \Rightarrow \) you can’t write a procedure that decides whether a Java program satisfies the property.
捕获

但我们可以“捕获”该性质，使用一个限制性语言：只允许程序具有以下形式：

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

这种子语言很容易识别。
Capture

Say a restricted metalanguage *captures* a property $\psi$ when:

1. Every program in the restricted language satisfies $\psi$;

2. Every program (in a general-purpose language) that has the property $\psi$ 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 \( \psi \). Although \( \psi \) is undecidable, we can sometimes find a sublanguage \( L_\psi \) that is decidable, such that for any program \( p \) that models \( \psi \), 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 \cite{Kleene(1950)} to classify noncomputable sets.
Class $\Delta^0_1$

Decidable sets.

Rice's theorem: there are no nontrivial program properties in this class.
Class $\Sigma_1^0$

- $\Delta_3^0$: Computably enumerable (c.e.) sets (aka r.e.)
- $\Sigma_2^0$: Sets with effective proof calculi
- $\Pi_2^0$: Sets with an effective inductive definition
- $\Delta_2^0$: Sets with a finite axiomatization
- $\Sigma_1^0$: Properties living here: (not very interesting ones)
- $\Pi_1^0$: "Metaprogram halts for at least one input."
- $\Delta_1^0$
Class $\Pi_1^0$

- 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 $\Sigma^0_2$

Sets that are c.e. relative to a $\Pi^0_1$ 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** $\Pi_2^0$

Sets that are co-c.e. relative to a $\Sigma_1^0$ 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).
(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 $\Pi_2^0$.

$\Rightarrow$ All languages capturing complexity classes $(\Sigma_2^0)$ have noncomputable blowup.

Silver lining: partial correctness properties are in $\Pi_1^0 \subset \Pi_2^0$. 
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. $\Pi^0_1$ properties seem to be a quagmire.
Capture is tantamount to proof

\[ L = \text{a general-purpose language} \]
\[ L_\psi = \text{a restricted language capturing } \psi \]

(Thm 6.2) Transforming a program \( p \) from \( L \) into an equivalent program in \( L_\psi \) 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 $\Sigma_1^0$

$\Sigma_1^0$ 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 $\psi$ is a property not in $\Sigma_1^0$, and $L_\psi$ captures $\psi$, 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.
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 $\rightarrow \infty$ (e.g., Haskell generics)
  
  ★ Giving expressive power primacy, and recouping safety as language complexity $\rightarrow \infty$ (e.g., C++ generics)
Meta-conclusion

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


