General Purpose Languages Should be Metalanguages

Jeremy G. Siek
University of Colorado at Boulder
General Purpose vs. Domain Specific

"Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things." – The Next 700 Programming Languages by P.J. Landin

General Purpose = things in terms of other things

Domain Specific = basic set of given things

ORTHOGONALITY
Domain-Specific Embedded Languages

- Physics
- Linear Algebra
- RegEx
- Encryption Algorithms

General Purpose Language
Domain-Specific Embedded Languages

- Nice separation of concerns:
  - Programming language and compiler experts build the general purpose language.
  - Domain experts build the DSELS (and the not general purpose features!)
- Interoperability between DSELS.
- DSEL leverages underlying language.
- DSEL provides high productivity for the domain.

<table>
<thead>
<tr>
<th>Physics</th>
<th>Linear Algebra</th>
<th>RegEx</th>
<th>Encryption</th>
</tr>
</thead>
</table>

General Purpose Language
Functions and beta reduction are a typical example of a general purpose feature:

\[(\lambda x. e_1) e_2 \rightarrow [x \mapsto e_2]e_1\]

Many language semantics factor out the basic set of primitives by parameterizing over a set of operations and their interpretation.

\[
op \in \text{Ops} \\
\delta \in \text{Value}^n \rightarrow \text{Expr} \\
op(v_1, \ldots, v_n) \rightarrow \delta(op, v_1, \ldots, v_n)
\]
Need More than Ops

Example: can’t express waiting for a condition variable.

```c
#define wait(lock, pred)\  
  while (!(pred))    
  lock.wait()
```

In this case I had to go outside the language (C) and use a preprocessor.
Evolution of DSELs

- Across many languages, programmers have found tricks for encoding DSELs
  - Haskell: type classes
  - Scala: implicits, operator overloading, generics
  - C++: template tricks, operator overloading
  - Scheme: macro tricks
  - Ruby: hash literals, blocks, etc.
Pitfalls of DSELs

- DSELs are *leaky abstractions*
- Syntax is not quite right
- Infamous compile-time error messages
- Abstraction layer becomes transparent during debugging
- Limited optimization
Syntax is Not Quite Right

Example from the Boost Spirit Library.

Instead of writing this:

```cpp
  double | (',' | double)*
```

one has to write this:

```cpp
  double_ >> *(char_(',')) >> double_)
```
Syntax is Not Quite Right

Example from Haskell/DB.

Instead of writing this:

```sql
SELECT X.Name, X.Mark
FROM Students As X, Students As Y
WHERE X.Mark = Y.Mark
    AND X.Name <> Y.Name
```

one has to write this:

```haskell
do{ x <- table students
    ; y <- table students
    ; restrict (x!mark .==. y!mark)
    ; restrict (x!name .<> y!name)
    ; project (name = x!name, grade = x!grade)
}
```
Error Messages are Generic

It’s great that we can perform domain-specific checking

```
quantity<length> L = 2.0*meters;
quantity<energy> E = kg*pow<2>(L/seconds);
cout << L+E << endl;
```

But wouldn’t it be even better if the error messages were also domain specific!

```
error: no match for 'operator+' in 'L + E'
```
Error Messages Expose Internals

Result of a typo when using Boost.Spirit:

```
rule.hpp: In member function 'rule<Iterator, T1, T2, T3>& rule<Iterator, T1, T2, T3>::operator=(const Expr&)':
calc1.cpp:37: instantiated from 'client::calculator<Iterator>::calculator()'
calc1.cpp:78: instantiated from here
rule.hpp:176: error: no matching function for call to 'assertion_failed(mpl_::failed************ (rule<Iterator, T1, T2, T3>::operator=(const Expr&))
...
```
Debugging

- Most debuggers treat DSELs like any other code.
- Makes it hard to visualize the domain-specific runtime representations.
- For example, when debugging an application that uses a linked list class, I don’t want to see the link pointers, just the sequence of elements.

THIS

NOT THIS
Limited Optimization

As a student, I built a DSEL for linear algebra: the Matrix Template Library.

It could optimize one statement at a time, but not across statements.

\[
x = A * p;
y = \text{trans}(A) * q;
\]

For now, I’ve switched back to building a from-scratch DSL: the Build to Order BLAS.
Holy Grail: A Great Metalanguage

- Overarching principle: user-defined abstractions should be *first class* in a broad sense.
- User-defined abstractions should have the same rights and privileges as built-in abstractions.
- Syntax
- Static Error Checking
- Runtime Debugging
- Performance Optimization
Syntax Matters

- Despite what some theoreticians say, syntax really does matter.
- Consider training time for domain experts, not programming experts.
- Two approaches to extensible syntax:
  - Extensible/modular parsers
  - User interfaces for directly editing and displaying ASTs
Extensible Parsers

- Cocke-Younger-Kasami (CYK), aka chart parsers, used by the Isabelle proof assistant.
- Scannerless Generalized LR Parsers, used in the Syntax Definition Formalism (SDF).
- Camlp4 (algorithm?)
Syntax definitions are integrated in the proof language, like notational definitions in math.

```plaintext
syntax
"@PROD" :: [idt, i, i] => i  
"@->" :: [i, i] => i
translations
"PROD x:A. B" => "Pi(A, %x. B)"
"A -> B"    => "Pi(A, _K(B))"
```

Ambiguity still a performance problem. Mixfix notation removed from the Isabelle Tutorial.

Suggestion: type check during parsing. Probably requires some ground rules on the grammars.
Tower of Babel?

- Will unlimited syntax extensibility lead to the Tower of Babel?
- No, we have cultural mechanisms for establishing conventions and standards.
Direct AST Manipulation

- As in Intentional Programming and JetBrains Meta Programming System
- The Ultimate Solution?
- Challenges to this approach:
  - Large infrastructure investment
  - Will we ever ween ourselves of programs as strings of characters?
Static Error Checking

- Domain-specific static checking
- Static type safety for program generators

```
meta program          object program
TYPE CHECK            TYPE SAFE
```
Domain-Specific Static Checks

- External type system extension
  - JavaCOP, Millstein et al.
  - *Scripting the Type Inference Process*, Heeren et al.
  - Polyglot

- Internal encodings
  - Phantom types, dependent types
  - C++ template tricks
  - *Static Computation and Reflection*, Ronald Garcia
template <typename T, int length, int mass, int time>
struct quantity {
    T v;
    ...
};

template <typename T, int length, int mass, int time>
quantity<T,length,mass,time>
operator+(quantity<T,length,mass,time>& lhs,
            quantity<T,length,mass,time>& rhs);

template <typename T, int length1, int length2,
          int mass1, int mass2, int time1, int time2>
quantity<T,length1+length2,mass1+mass2,time1+time2>
operator*(quantity<T,length1,mass1,time1>& lhs,
            quantity<T,length2,mass2,time2>& rhs);
Type-Reflective MP

In general, the metalanguage should be able to:

- query the object language type checker
- reflect on types, build type exprs, build classes
- i.e., treat types as data

Static Safety for Generators

Metaprograms lower DSELs into object programs
Debugging type errors in the generated code is unpleasant
Shipping metaprograms that generate ill-typed code is even worse
Better to catch the errors in the generator.

e.g., MetaML of Taha and Sheard
Static Safety for Reflective Generators?

\[(\lambda T : \text{type.} (\lambda x : \sim T.x))\]
Static Safety for Reflective Generators?

\((\lambda T : \text{type.}(\lambda x : \sim T.x)) : \Pi T : \text{type.}\text{code}(T \rightarrow T)\)

We’re exploring whether dependent types can deliver static safety for reflective metaprograms.
Incremental Type Checking

- Dependent types are a heavy-weight solution
- A light-weight partial solution is to perform incremental type checking
- That is, during meta evaluation, type check code fragments to the extent to which the types are known.
- Report an error as soon as a code fragment becomes ill typed.
Debugging

- Debuggers should provide specialized views for DSELs
- Some glimmers of hope in DDD and Eclipse
- *Debugging Domain-Specific Languages in Eclipse*, Wu et al.
DSEL Optimization

- Low hanging fruit: local rewrites. Mostly picked.
- But what about non-local optimizations?
- How to interoperate with the general purpose optimizer and static analyses?
  - Detailed API for compiler internals. (bad)
  - Declarative specifications. (good)
- There are big opportunities for optimization based on invariants established by DSLs.
DSEL Optimization

- ROSE compiler infrastructure, Quinlan et al.
- Polyglot, Nystrom et al.
Conclusion

- General purpose languages should be metalanguages.
- That is, they should be great for hosting domain-specific embedded languages.
- Our current languages are OK but not great.
- More research is needed along several fronts to produce great metalanguages.