Excess: a mainstream metacompiler

Emilio Santos
Email: emilio.santos@outlook.com

***Abstract*—In this paper, we present a metacompiler framework that is host-language agnostic, spans the width of the compilation process and allows a wider range of extensions than any other library surveyed. We introduce a novel compiler architecture that provides the tools to concisely and efficiently implement these extensions. We also identified a simple mechanism to integrate programming language extensions into existing tools.**

I. Introduction

Programming languages (PLs) are universally regarded as a challenging technical task. Meyerovich [1] shows that, in addition to technical obstacles, new programming languages have low probability of success: "language adoption follows a power law; a small number of languages account for most language use". The combination of these factors makes investment in programming languages an unattractive proposition, therefore stagnating innovation in the field.

It is the author's contention that programming languages will evolve as a social process, with the developer community at large being the driver of innovation. A cursory analysis of the processes driving PLs today reveals, in fact, the opposite trends:

1) The community is a passive consumer of PLs.
2) Innovation is driven by large companies at a slow pace.
3) Research is limited to academia and seldom crosses into the mainstream.

These observations are in line with the findings in [1] as well as empirical observation. A reversal of these trends would imply bringing PL research into the few languages which gather the largest share of usage. This task can be accomplished by extending such languages. We propose Excess as a tool to accomplish this goal.

Also in [1], deciding factors for adoption are found to be largely of usability nature as opposed to correctness. Lastly, the same authors note that: "the programming market supports many languages with niche user bases", which seems to justify the trend in research towards Domain Specific Languages (DSLs). While we understand the importance of this type of language, we feel the area of language improvement hasn't received the attention it rightfully warrants.

The phases of language extension were defined in 1975 by [2] as: orthophrase ("add 'orthogonal' features to a language"), paraphrase ("define something new by showing how to exchange it for something whose meaning is known") and metaphrase ("altering the interpretation rules of a language so that it processes old expressions in new ways"). We use these phases as guidelines for our system slightly modifying the meaning of paraphrase to denote lexicographic extensions. The evaluation of existing solutions of which several have been proposed in the literature uses these phases as well.

Language Workbenches [16], project user input in textual or graphical form into source code for a host language in a process that usually involve several sub-languages and that only concerns itself with orthophrasic extensions. Efforts such as [4] extend particular languages such as Scala, again addressing only extensions of orthophrasic nature. Others, such as [6], [7], build metaprogramming languages to add metaphrasic extensions but limiting usability by forcing developers to learn new, usually complex languages. Our contributions are:

- **Service based host-agnosticity**: Our compilation process is largely independent of the language it ought to extend. Only a trivial to realize service is required for the host language, which needs no modification in order to be extended by Excess and which grammar needs not to be specified.
- **Evolutive transformation pipeline**: Excess allows extensions in all three phases of the language extension process described in [2] in addition to a projective phase. These extensions are realized coherently through three unique processes: match, transform and schedule (MTS). Moreover, extensions are allowed to evolve alongside the compilation phases.
- **Application interoperability**: Extensions realized using our system operate in the application scope, allowing a class of extensions difficult or impossible to realize in similar systems.
- **Cooperative extensions**: We extend previous work [8] by providing a general cooperation mechanism between extensions.
- **Expanded projection**: We redefine the projection process [3] into two separate process: inward and outward projection, which generalizes the intended operation.

We consider, however, our greatest contribution to be in the field of usability and community reach. By being capable of extending any language, mainstream languages can be improved by the community at large. Moreover, our extensions can be used unobtrusively in existing projects and share their ecosystem (tools, libraries, etc) which is the main factor driving PL adoption. Our extensions tend to integrate automatically with debuggers and we present a mechanism to realize advanced functions such as refactoring.

II. Related Work

Our work relates to both language extension and embedding (which we will use interchangeably with DSLs). We

further classify the approaches to which our work relates the most as: projective [3], [5], host-dependent [4], [9] and metaprogramming languages [7], [6].

Projective language extensions are usually in the form of Language Workbenches such as [3], [5]. In [3], Voelter presents a discussion on the advantages of such systems. We find these tools complicated as users must specify several subsystems (grammars, validators, generators) using custom languages. For instance, a calculator demo [12] for JetBrains MPS involves over 100 screenshots. Our solution offers the simplicity of only three operations: match, transform and schedule (MTS).

We also notice the projection process to be final (i.e. devoid of further modification). We expand upon this by allowing model instances to be injected into a compilation and then further transformed down the pipeline. Additionally, we allow the whole application to be projected outwardly into a different code base.

Projective systems do offer an impressive set of development tools. We consider part of our future work to investigate visual tools to match our inward projective stage, but note that our process is more general in the sense that the input format is not fixed or proprietary. Also, our system provides integration with existing tools in the form of semi-automatic debuggability.

Host-dependent systems rely on features of the host language to accomplish extensions. In [4], Brown et al present a DSL framework based on Scala. It is becoming common to see syntactic macro systems [9] in functional languages. These are programming languages themselves, which adds a layer of complexity. Additionally, the transformations seem to be restricted to syntactic elements (i.e, no semantic information). In [8], Sujeeth et al present a composable framework for DSLs, noting the difficulty of using multiple DSLs together. We expand upon this work by introducing a general mechanism for extension cooperation.

The main limitation of host-dependent systems (with regards to our objectives) is that they extend languages that aren't very popular in terms of usage [10]. In contrast, our solution is host-agnostic and geared towards c-like languages that dominate the market as also evident in [10].

Closer to our work are metaprogramming languages [7], [6]. These systems are host-agnostic by virtue of receiving the grammar from the user. A transformation engine is provided via programming language to rewrite its extensions. In our opinion, this approach limits usability by forcing users to learn a new language. The added software complexity of maintaining a PL also works against the ability of these systems to evolve. For instance, adding new features to a metaprogramming language could become difficult.

Our approach instead relies on language services such as [11] which are capable of all facets of compilation including providing semantic information. The use of tested code bases as language services provides Excess with a degree of reliability and usability not obtainable by simply providing a grammar for the host language. Furthermore, it allows for the extensions to be written in a general purpose PL and its ecosystem.

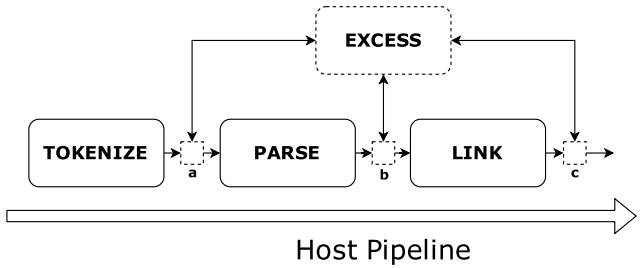

Fig. 1: Excess: a) paraphrase b) syntactic metaphrase c) semantic metaphrase

We could compare, as an example, Scala macros(viewed as a metaprogramming language) to a Roslyn [11] syntax node transformation and find them functionally equivalent. In both cases a syntax tree will be synthesized based on existing nodes. However, usability-wise, the latter is executed on a familiar language with debugging capabilities and the former on a foreign language without advanced tools.

The solutions reviewed in this chapter do not offer lexicographic (or paraphrasic) transformations. In order to provide a complete solution, Excess offers general token stream transformation engine compliant with the MTS model.

III. Overview

Figure 1 shows the overall pipeline of the excess compiler. This pipeline corresponds to the classical compiler architecture augmented with general transformations after each compilation step. This pipeline demonstrates the host-agnosticity of our approach: the host language will be asked to perform its classical tasks (tokenizing, AST creation, semantic annotation of syntax nodes) and Excess will transform the results of each stage before moving the process along.

Note that the host language only receives input in a format it understands (i.e the parser receives a token stream, etc). The host compiler perform its tasks as if Excess didn't exist at all, which implies that the host language doesn't have to change at all. Section III.A will formally define the functional requirements for a host implementation while Section III.B will provide an overview of the language extensions supported by Excess.

A. Host Service

An Excess compliant host is formally defined as a general software service or interface exposing traditional compiler functionality. These services could be provided directly by the compiler vendor as in [11]. A compliant service will implement:

1) **Tokenize**: The ability to transform source code into a stream of tokens and viceversa.
2) **Parse**: The ability to transform a token stream into an abstract syntax tree.
3) **Link**: The ability to assign semantic information to syntax nodes.
4) **Annotate**: The ability to add custom annotations to tokens and syntax nodes.

We deem this functionality to be traditional and well researched, therefore trivial to realize.

B. Extensions

Excess strive to offer the wider possible breadth of extensions, which directly ties into our search for usability. We identify four classes of language extensions realizable with Excess: Standard, Phrasic, Projective and Solution. In our opinion, while most of the systems reviewed in section 2 offer some of this classes of extensions, none offers them all.

Some of these extensions offer functionally orthogonal subclasses, the following list present an overview:

- **Standard**: Extensions following the syntax:

```
extension :=
    identifier ( parameters )
    {
        body
    }
```

  This extensions offer several subclasses determined by the contents of their body:

  1) **Host Extensions**: The body of these extensions is expected to be valid host syntax. Example (asynch, synch):

```
asynch()
{
  var result = expensive();
  synch()
  {
      notify(result);
  }
}
```

  2) **Custom Extensions**: The body of the extension is indeterminate, the extension writer must provide a custom parser.
  3) **DSL Extensions**: A subclass of custom extensions where the body is parsed by a user grammar. Example(json):

```
var result = json()
{
  value: 1,
  array: [2, 3, 4],
  object:{ a: 'hello',
           b: 'world'}
}
```

- **Phrasic**: Extensions realized by transforming individual compilation elements (tokens, syntax nodes, etc). Subclasses include:
  1) **Lexical Extensions**: realized by transforming a token stream.
  2) **Syntactical Extensions**: realized by transforming a syntax node.
  3) **Semantic Extensions**: realized by transforming an annotated syntax node.
  4) **Corrective Extensions**: realized by transforming a syntax node deemed erroneous by the host.
- **Projective**: Extensions intended to allow foreign entities to participate in the compilation process, either as inputs or outputs.
  1) **Inward Projection Extensions**: Transforms a collection of unrelated instances into compilation elements, which become part of the pipeline.
  2) **Outward Projection Extensions**: Serializes compilation elements into a different code base. Useful for library or language translations.
- **Solution**: Specify a separate compilation path for source files within a solution. Useful to apply *roles* to particular files, enabling "convention over configuration".

Standard extensions are functional equivalent of DSLs systems such as [4], [9], however, they allow generalized operations. For instance, while custom grammars provide all the expected functionality of traditional DSLs, a separate class of extensions (Host extensions) does not need a grammar whatsoever and instead apply transformations of metaphrasic nature to host-valid code. Moreover, a separate class of custom extensions are provided. While our system is independent of parser generators, its worth noting our current implementation supports ANTLR [17] grammars.

Phrasic extensions are simple element transformations. Of note would be our lexicographic extensions which uses a general pattern substitution engine. This type of systems has been criticized in the past for the risk of accidental capture (duplication) of identifiers. These systems are referred to as 'not hygienic'. Next section will present a methodology to produce provably hygienic extensions.

C. Hygienic Paraphrase

Our first insight is that this problem becomes acute on systems which only mechanism for language extension is the macro. Excess offer several more choices for extensions, some of which could guarantee hygienic-ness via semantic information. The first step in the methodology is:

Do not create model identifiers.

In practice it is seldom needed to add identifiers at this stage. The other potential problem during lexical analysis is the possibility of substituting existing constructs erroneously. Our next insight is that, to be considered a conflicting substitution, the model constructs to be preserved must be syntactically valid for the host language. The users unrelated working code must NOT be altered during paraphrase.

Only syntactically invalid constructs.

When a paraphrasic extension satisfies this constraint, we can formally prove this extension to be hygienic. The proof is by contradiction: Let **Cu** be a user construct erroneously transformed by extension **E**. Cu must be a valid syntactic construct, which is in direct violation of the constraint.

D. Example

Lets consider an "entity" extension with the requirement that entities only allow simple types. Listing 1 shows an example of valid input. Note this extension conforms to the established hygienic constraints since the sequence $entity \rightarrow identifier \rightarrow \{$ is invalid in c-like languages.

```
entity Example
{
   string Name;
   int Salary;
}
```

Listing 1: Example

```
lexical.match()
   .token('entity')
   .identifier()
   .token('{')
   .then(Transform);
```

Listing 2: Implementation

Listing 2 shows the lexical match to realize the extension. The transformation, then, can be written as (brackets denote compilation stages):

$$\left['\textbf{entity}' \rightarrow '\textbf{class}'\right] \Rightarrow \left[\textbf{OnlyFields}\right] \Rightarrow \left[\textbf{OnlySimpleTypes}\right]$$

Where $\rightarrow$ represents a substitution operation, $\Rightarrow$ scheduling, **OnlyFields** a syntactic transformation and **OnlySimpleTypes** a semantic transformation.

IV. Architecture

In this section we present the document-centric architecture of Excess. At the highest level of abstraction this type of architecture is common: each compilation source is called a *document* and they are grouped in *solutions*. Our compiler, however, differentiates itself on its definition of the compilation process: for us, to compile is to apply series of changes to each document in order to obtain host-compliant documents which are functionally equivalent to their original host-augmented versions. Moreover, we present a compact definition for the changes that is consistent throughout the different stages of compilation (as loosely depicted in Figure 1). Formally, our compilation process can be described as:

$$D_r = \sum_{1}^{n} C_i(D_i) \tag{1}$$

Where D_r is the document resulting from the compilation, D_i is the current document and C_i is the i-th change scheduled to transform the document. It should be noted that n is not constant since C_i is able to schedule future changes. Changes are expressed in terms of *Match, Transform, Schedule*.

Let Ω be the set of all compilation elements and Ω_i all elements in the ith stage of compilation. A change $c \in C$ is a 3-tuple (M, T, S) where:

$$\begin{aligned} M &= m_i : \Omega_i \rightarrow \{true|false\} \\ T &= t_i : \Omega_i \rightarrow \Omega_i \\ S &= s_{ij} : (\Omega_i, t_j) \rightarrow c_j | j >= i, c_j \in C, cj = (\tau_{ij}, t_j, \odot) \\ \tau_{ij} &= \Omega_i \rightarrow \Omega_j \end{aligned}$$

Informally: match functions (M) identify compilation elements, transform functions (T) apply the intended change and scheduling (S) create a new change on the current or a later stage. The function τ_{ij} projects a compilation element onto a stage. Figure 2 shows an outline of MTS operating on a token stream and scheduling syntactic changes, Figure 2c utilises τ_{ij} to locate $node_j$ and schedule the change starting with the syntactic match depicted in Figure 2d.

This architecture exposes several properties universally regarded as desirable:

- **Declarative**: The compilation can be described instead of implemented.
- **Simple**: Users must learn only three processes: Match, Transform, Schedule.
- **Functional**: All transformations are implemented using free-functions.
- **Dynamic**: The initial set of changes will evolve depending on the model being compiled.
- **Configurable**: The compiler is an empty shell only tasked with processing changes.

We believe that MTS could be useful for multiple problems but hasn't been properly identified as a software pattern.

A. Extensions

Excess extensions are realized by scheduling the initial set of changes into a compiler. This process happens along the lines of Listing 2: an extension will request a particular compilation stage (the 'lexical' variable), specify the elements it wants to transform ('lexical.match') and supply a transformation function ('Transform'). This formula is repeated for all stages of compilation.

The configurability of the compiler should be clear: the choice of extensions will determine the general functioning of the compiler. Solution extensions, as described in Section III are realized by leveraging the configurability of the compiler.

Naturally, the matching/transforming process is particular to each stage. Next section will detail the available stages and their functionality.

B. Compilation stages

In this section we will provide a review of the matching and transforming facilities at each stage. The code for the platform is open source, as such, the best way to obtain in-depth information would be to analyze the source code.

Function signatures described will include a 'scope' parameter which relates compilation state to the transformation. Section IV.C will describe this process in detail.

```
compiler.Syntax()
    .match<MethodDeclaration>(method =>
        method.ReturnType.IsMissing &&
        method.Identifier == "constructor")
    .then(Constructor);
```

Listing 3: Constructor Extension

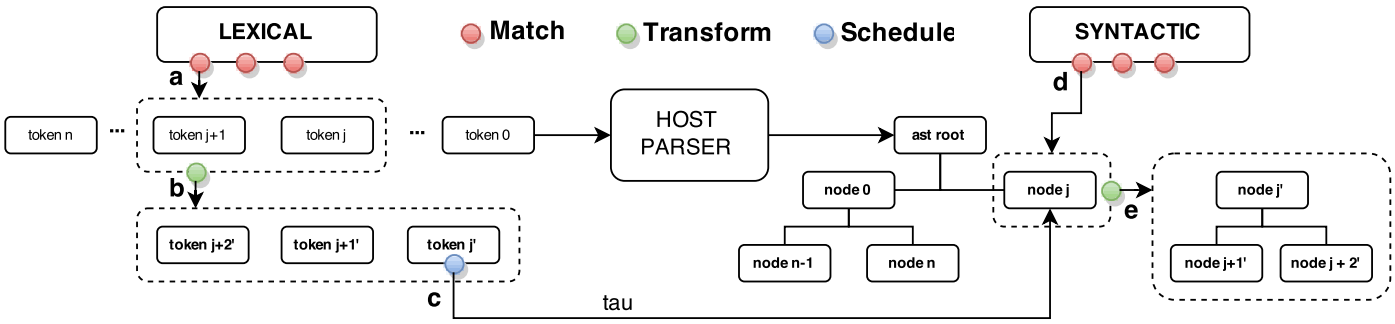

Fig. 2: MTS: a) Lexical match b) Lexical transform c) Schedule operation d) Scheduled syntactic match e) Scheduled syntactic transformation.

1) Lexical stage: The original token stream is transformed by substituting token patterns. Listing 2 shows an example of lexical change where the transformation function has the signature:

$$T(SyntaxToken[], Scope) \rightarrow SyntaxToken[]$$

Compilation stages usually offer several utilities that provide more information based on specific functionality. For instance, a 'grammar' match function is provided which accepts transformations intended to translate foreign grammar nodes into the host language. Such functions have the signature:

$$T(GrammarNode, Scope) \rightarrow SyntaxNode$$

2) Syntactic stage: At this point in the compilation a valid syntax tree has been synthesized. The match functions are based on node types. Listing 3 shows an example of syntactic matching. Transformation functions match the signature:

$$T(SyntaxNode, Scope) \rightarrow SyntaxNode$$

Both this and the lexical stage support an 'extension' match function for standard extensions with the signature of its transformations being:

$$T(SyntaxNode, ExtData, Scope) \rightarrow SyntaxNode$$

Where *ExtData* contains extension data such as the parameter list.

3) Semantic stage: The semantic stage does not offer matching capabilities since types are not known at declaration time. However, this stage is designed to work via scheduling. Since the host requirement is to annotate syntax nodes with semantic information, the transformations in this stage are very similar to the syntactic stage. The signature for transformations is:

$$T(SyntaxNode, Semantics, Scope) \rightarrow SyntaxNode$$

```
compiler
  .Semantics()
    .error('CS0246', FixMissingType);
```

Listing 4: Corrective Extension

Additionally, the semantic stage allows for corrective extensions as shown in Listing 4.

4) Projective stage: Inward projection, as described in Section III, is considered a special case of lexical stage that instead of tokens processes model instances. Naturally, documents meant for this stage do not support paraphrasic extensions. The signature of the projective transform is:

$$T(object, Scope) \rightarrow SyntaxNode$$

C. State Injection

From functional programming we understand the importance of keeping transformations stateless. It is clear, however, that some state will be needed at some point. Particularly when extension must cooperate amongst themselves to build a domain specific solution such as [8].

The solution we propose, which we call state injection, draws its inspiration from the well known concept of scope, defined as the collection of valid name bindings available at any point in the execution of a program. We replicate this concept with an artificial context, a hierarchical structure offering named bindings at each node. At compilation time, this scope is passed as a parameter (i.e. injected) into all transformations. Originally, our scope is designed to carry compiler level information (such as the current document) and aid parallelization efforts.

Although this pattern shouldn't be considered novel, to our knowledge it hasn't been formally defined before.

1) Extension interoperability: As a general instance repository, the scope proves to be useful as a communication channel between extensions. Lets suppose we have 2 extensions A and B that want to coordinate their respective changes. It would suffice for extension A to register a named instance to a shared scope and then extension B can simply retrieve it on demand. Listing 5 shows pseudo-code to demonstrate this concept.

It should be noted that only A and B are privy to the *ExtensionAManager* type, and as such, the scope is not polluted. In other words: the rest of the system is not aware of the existence of this particular manager and conflicts will not arise because of it.

```
void initExtensionA(Scope scope)
{
    scope.addAManager(new ExtensionAManager());
}

SyntaxNode transformB(
        SyntaxNode node,
        Scope scope)
{
    var manager = scope.getAManager();
    manager.interact(node);
}
```

Listing 5: Seamless extension interaction

Extensions that rely on an intermediate representations (IR) such as [8] are trivial to realize under this system: A single copy of the IR interface will be registered into the scope and then updated by the individual cooperating DSLs.

D. Tool integration

Excess effectively creates a bijective relationship between source code and output: from an input construct the compiler can find the corresponding output constructs and viceversa. This integrates with the most important function of development tools: debugging. Alas, most modern languages provide debug info capable of 'redirecting' the debug location of compiled statements. For instance, the **#line** directive in C++.

Debug mapping can be achieved automatically, so most Excess extensions are debuggable by default without any additional work on the part of the extension writer. We believe this a leap forward in usability compared to the available solutions. Some other advanced tools, like refactoring, are as well implementable using this model, but might not do so automatically. This is an area of future work.

V. Implementation and results

We include the source code for our implementation [14] of the concepts presented in this paper. This implementation has been realized in C# using Roslyn [11] as the host compiler. Additionally, we provide an online IDE (www.metaprogramming.ninja) capable of building and testing user extensions. We believe this online IDE to be the first of its kind.

Our implementation itself fits into our guidelines for simplicity: it is under 5K lines of code (LOC) with room for new types of extensions. Additionally, the extensions created are also very economical in size. We categorize our results in three areas: General programming languages, Domain specific languages and Language Extensions.

A. General programming languages

We ported the xs language [13] using Excess. The development of this language showed all the challenges associated with PLs: large, complex C++ code base and lack of adoption. When re-implemented, the effort totalled around 600 LOC. Moreover, the resulting language (being a superset of a mainstream language), contains all the modern constructs a developer may expect along with the tools to make the language useful. The language contains the following extensions to C#: **constructor**, **method**, **property**, **event**, javascript-style **function**s and arrays, and c-style **typedef**.

B. Domain specific languages

We created a demo implementation of R [15] which was selected by virtue of being one of the most popular DSLs in [10]. The implementation, including the underlying vector library and the transformation of the R grammar was realized in under 1500 LOC and integrates seamlessly with C# as seen in Listing 6.

```
void main()
{
  var data = new int[] {1, 2, 3};
  var result = R()
  {
    tmp <- c(0, data, 4);
    tmp1 <- seq(0, 4)

    tmp + tmp1
  }

  //result == [0, 2, 4, 6, 8]
}
```

Listing 6: R extension

This DSL was realized using an open source grammar for the R language. The general availability of these grammars makes us optimistic about the complexity of future developments.

C. Languages Extensions

Outside of full-fledged programming languages, we believe smaller language extensions are useful in order to improve existing programming languages. We have developed four of these extensions that we consider improvemental (the implementation's LOC in parenthesis):

1) **contract**(68): a syntactical check for several conditions.

```
contract()
{
    input > 0;
    input < 10;
}
```

2) **match**(153): an extended 'switch' statement:

```
match(input)
{
    case 0:
    case > 10:
    default: ...
}
```

In future work we would be extending the capabilities of this extension to mimic the functional match.

3) **asynch**(78): asynchronous constructs, a simple host extension.

4) **json**(151): JSON DSL, a grammar standard extension

It is our belief that the extensions presented in this section cover a wide enough range and show the versatility and simplicity of our metacompiler.

VI. Conclusion

We have presented a host-agnostic metacompiler capable to extend mainstream languages. We believe the extension of such languages is needed in order to reach the developer community at large. Extensions creation with Excess have proven to be simple by showing four different extensions averaging close to 100 LOC. This process creates extensions that largely integrate with native tools (such as debuggers).

The simplicity provided to extension writers can be attributed to its supporting architecture, which has been described as MTS. Ten separate classes of are discussed which are consistently handled by the architecture.

We've shown Excess ability to aid research and development of programming languages, either general purpose or domain specific. Traditional grammars have been used to realize some of the extensions (JSON, R). We have found the effort required to implement these languages to be proportional to the individual language complexity and not to the general complexity of PLs.

The work has been released open source, including an online IDE for ease of experimentation. We however recommend 'forking' the online repository as the best way to either contribute or create new research. Cooperation is cherished.

VII. Future Work

The most important task remaining in our work is the support for another mainstream language as to prove our host-agnosticity of our metacompiler. It has been noticed the possibility of host services been provided by the compiler makers such as [11] and its implications for usability and integration. In the absence of such facility we present the methodology we intend to follow:

1) Find an open source version of a compiler for the intended language (such as OpenJDK for Java)
2) Identify and intercept compiler stages (tokenization, AST construction, etc).
3) Implement MTS in the compiler's language to be called on interception.

In order to truly claim mainstream usability we must include support for the tools used the most by the community (Visual Studio and Eclipse, among others). Investigation on how to integrate advanced features of such IDEs is needed.

The projective stage of Excess is admittedly underdeveloped. A system of visual tools for the inward projection should be developed even though we notice that our compiler is capable of working with third party tools. The outward projection stage is yet to be implemented, a similar system was developed by the author in [13], we are hoping the insights will translate.

Acknowledgment

The authors would like to thank Luis Castillo Arce of the University of Minnesota for his invaluable help.

References

[1] Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical analysis of programming language adoption. SIGPLAN Not. 48, 10 (October 2013), 1-18.

[2] Thomas A. Standish. Extensibility in programming language design. SIGPLAN Not., 10(7):1821, 1975.

[3] M. Voelter. Language and IDE Modularization and Composition with MPS. International Summer School on Generative and Transformational Techniques in Software Engineering, GTTSE 2011, LNCS 7680, pp 383-430, 47 pages, 2011

[4] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain specific languages. PACT, 2011

[5] Lennart C. L. Kats, Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. OOPSLA 2010: 444-463

[6] Rascal: From algebraic specification to meta-programming J Bos, M Hills, P Klint, T Van Der Storm, JJ Vinju - arXiv preprint arXiv:1107.0064, 2011

[7] E. Visser, Stratego: A language for program transformation based on rewriting strategies, in: Proc. Rewriting Techniques and Applications RTA01, Lecture Notes in Computer Science 2051 (2001) 357361.

[8] Arvind K. Sujeeth , Tiark Rompf , Kevin J. Brown , HyoukJoong Lee , Hassan Chafi , Victoria Popic , Michael Wu , Aleksandar Prokopec , Vojin Jovanovic , Martin Odersky , Kunle Olukotun, Composition and reuse with compiled domain-specific languages, Proceedings of the 27th European conference on Object-Oriented Programming, July 01-05, 2013

[9] Matthew Flatt. 2012. Creating languages in Racket. Commun. ACM 55, 1 (January 2012), 48-56. DOI=10.1145/2063176.2063195 http://doi.acm.org/10.1145/2063176.2063195

[10] Tiobe.com,. 'TIOBE Software: The Coding Standards Company'. N.p., 2015. Web. 13 Apr. 2015.

[11] Neil McAllister, Microsoft's Roslyn: Reinventing the compiler as we know it, DEVELOPER WORLD, 2011-10-20

[12] "The Calculator Language Tutorial." JetBrains. Web. 13 Apr. 2015. https://www.jetbrains.com/mps/docs/tutorial.html.

[13] Santos, Emilio. (2013). XS Language. Zenodo.10.5281/zenodo.16795

[14] Santos, Emilio. (2015). Excess. Zenodo.10.5281/zenodo.16662

[15] R Development Core Team (2008). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

[16] M. Fowler. Language workbenches: The killer-app for domain specific languages? http://martinfowler.com/articles/languageWorkbench.html, 2005.

[17] Terence Parr. 2007. The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf.