Language Factories

Tony Clark
Thames Valley University, St. Mary's Road,
Ealing, London, W5 5RF, United Kingdom
tony.clark@tvu.ac.uk

Laurence Tratt
Bournemouth University, Poole, Dorset,
BH12 5BB, United Kingdom
laurie@tratt.net

Abstract

Programming languages are the primary mechanism by which software is created, yet most of us have access to only a few, fixed, programming languages. Any problem we wish to express must be framed in terms of the concepts the programming language provides for us, be they suitable for the problem or not. Domain Specific Languages (DSLs) suggest an appealing escape route from this fate, but since there is no real technology or theory underpinning them, new DSLs are rare. In this paper we present the *Language Factories* vision, which aims to bring together the theory and practice necessary to realise DSLs in a systematic way. In so doing, we hope to lower the barrier for language creation significantly, ultimately allowing software creators to use the languages most suited to them and their needs.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General

General Terms Languages

Keywords Domain specific languages

1. Introduction

10 years ago, in his influential OOPSLA talk [Steele 1999], Guy Steele made the following statements:

> ...a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language.
>
> ...from now on, a main goal in designing a language should be to plan for growth.

These statements capture two notions. First, that programmers need more than is provided by existing programming languages. Second, that programming language design has generally made little to no attempt to cater for extensibility or customisability. Indeed, for most programmers, in most programming languages, the only 'language extension' mechanism available is the library[1]. Libraries have allowed us to create systems of immense complexity and surprisingly high reliability, but have the disadvantage that they are visibly second-class citizens to language primitives. Features such as first-class functions and run-time meta-programming allow programming languages to be bent in slightly different directions, although the room for manoeuvre is still limited [Wilson 2005]. Whatever problem the user is trying to express must ultimately be encoded in terms of the limited set of features provided by the original language designer. While we can push at the edges of the padded cells that our programming languages provide us, the elasticity of the walls is slight and, ultimately, there is no way out.

DSLs suggest an appealing escape route from the restrictions of programming languages, enticing us with the idea of languages whose syntax and semantics can be customised to our purposes. The motivation for DSLs is well understood: they are customised languages which allow classes of related problems to be more quickly and precisely realised than with traditional techniques [Bentley 1986]. The potential use cases for DSLs are innumerable, ranging from taming the complexity of 'enterprise' systems (such as J2EE) to allowing non-programmers to express tax law in a precise fashion (as in Intentional Software's tool). Unfortunately, in practice, the overhead of creating DSLs is so significant that the idea is generally watered down to mean the clever abuse of existing language features to make 'literate' programming interfaces; Hudak outlines the reasons for this [Hudak 1998], while the Ruby community has embraced this approach with vigorous enthusiasm. In other words, despite the hype, we are basically stuck with libraries for language extension; programming languages have restricted us to their cells, while we incorrectly assumed we had new freedoms.

We have two fundamental contentions. Our first is that DSLs, in their purest form, represent a *desire for customis-*

[1] Lisp is the obvious counter-example. However, Lisp's minimalistic syntax is core to its approach, which then renders it unsuitable for many DSLs which need to drastically manipulate syntax.

able languages, where syntax is as malleable as semantics, and where we can choose the degree to which we are constrained (or not) by existing languages. Our second is that *DSLs are a desire lacking a philosophy*. It is the latter issue to which this paper addresses itself; until, and unless, the underlying mechanisms for reasoning about and creating DSLs are improved, their potential cannot be realised. A corollary of these two contentions is that, whilst DSLs are the specific target of current interest, there is nothing which fundamentally identifies a DSL as being distinct from other classes of languages. The main difference is that few people have a need to build 'big' languages, and those lucky few are comfortable with traditional techniques; many more people have a need for the 'small' languages that DSLs typically represent. Thus, while this paper is motivated by DSLs and we expect it to be most frequently applied to DSLs, the approach herein is applicable to languages in general, and we frame most of our discussion in terms of generic languages.

1.1 Where we are today

We start this paper from the position that the status quo is not an option, for two reasons. First, as Hudak notes, virtually nobody builds 'pure' DSLs because the costs are prohibitive [Hudak 1998]. Second, implicit in Steele's talk, is that languages tend to stop evolving as soon as they reach a certain point (typically a '1.0 release'). We suggest that the reason that languages stop evolving may be because: it's difficult to reason about the effects a change to a language will cause to both it and existing programs; and, there is little or no support for co-evolving programs with language changes (the TXL language is a rare example of this; it was designed specifically to allow programs written in the Turing language to be automatically evolved for a new version of the language [Cordy 2004]).

Asserting that the status quo is not an option is only useful if there are signs that something better could emerge. In recent years, several approaches have emerged which, while not complete in and of themselves, do provide such signs. We provide a more complete list in Section 6.2, but technologies such as Stratego/XT [Bravenboer and Visser 2004], XMF [Clark et al. 2008], and Converge [Tratt 2008] (the last two of these being by the authors of this paper) have shown that relatively powerful DSLs can be implemented at relatively low cost providing that the host language offers some key language building technologies. All of these approaches have seen some use in the real-world, although none has yet had a major impact. However none of these various emerging technologies has any great conceptual consistency, and none utilises any kind of theory about DSLs.

1.2 An outline of our proposal

This paper outlines our vision of *Language Factories*. In a nut-shell, Language Factories aim to make DSL design systematic by allowing languages to be realised from components which describe fragments of syntax, semantics, tooling capabilities, and so on. By providing a firm base on which to build languages, we believe that powerful, reliable, DSLs will be able to be built at significantly lower cost than today's ad-hoc approaches. Furthermore, a move to component-based language design offers greater potential for variable implementations of a language, and for the language to evolve in a predictable manner. Section 2 details Language Factories in detail.

We are mindful of two considerations in particular: one size rarely fits all; and 'jam tomorrow' does not sustain life today. To address the former point, Language Factories is an abstract concept – similar in spirit to design patterns – describing a *family* of related approaches; each Language Factory must conform to the basic principles of Language Factories, but can differ in various aspects. To address the latter point, it is possible to create a Language Factory which, by imposing various restrictions on the languages it can express, can then realise those languages within a normal programming language (Section 2.3 discusses the possible options, and trade-offs, in detail).

2. Language Factories

Language Factories are a component-based approach to the definition and construction of languages, as well as associated tooling, verification, and so on. Language Factories are, at the highest level, an abstract concept, describing a family of related approaches. An individual Language Factory is a concrete realisation of this concept which makes choices about what languages it can express, how they can be implemented, and so on. The common concept across all Language Factories is the *language component*.

2.1 What is a language?

At an abstract level, a language is a means of communication; in the case of computing that communication is generally between a human and a machine. In order to be usable, a language needs to have a way that participants can share communications (syntax) and an agreed shared meaning (semantics). Languages may form parts of larger languages (e.g. the sub-part of English used only in computing could be detached and reattached to the main language); they may be parameterisable (e.g. American and British English can be seen as variations on the single, abstract, language English); they may have variable syntaxes (e.g. Serbian is written in both the Cyrillic and Latin alphabets); and so on.

As far as is practical, the Language Factories concept tries not to be prescriptive. At a minimum, a language needs a syntax in which to express it (even if it is in a generic XML-like format) and an accompanying semantics of sorts (which can range from a formal mathematical definition, to the description of a translation into C). Exactly what form the syntax and semantics take is left to an individual Language Factory.

	C	Pascal	Java	Smalltalk	XMF	Converge	MetaLua	Stratego/XT
Syntax modification	○	○	○	○	●	○	●	●
Syntax extension	○	○	○	○	●	●	●	●
Introspection	○	○	●	●	●	●	●	n/a
Self-modification	○	○	○	●	●	●	●	n/a
Intercession	○	○	○	●	●	●	●	n/a
Compile-time meta-programming	Partial	○	○	○	●	●	●	n/a
Traceability	○	○	○	○	○	●	○	n/a

Table 1. A summary of some of the features in contemporary programming languages relevant to Language Factories.

2.2 Language Components

Language Factories break languages down into components, including the following parts:

Abstract syntax The single definition of its Abstract Syntax Tree (AST).

Concrete syntax(es) and syntactic mapping A definition of its concrete syntax(es) specified as e.g. a context free grammar, and a mapping from that concrete syntax to the abstract syntax.

Semantic aspect(s) Each semantic aspect defines (a possibly incomplete part of the) semantics. Semantic aspects may overlap with each other (e.g. an operational and denotational semantics) or describe completely different elements of the semantics (e.g. semantics of language types and semantics for text editors supporting tool-tips).

Constraints Describes constraints on how the language can be composed with others (both in terms of what the component provides, and what it requires of other components).

An individual Language Factory can decide which of the above parts are optional or mandatory, and which combinations are illegal. For example, a Language Factory may require only the concrete syntax to be defined, in order to allow the testing of syntactic composition; however, if a component wishes to define semantics aspects, it will have to define the relevant parts of the abstract syntax.

The granularity of components is left open to users; at its crudest, a component could be an entire language. Ideally, of course, one would like language components to capture smaller, reusable, aspects of languages. For example, many DSLs integrate expression languages [Hudak 1998]; since expression languages vary relatively little, it is feasible to share a single expression language across many larger languages [Tratt 2008]. Language components need not be fixed and immutable; just as with language libraries, the designer of a language component can trade implementation expense with flexibility.

2.3 Realizing Language Factories

We have, until this point, been deliberately abstract about how Language Factories could be realised. The reason for this is simple: Language Factories can be realised using different formalisms, tools, and with varying degrees of automation. The approach is a spectrum whose extreme positions can be thought of as *virtual* and *idealised*. At the virtual end, languages are specified but not implementable; at the idealised end, they have their own fully bespoke implementations. These two extremes also, in a sense, represent the status quo: specifying a language is relatively easy (a virtual language), but providing a stand-alone implementation is hard (hence idealised).

Fortunately, the many shades of grey between the two extremes are both meaningful and useful. In particular, we are keen that Language Factories can be realised using existing infrastructure (chiefly through currently available languages) when practical; this cuts down on implementation costs and also lowers the interoperability burden. Doing so inevitably involves some compromises, which depend on the language targeted. As an example of the compromises that need to be considered, Table 1 shows a representative selection of languages that are compared in terms of relevant features. Each Language Factory needs to consider what features it requires of a target language. For example, ‘syntax extension’ describes whether a language can allow new syntaxes to be naturally embedded within it; languages which allow this can directly express embedded languages; languages which do not allow this have to be translated into an intermediate form without the extension. Similarly, ‘traceability’ describes whether a language can trace the translated version of an embedded language; languages which do not allow this are likely to present many more challenges when debugging both Language Factory implementations, and programs written in it. In some cases, such compromises will not be acceptable, and a bespoke implementation of the Language Factory may be the best route.

3. An example Language Factory and case study

In order to make the description of Language Factories concrete, in this section we present an imaginary Example Language Factory (ELF) and a case study in it. ELF is a simple Language Factory intended to specify behavioural languages, using relatively traditional grammar specifications, and allowing language components to have both an operational semantics and a semantics via a translation into Java. The case study is based on the languages needed to specify different aspects of an aircraft's systems; for obvious space reasons, we are only able to tackle a small part of this in detail in this paper.

3.1 ELF language definition

ELF is a simple, but powerful, Language Factory for defining certain classes of textual languages. In particular, it provides practical support for composition of components, and allows running Java implementations to be produced from ELF language components. ELF itself uses a simple indentation based syntax, with the general syntax of an ELF language component being as follows:

```
lang:
  ast:
    ...
  grammar:
    ...
  semantics eval(env):
    ...
  semantics java:
    ...
    constraints:
      ...
```

In ELF, the `grammar` clause is mandatory, but all other clauses are optional. The `ast` clause is typically very simple, detailing the AST constructors required. The `grammar` clause allows grammars to be specified via a largely standard EBNF syntax from which ASTs are created; ELF assumes that implementations use an Earley parser [Earley 1970], so that any context free grammar can be used. An ELF language component may define semantics either operationally (through the `eval` clause) or via a translation to Java (the `java` clause). The operational semantics references an environment associating variable names with values. The Java translation uses quasi-quotes [Sheard and Peyton-Jones 2002] to represent code templates (generating Java code as an AST, not as a string). Both `semantics` clauses are driven by pattern matching. Finally, the `constraints` sub-clause within the `java` clause allows a component to specify its expectations about elements in the target Java environment.

3.2 The case study

Our case study involves a fictional, large aerospace company that is working on the design and implementation of an aircraft. The life-cycle of an aircraft project is lengthy and complex; many different aspects need to be precisely specified, implemented, evaluated, and altered. Such a project ideally requires a diverse collection of languages to support various parts of the life-cycle such as: planning; requirements; specification; implementation; testing; deployment; manufacture; and so on. Traditionally, a handful of small languages are co-opted to perform a wider variety of tasks than they are ideally suited to, with many of those languages being only informally specified. The fundamental question is therefore: can Language Factories help to quickly create robust languages that support the aircraft project life-cycle?

To make this both concrete, and tractable within the space confines of the paper, we concentrate on the specification aspect of the aircraft life-cycle, where both the hardware and software components of the system need to be precisely specified. This requires a language that can specify components and can specify both the differing states that the components can be in over time, and how it moves between them. For example, the landing gear of a plane is a distinct component that is either deployed, stowed, or moving. The specification of a landing gear component sets an initial height and speed; as the height and speed change, or the pilots issue new instructions, the landing gear will move through different states. We would therefore like to specify the landing gear as follows (we have elided several transitions in the interests of brevity):

```
component Landing_Gear(height:int, speed:float) {
  stm {
    state Moving_Up
    state Moving_Down
    state Deployed
    state Stowed
    transition up from Deployed to Moving_Up
      height_change[height>500ft and speed>100kn/s]
    transition down from Stowed to Moving_Down
      deploy
  }
}
```

Most of the above specification is fairly intuitive. A landing gear has an initial height and speed (both of which will change as the plane speeds up and slows down). Transitions have a name (e.g. `moving_up`) and specify a move from one transition to another, conditional on a named event occurring, and an (optional) guard being satisfied, with the latter two elements' syntax being *event* `[`*guard*`]`.

There are many different ways in which we could create a language to implement the above. The traditional approach would be to create a single language which captures all of the above features, bringing all the problems and costs noted in Section 1.1. However there are clearly different aspects to the language which can be teased apart: a superstructure that defines aircraft components; a state machine language (which has little to do with aircraft components as such); and an expression language which can express lengths and measures. In the following section we show how Language Factories can make the process of building this sort of language systematic, reliable, and realistic by breaking the language down into separate components, many of which are likely to already exist when such a language is created.

3.3 A reusable expression language

The first language component we define is a generic expression language, `Expr`. In order that we can show a sufficient breadth of the case study, we present a simplified version of the expression language, concentrating on variables, addition arithmetic, and integers—extrapolating the rest of the expression language from these examples is mechanical and largely trivial. Expr's AST is simple, requiring little explanation:

```
lang Expr:
  ast:
    Var(Str)
    Add(Expr, Expr)
    Num(Int)
```

The grammar for this component is as follows:

```
grammar:
  expr -> name:Id                  <Var(name)>
        | lhs:expr '+' rhs:expr <Add(lhs, rhs)>
        | num:Int                  <Num(num)>
```

The core of an ELF grammar is EBNF, with AST constructors contained between angled brackets. Terminals and non-terminals can be prefixed with a name, followed by a colon, which allows that element to be referred to in the AST constructor.

We can now easily give an operational semantics to Expr:

```
semantics eval(env):
    Var(x)    -> lookup(env, x)
    Add(x, y) -> eval(x, env) + eval(y, env)
    Num(x)    -> x
```

The operational semantics of expressions is defined with respect to an environment of variables env which maps names to values. The semantics consists of a series of definitions, from AST patterns (on the LHS) to integer expressions (on the RHS). For example, the first rule of eval matches against a variable and specifies that its semantics are given via the pre-defined lookup function, which returns the value of x in the environment env. As in the above, semantics clauses can always recursively call themselves via a function of the same name as the clause (i.e. eval in the above). Note that while ELF's operational semantics uses a fixed collection of data values and operators (e.g. integers and addition), [Ivanovic and Kuncak 2000], [Mosses 2004], and others have shown how language semantics can be made modular through the use of techniques including monads and labelled natural semantics systems, all of which could be included in a Language Factory.

Since Expr is intended to be reusable, and since Java does not allow operator over-loading, we cannot translate Expr types directly into Java base types, as this would not allow new types to be added to Expr. This is a standard example of the sort of real-world compromise that choosing an existing programming language as a target can impose. We are therefore forced to construct a new type hierarchy whose root is ExprObj, with concrete sub-classes such as ExprInt and so on. The Java semantics therefore looks as follows:

```
semantics java:
  Var(x) -> [j| ${x} |]
  Add(x, y) -> [j| ${java(x)}.plus(${java(y)}) |]
  Num(x) -> [j| new ExprInt(${x}) |]

  constraints:
    exists_class(ExprInt)
    exists_class(Expr)
    exists_static_method(Expr,plus)
```

Similarly to the operational semantics, the Java semantics is a series of definitions, from ELF patterns to Java ASTs. The latter can be built manually when necessary but, in general and as in the above example, be expressed by quasi-quoting. Quasi-quoting is a well understood technique (see e.g. Stratego/XT [Bravenboer and Visser 2004]) which allows abstract syntax to be expressed via concrete syntax expressed by quasi-quotes [| ... |]. In the above example the j in the quasi-quotes makes explicit that the AST being built by these particular quasi-quotes is a Java AST. Insertions ${...} allow ASTs to be built out of smaller chunks, or for ELF values to be 'lifted' to their Java AST equivalent (so ${2} creates a Java AST integer whose value is 2; see [Sheard and Peyton-Jones 2002] for more details of lifting). The constraints within the Java semantics are on the eventual Java program. ELF provides various predefined constraints: exists_class(*C*) asserts that the Java system in which the ELF component is translated into must have a class called *C*; exists_static_method(*C*, *M*) asserts that the Java system in which the ELF component is translated into must have a static method *M* in the class *C*. A more complete Language Factory would allow more complex constraints to be defined; the above should however give a sufficient flavour.

3.4 Measurement types

The reusable expression language in the previous section is missing one data type commonly needed in languages used in aircraft: a simple means of expressing measurements. In this subsection we define a very simple language component which allows expressions of the type 3kn/s (to be read as 'three knots per second') to be defined. The component itself is very simple, using exactly the same concepts as Expr (in the interests of brevity, we elide the operational semantics):

```
lang Measurement:
  ast:
    Ft(Float)
    KnPerH(Float)
    MiPerH(Float)

  grammar:
    measure -> dst:float 'ft'   <Ft(dst)
             | dst:float 'kn/h' <KnPerH(dst)>
             | dst:float 'mph'  <MiPerH(dst)>

  semantics java:
    Ft(x)     -> [j| new ExprFeet(${x}) |]
    KnPerS(x) -> [j| new ExprKnPerH(${x}) |]
    MiPerS(x) ->
      [j| new ExprKnPerH(${x*0.869}) |]

    constraints:
      exists_class(ExprFeet)
      exists_class(ExprKnPerH)
```

We can now compose Expr and Measurement together to produce a new expression language which can also express measurements. There are many different potential forms of composition and different Language Factories can provide composition operators which differ significantly in detail; in this case we can use ELF's simple composition operator:

```
merge(l1, l2, grammar:{...}, semantics:{...})
```

that merges l1 and l2 to produce a new language. In essence, merge constructs the union of the two languages in terms of their grammar and semantic rule-sets. The grammar and semantics parameters allow additional grammar and semantic rules to be added into the merged language, gluing the two sub-languages together. For example, grammar:{R1 -> R2} specifies that in the merged language, the grammar rule R1 should have a new alternative added that references R2. The semantics: parameter has a similar effect. For an expression language with measurements, merge is used as follows:

```
ExprMeasurement = merge(Expr, Measurement,
  grammar:{Expr::expr -> Measurement::measure},
  semantics:{})
```

In this case the merge of the two languages is simple, with only a reference needed from Expr's expr rule to the measure production.

While simple conceptually, this type of composition is a fundamental part of Language Factories, allowing language components to be reused and customised, even in ways that their original authors might not have anticipated.

3.5 A parameterisable language for statemachines

Aircraft components are frequently specified by statemachines. In this section we show a simple example of a generic statemachine, whose guard language can be parametrised via the guard_lang parameter. The language elements (ast, grammar, semantics) of the parameter can be used in appropriate places within the body of StateMachine:

```
lang StateMachine(guard_lang:Component):
  ast:
    STM([State | Transition])
    State(Str)
    Transition(Str,Str,Str,Str,guard_lang.ast)

  grammar:
    STM -> 'stm' '{' elems:(State | Transition)*
      '}' <STM(elems)>
    State -> 'state' name:Id <State(name)>
    Transition -> name:Id from:Id to:Id
      event:Id '['
      guard:guard_lang.grammar ']'
      <Transition(name,from,to,event,guard)>

  semantics java:
    STM(states, transitions) -> [j|
      class ${freshname()} {
        States state;
        enum States {${states}};
        ${self(transitions)};
      }
    |]
    Transition(name, from, to, event, guard) ->
    [j|
      @Transition
      public void ${name}(Event ev) {
        if (self.state == ${from} &&
          ev == ${event} &&
          ${guard_lang.semantics.java(guard)}) {
           self.state = ${to};
           return true;
          }
          return false;
      }
    |]
```

While the above is relatively detailed, we hope that most of it is, given what has come before, relatively intuitive. Note the use of the guard_lang parameter of type Component (for the avoidance of doubt, this type denotes a Language Factory language component). The StateMachine component includes the AST of the guard language as a component of the transition constructor and the guard language's start non-terminal is used to parse this element of a transition's concrete syntax. The Java translation semantics for a transition calls the guard language's Java translation semantics.

The advantage of defining the state-machine language in this way is that, since we know that statemachine languages often require subtly different expression languages, we can make that parametrisation easy. At its simplest, a user can pass the vanilla Expr language component as the parameter:

```
ExprSM = StateMachine(Expr)
```

In our case study however, we wish to use the expression language including measurements, so we instantiate a statemachine language as follows:

```
ExprMeasurementSM = StateMachine(ExprMeasurement)
```

With the ExprMeasurementSM language, we can now express fairly complex state machines, and use guards such as x < 10kn/h.

Merge (see Section 3.4) and parametrisation are related forms of language customization, each having its advantages and disadvantages. When a very specific instance of variability is known in advance, parametrisation is attractive as it makes the location and impact of the variability explicit. When a type of customisation is needed that could not have been originally envisaged – or if the type of customisation required is more sophisticated than parametrisation can achieve – then composition comes into play. There are inevitable shades of grey between the parametrisation and composition, and users of Language Factories will have to use their own judgement to decide when each is appropriate.

3.6 A language for components

Finally, we define a component ACComponent (AirCraft Component) which provides a standard way of expressing an aircraft component: each has a name and is instantiated with a list of variables which become the components attributes; the body of the aircraft component is not specified and is passed as a parameter to ACComponent.

```
lang ACComponent(body_component:Component):
  ast:
    Component(Str,[Var],body_component.ast)
    Var(Str, Str)
```

```
grammar:
  component -> 'component' '{' name:id
    vars:var*
    body:body_component.grammar '}'
    <Component(name, vars, body)>
  var -> name:id ':' type:id
    <Var(name, type)>

semantics java:
  Component(name, vars, body) -> [j|
    @Component
    class ${name} {
      ${java(vars)}
      ${body_component.semantics.java(body)}
    }
  |]
  Var(name, type) -> [j| ${type} ${name} |]
```

One item of note in the above is that since ACComponent does not know the name of the top-level AST element in the language component body_component, it cannot be referred to directly; instead it makes use of the fact that each language component's AST slot refers implicitly to the top-level AST element. The same mechanism is used to link the grammar of the body_component into ACComponent's grammar.

Finally we can instantiate a statemachine component, giving us the language component with which we can express the landing gear example of Section 3.2:

```
AicraftDesign = ACComponent(ExprMeasurementSM)
```

4. Beyond traditional syntax and semantics

Traditionally we think of languages as a combination of syntax and semantics and, up until this point, Language Factories have largely been couched in those terms. However modern software practices make use of both wider and narrower knowledge of languages than these notions capture. For example, modern IDEs can perform on-the-fly partial compilation (requiring sophisticated knowledge of the language that a traditional semantics does not need to bother with) and perform code completion (which requires understanding a tractable static subset of the language's semantics). Let us take a simpler example: syntax colouring. It is hard for many of us now to remember the old days when programming meant looking at semi-intelligible green pixels on a charcoal grey background: syntax colouring helps our brain to interpret source code more quickly than before. In order to make syntax colouring practical, AST elements need to be ordered into groups so that colouring can be applied to groups. An extension of ELF could easily cater for this by allowing language components to specify a colouring clause along the lines of:

```
colouring:
  group String: ASCIIString, UnicodeString
  group Number: Int, Float, Fractional
```

This information could then be used to automatically generate the necessary files needed to integrate into an IDE. Similarly, clauses for any other desired tooling related requirements can be defined by a Language Factory as required.

5. Meta-Language Factories

It is often said that the first test of a programming language is whether a compiler for it can be written in the language itself (thus finishing the bootstrapping of the language). Language Factories do share many similarities to programming languages and compilers, so an important question to ask is whether a Language Factory could be used to produce other Language Factories. The answer is clearly 'yes': Language Factories can specify and implement other Language Factories. When a Language Factory is used to realise another Language Factory we refer to it as a meta-Language Factory[2]. We do not envision that there will be a single meta-Language Factory, since Language Factories as a concept is in many ways more similar to (loose) design patterns than (strict) formal languages: different classes of Language Factories will require different meta-Language Factories.

6. Discussion and comparison

6.1 Advantages of Language Factories

There are two ways in which we imagine Language Factories shaping the future. First, and most obviously, Language Factories make the design and implementation of new languages (mostly in the form of DSLs) a realistic prospect for a much greater number of people than was previously the case. By making language components reusable, parameterisable, and composable, Language Factories can significantly reduce the burden associated with language creation.

Second, and thinking further ahead, Language Factories offer the potential to provide a new level of abstraction over libraries and frameworks, which are forced to express, often complex, domain specific information through the strait-jacket of normal programming languages. As shown by Stratego/XT [Bravenboer and Visser 2004], adding syntax and semantics specific to a library or framework can make using it significantly easier. As shown in the case study, since Language Factories can target existing programming languages, they provide all the necessary tools to make this a practical reality.

6.2 Comparison to related approaches

There are several existing techniques, tools, and languages to which Language Factories can be compared.

MDA [OMG 2003] and Software Factories [Greenfield et al. 2004] both share similarities of outlook with Language Factories, being (at least in part) based on the idea of building systems from components. However neither vision has yet been realised, in part, we assert because of the overly general problem they attempt to tackle: automating the process of building arbitrary software systems still seems to be immensely hard. Language Factories tackle a more tractable problem since, as we have shown in this paper, languages

[2] Note that meta is a relative term in this instance: there is no notion of an absolute meta-Language Factory.

often naturally decompose into components and the formalisms underlying languages are relatively well developed and understood, even if they have rarely (before Language Factories) been explicitly integrated together.

Extensible programming languages such as Lisp and Converge typically use compile-time meta-programming (often called 'macros', although that is more properly thought of as a specific form of compile-time meta-programming) to allow programs to embed different syntaxes and have them transformed into the base language. This class of languages is homogeneous in the sense that the language used to specify the transformation is the same as the language being translated into [Sheard 2001]. Homogeneous languages by their very nature are restricted to a single host language; while this tight coupling can bring benefits (see [Tratt 2008] for details), it is also inherently limiting. In their particular field, homogeneous languages will inevitably outperform Language Factories; however, because they need not be tied to any particular language, Language Factories' field is so much larger that the overlap is almost trivial.

AST-based systems such as JetBrain's MPS[3] and Intentional's tool [4] both work on the basis that users edit AST's, not text (i.e. both are syntax directed editors which do not allow the user to enter ill-formed ASTs). Fundamentally, both systems require programs written in their languages to be edited exclusively in their tools. While both go out of their way to make such editing more pleasant than previous generations of syntax directed editors, this is a serious restriction: it restricts some types of additional tooling; it hampers interoperability; and quite possibly alienates many potential users before they have even started. While some Language Factories may choose to use syntax directed editing, with the inevitable accompanying restrictions, most Language Factories are unlikely to choose to be so prescriptive.

Perhaps the closest extant technology to Language Factories are the term rewriting systems, the most advanced of which is arguably Stratego/XT [Bravenboer and Visser 2004]. Like Language Factories, Stratego/XT can arbitrarily compose together syntaxes, and are not restricted in the languages they compose or the languages they translate into (see e.g. [Visser 2008] for a large scale example). Unlike Language Factories, Stratego/XT has little or no knowledge of the semantics of the languages it is expressing: it is easy to build nonsensical intermediate representations, which can then cause chaos when debugging. Perhaps even more fundamentally, Language Factory language components have significant semantic information attached to them (e.g. constraints on the components they can be composed with), making the composition of such components much more well-defined than the ad-hoc composition found in current term rewriting systems.

[3] `http://www.jetbrains.com/mps/`

[4] `http://www.intentsoft.com/`

7. Conclusions

In this paper we presented the Language Factories vision. Language Factories can perhaps best be thought of as being similar to design patterns, describing a family of related approaches to component-based language building. We explained, via a case study, how one particular Language Factory allows powerful languages to be built in a far more systematic, flexible way than any existing approach.

Acknowledgments

Thanks to Naveneetha Vasudevan for insightful comments on a draft of this paper.

References

Jon Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In *Proc. OOPSLA'04*, October 2004.

Tony Clark, James Willans, and Paul Sammut. *Applied Metamodelling: A Foundation for Language Driven Development (Second Ed.)*. 2008. `http://itcentre.tvu.ac.uk/~clark/`.

James R. Cordy. TXL - a language for programming language tools and applications. In *Proc. LDTA 2004*, April 2004.

Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2), February 1970.

Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

Paul Hudak. Modular domain specific languages and tools. In *Proc. Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

Mirjana Ivanovic and Viktor Kuncak. Modular Language Specifications in Haskell. In *Theoretical Aspects of Computer Science with practical application*, 2000.

Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

OMG. OMG unified modeling language specification 1.4, 2003. `www.omg.org/docs/omg/03-06-01.pdf`.

Tim Sheard. Accomplishments and research challenges in meta-programming. *Proc. SAIG '01*, 2196:2–44, 2001.

Tim Sheard and Simon Peyton-Jones. Template meta-programming for Haskell. In *Proc. Haskell workshop 2002*. ACM, 2002.

Guy L. Steele, Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221 – 236, October 1999.

Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *TOPLAS*, 30(6):1–40, 2008.

Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *LNCS*, pages 291–373, 2008.

Gregory V. Wilson. Extensible programming for the 21st century. *Queue*, 2(9):48–57, January 2005.