Reusable Components for Lightweight Mechanisation of Programming Languages

Seyed (hossein) Haeri, Sibylle Schupp

Reusable Components for Lightweight Mechanisation of Programming Languages

Seyed H. HAERI (Hossein) and Sibylle Schupp

Institute for Software Systems, Hamburg University of Technology, Germany
{hossein,schupp}@tu-harburg.de

Abstract. Implementing Programming Languages (PLs) has always been a challenge for various reasons. One reason is the excess of routine tasks to be redone on every implementation cycle. This is despite the remarkable fraction of syntax and semantics usually shared between successive cycles. In this paper, we present a component-based approach to avoid reimplementation of shared PL fractions. We provide two sets of reusable components; one for syntax implementation and another for semantics. Our syntax and semantics components correspond to syntactic categories and semantics rules of a PL specification, respectively. We show how, in addition to their service to reusability in syntax and semantics, our components can cater reusable implementation of PL analyses.
Our current level of experimentation suggests that this approach is applicable wherever the following two features are available or can be simulated: Type Constraints and Multiple Inheritance. Implementing a PL using our approach, however, requires some modest programming discipline that we will explain throughout the text.

1 Introduction

Mechanisation of a PL is implementing it for the purpose of experimentally studying its characteristics and conduct. One interacts with the mechanisation to discover otherwise inapparent facts or flaws **in action**. PL mechanisation, however, can become very involved using traditional formal proof systems. Various other frameworks have, therefore, been crafted to help lightweight mechanisation. Different frameworks focus on facilitating different mechanisation tasks.

Mechanisation often enjoys cycles. Repeating implementation upon each cycle can form a considerable burden against mechanisation, especially because consecutive cycles often share a sizeable fraction of their syntax, only differ in few semantic rules, and, rarely add new analyses. Modularity becomes vital in that, upon extension, existing modules ought to be readily reusable.

As such, component-based mechanisation can provide even more reusability by facilitating implementation sharing for individual PL constructs. For example, it is not uncommon for different PLs to be extended using similar constructs (in the syntax or semantics). Component-based mechanisation cancels the need for reimplementing such constructs. In addition, our above interpretation of modularity comes as a side product in that PL modules already composed of components need not to be touched upon the addition of new components.

In this paper, we implement language-independent components for syntactic categories to obtain syntax code reuse. We also offer a collection of semantics-lenient derivation rules that are individually executable. These form our highly flexible components that cater various sorts of semantics code reuse upon composition. A particularly interesting consequence of the flexibility in our syntax and semantics components is analysis code reuse. These components elevate the programming level of mechanisation; they encourage coding in terms of themselves (as opposed to exact type constructors), viz. , addition of new type constructors imposes no recompilation on existing code. (C.f. Expression Problem [29].)

Whilst our use of multiple inheritance caters easy extension of mechanisation, type constraints help the compiler outlaw reuse of mechanisation when conceptually inapplicable. Our approach is applicable independent of the formalism used for specification. Yet, a modest programming discipline is required for enjoying our reusability. Most of the burden is, however, on the Language Definitional Framework (LDF). Our approach indeed minimises the PL implementer's effort when an extensive set of our components is available through the LDF.

We choose to embed our approach in Scala for its unique combination of **built-in** features that suit mechanisation [23]. Both multiple inheritance and type constraints have special flavours in Scala that are not shared universally amongst languages. However, we do not make use of those specialities. Hence, the applicability of our approach is deemed to only be subject to the availability of multiple inheritance and type constraints.

We start by reviewing the related literature in Section 2. In Section 3, we provide a minimal explanation of the Scala features we use. We exemplify our approach using five systems for lazy evaluation that we briefly present in Section 4. Next, in Section 5, we demonstrate our components for both syntax and semantics mechanisation. As an interesting extra consequence of our particular design of components, Section 6 shows how analysis mechanisation reuse is gained. Concluding remarks and discussion on future work come in Section 7. [1]

2 Related Work

Funcons of PLanCompS [13] are composable components for PL mechanisation each of which with a universally unique semantics. Funcons are similar to our syntax components except that, due to our decoupling of syntax and semantics, our components can have multiple pieces of semantics. On the other hand, a funcon's semantics is provided using Modular SOS [21] and Action Semantics [20], whilst we do not demand any particular formalism. The GLoo mini parsers enable scope-controlled extensions to its language by desugaring the extended syntax into core GLoo [19]. SugarHaskell [9] provides similar facilities, but in a layout-sensitive fashion and for Haskell. Polyglot [22] users can extend a PL compiler (including that of Polyglot itself) by providing (one or more) compiler passes that rewrite the original AST into a Java one. The difference between the

[1] online source code available at `http://www.sts.tuhh.de/~hossein/compatibility`.

last three works and ours is that we do not specifically target PL specification through syntactic desugaring. Our semantics components can be used with or without a core semantics, and, are not restricted to any particular formalism.

Two important ingredients of our approach are type constraints and multiple inheritance. Kiama [25] is an LDF that is embedded in Scala. Hence, Kiama does already have all the language support required by our approach. Maude [5], K [11], MMT [4], Redex [10], Liga [14], Silver [28], Rascal [16], UUAG [7], JastAdd [8], and Spoofax [15] are LDFs that ship with their own DSL as the meta-level PL. Maude is the only such LDF with support for both multiple inheritance and type constraints. JastAdd, UUAG, and Rascal each only provide built-in support for half the language features that our approach requires. Only a runtime simulation of our approach is possible in K, Redex, Liga, Spoofax, and Silver.

Finally, Axelsson [2] and Bahr [3] provide Haskell libraries to improve different aspects of embedded DSL mechanisation. They both build on Swierstra's *data types à la carte* [26] and proceed by offering a new abstract syntax model.

3 Scala Syntax

This section introduces the parts of Scala syntax that we use in this paper.

```
1  object O {def apply(n: Int) = ...}
2  class C0 {type NT1 = Int; type NT2}
3  class C1[T] {
4    def m[U]:  Int -> Int = ...
5  }
6  class C2[T1 <: T2]
7  class C3[T1 <: T2{type NT}]
8  class C4[+T]
9  class C5[T <: C2[_]]
10 class C6[T <: C0 with C2[T]]
```

The method `apply` (line 1) tells Scala to expand calls like `O(1)` to `O.apply(1)`. The nested types `NT1` and `NT2` of the class `C0` (line 2) can be referred to as `C0#NT1` and `C0#NT2`, respectively. This contrasts with Scala's dot notation for referring to members of a package. We say that `C0` binds `NT1` to `Int`. The nested type `NT2` is abstract in that `C0` itself does not bind it. Class `C1` is parametrised over type `T` (line 3). Likewise, method `m` is parametrised over type `U` (line 4). The type parameter `T1` of `C2` is constrained by an *upper bound* type `T2` (line 6). As a result, one can only instantiate `C2` with types which inherit from `T2`. The types to instantiate `C3` with need to also have a nested type `NT` (line 7). One can also place constraints on nested types of type parameters. The plus used before type parameter `T` in line 8 implies that when `T1` is a subtype of `T2`, `C4[T1]` is considered a subtype of `C4[T2]`. Use of underscore in line 9 indicates that one can instantiate `C5` with any type that inherits from `C2[T']`, *for some type* `T'`. Type parameters can be more than one, in which case they are separated using commas. Multiple nested types demanded by an upper bound are to be put on separate lines, or, separated using semicolons. Class `C6` places two upper bound

constraints on its type parameter `T` (line 10). Namely, the type parameter `T` has to inherit from both `C0` and `C2[T]`. (Note that `T` itself is used in its own latter upper bound.) Traits are like abstract classes, but can be multiply inherited.

4 The Implemented Family of Operational Semantics

Five systems in the family of lazy evaluation are: Abramsky and Ong [1], Launchbury [17], Sinot [24], van Eekelen and de Mol [27], and Haeri [12]. We will be referring to these as $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{S}_1$, and $\mathcal{S}_2$, respectively. [2] Moreover, we will refer to the syntax of a family member as a *member syntax*, and, to its semantics as a *member semantics*. Section 4.1 provides an overview of the family syntax. Section 4.2 exhibits only the parts of family semantics that we refer to in this paper. The reader may refer to the original papers for more explanation. The aim of this section is to give the reader enough understanding from the family so that they can follow our discussions. We chose these particular five systems because their good proximity makes demonstration easy. However, given the flexibility in our components, our approach is well applicable beyond just these five.

4.1 Syntax

Here, we briefly present the syntax of the implemented family. Notationally, our presentation is not exactly the same as the original ones. We unify the original notations and neglect the minor differences.

	$\mathcal{L}_0$	$\mathcal{L}_1$	$\mathcal{S}_1$	$\mathcal{S}_2$
$e ::= x$	✓	✓	✓	✓
$\lambda x.e$	✓	✓	✓	✓
$e\ x$	✓	✓	✓	✓
$\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e$		✓	✓	✓
$e_1\ \mathsf{seq}\ e_2$			✓	✓

$$b ::= x \mid Z(\overrightarrow{x})$$
$$e ::= b \mid \lambda x.b \mid e\ b \mid \mathsf{let}\ \{b_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e$$
$$v ::= \lambda x.b \mid x\ b_1 \ldots b_n$$

$$v ::= \lambda x.e \qquad \mathcal{L}_0, \mathcal{L}_1, \mathcal{S}_1$$
$$v ::= \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ \lambda x.e \quad (n \geq 0) \quad \mathcal{S}_2$$

$\mathcal{L}_0$= Abramsky and Ong, $\mathcal{L}_1$= Launchbury, $\mathcal{S}_1$= van Eekelen and de Mol, $\mathcal{S}_2$= Haeri

Fig. 1. Syntax for All the Family Members

Figure 1 shows the family syntax where e ranges over expressions, v ranges over values, and, x ranges over variables. The left half of the figure demonstrates the syntax common between $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{S}_1$, and $\mathcal{S}_2$. With the great degree of commonality between the four, we demonstrate them altogether in a single compact form to avoid repetition. Ticks show the constructors in each member syntax.

To the right, the top half shows complete $\mathcal{L}_2$ syntax, which is a bit different from the previous four. Here, $\overrightarrow{x}$ is a short form for $x_1\ x_2 \cdots x_n$ where $n \geq 2$. In $\mathcal{L}_2$, the *generalised identifier* b ranges over ordinary variables and *meta*variables ($Z(\overrightarrow{x})$). Function applications are likewise generalised. That is, application of

[2] $\mathcal{L}$ for lazy evaluation and $\mathcal{S}$ for selective strictness

functions is allowed to metavariables as well as variables. λ-abstractions are the only values at $\mathcal{L}_0$, $\mathcal{L}_1$, and $\mathcal{S}_1$. Whereas, in $\mathcal{S}_2$, let-*surrounded* λ-abstractions are also considered values, where $\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,\lambda x.e$ is a syntactic sugar for $\lambda x.e$ when $n = 0$. In the $\mathcal{L}_2$ syntax, successive applications of a variable to generalised identifiers $(x\ b_1 \cdots b_n)$ is also considered a value. We refer to this syntactic category as the *variable-to-identifier-applications*. In the entire family, subscripts do not impact the syntactic category, and, are allowed to be arbitrary.

4.2 Semantics

$$\frac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ (\mathbf{lam})_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{L}_2} \qquad \frac{\Gamma : e \Downarrow \Delta : v}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto v) : v}\ (\mathbf{var})_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{S}_2,\mathcal{L}_2}$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow \Theta : v}{\Gamma : e\ x \Downarrow \Theta : v}\ (\mathbf{app})_{\mathcal{L}_1,\mathcal{S}_1}$$

$$\frac{(\Gamma, x_i \mapsto e_i)_{i=1}^{n} : e \Downarrow \Delta : v}{\Gamma : \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,e \Downarrow \Delta : v}\ (\mathbf{let})_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{L}_2} \qquad \frac{\Gamma : e_1 \Downarrow \Theta : v_1 \quad \Theta : e_2 \Downarrow \Delta : v_2}{\Gamma : e_1\ \mathsf{seq}\ e_2 \Downarrow \Delta : v_2}\ (\mathbf{seq})_{\mathcal{S}_1,\mathcal{S}_2}$$

$\mathcal{L}_1$= Launchbury, $\mathcal{S}_1$= van Eekelen and de Mol, $\mathcal{S}_2$= Haeri, $\mathcal{L}_2$= Sinot

Fig. 2. Selected Parts of the Family Semantics

Figure 2 shows selected parts of the family semantics. Here, rule labels are of the form $(\mathbf{r})_\ell$ where r is the rule name and ℓ is the list of family members containing r in their semantics. Rules are of the form $\Gamma : e \Downarrow \Delta : v$ where capital Greek letters denote *heap*s. This rule form reads: Evaluation of e in Γ results in v and updates the bindings to Δ. We write $\Gamma : e \Downarrow_\Pi \Delta : v$ to emphasise that Π is the derivation tree for $\Gamma : e \Downarrow \Delta : v$. In $\mathcal{L}_1$'s terminology, heaps are partial functions from variables to expressions. $\mathcal{S}_1$ as well as $\mathcal{S}_2$ inherit the same terminology. In the semantics of $\mathcal{L}_2$, however, the domain of heaps consists of the set of variables **and** metavariables. $e[x/y]$ denotes capture-avoiding substitution of variable x in e by variable y. All the family members have a *distinct-name convention*, i.e., variable names are supposed to be distinct.

5 Components

Figure 3 gives a UML overview of our approach. At the top, elements of our approach for syntax mechanisation are illustrated, and, at the bottom, those of semantics. The left portions are class diagrams. The right portions are use cases for the left portion of the same row. We use a number of non-standard UML notations: The type Exp with which a LazyExp (top left portion) is instantiated has to inherit from LazyExp itself. There are similar constraints on the type parameter Exp of OpSem as well as Exp and OS of the apply method of Executable

Rule (all in the bottom left portion). Abusing the UML notation, we draw generalisation arrows that extend from the right portions (use cases) to the respective left (class diagrams). For instance, L1OpSem inherits from OpSem. Finally, in the top row, we use a non-standard dashed arrow "ntb" to specify that an Expression Trait binds nested types to its type constructors. As an example, L1Exp binds its nested type Val to its type constructor Lam. (See the top right portion.)

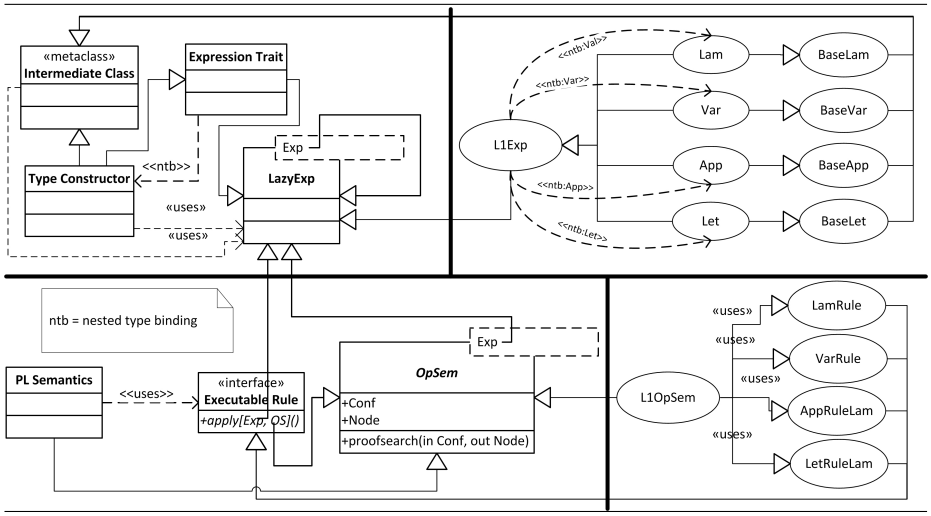

Fig. 3. Architecture of our Approach

Ideally, of the elements depicted in Figure 3, certain ones ought to be shipped by the LDFs. The PL implementer, then, uses these shipped elements for mechanisation of the desired PL. The Intermediate Class instances, LazyExp, the Executable Rule instances, and OpSem are of the former sort. (In Figure 3, BaseLam, BaseVar, BaseApp, and BaseLet are intermediate classes, whilst LamRule, VarRule, AppRuleLam, and LetRuleLam are executable rules.) The top right portion summarises how to mechanise the $\mathcal{L}_1$ syntax using shipped elements of the left half of the same row. The bottom right portion does the same for the $\mathcal{L}_1$ semantics.

For reasons of improved correctness and reusability that we explain later, we need to check whether a PL syntax contains a certain syntactic category or not. Implementing a syntax using a typical algebraic datatype will, hence, not suffice. This is mainly because an ordinary algebraic datatype does not provide a mechanism for **programmatically** querying its type constructors. In Section 5.1 where our syntax components are presented, we employ nested types as an *extra storage* that make such programmatic queries possible. Our approach enjoys a design-by-contract flavour in that the names and duties of these nested types are dictated by the intermediate classes – at their design time. This flavour will also be available in Section 5.2, where our semantics components are introduced.

5.1 Syntax Components

In an embedded setting, mechanisation of a PL syntax typically involves embedding its abstract and concrete syntax, pretty-printing and the rest of debugging cosmetics, and sanity checks. One can of course perform the same task repeatedly for each member syntax. This entails a total of at least 24 type constructors for Figure 1 with a great amount of reimplementation in the above syntax mechanisation tasks. We instead mechanise a PL syntax in terms of our reusable components that serve as syntactic building blocks. Each of these components – called "intermediate classes" – corresponds to one and only one syntactic category. (See Intermediate Class in the top left portion of Figure 3.) An intermediate class implements its *own part* of an abstract syntax, pretty-printing, and sanity checks. This one-off implementation, then, is readily available to whatever PL syntax that contains the respective syntactic category, and, can be used off-the-shelf. For example, for Figure 1, we have implemented a total of 9 intermediate classes that correspond to variables (`BaseVar`), λ-abstractions (`BaseLam`), function applications (`BaseApp`), let-expressions (`BaseLet`), selective strictness (`BaseSeq`), metavariables (`BaseMVar`), generalised identifiers (`BaseGenIdn`), let-surrounded λ-abstractions (`BaseVal`), and variable-to-identifier-applications (`BaseVarApp`). On the other hand, we embed concrete syntax in terms of `LazyExp` – once and for all. `LazyExp` is our root of expressions for the entire family. (Compare with LazyExp in the top left portion of Figure 3.)

In Section 5.1.1, we first explain how to put our syntax components together to gain a complete syntax mechanisation. We, then, take a deeper look into some internals of our code which made this possible in Section 5.1.2.

5.1.1 Syntax Mechanisation When appropriate intermediate classes and `LazyExp` are at hand, a simple discipline needs to be followed:

- A member syntax is mechanised using its own (algebraic data-) type. Such a type provides its extra storage using binding nested types to its respective type constructors. When a syntax is mechanised in such a fashion, we refer to its implementing datatype as an "expression trait." Furthermore, when a type constructor is bound in such a fashion, we say it is *register*ed (at the expression trait). To inherit the concrete syntax embedding, and to be of use to our semantics mechanisation utilities, an expression trait `T` always derives from `LazyExp[T]`.
- Type constructors themselves need as well to specify which syntactic category they belong to. With their type parameters that will be explained later, we consider our intermediate classes only *half-baked*. We say that an intermediate class gets *fully-baked* for an expression trait `T` when a type constructor of `T` inherits from their instantiation for `T`.

Figure 4 exemplifies our discipline for $\mathcal{L}_1$. Scala uses normal inheritance as a simple facility for extensible algebraic datatypes. Accordingly, `Var`, `Lam`, `App`, and `Let` (lines 11-14) are type constructors of `L1Exp`. They are fully-baked for $\mathcal{L}_1$'s

```
1  package l1
2
3  sealed trait L1Exp extends LazyExp[L1Exp] {
4    type Val = l1.Lam
5    type App = l1.App
6    type Var = l1.Var
7    type Let = l1.Let
8    ...
9  }
10
11 final case class Var(...) extends BaseVar(...)         with L1Exp
12 final case class Lam(...) extends BaseLam[L1Exp](...) with L1Exp
13 final case class App(...) extends BaseApp[L1Exp](...) with L1Exp
14 final case class Let(...) extends BaseLet[L1Exp](...) with L1Exp
```

Fig. 4. Mechanisation of the $\mathcal{L}_1$ Syntax Using our Programming Discipline

expression trait (`L1Exp`) because they inherit from `BaseVar`, `BaseLam[L1Exp]`, `BaseApp[L1Exp]`, and `BaseLet[L1Exp]`, respectively. (See Figure 1 for the syntactic categories of $\mathcal{L}_1$.) These four type constructors are registered in lines 4-7 where they get bound to the nested types `Val`, `App`, `Var`, and `Let` of `L1Exp`, respectively. (C.f. Figure 4 with the top right portion of Figure 3.)

5.1.2 Technicality In order for an intermediate class not to be exclusively suitable to a single PL, it has to be parameterised over the PL syntax. However, not every syntactic category is suitable to every PL syntax. An intermediate class has to act accordingly. Consider `BaseLet`, for example:

```
1 class BaseLet[+Exp <: LazyExp[_]{type Let <: BaseLet[_]}]
2   (val bs: Map[Idn, Exp], val e: Exp) {
3     if(...) //value type == λ-abstractions
4       require(!bs.isEmpty)
5     override def toString() = ...
6 }
```

Lines 3 and 4 above perform a sanity check pertaining to let-expressions. (It is only in $\mathcal{S}_2$ – where let-surrounded λ-abstractions are value types – that empty let-bindings are allowed.) Line 5 handles the pretty-printing. The constructor parameters `bs` and `e` (line 2) embed the abstract syntax part of this intermediate class. Note that the type of the latter parameter is not fixed. Instead, it is typed using the type parameter `Exp` (line 1). The upper bound on `Exp` makes `BaseLet` invariably available to every member syntax so long as `Exp` registers its `BaseLet`-derived type constructor under the name `Let`. (Namely, $\mathcal{L}_0$ is excluded.) Similar type constraints selectively determine the appropriate *classes of syntax*.

It remains to further expand on the role of `LazyExp`. In this section, we focus only on the syntactic parts of its role. Section 5.2 explains its role for semantics mechanisation. We implement all our concrete syntax embedding for `LazyExp` – once and for all. When applicable, a member syntax reuses the same embedding

through inheritance of its expression trait from `LazyExp`. The following table summarises our concrete syntax embedding: In each row, the code on the left gets automatically desugared into a piece of abstract syntax that represents the mathematical expression on the right. (`T` in line 2 is the corresponding expression trait of `e`.)

	code	math
1	`e("x1")...("xn")`	$((e\ x_1)\ \cdots\ x_n)$
2	`\[T]("x1",...,"xn")(e)`	$\lambda x_1 \cdots x_n.e$
3	`let ("x1" -> e1,...,"xn" -> en) in e`	$\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e$
4	`e1 seq e2`	$e_1\ \mathsf{seq}\ e_2$

Note that the code in line 2 embeds a let-surrounded λ-abstraction in $\mathcal{S}_2$'s syntax and ordinary λ-abstractions in other family members. Likewise, when `e` is a λ-abstraction, the code in line 3 embeds another λ-abstraction for $\mathcal{S}_2$'s syntax, and, let-expressions otherwise. On the other hand, the embedding in line 4 is only applicable when a syntactic category is available for selective strictness. Again, the selectivity on the classes of syntax is enabled by type constraints.

As also explained further above, our convention is that an expression trait `T` must derive from `LazyExp[T]`. (See line 3 in Figure 4 for `L1Exp`, for example.) Given that `T`'s type constructors inherit from it, they also inherit from `LazyExp` by transitivity of inheritance. Note how following our convention for expression traits makes them distinguishable from type constructors. The particular wiring used below for the type parameter `Exp` of `LazyExp` enforces the above convention. Section 5.2 contains an example where this convention comes handy.

```
1 trait LazyExp[+Exp <: LazyExp[Exp]] {...}
```

5.2 Semantics Components

Similar to the case for syntax, it is perfectly possible to program each member semantics separately. That is a total of 23 rules for the entire family, and, with a great deal of code repetition. Instead, we implement a collection of 14 reusable and executable rules, which can be plugged into a PL semantics mechanisation. Our design ships another artefact as well: `OpSem` is our root operational semantics class. This is an abstract base class with a method for distributing the semantics evaluation between executable rules. Yet, `OpSem` is flexible on its input/output to the extent that it allows several semantics mechanisations for a single syntax. In this paper, we only demonstrate the idea for rules which document the entire semantics evaluation, if successful. However, one can easily configure `OpSem` for rules which, for instance, merely work with the PL objects involved in the semantics specification. Although we only demonstrate mechanisation for operational semantics, we have no evidence to doubt the applicability of our approach to other formalisms. After all, it only amounts for the semantic rules to be implemented like our executable rules.

In Section 5.2.1, we first explain how to combine our components to mechanise a PL semantics. A closer look into our components themselves is then

provided in Section 5.2.2. Whilst the latter section targets LDF implementers more, the former one is more useful to the LDF users.

5.2.1 Semantics Mechanisation Assuming the availability of our components, the following programming discipline needs to be followed for semantics mechanisation: A member semantics is implemented as a stand-alone object that derives from `OpSem[T]`, where `T` is the expression trait of the corresponding member syntax. This object needs to implement a method `proofsearch` which distributes evaluation between pertaining executable rules. In such a case, we say the member semantics *plugs* its appropriate executable rules.

```
1  object opsem extends OpSem[LExp] {
2    type Conf = HBConf[LExp]
3    type Node = HBNode[LExp]
4
5    def proofsearch(g: LHeap, e: LExp): HBNode[LExp] = e match {
6      case l: Lam => HBLamRule[LExp, opsem.type](g, l)
7      case a: App => HBAppRuleLam[LExp, opsem.type](g, a)
8      case v: Var => HBVarRule[LExp, opsem.type](g, v)
9      case l: Let => HBLetRuleLam[LExp, opsem.type](g, l)
10   }
11   override def proofsearch(c: Conf): HBNode[LExp] =
12     proofsearch(c._1, c._2)
13 }
```

Fig. 5. Implementing the $\mathcal{L}_1$ Operational Semantics in Isolation

For example, for $\mathcal{L}_1$'s semantics, the method `proofsearch` in Figure 5 takes a heap along with an expression (line 5), and, produces a derivation tree, when successful: Here, the plugged executable rules are `HBLamRule`, `HBAppRuleLam`, `HBVarRule`, and `HBLetRuleLam` (lines 6 to 9, respectively) that we schematically depicted in Figure 3. (In our naming convention, prefix `HB` indicates a *heap-based* system. That includes all the family members in this paper except $\mathcal{L}_0$.) What comes after the executable rule names in square brackets is to guide Scala's type deduction. `HBNode` is the root of our hierarchy for nodes in the heap-based derivation trees. Each node class encapsulates relevant compile time/runtime sanity checks that make it easier to enforce correctness of the executable rules. Armed with such correctness enforcement mechanisms, the compiler would have stopped us, for any of the four cases, had we plugged in a rule which is incompatible with the respective characteristics of either $\mathcal{L}_1$'s syntax or semantics.

5.2.2 Technicality Implementing a rule in a way that is not exclusively useful to a particular PL entails parameterising it over both the syntax and semantics. And, indeed the type parameters of our executable rules *characterise* both the syntax and semantics they expect. Our rules can be plugged into any semantics

so long as their characteristic expectations hold. A compile error will be emitted otherwise. For example, below is our **(let)**$_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{L}_2}$ implementation:

```
1  object HBLetRuleLam {
2    def apply[Exp <: LazyExp[Exp]{type Val <: BaseLam[Exp]
3                                  type Let <: BaseLet[Exp]},
4              OS <: OpSem[Exp]{type Conf = HBConf[Exp]
5                               type Node = HBNode[Exp]}]
6      (g: Heap[Exp], lexp: Exp#Let with Exp)
7      (implicit opsem: OS): HBNode[Exp] = {
8        val (e, bs) = (lexp.e, lexp.bs)
9        val pi = opsem.proofsearch(g ++ bs, e)
10       val (d, z) = (pi.g2, pi.e2)
11       new HBLetNodeLam[Exp](pi, g, lexp, d, z)
12   }
13 }
```

The type parameters `Exp` and `OS` above (lines 2 and 4) signify the expression trait and the operational semantics, respectively. However, not every rule in Figure 2 is a part of every member semantics. For example, this **(let)** rule is only a part of the $\mathcal{L}_1$, $\mathcal{S}_1$, and $\mathcal{L}_2$ semantics. The constraint `type Val <: BaseLam[Exp]` (line 2) rules out $\mathcal{S}_2$. This constraint enforces on `Exp` the availability of a nested type `Val` that binds to a class derived from `BaseLam`, i.e., that λ-abstractions is a value type of the syntax. (See Figure 1.) `OS <: OpSem[Exp]` states that `OS` must be an operational semantics type over the expression type `Exp`. (More on `OpSem` shortly.) The constraint `type Conf = HBConf[Exp]` (line 4) on `OS` states that it inputs a pair of heap and expression. Similarly, `type Node = HBNode[Exp]` (line 5) specifies that `OS` outputs a heap-based derivation tree. These constraints rule out the semantics of $\mathcal{L}_0$ too, making `HBLetRuleLam` only applicable to the right family members. Lastly, note how the treatment of type parameters enables `opsem` to take a continuation-passing style role for handling "the rest of the evaluation" – again, only for the correct family members.

Here is a recap on the remaining points: The constraint `type Let <: BaseLet[Exp]` (line 3) ensures that `Exp` registers its `BaseLet`-derived constructor under the name `Let`. Furthermore, `lexp` (in line 6) is required to be an instance of both `Exp` and this registered type. In other words, `lexp` needs to be constructed using a type constructor of `Exp` that corresponds to let-expressions. Recall also that, as seen at the end of Section 5.1, the constraint `Exp <: LazyExp[Exp]` (line 1) ensures that `Exp` is an expression trait. `HBLetNodeLam` inherits from `HBNode` to be the node for let-expressions where λ-abstractions are a value type.

It remains to consider our `OpSem` trait:

```
1 trait OpSem[Exp <: LazyExp[_]] {
2   type Conf
3   type Node <: ProofTree
4   def proofsearch(c: Conf): Node
5 }
```

For each operational semantics, `proofsearch` inputs the initial *configuration*, and, produces the derivation tree according to the rules of the semantics. The ab-

stract type `Conf` represents the type signature of the input (line 2). Likewise, the abstract type `Node` is the derivation tree type an operational semantics outputs (line 3). The type `ProofTree` is our generic ADT for derivation trees.

6 Case Study: Analysis Code Reuse

Like the case of syntax and semantics, one should be able to reuse the code for analyses implemented over previous mechanisation cycles – but, only when they are still conceptually applicable. We address that need based on two facts:

Fact 1. Old code that is implemented in terms of the root of a hierarchy works for new classes that derive from the root.

Fact 2. Code that constraints its type parameters can employ the compiler to prevent its use for wrong types.

Information gathering over derivation tree traversals is the essence of many analyses. A crude idea can, thus, be implementing all the analyses over a single generic tree type. However, such a tree is unaware of the types its nodes contain. One would rather make all the derivation tree types **inherit** from such a generic type. This way, old code which operates on the generic type can remain intact over the addition of new derivation types. (C.f. Fact 1.) More precision can also be gained by giving this hierarchy extra intermediate nodes. On the other hand, by constraining the type parameters of analysis implementations, one can avoid their wrong application. Constraints can enforce applicability of an analysis to all derivation trees that say derive from a certain base. (C.f. Fact 2.) We call the process of organising derivation tree types in a hierarchy and implementing analyses in terms of the suitable hierarchy node "multi-levelling analyses."

Our hierarchy of derivation trees is rooted in `ProofTree` (seen first in Section 5.2). `ProofTree` has a minimal understanding of what it contains. All it knows is that a set of premisses leads to a conclusion using a rule label. `HBNode` and `HLNode` extend `ProofTree` for heap-based nodes and the heap-less ones, respectively. Nodes which represent derivation in the $\mathcal{L}_0$ operational semantics are instances of `HLNode`. All other nodes are of type `HBNode`. Both `HBNode` and `HLNode` provide more specific information. For example, the former also knows that its conclusion is always a 4-tuple for the $\Gamma : e \Downarrow \Delta : v$ scheme. Further down in the hierarchy come nodes that correspond to semantics rules, and hence, executable rules. These latter nodes know the syntactic category of their e in the above scheme. For instance, `HBVarNode` that corresponds to $(\mathbf{var})_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{S}_2,\mathcal{L}_2}$ knows that it works on variables. At the same level are types for the derivation trees of the individual family members. `L1Node`, for instance, is that of $\mathcal{L}_1$. Obviously, `L1Node` has more specific information at hand, e.g., the exact type of expressions/heaps it works with.

Generally, analyses remain invariably useful over several mechanisation cycles so long as they are implemented in terms of the right level at the derivation tree hierarchy. Most of what makes such a hierarchical craft of derivation trees helpful stems from the high degree of flexibility in executable rules. The `apply`

methods of the executable rules presented in this paper all have `HBNode` return types. However, it is trivial to configure executable rules otherwise and still enjoy them as reusable components for semantics mechanisation. We also gain other sorts of analysis reusability from the high degree of reusability in intermediate classes. For example, an analysis which deals with evaluation of a particular syntactic category can remain intact over consecutive mechanisation cycles even though the actual type constructors involved vary across the cycles. We will not demonstrate reusability of this latter sort in this paper.

As a first example, consider an analysis the right level for in our hierarchy is `ProofTree`: Counting the number of rules used over a derivation.

```
1 def rulecount(p: ProofTree): Int = if (p.prems.isEmpty) 1
2                                    else (0 /: p.prems) (_ + rulecount(_))
```

This analysis does not need any knowledge about the types involved over the proof search. It is a simple folding action over the premisses (line 2) with axioms as the basis of induction (line 1). An occasion where this counting might be useful is comparing the cost of designated computations across different semantics which are known to be observationally equivalent.

Our second example is on the analyses in Definition 1, which play a central role in the observational equivalence theorems on $\mathcal{S}_2$ [12]:

Definition 1. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$. *Define* $\mathit{diff}(\Pi) = \{x \in \mathit{dom}(\Gamma) \mid \Gamma(x) \neq \Delta(x)\}$. *Call* x ***atomic*** *in* Γ *when there exist* Δ_x, v_x, *and* Π_x *such that* $\Gamma : x \Downarrow_{\Pi_x} \Delta_x : v_x$ *and* $\mathit{diff}(\Pi_x) = \{x\}$.

In fact, as opposed to only $\mathcal{S}_2$, *diff* and *atomic* are analyses applicable to any heap-based semantics. Here is how we employ that observation:[3]

```
1  object diff {
2    def apply[Exp <: LazyExp[Exp]](pi: HBNode[Exp]): Set[Idn] =
3      for(x <- pi.g.dom; if(pi.g(x) != pi.d(x))) yield x
4  }//diff(pi) = {x ∈ dom(pi.g) | pi.g(x) != pi.d(x)}
5  object atomic {
6    def apply[Exp <: LazyExp[Exp]{type Var <: BaseVar},
7              OS <: OpSem[Exp]{type Conf = HBConf[Exp]
8                               type Node = HBNode[Exp]}]
9      (g: Heap[Exp], x: Idn)//x is atomic in g when...
10     (implicit opsem: OS, variabliser: Idn => Exp with Exp#Var): Boolean =
11       diff(opsem.proofsearch(g, x)) == Set(x)//... diff(pi_x) == {x},
12 }//where pi_x = opsem.proofsearch(g, x).
```

Notice that `atomic` characterises the syntax and semantics it is applicable to through the constraints on the type parameters (lines 6-8). Consequently, it remains applicable upon extensions of mechanisation so long as the characteristics remain intact. Thanks to our multi-levelling, in the implementation of `diff`, types are all correctly identified by the compiler. We would have not had such a pleasure, had we implemented it on `ProofTree`, which is oblivious of the types

[3] `variabliser` is our implementation detail in charge of reifying an identifier into an expression of the right type (i.e., a variable).

inside it. This static safety becomes clearer in the next example where we examine order of evaluation of variables for any heap-based semantics:

```
1  object EvalList {
2    def apply[Exp <: LazyExp[Exp]{type Var <: BaseVar}]
3      (hbn: HBNode[Exp]): List[Idn] = hbn match {
4        case HBVarNode(pi, g, xvar, d, _) => {//(var)_{L1,S1,S2,L2} in Fig. 2:
5          val prev = EvalList(pi)//what gets evaluated in the premisses...
6          val x = xvar.name
7          if(g(x) != d(x)) (prev:::List(x)) else prev
8        }//... plus x itself when g(x) != d(x).
9        case _ => //Otherwise: union what is evaluated in the premisses.
10         (for(p <- hbn.ps) yield EvalList(p)).toList.flatten
11   }//Note: hbn.ps == premisses of hbn
12 }
```

`EvalList` produces the list of evaluated variables in order. Due to space restrictions, we only report how the above single implementation of `EvalList` remains applicable upon extending mechanisation of $\mathcal{L}_1$ to $\mathcal{S}_2$. Let heap $\Gamma = \{id \mapsto \lambda t.t, y \mapsto (\lambda t_1 t_2.t_1)\ id\ x, x \mapsto (\lambda t_1 t_2.t_2)\ id\}$ be represented by `g`. For $\mathcal{L}_1$, `EvalList(g <::> "y")` produces `List(y)`, whilst `List(x, y)` is produced by `EvalList(g <::> ("x" seq "y"))` for $\mathcal{S}_2$. [4]

Due to multi-levelling, Scala precisely infers the types for `pi`, `xvar`, `g`, and `d`. There is no need for runtime casting. To get multi-levelling, we identified that this analysis is applicable to any heap-based derivation tree on expression traits with a syntactic category for variables. We enforced that by making `EvalList` applicable to any such tree through placing the type constraints at line 2. It is exactly this constraint that creates a flow of type information that automates type inference of the above variables. In the absence of that type information in scope, one has to manually set variable types and/or even resort to runtime casting to calm the type system.

7 Conclusion and Future Work

In this paper, we present our components for syntax and semantics mechanisation. As a driving example, we use five systems for lazy evaluation to show how these components can serve reusability in the mechanisation of PL syntax, semantics, and analysis. We also discuss the internals of our components and how, using type constraints and multiple inheritance, they are engineered for this particular sort of code reuse.

Using our components imposes some modest programming discipline. A PL implementer's part of this discipline is indeed minimal. And, yet, the effort to suit the reusability is incomparably smaller than reimplementation: For syntax, this effort amounts to simply deriving from an extra base class (i.e., intermediate classes for each type constructor) and binding the type constructors under some fixed nested type of the expression trait (Figure 4). For semantics, it takes

[4] `g <::> e` abbreviates `os.proofsearch(g, e)` when `os` is an implicit in scope.

deriving from an abstract base class (e.g., `OpSem`) and implementing a method (like `proofsearch`) that merely distributes the evaluation task between the appropriate executable rules (Figure 5).

Shipping our components is certainly a new burden on LDFs. Implementing our components can sometimes become clunky. This is mainly because, working with constrained type parameters as opposed to exact types makes some extra indirection inevitable. The burden on LDFs magnifies are they to ship an exhaustive set of our components which the PL implementer can freely mix-and-match; that is, after all, likely to take several rounds of refactoring on its way.

Whether or not our approach will scale is a topic for further research. One might also study the classes of extensions in terms of the refactoring they dictate. For example, having had implemented our approach for the other four family members, addition of $\mathcal{L}_2$ dictated some refactoring to our codebase. Regarding further extensions, the effort might vary: For example, adding integer arithmetic as sketched in the $\mathcal{L}_1$'s original paper [17] is routine. Addition of Eden's strict function application [18] would also be relatively easy. However, we anticipate that adding the lazy evaluation material of Danvy et al. [6] needs refactoring.

Our components enjoy composability, but, are not atomic. That is, whilst it is trivial to compose our components to acquire new ones, not every semantic rule can be composed out of existing ones. For example, the subtle difference between $\textbf{(app)}_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{L}_2}$ and the function application rule of $\mathcal{S}_2$ means that neither can be implemented in terms of another. The study of atomic support for implementing our components is yet another future work.

References

1. S. Abramsky and C.-H. Ong, *Full Abstraction in the Lazy Lambda Calculus*, Inf. & Comp. **105** (1993), no. 2, 159–267.
2. E. Axelsson, *A Generic Abstract Syntax Model for Embedded Languages*, Proc. 17^{th} ACM SIGPLAN Int. Conf. Func. Prog. (New York, NY, USA), ACM, 2012, pp. 323–334.
3. P. Bahr and T. Hvitved, *Parametric Compositional Data Types*, Proc. 4^{th} W. Math. Struct. Funct. Prog. (J. Chapman and P. B. Levy, eds.), Elec. Proc. Theo. Comp. Sci., vol. 76, February 2012, pp. 3–24.
4. F. Chalub and C. Braga, *Maude MSOS Tool*, ENTCS **176** (2007), no. 4, 133–146.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *The Maude 2.0 System*, Rewriting Techs & App. (RTA 2003) (R. Nieuwenhuis, ed.), LNCS, no. 2706, Springer-Verlag, June 2003, pp. 76–87.
6. O. Danvy, K. Millikin, J. Munk, and I. Zerny, *On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation*, Theo. Comp. Sci. **435** (2012), 21–42.
7. A. Dijkstra, J. Fokker, and S. D. Swierstra, *The Architecture of the Utrecht Haskell Compiler*, Proc. 2^{nd} ACM SIGPLAN Symp. on Haskell (New York, NY, USA), ACM, 2009, pp. 93–104.
8. T. Ekman and G. Hedin, *The JastAdd Extensible Java Compiler*, Proc. 22^{nd} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl., 2007, pp. 1–18.

9. S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann, *Layout-sensitive Language Extensibility with SugarHaskell*, Proc. 5^{th} ACM SIGPLAN Symp. on Haskell (J. Voigtländer, ed.), ACM, September 2012, pp. 149–160.
10. M. Felleisen, R. B. Findler, and M. Flatt, *Semantics Engineering with PLT Redex*, MIT Press, 2009.
11. T. Florin Şerbănuţă, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Roşu, *The K Primer (version 2.5)*, K'11 (M. Hills, ed.), ENTCS, to appear.
12. S. H. Haeri, *Observational Equivalence and a New Operational Semantics for Lazy Evaluation with Selective Strictness*, Proc. Int. Conf. Theo. & Math. Found. Comp. Sci. (TMFCS-10), 2010, pp. 143–150.
13. A. Johnstone, P. D. Mosses, and E. Scott, *An Agile Approach to Language Modelling and Development*, Innovations in Sys. & Soft. Eng. **6** (2010), 145–153.
14. U. Kastens and M. W. Waite, *Modularity and Reusability in Attribute Grammars*, Acta Informatica **31** (1994), 601–627.
15. L. C. L. Kats and E. Visser, *The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs*, Proc. 25^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (New York, NY, USA), ACM, 2010, pp. 444–463.
16. P. Klint, T. van der Storm, and J. Vinju, *EASY Meta-programming with Rascal*, Gener. & Transform. Techs Soft. Eng. III (J. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, eds.), LNCS, vol. 6491, Springer Berlin/Heidelberg, 2011, pp. 222–289.
17. J. Launchbury, *A Natural Semantics for Lazy Evaluation*, Proc. 20^{th} ACM SIGPLAN-SIGACT Symp. Princ. Prog. Lang., ACM, 1993, pp. 144–154.
18. R. Loogen, Y. Ortega-mallén, and R. Peña-marí, *Parallel Functional Programming in Eden*, J. Func. Prog. **15** (2005), no. 3, 431–475.
19. M. Lumpe, *Growing a Language: The* GLoo *Perspective*, Soft. Composition (C. Pautasso and É. Tanter, eds.), LNCS, vol. 4954, Springer, 2008, pp. 1–19.
20. P. D. Mosses, *Theory and Practice of Action Semantics*, Math. Found. Comp. Sci., 21^{st} Int. Symp., LNCS, vol. 1113, Springer, September 1996, pp. 37–61.
21. ———, *Modular Structural Operational Semantics*, J. Logic & Alg. Prog. **60–61** (2004), 195–228.
22. N. Nystrom, M. R. Clarkson, and A. C. Myers, *Polyglot: An Extensible Compiler Framework for Java*, Compiler Constr., 12^{th} Int. Conf., LNCS, vol. 2622, Springer-Verlag, April 2003, pp. 138–152.
23. M. Odersky and M. Zenger, *Scalable Component Abstractions*, Proc. 20^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (New York, NY, USA), ACM, 2005, pp. 41–57.
24. F.-R. Sinot, *Complete Laziness: a Natural Semantics*, ENTCS **204** (2008), 129–145.
25. A. Sloane, *Lightweight Language Processing in Kiama*, Gener. & Transform. Techs Soft. Eng. III (2011), 408–425.
26. W. Swierstra, *Data Types à la Carte*, J. Func. Prog. **18** (2008), no. 4, 423–436.
27. M. van Eekelen and M. de Mol, *Reflections on Type Theory, λ-Calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60^{th} Birthday*, ch. Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pp. 87–101, Radboud U. Nijmegen, 2007.
28. E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan, *Silver: an Extensible Attribute Grammar System*, Sci. Comp. Prog. **75** (2010), no. 1–2, 39–54.
29. P. Wadler, *The Expression Problem*, Java Genericity Mailing List, November 1998.