Component-Based Mechanisation of Programming Languages in Embedded Settings

Vom Promotionsausschuss der
Technischen Universität Hamburg-Harburg
zur Erlangung des akademischen Grades

Doktor(in) der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von
Seyed Hossein HAERI

aus
Teheran, IRAN

2014

Gutachter:
Prof. Dr. Sibylle Schupp
Prof. Dr. Klaus Ostermann

Tag der mündlichen Prüfung:
12. Dezember 2014

In the name of God, the compassionate, the merciful.

This is of the favour of my Lord.

Abstract

This thesis is about implementing programming languages for the specific purpose of experimentally studying their characteristics and conducts. More particularly, we aim at doing so in a component-based fashion to cater reuse at the level of language constructs. To that end, we ship the first set of reusable syntax, semantics, and analysis components for a small selection of lazy functional languages. We also explain the modest discipline that is needed for the use of our components. The components and this discipline together give rise to a new approach for programming language implementation that we discuss in detail in this thesis.

We define a new variation of the famous Expression Problem called the Expression Compatibility Problem. We formulate the eight concerns of the latter problem and present two solutions for it using our components. The first solution takes the form of additive component composition, whilst the second is rather feature-oriented.

We introduce the first formal model for component-based implementation of programming languages. We provide the syntax, static semantics, and dynamic semantics of the model. Our model closely takes after those for lightweight family polymorphism. We employ this model as a means for high-level description of our components and their use.

To the light of my heart
without whom it would have frozen...

Acknowledgements

The student would like to thank the invaluable helps and directions of Prof. Dr. Sibylle Schupp (his supervisor).

Contents

List of Figures

Chapter 1

Introduction

Implementation of Programming Languages (PLs) is similar to most software engineering tasks in its repetitive nature. As such, it often enjoys cycles; each containing individual steps: Typically, syntax and semantics are implemented; the implementation is tested against a repository of language analyses; and, based on benchmarking of the results, decisions are made on either shipping the implementation or a strategy for moving to the next cycle. Accordingly, PL implementation tasks can be grouped into those related to syntax, semantics, and analysis. To increase effectiveness of the process, certain features seem vital for each group of tasks.

Consecutive cycles are likely to share a sizeable fraction of their syntax. Syntax implementation involves: grounding the abstract syntax (i.e., encoding it, say, using algebraic datatypes); facilitating concrete syntax (i.e., keeping the programming notation in the closest possible proximity to the paper-and-pen one); assuring well-formedness and the like (i.e., implementing sanity checks that outlaw unwanted pieces of syntax); providing pretty-printing and other debugging cosmetics; and, possibly, micro-level profiling. Proper code reuse is needed to avoid repetition of all that upon each cycle.

Semantics implementation requires executability for the practitioner to experiment with it in action. Furthermore, semantics needs to be implemented in a notation which is close enough to the formalism used for its specification. Transcribing the formalism into the notation of the Language Definitional Framework (LDF) becomes error-prone otherwise. The LDF needs to be flexible about the formalisms used, or it only suits certain sorts of semantics. Most importantly, facilities for code reuse should be available to prevent unnecessary reimplementation of the semantics. One comes to a better acknowledgement of such facilities by noticing that successive cycles often only differ in few semantic rules. Updates might, however, infringe

the applicability of rules. Their reuse should be banned under such circumstances. Finally, debugging and profiling can be handy at the semantics level too.

As far as their function is concerned, analyses can be of two groups: They are either as simple as counting the number of predesignated programs that deliver the expected results, to which we refer as "result-match testing." Or, they can entail parsing or evaluation introspection, and examining details of **how** results are obtained. We refer to the latter group as the "introspective analyses." Result-match testing rarely needs reimplementation, and even if needed, the effort ought to be negligible. Having to reimplement introspective analyses upon every cycle can, however, form serious impediments against PL development – both in mature PLs (due to size) and early-stage ones (due to high frequency of updates). It follows that good support for (introspective) analysis code reuse is essential.

Similarly, one can divide analyses into two groups based on their chronological applicability. A group of analyses are exclusively applicable once the evaluation is terminated – to which we refer as "a posteriori." Others operate during the evaluation and are likely, hence, to need prior configuration. We refer to the latter as the "a priori" ones. Reusability is critically important to both the a priori and a posteriori analyses. Yet, similar to the case of semantics, reusing analysis implementations should be forbidden when inappropriate. Other analysis facilities that can uncover many otherwise unknowable facts are debugging and profiling.

Code reuse, however, does not only serve a single PL implementation. Research papers often build on extensions over some established core language. The number of papers which build on untyped λ-calculi is, for example, remarkably high. Reusability will minimise the need for reimplementing syntax, semantics, and analysis of the core language. One should merely extend the implementation for the new language features, if needed. As a result, different research groups can work independently by only extending the implementation for their own pertaining portion, and sharing the core. The need for **modular** implementation of PLs becomes obvious.

Whilst vital, as discussed above, we argue that modularity is not enough. In particular, *component-based* approaches can provide even more of the essential reusability by facilitating implementation sharing for individual PL constructs. It is not uncommon for different PLs to be extended using similar constructs (in the syntax or semantics). For example, *generics* were not initially included in Java and C# but were later added to both – albeit with slightly different flavours. Component-based development 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.

It has for long been an observation that the syntax of each implementation cycle gives rise to a *family* of mutually recursive classes. As such, the exercise of facilitating reusability for PL implementation shows itself as a special case of the *mutual extensibility* problem. Components can be of great help here in that the use of new components in a cycle does not invalidate the corresponding family of the older cycles. Good components are implemented independently of the family and only dictate their own functional expectations from the family they are to become a member of. In each family, the member interconnections give the family a *shape* identity. Consecutive PL implementation cycles share these shapes and differ based on the added or removed components.

One can implement a PL for a variety of purposes. We say a PL is *mechanised* when it is implemented for the specific purpose of experimentally studying its characteristics and conduct. The idea is that one interacts with the mechanisation to discover otherwise inapparent facts or flaws **in action**. PL mechanisation can, however, become very involved using traditional formal proof systems. Many LDFs, therefore, target **lightweight** mechanisation say at an early stage where proofs around the artefact are not of high priority. This is to let the PL designer experience their product after easy quick steps.

LDFs altogether foster PL mechanisation to a great deal. With the wide selection of LDFs that are available nowadays, most mechanisation projects can find their suitable LDF quickly. There currently is, however, no LDF that is designed for Component-Based Mechanisation (CBM) – especially, at the granularity of PL constructs. Hence, our above reuse aims entail resorting to embedding the mechanisation in a general purpose PL for none of the LDFs at hand is easy to manipulate to that end.

1.1 Contributions

The major contributions of this thesis are as follows:

We deliver syntax, semantics, and analysis components to address the three tasks of PL mechanisation, respectively. These highly reusable components are the first of their kind in that a syntax component is not bound to any particular semantics component or vice versa. Likewise, there is no one-to-one coupling between our syntax and analysis components or between our semantics and analysis components. We deliver 15 syntax components (Chapter 7), 11 semantics components (Chapter 8), and 30 analysis components (Chapter 9). Our components are tested against a group of 6 small

functional PLs (Chapter 3) and 11 analyses defined over them.

We define the *Expression Compatibility Problem* (ECP) – a new variation of the famous *Expression Problem* (EP) – and specify it by its eight concerns (Section 2.3). Most ECP concerns are familiar in the context of CBM, except "compatibility of extensions," which, crudely put, is about ensuring that PL extensions do not remove or replace constructs of the core PL. Given that this latter ECP concern is not well-studied so far, we devote special attention to it (Sections 6.1 and 6.3.3). We offer two solutions to ECP in this thesis that, although both are based on our mechanisation components, differ in their nature. The first (Sections 7.3 and 8.4) works by additive component composition, whilst the second (Section 9.2) uses feature-oriented techniques.

Inspired by the models of lightweight family polymorphism, we develop CBMCalc (Chapter 6): the first formal model for component-based mechanisation. We present the syntax (Section 6.2), static semantics (Section 6.3), and dynamic semantics (Section 6.4) of CBMCalc. Throughout this thesis, CBMCalc is used on a number of occasions to give a high-level description for different parts of our codebase and their behaviours.

Here is a more detailed summary in addition to our minor contributions:

Our syntax components can essentially be viewed as units of reuse for abstract syntax. But, we also embed concrete syntax, well-formedness, pretty-printing, profiling, and debugging cosmetics in terms of our syntax components. As a result, the concrete syntax and all that can be reused wherever the appropriate abstract syntax is present.

A PL semantics typically is comprised of semantic rules. Conventionally, one mechanises all the rules together to deliver a monolithic PL semantics mechanisation. To the contrary, we break semantics mechanisation into reusable components each corresponding to a single rule. Each semantics component of ours exclusively handles its own part of the semantics and leaves the rest to a continuation that it accepts as an argument. As such, our semantics components get to serve every appropriate syntax and semantics they are plugged into. This reusability is gained via parameterisation of our semantics components over the (abstract) syntax and semantics they can serve. The only similar gain of reusability is that of TinkerType, which still is different in significant ways. We encode the parameterisation in the type system such that the parameters are type entities that exhibit static and dynamic behaviour. In TinkerType, on the contrary, the component contents are plain strings.

Of the various available semantics formalisms, our component-based approach for mechanising semantics is only used for big-step (a.k.a. natural) semantics. Yet, given their rule-based nature, we expect our approach to be similarly applicable to other formalisms. All our semantics components are

executable whilst their embedding syntax takes a close resemblance to their paper-and-pen correspondents.

For analysis, from the viewpoint of their function, in this thesis, we rather focus on the introspective ones. Our techniques work for both a posteriori (Section 9.1) and a priori (Section 9.2) analyses. We employ Feature-Oriented Programming (FOP) for the former and monolithic pattern matching for the latter. As a side product, we show how the former techniques can also serve non-compositional PL transformation (Section 9.3). The reusability that we gain in both groups is through implementing our analyses in terms of our syntax and semantics components. As such, the implemented analyses are reusable amongst all the PLs mechanised using those syntax and semantics components, but are statically prohibited otherwise.

Finally, our components dictate a modest programming discipline. A minimal proportion of that is on the PL implementer (i.e., the component user). The rest remains on the component vendor. This thesis explains that discipline in detail. In addition, our components demand the availability of the following two features from their host PL (which, in embedded settings, acts also as the LDF): Type Constraints and Multiple Inheritance. Whilst our use of multiple inheritance caters easy extension of mechanisation, type constraints help the compiler outlaw reuse of mechanisation when conceptually inapplicable. We choose to embed our approach in Scala for its unique combination of built-in features that suit mechanisation. 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.

We would like to point out that our codebase is available online at: `http://www.sts.tuhh.de/~hossein/compatibility`.[1]

1.2 Structure

Here is how the rest of this thesis is organised: We start by motivating the thesis and its settings in Chapter 2. Given that our running example will be referred to throughout the rest of the thesis, Chapter 3 gives a brief introduction into that. A comprehensive review on the relevant literature follows in Chapter 4. Next, a quick overview of Component-Based Software Engineering (CBSE) and how that relates to our case comes in Chapter 5. We then provide a mathematical account in Chapter 6 of our expectations from a Component-Based Mechanisation (CBM) in an embedded setting. Armed with all that, in Chapters 7 and 8, we present our syntax and semantics

[1] password = "sts"

components, respectively. Chapter 9 builds on top of the latter two chapters to present how we mechanise PL analysis and transformation. The technical development of this thesis ends in Chapter 10 where Conclusion and Future Work are provided. Yet, Appendix A walks through the essentials of Scala that we use.

Chapter 2

Motivation

In this chapter, we start in Section 2.1 by expanding on why we choose to develop in an embedded setting. Section 2.2 then provides more grounds for taking a component-based approach for mechanisation. Next, Section 2.3 compiles a list of properties that we consider normative for components to serve Component-Based Mechanisation (CBM) in an embedded setting. Section 2.4, afterwards, takes a quick tour on the available CBM support. This chapter ends in Section 2.5 where the lack of support for the above list of normative properties is discussed. The conclusion to draw from the latter two sections is deemed to be the necessity of this thesis.

2.1 Why Embedded?

An LDF[1] typically ships a complete ecosystem to serve mechanisation. Skilful observation of the needs arisen over different mechanisation projects is the basis for inclusion of inhabitants in such an ecosystem. The language with which these inhabitants communicate is to be of the right level of abstraction and straight to the point for the ecosystem itself. To that end, many LDFs ship with their own bespoke Domain-Specific Languages (DSLs). The LDF provider extends or modifies their DSL upon need. But, the PL implementer rarely has much flexibility in doing so.

On the other hand, it has for long also come to notice that PL mechanisation[2] can involve general programming: i.e., programming using constructs

[1]The author of this thesis is unaware of any definition for a Language Definitional Framework (LDF) that is widely agreed upon in the relevant literature. However, his impression is that, in the PL folklore, LDF is by and large referred to tools for mechanical definition of PLs and rapid experimentation with them.

[2]Recall from Chapter 1 that we say a PL is mechanised when it is implemented for the

that are not specific to mechanisation tasks. To address that need, many LDFs – like Rascal [KvdSV11], JastAdd [EH07], and Polyglot [NCM03] to name a few – provide a back-door escape to a General-Purpose Language (GPL). The idea is that the PL implementer would *mostly* use the DSL and only *occasionally* resort to the GPL, when required. As such, the support for communication between the DSL and the GPL is often bare minimal. When the PL implementer faces unforeseen requirements for the communication, they are bound to accommodation endeavours of the LDF.

Although not properly documented, like general-purpose programming, mechanisation has over years developed its own design-patterns and paradigms. As will be discussed over the next sections, CBM is still like a completely new paradigm for mechanisation. Hence, for CBM under an existing LDF, we would have needed to act at a meta-level to the LDF itself.

Finally, amongst the benefits of an embedded DSL over a normal DSL is its relative ease of parsing: An embedded DSL employs the parser (and other tools in the toolchain) of its host PL to process **parsed trees** rather than raw strings.

Based on the above observations, we choose an embedded setting for our implementation because:

1. No LDF currently ships enough meta-level support for shoehorning a brand-new mechanisation paradigm onto their ecosystem.

2. Neither does any LDF currently facilitate general-purpose programming at the level we wanted, nor are their available GPL escapes accommodating enough for us. In particular, we expect: widely available libraries, powerful metaprogramming facilities, strong static typing, and support for multi-paradigm programming.

3. We prefer not to directly deal with parsing and rely on readily available facilities.

With our above concerns, we choose Scala for its outstanding richness for embedded DSL development. For the rest of this thesis, we refer to LDFs that are not embedded in GPL as the "unembedded" LDFs.

2.2 Why Component-Based?

With the observation of repeating constructs in consecutive cycles of mechanisation, the classical viewpoint of modularity is the reuse of a core upon

specific purpose of experimentally studying its characteristics and conduct.

extension. Modularity is a well-known essential for PL mechanisation that is also well supported by various LDFs. In fact, all the LDFs that we know (Section 4.1) do support modularity. PLs, however, also share individual language constructs even when they are not extensions to the same core. Trivial examples are loops and conditionals that are widely common amongst PLs of various groups.

Whilst the need for reusing PL constructs is not just recently realised [HK95], support for it is still minimal. The appropriate granularity for reuse was for years not clear. Reusable components with much coarser granularity than individual PL constructs were initially devised [vHD+01]. The need for a radical shift to CBM – as a new mechanisation **paradigm** – was first proposed by Mosses [Mos08]. In his very recent work [Mos13], Mosses also explains how attempts for simulating CBM using the available LDF support for modularity fails. Instead of repeating his comprehensive discussion on the necessity of CBM, the next paragraph summarises why we take a CBM approach:

Modularity is a valuable feature for multiple extensions to a single core. Whilst also supporting that, our understanding is that CBM is essential for the other direction, namely, a single extension to multiple cores. Note that this is not an uncommon scenario at all. For example, let-expressions are commonly added to many core λ-calculi: Call-by-Value, Call-by-Name, Call-by-Need, etc. To support such extensions, the corresponding code of the individual PL constructs (like those for let-expressions exemplified above) need to be shipped as stand-alone **component**s that are available for plugging together to form a PL mechanisation.

2.3 ECP Specification

The Expression Problem (EP) [Coo90, Rey75, Wad98] is amongst the most famous problems in the PL community. A wide range of EP solutions has thus far been proposed. Consider [BH12, OZ05a, OC12, Swi08, Tor04], to name a few. The essence of EP is the challenge of finding an implementation for an algebraic datatype (ADT) – defined by its cases and the functions on it – that:

E1. is *bidirectionally extensible*, i.e., both new cases and functions can be added.

E2. provides *weak static type safety*, i.e., applying a function f on a statically constructed ADT term t should fail to compile when f does not cover all the cases in t.

E3. upon extension, forces *no manipulation or duplication* to the existing code.

E4. accommodates *separate compilation*, i.e., compiling the extension imposes no requirement for repeating compilation or type checking of existing code. Such static checks should not be deferred to the link or run time either.

In an embedded setting, PL mechanisation is expected to address EP. The PL syntax is usually mechanised using ADTs. The static and dynamic semantics – as well as the analyses defined on the PL – are then functions on these ADTs. Addition of new pieces of syntax implies extending the semantics and analysis respectively. New analyses can also be added independently. The concerns E1 to E4 are highly desirable hereto.

With our emphasis on mechanisation based on components, we define the *Expression Compatibility Problem* (ECP) as the challenge of finding an ADT implementation using *reusable* and *composable* components for *families* of PLs [Oli09], which addresses the EP challenge and:

EC1. is *independently extensible* [Szy96, OZ05a] (a.k.a. *extensionally unifiable* [EGR12]), i.e., it should be possible to compose independently developed extensions.

EC2. is *backward compatible*, i.e., upon extension, the existing code retains its consistency and remains available for use – specially, to the extension. In other words, it guarantees both *non-destructive* extensibility [NQM06] and *object-level* extensibility [Tor04].

EC3. is *scalable* [NQM06], i.e., adding a new case or function takes proportional code.

EC4. ensures *completeness of component composition*, i.e., no valid combination of components should be rejected statically or dynamically.

EC5. *statically* ensures *soundness of component composition*, i.e., provides means to enforce compilation failure for invalid component combinations.

EC6. is *combination sensitive*, i.e., distinguishes between PL type identities by their component combinations. In particular, the type identity of a core PL should be distinguishable from its extension.

EC7. *accumulates nominal subtyping*, i.e., recognises the component combination c_1 as a structural subtype of c_2 when every case in c_1 is a nominal subtype of a case in c_2.

EC8. guarantees *syntactic compatibility upon extension*, i.e., statically rejects addition of new cases when the syntactic category of one existing case (or more) is neither retained intact nor refined.

We call the first six the *case concerns* of EC: those which are concerned with issues about ADT cases. Likewise, we call the first five plus the last two the *function concerns*: those concerned about functions defined on ADTs. That is, the first five are both case concerns and function concerns. One may wonder why EC8 in presence of EC5? Firstly, EC8 is particularly concerned about the sanity of extensions; whereas EC5 is also about core combinations ab initio. Secondly, EC5 is both a case and a function concern whilst EC8 is only a latter.

Here is why we find ECP concerns important to CBM: EC1 facilitates the independent development of PLs. As a side-product, it facilitates the mechanisation process in general by making it possible to merge the work of different groups independently extending a single core. EC2 is the embedded counterpart of what is famous as modularity in the mechanisation community. EC4, EC5, and EC8 aim to ensure correctness of the mechanisation. They do so by ensuring the correctness of component composition for PL mechanisation. EC6 and EC7 authorise reuse of code which is sensitive to certain component combinations. They allow reuse of core PL facilities for the extensions by making side-by-side testing possible for them. EC3 is obvious.

2.4 Available CBM Support

Limited CBM is available under PLanCompS [JMS10, BPM13, CMT14], bondi [JV13], TinkerType [LP03], Polyglot [NCM03], GLoo [Lum08], and the Sugar* family [ERKO11, ERRO12]. In PLanCompS, bondi, and TinkerType, each component is shipped as an integrated package of syntax, semantics, and (perhaps) analysis. Accordingly, no CBM support is available for independent mechanisation of (abstract/concrete) syntax, semantics, and analysis. In contrast, in our approach, there are syntax components (Chapter 7) and semantics components (Chapter 8). Mechanisation of syntax, semantics, and analysis can, hence, happen independently in our approach. (See Remark 8.1 for more.) The other three works of this group target mechanisation through syntactic desugaring into a core PL.

MontiCore [KRV10] and Neverlang 2 [CV13] are LDFs for DSL mechanisation. Although not designed for this particular purpose, the "compositional visitors" of MontiCore can be used for CBM. The focus in Neverlang is on

component-based compiler generation for DSLs with no emphasis on formal semantics.

As explained in Section 2.1, with the lack of direct support for CBM, we ended up embedding a simulation in Scala. Two important ingredients of our approach for CBM are type constraints and multiple inheritance. To explore the possibility of a similar simulation under available LDFs, the next paragraph discusses the availability of the two ingredients under each LDF. We do not claim the impossibility of simulating CBM in the absence of those two ingredients.

As an LDF, Kiama [SKV13] is itself embedded in Scala. Hence, Kiama does already have all the language support required by our approach. Maude [CDE+03], K [FŞAL+12], MMT [CB07], Redex [FFF09], Liga [KW94], Silver [VWBGK10], Rascal [KvdSV11], UUAG [DFS09], JastAdd [EH07], and Spoofax [KV10] 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. In the LDF π [KM09], the meta-level and the ordinary-level language are indistinguishable. Extending π with multiple inheritance is likely to be possible. Yet, π cannot currently accommodate type constraints for its lack of a static semantics.

2.5 Available ECP Support

The Expression Families Problem (EFP) [Oli09] is yet another variant of EP that poses a closer challenge to that posed by ECP. Section 2.5.1 explores solutions to EFP. Amongst the wide range of EC solutions, the ones that get closest to addressing ECP are discussed in Section 2.5.2. The conclusion to draw is the lack of support for ECP despite the existing proximity.

2.5.1 Expression Families Problem

Oliveira was the first to define EFP and discuss the necessity of solving it. He also offered Modular Visitor Components (MVCs) [Oli09] as an EFP solution. Oliveira and Cook [OC12] back this work up by the powerful and simple concept of *object algebras* [Gut78]. Object algebras outperform MVCs in that they do not require the clunky wiring of the VISITOR pattern [GHJV94]. Later, Oliveira et al. [OvdSLC13] address some awkwardness issues faced in their former paper upon composition of object algebras.

Our understanding is that, in EFP, the term 'component' is used as a correspondent for ADT cases. This guess is backed up at various occasions in his seminal paper, where Oliveira discusses his EFP solutions.[3] We admit that the notion of components can be hard to agree upon. Yet, we have to also express our failure in pinpointing the entity in the two latter works of Oliveira et al. that is to correspond with the above understanding of the term 'component'. The obvious first guess is their object algebras. However, to us, their object algebras are rather omnipotent packs of components (ADT cases). We are not sure how to make a meaningful comparison between components and packs of components.

On the other hand, although these works advertise that they address EFP as well, their main focus is on EP. As such, they do not, for example, demonstrate how they manage the Equality Test and Narrowing exercises (Section 9.2). Thus, we only concentrate on Oliveira's seminal work in what follows.

MVCs address all EC concerns except EC5 and EC8. (See Section 4.3 for the details of how.) Their support for EC7 is worth a special attention: In his machinery, Oliveira chooses an extension to be a *super*type of a core. Whilst the reason for that choice is not clear to us, we wonder whether the consequence is that they support EC7 in the same direction of ours (c_1 subtype of c_2) or the opposite one. Like Zenger and Odersky [ZO01] – and in line with the entire OOP literature – we choose an extension to be a *sub*type of a core PL. This is a special case of our EC7. More on EFP in Section 4.3.

2.5.2 Expression Problem

Of the rich literature on EP we only consider a couple that we deem close enough to this paper. Garrigue [Gar00] solves EP using global case definitions that, at their point of definition, become available to every ADT defined afterwards. Per se, a function that pattern matches on a group of these global cases can serve any ADT containing the selected group. With OCaml's built-in support for Polymorphic Variants, this work is considerably simpler than that of ours and addresses all the ECP concerns except EC5 and EC6. Yet, from ECP's standpoint, the most important missing factor is a notion of components in this work. Rompf and Odersky [RO10] employ a fruitful combination of the Scala features to present a very simple yet effective solution to EP using Lightweight Modular Staging (LMS). The support of LMS for E2 can be easily broken using an incomplete pattern matching.

[3] For instance, in [Oli09, Fig. 8], he names his solution for the Equality Test exercise `ExtendedComponents` rather than `ExtendedComponent`. Note the missing plural 's' at the end of the second name.

Yet, given that pattern matching is dynamic, whether LMS really relaxes E2 is debatable. Although LMS is not based on components, it scores all the ECs except EC5, EC6, and EC8. Details of how Polymorphic Variants and LMS fail to score the mentioned EC concerns are discussed in Section 4.2.

Discussion

Out of the available EP solutions, we use the material in [OZ05b] with similarities to LMS [RO10], MVCs [Oli09], and Polymorphic Variants [Gar00].[4] We believe many EP solutions can be augmented by properly taking syntactic categories into consideration to likewise obtain systematic prevention of syntactic incompatibility (Definition 6.1). We choose to build on top of LMS for its relative elegance and brevity. We choose to emulate Polymorphic Variants (in Scala) because that was the EP solution syntactic categories are given the best account in.[5]

It is worth noting that Attribute Grammars (AGs) can address EP too.[6] Instead of relying on the exact structure, modularity using AG is obtained by (either directly or through attribute forwarding [VWdMBK02]) relying on the availability of certain fields or attributes for the relevant type constructors. As a result, upon structural updates, the AG code remains intact so long as the relevant fields/attributes are retained. However, attributes will not notice it if such updates modify the syntactic categories of the type constructors whilst retaining the fields/attributes. This is because current AG systems [DFS09, EH07, KW94, Slo11, VWBGK10, RMRHV06] do not take syntactic categories as separate entities into consideration. Note also that the problem in LMS is not an "Inheritance is not Subtyping" one [CHC90]: The polymorphic function of a deriving trait does specialise that of the base.

[4] Unlike LMS, we do not target Polymorphic Embedding [HORM08] for DSLs. As such, we do not employ higher kinded types as they do. However, like LMS, we do not restrict the choice of components to predefined ones.

[5] Although the pivotal role of syntactic categories is not acknowledged in [Gar00].

[6] Although some AG systems may relax E4.

Chapter 3

Subject Lazy Languages

As our running example, in this thesis, we choose a number of PLs that are developed as core variations to lazy λ-calculus. This chapter is a quick tour of the syntax (Section 3.1) and semantics (Section 3.2) of those PLs; but, only to the extent that we will need throughout the thesis. For more details, one may consult the respective research papers. Here is a chronological account:

Abramsky and Ong ($\mathcal{L}_0$) [AO93] were the first to present a lazy λ-calculus. Launchbury ($\mathcal{L}_1$) [Lau93] adds sharing to their work to avoid reevaluation of variables. Sinot ($\mathcal{L}_2$) [Sin08], then, goes for complete laziness to also avoid reevaluation of unbounded subexpressions. On the other hand, selective strictness was first added to laziness by van Eekelen and de Mol ($\mathcal{S}_1$) [vd07]. Haeri ($\mathcal{S}_2$) [Hae10] improves the latter work by preventing increase of heap-expressiveness upon evaluation of let-expressions. Whilst the last two works focus on serial enforcement of strictness, Haeri and Schupp ($\mathcal{S}_3$) [HS14] focus on distributed ways for enforcing that.

Before dispensing the above PLs themselves, we would like to sketch their suitability to this thesis. It is instructive, at this point, to think for a moment of the above PLs as the different development cycles of a single imaginary PL for lazy evaluation. That way, $\mathcal{L}_0$ resembles a pre-history; $\mathcal{L}_1$ is reminiscent of a first α-release; $\mathcal{L}_2$ and $\mathcal{S}_1$ are like retracted branches; and, finally, $\mathcal{S}_2$ and $\mathcal{S}_3$ are reminders of exploration of possibilities for next releases. In fact, it was by virtue of the above thought that the author of this thesis came to recognise the need for ECP as a recurrent problem in PL mechanisation. At the end of this chapter, where the reader is deemed to be acquainted enough with the syntax and semantics of the above PLs, Section 3.3 offers a more detailed account of which ECP concerns were appreciated by which aspect of the above thought.

3.1 Syntax

Here, we briefly present the syntax of the implemented PLs. Notationally, our presentation is not exactly the same as the original ones. We unify the original notations and neglect the minor differences. In our presentation, e always ranges over expressions, v ranges over values, and x ranges over variables. Note also that we use the common notation in the Functional Programming community for function applications. In particular, an expression e applied to a variable x is $e\ x$.

	$\mathcal{L}_0$	$\mathcal{L}_1$	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_3$	SC	Description
$e ::= x$	✓	✓	✓	✓	✓	Var	variable
$\lambda x.e$	✓	✓	✓	✓	✓	Lam	λ-abstraction
$e\ x$	✓	✓	✓	✓	✓	App	function application
$\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e$		✓	✓	✓	✓	Let	let-expression
$e_1\ \mathsf{seq}\ e_2$			✓	✓		Seq	selective strictness
$e\#x$					✓	Srp	strict application
$\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ \lambda x.e$				✓	✓	Val	let-surrounded λ-abstractions

$\mathcal{L}_0$= Abramsky and Ong, $\mathcal{L}_1$= Launchbury, $\mathcal{S}_1$= van Eekelen and de Mol, $\mathcal{S}_2$= Haeri, $\mathcal{S}_3$= Haeri and Schupp, SC = Syntactic Category

Figure 3.1: Syntax of our Lazy Languages (1 # 2)

Figure 3.1 is the first part of the implemented syntax, i.e., that of $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$. With the great degree of commonality between the five, we demonstrate them altogether in a single compact form to avoid repetition. Ticks show the available syntactic categories of each PL syntax. In Figure 3.1, we also present the abbreviations used in this thesis for naming the syntactic categories.

		New Syntactic Category
b	$::= x \mid Z(\overrightarrow{x})$	metavariable (MVar)
e	$::= b \mid \lambda x.b \mid e\ b \mid \mathsf{let}\ \{b_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e$	generalised identifier (GIdn)
v	$::= \lambda x.b \mid x\ b_1 \ldots b_n$	variable-to-identifier application (VApp)

Figure 3.2: Syntax of our Lazy Languages (2 # 2)

Figure 3.2 shows complete $\mathcal{L}_2$ syntax, which is a bit different from the previous five. 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 functions is allowed to metavariables as well as variables. Figure 3.2 does not repeat $\mathcal{L}_2$'s syntactic category descriptions and abbreviations that are common with the previous five PLs.

v	$::=$	$\lambda x.e$	$\mathcal{L}_0, \mathcal{L}_1, \mathcal{S}_1$
v	$::=$	$\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ \lambda x.e \quad (n \geq 0)$	$\mathcal{S}_2, \mathcal{S}_3$

Figure 3.3: Values in our Lazy Languages

λ-abstractions are the only values at $\mathcal{L}_0$, $\mathcal{L}_1$, and $\mathcal{S}_1$. In retrospect, in $\mathcal{S}_2$ and $\mathcal{S}_3$, 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 $n = 0$. (The abbreviation we use to denote the latter λ-abstraction is Val.) This is summarised in Figure 3.3. 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*. (See Figure 3.2.) In the syntax of all these languages, subscripts do not impact the syntactic category, and are allowed to be arbitrary.

3.2 Semantics

The operational semantics of $\mathcal{L}_0$ consists of only two rules. $\mathcal{L}_1$ manipulates those and adds two more to capture sharing in lazy evaluation. $\mathcal{S}_1$ adds another rule to capture selective strictness. $\mathcal{S}_2$ manipulates three rules from the latter work to avoid increase of heap-expressiveness upon evaluation of let-expressions. $\mathcal{S}_3$ replaces the serial selective strictness rule of $\mathcal{S}_2$ by a distributed one. $\mathcal{L}_2$ splits the function application rule of $\mathcal{L}_1$ into two rules. Whilst keeping other rules of $\mathcal{L}_1$ intact, $\mathcal{L}_2$ also adds a new rule for variables as well as another for metavariables.

Figures 3.4 and 3.5 formalise the previous paragraph. Here, rule labels are of the form $(\mathbf{r})_\ell$ where r is the rule name and ℓ is the list of the chosen lazy languages containing r in their semantics. In the semantics of $\mathcal{L}_0$, rules are of the form $e \Downarrow v$ (read as e evaluates to v). For the other members, the form is $\Gamma : e \Downarrow \Delta : v$ where capital Greek letters denote *heaps*. The latter scheme 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. $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ inherit the same terminology. Γ, Δ, and Θ range over heaps. The contents of a heap are *binding*s denoted by $x \mapsto e$. 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. This is expanded in $\mathcal{L}_2$ to the case of metavariables. Wild-card ("_") is our place-holder for insignificant pieces of information. All the chosen lazy languages have a *distinct-name convention*, i.e., variable names are supposed to be distinct. Figures 3.4 and 3.5 classify rules according to the head-constructor of the input expressions. The former figure demonstrates the rules corresponding to values, function applications, and variables. The latter one continues to the rules for let-expression, enforcing strictness and metavariables.

The rules $(\mathbf{app})_{\mathcal{S}_2,\mathcal{S}_3}$ (Figures 3.4) and $(\#)_{\mathcal{S}_3}$ (Figure 3.5) use the notation $(v)^{-\lambda}[y/\_]$. This is basically a short form for $(\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e)[y/x]$ where the value v is $\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ \lambda x.e$. The intuition for the notation $\Theta_1 \bowtie \Theta_2$ in $(\#)_{\mathcal{S}_3}$ (Figure 3.5) is 'update Θ_1 with the appropriate parts of Θ_2.' In $(\mathbf{app})_{\mathcal{L}_1,\mathcal{S}_1}$, call Δ and $\lambda y.e'$ the *intermediate* heap and *intermediate* expression, respectively. We use a similar naming convention for $(\mathbf{app})_{\mathcal{S}_2,\mathcal{S}_3}$. The last two rules play important roles for the rest of this thesis. Therefore, the next paragraph explains them in more detail:

Applications proceed by first evaluating the function. The λ is then removed with the argument being substituted in the resulting body for the parameter bound by the corresponding λ. The resulting expression is evaluated in the intermediate heap, the results of which returned intact. Note the subtle difference between $(\mathbf{app})_{\mathcal{L}_1,\mathcal{S}_1}$ and $(\mathbf{app})_{\mathcal{S}_2,\mathcal{S}_3}$: The (potential) let-bindings that the values carry in $\mathcal{S}_2$ and $\mathcal{S}_3$ need to also be taken into consideration for calculating the intermediate expression in $(\mathbf{app})_{\mathcal{S}_2,\mathcal{S}_3}$. This extra care is not needed in $(\mathbf{app})_{\mathcal{L}_1,\mathcal{S}_1}$ because values do not carry let-bindings.

3.3 Discussion

Here are pointers on how the peculiarities of the PLs presented in this chapter echo relevant ECP concerns:

- See Section 8.4.3 for how, deciding to merge $\mathcal{S}_2$ and $\mathcal{S}_3$ into a next release calls for EC1.
- The uncertainty of basing the next release upon either $\mathcal{L}_1$ branch calls for EC2.

Values:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ (\mathbf{lam})_{\mathcal{L}_0} \qquad \frac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\ (\mathbf{lam})_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{L}_2}$$

$$\frac{}{\Gamma : v \Downarrow \Gamma : v}\ (\mathbf{val})_{\mathcal{S}_2,\mathcal{S}_3}$$

Function Application:

$$\frac{e \Downarrow \lambda y.e' \quad e'[x/y] \Downarrow v}{e\ x \Downarrow v}\ (\mathbf{app})_{\mathcal{L}_0}$$

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

$$\frac{\Gamma : e \Downarrow \Theta : v_e \quad \Theta : (v_e)^{-\lambda}[x/\_] \Downarrow \Delta : v}{\Gamma : e\,x \Downarrow \Delta : v}\ (\mathbf{app})_{\mathcal{S}_2,\mathcal{S}_3}$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.b' \quad (\Delta, y \mapsto b) : b' \Downarrow (\Theta, y \mapsto \_) : v}{\Gamma : e\ b \Downarrow \Theta : v}\ (\mathbf{app_1})_{\mathcal{L}_2}$$

$$\frac{\Gamma : e \Downarrow \Delta : x\ b_1 \cdots b_n}{\Gamma : e\ b \Downarrow \Delta : x\ b_1 \cdots b_n\ b}\ (\mathbf{app_2})_{\mathcal{L}_2}$$

Variable Lookup:

$$\frac{\Gamma : e \Downarrow \Delta : v}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto v) : v}\ (\mathbf{var})_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{S}_2,\mathcal{L}_2,\mathcal{S}_3} \qquad \frac{x \notin \Gamma}{\Gamma : x \Downarrow \Gamma : x}\ (\mathbf{var'})_{\mathcal{L}_2}$$

$\mathcal{L}_0$= Abramsky and Ong, $\mathcal{L}_1$= Launchbury, $\mathcal{S}_1$= van Eekelen and de Mol, $\mathcal{S}_2$= Haeri, $\mathcal{L}_2$= Sinot, $\mathcal{S}_3$= Haeri and Schupp

Figure 3.4: Semantics of our Lazy Languages (1 # 2)

let Expressions:

$$\frac{(\Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) : e \Downarrow \Delta : v}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : v}\ (\mathbf{let})_{\mathcal{L}_1,\mathcal{S}_1,\mathcal{L}_2}$$

$$\frac{(\Gamma, x_i{\mapsto}e_i)_{i=1}^n : e \Downarrow (\Delta, x_i{\mapsto}e_i')_{i=1}^n : v \qquad \text{when } x_i \text{ fresh, } 1 \leq i \leq n}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : \mathsf{let}\ \{x_i{=}e_i'\}_{i=1}^n\ \mathsf{in}\ v}\ (\mathbf{let})_{\mathcal{S}_2,\mathcal{S}_3}$$

Enforcing Strictness:

$$\frac{\Gamma : e_1 \Downarrow \Theta : v_1 \quad \Theta : e_2 \Downarrow \Delta : v_2}{\Gamma : e_1\ \mathsf{seq}\ e_2 \Downarrow \Delta : v_2}\ (\mathbf{seq})_{\mathcal{S}_1,\mathcal{S}_2}$$

$$\frac{\Gamma : e \Downarrow \Theta_1 : v_e \quad \Gamma : x \Downarrow \Theta_2 : \_ \quad \Theta_1 \bowtie \Theta_2 : (v_e)^{-\lambda}[x/\_] \Downarrow \Delta : v}{\Gamma : e\#x \Downarrow \Delta : v}\ (\#)_{\mathcal{S}_3}$$

Meta Variables:

$$\frac{\Gamma : e \Downarrow \Delta : v_e \qquad (\Delta, Z(\overrightarrow{x}) \mapsto v_e) : v_e[\overrightarrow{x}/\overrightarrow{y}] \Downarrow \Theta : v}{(\Gamma, Z(\overrightarrow{x}) \mapsto e) : Z(\overrightarrow{y}) \Downarrow \Theta : v}\ (\mathbf{mvar})_{\mathcal{L}_2}$$

$\mathcal{L}_0$= Abramsky and Ong, $\mathcal{L}_1$= Launchbury, $\mathcal{S}_1$= van Eekelen and de Mol, $\mathcal{S}_2$= Haeri, $\mathcal{L}_2$= Sinot, $\mathcal{S}_3$= Haeri and Schupp

Figure 3.5: Semantics of our Lazy Languages (2 # 2)

- The fact that, despite their difference in the choice of value syntactic category, $\mathcal{S}_1$ and $\mathcal{S}_2$ both include Seq, reveals the need for EC4.
- As also noted in Section 7.3.2, one feels the need for EC5 by observing that Lam and Val cannot coexist in a single PL.
- Most importantly, the need for EC8 becomes clear when retracting from $\mathcal{S}_1$ or $\mathcal{L}_2$. In such a circumstance, one would want the new branch taken after the retraction not to be mistakenly taken as an extension to the retracted branch. Section 8.4.2 outlines one such scenario. EC8 receives special attention in this thesis in many places, including Section 6.3.3, most notably. Our particular reason for that is the lack of appealing study in the mechanisation literature on new PL versions not

just adding new PL constructs but also modifying or deleting existing ones.

And, here is how a similar series of mechanisations for related PLs arouse the remaining ECP concerns:

- The necessity of EC6 is highlighted by the need for different PL dialects to coexist and cater for head-to-head comparison. See Sections 7.3.1 and 9.1 for more.
- As elaborated in Section 7.3.1, one comes to appreciate EC7 by feeling the need for reuse upon branching in PL development.
- Finally, EC3 is obvious.

Note also the collective discussion in Section 8.4.4 on the ECP concerns we do not dedicate individual demonstrations to which.

Remark 3.1. As our running example, we believe that the chosen PLs are, at the same time, both large enough for conveying the message and sufficiently small to keep the focus of this thesis. One might think that such PLs are already well-studied to the extent that their mechanisation is not motivated. See the work of Klein et al. [KCD+12] for various experiments where this suspicion goes wrong. That is achieved by reporting mechanisations that revealed mistakes in research papers on PL theory with rigorous formal semantics. As the peak of their presentation, they even report a mistake found in a Coq-verified system. In fact, our mechanisation too revealed a mistake in one of the PLs discussed in this thesis. See Section 10.2 for more.

Similarly, whilst, in this setting, update is admittedly not as frequent as in DSLs, it is still not infrequent either. For example, just since 1993 when the first semantics for lazy evaluation ($\mathcal{L}_0$) was introduced, 24 variations (of laziness) are presented. □

Chapter 4

Related Work

This chapter categorises related work and reviews each category compared to our own development. It starts in Section 4.1 where different LDFs are discussed along with how close they score to our needs. Then, it moves to EP and EFP solutions in Sections 4.2 and 4.3, respectively. A discussion on how family polymorphism, shape polymorphism, and genericity by shape are related to our work comes in Section 4.4. In Section 4.5, we take a look into the works that focus on validating mechanisation of concrete syntax. At last, other research on reuse of analysis mechanisation is discussed in Section 4.6.

4.1 Why not this LDF?

Martin Fowler coined the term "Language Workbench" as the toolchain for PL specification, (reuse of) implementation, and development of IDEs.[1] By the time of this writing, most language workbenches have a special emphasis on providing those services exclusively to DSLs. A comprehensive study of the workbenches that participated in the Language Workbench Challenge 2013 [2] comes in [EvdSV+13]. In this section, we do not repeat that discussion for such workbenches that can function as LDFs as well. In particular, we drop further discussion on Spoofax [KV10] and Sugar* [ERRO12, ERKO11]. We also find our Section 2.4 discussion on the following two workbenches thorough enough and drop further discussion on π [KM09] and Neverlang 2 [CV13]. In the remainder of this section, we provide detailed discussion on all the other LDFs that we are aware of. We do not claim completeness of

[1] See his 2005 blog post "Language workbenches: The killer-app for domain-specific languages?" online.

[2] http://www.languageworkbenches.net/index.php?title=LWC_2013

our below list of LDFs.

TinkerType [LP03] is the only LDF that is designed to serve CBM. The PL implementer can pick their favourite components (corresponding to familiar PL constructs) and customise them using the ready-baked TinkerType *feature*s. This FOP process is well-explained and can be used for production of a large selection of foreseen PLs. A significant drawback of TinkerType, however, is its undocumented processes for development and integration of new components and features. Hence, regarding its CBM practice, the provided documentation of TinkerType only facilitates judgement at the surface level. Yet, the accompanying paper makes it clear that TinkerType components ship with all their mechanisation tasks (syntax, semantics, analysis, etc.) packed together. The paper also reports on the two sizeable repositories that TinkerType is tried on: one for experimentation with type systems with various features and another for generation of proof snippets to help semi-automatic proof assistants. The TinkerType project is discontinued.

Maude [CDE+03] serves as a platform for language development using Equational and Rewriting Logic [Mes92, BM03]. Maude has successfully served a wide range of mechanisation projects.[3] Amongst the unembedded LDFs, the *paramterised modules* of Full Maude [CDEM07, Part II] are the only constructs that can serve CBM. This is because they can be parameterised over *theories* – a concept similar to ordinary interfaces of OOP. Furthermore, because they also support multiple inheritance, they are likely to even suit our approach for CBM. Maude is rather inclined towards specification and analysis development is not comfortable in it. It also offers constructs, data structures, and algorithms for general purpose programming. Finally, Maude has a very limited scripting runtime support. To our needs, the general purpose development tools of Maude make it not accessible enough. The Maude [CDE+03] tool MMT is the first LDF for MSOS. MMT supports CBM, but, only when the formalism used is MSOS.

K [RFŞ10] uses Rewriting Logic to rapidly produce executable language implementations. K has been used to give a formal specification to mainstream PLs such as C and Java [ER12, FCMR04]. However, K can serve semantics mechanisation only when the formalism used is "(small-step) operational semantics in the SOS style of reduction semantics (with evaluation contexts), or of the Chemical abstract machine (CHAM), or for describing ab-

[3] http://maude.cs.uiuc.edu/maude-tools.html

stract machines."[4] The module system of K is currently not ready for CBM. Yet, it has recently been used for a simulation of CBM under K [MV14]. Currently, K only ships with a limited set of simple built-in analyses. The K community has informally expressed their interest in developing APIs for interaction with GPLs.[5]

Rascal [KvdSV11] provides all that is needed for implementing a single complete pass of many mechanisation tasks. Real-world DSLs (like [vdBvdS11] for digital forensics) are reported to have been developed under Rascal.[6] As an unembedded LDF, the Rascal language is geared towards ordinary GPLs rather than mathematical formalism and is armed with general purpose programming constructs. Although deemed as being more flexible then, this makes it harder to mechanise semantics that is specified under the common mathematical formalisms. (Examples are denotational semantics, big-step and small-step semantics, MSOS.) A serious shortcoming of the current Rascal technology is that not much can be shared amongst successive development cycles. Below we discuss the reason.

Rascal uses what it traditionally refers to as "rewrite rules" for solving the EP faced upon PL extensions. Rewrite rules, however, are destructive extension mechanisms [NQM06]: The extended ADT **replaces** the core ADT. Hence, old code that is only in terms of the core ADT becomes unsafe upon extension. Furthermore, because this replacement is done **universally**, conflicting extensions cannot coexist in a single system. In other words, the behaviour of code spanning conflicting rewrite rules over different modules is not well-defined.

Kiama [SKV13] embeds Stratego [Vis04] in Scala and, therefore, enjoys the comfort of a GPL. Derivation rules embedded using Kiama usually tightly correspond to their mathematical specifications. Yet, sometimes this correspondence is lost in vital ways. For example, to avoid infinite execution loops, one has to leave axioms out when implementing an operational semantics.

PLT Redex [FFF09] is a DSL embedded in Racket [7] with a rich toolchain for interactive development of operational semantics. Redex makes lightweight

[4] email exchange on the K-User mailing list; available online at
http://lists.cs.uiuc.edu/pipermail/k-user/2012-February/000137.html

[5] personal conversation over SSLF12

[6] other Rascal success stories available online at
http://web.archive.org/web/20110901191210/http://www.rascal-mpl.org/Rascal/Stories

[7] http://racket-lang.org/

mechanisation of semantics particularly easy, but is tailored towards Reduction Semantics [FH92], which is not designed for modularity. Whether, under Redex, introspective analyses can remain intact over successive development cycles is currently unknown. But, even if so, being Racket-based, the Redex solution will not enjoy as much static type-safety as we need.

UUAG [**DFS09**] is a Haskell preprocessor based on AGs. An AG semantics for Haskell itself has been developed [Dij05] incrementally in UUAG. Attribute Grammars are more into compiler construction than a semanticist's usual mindset. Because Haskell does not support OOP, how to implement our CBM approach under UUAG is currently not known.

Liga [**KW94**] is amongst the oldest LDFs that support modularity. Liga is also AG-based and manages that using its *attribute module*s. Instead of depending on the exact abstract syntax, attribute modules depend on the availability of certain attributes in the grammar. These dependencies are taken care of at (the) linkage time when the modules are weaved together to get the PL executable. Liga is no longer maintained.

Silver [**VWBGK10**] is an LDF based on AGs that uses attribute forwarding to manage extensions. As explained in the last paragraph of Section 2, however, attribute forwarding fails to guarantee our desirable safety features upon extensions. As an AG system, Silver is used to extend its own core meta-level language with general-purpose features. Amongst other things, Silver has been successfully used for an extensible implementation of Java 1.4 [VWKBS07], static embedding of SQL into Java [VWBH06], and Lustre [GHVW07] (a PL for embedded safety-critical systems).

JastAdd [**EH07**] is yet another AG-based LDF that has been used for production of compilers for various Java versions. The Java compiler of JastAdd – called JastAddJ – is itself an extensible compiler for Java. The core of JastAddJ corresponds to Java 1.4, and extensions for Java 5, 6, and 7 are available [Ö12]. As a meta-level language, JastAddJ in essence supports object-orientation plus declarative AG syntax for extensions. This combination of abstractions is achieved using inter-type declarations [KHH+01], declarative rewrites [EH04], and circular reference attributes [MH07]. Neither the extension mechanism nor the (most up-to-date) JastAddJ language make it to the level of our needs. The JastAdd extension mechanism is, after all, based on AGs—which fail to address our extension safety concerns—

and, the JastAddJ language is essentially Java 7—which lacks multiple inheritance. (See Sections 2.1 and the closing paragraph of Chapter 2.) JastAdd has so far been used for mechanisation of more than 15 PLs in the JVM ecosystem.[8]

Ott [SZNO+10] is outstanding with its input notation. This is because the notation is in significant proximity to the PL specification formalisms. Ott can be used for simple tasks such as rapid transformation of plain ASCII specification to LaTeX. More complicated tasks that Ott is useful at include generating proof assistant code as well as OCaml implementation boilerplate code for the specified PL. The rapid sanity checks that Ott provides over these processes can be invaluable. Ott does not target semantics executability.

Lem [OBZNS11] can be considered as a successor of Ott that is much closer to PL implementation. Most remarkably, Lem is used to give a formal semantics to the C++11 [Cpp11] concurrency model [BOS+11]. In order to facilitate this new proximity, Lem replaces the specification of Ott (that is based on derivation rules) with simple discrete mathematical constructs. Lem does not directly target executability. Yet, it can be used for generation of proof assistant or OCaml code.

bondi [JV13] is an LDF based on the Pattern Calculus [Jay04, Jay09, JK09]. From a language standpoint, the mechanisation process has an FOP flavour in that PL features get added incrementally. bondi has a **limited** support for CBM; PL constructs are supposed to have packed all their relevant aspects together upon shipment: syntax, (static and dynamic) semantics, (presentational and debugging) cosmetics, etc. Foreseeing all those aspects is not realistic for it requires full knowledge about all the possible uses of the component. How bondi addresses that (perhaps via component renovation) is not well discussed.

The considerable flexibility of the Pattern Calculus enables bondi to serve mechanisation in a good variety of common programming paradigms, including OOP and FP. Yet, the syntax in which bondi manages that has little similarity to the common programming syntax. Hence, it is hard to imagine anyone other than the bondi authors to develop components using it. On the contrary, component combination is rather straightforward in bondi. When the mechanisation is simple component combination, thus, the PL implementer is likely to enjoy bondi as an LDF.

[8] http://www.jastadd.org/web/applications.php

Polyglot [NCM03] is designed for mechanisation of Java extensions as well as DSLs that *compile* to Java. Mechanisation in Polyglot is centred around one *scheduled compiler pass* (or more) to transform the Abstract Syntax Trees (ASTs) of the subject PL to Java ASTs. In fact, Polyglot can itself be considered as an extensible extension to Java. Per se, for general purpose programming tasks, Polyglot is bound to pure Java or its hand-crafted extensions that can be transformed back into it. We chose not to build on Polyglot for we do not find Java accommodating enough to our needs (Sections 2.1 and 2.4). Moreover, this AST back-transformation approach of Polyglot – although very effective – is not how formal semantics is usually specified. Interestingly enough, a back-transformation takes the form of a component in Polyglot. Hence, whilst radically different from the CBM approach of PLanCompS [JMS10] and that of ours, Polyglot too has a limited support for CBM. Polyglot has been used for mechanisation of various PLs; c.f. Nystrom et al. [NCM03, §5] for a detailed list.

GLoo [Lum08] has an open-ended meta-level language for mechanisation that offers its *mini-parsers* for CBM. Like our syntax components (Section 7), the mini-parsers of GLoo correspond to syntactic categories. The GLoo meta-level language is a dynamic FP language with constructs that are generally similar to the familiar OOP constructs. For further general purpose programming tasks, GLoo also facilitates interaction with Java. GLoo uses its *form extension operator* [Lum05] for PL composition. The problem with GLoo is the excess of syntactic noise in its meta-level language. This is to the degree that even following the worked-out examples of the paper is difficult. Besides, the preference of GLoo over embedded CBM (say in Java that it interacts with) is not clear.

MontiCore [KRV10] uses the familiar OOP inheritance for PL extension. The derived PL – i.e., the extension – overrides a specific method of the parent PL – i.e., the core – in which extra PL behaviour is specified. The body of such a method typically consists of a series of visitors, each (of which) predesignated for a specific analysis or transformation. In the terminology of MontiCore, this series of visitors is said to produce a "compositional visitor." A visitor in MontiCore takes the form of a (standalone) component. Hence, although not designed for this particular purpose, with the compositional visitors, MontiCore can be used for CBM. However, such a CBM support would enjoy much less static safety than that of our approach. This is because visitor composition of MontiCore is based on Java reflections, which is in no comparison with the static safety provided by type constraints (and even

multiple inheritance). (See Section 8 for how we employ type constraints and multiple inheritance.)

Misc Scala virtualisation (SV) [MRHO12] is a means for facilitating embedded lightweight PL engineering. It targets DSLs with no much emphasis on formal semantics. SV is not an LDF per se, but, under certain circumstances, it can provide extra help for embedded mechanisation under Scala. This is done by provision of means that can process ASTs of certain popular Scala constructs (such as loops and conditionals). By proper employment of those means, the PL implementer can replace the default behaviour of the chosen Scala constructs with the instructed ones. SV is shown to be particularly effective for facilitating heterogeneous parallelism [SRB+13, TBB+11]. We decided not to use SV because, for our chosen subject PLs (Section 3), customisation of those constructs does not add to the value of Scala.

4.2 Expression Problem

In this section we illustrate the failure of LMS and Polymorphic Variants in provision of ECP concerns outlined in Section 2.3.

LMS

With its simple and effective process, LMS is amongst the least-involved EP solutions. We now use the Scala code below to explain LMS. Each PL is mechanised in its own trait by defining its own ADT and functions on that. (See trait `PlainL1` below for $\mathcal{L}_1$.) To extend an existing mechanisation, one inherits from the corresponding base trait (like `PlainS2` in line 17). New cases and functions can be added in the derived trait. (Lines 20 and 21 add new cases. The `overriden eval` function in line 23 takes care of the new cases.) Prevention of manipulation is automatic through the Scala inheritance mechanism when the keyword `override` is not used. LMS avoids duplication by forwarding the existing cases to the base trait using base-type referral (like line 27).

LMS scores EC1 for free via Scala's built-in support for multiple inheritance: Supposing that traits `PlainL2` and `PlainS1` are LMS mechanisations for $\mathcal{L}_2$ and $\mathcal{S}_1$, respectively, one can derive a trait from both of `PlainL2` and `PlainS1` to mechanise a PL that unifies $\mathcal{L}_2$ and $\mathcal{S}_1$. LMS also scores EC2 because deriving from a Scala trait leaves the base trait intact. Likewise, LMS accommodates EC4 using trait inheritance: The cases available to a base trait are equally available to a derived one for case composition. The

support of EC3 by LMS becomes clear with the discussion in the next few paragraphs.

```
1  trait PlainL1 {
2    //Exp and its cases for Lam, Var, App, and Let
3    trait Exp
4    case class Var(name: Idn) extends Exp
5    case class Lam(x: Idn, e: Exp) extends Exp
6    case class App(e: Exp, x: Idn) extends Exp
7    case class Let(bs: Map[Idn, Exp], e: Exp) extends Exp
8
9    def eval(g: Heap, e: Exp) = e match {
10     case App(...) => {/* implementation of (app)_{L1, S1} */}
11     case Lam(...) => {...}
12     case Var(...) => {...}
13     case Let(...) => {...}
14   }
15 }
16
17 trait PlainS2 extends PlainL1 {
18   //the new Exp case for Seq and Val
19   //Oops! Lam is also a case Exp now.
20   case class Val(...) extends Exp
21   case class Sequ(e1: Exp, e2: Exp) extends Exp
22
23   override def eval(g: Heap, e: Exp) = e match {
24     //pattern matching for Val, Let, and Seq; but NOT App
25     case Val(...) => {...}
26     case Let(...) => {...}
27     case _ => super.eval(g, e)//Oops! App forgotten ⇒ silent data loss.
28   }
29 }
```

The traits `PlainL1` and `PlainS2` above demonstrate LMS mechanisation of $\mathcal{L}_1$ and $\mathcal{S}_2$, respectively. In the `eval` of `PlainS2` (lines 23–28), pattern matching for App is forgotten. The task is, hence, forwarded to `eval` of `PlainL1`. In other words, `PlainS2` implements $(\textbf{app})_{\mathcal{L}_1,\mathcal{S}_1}$ rather than $(\textbf{app})_{\mathcal{S}_2,\mathcal{S}_3}$. (See Figure 3.4 and the accompanying explanation for more details.) let-bindings are likely to be lost silently upon function applications. Consider the following example: In an empty heap, `PlainS2.eval` will reduce $(\mathsf{let}\, id = \lambda x.x\ \mathsf{in}\, \lambda z.\lambda x.\lambda y.x)\ t$ (for an arbitrary variable t) to $\lambda x.\lambda y.x$ instead of $\mathsf{let}\, id = \lambda x.x\ \mathsf{in}\, \lambda x.\lambda y.x$. This problem arises because over the update from $\mathcal{L}_1$ to $\mathcal{S}_2$, the syntactic category of values **changes**. LMS cannot detect this, namely, it does not offer EC8. Note that the failure of LMS in detecting violation of syntactic compatibility is not because it relaxes E2. The pattern matching does not run out of alternatives after all; it only fails to (statically) realise the absence of an **appropriate** one.

LMS fails to realise EC5 as well: By inheriting from `PlainL1`, the case `Lam` is silently a case for `Exp` of `PlainS2` too. That is, `PlainS2` is, in fact, not even a correct mechanisation of $\mathcal{S}_2$ for it is adding extra cases to the syntax. As such, referral to `Lam` objects in `PlainS2` goes unnoticed by Scala. Imagine a function `f` with no `super` call (like line 27 above), which is supposed to be exclusively available to the new cases of `PlainS2`. The pass of a mistakenly created expression using `Lam` will throw an unmatched pattern exception. If Scala could have been informed about combination of `Lam` and `Val` not being meaningful, it could have statically caught that. Yet, it fails to do so because we use the default way of Scala to define cases of `Exp`.[9] (See lines 4–7 above for `PlainL1` and 20–21 for `PlainS2`.) No mechanism in this simple way is responsible for ensuring meaningful combinations.

LMS also fails to provide EC6 **by default**. This is because, `Exp` in `PlainS2` is per se indistinguishable from the `Exp` inherited from `PlainL1`. That is, `Val` and `Seq` are also cases of the former `Exp`, yet, it is not differentiated from the latter one. In other words, in `PlainS2`, the two `Exp`s are indistinguishable although they combine different components. Fortunately, Scala provides easy support to manually circumvent that problem: `PlainS2.Exp` is exclusively the former type, and, `PlainL1.Exp` is exclusively the latter. One can get LMS to support EC7 in a similar way.

Polymorphic Variants

When an OCaml function proceeds by pattern matching on its argument `x`, instead of fixing the type of `x` rigidly, Polymorphic Variants enable the compiler to assign an **upper bound** of all the **considered ADT cases** to `x`. That is, whilst the opportunity for adding new cases is still present, existing cases are already acknowledged in the assigned type. In addition, new cases can be handled using new functions that forward the existing cases to existing functions by explicit nomination. This is the essence of how Polymorphic Variants solve EP. For example, in the code below, the type assigned to `eval_l`

```
1 type 'a lexp = [`Var of string | `Lam of string * 'a
2   | `App of 'a * 'a | `Let of ... * 'a]
3 let eval_l : 'a lexp -> 'a = function `Var x -> ...
4   | `App(e, x) -> ... | `Lam(x, e) -> ... | `Let(bs, e) -> ...
5
6 type 'a s2exp = [`Var of string | `Val of ... * string * 'a
7   | `App of 'a * 'a | `Let of ... * 'a | `Seq of 'a * 'a]
8 let eval_s2 : 'a s2exp -> 'a = function
```

[9] And, that is how the original LMS paper [RO10] presents the approach.

```
9   #lexp as e -> eval_l e | `Seq(e1, e2) -> ...
```

is `([> `Var of string | `App of 'a * 'a | `Lam of string * 'a | `Let of ... * 'a] as 'a) -> 'a`, which can be read as *any ADT that contains the $\mathcal{L}_1$ cases*. Next, `eval_s2` tries to seize this opportunity to extend its type of `'a` with the new `` `Seq of 'a * 'a `` case. Line 9 above performs that by providing a pattern match for the new case (`` `Seq ``) and forwarding the existing cases to `eval_l`. However, that forwarding attempt correctly fails to compile because `s2exp` substitutes `'Val of ...` for `` `Lam of ... ``. In other words, `'a` of `eval_s2` fails to model `'a` of `eval_l` (for the absence of `` `Lam ``). Whilst that addresses EC8, Polymorphic Variants still fail in the provision of EC5. This is because Polymorphic Variants are nothing but global case definitions with no pre-designated opportunity to express required soundness properties.

4.3 Expression Families Problem

As promised in Section 2.5.1, in this section we illustrate the failure of MVCs in addressing EC8 and EC5. We also discuss how the following works of Oliveira et al. – although allegedly solving EFP – are not quite related to ECP.

Modular Visitor Components

MVCs are essentially an elaboration on the famous VISITOR pattern. As such, they inherit the involved `accept`/`visit` wiring. This makes them considerably more complicated than our Intermediate Classes (ICs)[10]. The complication hits both the component developer **and** the component client. MVCs are parameterised over the visitor types, whereas ICs are parameterised over their minimal ADT shapes. Hence, to provide each ADT case, the MVC developer is obliged to provide two classes: the respective visitor as well as the case component. The use of F-Bounds [CCH+89] in ICs cancels that need. The obligation for the MVC client is to live with the extra `accept`/`visit` calls.[11] These calls can make the code logic unreadable by breaking it into unnatural pieces. See Fig. 8 in [Oli09] where the logic for each single case of the Equality Test exercise is broken into three separate classes. (Contrast that to our code in Section 9.2.) Readability aside, this solution fails to scale. Finally, in addition to what we expect from our host

[10] namely, our syntax components (Section 7.2.2)

[11] In fact, instead of `visit` Oliveira uses names that are more suggestive of the MVC functionality. But, the concept remains intact.

PL, MVCs expect the following: self-types, variance type annotations, and existential types[12].

Amongst other services, MVCs are a means for solving EP: New cases are added by mixing in new components; new functions are added by replacing used components with new ones that provide the new functionalities too:

```
trait EvalL1[V[-R, A]] extends lam[Exp[V], Val[V]] with
  app[Exp[V], Val[V]] with vAr[Exp[V], Val[V]] with let[Exp[V], Val[V]]
{/* four methods for evaluation of Lam, App, Var, and Let */}
trait EvalS2[V[-R, A]] extends EvalL1[V] with seq[Exp[V], String]
{/* a method for evaluation of Seq into a String rather than a Val */}
```

(See [Oli09] for more on the role of `Exp`, `V`, `R`, and `A`.) The trait `EvalL1` above evaluates $\mathcal{L}_1$ expressions into $\mathcal{L}_1$ values. `EvalS2`, then, extends `EvalL1` by handling `Seq`. The compiler will, however, not reject `EvalS2` despite the fact that it tries to evaluate `Seq` expressions into `String`s (instead of `Val[V]`) and leaving evaluation of the other cases to `EvalL1`. That is, MVCs do not give the compiler information about the soundness properties they expect upon composition.[13] This is basically because EFP does not demand that. Despite that, Oliveira [Oli09, §6.1] shows how to wrap a feature layer around MVCs and get an ad-hoc enforcement of some soundness properties. The bottom line, however, is that MVCs fail to systematically facilitate EC8 and EC5.

Our final remark about MVCs is that, rather than components in their CBSE sense, MVCs are components in a Component-Oriented Programming [McI69] sense. Testimony to that is the excess of inside knowledge about MVCs that is crucial for managing the Equality Test [Oli09, Fig. 8]. In CBSE, components are identified by their 'requires'/'provides' **interface** (Section 5.1). This is in contrast with CBSE components, where, for example, relying on the implementation details of **how** a component realises its interfaces is not acceptable (Section 5.3).

Other

Aiming at presenting an EP solution that is more accessible to everyday programmers, Oliveira and Cook [OC12] employ the concept of object algebras [Gut78]. The power and simplicity of object algebras gives their so-

[12] In fact, our solution for implementing disjointedness uses existential types as well. Yet, that is just how to get Scala for that purpose and is not inherent to our approach. In a host-PL where F-Bounds are not essential for implementing our approach, the type arguments that the method `disjointed` passes to disjointed ICs need not to be existential.

[13] `EvalL1` and `EvalS2` only demonstrates that for external MVCs. Using a similar scenario, one can get into the same problem with internal MVCs too. See [Oli09] for more on external MVCs versus internal ones.

lution an interestingly high level of abstraction, using which they are not faced with many of the heavy type annotations that we are entangled in. Oliveira and Cook claim that this solution also solves EFP. In the following few paragraphs, we discuss why we sincerely find their latter claim to be unsubstantiated.

Oliveira's specification of EFP in his seminal work starts as follows:

> "*The EFP can be defined as the problem of achieving reusability and composability across the* ***component****s involved in a family of related expression datatypes and corresponding operations over those datatypes.*" [Oli09, §1][14]

The above quotation, of course, leaves it open for a component to be of any granularity. His solution, however, is based on MVCs, which are components at the granularity of ADT cases. To leverage object algebras in the approach of Cook and Oliveira, on the other hand, one first needs to craft an *object algebra interface*. Here is one from Cook and Oliveira themselves [OC12, Fig. 2] for an ADT with cases for integral literals and addition:

```
1 interface IntAlg<A> {
2   A lit(int x);
3   A add(A e1, A e2);
4 }
```

In `IntAlg`, the aforementioned ADT cases are provided using methods `lit` and `add` above (lines 2 and 3, respectively). (See the next paragraphs for more on the usability problems of such a realisation of ADT cases.) As such, similar to class sharing (Section 4.4), case definition in this approach is impossible without (enclosing) object algebra interfaces. To our understanding, with such an inseparability of ADT and cases, Cook and Oliveira fail to basically ship components that, like MVCs, correspond to ADT cases. This is because `lit` and `add` above are not loosely coupled – on the contrary, they are tightly coupled to `IntAlg` – nor do they offer independence. As detailed in Section 5.1, famous CBSE texts consider both loose coupling and independence essential for components.

Cook and Oliveira argue that, because they use less advanced PL features, they present an EP solution that makes it to everyday programmers of the mainstream languages. With the humble experience of the author of this thesis in the programming industry, he is not aware of everyday programming problems that reduce to EP. The question aside, we seem to also not be convinced about the usability of their approach.

Using methods for ADT cases, in the approach of Cook and Oliveira, expressions too end up being only expressible using methods. The method

[14] Boldfacing the term 'component' here is due to the author of this thesis.

`make3Plus5` below is how to get a reusable AST for the expression $3 + 5$ [OC12, §3]:

```
<A> A make3Plus5(IntAlg<A> f) {
  return f.add(f.lit(3), f.lit(5));
}
```

Such an approach forces the ADT user to either only content themselves with statically built expressions, or create on-the-fly λ-abstractions at (the) runtime.

Oliveira, van der Storm, and Cook [OvdSLC13] use more advanced features of Scala to address some composability problems of object algebras in the above work. Yet, with the complexity that is present even in their contrived examples, we chose not base our ECP solution on the work of Oliveira, van der Storm, and Cook and avoid extra complication.

Rendel et al. [RBO14] employ a Church encoding [BB85] based on the above work of Oliveira et al. to carefully bridge between two powerful concepts: object algebras and AGs. They offer a comprehensive discussion on how to encode various classes of AGs into object algebras embedded in a modular fashion in Scala. To improve certain correctness features in the work of Oliveira et al., Rendel et al. leverage even more indirection. Whilst providing its novel services, this latter development of theirs, however, also adds to the complexity of constructing embedded sentences. Here is their version of $3 + 5$ [RBO14, §2.6]:

```
1  def threeplusfive2[Ctx, Out](alg1: CtxInhAlg[Ctx with Out, Ctx, Ctx],
2                               alg2: CtxExprAlg[Ctx with Out, Ctx, Out])
3      : Ctx => Ctx with Out =
4    ctx => {
5      val in3 = alg1.add1(ctx)
6      val out3 = alg2.lit(3)(in3)
7      val all3 = mix[Ctx, Out](in3, out3)
8      val in5 = alg1.add2(all3)(ctx)
9      val out5 = alg2.lit(5)(in5)
10     val all5 = mix[Ctx, Out](in5, out5)
11     mix[Ctx, Out](ctx, alg2.add(all3, all5)(ctx))
12   }
```

But, Rendel et al. need not to worry about the above complexity because they also craft a generator, which, given the AGs, produces the object algebra embedding automatically.

4.4 Shapes and Families

Family Polymorphism

Family Polymorphism aims to ensure that certain interrelationships between the constituents of a system – referred altogether as a *family* – are all extension invariant. The seminal work of Ernst [Ern01] on the matter has thus far influenced a handful of programming languages including gBeta [Ern99], CaeserJ [MO03, AGMO06], and Scala. It has also been particularly prolific to more theoretic research on PLs. Specifically, it inspired: Virtual Classes [Ern03, Ern06] and Tribe [CDNW07] with their recent dynamic compatibility check update [ME10], Dependent Classes [GMO07], and Component Hierarchies [RSK10].

Family Polymorphism is relevant to this thesis because the syntax of PLs can be considered as a family of the PL cases. It authorises the reuse of base PL services by the extension when the interrelationships are preserved by the extension. Family Polymorphism is a meritorious tool when different instances of an exact family structure are to be distinguished from one another. But, otherwise, sharing classes amongst distinct families (of even the same structure) is notoriously challenging. The common workaround is to pass "package objects" [ME10] around in **every** computation to witness the legality of the sharing. Even if possible in all mechanisation tasks, we find it more involved than practical for our particular needs. This is because, for example, every subexpression is demanded, then, to contain an extra package object that agrees with that of every other subexpressions of the same expression. Both the extra runtime overhead and the programming workload are simply too much for the PL mechanisation tasks.

Class Sharing

Jx [NCM04] starts a series of works on sharing nested classes amongst families. Jx itself only allows that amongst hierarchies of families. J& [NQM06] generalises that to intersection types [CP96] to cater extension composition (c.f. EC1). Of the ECP concerns, it also addresses EC3 (using late-binding) and EC2. In $J\&_s$ [QM09], sharing of classes amongst families is nomination-based but not restricted to those which are hierarchically related. Instances of a shared class C of a family Φ can be *view*ed as a C instance of any other family Φ' which shares C with Φ. Only families in which a class is overriden can share it, and the non-shared classes need to be explicitly *mask*ed. As such, $J\&_s$ has built-in support for (manual) bidirectional adaptation between a base family and its derived ones. (See Remark 9.3 for a discussion on how

we simulate that in Scala.) The support of $J\&_s$ is realised using Dependent Classes [GMO07] and Prefix Types [NQM06]. Qi and Myers report the use of $J\&_s$ for four PLs. Interestingly enough, out of those four, the λ-calculus one [QM09, §6.3] suggests inadequacy of $J\&_s$ features for the task. $J\&_h$ [QM10] moves to **homogeneous** class sharing, where a core family and all its extensions are *equivalent*. To that end, when a new family extends a core family, $J\&_h$ automatically updates all the equivalent families (including the core itself) with the extensions provided by the new family. Finally, JavaGI [WT11] mixes class sharing with a large selection of other PL features that are out of the scope of this thesis.

The similarity between nested family classes and our components makes $J\&_s$ particularly close to our work. Yet, there are at least three significant differences between this entire thread of research and that of ours: First, the sharing of components between a base family and its extensions is not limited in our approach to identical components; substitution for equivalent ones is equally acceptable. Second, in their research thread, the shared classes have tight dependencies to their enclosing families (hierarchically, heterogeneously, or homogeneously). That is, addition of a new class (for sharing) amounts to addition of a new family or updating an existing one. In either case, recompilation of an enclosing family is inevitable. Our components are, however, not dependent on any family. (In fact, they are even not enclosed by a family.) Addition of a new component implies compilation of nothing but the component itself. Third, with their independent nature, components can enforce constraints on the families they are chosen for inclusion in. Shared classes, on the other hand, have no control over that to the degree that they can be shared via arbitrary enclosing families (subject to certain workflow disciplines that we do not delve into here).

Lightweight Family Polymorphism

Having observed the heavy workload of family polymorphism in certain circumstances, Saito, Igarashi, and Viroli [SIV08] craft lightweight family polymorphism. Similar to family polymorphism, they target extensibility of mutually recursive classes. Yet, inspired by Jolly et al. [JDAO04], they adopt a "class-as-families" principle. As a result, they gain considerable simplification in the type system without much loss of expressiveness. They develop a formal model .FJ for lightweight family polymorphism, which builds on top of Featherweight Java (FJ) [IPW01]. They prove a correctness theorem of the .FJ type system. Furthermore, they establish a translation from .FJ to FJ that they show to preserve typing and semantics.

In .FJ, families and their members are represented using top-level and

nested classes, respectively. Our work is similar, hence, in families being top-level traits. However, in our codebase, members are represented using the Scala case classes. (See Figure 7.2.) .FJ takes an "inheritance-without-subtyping" approach for family members. That is, whilst a same-named nested class of an extending family does (automatically) inherit from that of the base family, there is no subtyping relationship between the two nested member classes. We take a similar but different approach. For example, with $\mathcal{S}_1$ being a compatible extension to $\mathcal{L}_1$, `S1Exp#App` is not a subtype of `L1Exp#App`. Yet, they share behaviour and interface because they both inherit from `IApp`. This is due to the fact that they are of the same syntactic category, i.e., App.

Kamina and Tamai [KT08a] notice the monolithic nature of (lightweight) family polymorphism in that family members are inseparable from their enclosing families. They craft FGJ# – another variant of FJ with the addition of what they call *type parameter members*. Using this addition, they can take interfaces of family members out of family bodies so that members are no longer nested classes of families. Consequently, families and their members gain independence to the level of separate compilation from each other. However, addition of a new member is still not possible without first providing a new family interface that adds a new type parameter to correspond to the new member. It is only thereafter that one is authorised to refer to the new member using the type parameter member notation.

Here is their type parameter member notation: Consider a family interface `U` that has a type parameter `M`. For a type parameter `T` with upper bound `U`, when the family `F` gets substituted for `T`, the type expression `T#M` refers to the member class `M` of `F`. (Recall that by nominating `U` as an upper bound of `T`, substitution of the family `F` for `T` is only valid when `T` implements `U`.) Kamina and Tamai [KT08a] do not discuss whether reuse of code that works for `U` is possible by an extension that takes new type parameters.[15]

Lightweight dependent classes [KT08b] is the next work of Kamina and Tamai, with remarkable proximity to that of ours. Aiming at reducing the boilerplate code of the above two works of the section, they take mechanisation one further step toward CBM. They provide yet another FJ variant called X.FGJ, which consists of member interfaces and families. In X.FGJ, a family *materialises* its members by providing nested classes with empty bodies that derive from their respective member interfaces. The great reward here is that, with such a wiring, X.FGJ manages to cancel the need for

[15] In the context of EP, this question can be paraphrased as follows: Is it valid for an FGJ# code that works for a base ADT to be reused for an extension which adds new cases?

F-Bounds. This also, accordingly, cancels the need in their previous work for a complicated `this` type inference algorithm [KT08a, Fig. 9]. Besides, by abstracting members to the level of member interfaces, defining extensions which refine existing ADT cases is rather straightforward.

Despite all its improvements, in X.FGJ too, defining new members is only possible in tandem with defining new families which materialise the respective new member interfaces. Interestingly enough, this is due to the fact that in X.FGJ, members are **nested** classes of their enclosing families. On the contrary, our ICs are completely detached from the families they are to be members of.[16] Our ICs get materialised into family members using (Scala case) classes that are defined outside the family body (Figure 7.2).

Shape Polymorphism and Genericity by Shape

Shape Polymorphism [JC94] is about reuse of operations regardless of the shape (data structure) in which data is stored. This is similar to our component-based approach in that we too enjoy reuse of operations for many different shapes. Yet, shape polymorphism fully disassociates data and shape to target reuse amongst **all** shapes. On the contrary, we exclusively legislate reuse amongst a core PL and all its compatible extensions. As explained in Section 8.2.2, this can be regarded as reuse amongst families that share the shape of the core.

Genericity by Shape [JLMaRY09] is about reuse of code amongst all the ADTs that have the exact same shape; where, "shape" here refers to the AST signature defined using sum and products. In other words, abstracting over the case names, Genericity by Shape is concerned about reuse in presence of the exact same ADT structure. We, on the other hand, are not particularly concerned about the ADT structure so long as the cases are replaced by equivalent ones or new cases are added (to obtain extensions).

4.5 Concrete Syntax

Schwerdfeger and Van Wyk [SVW09] offer a mechanism for verifying when the concrete syntax of a PL and its extensions conflict. Their solution can be used when the grammars are deterministic. Grimm [Gri06] and Allen et al. [ACN+09] focus on mechanisms for concrete syntax extensions when the syntax is defined using PEGs [For04]. Brabrand and Schwartzbach [BS02]

[16] Although they do enforce constraints on such families. But, that does not attach our ICs in any way to the families.

present a macro language that suits domain specific extensions and can guarantee type safety and termination. Lorenzen and Erdweg [LE13] focus on type soundness of syntactic sugars. Erdweg et al. [EVMV14] coin *typesmart constructors* to dynamically enforce well-formedness of generated concrete syntax. Our notion of compatibility examines the abstract syntax of a language with no constraints on determinism or the syntax formalism.

4.6 Analysis Reuse

Various groups report having tested their language implementations. Examples from result-match testing are: [Fox03, HSY06, SSZN+09, FM10] for assembly and machine languages, [Bak09] for VHDL in Maude, [Dij05] for Haskell, [Ell12] for C, [RL11] for OCL.

Introspective analysis is also reported. For instance, at each development cycle:

- Shyamshankar, Palmer and Ahmad [SPA12] measure the success rate in pattern-recognition for possible parallelism in codes not written for this purpose. They also check whether the right scheduling was met.
- Chafi et al. [CSB+11] measure naturalness of their developed DSL to the target domain, and then, its performance. They also check whether the DSL is at an appropriate level of abstraction.[17]
- Klein et al. [KCD+12] test validity of propositions made in 9 ICFP papers *à* la QuickCheck [CH00].

Finally, although K was originally designed for modularity in syntax and semantics, it has also been used with a certain degree of analysis reuse [ALR11, AAL12].

[17] That is, whether, for the domain experts, their DSL is both easy to express and simple to understand.

Chapter 5

Component-Based Software Engineering

CBSE is a method for software engineering that, over the recent years, has attracted increasing interest in the software industry. Its characteristics, strengths, and weaknesses are considered solid topics for both academic and industrial research. Discussion on all that is, however, well beyond the focus of this thesis. Hence, this chapter only presents a short review on the CBSE topics that we find relevant to CBM and to the rest of this work. This chapter is truly inspired by the respective chapters of the two famous handbooks of software engineering: Sommerville [Som11] and Pressman [Pre09]. For more details on CBSE, the interested reader is encouraged to consult chapters 17 and 10 of the above books, respectively.

This chapter starts in Section 5.1 by a brief introduction to CBSE and how that matters to PL mechanisation. Next, in Section 5.2, we consider the reasons why we think modules do not suffice for CBM. In Section 5.3, finally, we present our vision about the CBM roles and processes.

5.1 Introduction

As it turns out, many diverse definitions have so far been proposed for CBSE. The definitions given by Sommerville and Pressman are:

> "*CBSE is the process of defining, implementing, and integrating or composing loosely coupled, independent components into systems.*" [Som11, opening discussion of §17]

> "*[CBSE] is a process that emphasizes the design and construc-*

Figure 5.1: UML Notation for Components

> *tion of computer-based systems using reusable software 'components.'"* [Pre09, §10.6]

Likewise, many different definitions have so far been developed for the term 'component' itself. For example, the UML specification developed by the Object-Management Group defines a component to be " ... a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces." [Gro03] On the other hand, whilst the focus of the above definitions of Sommerville and Pressman is not directly on components, they do convey viewpoints of the authors in that components are loosely coupled, independent, and reusable. Sommerville proceeds by compiling a list of component *characteristics*. He discusses why he believes a component needs to be "standardized, independent, composable, deployable, and documented."

Specification of a component is typically given by its 'requires' and 'provides' interfaces. See the respective UML notation in Figure 5.1. Here are the definitions that Sommerville gives for the 'provides' and 'requires' interfaces: "The 'provides' interface defines the services provided by the component. ... The 'requires' interface specifies what services must be provided by other components in the system if a component is to operate correctly." An interesting observation about the 'requires' interface is that it does not risk loss of component independence. This is because that interface does not define **how** the requested services should be provided; it is rather on **what** services should be provided and the implementation is free on the 'how' part.

Remark 5.1. Given that no host PL or LDF has a first-class support for components, our codebase uses the existing Scala inhabitants to rather *simulate* components. In particular, our syntax and semantics components are Scala classes and objects (i.e., singleton classes). As such, our codebase employs type parameters for specification of 'requires' interfaces. The abstract class `ICBase` specifies the 'provides' interface of syntax components (c.f. Section 8.3).

In CBMCalc, however, components are first-class residents. CBMCalc uses *family parameterisation* for specification of 'requires' interface

and normal OOP classes for specification of the 'provides' interface (c.f. Section 6.2). □

Remark 5.2. A given component can be described in many ways. Tracz [Tra95] suggests the 3C model: concept, content, and context, amongst which context receives a special treatment in our codebase. This is because context places a component "within its domain of applicability." [Pre09, §10.6.4] Our semantics components use type parameters to specify their context, i.e., the PL semantics they can be a part of. (Check the parameter `OS` in Section 8.1.) □

Remark 5.3. The Open-Closed Principle of Martin states that: "A module [component] should be open for extension but closed for modification." [Mar00] On the other hand, Sommerville stipulates that the independence of a component, amongst other things, means that its implementation can be replaced by another, without changing other parts of the system. With these two in mind, CBMCalc has a special emphasis on validity of passing *equivalent* components in return of one another. That is, for an instantiation in CBMCalc, all the components with the same 'requires' and 'provides' interface are considered equally useful. CBMCalc components, besides, can extend one another. Yet, upon extension, they cannot override a parent component's interface. □

5.2 Why not modules?

It is natural at this point to wonder why, in presence of modules, one would need the new concept of components. Modules have traditionally served software development as a means for breaking software into manageable units. Modules have proved their success in many PLs. And, in fact, modules too can be used as a means for *simulation* of CBSE. But, certain inevitable shortcomings will also be faced.

Pressman presents comprehensive discussion on how to approach CBSE using modules and the shortcomings of that [Pre09, §10.2.1 and §10.5]. We do not repeat his discussion in this section. Instead, we focus on the points which relate to CBM in particular. We begin our discussion by a few quotations. Our humble understanding is that they well describe the PL mechanisation situation. In his 'Quick Look' box of the respective chapter, Pressman mentions the following to emphasise the necessity of CBSE: "You have to be able to determine whether the software will work **before** you build it." A few lines later, he adds: "[CBSE] provides a means for assessing whether data structures, interfaces, and algorithms **will** work." Sommerville hints at

similar scenarios by reminding it that the CBSE users "may be willing to change their minds if this means cheaper or quicker system delivery." Later on, he continues that "... after the system architecture has been designed, you should spend more time on component validation. You need to be confident that the identified components are really suited to your application; if not, then you have to repeat the search and selection processes."

To us, the above words are specifically pinpointing the following facts about the CBSE practice that are specially common in CBM too: The rhythm of change and its likelihood are both relatively high in CBSE. Consequently, the system wiring should be well flexible against replacement of constituents. In particular, it is not acceptable for component replacement to hinder system builds. Hence, unless one extends the traditional module concept to say the parameterised modules of Maude [CDEM07, §3.5.2], modules will not do.

To clarify our line of reasoning, we exemplify the pioneer work of Mosses and Vesely [MV14]. This recent work reports the success of embedding funcons (i.e., PLanCompS components) [CMT14] in K [RFŞ10]. Their tactful embedding is accomplished by subtle uses of the K module system. Firstly, Mosses and Vesely reduce the number of funcons per module to almost 1. Secondly, by carefully crafting the right module import configuration, they leverage the static open-endedness of K sorts.

This latter craft of theirs, however, can also be fragile against updates. Each embedded funcon is dependent on *the exact* module import configuration of its file. At the funcon development time, however, modification of the funcon implementation is likely to imply updates for module import configuration as well. The unavoidable consequence is a rebuild of the funcon's embedding as well as those of all its dependent ones.[1] As outlined above, the imposed rebuild is in clear contrast with the nature of CBSE. Note that this fragility is not particular to the K module system. Erdweg and Rieger [ER13] present a detailed list of 16 features common in many module systems. No module system in their study seems not to be as fragile as K for the above task.

5.3 Roles and Processes

This section explains our standpoint about the CBM roles and processes. Figure 5.2 gives an overview of the upcoming discussion of this section. As

[1] In fact, such an update will enforce rebuild of the entire system. This is because K does currently not offer separate module compilation. But, we find that off-topic here for it is not inherent to module systems.

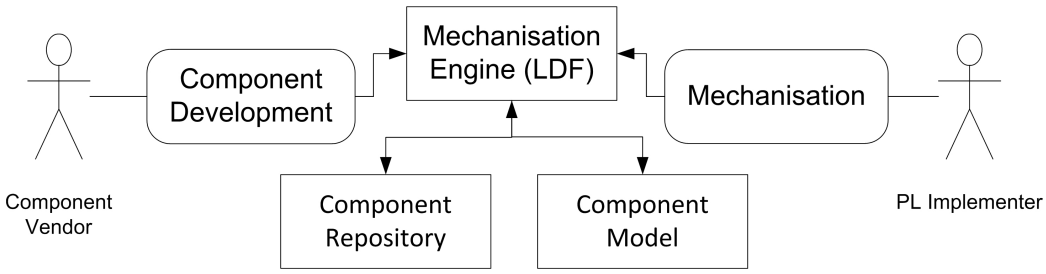


Figure 5.2: The CBM Roles and Processes

depicted there, the two main roles of CBM are played by the PL implementer and the component vendor. In between the two sits the LDF as a framework for development.

The task of a PL implementer in Figure 5.2 is PL mechanisation using the components provided to it by the LDF. A component vendor, on the other hand, takes the task of supplying components for PL mechanisation that are to be used via the LDF. Roughly speaking, those two tasks correspond to *CBSE with reuse* and *CBSE for reuse*, respectively. Sommerville defines the two as follows [Som11, §17.2]:

- CBSE for reuse is the process of developing reusable components and making them available for reuse through a component management system.
- CBSE with reuse is the process of developing new applications using existing components and services.

There are many different aspects of an LDF that can be helpful to CBM. We now employ a series of quotations about CBSE to explore the three LDF aspects that we find essential. One way of thinking about an LDF is as what Sommerville calls *middleware* (in CBSE): "[Middleware] provides software support for component integration. To make independent, distributed components work together, you need middleware support that handles component communications." More into the *infrastructure* nature of an LDF, the following definition of Sommerville is especially helpful: "Component infrastructures offer a range of standard services that can be used in application systems." Another way to think about an LDF is what Pressman calls *reuse environment* [Pre09, §10.6]. He states that a "reuse environment exhibits the following characteristics:

- A component database capable of storing software components and the classification information necessary to retrieve them.

- A library management system that provides access to the database.
- A software component retrieval system (e.g., an object request broker) that enables a client application to retrieve components and services from the library server.
- CBSE tools that support the integration of reused components into a new design or implementation."

In Figure 5.2, we refer to the first three bulletpoints above collectively as the "component repository." As a final remark on the LDF nature, we would like to remind that, as also discussed in Section 4.1, the services currently provided by LDFs are indeed diverse. That is mainly because there is no consensus on the essential and peripheral LDF services.

Another entity in Figure 5.2 that fosters collaboration between the component vendor and the PL implementer is the component model. Sommerville's following words describe some contents of a component model as well as its relation with the relevant stakeholders [Som11, §17.1]: "A component model is a definition of **standards** for component implementation, documentation, and deployment. These standards are for component developers to ensure that components can **interoperate**. They are also for providers of component execution infrastructures who provide middleware to **support** component operation." Later on, he adds: "The component model may therefore specify **how** the binary components can be customized for a particular deployment environment." Pressman finds the provision of standards by a component model so important that he writes [Pre09, §10.1]: "One of the key elements that lead to the success or failure of CBSE is the availability of component-based standards ..."

Remark 5.4. In our codebase, we use `LazyExp` as a model for our syntax components and `OpSem` for the semantics ones. (See Sections 7.2 and 8.1, respectively.) Both `LazyExp` and `OpSem` are our means for enforcing standards in our programming discipline. The former offers a medium for our syntax components to communicate their 'requires' interface. The latter is similar but for semantics components. Furthermore, the latter works by specification of the input and output types of the respective component integration mechanism.

As such, when a component of ours is to be replaced in a PL mechanisation, one needs to check conformance against `LazyExp` and `OpSem`. This is well in line with the following words of Sommerville: "Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment."

To that end, our codebase directs the compiler using type parameters. In CBMCalc, conformance rules come in Figures 6.8 and 6.10. The essence of those CBMCalc rules is *specification matching* [BGP95] on 'requires' interfaces. □

We would next like to expand on the mechanisation process performed by the PL implementer. The following quotation from Sommerville sets the stage [Som11, §17.2.2]: "The CBSE with reuse process has to include activities that find and **integrate** reusable components." In order to make this process easier and more effective, Pressman recommends the following sequence of actions [Pre09, §10.6.2]: "component qualification, component adaptation, and component composition."

Remark 5.5. Component qualification is a non-technical activity, hence, we disregard it in this thesis. Recall, however, that, with Scala's lack of first-class support for components, our codebase uses ordinary classes and objects to embed syntax and semantics components. Surprisingly enough, as far as component adaptation and composition is concerned, this implies more flexibility for the user of our codebase than that of CBMCalc. That is, the codebase user is, in principle, free to mix or adapt our syntax and semantics components in all valid ways for Scala classes and objects. (See the note below on our programming discipline.) CBMCalc, on the other hand, only allows component adaptation via its (component) extension mechanism. Moreover, its client is the only mechanism CBMCalc offers for component composition. □

Remark 5.6. Our programming discipline enforces semantics mechanisation exclusively via a task-distribution method called `proofsearch`. (See Section 8.2.) From the CBSE point of view, `proofsearch` corresponds to what Sommerville calls *glue code* [Som11, §17.3]: "Component composition is the process of integrating components with each other, and with specially written 'glue code' to create a system or another component." Of the different component composition methods he enumerates in the same section of his work, `proofsearch` chooses to follow *additive composition*. The interface of such a composition is a combination of the corresponding interfaces; the composed components are only accessed through the external interface of composition; and, components do not directly call each other. (See Figures 8.2 and 8.4 for more.) □

Discussion

On a side note, we would like to briefly reconsider an established piece of terminology in the community: LDF. [2] The emphasis in the term LDF is on mechanical PL **definition** – which is well reminiscent of the automatic checks on the PL description. Yet, it lacks a number of factors that we believe in the importance of. Firstly, executability is absent in the term LDF. Secondly, the necessity of reuse is not as widely accepted for definition as software. In other words, the term LDF is not particularly encouraging to reuse. Thirdly, it is not very suggestive of the business process.

The author of this thesis is under the humble impression that a term like "mechanisation engine" is much more suitable. This latter term is a clear adoption from the gaming industry where game engines are of a principal role. We believe a similar culture well suits PL mechanisation – especially when done in a component-based fashion: A mechanisation engine provides the foundational workplace in which components for the mechanisation of syntax, semantics, and analysis can be developed. It, furthermore, ships with a library of components and their specifications. In the same time, it grants component vendors with facilities for the development of new components or component variants. It remains for a PL implementer, then, to merely mix-and-match their suitable components. By definition, components are designed for maximal reuse. This process model additionally encourages that via promoting a plug-in nature for PL mechanisation components.

Hence, our understanding is that, as opposed to the term LDF, the term mechanisation engine is both more suggestive of the business process and informative of the available roles in (component-based) mechanisation of PLs. Nevertheless, in order to remain consistent with the relevant literature, we leave our own humble suggestion and stick to the term LDF throughout this thesis.

[2] The author was first exposed to this terminology in the works of Grigore Rosu. Yet, in a personal email, Dr. Rosu explained that he is not the origin of the term. He also said that even his PhD advisor (Prof. Joseph Goguen) used the term LDF ever since Dr. Rosu can remember.

Chapter 6

Formalisation

This thesis has so far used the term "component" informally in many occasions. We motivated component-based mechanisation in Sections 2.2 and 2.4. Then, in Sections 4.2 to 4.4, we considered various groups of related work for their facilitation of programming in a component-based fashion. In Chapter 5, we discussed component-based software engineering and explained our vision for how to relate that to component-based mechanisation. It is time now to formalise on the above grounds.

This chapter starts in Section 6.1 where a short and simple formalisation of ADTs and verification of their compatibility upon extension is presented. It turns out that this simple formalisation is enough for justifying the behaviour of our codebase (Chapters 7 to 9). Yet, given that (like other PLs currently available) Scala has no direct support for components and families, Scala tricks to emulate the CBM process for our running examples are ubiquitous in our codebase. This may hinder the unfamiliar reader by partly obscuring the essentials of the process behind the Scala wiring. To prevent that potential hinderance, we generalise the material in Section 6.1 (for simple ADTs and their cases) to a formal model of CBM that we call CBMCalc. Sections 6.2 to 6.4 explore different aspects of that. CBMCalc is classically inspired by lightweight family polymorphism (Section 4.4).

The CBMCalc world has three major role-players: components, families, and clients. A CBMCalc component takes after its well-known CBSE identity by specifying its 'requires' and 'provides' interfaces. It specifies its 'requires' interface using *family parameterisation*. The 'provides' interface of a CBMCalc component is specified just like a familiar OOP class. CBMCalc families are simply defined as a collection of their respective components. Clients in CBMCalc are pieces of code that are applicable to any family, provided that the family contains the specified minimal component

combination. (See Figure 6.10.) This is the first source of polymorphism in CBMCalc: A client (or component) code can be reused amongst all such families (but, not other ones). We refer to this property as the *minimal shape exposure*. Again, CBMCalc uses family parameterisation to that end.

Polymorphism in CBMCalc comes from another source as well: *component late-binding*. The family parameterisation used for a CBMCalc component or client enforces the availability of certain components. Instead of providing the exact requested components, however, the family is free to mix an *equivalent* component in. (See (S-SEq) in Figure 6.6.) As such, the exact behaviour of a family-parameterised code is not known until the actual family used for instantiation is provided. This is our notion of component late-binding.

In the presence of the above two sorts of polymorphism, CBMCalc components and methods are chosen to be *type*-monomorphic. Type polymorphism is already well researched, and, we see little extra service that type polymorphic components and methods can bring to CBSE, if any. For simplicity reasons, on the other hand, CBMCalc only models parameterisation on a single family parameter. We anticipate that generalising CBMCalc to multiple family parameters is routine. CBMCalc also restricts family parameterisation to simply requesting availability of components in the instantiation family. More general family parameterisation that can describe arbitrary relationships between components can soon slip into computability difficulties that we would like to avoid.

By specifying its 'requires' interface using family parameterisation, a CBMCalc client (or component) determines the component combination it expects from the family that is going to use it. The client (or component) code, then, will be statically bound to the requested combination. (See (WF-VCase) in Figure 6.9 and (TE-WF$_2$) in Figure 6.12.) That is, any unrequested component access is outlawed statically. We refer to this feature as static dependency on the 'requires' interface.

For the rest of this chapter, we maintain the following conventions: Unless otherwise stated, when we refer to components, families, and clients, we mean the CBMCalc ones. Moreover,"_" is our wildcard; it shows our lack of interest in the exact piece of information that "_" is used in place of.

6.1 Syntactic Compatibility in Isolation

In this section, we formalise our verbal explanation of EC8 using Definition 6.1. We use this definition to show that $\mathcal{S}_1$ is a compatible extension to $\mathcal{L}_1$ whilst $\mathcal{S}_2$ is not (Example 6.3). This section also offers two important

results that play the key roles in justifying the rightfulness of our development in Section 8.2: Proposition 6.5 justifies our designated error message in Section 8.2 whilst Proposition 6.4 justifies the absence of such a message. The justification is through translations (similar to the one in Section 8.4.1 that are) *from* Scala to the body of math developed in this section. Thus, the mission of this section is to show that such messages (and their absence) are not exclusively particular to the systems in this thesis (i.e., those of Section 3) and are relevant whenever similar mathematical conditions hold.

We start by offering a very simple encoding of ADTs. Our encoding is inspired by the famous Church encoding of ADTs [BB85] but is far less informative. We write $\alpha \stackrel{\text{def}}{=} \oplus_1^n \tau_i$ when $\tau_1, \ldots, \tau_n$ are the cases of α.[1] When n is known and can be neglected, we write $\alpha \stackrel{\text{def}}{=} \oplus \overline{\tau}$. For the rest of this thesis, α, α', and $\alpha_1, \alpha_2, \ldots$ range over ADTs. We write $\tau \in \alpha$ when τ is a case of α. Let $\mathfrak{T}$ be the set of τs. We assume the availability of an equivalence relation $\stackrel{s}{\equiv}$ on $\mathfrak{T}$. The intuition behind $\tau_1 \stackrel{s}{\equiv} \tau_2$ is that τ_1 and τ_2 can be considered of the same syntactic category.

Definition 6.1. *Let $\alpha \stackrel{def}{=} \oplus_1^k \tau_i$ and $\alpha' \stackrel{def}{=} \oplus_1^{k'} \tau'_i$ such that $k' \geq k$. Say a function w witnesses that α' is a* **syntactically compatible extension** *to α — written as $w \models \alpha' <_{\mathscr{C}} \alpha$ — when:*

i. *$w : \{\tau_i\}_1^k \to T$ is a bijection where $T \subseteq \{\tau'_i\}_1^{k'}$ and $||T|| = k$.*

ii. *$w(\tau_i) = \tau'_j$ iff $\tau_i \stackrel{s}{\equiv} \tau'_j$.*

Notation 6.2. *Write $\alpha' <_{\mathscr{C}} \alpha$ when there exists a witness w such that $w \models \alpha' <_{\mathscr{C}} \alpha$. Similarly, write $\alpha' \not<_{\mathscr{C}} \alpha$ when there is no w such that $w \models \alpha' <_{\mathscr{C}} \alpha$.*

Example 6.3. *In our encoding, $\mathcal{L}_1 \stackrel{def}{=} \mathsf{Lam} \oplus \mathsf{App} \oplus \mathsf{Var} \oplus \mathsf{Let}$, $\mathcal{S}_1 \stackrel{def}{=} \mathsf{Lam} \oplus \mathsf{App} \oplus \mathsf{Var} \oplus \mathsf{Let} \oplus \mathsf{Seq}$, and $\mathcal{S}_2 \stackrel{def}{=} \mathsf{Val} \oplus \mathsf{App} \oplus \mathsf{Var} \oplus \mathsf{Let} \oplus \mathsf{Seq}$. Take $w = \{(\tau, \tau) \mid \tau \in \mathcal{L}_1\}$. Trivially, $w \models \mathcal{S}_1 <_{\mathscr{C}} \mathcal{L}_1$.*

Showing that $\mathcal{S}_2 \not<_{\mathscr{C}} \mathcal{L}_1$ is a bit trickier: Suppose otherwise; that is, there exists a w' such that $w' \models \mathcal{S}_2 <_{\mathscr{C}} \mathcal{L}_1$. By Definition 6.1, then, $w'(\mathsf{Lam}) = \tau$ for some $\tau \in \mathcal{S}_2$. But, that is not possible because the implication is that $\mathsf{Lam} \stackrel{s}{\equiv} \tau$, which is expectably wrong for every $\tau \in \mathcal{S}_2$. □

Proposition 6.4. *$\leq_{\mathscr{C}}$ is transitive.*

Proof. Let $w \models \alpha' <_{\mathscr{C}} \alpha$ and $w' \models \alpha'' <_{\mathscr{C}} \alpha'$. It follows by the transitivity of $\stackrel{s}{\equiv}$ and the subset relation that $w' \circ w \models \alpha'' <_{\mathscr{C}} \alpha$. ■

[1] We take $\oplus$ to be commutative and associative.

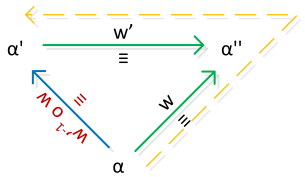


Figure 6.1: $\alpha' \not<_{\mathscr{C}} \alpha \ \wedge \ \alpha'' <_{\mathscr{C}} \alpha' \ \wedge \ \alpha'' <_{\mathscr{C}} \alpha \Rightarrow \bot$

Proposition 6.5. *Let* $\alpha \overset{def}{=} \oplus_1^k \tau_i$, $\alpha' \overset{def}{=} \oplus_1^{k'} \tau'_i$, *and* $\alpha'' \overset{def}{=} \oplus_1^{k''} \tau''_i$ *such that* $k'' \geq k' \geq k$. *Suppose that* $\alpha' \not<_{\mathscr{C}} \alpha$ *and* $\alpha'' <_{\mathscr{C}} \alpha'$. *Then,* $\alpha'' \not<_{\mathscr{C}} \alpha$.

Proof. Suppose otherwise, that is: $w \models \alpha'' <_{\mathscr{C}} \alpha$ for some appropriate w (w.r.t. Definition 6.1). By assumption, there exists w' such that $w' \models \alpha'' <_{\mathscr{C}} \alpha'$. But, then, because $\overset{s}{\equiv}$ is an equivalence relation, it follows that $w'^{-1} \circ w \models \alpha' <_{\mathscr{C}} \alpha$, which is a contradiction. Figure 6.1 depicts this diagrammatically.

To prove that $w'^{-1} \circ w \models \alpha' <_{\mathscr{C}} \alpha$, note first that $w'^{-1} \circ w$ is bijection because w and w' are bijections. Secondly, using elementary cardinal calculus, one observes that the codomain size of $w'^{-1} \circ w$ is the same as the domain size of w. It remains to show that $w'^{-1} \circ w(\tau_i) = \tau'_j$ iff $\tau_i \overset{s}{\equiv} \tau'_j$. Suppose that $w'^{-1} \circ w(\tau_i) = \tau'_j$. Then, there exists a τ''_l such that $w(\tau_i) = \tau''_l$ and $w'^{-1}(\tau''_l) = \tau'_j$. The latter equation can be rewritten as $w'(\tau'_j) = \tau''_l$. However, $w \models \alpha'' <_{\mathscr{C}} \alpha$ and $w' \models \alpha'' <_{\mathscr{C}} \alpha'$. By Definition 6.1, thus, the implication is that $\tau_i \overset{s}{\equiv} \tau''_l$ and $\tau''_l \overset{s}{\equiv} \tau'_j$. It follows by the transitivity of $\overset{s}{\equiv}$ that $\tau_i \overset{s}{\equiv} \tau'_j$, as desired. The other direction is similar. ■

Proposition 6.5 states that when α' is not a syntactically compatible extension to α, every compatible extension to α' is neither a syntactically compatible extension to α. Armed with Definition 6.1, we formalise EC8 to be the property of statically rejecting the attempt of extending α to α' when $\alpha' \not<_{\mathscr{C}} \alpha$.

Remark 6.6. Note how, in this section, the definition of $\alpha_1 <_{\mathscr{C}} \alpha_2$ is *existential* (Notation 6.2). Propositions 6.4 and 6.5 work on that basis by supposing existence or non-existence of an appropriate witness for compatibility of extensions. On the contrary, in the subsequent sections of this chapter, we introduce CBMCalc, in which provision of such a witness is a duty of the user. CBMCalc will reject an extension attempt when the witness that the user **manually** provides does not testify compatibility. (See Figure 6.8.) This is the case even when a different witness may exist. CBMCalc is designed to refrain from automatic search of the combinatorial space for possible witnesses. In a real-world PL, such an automatic search would be

Relatives

D_γ	$::=$ component γ<$X \lhd \oplus\overline{\gamma}$> $\lhd P[\gamma]\{\overline{Rf}\,;\ K\ \overline{M_\gamma}\}$	component definition
D_C	$::=$ client C<$X \lhd \oplus\overline{\gamma}$> $\lhd P[C]\{\overline{M_C}\}$	client definition
G	$::= \gamma \mid C$	relative definition name
$P[G]$	$::= G$<X> $\mid G$<X as $\oplus\,\overline{\gamma}$> $\mid \top$	parent declaration
R	$::= X \mid X.\gamma$	relative type
K	$::= \gamma(\overline{Rf})\{$super.$\overline{f} = \overline{f}\,;$ this.$\overline{f} = \overline{f}\,;\}$	constructor
M_γ	$::= R\ m_\gamma(\overline{R}\ \overline{x})\{$return $e_r\,;\}$	component method
M_C	$::= R\ m_C(\overline{R}\ \overline{x})\{$return $e_C\,;\}$	client method
e_r	$::= x \mid e_r.f \mid e_r.m_\gamma(\overline{e_r}) \mid$ new $X.\gamma(\overline{e_r})$	relative expression
e_C	$::= e_r \mid m_C(\overline{e_C}) \mid$ super.$m_C(\overline{e_C})$	client expression

Absolutes

A	$::=$	$\Phi \mid \gamma$<Φ>	absolute type
D_Φ	$::=$	family $\Phi = \oplus\overline{\gamma}$	family definition
t	$::=$	val $y =$ new γ<Φ>$(\overline{y})$	term (component instantiation)
I	$::=$	C<Φ> $\mid C$<Φ as $\oplus\,\overline{\gamma}$>	client instance
τ	$::=$	$D_\Phi\,;\overline{t}\,; I.m_C(\overline{y})$	test

Figure 6.2: The CBMCalc Syntax

considered uneconomic. Hoping to inspire such PLs, we designed CBMCalc to function likewise. □

6.2 The CBMCalc Syntax

Figure 6.2 shows the syntax of CBMCalc. Here, γ ranges over components, C over clients, K over constructors, m over methods, x over variables, e over expressions, f over fields, Φ over families, and y over top-level bound variables. We take this to be a reserved variable name and super a keyword. Priming a syntactic metavariable or subscripting it with numbers does not change its syntactic category. When distinction between a metavariable of a component and that of a client is required, we use subscripts γ and C. When referring to both categories collectively, we drop the subscript. For example, m_γ and m_C denote component methods and client methods, respectively, but m can be either an m_γ or an m_C. Note that neither subscript is related to a particular component γ or client C. A singularity in our syntax is X, which is exclusively used as the *family parameter* – with no primes, superscripts,

or subscripts.

The syntax in Figure 6.2 is divided into two parts: relative and absolute. The former syntax is all relative to the family parameter X and is about before substitution of a (real) family for X. The latter, on the other hand, is regarding after the introduction of a family, when the defined family gets employed for substitution for the family parameter X of the relative syntax.

Following the tradition of lightweight family polymorphism (Section 4.4), the overline notation is used for a list of objects. For example, for some known n: $\overline{\gamma}$ denotes $\gamma_1\gamma_2\ldots\gamma_n$; $G \vdash \overline{M}$ ok denotes $G \vdash M_1$ ok $\cdots$ $G \vdash M_n$ ok; $\overline{\gamma}\texttt{<:}\overline{\gamma'}$ denotes $\gamma_1\texttt{<:}\gamma'_1 \cdots \gamma_n\texttt{<:}\gamma'_n$; $\mathsf{this}.\overline{f} = \overline{f}$ means $\mathsf{this}.f_1 = f_1, \ldots, \mathsf{this}.f_n = f_n$. The notation $\overline{\gamma}$ is also overloaded to mean the list $\gamma_1, \gamma_2, \ldots, \gamma_n$, when appropriate. We extend that notation for $\oplus\overline{\gamma}$ to mean $\gamma_1 \oplus \gamma_2 \oplus \ldots \gamma_n$. We use ϵ for empty lists.

Using the relative syntax, one can define components (γ) and clients (C). Both clients and components take family parameters. The notation $X \lhd \oplus\overline{\gamma}$ stipulates that the family to be substituted for X has to include components (that are equivalent to) $\gamma_1, \gamma_2, \ldots, \gamma_n$. In such a case, we say that $\oplus\overline{\gamma}$ is the *upper bound* of X. Similarly, we also say that γ_i is an upper bound of X, for $i = 1, 2, \ldots, n$. Moreover, we call n the *upper bound size* of X. We likewise call n the upper bound size of G – written as $\#G = n$ – when $G\texttt{<}X \lhd \oplus\overline{\gamma}\texttt{>}$.

The syntax for declaring parents is similar for components and clients. Accordingly, we use the $P[G]$ meta-notation as a unification of the two. When G states that it derives from $G'\texttt{<}X\texttt{>}$, the same family substituted for X in G will be substituted for X in G'. On the other hand, $G \lhd G'\texttt{<}X\ \mathsf{as}\ \oplus\overline{\gamma}\texttt{>}$ signifies the situation when only the selected components $\overline{\gamma}$ of the family substituted for X in G will be substituted for X in G'. The use of this notation is a must when $\#G' < \#G$. The only exception is when $\#G' = n'$ and the first n' components in the list of G's upper bounds are to be passed, in their original order, to G'. In such a case, we allow $G \lhd G'\texttt{<}X\texttt{>}$ as a shorthand for $G \lhd G'\texttt{<}X\ \mathsf{as}\ \gamma_1 \oplus \cdots \oplus \gamma_{n'}\texttt{>}$.[2] For a given G, we overload the notation $P[G]$ for referring to the nominated (direct) parent of G. Finally, G might decide not to derive from any parent, in which case, $\top$ is taken to be its only parent. When γ is an upper bound of X, the relative type $X.\gamma$ – read the component γ of X – is the component substituted for γ when substituting a family for X.

The syntax for introduction of a family is as simple as enumerating the

[2]This syntax is also meaningful when the order of components to be passed to G' is not exactly the same as their order of appearance in G's definition. Yet, the discussion in this paragraph is not particularly concerned about the order in which components are enumerated. The order becomes significant in the subsequent sections of this chapter when **same-indexed**ness is assumed. See Figures 6.3, 6.8, and 6.10.

components it combines, interleaved by "$\oplus$". In order to bind a variable, a component has to be instantiated by substituting a family for its family parameter and possibly passing other bound variables to its constructor. To test a client on a family, one calls a method of the client by passing appropriate arguments to the method. To that end, either the whole family is used for client instantiation (C<Φ>) or a *projection* of it (C<Φ as $\oplus\overline{\gamma}$>). Note that, unlike components, we choose clients to store no data and simply act as a collection of methods. As such, clients need no constructors.

A test τ contains a family introduction, a number of term definitions, and a call of a method of a client instantiated by the introduced family (or a projection of it). The arguments passed to such a method call can be of the variables bound by the terms defined over the test. A CBMCalc program is a test along with the components and clients that it uses. For a CBMCalc program to compile and run, we assume the availability of two more ingredients: First, a customary *class table* CT, which is a simple mapping from the names of the program components, clients, and families to their bodies. Second, an equivalence relation $\overset{s}{\equiv}$ as a repository for the components to be considered equivalent. (More on the $\overset{s}{\equiv}$ of CBMCalc in Remark 6.9.)

When the premises of a CBMCalc rule contain a statement of the form $\gamma \lhd \cdots\{\cdots\}$, we are abbreviating $CT(\gamma) = \mathsf{component}\ \gamma \lhd \cdots\{\cdots\}$. The situation is similar for clients. Likewise, $\Phi = \oplus\overline{\gamma}$ abbreviates $CT(\Phi) = \mathsf{family}\ \Phi = \oplus\overline{\gamma}$. With such abbreviations, we drop explicit mention of CT over CBMCalc rules.

Remark 6.7. Note that, for a definition $\mathsf{component}\ \gamma$<$X \lhd \oplus\overline{\gamma}$>$\cdots\{\cdots\}$, it is valid that $\gamma' \in \overline{\gamma}$ whilst $\gamma \neq \gamma'$. That is, a CBMCalc component can enforce the availability of other components than itself. This is particularly useful for implementing The Common Closure Principle (CCP) of Martin [Mar00]: "Classes that change together belong together." See also togetherness for how we materialise that in our codebase.

Given that, by requesting the availability of its upper bounds, a component is expressing the services it expects, we find CCP a must for a model for CBSE. That is why CBMCalc offers upper bound declarations as **the** means for expressing the 'requires' interface of a component. CBMCalc could have similarly been designed to implement The Common Reuse Principle (CRP) of Martin [Mar00]: "Classes that aren't reused together should not be grouped together." Yet, we decided not to do so for whether CRP is a part of the 'requires' interface of a CBSE component is debatable. Hence, we avoided that in the favour of simplicity. With its special focus on CBM, our codebase, however, does address CRP via ensuring disjointedness and

uncongeniality. In fact, EC5 considers CRP an essential for CBM. □

Remark 6.8. Despite not supporting multiple inheritance, CBMCalc can simulate EC1 via linearisation of inheritance. Its support for EC2 is obvious for addition of new components and clients has no effect on existing CBMCalc programs. CBMCalc does also clearly support EC3. The support of CBMCalc for EC4 is through checking component and client well-formedness (Figure 6.10). EC5 was discussed above in Remark 6.7. The built-in support of CBMCalc for family definition addresses EC6. CBMCalc also offers direct support for EC7 by legislating the pass of an extension when a core is expected. This is done using its special X as $\oplus\,\overline{\gamma}$ notation, when needed. Section 6.3.3 details how CBMCalc supports EC8. □

6.3 The Static Semantics of CBMCalc

This section aims at providing typing rules for CBMCalc, as done in Section 6.3.5. Two major tools to that end are well-formedness (Section 6.3.4) and compatibility verification upon extension (Section 6.3.3). The subsections 6.3.1 and 6.3.2 present two other tools for that purpose: lookup functions and subtyping, respectively.

6.3.1 Lookup Functions

$$\frac{G\texttt{<}X \lhd \oplus\overline{\gamma}\texttt{>} \lhd G'\texttt{<}X\texttt{>}\{\cdots\} \qquad G'\texttt{<}X \lhd \oplus\overline{\gamma}'\texttt{>}\cdots\{\cdots\}}{rtup(R, G', G) = R[X.\overline{\gamma}/X.\overline{\gamma}']}\ (\text{RTUp-WFam})$$

$$\frac{G\texttt{<}\cdots\texttt{>} \lhd G'\texttt{<}X \text{ as } \oplus\,\overline{\gamma}\texttt{>}\{\cdots\} \qquad G'\texttt{<}X \lhd \oplus\overline{\gamma}'\texttt{>}\cdots\{\cdots\}}{rtup(R, G', G) = R[X.\overline{\gamma}/X.\overline{\gamma}']}\ (\text{RTUp-PFam})$$

Figure 6.3: Relative Type Updated for CBMCalc

Before we move to the lookup functions themselves, we explain some auxiliary functions that will become handy over the lookup. We begin by the function $rtup(.)$. When a type R relative to a parent G' is to be used by a child G, it needs to be updated as stated in Figure 6.3. This is because the relative types of the parent need not to make sense verbatim to the child. After all, a parent G' may enforce the availability of a component γ' that the child G does not. Yet, in such a case, G must have enforced an *equivalent*

component γ to substitute for γ'. (See (S-SEQ) in Figure 6.6.) That is why, in both (RTUP-WFAM) and (RTUP-PFAM), moving from G' to G, the function $rtup(.)$ **replace**s $X.\overline{\gamma'}$ by $X.\overline{\gamma}$. The difference between the rules of Figure 6.3 is on whether G passes X itself to G' or a projection of it.

Note that the replacement process is order-sensitive: For every appropriate i, the component γ'_i is replaced by the **same-indexed** component γ_i. We maintain a similar convention throughout CBMCALC: Component combinations are considered in their exact same order that they are listed by the programmer. The CBMCALC semantics does not put effort into trying other (equally meaningful) permutations.

As the other auxiliary function, we define $pfcmix(I) = (\Phi, \oplus\overline{\gamma})$ when either $I = C\texttt{<}\Phi\texttt{>}$ and $\Phi = \oplus\overline{\gamma}$, or $I = C\texttt{<}\Phi\ \texttt{as}\ \oplus\overline{\gamma}\texttt{>}$. In words, $pfcmix(I)$ is 'the *passed family and components mixture* to get I.'

$$fields(\top) = \epsilon \qquad \text{(F-TOP)} \qquad\qquad fields(X) = \epsilon \qquad \text{(F-FPAR)}$$

$$\frac{\texttt{component}\ \gamma\texttt{<}\cdots\texttt{>} \lhd \cdots \{\overline{Rf}; \cdots\} \quad fields(P[\gamma]) = \overline{R'f'} \quad \overline{R''} = rtup(\overline{R'}, G', G)}{fields(\gamma) = \overline{Rf}, \overline{R''f'}}\ \text{(F-COMP)}$$

$$fields(X.\gamma) = fields(\gamma) \qquad \text{(F-VCASE)}$$

$$fields(\gamma\texttt{<}X\ \texttt{as}\ \oplus\overline{\gamma}\texttt{>}) = fields(\gamma) \qquad \text{(F-FPROJ)}$$

$$\frac{\Phi = \gamma_1 \oplus \cdots \oplus \gamma_n \quad fields(\gamma_i) = \overline{Rf} \quad 1 \leq i \leq n}{fields(\gamma_i\texttt{<}\Phi\texttt{>}) = \overline{R}[\Phi/X, \overline{\gamma}\texttt{<}\Phi\texttt{>}/X.\overline{\gamma}]\ \overline{f}}\ \text{(F-CASE)}$$

Figure 6.4: Field Lookup of CBMCALC

Figures 6.4 and 6.5 are routine lookup functions for the fields and method types. We start the explanation by the former: The type $\top$ has no fields, hence, (F-TOP). The situation is similar for X justifying (F-FPAR). The rule (F-COMP) states that the fields of a component are those mentioned in its own body in addition to the duly updated ones inherited from its (direct or indirect) parents. It is especially important to understand the necessity of $rtup(.)$ in (F-COMP): Otherwise, the inherited fields will be of types that are relative to the parent they are originally defined in. Note also that we maintain a convention on the field names being distinct.

(F-VCASE) is specifically needed for when requesting the fields of a parent (right branch of the (F-COMP) premises) that receives a projection $\oplus\overline{\gamma}$ of the enforced components. Given that (F-VCASE) acts on relative types

(i.e., before the substitution of real family components for the requested ones), fields of a component are still those nominated. It is, finally, when a family (or a projection of it) is used for component instantiation that relative types are replaced by absolute ones. (F-Case) formulates the process for fields.

$$\frac{G\{\cdots \overline{M} \cdots\} \qquad R\ m(\overline{R}\ \overline{x})\{\cdots\} \in \overline{M}}{mtype(m, G) = \overline{R} \to R}\ \text{(MT-G)}$$

$$\frac{\begin{array}{cc} G\{\cdots \overline{M} \cdots\} & m \notin \overline{M} \qquad mtype(m, P[G]) = \overline{R} \to R \\ \overline{R'} = rtup(\overline{R}, G', G) & R' = rtup(R, G', G) \end{array}}{mtype(m, G) = \overline{R'} \to R'}\ \text{(MT-Super)}$$

$$\frac{\begin{array}{cc} pfcmix(I) = (\Phi, \oplus\overline{\gamma}) & I = C\texttt{<}\cdots\texttt{>} \\ C\texttt{<}X \lhd \oplus\overline{\gamma}'\texttt{>} & mtype(m_C, C) = \overline{R} \to R \end{array}}{mtype(m_C, I) = (\overline{R} \to R)[\Phi/X, \overline{\gamma}\texttt{<}\Phi\texttt{>}/X.\overline{\gamma}]}\ \text{(MT-Inst)}$$

Figure 6.5: Method Type Lookup of CBMCalc

When a method m is already in the body of G, the method type of m in G, as stated by (MT-G), is trivially calculated using its nominated argument and return types. Otherwise, when m is inherited from a parent G', the method type of m in G, as stated by (MT-Super), is the duly updated type of m in G'. Lastly, as stated by (MT-Inst), the method type of m in an instance of a client C is that of the C itself, updated with the instantiation information.

6.3.2 Subtyping

The subtyping rules of CBMCalc are presented in Figure 6.6. In line with Figure 6.2, the rules are divided into those pertaining to prior family introduction (Relative Subtyping) and after that (Absolute Subtyping). In the former group, the decision is made with respect to the relative enclosing definition (G), whereas, for the latter, a family (Φ) is the basis of decision. Figure 6.6 adds another group of rules, i.e., Free Subtyping. We call these the free subtyping rules because they are not bound to the family introduction action. The free subtyping relation is the transitive closure of the union of $\lhd$ and $\overset{s}{\equiv}$.

Most rules here are standard. Below, we explain the new ones: The rules (S-VCase), (S-Case), and (S-SEq) are specifically relevant to (component-based) mechanisation. For other CBSE, the $\overset{s}{\equiv}$ in the last rule may be replaced

Relative Subtyping $\boxed{G \vdash R <: R'}$

$$G \vdash R <: R \quad \text{(S-ReflG)} \qquad G \vdash \gamma <: P[\gamma] \quad \text{(S-CParG)}$$

$$\frac{G \vdash R <: R' \qquad G \vdash R' <: R''}{G \vdash R <: R''}\ \text{(S-TranG)} \qquad \frac{fpub(G) = \oplus\overline{\gamma}}{G \vdash X.\gamma_i <: X}\ \text{(S-VCase)}$$

$$\frac{P[G] \vdash R <: R'}{G \vdash rtup(R, P[G], G) <: rtup(R', P[G], G)}\ \text{(S-Super)}$$

Absolute Subtyping $\boxed{\Phi \vdash A <: A'}$

$$\Phi \vdash A <: A \quad \text{(S-ReflF)} \qquad \frac{\Phi = \oplus\overline{\gamma}}{\Phi \vdash \gamma_i\langle\Phi\rangle <: \Phi}\ \text{(S-Case)}$$

$$\frac{\Phi \vdash A <: A' \qquad \Phi \vdash A' <: A''}{\Phi \vdash A <: A''}\ \text{(S-TranF)}$$

Free Subtyping $\boxed{\gamma <: \gamma'}$

$$\frac{\gamma \overset{s}{\equiv} \gamma'}{\gamma <: \gamma'}\ \text{(S-SEq)} \qquad \gamma <: P[\gamma] \quad \text{(S-CPar)} \qquad \frac{\gamma <: \gamma' \quad \gamma' <: \gamma''}{\gamma <: \gamma''}\ \text{(S-Trans)}$$

Figure 6.6: The Three Sorts of Subtyping in CBMCalc

by the component equivalence relationship of the discourse. The two former rules are along the lines that the cases of an ADT are all different forms of it. The auxiliary function *fpub*(.) – *family parameter upper bound* – used in the former two rules is defined as:

$$fpub(G\langle X \ \lhd \ \oplus\overline{\gamma}\rangle) = \oplus\overline{\gamma}.$$

Remark 6.9. As announced in Section 6.2, like the case with Section 6.1, CBMCalc assumes the availability of an equivalence relation $\overset{s}{\equiv}$. Replacement of a component by an equivalent should go unobserved by the recipient code. The two aspects of this unobservedness are: constructibility with the same syntax and provision of the same interface.[3] In presence of those aspects, CBMCalc takes care of performing the requested replacements in

[3]This is similar to the *substitutability* of Liskov [Lis87], except that her work was exclusive to inheritance.

two places: *rtup*(.) for the static semantics (Figure 6.3) and *trace-ft*(.) for the dynamic semantics (Figure 6.13). Note that, whilst we require equivalent components to share the observed interface, their implementation of the requested services – as well as the rest of their interfaces – can well differ.

Many real-world PLs do already have means that can be leveraged to guarantee the above unobservedness. Examples are the ordinary C++ template mechanism, C++ concepts or HASKELL type classes, and the implicits of Scala. The study of advantages and disadvantages of each means is a topic for future research. Given our development in Scala, we use the last means in our codebase. (See Section 8.3 for more.) □

6.3.3 Compatibility of Extensions

$$\frac{\mathit{fpub}(G) = \oplus\overline{\gamma} \qquad \{\overline{\gamma_p}\} \subseteq \{\overline{\gamma}\}}{\mathit{valid\text{-}as}(G, \oplus\overline{\gamma_p})}\ (\textsc{Valid-As-G})$$

$$\frac{\Phi = \oplus\overline{\gamma} \qquad \{\overline{\gamma_p}\} \subseteq \{\overline{\gamma}\}}{\mathit{valid\text{-}as}(\Phi, \oplus\overline{\gamma_p})}\ (\textsc{Valid-As-F})$$

Figure 6.7: Valid Projection of Component Combinations in CBMCALC

This section is at the heart of this chapter. The material in this section is what we use to validate the legitimacy of extensions to a component or a client, which corresponds to EC8. As also stated earlier on, the special attention we pay to EC8 is because it is the least-studied ECP concern.

We start by the auxiliary function *valid-as*(.) in Figure 6.7. This function answers the question of whether the component combination passed as its second argument is a valid projection of its first argument. The essence of *valid-as*(.) is the simple subset check $\{\overline{\gamma_p}\} \subseteq \{\overline{\gamma}\}$ where $\oplus\overline{\gamma}$ is the available component combination and $\oplus\overline{\gamma_p}$ is the requested projection. The rule (VALID-AS-G) handles the case for components and clients, whilst (VALID-AS-F) handles that for families. The latter rule will be used in the next subsections but not in checking compatibility of extensions. Yet, we included it in Figure 6.7 for its share of nature with the former rule. We now move to checking compatibility of an extension.

Recalling our meta-notation from Figure 6.2 for parent declaration ($P[G]$), Figure 6.8 divides the decision on compatibility of extensions into the three respective cases: The rule (CE-TOP) explains it that G is always a compatible extension to its parent when the parent is $\top$. When, upon extension, G

$$\frac{P[G] = \top}{G <_{\mathscr{C}} P[G]}\ (\text{CE-Top})$$

$$\frac{\mathit{fpub}(G) = \oplus\overline{\gamma} \qquad \mathit{fpub}(P[G]) = \oplus\overline{\gamma}' \qquad \overline{\gamma} <: \overline{\gamma'}}{G <_{\mathscr{C}} P[G]}\ (\text{CE-WFam})$$

$$\frac{\begin{array}{l} P[G] = G'\texttt{<}X \ \ \texttt{as}\ \ \oplus\,\overline{\gamma_p}\texttt{>} \qquad \mathit{valid\text{-}as}(G, \oplus\overline{\gamma_p}) \\ \mathit{fpub}(G') = \oplus\overline{\gamma}' \qquad\qquad\qquad\qquad\qquad\quad \overline{\gamma_p} <: \overline{\gamma'} \end{array}}{G <_{\mathscr{C}} P[G]}\ (\text{CE-PFam})$$

Figure 6.8: Compatible Extensions in CBMCalc

passes the family parameter X intact to its nominated parent, the rule (CE-WFam) stipulates it that the extension is only valid when the requested components of G are all subtypes of the **same-indexed** components requested by its parent. The last rule is (CE-PFam), which corresponds to the situation when, upon extension, G passes a projection of its requested components to its parent. (This is expressed by the premise $P[G] = G\texttt{<}X\ \texttt{as}\ \oplus\overline{\gamma_p}\texttt{>}$ of (CE-PFam).) In such a case, the additional check with respect to (CE-WFam) is on whether the projection $\oplus\overline{\gamma_p}$ is valid for the components that G requests. Note that, for (CE-PFam) as well, the order in which the projected components of G are passed to G' is significant for validity of the extension.

6.3.4 Well-Formedness

In correspondence with Figure 6.2, Figures 6.9 and 6.10 divide the well-formedness rules into two parts: relative and absolute, respectively. When a relative entity r is well-formed in G, we write "$G \vdash r$ ok". Likewise, when an absolute entity a is well-formed, we write "a ok". The rest of this section provides informal explanation for both. An auxiliary function that we use for the latter part is $\mathit{fpub}^*(.)$, which is essentially a transitive closure of $\mathit{fpub}(.)$:

$$\mathit{fpub}^*(G) = \mathit{fpub}(G) \cup \bigcup_{\gamma \in \mathit{fpub}(G)} \mathit{fpub}^*(\gamma).$$

The rules (WF-MComp) and (WF-MCli) are routine. They pertain to the well-formedness of component and client methods, respectively. (WF-Ctor) is about (component) constructors. It works by checking that the expressions used for initialisation of the self fields are of the exact same nominated types and that those used for the parent are of a subtype of the nominated ones. Note our discrimination here between initialisation of self fields

$$\frac{\overline{x} : \overline{R}, \mathsf{this} : X.\gamma; \gamma \vdash e_\gamma : R' \quad \gamma \vdash R' <: R \qquad \gamma \vdash R, R', \overline{R}\ \text{ok}}{\gamma \vdash R\ m_\gamma(\overline{R}\ \overline{x})\{\mathsf{return}\ e_\gamma;\}\ \text{ok}}\ (\text{WF-MComp})$$

$$\frac{\overline{x} : \overline{R}; C \vdash e_C : R' \qquad C \vdash R' <: R \qquad C \vdash R, R', \overline{R}\ \text{ok}}{C \vdash R\ m_C(\overline{R}\ \overline{x})\{\mathsf{return}\ e_C;\}\ \text{ok}}\ (\text{WF-MCli})$$

$$\frac{\mathit{fields}(P[\gamma]) = \_\ \overline{f'} \qquad \mathit{fields}(\gamma) = \overline{R}\ \overline{f}, \overline{R'}\ \overline{f'} \qquad \gamma \vdash R'' <: R'}{\gamma(\overline{R}\ \overline{f}, \overline{R''}\ \overline{f''})\{\mathsf{super}.\overline{f'} = \overline{f''};\ \ \mathsf{this}.\overline{f} = \overline{f};\}\ \text{ok}}\ (\text{WF-Ctor})$$

$$\frac{\gamma <_{\mathscr{C}} P[\gamma] \qquad K\ \text{ok} \qquad \gamma \vdash \overline{M_\gamma}\ \text{ok} \qquad \gamma \vdash \overline{R}\ \text{ok}}{\mathsf{component}\ \gamma \lhd P[\gamma]\{\overline{R}\ \overline{f};\ K\ \overline{M_\gamma}\}\ \text{ok}}\ (\text{WF-Comp})$$

$$\frac{C <_{\mathscr{C}} P[C] \qquad C \vdash \overline{M_C}\ \text{ok}}{\mathsf{client}\ C \lhd P[C]\{\overline{M_C}\}\ \text{ok}}\ (\text{WF-Cli})$$

$$\frac{}{G \vdash \epsilon\ \text{ok}}\ (\text{WF-Empty}) \qquad \frac{}{G \vdash X\ \text{ok}}\ (\text{WF-FPar})$$

$$\frac{\mathit{fpub}(G) = \oplus_1^n \gamma_i}{G \vdash X.\gamma_i\ \text{ok}}\ (\text{WF-VCase})$$

Figure 6.9: Relative Well-Formedness in CBMCalc

and inherited ones: For better accommodation of component refinement using inheritance, we allow a constructor to submit expressions of subtypes for initialisation of inherited fields. (WF-Comp) states that a component is well-formed when it is a compatible extension to its nominated parent and the following are all well-formed: its constructor, its methods, and field types. Given that a client has no constructor or fields, (WF-Cli) is similar but only checks compatibility of the nominated extension and the well-formedness of (client) methods. The rules (WF-Empty) and (WF-FPar) state, respectively, that the empty list and the family variable are always well-formed with respect to a relative definition. Finally, due to (WF-VCase), $X.\gamma_i$ – i.e., the component γ_i of the family parameter X – is only well-formed when the enclosing relative definition G has already enforced the availability of γ_i in the family with which to instantiate G.

According to (WF-Family), a family definition is well-formed when all the components transitively requested by the ones mixed into the family (or a subtype of them) are indeed in the family mixture. The rule (WF-WFInst) mentions two groups of premises for instantiation of a client C

$$\frac{\forall \gamma_i' \in \mathit{fpub}^*(\gamma_i) \exists \gamma_j.\ \gamma_j <: \gamma_i' \qquad (1 \le i, j \le n)}{\mathsf{family}\ \Phi = \gamma_1 \oplus \cdots \oplus \gamma_n\ \mathrm{ok}}\ (\text{WF-FAMILY})$$

$$\frac{\Phi = \oplus_1^n \gamma_i \qquad C\texttt{<}X \lhd \oplus_1^n \gamma_i'\texttt{>} \qquad \overline{\gamma} <: \overline{\gamma'} \qquad \forall \gamma'' \in \mathit{fpub}^*(\overline{\gamma'}) \exists \gamma_j.\ \gamma_j <: \gamma'' \qquad (1 \le i, j \le n)}{C\texttt{<}\Phi\texttt{>}\ \mathrm{ok}}\ (\text{WF-WFINST})$$

$$\frac{\mathit{valid\text{-}as}(\Phi, \oplus_1^n \gamma_i) \qquad C\texttt{<}X \lhd \oplus_1^n \gamma_i'\texttt{>} \qquad \overline{\gamma} <: \overline{\gamma'} \qquad \forall \gamma'' \in \mathit{fpub}^*(\overline{\gamma'}) \exists \gamma_j.\ \gamma_j <: \gamma'' \qquad (1 \le i, j \le n)}{C\texttt{<}\Phi\ \mathsf{as}\ \oplus_1^n \gamma_i\texttt{>}\ \mathrm{ok}}\ (\text{WF-CPINST})$$

Figure 6.10: Well-Formedness of Absolutes in CBMCALC

using the family Φ to be well-formed: First, for every requested component γ' of C, the **same-indexed** component γ mixed into Φ has to be a subtype of γ'. (Note that Φ has to mix the same number of components as requested by C.) Second, Φ has to also mix all the components transitively requested by C. (WF-CPINST) is similar except that, instead of being based on the entire Φ combination, it is based on the requested projection of Φ. As such, it first checks the validity of the requested projection. Furthermore, the requested projection (as opposed to Φ itself) has to be of the same size as the number of components requested by C. The other premises of (WF-CPINST) are like those of (WF-WFINST).

6.3.5 Typing

Figure 6.11 illustrates the typing rules for CBMCALC expressions. The scheme of these rules is $\Gamma; G \vdash e : R$. Here, Γ is a type environment, which is a partial function from variable names to their types. The scheme reads 'with the type environment Γ and inside G, the expression e is typed to R.'

The rules (T-VAR) and (T-FIELD) are routine. The rule (T-INVK$_1$) is less straightforward in that it allows the arguments passed to a method to also be of subtypes of the respective nominated parameter types. (T-NEW) offers the same flexibility when sending arguments to a component γ's constructor. In addition, it checks if γ is basically well-formed in G. (T-INVK$_2$) is like (T-INVK$_1$) but simpler: After all, unlike an m_γ, an m_C is not called on an expression. According to (T-SUPER), a super invocation in a client is a normal invocation in its parent.

Figure 6.12 is on our single rule for typing a test τ. (Recall the syntax in Figure 6.2.) Here, we employ a new type environment that we call the

$$\boxed{\Gamma; G \vdash e : R}$$

$$\Gamma; G \vdash x : \Gamma(x) \qquad \text{(T-Var)}$$

$$\frac{\Gamma; \gamma \vdash e_\gamma : R \qquad \mathit{fields}(R) = \overline{R}\ \overline{f}}{\Gamma; \gamma \vdash e_\gamma.f_i : R_i}\ \text{(T-Field)}$$

$$\frac{\begin{array}{ll}\Gamma; G \vdash e : X.\gamma & \mathit{mtype}(m, \gamma) = \overline{R} \to R \\ \Gamma; G \vdash \overline{e} : \overline{R'} & G \vdash \overline{R'} <: \overline{R}\end{array}}{\Gamma; G \vdash e.m(\overline{e}) : R}\ \text{(T-Invk}_1\text{)}$$

$$\frac{\begin{array}{ll}G \vdash X.\gamma\ \text{ok} & \mathit{fields}(X.\gamma) = \overline{R} \to R \\ \Gamma; G \vdash \overline{e} : \overline{R'} & G \vdash \overline{R'} <: \overline{R}\end{array}}{\Gamma; G \vdash \mathsf{new}\ X.\gamma(\overline{e}) : X.\gamma}\ \text{(T-New)}$$

$$\frac{\mathit{mtype}(m_C, C) = \overline{R} \to R \qquad \Gamma; C \vdash \overline{e} : \overline{R'} \qquad C \vdash \overline{R'} <: \overline{R}}{\Gamma; C \vdash m_C(\overline{e}) : R}\ \text{(T-Invk}_2\text{)}$$

$$\frac{\Gamma; P[C] \vdash m_C(\overline{e}) : R}{\Gamma; C \vdash \mathsf{super}.m_C(\overline{e}) : R}\ \text{(T-Super)}$$

Figure 6.11: Expression Typing in CBMCalc

test type environment. This type environment is built inductively out of the terms introduced by a test. (TE-WF$_1$) states that the test type environment is initially well-formed when no term is introduced. The inductive step is taken by (TE-WF$_2$) that builds on top of the simple lookup process of (TE-Lkup). The rule (TE-WF$_2$) describes when addition of a new term t to a well-formed test type environment $\Gamma(\overline{t})$ results in a new well-formed test type environment $\Gamma(\overline{t}t)$[4]: The (old) test type environment $\Gamma(\overline{t})$ should testify that its (already) bound variables that are used for initialising the new term (i.e., $\overline{y}$) are all appropriately typed for that purpose. Note that an important consequence of (TE-WF$_2$) is the ban of mutual recursion: Only use of previously bound variables ($\overline{y}$ in (TE-WF$_2$)) is allowed for binding a new variable (y in (TE-WF$_2$)). Lastly, according to (T-Test): In order for a test to qualify for being of a type, the family it introduces, the test type environment that it accumulates, and its client instantiation need to all be well-formed. Besides, the arguments it invokes the client method with should all be appropriately typed with respect to the family it introduces.

[4]Recall from Section 6.2 that $\overline{t}t$ is a short form for $t_1 \cdots t_n t$.

Test Type Environments

$$\Gamma(\epsilon)\ \text{ok} \qquad (\text{TE-WF}_1)$$

$$\frac{\begin{array}{lll} t{=}\mathsf{val}\ y = \mathsf{new}\ \gamma\texttt{<}\Phi\texttt{>}(\overline{y}) & \gamma \in \Phi & \Gamma(\bar{t})\ \text{ok} \\ \Gamma(\bar{t}) \vdash \overline{y} : \overline{A} & \mathit{fields}(\gamma\texttt{<}\Phi\texttt{>}) = \overline{A'}\ \_ & \Phi \vdash \overline{A} \texttt{<:} \overline{A'} \end{array}}{\Gamma(\bar{t}t)\ \text{ok}}\ (\text{TE-WF}_2)$$

$$\frac{t_i{=}\mathsf{val}\ y = \mathsf{new}\ \gamma\texttt{<}\Phi\texttt{>}(\cdots)}{\Gamma(\bar{t}) \vdash y : \gamma\texttt{<}\Phi\texttt{>}}\ (\text{TE-Lkup})$$

Test Typing $\boxed{\tau : A}$

$$\frac{\begin{array}{lll} D_\Phi\ \text{ok} & I\ \text{ok} & \mathit{mtype}(m_C, I) = \overline{A} \to A \\ \Gamma(\bar{t})\ \text{ok} & \Gamma(\bar{t}) \vdash \overline{y} : \overline{A'} & \Phi \vdash \overline{A'} \texttt{<:} \overline{A} \end{array}}{D_\Phi;\ \bar{t};\ I.m_C(\overline{y}) : A}\ (\text{T-Test})$$

Figure 6.12: The Test Typing of CBMCalc and its Auxiliaries

6.4 The Dynamic Semantics of CBMCalc

For the sake of completeness, we now present a dynamic semantics for CBMCalc. We, however, do not use this semantics in this thesis.

Evaluation of the $I.m_C(\overline{y})$ calls in a test gives rise to new expressions that are results of injecting bound variables (and what they are bound to) into relative expressions. Expectably, we refer to such new expressions as the *absolute expressions*:

$$e_a ::= \ y \mid v \mid e_a.f \mid e_a.m(\overline{e_a}) \mid \mathsf{new}\ \gamma\texttt{<}\Phi\texttt{>}(\overline{e_a}) \mid m(\overline{e_a}) \mid \mathsf{super}.m(\overline{e_a})$$

where v ranges over *values*. More on values in Section 6.4.2.

The dynamic semantics of CBMCalc is based on the operational semantics in Section 6.4.2. The auxiliary functions needed for that semantics are explained in Section 6.4.1. As described in Section 6.4.3, however, a test τ needs to first go under certain transformations to be used by the above operational semantics. Note that, unlike the works on lightweight family polymorphism (Section 4.4), the semantics that we present is big-step.

6.4.1 Auxiliaries

For a client expression e_C, the function $rtypes(.)$ returns the list of relative types mentioned in e_C:

$$\begin{array}{ll} rtypes(x) & = \epsilon \\ rtypes(e_r.f) & = rtypes(e_r) \\ rtypes(e_r.m_\gamma(\overline{e_r})) & = rtypes(e_r), rtypes(\overline{e_r}) \\ rtypes(\mathsf{new}\ X.\gamma(\overline{e_r})) & = \gamma, rtypes(\overline{e_r}) \\ rtypes(m(\overline{e_C})) & = rtypes(\overline{e_C}) \\ rtypes(\mathsf{super}.m(\overline{e_C})) & = rtypes(\overline{e_C}) \end{array}$$

Recall from the previous section that upon extension, G substitutes components out of the one it requests for those requested by its parent $P[G]$. Let $\lhd^*$ be the reflexive and transitive closure of $\lhd$. Supposing that $G \lhd^* G'$, the function $trace\text{-}ft(R, G', G)$ traces the substitution chain for the relative type R from G' down to G. Figure 6.13 depicts the algorithm. The search process is bottom-up: Starting from G, it climbs the hierarchy up until it reaches G'.

(TFT-FPar) is an axiom because, as a principle, the family parameter X never gets substituted. (TFT-Self) explains the situation when the search arrives at the destination. The inductive step is taken by the next two rules. (TFT-WFam) and (TFT-PFam) both relay the task to the parent and return the appropriately updated relative type. The update is done based on whether G passes to its parent G'' the exact same component combination that it requests or a projection of that. What is especially noteworthy here is $X.\gamma_i$ and $X.\gamma'_i$ are **same-indexed**. Moreover, given that $\top$ requests no components, there is no case in Figure 6.13 for $trace\text{-}ft(\_, \top, \_)$.

The function $mbody(m, G)$ returns the body of a method m of G along with the set of formal parameters that m takes and the definition site of m. This is specified in Figure 6.14. We overload the $mbody(.)$ function to also handle the case when a G is to be instantiated using the family Φ. When m is defined by G itself, (MB-G) handles the task easily. Otherwise, as stated by (MB-Super), we search for m in $P[G]$ and return the same pieces of information accordingly.

(MB-Inst) is slightly more complicated and deserves extra explanation. When G is to be instantiated for evaluation, the relative expressions it contains need to be realised into their corresponding absolute expressions. In order to do so, the appropriate parts of G are first extracted using the plain $mbody(m, G)$ call. Next, the relative types of the expression m returns are acquired. Such types are, then, replaced by their counterparts that are relative to G. The result is, finally, updated by replacing the most up-to-date relative types with their respective absolute ones.

$$\frac{}{\mathit{trace\text{-}ft}(X, G', G) = X}\ (\text{TFT-FPar})$$

$$\frac{G = G' \qquad \gamma \in \mathit{fpub}(G)}{\mathit{trace\text{-}ft}(X.\gamma, G', G) = X.\gamma}\ (\text{TFT-Self})$$

$$\frac{G\texttt{<}X \lhd \oplus\overline{\gamma}\texttt{>} \lhd G''\texttt{<}X\texttt{>}\{\cdots\} \qquad \mathit{fpub}(G'') = \oplus\overline{\gamma}' \quad \mathit{trace\text{-}ft}(X.\gamma, G', G'') = X.\gamma_i'}{\mathit{trace\text{-}ft}(X.\gamma, G', G) = X.\gamma_i}\ (\text{TFT-WFam})$$

$$\frac{G\texttt{<}\cdots\texttt{>} \lhd G''\texttt{<}X\ \mathsf{as}\ \oplus\overline{\gamma}\texttt{>}\{\cdots\} \qquad \mathit{fpub}(G'') = \oplus\overline{\gamma}' \quad \mathit{trace\text{-}ft}(X.\gamma, G', G'') = X.\gamma_i'}{\mathit{trace\text{-}ft}(X.\gamma, G', G) = X.\gamma_i}\ (\text{TFT-PFam})$$

Figure 6.13: Trace the Relative Type R from G' Down to G

6.4.2 Operational Semantics

Armed with the auxiliary functions introduced by the previous section, we are now ready to move to the operational semantics – as presented by Figure 6.15. The judgements in this semantics are of the form $\tau \vdash e_a@C \Downarrow v@C'$, where: '$\tau$ is the test in which evaluation of e_a *at the level* C will result in v at the level C'.' Note that clients form hierarchies with possibly many levels. In Figure 6.15, we use a number of conventions that we explain below. For a test $\tau = D_\Phi\,;\ \overline{t}\,;\ I.m_C(\overline{y})$,

1. when $\mathsf{val}\ y = \mathsf{new}\ \gamma\texttt{<}\Phi\texttt{>}(\overline{e_a}) \in \overline{t}$, define $\tau(y) ::= \mathsf{new}\ \gamma\texttt{<}\Phi\texttt{>}(\overline{e_a})$.
2. where $I = C\texttt{<}\Phi_I\texttt{>}$, define $\Phi' ::= \Phi \mid \Phi_I$. Recall from Figure 6.2 that Φ_I can either be Φ itself or a projection of it. In the operational semantics, Φ' takes the value Φ when calling a method on a y. It takes the value Φ_I otherwise. More on Φ_I in Section 6.4.3.

Furthermore, as customary in lightweight family polymorphism (Section 4.4), we assume the availability of some *base value types*. For a list of values $\overline{v}$, an appropriate γ, and an appropriate Φ, we take $\mathsf{new}\ \gamma\texttt{<}\Phi\texttt{>}(\overline{v})$ to be a value as well. Next, we explain the rules one-by-one.

The rule (R-Lkup) says that a y evaluates to what the expression it is bound to evaluates. (R-Val) is on values reducing to themselves. (R-Field) only authorises field access on an expression e when what it reduces to does provide the requested field. (R-Invk$_1$) is slightly more complicated. For the call of a method m on an expression e with arguments $\overline{e'}$, the expression e is first evaluated at the requested level C; the arguments $\overline{e'}$ are, then,

$$\frac{G\{\cdots \overline{M} \cdots\} \qquad R\ m(\overline{R}\ \overline{x})\{\mathsf{return}\ e;\} \in \overline{M}}{mbody(m, G) = (\overline{x}, e, G)}\ (\text{MB-G})$$

$$\frac{G\{\cdots \overline{M} \cdots\} \qquad R\ m(\overline{R}\ \overline{x})\{\mathsf{return}\ e;\} \notin \overline{M} \qquad mbody(m, P[G]) = (\overline{x}, e, G')}{mbody(m, G) = (\overline{x}, e, G')}\ (\text{MB-Super})$$

$$\frac{\begin{array}{ll} mbody(m, G) = (\overline{x}, e, G') & rtypes(e) = \overline{R'} \\ \overline{R} = trace\text{-}ft(\overline{R'}, G', G) & e' = e[\overline{R}/\overline{R'}] \\ fpub(G) = \oplus\overline{\gamma'} & \Phi = \oplus\overline{\gamma} \end{array}}{mbody(m, G, \Phi) = (\overline{x}, e'[\Phi/X, \overline{\gamma}\texttt{<}\Phi\texttt{>}/X.\overline{\gamma'}], G')}\ (\text{MB-Inst})$$

Figure 6.14: Body of Method m of G

evaluated at the resulting level C'; next, the body of m is extracted; finally, the appropriately substituted body of m is evaluated at C', returning the resulting value v and level C''. (R-New) expresses it that a new-expression is only a value when the arguments passed to it are all values. (R-Invk$_2$) manifests the evaluation of a client method m with arguments $\overline{e}$ at the level C. To that end, all the arguments are first evaluated at the same level, namely C; the method body is, then, extracted with the level *updated* to C'; it is at this new level C', next, that the appropriately substituted body of m is evaluated to obtain the overall resulting value v and level C''. (R-Super) stipulates it that a super call at the level of a client C is a normal call at the level of C's parent.

6.4.3 Test Evaluation

As defined in Figure 6.2, a test τ takes the form $D_\Phi\,;\overline{t}\,;I.m_C(\overline{y})$. Yet, the operational semantics we illustrated in Figure 6.15 has no pattern to match against such a form. Despite that, one can easily evaluate a test **using** the operational semantics.

Definition 6.10. *Say* $\tau = D_\Phi\,;\overline{t}\,;I.m_C(\overline{y})$ *evaluates to* v *when* $\tau \vdash m_C(\overline{y})@C \Downarrow \_ : v$, *where* $I = C\texttt{<}\cdots\texttt{>}$. *In such a case, write* $\tau \rightsquigarrow v$.

Remark 6.11. Recall from Figure 6.2 that $I = C\texttt{<}\Phi\texttt{>} \mid C\texttt{<}\Phi\ \mathsf{as}\ \oplus\overline{\gamma}\texttt{>}$. Relating back to the opening paragraph of Section 6.4.2, $\Phi_I = \Phi$ when the first alternative is used for I, and $\Phi_I = \oplus\overline{\gamma}$ otherwise. Note that the well-formedness of I implies well-formedness of a temporary family $\Phi_I = \oplus\overline{\gamma}$.

$$\boxed{\tau \vdash e_a@C \Downarrow v@C'}$$

$$\frac{\tau \vdash \tau(y)@C \Downarrow v@C'}{\tau \vdash y@C \Downarrow v@C'}\ (\text{R-Lkup}) \qquad \frac{}{\tau \vdash v@C \Downarrow v@C}\ (\text{R-Val})$$

$$\frac{\tau \vdash e@C \Downarrow \mathsf{new}\ \gamma\texttt{<}\Phi'\texttt{>}(\overline{v})@C' \qquad \mathit{fields}(\gamma\texttt{<}\Phi'\texttt{>}) = \overline{A}\ \overline{f}}{\tau \vdash e.f_i@C \Downarrow v_i@C'}\ (\text{R-Field})$$

$$\frac{\begin{array}{c}\tau \vdash e@C \Downarrow \mathsf{new}\ \gamma\texttt{<}\Phi'\texttt{>}(\overline{v})@C' \qquad \tau \vdash \overline{e}@C' \Downarrow \overline{v'}@\_ \\ \mathit{mbody}(m, \gamma, \Phi') = (\overline{x}, e', \_) \\ \tau \vdash e'[\overline{v'}/\overline{x}, \mathsf{new}\ \gamma\texttt{<}\Phi'\texttt{>}(\overline{v})/\mathsf{this}]@C' \Downarrow v@C''\end{array}}{\tau \vdash e.m(\overline{e})@C \Downarrow v@C''}\ (\text{R-Invk}_1)$$

$$\frac{\tau \vdash \overline{e}@C \Downarrow \overline{v}@\_}{\tau \vdash \mathsf{new}\ \gamma\texttt{<}\Phi'\texttt{>}(\overline{e})@C \Downarrow \mathsf{new}\ \gamma\texttt{<}\Phi'\texttt{>}(\overline{v})@C}\ (\text{R-New})$$

$$\frac{\begin{array}{c}\tau \vdash \overline{e}@C \Downarrow \overline{v}@\_ \qquad \mathit{mbody}(m, C, \Phi) = (\overline{x}, e, C') \\ \tau \vdash e[\overline{v}/\overline{x}]@C' \Downarrow v@C''\end{array}}{\tau \vdash m(\overline{e})@C \Downarrow v@C''}\ (\text{R-Invk}_2)$$

$$\frac{P[C] = C'\texttt{<}\cdots\texttt{>} \qquad \tau \vdash m(\overline{e})@C' \Downarrow v@C''}{\mathsf{super}.m(\overline{e})@C \Downarrow v@C''}\ (\text{R-Super})$$

Figure 6.15: The Operational Semantics of CBMCalc

(Check the second line of (WF-CPInst) premises against the premise of (WF-Family).) □

Conjecture 6.12. *Suppose that* $\tau : A$ *and* $\tau \rightsquigarrow v$. *When* v *is not of a base value type, it takes the form* $\mathsf{new}\ \gamma\texttt{<}\Phi'\texttt{>}(\cdots)$, *for some* Φ' *such that* $\Phi' \vdash \gamma\texttt{<}\Phi'\texttt{>}\texttt{<:}A$.

The above conjecture is a big-step version of the famous preservation results: Evaluation (Figure 6.15) preserves expression types (Figure 6.11).

Chapter 7

Syntax Components

The purpose of this chapter is to introduce our components for syntax mechanisation. To that end, we start by an overview of our entire approach (for syntax and semantics mechanisation) in Section 7.1. Next, Section 7.2 gets deeper into how to assemble the components together for a complete PL mechanisation (Section 7.2.1) and the component internals (Section 7.2.2). How two relevant ECP concerns are addressed is, then, explained in Section 7.3. This chapter ends in Section 7.4 where our concrete syntax mechanisation techniques are explored.

7.1 Overview

Figure 7.1 gives a UML overview of our approach for both syntax and semantics mechanisation. 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 cases. As an example, L1Exp binds its nested type Val to its case Lam. (See the top right portion.)

Ideally, of the elements depicted in Figure 7.1, certain ones ought to be

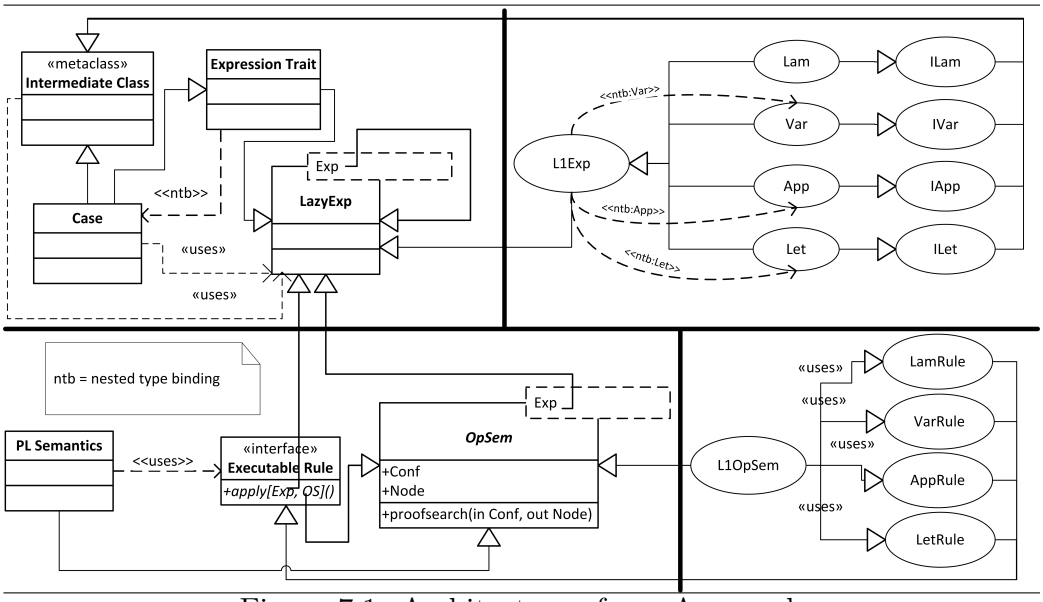


Figure 7.1: Architecture of our Approach

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 7.1, ILam, IVar, IApp, and ILet are intermediate classes, whilst LamRule, VarRule, AppRule, and LetRule 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 precision and reusability for concrete syntax that we explain later, we need to query a PL syntax for whether it 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 cases. In Section 7.4 where our concrete syntax is 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 8.1, where our semantics components are introduced.

7.2 The 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 30 case for the syntax in Section 3.1, 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 7.1.) 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 Section 3.1, we have implemented a total of 10 intermediate classes that correspond to variables (`IVar`), λ-abstractions (`ILam`), function applications (`IApp`), let-expressions (`ILet`), selective strictness (`ISeq`), metavariables (`IMVar`), generalised identifiers (`IGenIdn`), let-surrounded λ-abstractions (`IVal`), strict applications (`ISrp`), and variable-to-identifier-applications (`IVarApp`). 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 7.1.)

In Section 7.2.1, 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 7.2.2.

7.2.1 Syntax Mechanisation using ICs

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

- A PL syntax is mechanised using its own (algebraic data-) type. Such a type provides its extra storage using binding of nested types to its respective cases. When a syntax is mechanised in such a fashion, we refer to its implementing datatype as an "expression trait." Furthermore, when a case 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

```
1  trait L1Exp extends LazyExp[L1Exp]
2
3  case class Var(...) extends IVar(...)                  with L1Exp
4  case class Lam(...) extends ILam[L1Exp, Lam](...)      with L1Exp
5  case class App(...) extends IApp[L1Exp, App](...)      with L1Exp
6  case class Let(...) extends ILet[L1Exp, Lam, Let](...) with L1Exp
```

(a) Abstract Syntax Mechanisation for $\mathcal{L}_1$

```
1  package l1
2
3  trait L1Exp extends LazyExp[L1Exp] {
4    type Var = l1.Var
5    type Val = l1.Lam
6    type App = l1.App
7    type Let = l1.Let
8    ...
9  }
10 //the case definitions like Figure 7.2a
```

(b) Concrete Syntax Mechanisation for $\mathcal{L}_1$

Figure 7.2: Mechanisation of the $\mathcal{L}_1$ Syntax Using our Programming Discipline

> 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 case of `T` inherits from their instantiation for `T`.

Figure 7.2 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 3–6 in Figure 7.2a) are cases of `L1Exp`. They are fully-baked for $\mathcal{L}_1$'s expression trait (`L1Exp`) because they inherit from `IVar`, `ILam[L1Exp, ...]`, `IApp[L1Exp, ...]`, and `ILet[L1Exp, ...]`, respectively. (See Figure 3.1 for the syntactic categories of $\mathcal{L}_1$.) This is all one needs to do to get the $\mathcal{L}_1$ abstract syntax mechanised. To also get the concrete syntax mechanisation for $\mathcal{L}_1$, these four cases are registered in lines 4–7 of Figure 7.2b. To that end, they get bound to the nested types `Val`, `App`, `Var`, and `Let` of `L1Exp`, respectively. (Compare Figure 7.2 with the top right portion of Figure 7.1.)

Remark 7.1. As a programming discipline, the code in Figure 7.2a is sheerly terse and straightforward to implement. Observe, however, that the code is by far more verbose than the CBMCalc syntax: $\mathsf{family}\ \mathcal{L}_1 = \mathsf{Val} \oplus \mathsf{App} \oplus \mathsf{Var} \oplus \mathsf{Let}$. Alternatively, one can compare Figure 7.3 (for the CBMCalc

mechanisation of the $\mathcal{L}_1$ syntax) against the top right corner of Figure 7.1. As an LDF, Scala was outstanding to allow us to leverage its multiple inheritance and type constraints to emulate CBMCalc. However, the difficulty of observing the above CBMCalc family introduction from behind the heavy Scala wiring still needs to be coped with. That clearly constitutes a topic for future work. □

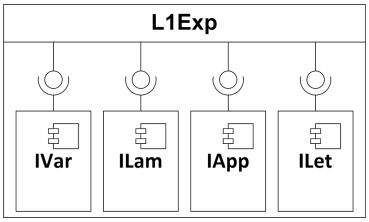


Figure 7.3: Component Diagram for the CBMCalc Mechanisation of $\mathcal{L}_1$ Syntax

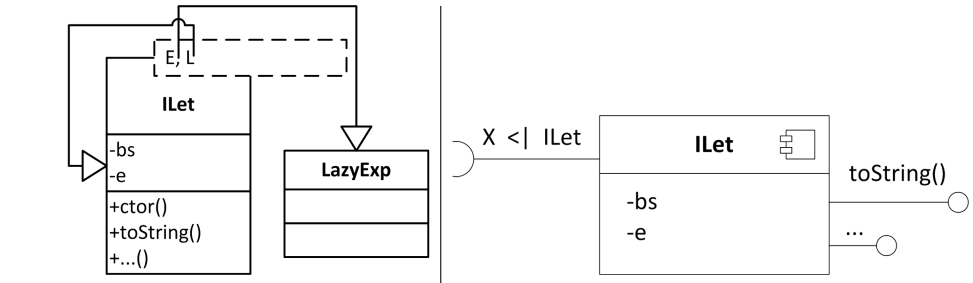
Here are the remaining details of Figure 7.2:

Note that, in line 4 of Figure 7.2b, we bind the nested type `Var` to `l1.Var` (as opposed to `Var` itself). This is because, otherwise, the compiler will mistakenly think that the nested type `Var` is being bound to itself – rather than the case `Var` of `L1Exp` in line 3 of Figure 7.2a – and, fail. Due to the scope resolution, inside `L1Exp`, the unqualified name `Var` refers to the nested type `Var`. The same argument holds for the three other nested types of `L1Exp`.

Note also that, in line 5 of Figure 7.2b, the nested type `Val` (as opposed to a nested type `Lam`) is bound to `l1.Lam`. Whilst the PL implementer is free to choose their favourite names for the cases, the names of nested types are predefined by the ICs. In this very case, the name `Val` is imposed by `ILam`. The name binding is essentially informing the compiler about `Lam` being the value category of `L1Exp`. (Refer back to Figure 3.3.) It follows that `L1Exp` can enjoy from the concrete syntax for λ-expression. See Section 7.4 for more.

7.2.2 IC Technicality

In order for an intermediate class not to be exclusively suitable to a single PL, it is chosen 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 `ILet`, for example:



(a) Class Diagram for `ILet` (b) Component Diagram for `ILet`

Figure 7.4: The Class Diagram of `ILet` vs. its Component Diagram

```
1 class ILet[
2           E <: LazyExp[E],
3           L <: ILet[E, L] with E
4          ](val bs: Map[Idn, E], val e: E) {
5     if(...) //value type == λ-abstractions
6       require(!bs.isEmpty)
7     override def toString() = ...
8 }
```

Lines 5 and 6 above perform a sanity check pertaining to let-expressions. (It is only in $\mathcal{S}_2$ and $\mathcal{S}_3$ – where let-surrounded λ-abstractions are value types – that empty let-bindings are allowed.) Line 7 handles the pretty-printing. The constructor parameters `bs` and `e` (line 4) 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 `E` (line 2). The type parameters of `ILet` (lines 2 and 3) make it invariably available to every syntax in Section 3.1 so long as an `ILet`-derived case `L` is manually presented. (Namely, $\mathcal{L}_0$ is excluded.) Similar type constraints in our codebase selectively determine the appropriate *classes of syntax*.

Using the CBMCalc syntax, the first three lines of the above code can be written as: component $ILet\langle X \lhd ILet\rangle(\cdots)\{\cdots\}$. Despite its relative brevity as a piece of code, the Scala type parameter wiring needed to determine the appropriate syntax is admittedly more verbose. Figure 7.4 depicts the difference diagrammatically. Note, however, that the use of type constraints for similar purposes is not coined by us; neither is that particularly unwelcome in embedding settings. See [Oli09, §7.3] for a chronological discussion on earlier research with similar type treatments.

7.3 ECP Concerns

As announced before, we employ two groups of reusable and composable components that are designed to serve syntax mechanisation and semantics mechanisation of PL families. In this section, we present the former group in terms of the ECP (case) concerns they address. How the latter group addresses ECP (function) concerns is discussed in Section 8.4. These two sections omit presentation for certain ECP concerns though: the ones presentation of our solution accomplishing which neither needs code nor mathematics. Section 8.4.4 provides verbal remarks on those remaining ones. We do not duplicate explanation for EC1 to EC5, which are both case concerns and function concerns.

7.3.1 Implementing EC6

We choose to address EC6 by giving a separate expression trait to every PL. We consider an expression trait the *identity* type of a PL syntax. For example, each PL in Section 3 gets its own syntax type identity. This is realised by each PL mixing and matching its relevant ICs, as depicted in Figure 7.2a for $\mathcal{L}_1$. Similarly, one gets `L0Exp`, `L2Exp`, `S1Exp`, `S2Exp`, and `S3Exp` for $\mathcal{L}_0$, $\mathcal{L}_2$, $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$, respectively.

Technically, two distinctive differences between the code in Figure 7.2a and the usual ADT implementation are:

1. `L1Exp` fixes the F-Bound [CCH+89] of `LazyExp` on itself (line 1).[1] As explained in Section 7.2.1, our programming discipline requires that each PL type identity does the same.

2. In addition to deriving from the PL type identity, each case derives from the corresponding IC (lines 3–6). Amongst other things, this latter derivation fixes the case' syntactic category w.r.t. the appropriate PL type identity. For example, by deriving from `ILam[L1Exp, ...]` in line 4 of Figure 7.2a, `Lam` is fixed to be the `Lam` case of $\mathcal{L}_1$.

Note that, unlike many EP solutions, just because $\mathcal{S}_1$ adds `Seq` to $\mathcal{L}_1$, upon the addition, `S1Exp` will not *replace* the old `L1Exp` definition. Contrast that with LMS in Section 4.2 where `Exp` of `PlainS2` simply replaces the `Exp` of `PlainL1`.

[1] Alike F-Bound treatments are, in fact, popular for solving similar problems in earlier research. Consider Torgersen[Tor04] for EP, Kamina and Tamai [KT08a] for lightweight family polymorphism, and Bruce et al. [BOW98] and Madsen and Ernst [ME10] for virtual classes.

7.3.2 Implementing EC5

Depending on the context, component combination can become invalid in many ways. That is why the statement of ECP refrains from fixing a notion of invalidity. Yet, it expects a solution to facilitate catching invalid combinations statically. We identify four kinds of restrictions that, depending on the conditions, might apply to component combinations. In our codebase, a combination that does not accept either of these restrictions is invalid. As an evidence for our solution addressing EC5, we illustrate how we enforce those restrictions.

Generally, like the lines 4–6 of Figure 7.2a, we implement enforcement via IC type parameters. Of these parameters, one always relates to the PL type identity. The role of this parameter is to specify the corresponding syntactic category of which PL the case being defined is. (Section 9.2.1 contains an example where this piece of knowledge comes handy.) Amongst other things, the remaining type parameters enforce the following, when present:

case identity. Whilst constructing `App` in line 4 above, we pass the type `App` itself as the second type argument of `IApp`. Enforcement of that by `IApp` is alike line 3 of the `ILet` code shown at the beginning of Section 7.2.2. In CBMCalc, one would gain the same enforcement as follows: component $\mathit{IApp}\texttt{<}X \lhd \mathit{IApp}\texttt{>} \cdots \{\cdots\}$.

Programmatically, similar type treatments give the IC access to the exact case type of the respective PL. Of the benefits of this access is ensuring that, for every PL, at most one case corresponds to each syntactic category. By requiring explicit nomination, it also prevents instantiating ICs when the PL does not contain the respective syntactic category. Having the exact case type (as opposed to the IC, which is a bare skeleton), furthermore, enables the IC to accurately clone expressions.

disjointedness. Sometimes it is meaningful to consider certain classes of syntactic categories related whilst only one class member can be present in any given syntax. For instance, both Lam and Val serve lazy PLs as values. Yet, a lazy PL can only contain one of them.[2] We address that by (statically) banning it that a case inherits from both `ILam` and `IVal`. The Scala implementation trick is including the following method in both ICs:

```
final def disjointed[X <: ILam[_, _] with IVal[_, _, _]] = {}
```

This is a special case of Martin's Common Reuse Principle [Mar00]. As

[2] Lazy PLs can, in fact, be partitioned based on their value type: either Lam or Val.

also explained in Remark 6.7, for simplicity reasons, the current design of CBMCalc is not to support such family parameter restrictions.

uncongeniality. It sometimes happens that certain couples (or larger sets) of syntactic categories cannot coexist in a PL. For instance, in our codebase, we have an IC called `ILetVal` that should only be used when the value type of the lazy PL is not `Lam`. This IC (statically) ensures that uncongeniality using the Shapeless type operator `<:!<` in its last parameter pack:

```
(implicit witness: V <:!< ILam[_, _])
```

where `V` is the second type parameter of `ILetVal`, i.e., the type corresponding to values. In short, one can consequently only instantiate `ILetVal` when there is a witness that `V` is not substituted for a case that inherits from `ILam`. This is another case of Martin's Common Reuse Principle [Mar00] that CBMCalc does currently not support.

togetherness. Another probable situation for component collaboration is that a group of them has functional interdependencies. In such a case, a component would enforce the existence of all other components that it needs to collaborate with. For example, `ILetVal` is only meaningful when the value category of a PL syntax is `Val`. In our codebase, `ILetVal` enforces that using its second type parameter below.

```
class ILetVal[
              E <: LazyExp[E],
              V <: IVal[E, V, L] with E,
              L <: ILetVal[E, V, L] with E
             ](...) {...}
```

This is an instance of Martin's Common Closure Principle that CBMCalc supports using the following simple syntax:

$$\textsf{component}\ \mathit{ILetVal}\texttt{<}X \lhd \mathit{Val} \oplus \mathit{ILetVal}\texttt{>}\cdots\{\cdots\}$$

7.4 Concrete Syntax

It remains to further expand on the role of `LazyExp`. In this section, we focus only on the syntactic parts of its role. Chapter 8 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`	$e_1\ \mathsf{seq}\ e_2$
5	`v ^-\ (x)`	$(v)^{-\lambda}[x/\_]$

Note that the code in line 2 embeds a let-surrounded λ-abstraction in the syntax of $\mathcal{S}_2$ and $\mathcal{S}_3$ 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 the syntax of $\mathcal{S}_2$ and $\mathcal{S}_3$ 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. For example, the seq concrete syntax is embedded in terms of `ISeq` below.

```
class ISeq[
          E <: LazyExp[E],
          S <: ISeq[E, S] with E
         ](...){...}
```

As also explained further above, our discipline is that an expression trait `T` must derive from `LazyExp[T]`. (See line 1 in Figure 7.2a for `L1Exp`, for example.) Given that `T`'s cases inherit from it, they also inherit from `LazyExp` by transitivity of inheritance. Note how following our discipline for expression traits makes them distinguishable from their cases. A case type `C`, after all, does not inherit from `LazyExp[C]`; it inherits from `LazyExp[T]`, where `T` is the expression trait of `C`. (See the top row of Figure 7.1.) The particular wiring used below for the type parameter `E` of `LazyExp` enforces the above discipline about expression traits. Chapter 8 contains an example where this discipline comes handy.

```
trait LazyExp[E <: LazyExp[E]] {
  ... /* See Section 8.3 for the contents */ ...
}
```

Interestingly enough, this simple F-Bound enforcement relieves us of the difficulties faced by Kamina and Tamai [KT08a, KT08b] for automatic inference of an expression trait's self type.

A final note on how we were forced to employ nested types to augment normal ADT definitions to expression traits is worth here. Consider line 3 in the above table (on embedding the concrete syntax of let-expressions). The Scala way of doing that is to define `apply` methods for an object `let`. Yet, like partly explained above, the true behaviour of such a method, amongst other things, depends on whether the value category is Lam or Val. Opting for static choice of the correct return type, we were limited to Scala's function

overloading directed by type constraints. Here, we were severly hit whilst trying to cross the boundaries of Scala for the interplay between overload resolution, F-Bounds, and automatic type parameter deduction. Below, we explain how we finally made it to our current solution – the only that we have thus far been able to find.

When it comes to type parameters, Scala's automatic type deduction is limited to the exact static type of a formal parameter (and what is deducible from there on). For example, for the function `f` below

```
def f[A, B](a: A) = {...}
```

Scala will not be able to automatically deduce a correct type for `B`, for it has no evidence in the formal arguments of `f` to work on the basis of. Automatic type deduction will, however, successfully proceed for `A` for the existence of the argument `a`. Hence, the only way to get `B` automatically deduced is to make it dependent on `A`. Our way for that was to make `B` a nested type of `A`:

```
def f[A{type B}](a: A) = {...}
```

That is why, in order to enjoy the embedding of concrete syntax, expression traits have to register their case types: Upon type deduction, expression traits will be queried for their corresponding nested types.

Summary

We now summarise the parts of correspondence between CBMCalc and syntax mechanisation that were seen in this chapter. The following table is the outline.

CBMCalc	codebase	example
component	IC	`ILam`, `ILet`, ...
X	expression trait parameter	usually `E` or `Exp`
$X \lhd \oplus\overline{\gamma}$	case parameters of ICs	`L` in `ILet`, `S` in `ISeq`
family $\Phi = \oplus\overline{\gamma}$	full baking and registration	Figure 7.2
$X.\gamma$	component referral	`IApp[E, A]`

In the above table, the rightmost column contains the CBMCalc elements seen in this chapter; the middle column presents the equivalent elements in our codebase; and, the right column gives examples from our codebase.

Chapter 8

Semantics Components

This chapter presents our semantics mechanisation components and how to use them. We start by introducing the components themselves in Section 8.1. Two ways to use the components to get a complete PL semantics mechanised are, then, discussed in Section 8.2. Next, in Section 8.3 we seize the opportunity to expose more details about `LazyExp`, as promised in the previous section. Finally, Section 8.4 is on our addressing of the remaining ECP concerns using our semantic components.

We begin by a short quantitative assessment of the benefits our codebase has over plain mechanisation. Similar to the case for syntax, it is perfectly possible to program each PL semantics separately. That is a total of 28 rules for the entire semantics presented in Section 3.2, including a great deal of code repetition. Instead, we implement a collection of 15 reusable and executable rules, which can be plugged into a PL semantics mechanisation. Examples are LamRule, VarRule, AppRule, and LetRule that we schematically depicted in the bottom right corner of Figure 7.1.

To assemble executable rules into full semantics mechanisations, our design ships another artefact: `OpSem` is our root operational semantics mechanisation 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 and output to the extent that it allows several semantics mechanisations for a single syntax. In this chapter, we only demonstrate the idea for rules which document the entire semantics evaluation, if successful. However, as illustrated in Section 9.3, 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. We now delve into our pool of executable rules.

8.1 The Semantics Components

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 (see Figure 3.5):

```
1  object HBLetRuleLam {
2    def apply[
3              Exp <: LazyExp[Exp],
4              Val <: ILam[Exp, Val] with Exp,
5              Let <: ILet[Exp, Let] with Exp,
6              OS  <: OpSem[Exp]{type Conf = HBConf[Exp]
7                                 type Node = HBNode[Exp, Val]}
8             ](g: Heap[Exp], lexp: Let)
9              (implicit opsem: OS): HBNode[Exp, Val] = {
10     val (e, bs) = (lexp.e, lexp.bs)
11     val pi = opsem.proofsearch(g ++ bs, e)
12     val (d, z) = (pi.g2, pi.e2)
13     new HBLetNodeLam[Exp, Val, Let](pi, g, lexp, d, z)
14   }
15 }
```

The type parameters `Exp` and `OS` above (lines 3 and 6–7) signify the expression trait and the operational semantics, respectively. However, not every rule in Figure 3.5 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 `Val <: ILam[Exp, Val] with Exp` (line 4) rules out $\mathcal{S}_2$ and $\mathcal{S}_3$. This constraint enforces on `Exp` the availability of a case `Val` that binds to a class derived from `ILam`, i.e., that λ-abstractions is a value type of the syntax (see Figure 3.3). `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 6) on `OS` states that it inputs a pair of heap and expression. Similarly, `type Node = HBNode[Exp]` (line 7) specifies that `OS` outputs a heap-based derivation tree.

```
type HBConf[E <: LazyExp[E]] = (Heap[E], E)
```

(More on `HBNode` soon.) 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 about `HBLetRuleLam`: The constraint `Let <: ILet[Exp, Let] with Exp` (line 5) enforces provision of an `ILet`-derived case for `Exp`. Such a case will be bound under the name `Let` in the body of `HBLetRuleLam`. Furthermore, `lexp` (in line 8) is required to be an instance of this case type. In other words, `lexp` needs to be constructed using the case of `Exp` that corresponds to let-expressions. Recall also that, as seen in Section 7.4, the constraint `Exp <: LazyExp[Exp]` (line 3) ensures that `Exp` is an expression trait. `HBLetNodeLam` inherits from `HBNode` to be the node for let-expressions where λ-abstractions are a value category.

Speaking in terms of CBMCalc, one notices that not all the constraints in lines 3–7 above are family parameterisations of Chapter 6 kind. The first three are, and can easily be expressed under CBMCalc as follows:

$$\mathsf{component}\ \mathit{HBLetRuleLam}\texttt{<}X \lhd \mathit{Val} \oplus \mathit{Let}\texttt{>} \cdots \{\cdots\}$$

where X signifies the syntax family. The story about the constraints in lines 6 and 7, however, is different: Firstly, they are on a different family parameter, i.e., that of semantics. Secondly, they specify a different kind of 'requires' interface. This latter kind is **not** about availability of certain (semantic) components in the (semantic) family; it rather constrains other internals of the family; that is, the type (signature) of a family service, i.e., `proofsearch`. (More on `proofsearch` below.) CBMCalc deliberately avoids multiple family parameters and general family parameter constraints. The consequence is that `HBLetRuleLam` is currently not expressible using CBMCalc.

It remains to consider our `OpSem` trait:

```
1 trait OpSem[Exp <: LazyExp[Exp]] {
2   type Conf
3   type Node
4   def proofsearch(c: Conf): Node
5   ... //See Section 9.3
6 }
```

For each operational semantics, `proofsearch` inputs the initial *configuration* and produces the derivation tree according to the rules of the semantics. The abstract 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). More on nodes in Sections 9.1 and 9.3.

```
1  object l1opsem extends OpSem[L1Exp] {
2    type Conf = HBConf[L1Exp]
3    type Node = HBNode[L1Exp]
4
5    def proofsearch(g: LHeap, e: L1Exp): HBNode[L1Exp] = e match {
6      case l: Lam => HBLamRule[L1Exp, opsem.type](g, l)
7      case a: App => HBAppRuleLam[L1Exp, opsem.type](g, a)
8      case v: Var => HBVarRule[L1Exp, opsem.type](g, v)
9      case l: Let => HBLetRuleLam[L1Exp, opsem.type](g, l)
10   }
11 }
```

Figure 8.1: Implementing the $\mathcal{L}_1$ Operational Semantics in Isolation

8.2 Mechanisation using Executable Rules

In this section, we offer two ways to mechanise a PL semantics using our executable rules. Section 8.2.1 offers the first, in which each PL semantics is mechanised by simply assembling the relevant executable rules – but, without taking other (possibly related) PLs into consideration. The one offered by Section 8.2.2, on the contrary, is designed for reuse of a semantics mechanisation as a whole – as well as reuse of our executable rules. A presentation of how to get both mechanisations to perform expression evaluation comes in Section 8.2.3.

8.2.1 Single Mechanisation

Assuming the availability of our components, the following programming discipline needs to be followed to get a **single** semantics mechanised: 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.

For example, for $\mathcal{L}_1$'s semantics, the method `proofsearch` in Figure 8.1 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 the bottom right corner of Figure 7.1. (In our naming convention, prefix `HB` indicates a *heap-based* system. That includes all PLs of Chapter 3 except $\mathcal{L}_0$.) What comes after the executable rule names in square brackets is to guide Scala's type deduction. A UML

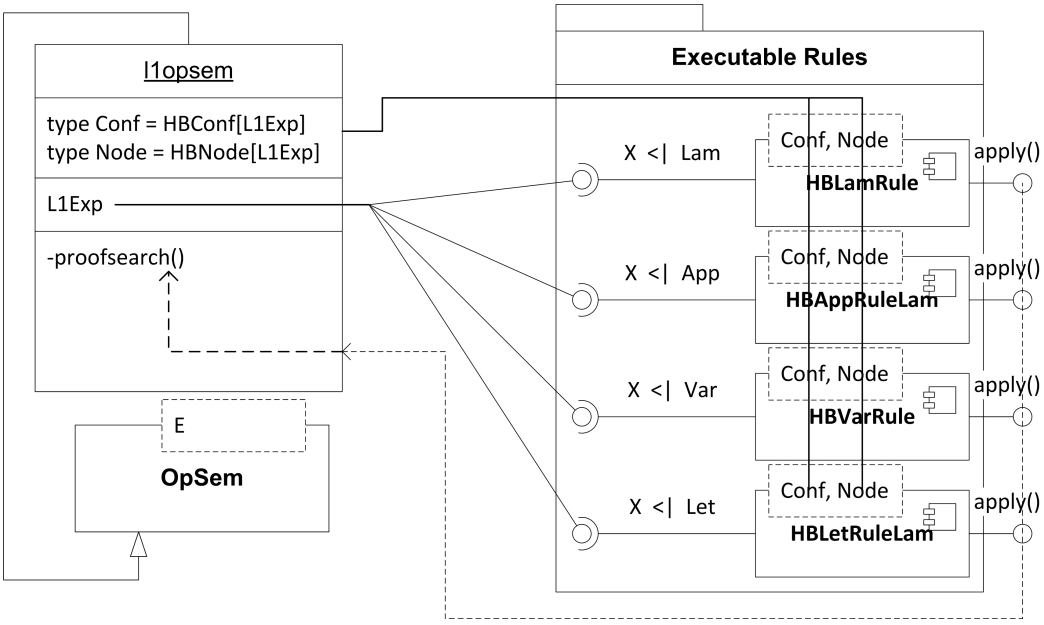


Figure 8.2: UML Diagram for `l1opsem`

summary of how to get `l1opsem` comes in Figure 8.2. The `l1opsem` object instantiates the selected executable rules by substituting `L1Exp` for their family parameters and substituting `HBConf` and `HBNode` for their type parameters. As a result, `proofsearch` is enabled to distribute the evaluation task between the `apply` methods of the selected executable rules.

`HBNode` is the root of our hierarchy for nodes in the heap-based derivation trees. Each node class encapsulates relevant compile time and 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. (C.f. Section 9.1 for more on `HBNode`.)

Remark 8.1. Observe that providing another semantics for the $\mathcal{L}_1$ syntax is as easy as extending `OpSem[L1Exp]` in another object with the new desired `proofsearch` method. In other words, addressing Polymorphic Embedding [HORM08] is an easy side-product of our design. □

```
trait L1OpSem[
              Exp <: LazyExp[Exp],
              Val <: ILam[Exp, Val]      with Exp,
              App <: IApp[Exp, App]      with Exp,
              Var <: IVar                with Exp,
              Let <: ILet[Exp, Val, Let] with Exp
             ] extends OpSem[Exp] {
  def proofsearch(g: Heap[Exp], e: Exp): HBNode[Exp] = e match {
      case v: Val => HBLamRule(g, v)
      case a: App => HBAppRule(g, a)
      case v: Var => HBVarRule(g, v)
      case l: Let => HBLetRule(g, l)
    }
}
```

Figure 8.3: Semantics Mechanisation for α such that $\alpha <_{\mathscr{C}} \mathcal{L}_1$

8.2.2 Extensible Mechanisation

The development in the previous section accommodates reuse of the semantics constructs to the following extent: To mechanise a PL semantics, one needs not to reimplement the semantics rules; a simple assembly of executable rules suffices. Whilst beneficial to reuse, that technique still implies reimplementation in that mechanisation of the next PL semantics requires a brand new assembly of the (reused) executable rules. Observing that consecutive mechanisation cycles often only differ in few rules, the following question naturally arises: To move to the next semantics mechanisation cycle, is it not possible to reuse the available (executable rule) assembly of the current cycle? In this section, we show how to bundle executable rules together to acquire components of coarser granularity that correspond to assemblies of executable rules. We will, then, in Section 8.4 explore the consequences this growth of granularity bears on ensuring legality of reuse.

Consider Figure 8.3. So long as evaluation is concerned, the real work in `L1OpSem` still takes place in its `proofsearch`, which distributes the evaluation based on the cases — nothing new so far. Each executable rule, however, employs type constraints to enforce implementation of its 'requires' interface. Recall how `l1opsem` in Figure 8.1 implements the 'requires' interfaces of the relevant executable rules using a fixed expression trait: `L1Exp`. Opting for reuse of the semantics mechanisation as a complete assembly, `L1OpSem`, on the other hand, refrains from implementing them itself; it instead states that its user needs to implement **all** the required interfaces. (In other words, the user needs to satisfy all the applicable constraints.) As we shall see in the Section 8.4, this trick enables reuse of a mechanisation amongst **all the**

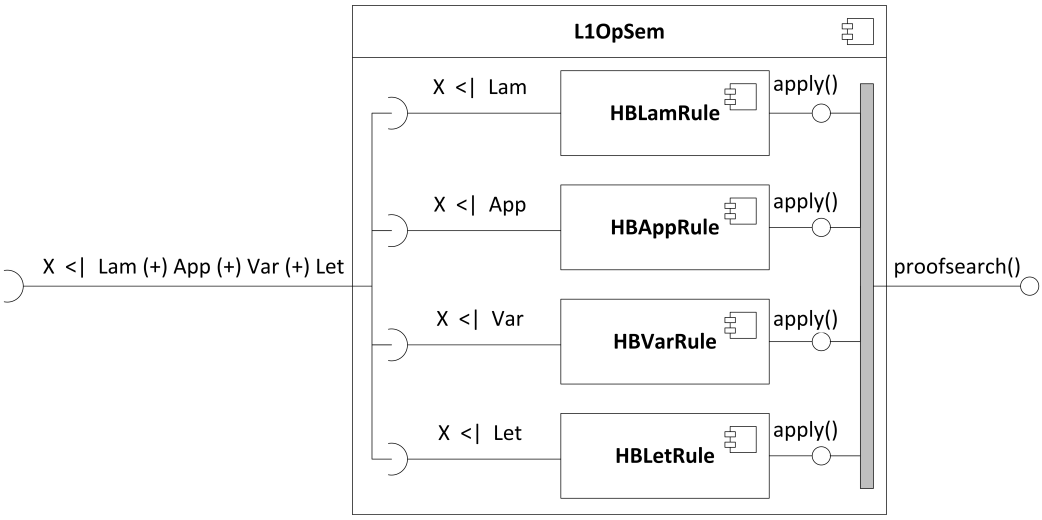


Figure 8.4: Component Diagram for `L1OpSem`

compatible extensions of a PL and bans its use otherwise. In terms of CBMCalc, by bundling `ILam`, `IApp`, `IVar`, and `ILet` together, one gets

$$\text{client } \mathit{L1OpSem}\langle X \lhd \mathit{Lam} \oplus \mathit{App} \oplus \mathit{Var} \oplus \mathit{Let}\rangle\{\cdots\}.$$

Figure 8.4 illustrates the process using a UML Component Diagram. Recall from Remark 5.6 that, in additive component composition, components do not communication directly. The grey bar to the right of the L1OpSem indicates the (indirect) communication media, which, in this case, is the `proofsearch` glue code.

So long as the programming discipline is concerned, the key point here is that, despite enumerating four cases for `Exp`, the type constraints of `L1OpSem` deliberately avoid any further unveiling of `Exp`'s *shape*. In Oliveira's words [Oli09], the constraints are only exposing "*minimal shape information*" here. (This is what inspires minimal shape exposure of CBMCalc.) We demonstrate it in the next subsections that `L1OpSem` is **exclusively** available to the family of PLs that share the shape of $\mathcal{L}_1$.

As a final remark, we would like to invite the reader to also observe the following improvements of `L1OpSem` over `l1opsem`: Firstly, upon calling the relevant executable rules, the former needs not to manually guide the Scala compiler by explicitly stating the instantiation types. (Contrast lines 9–12 in Figure 8.3 with lines 6–9 in Figure 8.1.) Secondly, from the CBSE standpoint, the former is at a higher level of abstraction to the extent that it needs no longer to be concerned about type parameters that do not deal with family

parameterisation. (Contrast the details exposed in Figure 8.4 with those in Figure 8.2.)

8.2.3 Usage and CBMCalc Tests

Having the semantics itself mechanised, it remains to show how to get that employed for evaluation. We only offer a short presentation here. More details can be found in our online repository.

Given that `l1opsem` is already an object, it needs no instantiation. But, as also seen for `HBLetRuleLam` in Section 8.1 (line 9), our executable rules expect an *implicit* operational semantics argument. Scala objects, however, cannot be implicit. Hence, one needs a dummy alias first

```
implicit val os = opsem
```

to get the evaluation of an expression e in a heap g

```
os.proofsearch(g, e)
```

`L1OpSem`, on the other hand, is a trait. It, thus, has to first be instantiated. We mix that with the implicit business.

```
implicit object os extends L1OpSem[LaunchExp, Lam, Var, App, Let]
os.proofsearch(g, e)
```

Both `proofsearch` calls above correspond to the following CBMCalc test:

$$\begin{array}{l}\mathsf{family}\ \Phi_{\mathcal{L}_1} = \mathit{Lam} \oplus \mathit{App} \oplus \mathit{Var} \oplus \mathit{Let}\,;\\ \mathsf{val}\ e = \ldots\,;\\ \mathsf{val}\ g = \ldots\,;\\ \mathit{L1OpSem}\texttt{<}\Phi_{\mathcal{L}_1}\texttt{>}.\mathit{proofsearch}(g, e)\,;\end{array}$$

The reader is invited to type check and run the above test using the CBMCalc static and dynamic semantics in Sections 6.3 and 6.4.

8.3 More `LazyExp` Details

The attentive reader might have already noticed it that our executable rules do not have information about *how* to construct instances of the ADT cases they work with. Yet, they build new expressions out of the ones that they receive as arguments or as results of intermediate computations. One, then, may wonder how this is possible. The short answer is that, in fact, there are factory methods involved that we disguise using expression traits and that are passed around at the appropriate points using Scala's implicit mechanism. In this section, we shed more light on that and fulfil our promise in Section 7.4 for demonstrating more internals of `LazyExp`.

```
1  trait LazyExp[E <: LazyExp[E]] {
2    this: E =>
3    ...
4    type App <: IApp[E, App] with E
5    def apply(x: Idn): E = incarnator(new IApp[E, App](this, x))
6
7    def incarnator: ICBase => E
8  }
```

Line 2 above indicates that the self-type inside every expression trait `E` is `E` itself (as opposed to `LazyExp[E]`). That helps us to enforce more type correctness. Consider line 4, for example, which enforces that, for every expression trait `E`, the nested type `E#App` has to be bound to the `App` case of `E` as opposed to that of any other type `E'`. As a result, a compile error will be emitted if, for instance, `App` of $\mathcal{L}_1$ mistakenly chooses to inherit from `IApp[S1Exp]`. This becomes especially handy in the legislation of the `IApp` creation at line 5. Without that, the type of `this` inside the `LazyExp` body would have been different from `E` and one could have not acquired an `App` instance of `E`.

The method `apply` in line 5 above is how we embed the function application concrete syntax. (See row 1 in the concrete syntax table at Section 7.4.) The body of this method introduces employment of the factory method `incarnator`. For an expression trait `E`, the abstract method `incarnator` takes a half-baked object and returns a fully-baked object of the same syntactic category, if that syntactic category is of those of `E`. (`ICBase` is the root of our ICs.)

That is why, in line 5 above, `incarnator` is left abstract: Based on the ICs it mixes in, every expression trait has to implement its **own** factory method. Here is the point to relate back to CBMCALC's policy of allowing substitution of **equivalent** components instead of the exact same requested ones. Recall that Remark 6.9 considers two requisites for that policy to be: constructibility with the same syntax and provision of the same interface. Our Scala codebase implements the latter by simply enforcing upper bounds (`<:`) on ADT cases – as opposed to exact type bounds (`=`). (Examples are line 4 above, lines 3–6 in Figure 8.1, lines 5 and 6 in `HBLetRuleLam`, and line 3 in `ILet`.) On the other hand, `incarnator` is how our Scala codebase implements the former. Our executable rules[1] only instantiate ICs (of supposedly right types and contents). It is the duty of `incarnator` to transform the instantiated ICs to fully-baked expressions of the right PL syntax.

Note that because this method is of the properties of an expression trait (as opposed to an instance of that type), it ought to be static. However,

[1] as well as other users of our syntax mechanisation material; see Chapter 9 for analysis

```
1  trait S1OpSem[
2                Exp <: LazyExp[Exp],
3                Val <: ILam[Exp, Val]      with Exp,
4                App <: IApp[Exp, App]      with Exp,
5                Var <: IVar                with Exp,
6                Let <: ILet[Exp, Val, Let] with Exp,
7                Seq <: ISeq[Exp, Seq]      with Exp
8              ] extends L1OpSem[Exp, Val, Var, App, Let] {
9    override def proofsearch(g: Heap[Exp], e: Exp): HBNode[Exp] = e match {
10     case s: Seq => HBSeqRule(g, s)
11     case _      => super.proofsearch(g, e)
12   }
13 }
```

Figure 8.5: Extending `L1OpSem` to Semantics Mechanisation for α such that $\alpha <_{\mathscr{C}} \mathcal{S}_1$

Scala's current support for static methods is unfortunately not powerful enough to accommodate that. We have to resort to ordinary methods thus. To demonstrate the resulting inappropriateness, we only offer a single example here. Our codebase contains many similar spots. In the body of `HBAppRuleVal`, we employ the `v ^-\ (x)` syntax to produce the intermediate expression needed by $(\textbf{app})_{\mathcal{S}_2,\mathcal{S}_3}$.[2] Here is a snippet of how the method `^-\` is defined:

```
e.incarnator(e.subst(y, be))
```

Observe how nonsensical it is to call the method `incarnator` above on `e`.

8.4 ECP Concerns

Out of the remaining EC concerns, we first explain in Sections 8.4.1, 8.4.2, and 8.4.3 how we address EC7, EC8, and EC1, respectively. Then, in Section 8.4.4, we end this chapter by a discussion on the EC concerns for support of which we do not offer dedicated demonstration.

8.4.1 Implementing EC7

Using the discussion of Section 8.2.2, it is not hard to figure out that `Exp` in Figure 8.5 needs to nominate its cases that correspond to `ILam`, `IApp`, `IVar`, `ILet`, and `ISeq` (lines 2–7). In other words, every case of `Exp` in `S1OpSem` is

[2] See row 5 in our concrete syntax table at Section 7.4 for the piece of concrete syntax. See Figure 3.4 for $(\textbf{app})_{\mathcal{S}_2,\mathcal{S}_3}$.

a subtype of case of `Exp` in `L1OpSem`. Thus, line 11 in Figure 8.5 is how we implement EC7: An expression of the former type (`e`) can be used where an expression of the latter type is expected (`super.eval(..., e)`). That is, by the Liskov Substitution Principle [Lis87], the former type acts like a subtype of the latter.

Mathematical Interpretation

We promised in Section 6.1 to reflect our mathematical development in our code. The reader is now ready enough for that: Let $\mathfrak{T}$ be the set of all case definitions in Scala. Fix the set $\mathfrak{S} = \{\texttt{ILam}, \texttt{IApp}, \texttt{IVar}, \texttt{ILet}, \texttt{ISeq}, \texttt{IVal}\}$ of all the ICs. We say τ_1 and τ_2 **are together in a syntactic category** ($\tau_1 \overset{s}{\equiv} \tau_2$) when $\exists s \in \mathfrak{S}.\ (\tau_1 \ \texttt{<:}\ \ s) \wedge (\tau_2 \ \texttt{<:}\ \ s)$.

Notice next that `L1Exp` $\overset{\text{def}}{=}$ `ILam` $\oplus$ `IApp` $\oplus$ `IVar` $\oplus$ `ILet` whilst `S1Exp` $\overset{\text{def}}{=}$ `ILam` $\oplus$ `IApp` $\oplus$ `IVar` $\oplus$ `ILet` $\oplus$ `ISeq`. (See Section 6.1.) Using a similar argument to the one presented in Example 6.3, one concludes `S1Exp` $<_{\mathscr{C}}$ `L1Exp`. On the other hand, it follows for the type parameter `Exp` of `L1OpSem` and `S1OpSem`, respectively, that `Exp` $<_{\mathscr{C}}$ `L1Exp` and `Exp` $<_{\mathscr{C}}$ `S1Exp`. Hence, the most important observation is that Proposition 6.4 justifies the rightfulness of extending `L1OpSem` to `S1OpSem`. This is done by implying that it also holds for `Exp` of `S1OpSem` that `Exp` $<_{\mathscr{C}}$ `L1Exp`.

Our referral in Remark 6.6 to the above approach as being *existential* is now more clear. The CBMCalc version of Figure 8.5 is

```
client S1OpSem<X ◁ Lam ⊕ App ⊕ Var ⊕ Let ⊕ Seq> ◁
    L1OpSem<X as Lam ⊕ App ⊕ Var ⊕ Let> {
  ... proofsearch(g, e) {
    ... super.proofsearch(g, e) ...
  }
}
```

where, in order to derive from *L1OpSem* (Section 8.2.2), *S1OpSem* has to manually *construct* its component combination in the second line and direct the instantiation of *L1OpSem*. Taking $\overset{s}{\equiv}$ to be the identity relation, one easily observes that the above derivation of *S1OpSem* from *L1OpSem* is well-formed due to (CE-PFam) in Figure 6.8. Recall that validity of projecting the required components of *S1OpSem* to those passed to *L1OpSem* is a **same-indexed** check. The UML diagram for the above CBMCalc snippet is illustrated in Figure 8.6.

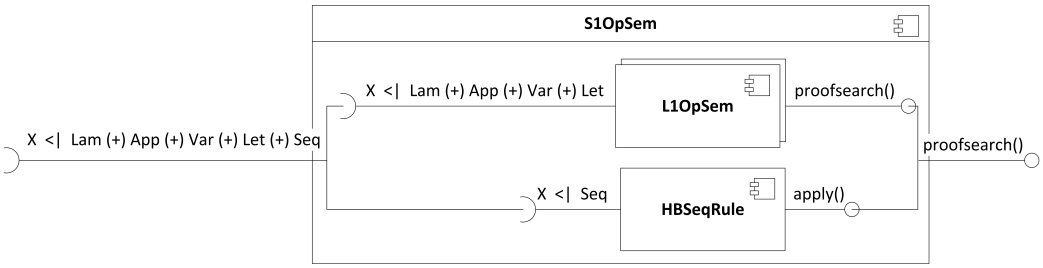


Figure 8.6: Component Diagram for `S1OpSem`

8.4.2 Implementing EC8

The code below is an attempt to (mistakenly) extend `L1OpSem` to a mechanisation for $\mathcal{S}_2$ (and all its compatible extensions). This is done in the same fashion as `S1OpSem` in Figure 8.5 by attempting to derive `BadS2OpSem` from `L1OpSem` and passing the right type parameters of the former as the corresponding type argument in the latter (line 8).

```
1  trait BadS2OpSem[
2                   Exp <: LazyExp[Exp],
3                   Val <: IVal[Exp, Val, Let] with Exp,
4                   Var <: IVar                with Exp,
5                   App <: IApp[Exp, App]      with Exp,
6                   Let <: ILet[Exp, Val, Let] with Exp,
7                   Seq <: ISeq[Exp, Seq]      with Exp
8                 ] extends L1OpSem[Exp, Val, Var, App, Let] {
9    override def eval(g: Heap[Exp], e: Exp): HBNode[Exp] = e match {
10     case v: Val => HBValRule(g, v)
11     case l: Let => HBLetRule(g, l)
12     case s: Seq => HBSeqRule(g, s)
13     case _      => super.eval(g, e)
14   }
15 }
```

Thanks to our type system resort, the compiler rejects the above code by issuing the following error (Figure 8.7 shows the Eclipse the screen shot):

```
type arguments [Exp,Val,Var,App,Let] do not conform to
trait L1OpSem's type parameter bounds
[Exp <: LazyExp[Exp],Val <: ILam[Exp,Val] with Exp,
 Var <: IVar with Exp,App <: IApp[Exp,App] with Exp,
 Let <: ILet[Exp,Val,Let] with Exp]
```

Note that $\texttt{S2Exp} \stackrel{\text{def}}{=} \texttt{IVal} \oplus \texttt{IApp} \oplus \texttt{IVar} \oplus \texttt{ILet} \oplus \texttt{ISeq}$. That is, for the type parameter `Exp` of `BadS2OpSem`, it holds that $\texttt{Exp} <_{\mathscr{C}} \texttt{S2Exp}$. However, using a similar argument to Example 6.3, one realises that $\texttt{S2Exp} \not<_{\mathscr{C}} \texttt{L1Exp}$. It

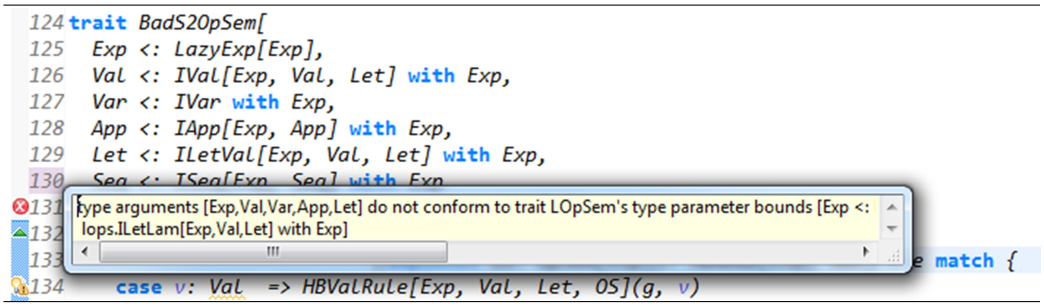


Figure 8.7: Error Message for Syntactically Incompatible Extension

follows by Proposition 6.5 for the `Exp` of `BadS2OpSem` that `Exp` $\not<_{\mathscr{C}}$ `L1Exp`. This justifies the above error message. The brownie we collect over this exercise is that, unlike in the codes of Section 4.2, the missing `App` pattern match in `BadS2OpSem` is not harmful because the code fails to compile in the first place.

This failure of compilation is easier to explain in terms of CBMCalc, because, according to (CE-PFam)

$$\mathsf{client}\ \mathit{BadS2OpSem}\texttt{<}X \lhd \mathit{Val} \oplus \cdots \texttt{>} \lhd \mathit{L1OpSem}\texttt{<}X\ \mathsf{as}\ \mathit{Val} \oplus \cdots \texttt{>}\ \{\cdots\}$$

is not well-formed because $\mathit{Val} \not<: \mathit{Lam}$.

8.4.3 Implementing EC1

With the mechanisation of PLs using our components, the possibility of packing independently developed components together – and addressing EC1 – is clear. We now manifest more of the strength of our solution using a slightly less trivial situation, i.e., composition of (semantics) mechanisation without direct use of components. To that end, let `S3OpSem` be the mechanisation of $\mathcal{S}_3$ semantics as an **extension** to a predecessor of its. Suppose, next, that we are about to mechanise a PL called $\mathcal{S}_4$ that adds the strict application of $\mathcal{S}_3$ to $\mathcal{S}_2$. The class `S4OpSem` below illustrates how to obtain the semantics mechanisation of $\mathcal{S}_4$ for free by simply composing those of $\mathcal{S}_3$ and $\mathcal{S}_2$ (lines 9 and 10). Note that this is in fact stronger than Extension Composition of Nystrom et al. [NQM06] (or even EC1) in that the type parameters (lines 2–8) guarantee soundness of composition w.r.t. syntactic categories.

```
trait S4OpSem[
               Exp <: LazyExp[Exp],
               Val <: IVal[Exp, Val, Let] with Exp,
               Var <: IVar                with Exp,
               App <: IApp[Exp, App]      with Exp,
               Let <: ILet[Exp, Val, Let] with Exp,
               Squ <: ISeq[Exp, Squ]      with Exp,
               Srp <: ISrp[Exp, Srp]      with Exp
             ] extends S2OpSem[Exp, Val, Var, App, Let, Squ] with
                       S3OpSem[Exp, Val, Var, App, Let, Srp]
```

(The `ISrp` above is our IC for $e\#x$. See Figure 3.1 for the syntax.)

8.4.4 Discussion

The fact that we support EC2 (backward compatibility) is obvious: Addition of new components has no effect on the existing code; neither does mechanising a new PL syntax; nor does extending a PL semantics mechanisation. Scalability (i.e., EC3) of our solution is also clear: Addition of new cases or functions amounts to employing the respective components.[3] (More on this in Sections 9.3. See also 4.3.) We also get completeness of composition (i.e., EC4) for free in any language that supports multiple inheritance.

A note on EC5 is worth mentioning here: Just like their syntax counterparts (namely, ICs), our executable rules enjoy enforcement of their 'requires' interface using type parameters. (See Section 8.2.) That is, we also provide EC5 for semantics.

We would like to invite the reader to observe that our approach for semantics mechanisation, in fact, simulates Polymorphic Variants in Scala. We employ type constraints and multiple inheritance for that purpose. The same can be done in any host language that supports those two features.

Finally, the few parts of correspondence between CBMCalc and our codebase that are new to this chapter are listed below:

CBMCalc	codebase	example
client	semantics mechanisation	`S1OpSem` and `BadS2OpSem`
client C<$X \lhd \oplus\overline{\gamma}$> $\lhd \cdots$	extension to semantics mechanisation	Section 8.4.1 and Section 8.4.2
$I.m_C(\overline{y})$	expression evaluation	`os.proofsearch(g, e)`

[3] One would of course need to implement the missing components, if any. Yet, that is regarded as *out of consideration* in the context of Component-Based Software Engineering. See [Som11, Chapter 17] and [Pre09, Chapter 10].

Chapter 9

Analysis and Transformation

To show that our approach is useful to full-cycle mechanisation, it remains to show how it suits mechanisation of analysis and transformation. In this chapter, we demonstrate various techniques in our approach for that purpose. Section 9.1 focuses on mechanisation of analyses (and transformations) that are applicable only once the evaluation is done. Next, in Section 9.2 we present Feature-Oriented Programming [Pre97, BSR03, ABKS13] techniques for a pair of compositional analysis and transformation that can be used both before and after evaluation. Finally, in Section 9.3, we demonstrate how our approach also suits non-compositional transformations.

9.1 A Posteriori Analysis

One can analyse many aspects of a PL mechanisation using a variety of techniques. This section is specially devoted on *a posteriori analyses*: the PL analyses that are performed **after** the evaluation is done. In particular, we implement five analyses (and one transformation) that all work on the derivation trees produced using the semantics rules of Section 3.2. In this section, we only present three analyses out of the five. We believe the presentation is enough for understanding the approach. The interested reader is encouraged to find more examples in our codebase.

Similar to the case for syntax and semantics, the aim of this section is to foster analysis reuse. That is reuse of the code implemented over previous mechanisation cycles – but, only when they are still conceptually applicable. Our development to that end is based on the following 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.

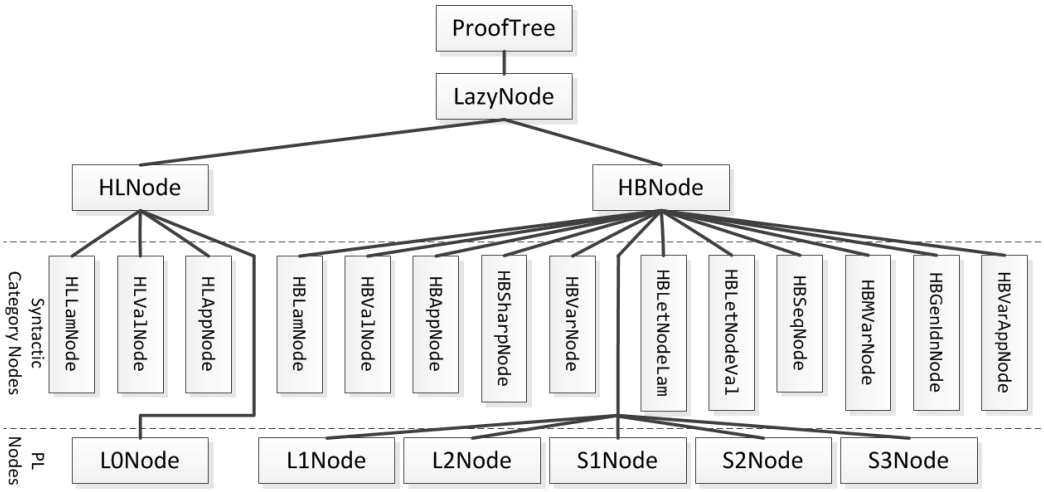


Figure 9.1: The Hierarchy of our Derivation Trees

Fact 2. Code that constrains 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 of analyses."

Our hierarchy of derivation trees is depicted in Figure 9.1. Nodes in this figure are presented using rectangles. Lines show inheritance: When connected, the lower node derives from the upper one.

The hierarchy is rooted in `ProofTree`, which 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. Then, comes `LazyNode`, which in addition to `ProofTree` only knows that it is responsible for holding the nodes of a

lazy PL. `HBNode` and `HLNode` extend `LazyNode` 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. Each rule in Section 3.2 has a correspondent here. These latter nodes know the syntactic category of their e in the above scheme. They also know that e can be an expression of every relevant PL. For instance, `HBVarNode` that corresponds to $\textbf{(var)}_{\mathcal{L}_1,\mathcal{L}_2,\mathcal{S}_1,\mathcal{S}_2,\mathcal{S}_3}$ knows that it works on `Var`. It also knows that the expression it works on can be an expression of the following PLs: $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$. (See Figure 3.4.) 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. Yet, it knows not the exact syntactic category of the expression it works on.

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 Section 8.1 all have `HBNode` or `HLNode` return types. (Nevertheless, it is trivial to configure executable rules otherwise and still enjoy them as reusable components for semantics mechanisation. See `S2NPOpSem` in Section 9.3 for an example.) 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 section.

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). In line 2, `/:` is Scala's left-folding; 0 is the initial value; and, `(_ + rulecount(_))}` is the binary function. (Scala uses `_` for placeholders. As such, the two occurrences of `_` in line 2 a placeholders for the two parameters of the binary function.) An occasion where

this counting might be useful is comparing the cost of predesignated computations across different semantics which are known to be observationally equivalent.

Our second example is on the analyses in Definition 9.1, which play a central role in the observational equivalence theorems on $\mathcal{S}_2$ [Hae10]:

Definition 9.1. *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:[1]

```
1  object diff {
2    def apply[E <: LazyExp[E]](pi: HBNode[E]): 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[
7              E <: LazyExp[E],
8              V <: IVar with E,
9              OS <: OpSem[E]{type Conf = HBConf[E], type Node = HBNode[E]}
10           ](g: Heap[E], x: Idn) : Boolean = //x is atomic in g when...
11     diff(g <::> x) == Set(x)//... diff(pi_x) == {x},
12 }//where pi_x = g <::> x.
```

Notice how, like our ICs and executable rules, `atomic` characterises the syntax and semantics it is applicable to using the constraints on the type parameters (lines 7–9). Consequently, it remains applicable upon extensions of mechanisation so long as the characteristics remain intact. That is, `atomic` works for $\{\alpha \mid \alpha <_{\mathscr{C}} \mathcal{L}_1 \vee \alpha <_{\mathscr{C}} \mathcal{S}_2\}$. (See Section 6.1.)

Thanks to our multi-levelling, in the implementation of `diff`, types are all correctly identified by the compiler. That is, the compiler can statically verify it that the formal parameter `pi` (line 2) has fields `g` and `d` that are heaps (corresponding to Γ and Δ in the scheme $\Gamma : \_ \Downarrow_\Pi \Delta : \_$.) We would have not had such a pleasure, had we implemented it on `ProofTree`, which is oblivious of the types 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] `g <::> e` abbreviates `os.proofsearch(g, e)` when `os` is an implicit in scope.

```
1  object EvalList {
2    def apply[E <: LazyExp[E], V <: IVar with E]
3      (hbn: HBNode[E]): List[Idn] = hbn match {
4        case HBVarNode(pi, g, xvar, d, _) => {//(var)_{L1,L2,S1,S2,S3} in Fig. 3.4:
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. It proceeds by checking whether the provided node is one that corresponds to the $(\mathbf{var})_{\mathcal{L}_1,\mathcal{L}_2,\mathcal{S}_1,\mathcal{S}_2,\mathcal{S}_3}$ rule (line 4). In that case, when the binding of the variable is manipulated (line 7), `EvalList` takes a note of the variable. Otherwise, it simply accumulates the evaluated variables of the node's premisses. (Scala uses the notation `:::` (line 7) for appending an element to a list.)

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. That is, the CBMCalc equivalent is the following snippet:

$$\mathsf{client}\ \mathit{EvalList}\langle X \lhd \mathit{Var}\rangle\{\cdots\}$$

We enforced that by making `EvalList` applicable to any such tree through placing the type constraints at line 2. It is exactly this use of 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 or even resort to runtime casting to calm the type system.

Here is a short demonstration on how the above single implementation of `EvalList` remains applicable upon moving from $\mathcal{L}_1$ to $\mathcal{S}_2$. For an expression trait `E`, the definition

```
val g = Heap("id" -> \[E]("t")("t"),
             "y"  -> \[E]("t1", "t2")("t1")("id")("x"),
             "x"  -> \[E]("t1", "t2")("t2")("id"))
```

sets `g` to represent the heap $\Gamma = \{id \mapsto \lambda t.t, y \mapsto (\lambda t_1 t_2.t_1)\ id\ x, x \mapsto (\lambda t_1 t_2.t_2)\ id\}$. (Refer back to Section 7.4 for our concrete syntax mechanisation.) Then,

```
EvalList(g <::> "y")
```

produces `List(y)` for $\mathcal{L}_1$, whilst

```
EvalList(g <::> ("x" seq "y"))
```
produces `List(x, y)` for $\mathcal{S}_2$.

9.2 Expression Families Problem

In his seminal work on EFP [Oli09], Oliveira provides two self-contained examples that obviate the necessity of solving EFP and the strength of MVCs: Equality Tests and Narrowing. We show in this section that, modulo aggregate subtyping, ECP is a variation of EFP. (See Section 4.3 for the comparison on dealing with aggregate subtyping.) That we do by offering solutions to Equality Tests and Narrowing in Sections 9.2.1 and 9.2.2, respectively. In Section 8.2, we chose to build on top of LMS. Whilst the same is possible here, we choose to do otherwise for reasons that will become clear in Section 9.2.3. One can use similar arguments to those in Sections 7.3 and 8.4 to observe that the solution in this section also solves ECP.

9.2.1 Equality

The purpose of the Equality Test exercise is to provide a statically safe solution for *multiple dispatching* that is also extensible. The driving example is structural equality between expressions. Like that of Oliveira, we offer a solution that simulates multi-methods [CL95, CLCM00] in Scala.

The key idea in our solution is what we call *integration of a decentralised pattern matching*. That is, instead of centralising the pattern matching in a single place, we distribute it amongst components that correspond to the cases of structural equality test. Let us call these new components the *equality component*s. Each equality component takes the exclusive responsibility of its own part of structural equality test. This is done using a method called `this_case`. For example, `EqVar` and `EqApp` below handle equality test for when the two expressions at hand are Vars and Apps, respectively:

```
1  trait EqVar[E <: LazyExp[E]] extends EqBase[E] {
2    override def this_case(e1: E, e2: E): Boolean = (e1, e2) match {
3      case (v1: IVar, v2: IVar) if v1.name == v2.name => true
4      case _ => super.this_case(e1, e2)
5    }
6  }
7  trait EqApp[E <: LazyExp[E]] extends EqBase[E] {
8    override def this_case(e1: E, e2: E): Boolean = (e1, e2) match {
9      case (a1: IApp[_, _], a2: IApp[_, _]) if a1.x == a2.x =>
10       areeq(a1.e.asInstanceOf[E], a2.e.asInstanceOf[E])
11     case _ => super.this_case(e1, e2)
```

```
12    }
13  }
```

Note that line 3 above is the only non-trivial equality case for `EqVar`. Likewise are lines 9 and 10 for `EqApp`. Hence, lines 4 and 11 forward the other cases of equality test to the **next level above** in the stack of mixed-in equality components. For example, the mixed traits in `eqer` below

```
object eqer extends EqBase[S1Exp] with EqVar[S1Exp] with EqLam[S1Exp] with
                    EqApp[S1Exp]  with EqLet[S1Exp] with EqSeq[S1Exp] with
                    FinalEq[S1Exp]
import eqer.areeq
```

enable us to perform the equality tests like

```
println("sel1st == id? " + areeq(sel1st, id))
```

to check whether, under $\mathcal{S}_1$, the following two expressions are the same: $\lambda x.x$ and $\lambda xy.x$. (Recall from Section 7.3.1 that `S1Exp` is the expression trait of $\mathcal{S}_1$.)

The last two pieces of this puzzle are `EqBase` and `FinalEq`.

```
1  trait EqBase[E <: LazyExp[E]] {
2    def this_case(e1: E, e2: E): Boolean = false
3    def areeq(e1: E, e2: E): Boolean
4  }
5  trait FinalEq[E <: LazyExp[E]] extends EqBase[E] {
6    override def areeq(e1: E, e2: E): Boolean = super.this_case(e1, e2)
7  }
```

The role of `EqBase` is two-fold: its `this_case` method implements the default equality test case (to false); and, being the base class of all the equality components, it outlaws use of equality components until a concrete `areeq` is mixed in. (Note that the method `areeq` in line 3 above is abstract.) Finally, a concrete implementation of `areeq` is provided by `FinalEq`, which simply forwards the overall task to the next level above in the mixin stack.

With this wiring, a call like `areeq(sel1st, id)` will start the equality test from `FinalEq`; go up to `EqSeq`; and, keep climbing the ladder up until it finds the right handler, if any. In the mean time, if it needs to test the equality of sub-expressions, `areeq` (which reunions the distributed pattern matching) will be called. An example of this call is line 10 in `EqApp`.

Note that the two casts `e.asInstanceOf[E]` in line 10 (in `EqApp`) are not intrinsic to our solution; they are a consequence of Scala's choice for type erasure in pattern matching.[2] In other words, in an unerased host PL, we could simply write `case (a1: IApp[E, _], a2: IApp[E, _])` in line 9 of `EqApp` and discard the cast. It is also worth noting that these casts are guaranteed not to fail at runtime; `e1` and `e2` are already of type `E`, and so

[2]It is not exclusively our solution, however, that is challenged by Scala's type erasure. Madsen and Ernst [ME10, §4] report similar issues in their recent development on Virtual Classes.

are `a1` and a2. The first type parameter of `IApp` ensures then that `a1.e` and `a2.e` are of type `E`. Refer back to the paragraph above case identity in Section 7.3.2.

Our solution caters for extensibility by leaving open the possibility of mixing-in new equality components without the need to touch the existing equality cases. Note that, in this very case, the structural exercise happened to come with a default case. As noticed first by Zenger and Odersky [ZO01] and several others afterwards, a default is not necessarily available. Section 9.2.2 contains an example where our solution works in the absence of default cases.

9.2.2 Conversion and Narrowing

Following Oliveira [Oli09], we say an expression is **narrow**ed when all its ADT cases of a given group are cancelled into other case combinations that are deemed to be equivalent. It is common in the PL design community to provide extensions to a core PL such that the extension programs would then be narrowed to the core (for evaluation and the like). For example, GPH [THLP98] and Utrecht Haskell [DFS09] are both developed like that. Oliveira shows how his MVCs can be leveraged in favour of correctness for narrowing as a static guarantee that the result of this process will not contain instances from the unwanted ADT cases; it will instead contain other case combinations that are deemed equivalent.

In this section, we generalise the narrowing exercise to conversion from one subject PL to another. In particular, we discuss expression conversion from α to α' such that $\alpha <_{\mathscr{C}} \mathcal{S}_2$ and $\alpha' <_{\mathscr{C}} \mathcal{S}_3$. Our conversion provides similar guarantees to those of Oliveira for narrowing. Hence, applying this conversion from $\mathcal{S}_4$ to itself will result in narrowing for it. Recall from Section 8.4.3 that $\mathcal{S}_4$ is a compatible extension to both $\mathcal{S}_2$ and $\mathcal{S}_3$. (More on `Seq2Srp` later on.)

```
val narrowizer = new Seq2Srp[S4Exp, S4Exp] () {}
```

Besides, we extend the expression conversion (in the obvious pointwise manner) to heaps and derivation trees. We show that this extension enables us to test a certain commutativity property between $\mathcal{S}_2$ and $\mathcal{S}_3$. (See Equation 9.6.)

Being based on a single subject PL, narrowing is a single dispatching problem. As will be discussed later for `SeqConversion`, however, with the level of correctness that it guarantees, our solution for the conversion exercise also addresses type-safe multiple dispatching – even in the absence of a default case. (Note that there is no default case in the Conversion exercise.)

Our conversion solution too is based on integration of a decentralised pattern matching. We do so using yet another group of components: *conver-*

```scala
trait SeqConversion[
                    Exp1 <: LazyExp[Exp1],
                    Seq1 <: ISeq[Exp1, Seq1]       with Exp1,
                    Exp2 <: LazyExp[Exp2],
                    Val2 <: IVal[Exp2, Val2, Let2] with Exp2,
                    Var2 <: IVar                   with Exp2,
                    Let2 <: ILet[Exp2, Val2, Let2] with Exp2,
                    Srp2 <: ISrp[Exp2, Srp2]       with Exp2
                   ] extends Conversion[Exp1, Exp2] {
  abstract override protected def this_case(e1: Exp1) = e1 match {
    case s1: ISeq[_, _] => {... /* See Figure 9.3 */ ...}
    case _ => super.this_case(e1)
  }
}
```

Figure 9.2: Mechanisation Skeleton of Equation 9.5

sion components. Our conversion components are like equality components of Section 9.2.1 except that they concern conversion as opposed to equality. However, as we will see, conversion components are more restrictive in their type parameters. They provide stronger static safety in the following sense: Mixing-in an equality component for an ADT lacking the respective case will not fail at runtime – the pattern match will simply never succeed; no compile error will be emitted either. On the contrary, mixing a conversion component for ADTs missing the respective case will be rejected statically. The CBMCalc way of expressing the reason is that conversion components enjoy family parameterisation but equality components do not.

$$
\begin{aligned}
[\![x]\!] &= x && (9.1)\\
[\![\lambda x.e]\!] &= \lambda x.[\![e]\!] && (9.2)\\
[\![e\ x]\!] &= [\![e]\!]\ x && (9.3)\\
[\![\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e]\!] &= \mathsf{let}\ \{x_i{=}[\![e_i]\!]\}_{i=1}^{n}\ \mathsf{in}\ [\![e]\!] && (9.4)\\
[\![e_1\ \mathsf{seq}\ e_2]\!] &= \mathsf{let}\ \{x_1 \mapsto [\![e_1]\!],\ x_2 \mapsto [\![e_2]\!]\}\ \mathsf{in} \\
&\qquad ((\lambda xy.y)\#x_1)\ x_2 && (9.5)
\end{aligned}
$$

(Refer back to Section 3.1 for more on the strict function application $e\#x$.) The equations above show the different cases of the conversion function $[\![.]\!]$. So long as the work being done is concerned, the only non-trivial case is Equation 9.5 in which x_1, x_2, x, and y are fresh. Likewise, the only non-trivial conversion component is `SeqConversion`, which is responsible for Equation 9.5.

`SeqConversion` (Figure 9.2) differs from our equality components in a number of ways. Firstly, it is parameterised over both the source and the

destination identity types (`Exp1` and `Exp2` in lines 2 and 3, respectively). This is the multiple dispatching nature of conversion we formerly spoke about. Secondly, it demands the availability of certain cases for the ADTs (line 3 for `Exp1` and lines 5–8 for `Exp2`). Extending CBMCalc to allow multiple family parameters, the following CBMCalc snippet would summarise the above two points:

$$\textsf{component}\ \mathit{SeqConversion}\texttt{<}X_1 \lhd \mathit{Seq}, X_2 \lhd \mathit{Val} \oplus \mathit{Var} \oplus \mathit{Let} \oplus \mathit{Srp}\texttt{>}\{\cdots\}$$

Each conversion component places type constraints that solely mirror the syntactic categories that are relevant to their own part of the conversion recipe (namely, Equations 9.1 to 9.5). This is where the minimal shape exposure dictates the type safety: Would the programmer mistakenly attempt to access other cases of either ADT than the explicitly nominated ones (in lines 2 to 8), compilation will fail. (Compare that with (WF-VCase) in Figure 6.9.)

The third difference is that `SeqConversion` is uneven regarding the cases it demands from the two ADTs. Amongst our conversion components, this latter difference is unique to `SeqConversion`. The only place where one syntactic category gets replaced by a combination of **other**s is, after all, Equation 9.5.

Mixing the components into `Seq2Srp` is similar to `eqer` in Section 9.2.1. The trait `Conversion` used in line 9 of Figure 9.2 is the counterpart of `EqBase` in Section 9.2.1.

```
1 trait Conversion[E1 <: LazyExp[E1], E2 <: LazyExp[E2]] {
2   protected def this_case(e1: E1): E2
3   def convert(e1: E1): E2
4 }
```

However, given that there is no default case in the conversion exercise, instead of inheriting from `Conversion`, the conversion counterpart of `FinalEq` inherits from `VarConversion`.

```
1 trait ConversionFinal[
2                         E1 <: LazyExp[E1],
3                         V1 <: IVar with E1,
4                         E2 <: LazyExp[E2],
5                         V2 <: IVar with E2
6                       ] extends VarConversion[E1, V1, E2, V2] {
7   override def convert(e1: E1): E2 = super.this_case(e1)
8 }
```

`VarConversion` is the corresponding conversion component of Equation 9.1. What is special about `VarConversion` amongst our conversion components is that it ties the recursive knot. Notice that, in Equation 9.1, there is no recursive $[\![.]\!]$ call. Here is `VarConversion` itself:

```
trait VarConversion[
                     E1 <: LazyExp[E1],
                     V1 <: IVar with E1,
                     E2 <: LazyExp[E2],
                     V2 <: IVar with E2
                    ] extends Conversion[E1, E2] {
  override protected def this_case(e1: E1): E2 = e1 match {
    case v1: IVar => incarnator(new IVar(v1.name))
    case _ => throw new IllegalArgumentException("...")
  }
}
```

(See Section 8.3 for the role of `incarnator`.)

Using a similar approach to how we built `eqer`, one can get `convert` below by mixing in all the relevant conversion components. (See our codebase for more details.) Once `convert` is at hand for both the expression conversion and its extensions to heaps and derivation trees, the line below

```
s3opsem.eval(convert(g), convert(e)) == convert(s2opsem.eval(g, e))
```

can be used to test whether, for a given expression e and a heap Γ, the following commutativity property holds:

$$\Gamma : e \Downarrow_{\mathcal{S}_2} \Delta : v \text{ iff } [\![\Gamma]\!] : [\![e]\!] \Downarrow_{\mathcal{S}_3} [\![\Delta]\!] : [\![v]\!]. \tag{9.6}$$

In words, Equation 9.6 says: 'One can either perform an $\mathcal{S}_2$ evaluation and then convert to $\mathcal{S}_3$, or convert first and perform the evaluation under $\mathcal{S}_3$—the result will be the same.' That is, conversion and evaluation commute from $\mathcal{S}_2$ to $\mathcal{S}_3$.

It now only remains to complete the picture of `SeqConversion` as promised in Figure 9.2. The implementation body of `SeqConversion` comes in Figure 9.3. Note that the cast in line 4 of Figure 9.3 is again because, due to erasure, Scala loses the relevant type information upon the pattern matching in line 3. Yet, just like the case for `EqApp` in Section 9.2.1, the cast in line 4 is guaranteed to never fail. The difference here is that, instead of Scala's built-in conversion mechanism, we are this time employing the Shapeless `cast` method in line 4.

9.2.3 Discussion

In terms of ECP, there is a categorical difference between the solutions in Section 9.2 and those of Chapters 7 and 8. Unlike the latter, the former addresses function concerns using a verified software product-line [TBKC07, TAK+14]. (In fact, the former is the only instance of a Feature-Family-Product-Based Analysis [TAK+14] that we are aware of.) The difference becomes further clear when one considers the coding discipline each group of components is

```
trait SeqConversion[...] extends Conversion[Exp1, Exp2] {
  abstract override protected def this_case(e1: Exp1): Exp2 = e1 match {
    case raw_s1: ISeq[_, _] => {
      val s1 = raw_s1.cast[ISeq[Exp1, Seq1]].get
      val ret_e1 = convert(s1.e1)//ret_e1 == [[e1]]
      val ret_e2 = convert(s1.e2)//ret_e2 == [[e2]]

      val x = fresh()
      val y = fresh()

      val id: Exp2 = IVal[Exp2, Val2, Let2](y, new IVar(y))//id == λy.y
      val l: Exp2 = IVal[Exp2, Val2, Let2](x, id)//l == λx.id
      val ei: Exp2 = ISrp[Exp2, Srp2](l, x + "1")//ei == l # x1
      val eii: Exp2 = ei(x + "2")//eii == ei x2
      //return let {x1 ↦ [[e1]], x2 ↦ [[e2]]} in eii
      new ILetVal[
                  Exp2,
                  Val2,
                  Let2
                 ](Map(x + "1" -> ret_e1, x + "2" -> ret_e2), eii)
    }
    case _ => super.this_case(e1)
  }
}
```

Figure 9.3: Mechanisation Body of Equation 9.5

shipped to their client with. For example, compare the following two: how components are used to get `eqer` in Section 9.2.1 versus how they are used to get `L1OpSem` in Section 8.2. In order to use the latter, the client has to provide a function (`eval`) that distributes the task amongst the (semantics) components by explicitly calling the appropriate ones after a client-side pattern match. For the former, the client assembles the appropriate components using simple mixin composition; and, uses a readily available method of the mixin stack that automatically performs the pattern matching. In the terminology of Sommerville, the former is a *sequential composition* [Som11, §17.3] whilst the latter is an additive one. (See Remark 5.6 for more on the latter.) Despite that difference, ensuring soundness of composition is similar for equality and conversion components and executable rules. They both, after all, build on top of our ICs to address function concerns.

On the other hand, more elaboration is needed for the solutions in Section 9.2 to implement E2: Compilation succeeds if the `eqer` programmer forgets to mix `EqVar`, for example; when used for variables, a runtime exception reporting the absence of a suitable pattern match will be thrown. In

this very case, one knows that the set of components to mix in for `eqer` suit compatible extensions to $\mathcal{S}_1$. Accordingly, one can easily employ self-type annotation in the equality components to statically enforce mixing all the $\mathcal{S}_1$-related components. This will guarantee E2. Yet, to implement E3 as well, for every new concrete extension to $\mathcal{S}_1$, one needs to: implement new equality components that inherit from the existing ones but provide the same functionalities; implement new equality components for the additional ADT cases; and, augment the self-type annotations in all the new components to also enforce the availability of pattern matching for the additional ADT cases. In fact, this very involved technique solves EP *at the component level*.

It is worth noting that the resort to stackability of mixins is not fundamental to the solutions presented in Section 9.2. In a host PL like C++ where template specialisation is provided, the decentralisation components all have the same name, and, the compiler integrates the pattern matching automatically. In the absence of that, we simulate it by manually directing the flow of execution.

Remark 9.2. The phrase "Separate Compilation" is used to refer to two different concepts in this thesis and the C++ language. In C++, classes and functions can have declarations as well as definitions. For various reasons, the common practice is to put declarations in header files and definitions in implementation files. As such, it is the declaration and definition that get to compile separately. And, then, the right implementation will be chosen at link time.

Due to technical reasons, C++ templates cannot enjoy this separate compilation (unless used under EDG with the `export` keyword, which was deprecated from C++11 [Cpp11] onwards). In fact, the declaration and definition of a template can indeed be compiled separately. However, when only the declaration is available at a call site, the linker will fail to find the right definition for that entails "instantiating" a separately compiled definition – which is not quite what a linker does.

Despite all that, templates do enjoy our EC4 notion of separate compilation in that both compilation and linkage of different template specialisation definitions can well happen separately – so long as they are equally in scope. (Note that this is not exactly correct about different template specialisation declarations – the practice of which might very barely be chosen these days ever.) □

Remark 9.3. Using similar narrowing techniques for semantics, one can gain semantics mechanisation narrowing. Combining the result with the compatible extension techniques developed in Section 8.4.1 gives rise to a manual simulation for the bidirectional adaptation of $J\&_s$ [QM09]. That is,

```
1 trait EtaRed extends EtaReducer[S2Exp] with
2   VarEtaReducer[S2Exp, Var]            with
3   AppEtaReducer[S2Exp, Val, App, Let]  with
4   ValEtaReducer[S2Exp, Val, Let]       with
5   LetEtaReducer[S2Exp, Val, Let]       with
6   SeqEtaReducer[S2Exp, Sequ]           with
7   EtaReducerFinal[S2Exp, Var]

8 implicit val eta_red_exp = {e => (new EtaRed () {}).reduce(e)}
```

Figure 9.4: η-Reduction for $\mathcal{S}_2$

one uses the latter techniques to get from a PL to its extension; the former techniques can get one back to a core PL. □

9.3 Optimisation

Hofer and Ostermann [HO10] say a DSL interpretation is *compositional* when "the meaning of an expression may depend only on the meaning of its sub-expressions, not on their syntactic form or some context." Our executable rules have already shown how we manage compositional interpretations. This section shows how the technique used in Section 9.2 (i.e., integration of a decentralised pattern matching) serves *non-compositional transformations* as well.[3] Our working example will be operational semantics optimisation. We accommodate institution of measures that statically reject inapplicable optimisations.

We employ two pre-evaluation optimisations: removal of unaccessed let-bindings and the famous η-reduction.[4] As a fully syntactic optimisation, `EtaRed` in Figure 9.4 is the type that integrates the decentralised pattern matching for $\mathcal{S}_2$ **syntax**; `eta_red_exp` (line 8) is an instance of that type. Just like `eqer` and `Seq2Srp` of Section 9.2, the components used in lines 1 to 7 of Figure 9.4 each add their own part of pattern matching for the mix to cover all the $\mathcal{S}_2$ cases of η-reduction. Here is how to optimise an `S2OpSem` instance with `eta_red_exp` (topped up with `cancel_el` for removal of empty let-bindings):

[3] See `prephs.Augment` for another compositional transformation in our codebase that implements derivation tree augmentation [Hae10, Definition IV.6] for compatible extensions to $\mathcal{S}_2$.

[4] Check our codebase for other optimisations developed in the same way: `Strictness-Dismissal` for $\alpha <_{\mathscr{C}} \mathcal{S}_3$ [Hae13, Corollary 1] and `cancel_el` for transforming let {} in e into e.

```
val os = s2opsem optimise_using (eta_red_exp andThen cancel_el)
```

The other optimisation that we consider in this section is based on the following observation for the family of lazy PLs the value type of which is Val:

Proposition 9.4. $\Gamma : e \Downarrow_{\Pi'} \Delta : v$ *if and only if* $\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow_{\Pi} \Delta : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ v$ *when* x_i*s are all fresh in* Γ *and do not occur in* e.

Thus far, we have only shown semantics mechanisation when the **complete derivation tree** of an evaluation was expected of our `proofsearch` methods. Let us call these mechanisations the *profiling* ones. An alternative would be a mechanisation where only the pair of final heap and final value are expected (line 7 below). We call the latter mechanisations the *non-profiling* ones. All our non-profiling semantics mechanisations derive from `NProfOpSem` below.

```
1 trait NProfOpSem[
2                  Exp <: LazyExp[Exp],
3                  Val <: AbsVal[Exp, Val] with Exp
4                ] extends OpSem[Exp] {
5   implicit val opsem: NProfOpSem[Exp, Val]

6   type Conf = HBConf[Exp]
7   type Node = (Heap[Exp], Val)

8   type This = NProfOpSem[Exp, Val]
9 }
```

More on `This` in line 8 shortly. See Section 8.1 for `opsem` and `HBConf` in lines 5 and 6 above. `AbsVal` is the common base class of `ILam` and `IVal`.

The reason why we consider Proposition 9.4 an optimisation is that it cancels the need for copying unneeded let-bindings (those of x_is). It legislates computing Π' (instead of Π) and returning the Δ and $\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ v$ pair. However, for profiling mechanisations, the entire Π needs to be returned. But, to produce that out of Π', one has to augment every heap with the removed bindings – which defeats the purpose of Proposition 9.4. That is, this Proposition can only serve as an optimisation to non-profiling mechanisations because the following attempt to produce Π is not a valid $(\mathbf{let})_{\mathcal{S}_2}$ instance [Hae10]:

$$\frac{\Gamma : e \Downarrow_{\Pi'} \Delta : v}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ v}\ (\textsc{Wrong-Let})_{\mathcal{S}_2}.$$

We enforce correct application of Proposition 9.4 similar to how Scala employs `CanBuildFrom` for its Collection Library [OSV11, §25.2]. For a se-

mantics mechanisation `os` of type `OS` and an optimiser `o` of type `O`, the compiler will only allow optimisation of `os` using `o` when a unique implicit of type `CanBeOptUsing[OS, O]` is accessible. (Again, more on the role of `This` in line 3 below shortly.)

```
1  trait OpSem[Exp <: LazyExp[Exp]] {
2    ...
3    type This >: (this.type) <: OpSem[Exp]
4    def optimise_using[O](o: O)
5                         (implicit ev: CanBeOptUsing[This, O]): This =
6      ev.build_optos(this, o)
7  }

8  trait CanBeOptUsing[OS <: OpSem[_], O] {
9    //OptOS == type of the optimised operational semantics, i.e., the
10   //operational semantics mechanisation obtained by optimising OS using O
11   type OptOS <: OS
12   def build_optos(os: OS, o: O): OptOS
13 }
```

By providing an implicit for the non-profiling mechanisations and not doing so for the profiling ones, we statically ban any use of Proposition 9.4 for the latter group. In our codebase, `S2NPOpSem` is a non-profiling mechanisation of the $\mathcal{S}_2$ semantics. Hence, an `S2NPOpSem` instance legally tops up several optimisations, including `ulremover` – the one that performs Proposition 9.4:

```
val os = new S2NPOpSem[...] () {} optimise_using ulremover optimise_using
  (eta_red_exp andThen cancel_el)
```

Unlike `S2NPOpSem`, trying, for example, to use `ulremover` for an `L1OpSem` instance will be rejected statically for the unavailability of a legislating `CanBeOptUsing`.

As an example of how to provide the optimisation legislating implicit, consider the code below. Using its type constraints in lines 2 to 4, `ulOptNPOS` performs that action for every non-profiling operational semantics mechanisation on every PL syntax with `Val` and `Let` cases. It is using the code below that optimising `S2NPOpSem` using `ulremover` is legislated.

```
1  implicit def ulOptNPOS[
2                         Exp <: LazyExp[Exp],
3                         Val <: IVal[Exp, Val, Let] with Exp,
4                         Let <: ILet[Exp, Val, Let] with Exp
5                        ] =
6    new CanBeOptUsing[NProfOpSem[Exp, Val], UselessLetRemTag] {
7      type OptOs = UselessLetRem[Exp, Val, Let, NProfOpSem[Exp, Val]]

8      override def build_optos(os: NProfOpSem[Exp, Val],
9                               dummy: UselessLetRemTag): OptOS =
10       new UselessLetRem[Exp, Val, Let, NProfOpSem[Exp, Val]](os)
```

```
11 }
```

Similarly, for a new optimisation or operational semantics, all one needs to do is to provide an implicit `CanBeOptUsing` of the right type. The scalability of instituting static measures in this fashion is clear. We would like to invite the reader to check our online codebase. Whilst we omit its presentation here, one will observe it then that providing a new `CanBeOptUsing` is trivial enough to alleviates many composability concerns. In particular, unlike [OC12] and [OvdSLC13], we need not ship automatic combinators for `CanBeOptUsing`.

It is now time to fulfil our promise on considering `This` in line 8 of `NProfOpSem` and line 3 of `OpSem`. The Scala automatic type inference has the interesting property that it tries to be as specific as possible. This property often plays in favour of the Scala programmer by dismissing their need for being explicit about types. Yet, there are situations where the "most specific" type is just *too specific* for a given application. An example of that is our above resort to `This`. The works of Bruce et. al, in particular, tackle many different aspects of similar challenges [BPF97, BOW98, Bru03, BF04].

The problem here is that, if line 5 in `OpSem` is replaced by

```
5 (implicit ev: CanBeOptUsing[this.type, O]): this.type =
```

the optimisation will exclusively be available to the very instance of operational semantics mechanisation for which the corresponding `CanBeOptUsing` is in scope. In particular, it will no longer be available to the compatible extensions of the system. And, when replaced by

```
5 (implicit ev: CanBeOptUsing[OpSem[Exp], O]): OpSem[Exp] =
```

one cannot stack different optimisations on top of one another. To surmount that obstacle, we leave `This` open for the operational semantics mechanisations to define their domain of optimisation top-ups. Yet, as shown in line 3 of `OpSem`, the abstract type `This` needs to still be an `OpSem[Exp]`. Furthermore, `this` (in the body of `OpSem`) needs to be an instance of `This`.

Chapter 10

Final Remarks

In this chapter we provide our final remarks about this thesis itself. We start by Section 10.1 to report limitations we faced over trying our approach against the lazy PLs of Chapter 3. Conclusion comes next in Section 10.2. Then, future work is discussed in Section 10.3.

10.1 Limitations

As discussed in detail in Section 5.3, in our approach for CBM, there are two (main) roles: the PL implementer and the component vendor. We also explained our programming discipline for syntax, semantics, and analysis in Chapters 7 to 9. In this section, we explore limitations of our approach in terms of its two CBM roles. The discussion in this chapter mostly considers scalability impediments against our approach. Fortunately enough, in most cases, the LDF can easily play in the favour of our approach. Note that, whilst we used Scala as our LDF, over the discussion provided in this section, LDF is not necessarily Scala.

Given the minimal programming discipline that our approach imposes on the PL implementer (a.k.a. the LDF user), the impediments are very few as far as the implementer is concerned. In presence of a prolific arsenal of our components, the most important one is perhaps accessibility matters. Other than that, it only remains to avoid some boilerplate coding. These are discussed in Section 10.1.1 in more details.

The story is totally different for the component vendor because they need to accommodate new requirements of existing and new PLs. To that end, various scenarios are probable. Sometimes, mere addition of the requested new components suffices. In other occasions, rather than requesting a new

component, the LDF user is in need for new auxiliary **tools** that serve the components. For example, in the realm of lazy evaluation, a heap is a keystone of the operational semantics that will not necessarily be noticed by the PL user. This is because heaps are not a part of the PL syntax and not a component of the semantics. However, the semantic components make use of them as an auxiliary tool. Sometimes new requirements impose refactoring to the related ecosystem. As explored in Section 10.1.2, the last scenario describes most CBM impediments under LDFs.

There are also scalability issues that pertain to the interaction between the LDF and its user. These have mainly to do with the LDF providing an easily-understandable ontology of components that can also prevent some wrong choices. We do not discuss such issues and their HCI-related matters in this section.

10.1.1 PL Implementer Issues

This section discusses the impediments faced by the PL implementer that we currently anticipate for scalability of our approach. We divide these impediments into two groups:

Accessibility

Finding the right component is always a matter of concern from the user's viewpoint. Obviously, this concern becomes harder to alleviate as the repository grows. Proper query languages, improved GUI (and the rest of HCI matters), and proper theory support for mechanical decision making on the observational properties of components are most important to address here. These are discussed in more detail in [JMS10]. Other than those, the need for specification of ontological relationships between components is also serious. See Section 7.3.2 for a few such relationships.

Incarnation

Our programming discipline demands that every expression trait provides a method

```
def incarnator: ICBase => Exp
```

that is responsible for turning an IC into a PL's case of the right type, if any. (Refer back to Section 8.3 for `incarnator`.) The body of this `incarnator` is currently a trivial pattern-match over `ICBase` that given its arguments, will simply pass the same arguments to the right case of the expression trait.

Fortunately, this boilerplate can be avoided using proper GUI support of the LDF or Scala macros.

Note that such incarnations can unfortunately not take place in the ICs themselves. Consider the `ILam`, for example:

```
class ILam[
          E <: LazyExp[E], V <: ILam[E, V] with E
         ](...) extends ... {...}
```

The problem is that all `ILam` knows about `E` is that it has a case `V` that inherits from `ILam`. That piece of information, however, does not suffice for creating a `V` instance. This is because rather than pinpointing its exact type, it only describes a property of `V`. As a result, the incarnation cannot take place here, and, has to be postponed to when the exact `V` type is known. That is when the expression trait itself (i.e., the exact `E` instance) is known. Note that, in current Scala technology, type constraints cannot enforce a particular constructor (signature) on type parameters.

10.1.2 Component Vendor's Issues

This section lists a few scenarios when new mechanisation requirements are likely to cause refactoring in the component repository (of an LDF or a component vendor).

Encoding Component Requirements

Right now, to enforce component requirements statically, we encode them in the (Scala) type system. However, no type system is originally designed for this specific purpose. As a result, the encoding is rather a hack into the type system that has to learn to live with the type system limitations. A complimentary issue is portability, which is due to the different levels of expressiveness type systems have. Furthermore, a requirement that is easily expressed in one type system might need a complicated formulation in another. These issues become increasingly severe as the LDF scales and the requirements get complicated. Remember that the LDF type systems themselves are also subject to updates that might influence expressiveness.

As far as CBM is concerned, what we understand from the previous paragraph is the necessity of a first-class support for component requirement specification. Various aspects of such a support are still not studied. Studying the difficulties faced over type system encoding can be instructive. An initial study is made in Section 6.3. But, proofs are missing and the type system has still not been tested against real examples. Regardless of the type system studies, paragraphs **Invalidation** and **Reformulation** explain two such difficulties that we anticipate may hinder scalability.

Invalidation. The first four systems that we tried our approach for ($\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{S}_1$, and $\mathcal{S}_2$) had a simple notion of identifiers. $\mathcal{L}_2$, however, builds on $\mathcal{L}_1$ whilst adding a new syntactic category for metavariables. (See Figure 3.2.) Addition of the respective IC (i.e., `IMVar`) was an easy task per se. Yet, that addition turned out to **invalidate** some requirement formulations.

For example, before that, we had implemented a single IC for function application that could be uniformly used by $\mathcal{L}_1$, $\mathcal{S}_1$, and $\mathcal{S}_2$. This reusability was due to the satisfiability of the respective IC's specified requirements by all the latter systems:

```
1  object HBAppRule {
2    def apply[
3              Exp <: LazyExp[Exp],
4              Val <: AbsVal[Exp, Val] with Exp,
5              App <: IApp[Exp, App] with Exp,
6              OS <: OpSem[Exp]{type Conf = HBConf[Exp]
7                               type Node = HBNode[Exp, Val]}
8             ](g: Heap[Exp], ex: App)
9              (implicit opsem: OS): HBNode[Exp, Val] = ...
10 }
```

$\mathcal{L}_2$ too did meet the specified requirements (lines 3–7 above) of the respective IC (`HBAppRule`). However, the IC is not suitable for the latter system. (Check the Function Application part of Figure 3.5.) In other words, the IC's old requirement specification was no longer valid and had to be updated accordingly. More precisely, a new type constraint needed to be added to enforce availability of a `Var` case. Note that line 6 below is absent in the above version of `HBAppRule`.

```
1  object HBAppRule {
2    def apply[
3              Exp <: LazyExp[Exp],
4              Val <: AbsVal[Exp, Val] with Exp,
5              App <: IApp[Exp, App] with Exp,
6              Var <: IVar with Exp,
7              OS <: OpSem[Exp]{type Conf = HBConf[Exp]
8                               type Node = HBNode[Exp, Val]}
9             ](g: Heap[Exp], ex: App)
10             (implicit opsem: OS): HBNode[Exp, Val] = ...
11 }
```

This is to exclude $\mathcal{L}_2$, which has no case that derives from `IVar` (but one that derives from `IMVar`).

Reformulation. Having tried our approach for $\mathcal{L}_1$, $\mathcal{S}_1$, and $\mathcal{S}_2$, once we started mechanisation of $\mathcal{L}_0$, we faced a subtlety that was not discussed in the respective paper of the latter system [AO93]. All these systems target lazy evaluation where λ-abstractions are designed to be the reduction fixpoints.

With this observation, we based our IC requirement specification in terms of the syntactic category used for λ-abstractions. For example, consider `HBNode` which is used in line 10 above for `HBAppRule`:

```
class HBNode[...](...,
                  val g1: Heap[Exp], val e1: Exp,
                  val g2: Heap[Exp], val e2: Val,
                  ...) {...}
```

Recall from Section 9.1 that the constructor arguments of `HBNode` reflect its scheme of $\Gamma_1 : e \Downarrow \Gamma_2 : v$, where v is a value. However, because $\mathcal{L}_0$ does not consider sharing, reduction also stops when it hits variables. That is, without Abramsky and Ong [AO93] being explicit about it, variables are also irreducible in $\mathcal{L}_0$. We reflect that by providing a new node class for heap-less systems (Figure 9.1) that specifies its requirements in terms of `Irreducible` rather than `IVal`:

```
class HLVarNode[
                 Exp <: LazyExp[Exp],
                 Val <: ILam[Exp, Val] with Exp,
                 Var <: IVar with Exp with Irreducible,
                 Irreducible >: Val <: Exp
               ](val x: Var)
  extends HLNode[Exp, Val, Irreducible](...)
```

Subtle Executable Rule Differences

A distinctive difference between $\mathcal{S}_2$ and $\mathcal{S}_3$ on the one hand, and $\mathcal{L}_1$ and $\mathcal{S}_1$ on the other hand, is the former group's notion of garbage-collection for let-bindings. A consequence of this garbage-collection is that, unlike the other three works, let-bindings need to be taken into consideration over the function application semantic rule **(app)** too. Hence, neither **(app)** of [Hae10] can be written in terms of the other **(app)** rules, nor the other way around. Over trying our approach for $\mathcal{S}_3$, however, we happened to realise that a syntactic trick can unify these. In words, what we call "unlambdafication" of a λ-abstraction is removal of its λ-binding and replacing the (solo-) λ-bound variable by the other passed variable in the result of λ-removal:

$$\begin{aligned} (\lambda y.e)^{-\lambda}[x/\_] &= e[x/y] \\ (\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ \lambda y.e)^{-\lambda}[x/\_] &= \\ \mathsf{let}\ \{x_i{=}e_i[x/y]\}_{i=1}^n\ \mathsf{in}\ e[x/y] & \end{aligned}$$

(Note that, for $\mathcal{S}_2$ and $\mathcal{S}_3$, the former case is a special case of the latter when $n = 0$. Only the first case is applicable, however, for $\mathcal{L}_1$ and $\mathcal{S}_1$.) Armed with the unlambdafication, the two function application rules can be unified to

$$\frac{\Gamma : e \Downarrow \Theta : v_e \quad \Theta : (v_e)^{-\lambda}[x/\_] \Downarrow \Delta : v}{\Gamma : e\, x \Downarrow \Delta : v}\ \textbf{(app)}.$$

One of the dangers against usability of our approach once its number of components grows is, obviously, spotting the right component. Addition of new components, therefore, needs to be done with a lot of care – mainly to prevent reinvention of wheel. We presented this **(app)** example to demonstrate how tricky that task might be even to a human expert. Hence, unless a proper support for mechanical verification of observational equivalence between rules is available, the addition of new components will be a scalability impediment.

Syntactic Categories

One of the minor issues we grapple with is our lack of an acceptable definition for the widely-used term "syntactic category." We use this term without formally defining it. The trivia is that a syntactic category is a non-terminal symbol in the PL syntax. This definition is unfortunately not precise enough for our purposes; it, in particular, does not model the relations between syntactic categories as explained in Section 7.3.2. Hence, it can hardly serve as a means for EC5. Although we did demonstrate in Section 7.3.2 how to address EC5, there are scalability issues that we would like to discuss here.

Recall that to address disjointedness, the trick is to include the following method in both `ILam` and `IVal`.

```
final def disjointed[X <: ILam[_, _] with IVal[_, _, _]] = {}
```

This trick, however, does not scale well for a simple reason. Upon the addition of a new disjoint IC, one needs to

- ensure that the new IC provides the right-signatured `disjoint` method – namely, one that correctly enforces its disjointedness from all the correct ICs.
- update all the existing relevant `disjointed` methods.

Fortunately, this impediment can be removed if the LDF provides a mechanism for specifying it when syntactic categories deal with the same matter. The LDF would then manage the above two updates generatively.

Recall also that to address uncongeniality, our trick is the inclusion of the following line

```
(implicit witness: V <:!< ILam[_, _])
```

of every IC that is uncongenial to `ILam`. The scalability impediment here is

that, upon every addition of a new value syntactic category, all the incompatible ICs need to be updated. This need for update is recurrent whenever a syntactic category is applicable exclusively in presence of a particular other syntactic category from an uncongenial set of categories. Fortunately, the impediment can again be handled automatically by the LDF when it is aware that certain syntactic categories are uncongenial. In such a situation, the LDF can update all the pertaining uncongeniality specifications as soon as an IC restricts its applicability to one of an uncongenial set of categories.

Note that in this very occasion, because we have already used the `disjointed` method trick to prevent inheritance from both `ILam` and `IVal`, the use of `<:!<` may be found redundant. Not always is, however, uncongeniality due to presence of disjoint categories. In such occasions, the `<:!<` will still work. Yet, the user has to be careful about either placing `<:!<` everywhere suitable, or, marks the occasions for the LDF to take care of. Neither option will obviously scale.

10.2 Conclusion

In this thesis, we describe how to adapt CBSE for PL mechanisation. We start in Chapter 2 by discussing the necessity of that and the lack of CBM support in the current mechanisation technology. We proceed by justifying our choice of performing CBM in an embedded setting (rather than under the readily available LDFs). We define ECP by carefully formulating its eight concerns. We also explain how solving ECP is crucial to CBM in an embedded setting. Then, we explore the few embedded mechanisation technologies that make it to the vicinity of ECP. We show how none of such technologies, however, manages to address all the ECP concerns.

In Chapter 3, we take a short look into the group of lazy functional PLs that we choose as our running example of thesis. Next, in Chapter 4, we review the related work. Most notably, in the latter chapter, we present detailed comparison between works on the following topics and this thesis: various LDFs, EP and EFP, as well as family and shape polymorphism. Our vision about CBM, i.e., its roles and processes is then presented in Chapter 5. In particular, we explain our understanding that, in CBM, the PL implementer and the component vendor cooperate towards PL mechanisation with the LDF acting as a the framework for common development.

The main development of this thesis starts in Chapter 6. We begin the chapter by offering a stand-alone formal definition for the least studied ECP concern: compatibility upon extensions. Next, we introduce CBMCalc – a formal model for CBM – by presenting its syntax, static semantics, and

dynamic semantics. All the developments of Chapter 6 are the first of their kind. The *raison-d'être* of these developments is to serve as formal descriptions for our approach for embedded CBM. To that end, we use the material of Chapter 6 in the upcoming chapters to communicate ideas by abstracting away from implementation technicalities.

Then, in Chapters 7 to 9, we provide our reusable components for embedded mechanisation of PL syntax, semantics, and analysis. We show how these components are the first to address all the ECP concerns. Together, these components are also the first to foster full-cycle CBM. These three chapters study both our components themselves and how to assemble them to acquire complete PL mechanisations. They explain our programming discipline for both the component vendor and the PL implementer. These three chapters all showcase their developments over the lazy PLs of Chapter 3. More specifically, we show how our components can serve as a means for reuse of mechanisation when moving from one of these PLs to another, whilst preventing wrong reuse as well.

In Chapter 8, we present two techniques for semantics mechanisation. We show how the second one (Section 8.2.2) scores all the relevant ECP concerns. Then, in Chapter 9, we present various PL analyses and their mechanisation in our approach. The latter chapter provides two techniques for dealing with analysis mechanisation. The second one (Section 9.2) can also be used for semantics mechanisation and does score its relevant ECP concerns. We show how the two ECP solutions of Section 8.2.2 and Section 9.2 are correct w.r.t. our mathematical development in Chapter 6.

Our second ECP solution can support non-compositional transformations as well (Section 9.3). Our two ECP solutions both institute new semantics mechanisation approaches. The mechanisation approach presented in this thesis is inspired by LMS, Polymorphic Variants, and verified software product-lines. Yet, the only host PL ingredients which are necessary to our approach are multiple inheritance and type constraints. Our mathematical development of Chapter 6, on the other hand, is inspired by the Church encoding of ADTs and the three calculi offered by earlier research on lightweight family polymorphism.

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., ICs for each case) to get the abstract syntax and binding the case constructors under some fixed nested type of the expression trait for the concrete syntax. (See Section 7.2.1.) For semantics, it takes 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. (See Section 8.2.) One can alternatively mix components like features as described in Section 9.2.

We would like to finish this section by a brief account of the lessons learned over mechanisation of research PLs. As outlined in our paper [HS14, §V], mechanisation of $\mathcal{S}_3$ revealed an unnoticed mistake in our own theory work. It also helped us notice the necessity of a side condition in one of our theorems on $\mathcal{S}_3$. Finally, mechanisation of $\mathcal{L}_2$ revealed it that the system fails to refuse recursion with direct dependency. This was done by our mechanisation showing it that the derivation in Fig. 10 of the respective paper [Sin08] is in fact wrong.[1]

10.3 Future Work

A survey on the consequences of failing to follow our discipline (whilst using our approaches) is on our plan. Embedded PL mechanisation can become very clunky in comparison to when done under stand-alone LDFs. (See our plethora of type constraints throughout the thesis.) It would be interesting to try our solutions under the latter circumstances, especially in presence of first-class support for syntactic categories. We also aim to solve the expression problem at the component level. (See Section 9.2.3.)

On the other hand, the essence of components is their isolation. Yet, proving properties of PLs currently only happens holistically for the entire PL syntax and semantics. (The very few exceptions, such as [BB08], focus on variations to a wholly available core PL.) Whilst proving the properties of our mechanisation components is possible in isolation, the lack of a counterpart component-based induction is painful in our proofs. (See the volume of proof repetition for $\mathcal{S}_3$ [Hae13] in comparison with that for $\mathcal{S}_2$ [Hae09], for example.) Provision of a component-based induction principle is another fascinating future work of ours. The recent work of Churchill and Mosses [CM13] pioneers that similar research by providing modular proofs for PLanCompS funcons. Another work that can be similarly inspiring is that Schwaab and Siek [SS13] on modularising Agda formal proofs.

Shipping our components is certainly a new burden on LDFs. Working with constrained type parameters as opposed to exact types makes some extra indirection inevitable. The use of CBMCalc in the rest of this thesis suggests the necessity of family polymorphism. That would be even more radical shift for LDFs to accommodate CBM. The burden on LDFs magnifies when they are to ship an exhaustive set of our components which the PL

[1] The author is already informed of this mistake.

implementer can freely mix-and-match; that is, after all, likely to take several rounds of refactoring on its way. How to extend available LDFs for all that is another topic next research. The recent work of Mosses and Vesely [MV14] on embedding PLanCompS funcons in K can be inspiring.

Whether or not our approach will scale is a topic for further research. The preliminary experiment report of Section 10.1 suggests that, with better LDF support, that is not unlikely. 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 [Lau93] is routine. Addition of Eden's strict function application [LOMP05] would also be relatively easy. However, we anticipate that adding the lazy evaluation material of Danvy et al. [DMMZ12] 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 easily be composed out of existing ones. For example, the subtle difference between $(\mathbf{app})_{\mathcal{L}_1,\mathcal{S}_1}$ and the function application rule of $\mathcal{S}_2$ suggests that neither can be implemented in terms of another. The study of atomic support for implementing our components is yet another future work.

We end by a comment on embedding concrete syntax. Of the reasons why we chose Scala for embedded CBM was its outstanding support for coining new concrete syntax. However, when embedding a family of similar pieces of concrete syntax, the job can become severly difficult, especially when aiming for backward compatibility (EC2). (See Section 7.4 for more.) A promising alternative is factoring the concrete syntax embedding out and leaving it completely to the SugarScala [Jak14]. Yet, we did not try that idea because SugarScala is based on SDF [HHKR89], which is untyped. Hence, unlike our approach, one cannot use a type system to cater automatic distinction between similar but different pieces of concrete syntax. How to overcome that is another challenge in future work.

Appendix A

Scala Essentials

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

```
1 object O {def apply(n: Int) = ...}
2 class C1[T] { def m[U]:  Int -> Int = ... }
3 class C2[T1 <: T2]
4 class C3[T1 <: T2{type NT}]
5 class C4[T <: C2[_]]
```

The method `apply` (line 1) tells Scala to expand calls like `O(1)` to `O.apply(1)`. Class `C1` is parametrised over type `T` (line 2). Likewise, method `m` is parametrised over type `U` (line 2). The type parameter `T1` of `C2` is constrained by an *upper bound* `T2` (line 3). 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 4). One can place constraints on nested types of type parameters too. Use of underscore in line 5 indicates that one can instantiate `C4` 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. Traits are like abstract classes, from more than one of which a single class or trait can inherit at the same time. Above words about class type parameters apply to traits too. Scala uses `T1 with T2` for the intersection type of `T1` and `T2`.

```
1 val f = (n: Int) => n + 2
2 def g(n: Int)(implicit d: Double) = ...
3 implicit val pi: Double = 3.14
```

In line 1 above, `f` is bound to an *anonymous* function that, given an integer n, returns $n + 2$. The method `g` in line 2, takes two *parameter packs*, the last of which being *implicit*. The consequence is that, when a unique implicit double `id` is in scope, a call like `g(10)` is equivalent to `g(10)(id)`. Line 3 above demonstrates how to define an implicit.

Glossary

ADT Algebraic Datatype. 15

AST Abstract Syntax Tree. 34

CBM Component-Based Mechanisation. 3

CBSE Component-Based Software Engineering. 11

DSL Domain-Specific Language. 13

ECP Expression Compatibility Problem. 10

EFP Expression Families Problem. 18

EP Expression Problem. 10

FOP Feature-Oriented Programming. 11

GPL General-Purpose Language. 14

IC Intermediate Class. 38

IDE Interactive Development Environment. 29

LDF Language Definitional Framework. 7

LMS Lightweight Modular Staging. 19

PL Programming Language. 1

Bibliography

[AAL12] M. Asavoae, I. Asavoae, and D. Lucanu, *On Abstractions for Timing Analysis in the K Framework*, Found. & Prac. Aspects of Resource Analysis (R. Peña, M. van Eekelen, and O. Shkaravska, eds.), Lect. Notes in Comp. Sci., vol. 7177, Springer Berlin/Heidelberg, 2012, pp. 90–107.

[ABKS13] S. Apel, D. S. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*, Springer-Verlag, October 2013.

[ACN+09] E. Allen, R. Culpepper, J. D. Nielsen, J. Rafkind, and S. Ryu, *Growing a Syntax*, Proc. Int. W. Found. Obj.-Oriented Lang., 2009.

[AGMO06] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann, *An Overview of CaesarJ*, Trans. Aspect-Oriented Soft. Dev. I (A. Rashid and M. Aksit, eds.), Lect. Notes in Comp. Sci., vol. 3880, Springer, 2006, pp. 135–173.

[ALR11] M. Asavoae, D. Lucanu, and G. Roşu, *Towards Semantics-Based WCET Analysis*, Proc. 11^{th} Int. W. Worst-Case Execution-Time Analysis, July 2011.

[AO93] S. Abramsky and C.-H. Ong, *Full Abstraction in the Lazy Lambda Calculus*, Inf. & Comp. **105** (1993), no. 2, 159–267.

[Bak09] S. Z. Bak, *Industrial Application of the System-Level Simplex Architecture for Real-Time Embedded System Safety*, Master's thesis, Dept. Comp. Sci., Coll. Eng., U. Illinois at Urbana-Champaign, IL, USA, 2009.

[BB85] C. Böhm and A. Berarducci, *Automatic Synthesis of Typed Lambda-Programs on Term Algebras*, Theo. Comp. Sci. **39** (1985), 135–154.

[BB08] D. S. Batory and E. Börger, *Modularizing Theorems for Software Product Lines: The Jbook Case Study*, J. Univ. Comp. Sci. **14** (2008), no. 12, 2059–2082.

[BF04] K. B. Bruce and J. N. Foster, *LOOJ: Weaving LOOM into Java*, Proc. 18^{th} Euro. Conf. Obj.-Oriented Progr. (M. Odersky, ed.), Lect. Notes in Comp. Sci., vol. 3086, Springer, June 2004, pp. 389–413.

[BGP95] R. Bellinzona, M. G. Gugini, and B. Pernici, *Reusing Specifications in OO Applications*, IEEE Soft. (1995), 65–75.

[BH12] 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.

[BM03] R. Bruni and J. Meseguer, *Generalized Rewrite Theories*, Automata, Lang. Prog. 30^{th} Int. Coll., Proc. (C. M. J. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, eds.), Lect. Notes in Comp. Sci., vol. 2719, Springer-Verlag, June 2003, pp. 252–266.

[BOS+11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, *Mathematizing C++ Concurrency*, Proc. 38^{th} ACM SIGPLAN-SIGACT Symp. Princ. Prog. Lang. (T. Ball and M. Sagiv, eds.), ACM, January 2011, pp. 55–66.

[BOW98] K. B. Bruce, M. Odersky, and P. Wadler, *A Statically Safe Alternative to Virtual Types*, Proc. 12^{th} Euro. Conf. Obj.-Oriented Progr. (E. Jul, ed.), Lect. Notes in Comp. Sci., vol. 1445, Springer, July 1998, pp. 523–549.

[BPF97] K. B. Bruce, L. Petersen, and A. Fiech, *Subtyping Is Not a Good "Match" for Object-Oriented Languages*, Proc. 11^{th} Euro. Conf. Obj.-Oriented Progr., 1997, pp. 104–127.

[BPM13] C. Bach Poulsen and P. D. Mosses, *Generating Specialized Interpreters for Modular Structural Operational Semantics*, Proc. 23^{rd} Int. Symp. Logic-Based Prog. Synth. & Transf.,

Lect. Notes in Comp. Sci., Springer, September 2013, To Appear.

[Bru03] K. B. Bruce, *Some Challenging Typing Issues in Object-Oriented Languages*, Elec. Notes Theo. Comp. Sci. **82** (2003), no. 7.

[BS02] C. Brabrand and M. I. Schwartzbach, *Growing Languages with Metamorphic Syntax Macros*, Proc. 9^{th} ACM SIGPLAN W. Partial Eval. & Prog. Manip., ACM Press, 2002, pp. 31–40.

[BSR03] D. S. Batory, J. N. Sarvela, and A. Rauschmayer, *Scaling Step-Wise Refinement*, Proc. 25^{th} Int. Conf. Soft. Eng. (L. A. Clarke, L. Dillon, and W. F. Tichy, eds.), IEEE Computer Society, May 2003, pp. 187–197.

[CB07] F. Chalub and C. Braga, *Maude MSOS Tool*, Elec. Notes Theo. Comp. Sci. **176** (2007), no. 4, 133–146.

[CCH+89] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell, *F-Bounded Polymorphism for Object-Oriented Programming*, FPCA '89: Proc. 4^{th} Int. Conf. Func. Prog. Lang. & Comp. Arch., September 1989, pp. 273–280.

[CDE+03] 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.), Lect. Notes in Comp. Sci., no. 2706, Springer-Verlag, June 2003, pp. 76–87.

[CDEM07] M. Clavel, F. Durán, S. Eker, and J. Meseguer, *All about Maude: A High-Performance Logical Framework*, Springer-Verlag Berlin Heidelberg, February 2007.

[CDNW07] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad, *Tribe: A Simple Virtual Class Calculus*, Proc. 6^{th} Int. Conf. Aspect-Oriented Soft. Dev. (B. M. Barry and O. de Moor, eds.), ACM Int. Conf. Proc. Series, vol. 208, ACM, March 2007, pp. 121–134.

[CH00] K. Claessen and J. Hughes, *QuickCheck: A Lightweight Tool for Random Testing of* Haskell *Programs*, Proc. 5^{th} ACM

SIGPLAN Int. Conf. Func. Prog. (M. Odersky and P. Wadler, eds.), ACM, 2000, pp. 268–279.

[CHC90] W. R. Cook, W. L. Hill, and P. S. Canning, *Inheritance is not Subtyping*, Proc. 17^{th} ACM SIGPLAN-SIGACT Symp. Princ. Prog. Lang. (San Francisco, CA, USA), ACM, 1990, pp. 125–135.

[CL95] C. Chambers and G. T. Leavens, *Typechecking and Modules for Multimethods*, Trans. Prog. Lang. & Sys. **17** (1995), no. 6, 805–843.

[CLCM00] C. Clifton, G. T. Leavens, C. Chambers, and T. D. Millstein, *MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java*, Proc. 15^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (Minneapolis, Minnesota, USA), ACM, 2000, pp. 130–145.

[CM13] M. Churchill and P. D. Mosses, *Modular Bisimulation Theory for Computations and Values*, Proc. 16^{th} Int. Conf. Found. Soft Sci. Comp. Struct. (F. Pfenning, ed.), Lect. Notes in Comp. Sci., vol. 7794, Springer, March 2013, pp. 97–112.

[CMT14] M. Churchill, P. D. Mosses, and P. Torrini, *Reusable Components of Semantic Specifications*, Proc. 13^{th} Int. Conf. Modularity (Lugano, Switzerland) (W. Binder, E. Ernst, A. Peternier, and R. Hirschfeld, eds.), ACM, 2014, pp. 145–156.

[Coo90] W. R. Cook, *Object-Oriented Programming Versus Abstract Data Types*, Proc. Int. W. Found. Obj.-Oriented Lang. (Noordwijkerhout (The Netherlands)) (J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds.), Lect. Notes in Comp. Sci., vol. 489, 1990, pp. 151–178.

[CP96] A. B. Compagnoni and B. C. Pierce, *Higher-Order Intersection Types and Multiple Inheritance*, Math. Struct. Com. Sci. **6** (1996), no. 5, 469–501.

[Cpp11] *ISO/IEC 14882:2011: Information Technology – Programming Languages – C++*, 2011.

[CSB+11] H. Chafi, A. K. Sujeeth, K. J. Brown, HJ Lee, A. R. Atreya, and K. Olukotun, *A Domain-Specific Approach to Heterogeneous Parallelism*, Proc. 16^{th} ACM Symp. Princ. & Practice

Parallel Prog. (San Antonio, TX, USA), PPoPP '11, ACM, 2011, pp. 35–46.

[CV13] W. Cazzola and E. Vacchi, *Neverlang 2 — Componentised Language Development for the JVM*, Proc. 12^{th} Int. Conf. Soft. Composition (W. Binder, E. Bodden, and W. Löwe, eds.), Lect. Notes in Comp. Sci., vol. 8088, Springer, June 2013, pp. 17–32.

[DFS09] A. Dijkstra, J. Fokker, and S. D. Swierstra, *The Architecture of the Utrecht Haskell Compiler*, Proc. 2^{nd} ACM SIGPLAN Symp. on Haskell (Edinburgh, Scotland), ACM, 2009, pp. 93–104.

[Dij05] A. Dijkstra, *Stepping through Haskell*, Ph.D. thesis, Utrecht Uni., Dept. Inf. & Comp. Sci., 2005.

[DMMZ12] 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.

[EGR12] S. Erdweg, P. G. Giarrusso, and T. Rendel, *Language Composition Untangled*, Proc. 5^{th} Int. W. Lang. Descr. Tools & App., ACM, 2012, pp. 49–56.

[EH04] T. Ekman and G. Hedin, *Rewritable Reference Attributed Grammars*, Proc. 18^{th} Euro. Conf. Obj.-Oriented Progr. (M. Odersky, ed.), Lect. Notes in Comp. Sci., vol. 3086, Springer, June 2004, pp. 144–169.

[EH07] _____, *The JastAdd Extensible Java Compiler*, Proc. 22^{nd} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (R. P. Gabriel, D. F. Bacon, C. Videira Lopes, and G. L. Steele Jr., eds.), ACM, October 2007, pp. 1–18.

[Ell12] C. Ellison, *A Formal Semantics of C with Applications*, Ph.D. thesis, Dept. Comp. Sci., Coll. Eng., U. Illinois at Urbana-Champaign, July 2012.

[ER12] C. Ellison and G. Roşu, *An Executable Formal Semantics of C with Applications*, Proc. 39^{th} ACM SIGPLAN-SIGACT Symp. Princ. Prog. Lang. (Philadelphia, Pennsylvania, USA)

(J. Field and M. Hicks, eds.), ACM, January 2012, pp. 533–544.

[ER13] S. Erdweg and F. Rieger, *A Framework for Extensible Languages*, Proc. Int. 12^{th} Conf. Gener. Prog. & Component Eng. (J. Järvi and C. Kästner, eds.), ACM, October 2013, pp. 3–12.

[ERKO11] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann, *SugarJ: Library-based Syntactic Language Extensibility*, Proc. 26^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (Oregon, Portland, USA) (Lopes C. V. and K. Fisher, eds.), ACM, October 2011, pp. 391–406.

[Ern99] E. Ernst, *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*, Ph.D. thesis, Dept. Comp. Sci. , Uni. Århus, Århus, Denmark, 1999.

[Ern01] _____, *Family Polymorphism*, Proc. 15^{th} Euro. Conf. Obj.-Oriented Progr. (J. Lindskov Knudsen, ed.), Lect. Notes in Comp. Sci., vol. 2072, Springer, June 2001, pp. 303–326.

[Ern03] _____, *Higher-Order Hierarchies*, Proc. 17^{th} Euro. Conf. Obj.-Oriented Progr. (L. Cardelli, ed.), Lect. Notes in Comp. Sci., vol. 2743, Springer, July 2003, pp. 303–328.

[Ern06] _____, *Reconciling Virtual Classes with Genericity*, Modular Prog. Lang., 7^{th} Joint Modular Lang. Conf., Proc. (D. E. Lightfoot and C. A. Szyperski, eds.), Lect. Notes in Comp. Sci., vol. 4228, Springer, September 2006, pp. 57–72.

[ERRO12] 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.

[EvdSV+13] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, Kelly S., A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning, *The State of the Art in Language Workbenches – Conclusions from the Language Workbench Challenge*, Proc.

6^{th} Int. Conf. Soft. Lang. Eng. (M. Erwig and van Wyk E. Paige, R. F., eds.), Lect. Notes in Comp. Sci., vol. 8225, Springer, October 2013, pp. 197–217.

[EVMV14] S. Erdweg, V. A. Vergu, M. Mezini, and E. Visser, *Modular Specification and Dynamic Enforcement of Syntactic Language Constraints when Generating Code*, Proc. 13^{th} Int. Conf. Modularity (Lugano, Switzerland) (W. Binder, E. Ernst, A. Peternier, and R. Hirschfeld, eds.), ACM, 2014, pp. 241–252.

[FCMR04] A. Farzan, F. Chen, J. Meseguer, and G. Roşu, *Formal Analysis of Java Programs in JavaFAN*, Proc. Comp.-aided Verification, Lect. Notes in Comp. Sci., vol. 3114, 2004, pp. 501–505.

[FFF09] M. Felleisen, R. B. Findler, and M. Flatt, *Semantics Engineering with PLT Redex*, MIT Press, 2009.

[FH92] M. Felleisen and R. Hieb, *A Revised Report on the Syntactic Theories of Sequential Control and State*, Theo. Comp. Sci. **103** (1992), no. 2, 235–271.

[FM10] A. C. J. Fox and M. O. Myreen, *A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture*, ITP (Edinburgh, UK) (M. Kaufmann and L. C. Paulson, eds.), Lect. Notes in Comp. Sci., vol. 6172, Springer, July 2010, pp. 243–258.

[For04] B. Ford, *Parsing Expression Grammars: a Recognition-Based Syntactic Foundation*, Proc. 31^{st} ACM SIGPLAN-SIGACT Symp. Princ. Prog. Lang. (Venice, Italy), ACM, 2004, pp. 111–122.

[Fox03] A. C. J. Fox, *Formal Specification and Verification of ARM6*, TPHOLs (D. A. Basin and B. Wolff, eds.), Lect. Notes in Comp. Sci., vol. 2758, Springer, September 2003, pp. 25–40.

[FŞAL+12] 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.), Elec. Notes Theo. Comp. Sci., 2012.

[Gar00] J. Garrigue, *Code Reuse through Polymorphic Variants*, W. Found. Soft. Eng., no. 25, 2000, pp. 93–100.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addisson-Wesley Professional Computing Series, AW Professional, October 1994.

[GHVW07] J. Gao, M. P. E. Heimdahl, and E. Van Wyk, *Flexible and Extensible Notations for Modeling Languages*, 10^{th} Int. Conf. Fund. Approaches Soft. Eng., Proc. (M. B. Dwyer and A. Lopes, eds.), Lect. Notes in Comp. Sci., vol. 4422, Springer, March 2007, pp. 102–116.

[GMO07] V. Gasiunas, M. Mezini, and K. Ostermann, *Dependent Classes*, Proc. 22^{nd} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (R. P. Gabriel, D. F. Bacon, C. Videira Lopes, and G. L. Steele Jr., eds.), ACM, October 2007, pp. 133–152.

[Gri06] R. Grimm, *Better Extensibility through Modular Syntax*, Proc. ACM SIGPLAN Conf. Prog. Lang. Design & Impl. (Ottawa, Ontario, Canada), ACM, 2006, pp. 38–51.

[Gro03] Object Management Group, *OMG Unified Modeling Language Specification, version 1.5*, March 2003, available online at www.rational.com/uml/resources/documentation/.

[Gut78] Guttag, J. V. and Horning, J. J., *The Algebraic Specification of Abstract Data Types*, Acta Informatica **10** (1978), 27–52.

[Hae09] S. H. Haeri, *Observational Equivalence between Lazy Programs in Presence of Selective Strictness*, Technical report, MuSemantik, 2009, available from the author.

[Hae10] _____, *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.

[Hae13] _____, *A New Operational Semantics for Distributed Lazy Evaluation*, Technical Report TR2013-1, Inst. Soft. Sys., Hamburg U. Tech., April 2013, http://www.sts.tuhh.de/tech-reports/2013/haer2013.pdf.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers, *The Syntax Definition Formalism SDF – Reference Manual*, SIGPLAN Not. **24** (1989), no. 11, 43–75.

[HK95] J. Heering and P. Klint, *The Prehistory of* ASF+SDF *(1980–1984)*, Proc. ASF+SDF'95 W. Gener. Tools Alg. Spec. (M. G. J. van den Brand, A. van Deursen, T. B. Dinesh, J. Kamperman, and E. Visser, eds.), Technical Report P9504, Programming Research Group, University of Amsterdam, May 1995, pp. 1–4.

[HO10] C. Hofer and K. Ostermann, *Modular Domain-Specific Language Components in Scala*, Proc. Int. 9^{th} Conf. Gener. Prog. & Component Eng. (Eindhoven, The Netherlands) (E. Visser and J. Järvi, eds.), ACM, 2010, pp. 83–92.

[HORM08] C. Hofer, K. Ostermann, T. Rendel, and A. Moors, *Polymorphic Embedding of DSLs*, Proc. Int. 7^{th} Conf. Gener. Prog. & Component Eng. (Nashville, TN, USA) (Y. Smaragdakis and J. G. Siek, eds.), ACM, October 2008, pp. 137–148.

[HS14] S. H. Haeri and S. Schupp, *Distributed Lazy Evaluation: A Big-Step Mechanised Semantics*, 4PAD'14, IEEE, February 2014, pp. 751–755.

[HSY06] D. Hardin, E. Smith, and B. Young, *A Robust Machine Code Proof Framework for Highly Secure Applications*, Proc. 6^{th} Int. W. ACL2 Theorem Prover & its Appl. (Seattle, WA) (P. Manolios and M. Wilding, eds.), ACM Digital Library, August 2006.

[IPW01] A. Igarashi, B. C. Pierce, and P. Wadler, *Featherweight Java: A Minimal Core Calculus for Java and GJ*, Trans. Prog. Lang. & Sys. **23** (2001), no. 3, 396–450.

[Jak14] F. Jakob, *SugarScala: Syntactic Extensibility for Scala*, Master's thesis, Software Technology Group, Technische Universität Darmstadt, Darmstadt, Germany, 2014.

[Jay04] C. B. Jay, *The Pattern Calculus*, ACM Trans. Prog. Lang. Sys. **26** (2004), no. 6, 911–937.

[Jay09] ———, *Pattern Calculus: Computing with Functions and Structures*, Springer, 2009.

[JC94] C. B. Jay and J. R. B. Cockett, *Shapely Types and Shape Polymorphism*, Prog. Lang. & Sys., 5^{th} Euro. Symp. Prog.,

Proc. (D. Sannella, ed.), Lect. Notes in Comp. Sci., vol. 788, Springer, April 1994, pp. 302–316.

[JDAO04] P. Jolly, S. Drossopoulou, C. Anderson, and K. Ostermann, *Simple Dependent Types: Concord*, Proc. 6^{th} W. Formal Tech. Java-like Prog., June 2004, Tech. Rep. nr. NIII-R0426, Uni. Nijmegen.

[JK09] B. Jay and D. Kesner, *First-Class Patterns*, J. Func. Prog. **19** (2009), no. 2, 191–225.

[JLMaRY09] J. Jeuring, S. Leather, J. P. Magalhães, and A. Rodriguez Yakushev, *Libraries for Generic Programming in* Haskell, Advanced Functional Programming (P. Koopman, R. Plasmeijer, and D. Swierstra, eds.), Lect. Notes in Comp. Sci., vol. 5832, Springer, 2009, pp. 165–229.

[JMS10] 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.

[JV13] B. Jay and J. Vergara, *Growing a Language in Pattern Calculus*, 7^{th} Int. Symp. Theo. Aspects Soft. Eng., 2013, pp. 233–240.

[KCD+12] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler, *Run Your Research: On the Effectiveness of Lightweight Mechanization*, Proc. 39^{th} ACM SIGPLAN-SIGACT Symp. Princ. Prog. Lang., ACM, 2012, pp. 285–296.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, *An Overview of AspectJ*, Proc. 15^{th} Euro. Conf. Obj.-Oriented Progr. (J. Lindskov Knudsen, ed.), Lect. Notes in Comp. Sci., vol. 2072, Springer, June 2001, pp. 327–353.

[KM09] R. Knöll and M. Mezini, *π: A Pattern Language*, Proc. 24^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (Orlando, Florida, USA), ACM, 2009, pp. 503–522.

[KRV10] H. Krahn, B. Rumpe, and S. Völkel, *MontiCore: a Framework for Compositional Development of Domain Specific Languages*, Int. J. Soft. Tools for Tech. Transfer (STTT) **12** (2010), no. 5, 353–372.

[KT08a] T. Kamina and T. Tamai, *A Design and Implementation of Lightweight Constructs for Mutually Extensible Components*, Submitted Jan. 2008 to Elsevier.

[KT08b] ———, *Lightweight Dependent Classes*, Proc. Int. 7^{th} Conf. Gener. Prog. & Component Eng. (Nashville, TN, USA) (Y. Smaragdakis and J. G. Siek, eds.), ACM, October 2008, pp. 113–124.

[KV10] 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. (Reno/Tahoe, Nevada, USA) (W. R. Cook, S. Clarke, and M. C. Rinard, eds.), ACM, October 2010, pp. 444–463.

[KvdSV11] P. Klint, T. van der Storm, and J. Vinju, *EASY Metaprogramming with Rascal*, Gener. & Transform. Techs Soft. Eng. III (J. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, eds.), Lect. Notes in Comp. Sci., vol. 6491, Springer Berlin/Heidelberg, 2011, pp. 222–289.

[KW94] U. Kastens and M. W. Waite, *Modularity and Reusability in Attribute Grammars*, Acta Informatica **31** (1994), 601–627.

[Lau93] J. Launchbury, *A Natural Semantics for Lazy Evaluation*, Proc. 20^{th} ACM SIGPLAN-SIGACT Symp. Princ. Prog. Lang., ACM, 1993, pp. 144–154.

[LE13] F. Lorenzen and S. Erdweg, *Modular and Automated Type-Soundness Verification for Language Extensions*, Proc. 18^{th} ACM SIGPLAN Int. Conf. Func. Prog. (Boston, MA, USA) (G. Morrisett and T. Uustalu, eds.), ACM, September 2013, pp. 331–342.

[Lis87] B. Liskov, *Keynote Address – Data Abstraction and Hierarchy*, ACM SIGPLAN Not. **23** (1987), no. 5, 17–34.

[LOMP05] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí, *Parallel Functional Programming in Eden*, J. Func. Prog. **15** (2005), no. 3, 431–475.

[LP03] M. Y. Levin and B. C. Pierce, *TinkerType: A Language for Playing with Formal Systems*, J. Func. Prog. **13** (2003), no. 2, 295–316.

[Lum05] M. Lumpe, *A Lambda Calculus with Forms*, Proc. 1^{st} Int. Conf. Soft. Composition (T. Gschwind, U. Aßmann, and O. Nierstrasz, eds.), Lect. Notes in Comp. Sci., vol. 3628, Springer, April 2005, pp. 83–98.

[Lum08] ———, *Growing a Language: The* GLoo *Perspective*, Proc. 7^{th} Int. Conf. Soft. Composition (C. Pautasso and É. Tanter, eds.), Lect. Notes in Comp. Sci., vol. 4954, Springer, 2008, pp. 1–19.

[Mar00] R. C. Martin, *Design Principles and Design Patterns*, 2000, online article available from the ObjectMentor website.

[McI69] M. D. McIlroy, *Mass Produced Software Components*, Proc. NATO Conf. Soft. Eng. (New York, US), Petrocelli/Charter, 1969, pp. 138–155.

[ME10] A. B. Madsen and E. Ernst, *Revisiting Parametric Types and Virtual Classes*, Obj., Models, Components, Patterns, 48^{th} Int. Conf. Proc. (J. Vitek, ed.), Lect. Notes in Comp. Sci., vol. 6141, Springer, June 2010, pp. 233–252.

[Mes92] J. Meseguer, *Conditional Rewriting Logic as a Unified Model of Concurrency*, Theo. Comp. Sci. **96** (1992), no. 1, 73–155.

[MH07] E. Magnusson and G. Hedin, *Circular Reference Attributed Grammars: Their Evaluation and Applications*, Sci. Comp. Prog. **68** (2007), no. 1, 21–37.

[MO03] M. Mezini and K. Ostermann, *Conquering Aspects with Caesar*, Proc. 2^{nd} Int. Conf. Aspect-Oriented Soft. Dev. (New York, NY, USA) (W. G. Griswold, ed.), ACM, March 2003, pp. 90–99.

[Mos08] P. D. Mosses, *Component-Based Description of Programming Languages*, BCS Int. Acad. Conf. (E. Gelenbe, S. Abramsky, and V. Sassone, eds.), Brit. Comp. Soc., 2008, pp. 275–286.

[Mos13] ———, *Semantics of Programming Languages: Using* ASF+SDF, Sci. Comp. Prog. (2013), To Appear.

[MRHO12] A. Moors, T. Rompf, P. Haller, and M. Odersky, *Scala-Virtualized*, Proc. 21^{st} ACM SIGPLAN W. Partial Eval. & Prog. Manip. (Philadelphia, PA, USA), ACM, 2012, pp. 117–120.

[MV14] P. D. Mosses and F. Vesely, *FunKons: Component-Based Semantics in K*, W. Rewrit. Logic & App., Lect. Notes in Comp. Sci., Springer, 2014, To appear.

[NCM03] N. Nystrom, M. R. Clarkson, and A. C. Myers, *Polyglot: An Extensible Compiler Framework for Java*, 12^{th} Int. Conf. Compiler Constr., Lect. Notes in Comp. Sci., vol. 2622, Springer-Verlag, April 2003, pp. 138–152.

[NCM04] N. Nystrom, S. Chong, and A. C. Myers, *Scalable Extensibility via Nested Inheritance*, Proc. 19^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (J. M. Vlissides and D. C. Schmidt, eds.), ACM, October 2004, pp. 99–115.

[NQM06] N. Nystrom, X. Qi, and A. C. Myers, *J&: Nested Intersection for Scalable Software Composition*, Proc. 21^{st} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (Portland, Oregon, USA), ACM, 2006, pp. 21–36.

[Ö12] J. Öqvist, *Implementation of Java 7 Features in an Extensible Compiler*, Master's thesis, Dept. Comp. Sci., Lund Uni., 2012.

[OBZNS11] S. Owens, P. Böhm, F. Zappa Nardelli, and P. Sewell, *Lem: A Lightweight Tool for Heavyweight Semantics*, Interactive Theorem Proving: Second International Conference, ITP 2011 (M. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, eds.), Lect. Notes in Comp. Sci., vol. 6898, Springer, August 2011, pp. 363–369.

[OC12] B. C. d. S. Oliveira and W. R. Cook, *Extensibility for the Masses – Practical Extensibility with Object Algebras*, Proc. 26^{th} Euro. Conf. Obj.-Oriented Progr., Lect. Notes in Comp. Sci., vol. 7313, Springer, 2012, pp. 2–27.

[Oli09] B. C. d. S. Oliveira, *Modular Visitor Components*, Proc. 23^{rd} Euro. Conf. Obj.-Oriented Progr., Lect. Notes in Comp. Sci., vol. 5653, Springer, 2009, pp. 269–293.

[OSV11] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-step Guide*, 2^{nd} ed., Artima Inc., November 2011.

[OvdSLC13] B. C. d. S. Oliveira, T. van der Storm, A. Loh, and W. R. Cook, *Feature-Oriented Programming with Object Algebras*, Proc. 27^{th} Euro. Conf. Obj.-Oriented Progr. (Montpellier, France) (Giuseppe Castagna, ed.), Lect. Notes in Comp. Sci., vol. 7920, Springer, 2013, pp. 27–51.

[OZ05a] M. Odersky and M. Zenger, *Independently Extensible Solutions to the Expression Problem*, Proc. Int. W. Found. Obj.-Oriented Lang., January 2005.

[OZ05b] ———, *Scalable Component Abstractions*, Proc. 20^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (San Diego, CA, USA), ACM, 2005, pp. 41–57.

[Pre97] C. Prehofer, *Feature-Oriented Programming: A Fresh Look at Objects*, Proc. 11^{th} Euro. Conf. Obj.-Oriented Progr. (Jyväskylä, Finland) (M. Aksit and S. Matsuoka, eds.), Lect. Notes in Comp. Sci., vol. 1241, 1997, pp. 419–443.

[Pre09] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 7^{th} ed., McGraw-Hill, 2009.

[QM09] X. Qi and A. C. Myers, *Sharing Classes between Families*, Proc. ACM SIGPLAN Conf. Prog. Lang. Design & Impl. (M. Hind and A. Diwan, eds.), ACM, June 2009, pp. 281–292.

[QM10] ———, *Homogeneous Family Sharing*, Proc. 25^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (Reno/Tahoe, Nevada, USA) (W. R. Cook, S. Clarke, and M. C. Rinard, eds.), ACM, October 2010, pp. 520–538.

[RBO14] T. Rendel, J. Brachthäuser, and K. Ostermann, *From Object Algebras to Attribute Grammars*, 2014, To Appear.

[Rey75] J. C. Reynolds, *User-Defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction*, New Dir. in Algo. Lang. (S. A. Schuman, ed.), 1975, pp. 157–168.

[RFŞ10] G. Roşu and T. Florin Şerbănuţă, *An Overview of the K Semantic Framework*, J. Logic & Alg. Prog. **79** (2010), no. 6, 397–434.

[RL11] V. Rusu and D. Lucanu, *K Semantics for OCL — a Proposal for a Formal Definition for OCL*, K Workshop, 2011.

[RMRHV06] D. Rebernak, M. Mernik, P. Rangel Henriques, and M. J. Varanda, *AspectLISA: An Aspect-Oriented Compiler Construction System Based on Attribute Grammars*, Elec. Notes Theo. Comp. Sci. **164** (2006), no. 2, 37–53.

[RO10] T. Rompf and M. Odersky, *Lightweight Modular Staging: a Pragmatic Approach to Runtime Code Generation and Compiled DSLs*, Proc. Int. 9^{th} Conf. Gener. Prog. & Component Eng. (Eindhoven, The Netherlands), ACM, 2010, pp. 127–136.

[RSK10] M. Rosenmüller, N. Siegmund, and M. Kuhlemann, *Improving Reuse of Component Families by Generating Component Hierarchies*, Proc. 2^{nd} Int. W. Feature-Oriented Soft. Dev. (New York, NY, USA), ACM, October 2010, pp. 57–64.

[Sin08] F.-R. Sinot, *Complete Laziness: a Natural Semantics*, Elec. Notes Theo. Comp. Sci. **204** (2008), 129–145.

[SIV08] C. Saito, A. Igarashi, and M. Viroli, *Lightweight Family Polymorphism*, J. Func. Prog. **18** (2008), no. 3, 285–331.

[SKV13] A. M. Sloane, L. C. L. Kats, and E. Visser, *A Pure Embedding of Attribute Grammars*, Sci. Comp. Prog. **78** (2013), no. 10, 1752–1769.

[Slo11] A. Sloane, *Lightweight Language Processing in Kiama*, Gener. & Transform. Techs Soft. Eng. III (2011), 408–425.

[Som11] I. Sommerville, *Software Engineering*, 9^{th} ed., Addison Wesley, 2011.

[SPA12] P. Shyamshankar, Z. Palmer, and Y. Ahmad, *K3: Language Design for Building Multi-Platform, Domain-Specific Runtimes*, 1^{st} Int. W. Cross-Model Lang. Design and Impl., 2012.

[SRB+13] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun, *Composition and Reuse with Compiled Domain-Specific Languages*, Proc. 27^{th} Euro. Conf. Obj.-Oriented Progr. (G. Castagna, ed.), Lecture Notes in Computer Science, vol. 7920, Springer, July 2013, pp. 52–78.

[SS13] C. Schwaab and J. G. Siek, *Modular Type-Safety Proofs in Agda*, Proc. 7^{th} W. Prog. Lang. Meets Prog. Verific. (M. Might, D. Van Horn, A. Abel, and T. Sheard, eds.), ACM, January 2013, pp. 3–12.

[SSZN+09] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, *The Semantics of x86-CC Multiprocessor Machine Code*, Proc. ACM SIGPLAN Princ. Prog. Lang., 2009, pp. 379–391.

[SVW09] A. Schwerdfeger and E. Van Wyk, *Verifiable Composition of Deterministic Grammars*, Proc. ACM SIGPLAN Conf. Prog. Lang. Design & Impl. (M. Hind and A. Diwan, eds.), ACM, June 2009.

[Swi08] W. Swierstra, *Data Types à la Carte*, J. Func. Prog. **18** (2008), no. 4, 423–436.

[SZNO+10] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša, *Ott: Effective Tool Support for the Working Semanticist*, J. Func. Prog. **20** (2010), no. 1, 71–122.

[Szy96] C. Szyperski, *Independently Extensible Systems – Software Engineering Potential and Challenges*, Proc. 19^{th} Australasian Comp. Sci. Conf., 1996.

[TAK+14] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, *A Classification and Survey of Analysis Strategies for Software Product Lines*, ACM Computing Surveys (2014), accepted for publication Jan 30, 2014.

[TBB+11] S. T. Taft, J. Bloch, R. Bocchino, S. Burckhardt, H. Chafi, R. Cox, B. R. Gaster, G. L. Steele Jr., and D. Ungar, *Multicore, manycore, and cloud computing: is a new programming language paradigm required?*, Companion to Proc. 26^{th} ACM Int. Conf. Obj.-Oriented Prog. Sys. Lang. & Appl. (C. Videira Lopes and K. Fisher, eds.), ACM, October 2011, pp. 165–170.

[TBKC07] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook, *Safe Composition of Product Lines*, Proc. Int. 6^{th} Conf. Gener. Prog. & Component Eng. (Salzburg, Austria) (C. Consel and J. L. Lawall, eds.), ACM, 2007, pp. 95–104.

[THLP98] P.W. Trinder, K. Hammond, H-W. Loidl, and S. Peyton Jones, *Algorithm + Strategy = Parallelism*, J. Func. Prog. **8** (1998), no. 1, 23–60.

[Tor04] M. Torgersen, *The Expression Problem Revisited*, Proc. 18^{th} Euro. Conf. Obj.-Oriented Progr. (Oslo (Norway)) (M. Odersky, ed.), Lect. Notes in Comp. Sci., vol. 3086, June 2004, pp. 123–143.

[Tra95] W. Tracz, *Third International Conference on Software Reuse Summary*, SIGSOFT Soft. Eng. Notes **20** (1995), no. 2, 21–22.

[vd07] 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.

[vdBvdS11] J. van den Bos and T. van der Storm, *Bringing Domain-Specific Languages to Digital Forensics*, Proc. 33^{rd} Int. Conf. Soft. Eng (Waikiki, Honolulu, HI, USA) (R. N. Taylor, H. Gall, and N. Medvidovic, eds.), ACM, May 2011, pp. 671–680.

[vHD+01] A. van Deursen, J. Heering, H. A. De Jong, M. De Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. J. Vinju, E. Visser, and J. Visser, *The* ASF+SDF *Meta-Environment: A Component-Based Language Development Environment*,

10^{th} Int. Conf. Compiler Constr. (Genova, Italy) (R. Wilhelm, ed.), Lect. Notes in Comp. Sci., vol. 2027, Springer, April 2001, pp. 365–370.

[Vis04] E. Visser, *Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9*, Domain-Specific Prog. Gener. (C. Lengauer et al., eds.), Lect. Notes in Comp. Sci., vol. 3016, Spinger-Verlag, June 2004, pp. 216–238.

[VWBGK10] 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.

[VWBH06] E. Van Wyk, D. Bodin, and P. Huntington, *Adding Syntax and Static Analysis to Libraries via Extensible Compilers and Language Extensions*, Proc. 2^{nd} Int. W. Library-Centric Soft. Design (A Priesnitz and S. Schupp, eds.), Chalmers University of Technology and Göteborg University, October 2006, Technical Report No. 06-18, pp. 35–44.

[VWdMBK02] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski, *Forwarding in Attribute Grammars for Modular Language Design*, 11^{th} Int. Conf. Compiler Constr., Lect. Notes in Comp. Sci., vol. 2304, Springer-Verlag, 2002, pp. 128–142.

[VWKBS07] E. Van Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger, *Attribute Grammar-Based Language Extensions for Java*, Proc. 21^{st} Euro. Conf. Obj.-Oriented Progr. (E. Ernst, ed.), Lect. Notes in Comp. Sci., vol. 4609, Springer, July 2007, pp. 575–599.

[Wad98] P. Wadler, *The Expression Problem*, Java Genericity Mailing List, November 1998.

[WT11] S. Wehr and P. Thiemann, *JavaGI: The Interaction of Type Classes with Interfaces and Inheritance*, ACM Trans. Prog. Lang. Syst. **33** (2011), no. 4, 12.

[ZO01] M. Zenger and M. Odersky, *Extensible Algebraic Datatypes with Defaults*, Proc. 6^{th} ACM SIGPLAN Int. Conf. Func. Prog. (Firenze (Florence), Italy), ACM, 2001, pp. 241–252.