KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Functional Techniques for Representing and Specifying Software

for ad hoc polymorphism, context-free grammars, meta-programming and effect polymorphism

Dominique Devriese

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

January 2014

Functional Techniques for Representing and Specifying Software

for ad hoc polymorphism, context-free grammars, meta-programming and effect polymorphism

Dominique DEVRIESE

Examination committee:
Prof. dr. ir. Ludo Froyen, chair
Prof. dr. ir. Frank Piessens, supervisor
Prof. dr. Dave Clarke, co-supervisor
Prof. dr. ir. Wouter Joosen
Prof. dr. ir. Eric Steegmans
Prof. dr. ir. Tom Schrijvers
(UGent)
Prof. dr. Tim Sheard
(Portland State University)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

January 2014

ISBN 978-94-6018-773-5
D/2014/7515/4

Preface

This thesis is the result of 4 years of work, during which I have learned a lot about software and scientific research. I have been very fortunate to work under the excellent guidance of Frank Piessens, whose advice has consistently and strongly improved the quality of my work and often pointed me in interesting directions. I am especially grateful that he has given me the freedom and responsibility to develop my own ideas, even when they moved out of his core area of expertise.

During the work on this thesis, I have enjoyed collaborating with several excellent researchers. I want to explicitly mention Nataliia Bielova, Fabio Massacci, Gilles Barthe, Juan Manuel Crespo, Exequiel Rivas, Willem De Groef, Nick Nikiforakis, Tom Reynaert, Lieven Desmet, Ilya Sergey, Dave Clarke, Matthew Might, Jan Midtgaard, David Darais, Tom Schrijvers, Thomas Winant and Jesper Cockx. I have enjoyed my role as teaching assistant for the courses "Informatica-Werktuigen", "Objectgericht Programmeren", "Formele Systemen en hun Toepassingen", "Declaratieve Talen" and "Comparative Programming Languages", as well as the mentoring of several good or excellent master thesis students: Niels Gabriëls, Willem De Groef, Tom Reynaert, Vincent Goelen, Thomas Winant, Jesper Cockx, Ramses De Norre, Bob Reynders, Sophie Van Herck and Jannes Van Ussel (several of whom are now working on their own PhD).

I am also grateful to the members of my PhD jury: the members of the supervisory committee Frank Piessens, Dave Clarke, Wouter Joosen and Eric Steegmans, the external members Tom Schrijvers and Tim Sheard and the chairman Ludo Froyen.

In the last years, I have much enjoyed many casual discussions about life and work. I want to explicitly mention Pieter Agten, Dave Clarke, Jesper Cockx, Wouter De Borger, Willem De Groef, Philippe De Ryck, Thomas Delaet, Lieven Desmet, Fatih Gey, Thomas Heyman, Bart Jacobs, Jef Maeriën, Dimiter

Milushev, Adriaan Moors, Nick Nikiforakis, Job Noorman, Steven Op de beeck, Marco Patrignani, Willem Penninckx, Pieter Philippaerts, José Proença, Ansar Rafique, Rula Sayaf, Ilya Sergey, Jan Smans, Raoul Strackx, Steven Van Acker, Dimitri Van Landuyt, Mathy Vanhoef, Dries Vanoverberghe, Gijs Vanspauwen, Frédéric Vogels, Marko van Dooren, Alexander van den Berghe and Thomas Winant. I have also very much enjoyed the occasional football games between colleagues.

Finally, this work would not have been possible without the financial support of

- The BCRYPT project, funded by the Interuniversity Attraction Poles of the Belgian Science Policy.
- The Research Foundation - Flanders (FWO), which has granted me a PhD fellowship.
- The EU FP7 project NESSoS.

Abstract

All software is represented as source code in a programming language. The programming language defines the meaning or *semantics* of the code, for example, its operational behaviour. Computational source code is often accompanied by additional *specifications* that define how the source code should be interpreted or provide additional information about the software's semantics. They make it possible for programmers to express and verify that their software has the intended semantics and to express inter-component semantic assumptions. Good representations and specifications of software components are crucial for efficiently producing software that is reliable, efficient and secure, and for preserving these qualities during the software's evolution.

Many types of software components and their desired semantic properties can be challenging to represent and specify. In this work, I contribute novel functional techniques for the representation and specification of four types of software components:

- *Ad hoc polymorphic functions*: functions whose behaviour depends on the types of their arguments or result. I present *instance arguments*: a type system extension for representing ad hoc polymorphic functions in the dependently-typed programming language Agda. Compared to existing proposals, instance arguments do not introduce an additional structuring concept and ad hoc polymorphic functions using them are fully first-class. Furthermore, they avoid introducing a separate, powerful form of type-level computation and existing Agda libraries using records do not need modifications to be used with them. My implementation has been part of Agda since version 2.3.0 and I demonstrate a variety of applications of instance arguments.
- *Context-free grammars*: a standard way to define the syntax of formal languages. I present a technique for representing context-free grammars in an embedded domain-specific language (EDSL). It avoids the restrictions

of existing parser combinator libraries using a novel explicit representation of recursion based on advanced type system techniques in the Haskell programming language. As a byproduct, grammars are decoupled from sets of semantic actions. On the flip side, the approach requires the grammar author to provide a type- and value-level encoding of the grammar's domain and I can provide only a limited form of constructs like `many`. I demonstrate the approach with five grammar algorithms, including a pretty-printer, a reachability analysis, a translation of quantified recursive constructs to standard ones, and an implementation of the left-corner grammar transform. This work forms the basis of my `grammar-combinators` parsing library.[1]

- *Meta-programs*: programs that generate or manipulate other programs. I present a novel set of meta-programming primitives for use in a dependently-typed functional language. The meta-programs' types provide strong and precise guarantees about the meta-programs' termination, correctness and completeness. The system supports type-safe construction and analysis of terms, types and typing contexts. Unlike alternative approaches, meta-programs are written in the same style as normal programs and use the language's standard functional computational model. I formalise the new meta-programming primitives, implement them as an extension of Agda, and provide evidence of usefulness by means of two compelling applications in the fields of datatype-generic programming and proof tactics.

- *Effect polymorphic software*: programs that support arbitrary implementations of effectful APIs and only produce effects through those implementations. Static effectful APIs and global mutable state in object-oriented programming languages make it hard to modularly control effects. Object-capability (OC) languages solve this by enforcing that effects can only be triggered by components that hold a reference to the object representing the capability to do so. I study this encapsulation of effects through a formal translation to a typed functional calculus with higher-ranked polymorphism (I use a subset of Haskell for presentation). Based on an informal view of effect-polymorphism as the fundamental feature of OC languages, I translate an OC calculus to effect-polymorphic Haskell code, i.e. computations that are universally quantified over the monad in which they produce effects. The types of my translations assert the object-capability property and I can show and exploit this using Reynolds' parametricity theorem. An important new insight is that current OC languages and formalisations leave one effect implicitly available to all code, without a capability: the allocation of new mutable state; adding a

[1] `http://projects.haskell.org/grammar-combinators`

capability for it has important theoretical and practical advantages. My work establishes a new link between object-capability languages and the well-studied fields of functional programming and denotational semantics.

Beknopte samenvatting

Alle software wordt voorgesteld als broncode in een programmeertaal. De programmeertaal bepaalt de betekenis of *semantiek* van de code, bijvoorbeeld haar operationeel gedrag. Uitvoerbare broncode wordt vaak vergezeld door bijkomende *specificaties* die definiëren hoe de broncode moet geïnterpreteerd worden of extra informatie geven over de semantiek van de software. Ze maken het mogelijk voor programmeurs om uit te drukken en te verifiëren dat hun software de bedoelde semantiek heeft en om semantische veronderstellingen uit te drukken tussen componenten onderling. Goede representaties en specificaties van softwarecomponenten zijn cruciaal om efficiënt software te produceren die betrouwbaar, performant en veilig is en om deze kwaliteiten te blijven garanderen tijdens de evolutie van de software.

Veel soorten softwarecomponenten en hun gewenste semantische eigenschappen zijn moeilijk voor te stellen of te specifiëren. In deze thesis draag ik nieuwe functionele technieken bij voor de voorstelling en specificatie van vier soorten softwarecomponenten:

- *ad hoc-polymorfe functies*: functies wiens gedrag afhangt van het type van hun argumenten of resultaat. Ik stel *instance arguments* voor: een typesysteemuitbreiding voor het voorstellen van ad hoc-polymorfe functies in de afhankelijk-getypeerde (dependently-typed) programmeertaal Agda. In vergelijking met bestaande voorstellen, introduceren instance arguments geen bijkomend structureel concept en zijn functies die instance arguments gebruiken volwaardig ondersteund. Daarenboven vermijden ze de toevoeging van een afzonderlijke en krachtige vorm van uitvoering op type-niveau. Bestaande Agda libraries die records gebruiken hoeven ook niet te worden aangepast om ze te gebruiken. Mijn implementatie is een deel van Agda sinds versie 2.3.0 en ik toon verschillende toepassingen van instance arguments.

- *Contextloze grammatica's*: een standaardtechniek voor het definiëren van formele talen. Ik stel een techniek voor om contextloze grammatica's voor te stellen in een ingebedde domein-specifieke programmeertaal (EDSL). De techniek vermijdt de beperkingen van bestaande parser-combinator bibliotheken met behulp van een vernieuwende expliciete voorstelling van recursie, gebaseerd op geavanceerde typesysteemtechnieken in de Haskell programmeertaal. Als bijkomend voordeel worden grammatica's losgekoppeld van semantische acties. Aan de andere kant vereist de aanpak dat de grammatica-auteur een codering voorziet van het domein van de grammatica op type- en waardeniveau, en kan ik slechts een beperkte versie voorzien van constructies als `many`. Ik demonstreer de aanpak met vijf grammatica-algoritmes, o.a. een weergeefalgoritme (pretty-printer), een bereikbaarheidsanalyse, een vertaling van gekwantificeerde recursieconstructies naar gewone en een implementatie van de linkerhoek- of left corner-grammaticatransformatie. Dit werk vormt de basis van mijn `grammar-combinators` parsing-bibliotheek.[2]

- *Meta-programma's*: programma's die andere programma's genereren of manipuleren. Ik stel vernieuwende meta-programmeerprimitieven voor voor gebruik in een afhankelijk-getypeerde functionele programmeertaal. De types van de meta-programma's leveren sterke en precieze garanties over hun terminatie, correctheid en volledigheid. Het systeem ondersteunt type-veilige constructie en analyse van termen, types en typeringscontexten. In tegenstelling tot andere aanpakken, worden ze geschreven in dezelfde stijl als normale programma's en gebruiken ze het standaard functioneel uitvoeringsmodel van de programmeertaal. Ik formaliseer de nieuwe meta-programmeerprimitieven, implementeer ze als een uitbreiding van Agda en lever bewijs van hun nut met twee overtuigende toepassingen in de domeinen van datatype-generisch programmeren en bewijstactieken (proof tactics).

- *Effect-polymorfe software.* Statische APIs met neveneffecten en globale wijzigbare toestandinformatie in object-georiënteerde programmeertalen maken het moeilijk om effecten modulair te controleren. Object-bekwaamheidsprogrammeertalen (Object-Capability of OC) lossen dit probleem op door af te dwingen dat effecten alleen kunnen veroorzaakt worden door componenten die een referentie hebben naar het object dat de bekwaamheid voorstelt om dat te doen. Ik bestudeer deze inkapseling van effecten door een formele vertaling naar een getypeerde functionele calculus met hogere-rang polymorfisme (ik gebruik een beperkte versie van Haskell voor de presentatie). Op basis van een informele visie van effect-polymorfisme als het fundamentele kenmerk van OC-talen,

[2] `http://projects.haskell.org/grammar-combinators`

vertaal ik een OC calculus naar effect-polymorfe Haskell code, d.w.z. berekeningen die uniform gekwantificeerd zijn over de monade waarin ze effecten veroorzaken. De types van onze vertalingen drukken de object-bekwaamheidseigenschap uit en we kunnen dit aantonen en gebruiken met behulp van Reynolds' parametriciteitsstelling. Een belangrijk nieuw inzicht is dat bestaande OC talen en formalisaties één effect impliciet beschikbaar laten: de allocatie van nieuwe wijzigbare toestandsinformatie; hier een bekwaamheid voor toevoegen heeft belangrijke theoretische en praktische voordelen. Mijn werk opent een belangrijke nieuwe verbinding tussen object-bekwaamheidsprogrammeertalen en de goed onderzochte velden van functioneel programmeren en denotationele semantieken.

Contents

List of Figures

Chapter 1

Introduction

Software today has become vital in many areas of the economy and society and it is applied to increasingly complex tasks. Programs are becoming larger and more complex than before, and if software fails, the consequences can be increasingly severe. Recent technological developments in other areas (e.g. smartphones, voice-over-IP, 3D-printing, driverless cars, engine control computers or enterprise resource planning software) increasingly rely on complex computers and software, making it unlikely for this trend to change in the foreseeable future.

Software engineers have so far been able to manage this increasing complexity and keep software failure reasonably uncommon. To achieve this, they rely on software development techniques that facilitate the construction of large software and ensure its correctness, specifically the use of higher-level programming languages and libraries, modular development and various specification techniques.

In this thesis, I study representations and specifications for software components in four domains: ad hoc polymorphism, context-free grammars, meta-programming and effect polymorphism. To introduce this work, I present a helicopter overview of the domain of software development in Section 1.1 and in Section 1.2, I zoom in on the techniques I build upon. My aim is to give the reader an idea of how my contributions fit into a larger context. After that, I will present the four domains of interest and the contributions I have made to each of them in Section 1.3. Section 1.4 gives a short overview of research I worked on that is not presented in this thesis and Section 1.5 gives an overview of the following chapters.

1.1 A semantic view on software development

All software is constructed as *source code* in a *programming language*. The language comes equipped with a *semantics* that defines the meaning of the code. This often consists of the operational behaviour of programs for arbitrary input on a hypothetical computer. A programmer always works with a certain language semantics in mind, even if it is just an informal understanding. His goal is to construct a program whose semantics satisfies certain requirements.

To make this more concrete, consider the example of the first computer program ever written: a series of instructions that female British mathematician Ada Lovelace[1] constructed for Charles Babbage's *Analytical Engine* (see Figure 1.1) [128, 77]. She wrote the program so that under the mechanical functioning of Babbage's device, the numbers of a certain mathematical series would be printed. For this example, the programming language was the convention under which the machine accepted the instructions (metal cards with holes punched), the semantics was the behaviour produced by the machine for the instructions and Lovelace's requirement for her program's semantics was that it should print the desired series.

While Lovelace constructed her program and made it satisfy the requirements by reasoning directly about the mechanical behaviour of the machine, software engineers today have a wide range of techniques and tools at their disposal. At a high level, we can distinguish three kinds of techniques: *higher-level* languages and libraries, *modular* programming and *specifications*. In the next section, I consider each in more detail.

1.1.1 Higher-level languages and libraries

By *higher-level programming*, I mean programming in a language whose semantics is simpler and more abstract than the target semantics, and then using a tool or library to map the simpler semantics to the target semantics. One approach uses a higher-level programming language. An example of this is Dijkstra's recommendation in 1968 that programming languages should not include a `goto` statement [62]. Instead, they should only provide *structured* conditional and looping constructs like `if` and `while` and a compiler should translate them into assembly language, in such a way that the meaning of a higher-level program corresponds to the behaviour of its translation.

The removal of `goto` makes the programming language semantically simpler. A standard understanding of programs using the primitive requires an in-memory

[1] By a coincidence of history, she was the daughter of famous poet Lord Byron.

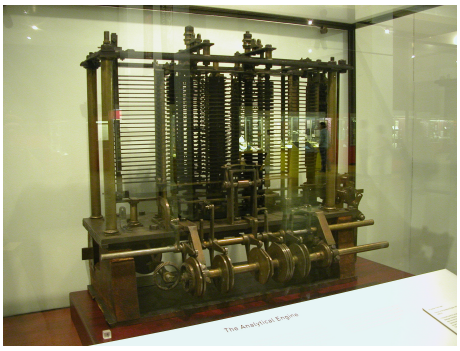

Figure 1.1: Trial model of a part of Babbage's Analytical Engine, defining the first programming language and its semantics. Displayed in the Science Museum (London). © Bruno Barral, Wikimedia Commons, CC-BY-SA [26].

```
int fact1(int n) {
  int res = 1;
 loop:
  res *= n--;
  if (n > 1)
    goto loop;
  return res;
}
```

```
int fact2(int n) {
  int res = 1;
  while( n > 1 )
    res *= n--;
  return res;
}
```

```
int fact3(int n) {
  if(n <= 1)
    return 1;
  else
    return n * fact3(n-1);
}
```

Figure 1.2: Three C implementations of the factorial function.

representation of code and a machine executing according to an instruction pointer. Dijkstra's structured constructs can instead be given simpler, more abstract semantics as a reduction system where the code does not need to be encoded in memory (see e.g. [187]). Scott and Strachey have shown that the programs can also be given a *denotational semantics*; a more mathematical meaning as elements of continuous function spaces between complete partial orders of values (*Scott-Strachey domains*) [205].

As a concrete example, Figure 1.2 shows three C implementations of a factorial function, the first one using `goto` and the second using the `while` construct instead. The second code snippet can be given meaning as sequentially executed instructions for a machine with just mutable memory, with each command semantically independent of its predecessors and successors. The `goto` command in the first implementation however, can only be given meaning in the context of the remaining instructions, making the commands semantically interdependent.

It's hard to understand the first implementation without imagining a machine executing the in-memory code according to an instruction pointer.

Since then, Dijkstra's recommendation to drop the `goto` command has gained wide acceptance. In his 1978 Turing award acceptance speech, John Backus promoted another form of higher-level language that is still far from universally accepted. Backus argued that programming languages should move away from the *von Neumann style* where program semantics are defined by their behaviour on abstract machines with mutable state. He instead recommended (what we now know as) *purely functional programming languages*, where programs can be given simpler meaning using a memory-less reduction-system [10]. The denotational semantics of purely functional programs is also simpler; they directly denote mathematical functions that do not have the entire state of the memory before and after execution as an input argument and result value. As a concrete example, consider the third implementation in Figure 1.2, which simply corresponds to the recursive definition of the factorial.

Compilers from high-level languages to assembly are not the only example of higher-level programming. *Parser generators* are separate tools that receive a *grammar* as input. A grammar describes a *formal language*, a possibly infinite set of strings of characters in a certain *alphabet*. Grammars are written in a special-purpose language, often based on the *Backus-Naur-Form* (BNF) [109, 176]. A parser generator translates such a grammar to the source code for a program that *parses* sentences from a textual input, translating them to a more structured form. Parser generators are a form of higher-level programming as the grammar has a direct, more abstract meaning through the formal language it defines. The generated parser semantically corresponds to that abstract meaning in the sense that it parses only the strings in the formal language.

Even within a single language, we can apply higher-level programming. Writing a regular expression (a grammar for a very simple formal language) and applying it using a library is in principle not very different from using a parser generator. Even when we use a simple structured representation of data, such as a URL record with fields for the protocol, domain name and path, we are essentially defining a range of terms with an abstract semantics that can be translated to more primitive semantics (e.g. a string representation of URLs).

It is interesting to contrast the approach of compiling programs in a higher-level language to a lower-level one (as done by compilers and parser generators) to the approach of embedding a higher-level language by representing it *inside* the lower-level language (as used by regular expression libraries and abstract data types). The second type has been called *Embedded Domain Specific Languages (EDSLs)* and they have important advantages. The higher-level language can cheaply reuse features of the host language, e.g. abstraction

mechanisms, specification techniques, debuggers. Those features are then often more consistent across both languages. Nevertheless, the lower-level language needs to be sufficiently powerful to accurately represent terms in the EDSL. Finding such a representation can be non-trivial, especially for complex EDSLs. One of the contributions of this text is the proposal and study of an accurate representation of a grammar EDSL in the Haskell programming language (see Chapter 3). EDSLs are often constrained to use less natural syntax within the host language, but many programming languages include features designed explicitly to facilitate this, e.g. Racket [224] and Scala [158].

1.1.2 Modular programming

The second approach to facilitate the construction of programs with a desired semantics is *modular programming*: the sub-division of programs into *components*. For the purpose of this introduction, I define components as those parts of a program that can be given a semantics independently from the rest of the program.[2] This can include, for example, procedures, functions, classes, traits, aspects and executables. The advantage of sub-dividing programs into components is that they can be developed separately. This enables parallel software development in large teams and the reuse of components over time and across different programs. In many scenarios, program components can even fall under the control of (partially) distrusting parties.

The structuring of a program into components is known as its *architecture* or *design*. Good software architecture makes it easy to develop a program and ensure its intended semantics, for example, by defining reusable, general and high-quality components and by structuring the program so that a semantic property of the program follows from the semantics of only one (or a few) components (*separation of concerns*).

Generally, it's advantageous for components' semantics to be as independent of each other as possible (*loose coupling*). Some programming languages inherently make it hard to achieve such semantic independence. Consider, for example, LaTeX, which (among other problems) provides only dynamically scoped variables, making it hard to construct components (in this case macros) that are unaffected by the behaviour of other components [122]. Similarly, JavaScript permits non-locally overriding the behaviour of primitive classes like `String` (see e.g. [149]) and C and C++ suffer from buffer overflows (see e.g. [4, 45]).

[2] So I will not use more restricted definitions of the terms *modular programming* and *components* that are used in some other fields of computer science.

Different programming styles or *paradigms* exist which prescribe the use of components with certain types of individual semantics and certain ways of combining the components' semantics. I have already mentioned functional programming, prescribing components that denote mathematical functions and function composition, application and abstraction as composition constructs [10].

Of relevance to this thesis is the widely-used paradigm of object-oriented programming [163, 114]. It recommends the use of *objects*: semantic entities that couple a private piece of mutable state with a set of methods acting on that state. Components called classes describe the private state and behaviour of a set of objects; the class's *instances*. One of the contributions of this thesis is the identification and formalisation of the principle of *effect polymorphism*, a property that is central to object-oriented programming (see Chapter 5). Some other programming paradigms that I do not go into are *aspect-oriented programming* [117] and *logic programming* [236].

Although it is beneficial to make components' semantics maximally independent, components are by nature intended to be combined and even well-designed components have to rely on semantic assumptions about the components they are combined with. For example, a function that sorts a list according to a comparison criterion may only work correctly if the criterion produces a boolean result for any two elements of the list, and if these results correspond to a transitive and antisymmetric relation. Also, a web server may contain a component responsible for network communication that is not permitted to access the computer's storage directly. As such, even though abstract reusable components can help in the construction of software with an intended semantics, the approach also requires the programmer to ensure that inter-component semantic assumptions are fulfilled. In systems that contain components under the control of different parties, components may actively try to violate other components' assumptions in order to influence their behaviour; such situations are considered in the field of *computer security*.

1.1.3 Specifications

A third general approach to ensure programs and their components have a desired semantics is the use of *specifications*: explicit statements in the source code of their semantic properties and their semantic assumptions about components they can be combined with. They can help ensure that a program satisfies semantic requirements and find inter-component assumption violations even before components are actually combined together. This is especially important in large, multi-developer and multi-party systems and for programs that remain used and under development for long periods of time. In a high-level language,

semantic properties expressed by specifications can often help the compiler derive better translations to the target language. Another purpose of specifications is documentation: an additional explanation by the programmer of the intended semantics of his code.

There exists a wide variety of techniques for the specification of programs and components: informal documentation, design and naming conventions, unit tests, static type systems, contract systems and program logics. For illustration, Figure 1.3 shows specifications of sorting functions in a variety of programming languages using a number of different specification techniques: types and semi-formal documentation in Java (1.3a), a Java unit test (1.3b), C types and pre- and postconditions in a separation logic-based program logic (1.3c), types and the pre- and postconditions of Eiffel's contract system (1.3d) and a semantic proof in a dependent type system (1.3e) (you are not expected to understand all of them in detail). Desirable qualities of specification techniques are

Precision Specifications should unambiguously correspond to the intended semantic property.

Validation It should be possible to verify during development that components' semantics correspond to specifications. Preferably validation should be automatic, so that it can be repeated cheaply during development.

Modularity Validation of a component's properties should depend only on that component's source code.

Overhead It should require little additional programmer effort to formulate properties and make them validatable.

Unfortunately, there is no universal specification technique that achieves all these goals for any given component. Rather, there is a variety of techniques that can be used to state certain types of properties for certain types of components. Therefore, software engineers need to consider the important semantic properties of the components defined and the specification techniques that can be used to specify them when defining a program's architecture.

Some types of specifications are limited in one or more of the above dimensions. Informal specification techniques like natural-language documentation (used, for example, in Figure 1.3a), naming and design conventions are imprecise and cannot be automatically validated. *Unit tests*, although precise, modular and automatically validatable can specify only a finite subset of the typically infinite semantics of a program, so that the specified properties are typically much less general than desired. The unit test in Figure 1.3b, for example, only specifies correct sorting behaviour for one particular list of strings. In (dynamic)

```
class Collections {
  /**
   * Sorts the specified list according to the natural ordering of its elements. ...
   * @param list the list to be sorted.
   * @throws ClassCastException if the list contains elements that are not
   *                mutually comparable (for example, strings and integers).
   */
  public static <T extends Comparable<? super T>> void sort(List<T> list) {
    ... }
}
```

(a) A sort function in Java, specified using a parametrically polymorphic *type* annotation with subtyping constraints, and additionally *natural language and semi-formal documentation.* They specify the intuitive behaviour and the types of lists supported by the function.

```
class CollectionsTest {
  @Test public void testSort() {
    String[] arr = {"c", "a", "b"};
    String[] expected = {"a", "b", "c"};
    sort(Arrays.asList(arr));
    assertArrayEquals(expected, arr);
  }
}
```

(b) A *unit test* specifying the behaviour of the Java sort function in (a). It requires that a particular array of strings should be correctly sorted.

```
void sort(int n, int *xs)
    //@ requires int_array(xs, n);
    //@ ensures int_array(xs, n);
```

(c) A specification of an integer-only C sort function in VeriFast [107]. The specification is in a *program logic* extending Reynolds' separation logic [195]. It requires that other components cannot simultaneously access the input array and specifies that the length of the array does not change.

```
deferred class DS_SORTABLE [G] inherit DS_CONTAINER [G]
  feature sort (a_sorter: DS_SORTER [G])
              -- Sort container using a_sorter's algorithm.
    require a_sorter_not_void: a_sorter /= Void
    ensure sorted: sorted (a_sorter)
end
```

(d) An Eiffel specification of a sort function with pre- and postconditions, part of Eiffel's *contract system.* Eiffel optionally checks this contract during execution.

```
sort : List ℕ → List ℕ
sort−correct : (as : List ℕ) → Sorted (sort as)
```

(e) A sort function for natural number lists in Agda. The *sort-correct* proof specifies that *sort* must produce *sorted* lists, using Agda's dependent *type system.*

Figure 1.3: Specification of sort functions, in different languages using different types of specifications.

contract systems (e.g. Figure 1.3d), validation is limited to the verification of pre- and postconditions *during* executions of the program, leaving specifications unvalidated for executions not seen during development. All of this does not mean that these techniques are useless in practice, far from it. They are often the most cost-effective way to specify certain properties. However, it is an ongoing challenge for computer scientists to develop techniques that can do better. It is in this process that this thesis should be situated.

Some very powerful specification techniques are based on *program logics*. They descend from techniques proposed in the late 1960s, particularly by Hoare [97]. For proving properties about imperative programs, Hoare proposed to formulate logical properties expected to hold before and after the execution of a command. A command C together with its pre- and postconditions P and Q is written as $\{P\}\ c\ \{Q\}$ and known as a *Hoare triple*. Hoare defined a set of axiomatic rules for proving the validity of a command's specifications based on the command's syntactic form and the pre- and postconditions of its components (*Hoare logic*). In 2002, Reynolds presented an extension of Hoare logic, called *separation logic*, that essentially allows components to require sole access to parts of the heap memory. For example, the separation logic specification of the C sort function in Figure 1.3c says (implicitly) that it will not crash if the argument pointer identifies an integer array in memory and no other thread can simultaneously access the array. Separation logic enables modular reasoning about components that use shared data structures [195].

In this text, I will not use program logics but a different class of techniques known as *type systems*, used in Figures 1.3a, 1.3c, 1.3d and 1.3e. I will present type systems in more detail further on.

1.1.4 The techniques used

The three above approaches to software development (higher-level programming, modular programming and specifications) should not be interpreted as mutually exclusive. Much to the contrary, they are complementary and the use of one class of techniques may increase the need for another. Consider, for example, higher-level programming. Higher-level EDSLs rely on the definition of sub-languages with semantic sub-domains. Specifications can play a role in delineating such sub-domains and specifying properties of their terms, as we will see for the example of a Haskell grammar EDSL in Chapter 3. Higher-level languages may also introduce new types of components and new ways of combining them, as we will also see with the decoupling of grammars and sets of semantic actions in the grammar EDSL. Similarly, modular programs with many components typically feature many inter-component assumptions for specifications to keep

track of. In Chapter 5, for example, I present specifications for properties about the side-effects that imperative components produce.

This thesis deals with representing and specifying components in four domains. I have chosen to use a number of techniques for representing those components with the desired semantics and specifying their correct behaviour.

A first important choice is that I exclusively work with purely functional programming languages, specifically Haskell and Agda. In three cases (Chapters 2, 3 and 4), this is because the concerned components' semantics did not have an inherently imperative nature, so that their semantics can be best approximated using a functional style. In the fourth case, I consider effectful components in an imperative object-oriented programming language. However, to study the discipline that is imposed on side-effects in these languages, I build crucially on the use of *monads*: a representation for code with side-effects used in the purely functional programming language Haskell.

Another choice is that I make use of only one type of specification technique: type systems. However, I employ Haskell and Agda: two programming languages whose type systems are significantly different, and I rely on some of their most powerful features. In fact, before I present the contributions of this thesis in Section 1.3, I present type systems in the next section, focusing specifically on the features that my work builds on.

1.2 Type systems

Type systems are a widely used specification technique, based on the idea of assigning a *type* to any component and value in the source code. This is done in a *compositional* way, i.e. the type of a composed component is a combination of the types of the components.

The type of a value or component expresses semantic facts about it. Like for specification techniques in general, many different type systems exist that can express different types of properties about different types of components. Some type systems can only express simple properties such as "n is an integer number" or "g is a function mapping booleans to strings" (e.g. the C type system used in Figure 1.3c). Others can express more complex properties as well: general properties like the parametricity of polymorphic functions (used in Figure 1.3a and 1.3d), which I will discuss further on, but also more specific properties, for example, related to the aliasing of object references [42], the respect of information flow security policies [200], or the correct behaviour of meta-programs (see Chapter 4). Very often, a programming language is

designed to support one or several programming paradigms, and its type system is designed to express desirable properties about the components recommended by the paradigm (e.g. pure functions in Haskell, classes in Java or Eiffel).

Many programming languages include a type system as a vital part of the language. This is, for example, the case in C [115], Java [87] or Haskell [101]. Often, the type of a component or value is not considered as an optional specification of some of its semantic properties, but rather as an essential part of its definition, determining its meaning *together* with the implementation. This view is known as *intrinsic* or *Church-style typing*, as opposed to the *extrinsic* or *Curry-style typing* that I have assumed before. One of the contributions of this text applies to *ad hoc polymorphic functions*, which go especially far in this direction. They are functions that behave differently when used at different types (see Chapter 2).

It is interesting to note that the origin of type systems lies in the field of mathematical logic, where they were used to prevent logical paradoxes in the foundations of mathematics, notably by Russell [240] and Ramsey [190]. Alonzo Church proposed to use his *lambda calculus* as a notation for Ramsey's logic, producing the *simply typed lambda calculus* [39], an important system of reference in the theory of type systems. So, from the beginning types were used as a notation for propositions in mathematical logic, and terms in the typed lambda calculus were considered as a notation for proofs of those theorems. It is only later that the lambda calculus came to be used as a core part of a programming language, with terms representing programs and types specifying their behaviour, an idea notably promoted by McCarthy [146], Landin [123] and Morris [159]. The observation that terms and types in typed lambda calculi can be semantically interpreted both as programs/specifications *and* proofs/propositions is known as the Curry-Howard correspondence [99, 46]. This correspondence carries through for many useful type systems and corresponding logics and it has motivated developments on both sides.

It is not my intention here to give a full overview of the research area of type systems. I will instead restrict further discussion to three important topics relevant to this thesis: the polymorphic lambda calculus (also known as System F) and its parametric polymorphism, specifications for side-effects, specifically using monads in Haskell, and dependent type systems, specifically in Agda.

1.2.1 System F and parametric polymorphism

Girard and Reynolds' *polymorphic lambda calculus* or *System F* [80, 193] is a typed lambda calculus that allows the definition of *polymorphic functions*:[3] functions with a single general type that can be instantiated to an entire family of more specific types. A prototypical example is the identity function $\lambda x.x$, which can be assigned the type $a \to a$ for any type a. This property can be made explicit in System F with the polymorphic type $\forall a.a \to a$. The sort functions in Figures 1.3a and 1.3d also both have polymorphic types indicating they can sort lists with arbitrary types of elements (although both require a way of comparing values of the type).

A polymorphic type must be usable at any type in the family and there is no way for a polymorphic function to find out which type it is being used at. As a result of this, its semantics can be quite strongly restricted; for example, the above polymorphic type $\forall a.a \to a$ is *only* inhabited by the identity function. These restrictions give rise to a formal property called *parametricity* that was first described by Reynolds [192]. It states that a component with a polymorphic type satisfies a certain semantic property that can be derived from the type.

Parametric polymorphism has found its way to several programming languages: most famously ML [154], but also Haskell [101] and even Java [104]. Many extensions of System F have seen the light in the process: for example, type-level functions in Girard's System F_ω [80], type equality proofs in System F_C [217], or bounded quantification to support subtyping [29]. As a remarkable example of keeping theory close to practice, the Glasgow Haskell Compiler uses the System F_C calculus as a typed intermediate language.

We should note that besides the study of parametric polymorphism, parametricity is also crucial in the study of *abstract data types (ADTs)*. These are user-defined types with an internal representation and an external interface of functions implemented in terms of the internal representation. Programs using only the external interface can be considered as programs that are (parametrically) polymorphic in the representation type of the ADT, so that parametricity states properties that they satisfy.

The property of parametricity has itself been the subject of a lot of research. After proposing the parametricity theorem using a set-theoretic semantics of polymorphic functions [192], Reynolds proved that such a semantics cannot in fact exist [194]. Semanticists have since then proven versions of parametricity

[3] Note: we use the word polymorphism to indicate *parametric polymorphism*. This should not be confused with the concept of subtype polymorphism, which is also sometimes called polymorphism in the context of object-oriented programming. In OO languages, (parametrically) polymorphic functions are often referred to as *generics*.

of System F and many extensions, based on (often category-theoretical) denotational semantics, e.g. [6, 14, 15, 230]. I recommend, for example, the relatively accessible text by Atkey on a parametric model for System F_ω with some kind-level extensions [6]. Wadler has written a text popularising parametricity towards a larger audience of functional programmers using the slogan "Theorems for free" [231], demonstrating some compelling applications of the property. Voigtländer has more recently done the same for parametricity applied to monadic type functors in System F_ω [228]. In chapter 5, I heavily rely on parametricity in System F_ω to prove the properties of effect polymorphic functions, which I propose to use in the semantics of certain object-oriented languages.

A downside of polymorphic type systems is that their validation often requires quite a bit of programmer effort in the form of type annotations, i.e. explicit assignments of types to components and terms or explicit annotations of the type that a polymorphic function is being used at. In order to reduce this effort, *type inference* (also known as *type reconstruction*) algorithms can be used to automatically deduce (part of) the required type information from the implementations. Hindley [94] and Milner [153] independently described a polymorphic type system with an automatic type inference algorithm based on the idea of *unification*. In most current type systems, fully automatic inference is not a design goal, not in the least because it cannot be achieved for many of the more powerful type systems (e.g. System F [239]). Nevertheless, even in such languages, techniques based on Hindley-Milner type inference are in wide use to reduce the amount of annotations needed to specify the types of programs. This is, for example, the case in the Glasgow Haskell Compiler (GHC) [229].

1.2.2 Specifying side-effects, specifically in Haskell

The purpose of most software includes performing *input/output (I/O)*: interacting with the world outside the computer, e.g. communication with other computers over a network, communication with the user over a display or keyboard/mouse, or the operation of devices connected with the computer, such as printers, GPS receivers or robot arms. Software performing input/output is very often represented using imperative statements with *side-effects*. Semantically, these statements correspond to procedures whose execution — in addition to returning a value based on their input arguments — triggers some form of additional *side-effect*, e.g. an interaction with the outside world. For example, the Java statement `System.out.println("Hello world!")` is a command that simply returns `void`, but additionally prints a message on a textual output channel. The approach of using statements with side-effects originates from low-level machine languages, where it is standard.

Side-effects are often not limited to interactions with the outside world per se. They may also consist of interactions with computer memory (reading or writing of mutable memory cells), interactions with other execution threads inside the program or with other programs (e.g. synchronisation primitives like semaphores) etc. In some languages, statements may even modify or add source code to the program currently executing, either in a textual representation (e.g. JavaScript `eval` [196]) or in a bytecode or assembly language (e.g. code injection in C [245] or code generation in Java [25]). With a little more imagination, even the consumption of computation time [49], or potentially causing the non-termination of a program [28, 50] have been treated as side-effects.

With some exceptions, the semantics of side-effecting statements are badly supported by specification techniques. For example, most type systems for languages with side-effecting statements leave the side-effects of components completely free; for example, a C function[4] or ML function that accepts no arguments and returns `void` may perform arbitrary side-effects during execution. This is a bad state of affairs, because in practical development, side-effects are often the topic of important architectural requirements of components (e.g. all network access is encrypted, the front-end can only access the database through the business layer), of crucial security assumptions between components (e.g. an advertisement in an online mail application cannot inspect e-mails [131]) or functional properties of components or applications (e.g. all printed documents should contain a company header).

In the Haskell programming language, the type system does allow specifications about the side-effects of programs, through its use of monads. Monads are a structure from *category theory* (a highly abstract field of mathematics) that Moggi [156] proposed to use for representing the semantics of statements with side-effects of different types. He showed that monads presented a unifying *notion of computation*, covering side-effects like non-determinism, mutable state, exceptions, partiality etc. The idea is to represent a computation that yields a result of a type `a` as a value of type `m a`, where `m` is a monad representing the kind of effects produced by the computation. The definitional requirements for monads in category theory correspond nicely to natural requirements for such an `m`: that there should be a way to embed values as non-side-effecting (or *pure*) computations and a way to sequence two computations where the second depends on the first's result. Additionally, standard monad requirements imply certain natural axioms about combinations of these primitives.

Haskell uses monads directly to represent side-effecting computations [232, 233] and supports the *do-notation* [232] to write such computations in a natural

[4] Procedures in C are called "functions", arguably a misnomer since they are semantically quite different from mathematical functions.

syntax. This approach allows it to strictly enforce purely functional programming by default but still present an imperative API with side-effects for when it is needed. More specifically, Haskell defines the built-in monad `IO` in which essentially any side-effect can be produced (including input/output, mutable state, invoking C functions and concurrency [180]), and the GHC compiler adds more restricted monads for working with mutable memory (the `ST` monad [124]) and software transactional memory (the `STM` monad [92]). Additionally, programmers can define custom parameterised types and declare them to be monads by instantiating the `Monad` type class [110].

Haskell essentially uses monads to separate purely functional components from imperative computations. This was at least partly motivated by its semantics of pure expressions, which uses a non-strict evaluation order incompatible with side-effecting procedures [100]. Nevertheless, it is now clear that Haskell's type-level separation of pure expressions is a more beneficial characteristic than its lazy evaluation order, because it creates the possibility of specifying the side-effects (or at least the absence of side-effects). Evidence for this is the more recent appearance of pure but strict programming languages (e.g. Chlipala's Ur [36] or Czaplicki and Chong's Elm [47]).

Besides Haskell's use of monads, we have to mention some other exceptions to the rule that specification systems do not cover side-effects. In Reynolds' separation logic, which we have mentioned before, specifications imply a strict upper bound on the mutable memory locations that a component can access. There is promising research on applying separation logic for other kinds of side-effects as well (e.g. [98]), but this is much less mature than its use for mutable memory locations. Current separation logic tools typically leave I/O behaviour of components unspecified, e.g. VeriFast [107].

Another way to specify the side-effects of imperative procedures that we do not go into is based on *type and effect systems*. They are a form of type systems where a statement's type includes a bound on the side-effects it may produce [129]. In a sense, this definition encompasses Haskell's use of monads and the two approaches are indeed closely related, see e.g. [234]. Effect types can describe many kinds of side-effects, including locking behaviour in concurrent programs [71], aliasing in object-oriented programs [42] and region-based automatic memory management [225]. Java's *checked exceptions* [87] are another well-known example. In the literature, effect type systems are most often used as part of an automated analysis, especially in the context of region-based memory management [225]. However, they are also useful for specification purposes.

A rather different approach at managing the side-effects of components is not based on specifications but rather on the definition of components with

```
main : Signal Element
main = lift clock seconds

clock : Float → Element
clock t =
  collage 400 400
    [ filled black (circle 110)
    , hand red 100 (t/60)
    , hand grey 100 (t/(60∗60))
    , hand grey 60 (t/(60∗60∗12)) ]

hand clr len t =
  traced (solid clr) (segment (0,0)
           (len ∗ sin a, len ∗ cos a))
  where a = turns t
```

(a) An implementation using a Functional Reactive Programming (FRP) style in the Elm programming language. `main` is semantically a time-varying GUI element, constructed by applying the pure function `clock` to a time-varying `seconds` signal.

```
everySecond(function (t,canvas) {
  canvas.drawCircle(black,0,0,110);

  drawHand(red, 100, t/60);
  drawHand(grey, 100, t/(60∗60));
  drawHand(grey, 60, t/(60∗60∗12));

  function drawHand(clr, len, t) {
    var a = turns(t);
    canvas.drawLine(clr,0,0,
      len∗cos(a), len∗sin(a));
  }});
```

(b) An implementation using an imperative callback in JavaScript. The code is understood semantically by reasoning about the side-effects of the event handler when executed in response to a timer event.

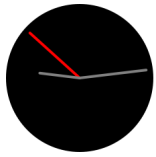

Figure 1.4: Two implementations of an analogue clock, contrasting a higher-level functional reactive programming representation to one based on side-effecting statements. A screenshot of the clock is shown at the bottom.

alternative semantics. This includes the idea of *algebraic effects*: imperative procedures whose side-effects can be reinterpreted by handlers [188]. Another example is *functional reactive programming (FRP)* [69], a paradigm that recommends components that semantically represent *signals* or *behaviours* (time-varying values) and *event streams* (channels on which values arrive at specific times). For illustration, Figure 1.4 contrasts two implementations of an analogue clock, one using FRP and the other using a more traditional imperative callback-based approach. Semantic properties can more easily be checked for the first implementation, for example, the property that the current screen only depends on the current time. For the second implementation, we need to inspect the side-effects produced by each statement in the handler, while for the first,

this already follows from just the implementation of `main`. Functional reactive programming was first proposed for graphical user interface libraries [69] but its use is investigated for other applications as well (e.g. [74]). The paradigm is being popularised in the object-oriented community by Maier and Odersky under the slogan "Deprecating the observer pattern" [135].

In Section 1.3.4, I will explain that the notion of *encapsulation* in object-oriented languages is also a way of managing the side-effects of components and that the principle is generalised to arbitrary effects in *object-capability languages*. In Chapter 5, I will use Haskell monads as a crucial tool for studying a property called *effect polymorphism* that I propose as a defining property for such languages.

1.2.3 Agda: dependent types

Dependent type systems are advanced type systems that first appear in the work of Martin-Löf [140], Howard [99] and in de Bruijn's AUTOMATH system [55], with further important contributions by Coquand and Huet [44] and Luo [130]. Characteristically, a dependent type system allows types that mention values or components in such a way that the semantic property expressed by a type depends on the semantics of those values or components. This makes the type system so powerful that arbitrary semantic properties of components can be expressed and — under the Curry-Howard correspondence — arbitrary mathematical propositions can be formulated and their proofs verified.

Consider, for example, again the sort function from Figure 1.3e:

$$\text{sort} : \text{List}\ \mathbb{N} \to \text{List}\ \mathbb{N}$$
$$\text{sort}-\text{correct} : (\text{as} : \text{List}\ \mathbb{N}) \to \text{Sorted}\ (\text{sort}\ \text{as})$$

The function sort has a standard non-dependent type but sort-correct's type is more unusual. For any input list as, sort-correct returns a value of type Sorted (sort as): a proof that the result of applying the function sort to as produces a sorted list. We have omitted implementations, but for an appropriate definition of the Sorted predicate, sort-correct can only be implemented if sort always produces sorted lists.

This example shows that dependent types allow one to express both programs and proofs about those programs' semantics. On a much larger scale, this is demonstrated by Leroy's provably correct C compiler CompCert [126]. However, dependent types can also be used for formal verification of purely mathematical theory. This was demonstrated — on an equally large scale as CompCert —

by Gonthier with verified proofs of the Four Colour Theorem about planar graphs [83] and the Feit-Thompson theorem from group theory [84].

Unfortunately, a dependent type system imposes heavy requirements on the programming language. In order for the type system to be consistent and its type-checking problem decidable, its functions must typically all be total (i.e. terminate successfully for all input). However, such restrictions do *not* make the language Turing-incomplete; non-terminating programs can still be modelled, albeit with a bit more care and effort [28, 50, 142].

Programming languages with dependent types include Coq (which Leroy and Gonthier used for their developments) [223], a language which was mainly designed as a mathematical proof assistant. Several other dependently-typed languages have been developed in a similar spirit, but with different design choices, for example, the Edinburgh Logical Framework [91] and its descendants (e.g. [184, 185]), NuPRL [43], Isabelle [179] and Alf [134]. Other languages with dependent types are aimed more towards use as a programming language, e.g. Cayenne [7], Idris [21], Epigram [142] and Agda [161]. Recently, other programming languages are introducing what one might call lightweight dependent types: enriched type systems that share some features with dependently-typed systems, without paying the cost of making the entire language total, e.g. Dependent ML [242], Ur [36], F* [218], Ωmega [207], Scala [165] and recent Haskell extensions like McBride's SHE [141], data kinds and kind-polymorphism [244] and closed type families [68]. Lightweight dependent types have also been applied to assembly languages [243].

In Chapters 2 and 4, I work with the dependently-typed programming language Agda [161]. As mentioned, it is mainly intended for use as a programming language, although its design does not exclude use as a proof assistant. Some of its characteristic features are its use of *dependent pattern matching* [44, 82] and its support for advanced constructs like induction-recursion [67]. In Chapter 2, I propose an extension of Agda's type system in order to support ad hoc polymorphic functions. In Chapter 4, I propose novel dependently-typed primitives to represent and specify strongly typed meta-programs in Agda.

1.3 Representing and specifying components in four domains

The contributions of this thesis are the definition and study of novel functional techniques for the representation and specification of software. Specifically, we present techniques for four types of components and associated semantic

properties: ad hoc polymorphism, context-free grammars, meta-programming and effect-polymorphism. I will introduce them in the next four sections.

1.3.1 Ad hoc polymorphism

The first contribution of this text is the design and implementation of *instance arguments* in Agda. They can be used for assigning types to functions that are *ad hoc polymorphic*.

A component is *ad hoc polymorphic* or *overloaded* when it can be used at many different types, but may behave differently for each of them. A typical example is an equality decision procedure: a function that receives two arguments and checks whether they are *equal*. Such a function is ad hoc polymorphic, since equality can mean different things for different types of values. Often, ad hoc polymorphic primitives are not or cannot be implemented for certain types. Equality, for example, cannot be decided for functions on an infinite domain. To make sure that ad hoc polymorphic functions are not used at unsupported types, some type systems provide special support. They provide types specifying the ad hoc polymorphic primitives used by a component, so that it is possible during development to check for each component separately that ad hoc polymorphic functions are only ever used at supported types.

Although Standard ML's type system applied this idea specifically for the overloaded built-in equality decision procedure with its `eqtype` variables [155], Haskell was the first to extend the idea to arbitrary overloaded functions with its *type classes* [235]. Later, alternative designs with different characteristics have been made for programming languages like Scala [167], Coq [213], C++ [211, 212] and other languages.

Instance arguments are another alternative to type classes and these other features, which we have designed and implemented for use in the Agda programming language. Compared to existing systems, instance arguments are based on some novel choices. Most importantly, they avoid the introduction of type classes as a new kind of data type, ad hoc polymorphic functions are treated as first class components and they avoid the introduction of a separate powerful type-level computation primitive. More details follow in Chapter 2.

1.3.2 Context-free grammars

A *formal language* is a collection of strings in a certain alphabet, e.g., the set of well-formed English sentences or the set of syntactically valid source code in a programming language. Formal languages can be defined using *formal*

```
Line   → Expr EOF
Expr   → Expr '+' Term
       → Term
Term   → Term '*' Factor
       → Factor
Factor → '(' Expr ')'
       → Digit+
Digit  → '0' | '1' | '2' | ... | '8' | '9'
```

Figure 1.5: A context-free grammar for arithmetic expressions.

grammars. The concepts have applications in the study of natural languages, but for our purposes their most important application is the definition of the syntax of programming languages. Backus was the first to describe the syntax of ALGOL 58 as a formal language, in a grammar notation now known as the *Backus-Naur-Form (BNF)* [11].

Formal grammars come in different shapes and forms, but an important class are the *context-free grammars*, first studied by Chomsky [38]. Such grammars define strings in the language using a set of recursive equations or *rules*, where each rule defines how a certain sub-structure of the language (a *non-terminal*) is obtained by sequencing characters and other non-terminals in a certain order. Figure 1.5 shows an example context-free grammar for the language of arithmetic expressions like $(6 * (4 + 2)) + 6$. Its non-terminals include expressions, factors, terms and digits. Every line in the grammar contains a rule defining how a non-terminal can be formed from characters and/or other non-terminals. For example, the term $6 * (4 + 2)$ is formed from the term 6, the character $*$ and the factor $(4 + 2)$ according to the fourth rule of the grammar.

Rules in context-free grammars can be *recursive*, i.e. recursively define a non-terminal in terms of itself, like the second and fourth rules in Figure 1.5. They can also be *mutually recursive*, like the second, fourth and sixth rules in Figure 1.5, defining non-terminals Expr, Term and Factor cyclically in terms of each other. Such rules are common in grammars for programming languages, where statements and expressions often contain other statements or expressions, e.g. the C expression `f() + 3` contains two sub-expressions: `f()` and `3`.

Programs that interpret strings of a formal language (e.g. compilers that accept source code in a programming language) generally use a *parser*: a component that accepts arbitrary strings as input and checks whether they are in the formal language. If they are, the parser typically returns a structured representation that reveals how the strings can be constructed from the grammar. The

construction of efficient parsers for formal grammars is a well-studied topic; a wide variety of algorithms have been studied in the literature (see e.g. [3]).

Two standard requirements for parser components are that they should parse the strings of a certain formal language and that they should do it reasonably efficiently. The first requirement is especially hard to guarantee when writing parsers as general string-processing functions in a general-purpose language. From a semantic point of view, the problem is that the requirement is formulated at a higher level of abstraction (formal grammars and languages) than the semantics of the implementation (general string-processing functions). The standard approach to solve this uses the *parser generators* mentioned before [109, 176]. In this approach, the parser is not written by the programmer, but rather the intended grammar is specified in a special-purpose higher-level language (often based on BNF). The higher-level semantics of a grammar is the formal language it defines, and the correspondence to the intended language is much easier to guarantee or verify. The grammar can also be more easily adapted to changes in the formal language.

The parser generator will compile such a higher-level grammar to source code for a parser in a target programming language. It typically uses a fixed parsing algorithm and applies restrictions to the grammar to ensure the parsing algorithm supports it. Often, the grammar contains a *semantic action* for every rule: a piece of source code in the target language which is executed by the parser when it successfully identifies an instance of the rule in the input text. The results of semantic actions are called *semantic values*. The semantic action receives semantic values for the components of the matched rule and returns a semantic value for the resulting non-terminal. Sometimes, the semantic action may perform side-effects in the process, but typically it will just generate an in-memory representation of the parsed structure of the input file (the *abstract syntax tree* or *AST*).

Parser generators are often designed for use in programming language compilers. In such scenarios, there is little need for modularity within the parser specification: there is only a single language to be parsed and it doesn't change very often. Additionally, only a single set of semantic actions is used, producing ASTs. As a result, the grammar specifications used by these tools generally couple grammars with semantic actions in one component, although it could be useful to separate the two. Similarly, most tools support only one parsing algorithm although it would be useful to use different algorithms for a single grammar (e.g. a very fast algorithm for batch compilation and a slower one with high-quality error messages for interactive use). Multiple grammars can often not be easily combined and the grammar DSL often does not support abstraction in the definition of non-terminals. Furthermore, integration with the target language is often limited to the translation performed by the parser

generator. If the target language has a type system, the semantic actions are only validated after translation. For most parser generator systems, there are limited tools available for developing the grammar (e.g. for debugging or specifying the semantics of the grammar), although some systems do fairly well in this respect (e.g. the ANTLRWorks tool for the ANTLR parser generator [18]).

An alternative to parser generators is the use of *parser combinators*. These are a type of functional library in which individual parsers are components that can be combined using *combinators*. For example, a parser can be represented as a function that takes an input string and either fails or returns a parse result and the remaining input. In libraries based on the `Applicative` abstraction, a sequencing combinator $\circledast$ combines parsers p_1 and p_2 into a parser $p_1 \circledast p_2$ that first applies p_1 on the full input and then applies p_2 to the remaining input, combining both parse results into a combined result. Similarly, the $\lhd\rhd$ disjunction combinator combines two parsers p_1 and p_2 into a parser $p_1 \lhd\rhd p_2$, which first tries p_1 on the full input and returns the remaining input and result if p_1 succeeds. If p_1 fails, then p_2 will be tried on the original input and its results returned. Primitive parsers are provided which succeed if and only if a certain character is present in the input and returns the remaining input. To achieve the equivalent of semantic actions, a combinator $\$$ constructs a parser $f \mathbin{\$} p$ that behaves similarly to p, but applies the function f to its parse result. Finally, recursive parsers can be modelled using the recursion of the host language. In this way, a parser for a desired language can be constructed using combinators and primitive parsers whose semantics are similar to those of the language constructs in a grammar language like BNF.

Parser combinator libraries solve some of the problems of parser generators: abstraction facilities from the programming language they are written in (the *host language*) can typically be reused automatically for parsers. Similarly, host language specification techniques like type systems can be used to specify parsers' semantics (e.g. their result types). Using higher-rank polymorphism in languages like Haskell, one can even write parsers that support any library that supports a certain set of combinators, so that one obtains a component that does not really represent a parser but rather a grammar that can be used with different parsing algorithms.

To build a grammar EDSL whose semantics closely correspond to that of context-free grammars, one needs to use a purely functional language as grammars semantically have no side-effects. However, we will explain in Chapter 3 that in purely functional languages, parser combinators inherit an important limitation from the host language's recursion. The semantics of recursive parsers in such languages corresponds does not correspond to that of context-free grammars, but rather to what I call ω-regular grammars. The problem with this is that those grammars cannot be used with many standard and efficient parsing algorithms

such as those based on bottom-up LR algorithms, and do not permit many other kinds of grammar analysis and transformation algorithms.

Chapter 3 presents my contribution to the problem of building a higher-level grammar EDSL. I present a way to circumvent the limitations of Haskell's recursion by using an explicit representation of recursion in a parser combinator library. Semantically, this representation closely models the recursion of context-free grammars. It is based on a value- and type-level representation of non-terminal identifiers that makes extensive use of advanced Haskell type system features, particularly GADTs [183] and type families [204]. A side-effect of my approach is that I am able to decouple grammars from semantic actions into separate components, permitting a precise model of the complex semantic interactions between a grammar, a parsing algorithm and a set of semantic actions. On the flip side, the encoding of non-terminals imposes a certain overhead on the programmer and certain combinators can only be provided in a limited way.

In Chapter 3, I motivate and introduce our approach and present five grammar algorithms exploiting my model. They form the basis of the publicly available `grammar-combinators` parsing library.

1.3.3 Meta-programs

Meta-programs are programs that generate or manipulate other programs. They are widely used in practice, with well-known examples including parser generators [176, 109], reflection and byte-code generation in Java-like languages [173, 25] and eval primitives in languages like JavaScript [196]. Very often, meta-programs are used to implement features that would otherwise require extending the programming language.

The semantics of meta-programs are intrinsically very complex, because they are situated on two levels. Their semantics is to generate or manipulate other programs, but those programs have their own semantics as well. Many of the semantic properties that one would like to hold for a meta-program simply require other properties to hold for the programs that it works with. A parser generator, for example, should generate programs that are parsers, i.e., functions or procedures that accept text as input and return the intended type of results.

Such two-level semantic properties are so complex that they are out of reach for most specification techniques. Nevertheless, explicit statements of meta-programs' properties provide the same benefits as for normal programs: tracking assumptions between them and other (meta-)programs, to prevent bugs and facilitate modular development. Often, meta-program authors settle for what

one might call *weak correctness* specifications: specifications about the *generated* programs that are only validated *after* execution of the meta-program. Such solutions are better than nothing, but they are not modular: meta-programs' properties and assumptions cannot be specified for arbitrary executions, making it hard to ensure their semantic correctness and track assumptions between (meta-)programs that are combined together.

Chapter 4 presents the contribution of this thesis for meta-programming: the definition and study of a novel set of meta-programming primitives in the programming language Agda. The power of Agda's dependent type system allows the assignment of precise types to these primitives and meta-programs that use them. One limitation is that I have not proved the foundational correctness of the primitives themselves. Contrary to some other proposals, meta-programs in my approach are written in the same (functional) style as normal Agda programs. To demonstrate the power of the approach, I show meta-programs in two application domains: *datatype-generic programming* (the definition of algorithms that can be applied to an entire class of data types and whose behaviour is defined in terms of the structure of the data type) and *tactics* (programmer-written automated procedures for proving sub-theorems in mathematical proof assistants). My example in the first domain is a function that generates a serialisation function from the definition of a data type. In the domain of tactics I show the implementation of an `assumption` tactic; an elementary tactic that finds proofs for sub-theorems known to hold from previous assumptions. In general, my approach is the first to support general meta-programs in a standard functional style with types that express strong correctness properties such as type correctness and termination.

1.3.4 Effect polymorphism

The fourth and final contribution of this thesis is related to the *object-oriented paradigm* [163, 105]. Definitions of the paradigm depend somewhat on who you ask, but we will describe some common characteristics following Pierce [187]. The paradigm recommends the use of *objects*: semantic entities that couple a piece of private mutable state (*instance state*) with a set of procedures that have access to that state (*methods*). An object's methods implement an *interface* and interfaces can be extended to form a hierarchy with often a *subtyping* relationship between them in the type system (if there is one). Object methods can be implemented using *classes*, components which define a type of instance state and implement an interface in terms of it. One class can give rise to any number of *instances*: objects sharing the class's method implementations but not the instance state. Classes allow *inheritance*: defining *subclasses* which *override* and/or reuse superclass methods to implement a possibly extended

interface. Under *open recursion* or *late binding*, calling an object's method from within a method of the same object invokes the same method as calls from outside. This gives subclasses additional opportunity to override superclass behaviour. Not all object-oriented languages use the concept of classes to implement interfaces; some provide other mechanisms to implement objects with corresponding notions of inheritance (e.g. *prototypes* in JavaScript [89]). Many languages provide other forms of inheritance *in addition to* classes, e.g. *traits* [202], *aspects* [117] or *subobjects* [226]. Other forms of inheritance can sometimes be encoded, e.g. *object algebras* [170].

Encapsulation is the crucial object-oriented principle that instance state should be private, i.e. only the object's own methods can read it or write to it.[5] As hinted previously, the principle should actually be regarded as a semantic technique that limits the side-effects of components. In this view, it states that methods and procedures may directly perform arbitrary side-effects *except* read from or write to the instance state of other objects. Encapsulation is already a very useful way to make components more semantically independent. This has been formally demonstrated by Jeffrey and Rathke, who demonstrated that an object in a certain language can be semantically fully described by the set of all possible traces of incoming and outgoing method calls on the object [108]. This implies that the object's mutable state is not directly observable from the outside world.

The principle of encapsulation can be strongly strengthened by extending it to other side-effects than just mutable state. This is the essential idea behind the *object-capability (OC) model* [152]. According to this model, an object can represent a *capability*: the ability or permission to perform certain side-effects. There may, for example, be an object representing the ability to make outgoing connections via the network, to read and write files on the file system or to access the representation of a webpage [151]. Using *wrapper* objects, custom restricted capabilities can be defined from existing ones, e.g. the ability to connect to a certain host on the internet, or to read but not write from any file below a certain file system location.

Using object capabilities, one can obtain precise bounds on the side-effects that procedures can produce: the set of capabilities of all the objects it can access. However, for this to work, procedures must not be able to perform effects in other ways or bypass objects' interfaces. *Object-capability programming languages* therefore forbid the use of such features, most importantly static functions with side-effects, globally accessible static mutable state variables and features that allow access to objects' private instance state (e.g. certain

[5] Often, the principle is relaxed a bit, so that, for example, other objects of the same class can also access the instance state.

uses of reflection in Java) [148, 151]. When side-effects need to be restricted for security purposes, attackers must not be able to bypass such restrictions in any way.

Chapter 5 presents my contribution to the study of object-capability languages. It is related to their formal characterisation. Existing formalisations of object-capability languages, notably by Maffeis et al. [131], essentially start from the *reference graph*: the graph of all objects in a running program, connected by references that the objects hold to each other. These formalisations (such as Maffeis et al.'s *capability safety* and *authority safety* properties) restrict the way in which the reference graph is allowed to evolve over time, depending on the references held by the currently executing program. In this way, the properties imply a restriction on the heap changes that the program may perform as side effects. The downside of such formalisations is that they are very specific to one type of side effect (write access to mutable heap variables), while the object-capability model applies to other side effects just as well. They are also quite operational in nature and do not provide great insight in the nature of the necessary restrictions. Finally, they are hard to use for formal reasoning about the enforced properties.

My contribution is a more denotational characterisation of the object capability model. The main idea is that the use of object capabilities introduces *effect polymorphism*: functions or methods that can only produce effects by invoking methods of their argument objects, must necessarily be polymorphic in their side-effects; if the function is invoked with objects producing certain effects, then the function will produce those effects, and only those. To make this idea precise, I define a translation of methods in an object-oriented language to Haskell,[6] using a monadic representation of side-effecting computations. Specifically, I translate to computations that can be executed in an arbitrary monad, provided they receive arguments which produce effects in that monad. Using the higher-rank polymorphism of Haskell (and System F_ω), these computations can be specified to be *polymorphic* in the monad, so that the general *parametricity* property (see Section 1.2.1) can be used to derive properties about them. A large part of the work in Chapter 5 is a formal demonstration that the parametricity of the method translations generalises Maffeis et al.'s *capability safety*.

In addition to this, Chapter 5 presents and explains some novel insights that our new characterisation of object-capability languages offers. I show, for example, that current OC languages and formalisations leave one effect implicitly available: the allocation of new mutable state; adding a capability for it has important theoretical and practical advantages. Furthermore, my translation

[6] Or a more principled calculus like System F_ω, but we use Haskell as a convenient type-inferenced notation for it.

establishes a new link between object-capability languages and the well-studied fields of functional programming and denotational semantics and this presents opportunities for both fields to learn from each other. Concretely, I present a form of *local capabilities*, inspired by Haskell's ST monad and related to an OO technique called *borrowed references* [41]. Additionally, I hope to elaborate some other related tracks in future work, as I will discuss further in Section 5.9 and Chapter 6.

1.4 Other research conducted

The work presented in this thesis is a subset of the research I have done in the past four years. I selected the work of which I was the principal author and whose topic fits in the scope of this text. For completeness, the following is a list of other research I have worked on, with a short description of the content, references to publications and a description of my role in the research.

Secure multi-execution My first research project was a technique called *secure multi-execution*, through which information flow policies can be enforced for unmodified programs. The technique is based on a modified semantics which executes a single program several times, once for each security level. The behaviour of I/O commands is correspondingly modified so that the behaviour of programs respecting the policy is indistinguishable from the original at any security level and so that any program automatically respects the information flow policy. I was the principal author of this publication, although my promotor Frank Piessens is responsible for the original idea and contributed strongly to the text.

Publication data:

> Dominique Devriese and Frank Piessens. Non-interference through secure multi-execution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP 2010)*, pages 109–124. IEEE Computer Society, May 2010.

After this initial publication, I have collaborated with different researchers, further researching the technique of secure multi-execution. Together with Nataliia Bielova, Fabio Massacci and Frank Piessens, we studied applying the technique to reactive programs in the Featherweight Firefox browser model. Nataliia Bielova was the principal author of this work.

Publication data:

Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. Reactive non-interference for a browser model. In *Proceedings of the 5th International Conference on Network and System Security (NSS 2011)*, pages 97–104, September 2011.

Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. Reactive non-interference for the browser: extended version. CW Reports CW602, Department of Computer Science, K.U.Leuven, February 2011.

Another collaboration started with Willem De Groef's master thesis, which I mentored. He constructed FlowFox, an implementation of secure multi-execution in a real-world browser (Mozilla Firefox). This work continued during his PhD under the supervision of Frank Piessens. Willem De Groef is the main author of this work.

Publication data:

Willem De Groef, Dominique Devriese, and Frank Piessens. Better security and privacy for web browsers: A survey of techniques, and a new implementation. In *Formal Aspects of Security and Trust (FAST 2011)*, volume 7140 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2012.

Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*, pages 748–759. ACM, 2012.

Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal on Computer Security*, 2014. Accepted.

The study of secure multi-execution continued in collaboration with Gilles Barthe, Juan Manuel Crespo, Exequiel Rivas and Frank Piessens by showing that the technique can also be implemented as a program transformation. My contribution to this work was a formalisation and proof of the technique in Agda. Juan Manuel Crespo made a formalisation on paper and Exequiel Rivas was responsible for an implementation for JavaScript based on Google Caja.

Publication data:

Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. Secure multi-execution through static program transformation. In *Formal Techniques for Distributed Systems (FMOODS/FORTE 2012)*, volume 7273 of *Lecture Notes in Computer Science*, pages 186–202. Springer Berlin Heidelberg, June 2012.

> Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. Secure multi-execution through static program transformation: extended version. CW Reports CW620, Department of Computer Science, KU Leuven, April 2012.

Together with Willem De Groef, I mentored Tom Reynaert during his master thesis on a privacy-enhanced social application platform (PESAP) which used FlowFox and secure multi-execution for enforcing an information flow policy on the client side. Tom Reynaert is the main author of this work.

Publication data:

> Tom Reynaert, Willem De Groef, Dominique Devriese, Lieven Desmet, and Frank Piessens. PESAP: a privacy enhanced social application platform. In *International Workshop on Security and Privacy in Social Networks (SPSN 2012)*, September 2012.

A side track of my research on information flow enforcement was the development of a Haskell library implementing three different types of information flow enforcement as monad transformers over an abstract underlying monadic library. I am the main author of this work.

Publication data:

> Dominique Devriese and Frank Piessens. Information flow enforcement in monadic libraries. In *Proceedings of the 7th ACM SIGPLAN workshop on Types in language design and implementation (TLDI 2011)*, pages 59–72, January 2011.

Functional Programming In collaboration with Ilya Sergey, Dave Clarke and Frank Piessens, I worked out an alternative to the work presented in Chapter 3. It is another approach to the representation of grammars and other recursive DSLs based on a recursion primitive called `afix` in a type class *ApplicativeFix*. The primitive has a rank-2 type ensuring a certain natural property of the recursive DSL and we designed and implemented a form of syntactic sugar to obtain a natural syntax. Ilya Sergey and I are the main authors of this work.

Publication data:

> Dominique Devriese, Ilya Sergey, Dave Clarke, and Frank Piessens. Fixing idioms: a recursion primitive for applicative DSLs. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation (PEPM 2013)*, pages 97–106. ACM, January 2013.

I also collaborated with Ilya Sergey, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke and Frank Piessens on *monadic abstract interpreters*: a unifying representation of techniques in the field of *abstract interpretation*, a theoretically supported framework for the static analysis of source code. Using monads as a unifying notion for abstract machines, we were able to implement abstract interpretation techniques like context sensitivity, polyvariance, flow-sensitivity, reachability-pruning, heap-cloning and cardinality bounding, independently from any particular programming language semantics. The main authors for this work were Ilya Sergey, Matthew Might, Jan Midtgaard, David Darais and myself. My contributions centred around the design of the monadic representation.

Publication data:

> Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic abstract interpreters. In *Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2013).*, pages 399–410. ACM, June 2013.

I mentored Thomas Winant during his master thesis on *partial type signatures* for Haskell. He designed and implemented this form of type signatures to allow specifying a type partially and leaving other parts (indicated by *wildcards*) to be inferred by the inference engine. He formalised this as an extension of the OutsideIn(X) type inference framework and implemented it in the GHC Haskell compiler. Thomas has currently started PhD research, where he will develop this project further. Thomas is the main author of this work.

Publication data:

> Thomas Winant, Dominique Devriese, Frank Piessens, and Tom Schrijvers. Partial type signatures for Haskell. Symposium on Practical Applications of Declarative Languages, 2014. Accepted.

Finally, I mentored Jesper Cockx during his master thesis on *Overlapping and Order-Independent Patterns* for Agda. He designed and implemented a form of overlapping and order-independent patterns for Agda. He designed this language feature and formalised it, based on an existing formalisation of standard pattern matching for dependently-typed languages. He implemented his design in an extension of the Agda programming language. Jesper has currently started his own PhD research. He is the main author of this work.

Publication data:

Jesper Cockx, Frank Piessens, and Dominique Devriese. Overlapping and order-independent patterns - definitional equality for all. In *European Symposium on Programming (ESOP)*. Springer-Verlag, 2014. Accepted.

1.5 Outline

In the next sections, I present the contributions of this thesis: novel functional techniques for the representation and specification of software in four domains. We have already introduced the four domains and our contributions in Section 1.3, so we will be brief here. Chapter 2 discusses instance arguments in Agda and Chapter 3 presents our Haskell representation for context-free grammars. In Chapter 4, we explain our approach for typed syntactic meta-programming and finally, in Chapter 5, we explain our work on effect polymorphism as a defining feature for object-capability languages.

I have chosen to present original papers with only minor modifications w.r.t. the original conference or journal publication. Each section starts with the original title and abstract as well as details about the publication venue or journal. I was the principal author of each of the four presented papers.

Finally, in Chapter 6, I conclude this thesis by taking a step back to reflect on the results of my work and look forward on the directions it presents for future research and practice.

Please note that this text uses per-chapter appendices so that, for example, Appendix 2.A can be found on page 72, directly after Chapter 2 but before Chapter 3.

Chapter 2

Instance Arguments in Agda

Publication data

Dominique Devriese and Frank Piessens. Instance Arguments in Agda. *Higher-order and Symbolic Computation*, 2014. Accepted.

Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in Agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*, pages 143–155. ACM, September 2011.

Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in Agda. *ACM SIGPLAN Notices*, 46(9):143–155, September 2011.

Abstract

We present *instance arguments*: an alternative to type classes and related features. Instance arguments are a new, general type of function arguments, resolved at the call-site scope in a type-directed way. The concept is inspired by both Scala's implicits and Agda's existing implicit arguments, but differs from both in important ways. Our mechanism is designed and implemented for the dependently typed, purely functional programming language/proof assistant Agda, but our design choices can be applied to other programming languages as well.

Like Scala's implicits, we do not provide a separate structure for type classes and their instances, but instead rely on Agda's standard dependently typed records, so that we can reuse standard language mechanisms to provide features that are missing or expensive in other proposals. Like Scala, we support the equivalent of local instances. Unlike Scala, functions taking our new arguments are first-class citizens and can be abstracted over and manipulated in standard ways. Compared to other proposals, we avoid the pitfall of introducing a separate type-level computational model through the instance search mechanism. All values in scope are candidates for instance resolution. A final novelty of our approach is that existing Agda libraries using records gain the benefits of instance arguments without any modification.

We discuss our implementation in Agda (part of Agda v2.3.0 onward) and we use monads as an example to show how it allows existing concepts in the Agda standard library to be used in a similar way as Haskell code uses type classes. We also demonstrate and discuss equivalents and alternatives to some advanced type class-related patterns from the literature and some new patterns specific to our system.

2.1 Introduction

2.1.1 Type classes

Around 1989, a group of scholars on the Haskell Committee were facing the problem of fixing the types of the numeric and equality operators in the emerging Haskell programming language [100]. Such operators introduce a natural requirement for overloading or "ad hoc" polymorphism. For example, the == operator, of type $t \to t \to \mathit{Bool}$, should only be defined for certain types t (e.g. Bool, Integer) and not for others (e.g. function types). Additionally, different implementations are required for different types t.

The committee at the time recognized the issue as an instance of a more general problem in need of a general solution and adopted Wadler's proposal for what became known as the Haskell type class system. For the == operator, the approach is based on a type class $\mathit{Eq}\ t$, with instances for appropriate types t. To avoid troubling this section with notations for infix operators, we write *equal* for ==.

$$\begin{array}{lll} \textbf{class} & \mathit{Eq}\ t & \textbf{where}\ \mathit{equal} :: t \to t \to \mathit{Bool} \\ \textbf{instance} & \mathit{Eq}\ \mathit{Bool} & \textbf{where}\ \mathit{equal} = \mathit{primEqBool} \\ \textbf{instance} & \mathit{Eq}\ \mathit{Integer} & \textbf{where}\ \mathit{equal} = \mathit{primEqInteger} \end{array}$$

$$
\begin{aligned}
&\mathit{neq} :: \mathit{Eq}\ t \Rightarrow t \to t \to \mathit{Bool}\\
&\mathit{neq}\ a\ b = \mathit{not}\ (\mathit{equal}\ a\ b)\\
&\mathit{test} :: \mathit{Bool}\\
&\mathit{test} = \mathit{neq}\ (5 :: \mathit{Integer})\ 5
\end{aligned}
$$

Subclasses can also be defined:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Ordering} = \mathit{LT} \mid \mathit{EQ} \mid \mathit{GT}\\
&\textbf{class}\ \mathit{Eq}\ t \Rightarrow \mathit{Ord}\ t\ \textbf{where}\ \mathit{compare} :: t \to t \to \mathit{Ordering}
\end{aligned}
$$

An essential requirement for type classes to work is that for functions like *neq* which use the *equal* operator for a universally quantified type t, this is made explicit in their types. The compiler can then check that the required instances are available when t is instantiated to a concrete type: in the definition of *test* when *neq* is called on two *Integer* values, it finds an *Integer* instance of the type class and calls *neq* with that instance.

Before we continue, we want to make clear that when talking about Haskell, we will ignore the distinction between the type class concept in Haskell proper and common and uncontroversial extensions of it like *FlexibleContexts*, *FlexibleInstances*, *MultiParamTypeClasses*, *TypeFamilies* and *RankNTypes*.

Note also that when we mention ad hoc polymorphism, we mean *open* ad hoc polymorphism. This means that additional instances of abstract concepts can be added independently by users of functions that require the concept.

2.1.2 The downsides of an extra structuring concept

A disadvantage of Haskell's type class system is that classes and instances form a separate, special-purpose structuring concept, in addition to the more standard algebraic data types (ADTs). Because of this duplication of functionality, many of the features that have in the past been introduced as extensions of type classes duplicate features that already existed for ADTs. Constraint families [174] (allowing classes to have abstract constraints on type class parameters) and associated type families [204] (allowing classes to specify abstract types) both roughly correspond to how generalized algebraic data type [183] values can carry types or type functors that are not parameters of the data type. In the area of generic programming, successful techniques existed to build generic algorithms over ADTs [120], but these have had to be adapted for use with type classes [121, 237].

Similarly, higher-rank types [182] have long allowed ADTs to be abstracted over. However, in a paper about a datatype-generic programming technique [121],

Lämmel and Peyton Jones note that this is not possible for type classes. In the following pseudo-code, they wanted to abstract over a type class using a variable *cxt* of kind $* \to Constraint$ (the meaning of these type classes is not important here):

```
   -- Pseudo-Haskell
class (Typeable a, cxt a) ⇒ Data cxt a where
   gmapQ :: (forall b.Data cxt b ⇒ b → r) → a → [r]
```

This pseudo-code is not legal Haskell so Lämmel and Peyton Jones provide a solution based on a "generic" type class *Sat* parameterised by the type of a dictionary record that it should carry. The abstraction over the variable *cxt* of kind $* \to Constraint$ is then replaced by abstraction over a variable *cxtD* of kind $* \to *$.

```
class Sat a where dict :: a
class (Typeable a, Sat (cxtD a)) ⇒ Data cxtD a where
   gmapQ :: (forall b.Data cxtD b ⇒ b → r) → a → [r]
```

This was a clever solution, but it amounts to replacing the type class *cxt* with an ADT *cxtD* for which the desired feature (abstraction over it) is available. Only recently, the new `ConstraintKinds` extension offers this kind of abstraction for type class constraints, even though it has been available for data types for a long time [244].

2.1.3 Dictionary translation

Wadler and Blott formalise type classes in their 1989 paper using a translation to a standard Hindley-Milner typed functional language [235]. This translation is known as the *dictionary translation* and not only serves as an implementation strategy, but also gives an accurate semantic model of type classes. A type class is modelled as a dictionary record type, with the type class operations as record fields. Instances become record values containing the definitions in the instance as fields. The above code translates to the following:

```
data Eq t = EqDict { equal :: t → t → Bool }
data Ord t = OrdDict { eqDict :: Eq t, compare :: t → t → Ordering }
boolEq :: Eq Bool
boolEq = EqDict primEqBool
intEq :: Eq Integer
intEq = EqDict primEqInteger
neq :: Eq t → t → t → Bool
```

$$neq\ dict\ a\ b = not\ (equal\ dict\ a\ b)$$
$$test :: Bool$$
$$test = neq\ intEq\ 5\ 5$$

A striking property of this translation is that the resulting code is not actually that far from the original. Apart from the additional naming of instances (which has also been proposed for Haskell [112]), the translation only produces extra verbiage in the implementation of functions that use the type class's operations. In the *neq* function, the dictionary of type $Eq\ t$ is now passed around explicitly where this happened implicitly before. Additionally, in the definition of *test*, we need to explicitly specify the *intEq* dictionary as an extra parameter whereas it was inferred by the compiler before.

Apart from the automatic inference of instances, the dictionary model has many advantages over the standard type class system. All the power of normal language record mechanisms is available, and they can be defined, manipulated and abstracted over in standard ways.

2.1.4 Scala implicits

The Scala programming language provides an alternative approach to ad hoc polymorphism called implicits that avoids the introduction of a special-purpose structuring mechanism [167, 164]. Because of this, powerful mechanisms like abstract type declarations [164, §4.3] can be used to model features that have had to be specifically defined and implemented for type classes. Our running example can be encoded in Scala as follows:

$$\textbf{trait}\ Eq\ [A]\ \{\textbf{def}\ eq\ (x : A, y : A) : Boolean\}$$
$$\textbf{def}\ equal\ [A]\ (x : A, y : A)\ (\textbf{implicit}\ eqA : Eq\ [A]) = eqA.eq\ (x, y)$$
$$\textbf{implicit object}\ boolEq\ \textbf{extends}\ Eq\ [Boolean]\ \{...\}$$
$$\textbf{implicit object}\ intEq\ \textbf{extends}\ Eq\ [Int]\ \{...\}$$
$$\textbf{def}\ neq\ [A]\ (x : A, y : A)\ (\textbf{implicit}\ eqA : Eq\ [A]) = !\ equal\ (x, y)$$
$$\textbf{val}\ test = neq\ (5, 5)$$

The type class *Eq* is modelled as a dictionary trait $Eq\ [A]$.[1] Traits are a general object-oriented structuring concept provided by Scala, similar for our purposes to records. Two dictionary objects *intEq* and *boolEq* are introduced and annotated with the **implicit** modifier. Both functions *equal* and *neq* take three arguments: x and y of type A and *eqA* of type $Eq\ A$ which carries the actual implementation

[1] Square brackets in Scala denote type application or abstraction.

of *equal*. The third argument is marked as **implicit**. When the function *neq* is called in *test*, and the implicit argument is not explicitly provided, its value is inferred by the compiler.

Unfortunately, functions with implicit arguments are not first-class citizens in Scala, mostly due to syntax-technical problems. Some important features of Scala's standard functions (currying, partial application, lambda expressions) are not available for implicits; see Section 2.6.1 for more details.

2.1.5 Instance resolution

An aspect of type classes and implicits we have not yet touched upon is instance resolution. Haskell allows *parametric instances* like

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a]\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{equal}\ [\,] & [\,] & = \mathit{True} \\
\mathit{equal}\ (a : as)\ (b : bs) & & = \mathit{equal}\ a\ b \wedge \mathit{equal}\ as\ bs \\
\mathit{equal}\ \_ & \_ & = \mathit{False}
\end{array}
\end{array}
$$

With this instance, Haskell will resolve constraints of the form $\mathit{Eq}\ [a]$ by recursively resolving the constraint $\mathit{Eq}\ a$ and then using that in the above definition of *equal*. From the perspective of the dictionary translation, this corresponds to functions from dictionaries to dictionaries that are implicitly used to construct any needed dictionaries. This mechanism makes the instance resolution algorithm more powerful and complex. A set of restrictions is enforced on the structure of the types involved in instance contexts to ensure that the instance resolution remains decidable. Two widely used Haskell extensions (associated type families [204] and functional dependencies [111]) introduce further complexity by adding what are essentially decidable type-level computation primitives, which can be triggered during the instance resolution process. As an reviewer (of the conference version of this chapter) notes, these extensions effectively expose an interpreter for a simple logic programming language (no backtracking) which can reason about Haskell types.

The resolution algorithm that Scala uses to infer a value for implicit arguments is similar. To resolve an implicit argument of type T, Scala will consider values in scope that have been marked "implicit", but also values defined in certain modules related to T. It will consider values of type T, but also functions that themselves take only implicit arguments and return a value of type T. This makes the resolution recursive, like for Haskell. To ensure decidability, Scala keeps track of the "call stack" of the resolution and detects infinite loops using a conservative criterion.

In non-dependently typed languages like Haskell or Scala, type-level computation is not directly available in the base language. Therefore, the type-level computation that can be achieved using these primitives fills a certain gap in the language and various people have demonstrated the surprising amount of power that these extensions offer [118, 141, 167]. However, the computational model for these primitives differs strongly from the languages' standard models: a form of structural recursion is used instead of non-structural (although many compilers provide an option to change this), pattern matching is open and unification-based (similar to Prolog) instead of closed and functional and the syntactic order of pattern matching and recursive calls is reversed.

2.1.6 Implicit function arguments in Agda

A final language feature we want to present before introducing our proposal, can be found in our target language itself: Agda's *implicit* or *hidden arguments* [161]. Agda allows function arguments to be marked as "implicit", indicating that they do not need to be provided explicitly at the call site. For example, a polymorphic identity function is defined as follows:

$$
\begin{aligned}
&id\ :\ \{A\ :\ Set\}\ \rightarrow\ A\ \rightarrow\ A\\
&id\ v\ =\ v
\end{aligned}
$$

When type checking the expression *id true*, Agda silently inserts a meta-variable (as if the expression were $id\ \{\_\}\ true$), and Agda's type inference will instantiate this meta-variable to *Bool*. The argument may be specified explicitly, by writing $id\ \{Bool\}\ true$. Implicit arguments are pervasive in most Agda code, and Agda would be nearly unusable without it.

Unfortunately, Agda's implicit arguments are of no help for implicitly passing around and inferring type class dictionaries. The reason for this is that Agda will only infer implicit arguments in two situations. Either the value is fixed by the types of the other arguments or the result (consider how $A\ =\ Bool$ was fixed in the call *id true* above), or Agda can statically decide that only a single value can exist of the required type. This makes the feature unsuitable for passing type class dictionaries, because, for example, for a type t, many values of type $Eq\ t$ can typically be defined. For example, even for a simple type like *Bool*, we can define a trivial equality operator $equal\ \_\ \_\ =\ true$ in addition to the standard one.

However, unlike Scala, functions taking an implicit argument in Agda are first-class citizens. They can be abstracted over, their types can be spelled out, anonymous functions with hidden arguments are no problem and syntax is

available to keep a tight control over whether or not implicit arguments are inferred or not. In some cases this requires writing an eta-expanded version of a function call (e.g. $\lambda\,\{A\} \to id\,\{A\}$ instead of id) to make sure that Agda does not try to infer the hidden argument.

2.1.7 Instance arguments

The feature we propose is inspired by both Agda's implicit arguments and Scala's implicits. It is a new kind of function arguments, which we call *instance arguments*. Like Haskell type classes and Scala implicits, they provide *open* ad hoc polymorphism, i.e. instances of abstract concepts can be added independently from the definition of the concept. If openness is not required, Agda supports other solutions based on the definition of a universe representing the complete set of types that satisfy the concept (see e.g. [5]).

To use instance arguments with our running example, we need to define a standard Agda record *Eq* corresponding to the *Eq* type class, and instances for the $\mathbb{N}$ and *Bool* type from the Agda standard library [51].

```
record Eq (t : Set) : Set where
  field equal : t → t → Bool

eqBool : Eq Bool
eqBool = record {equal = primEqBool}

eqNat : Eq ℕ
eqNat = record {equal = primEqNat}
```

All of this is standard Agda code. Our modified version of Agda allows us to write the following:

```
equal : {t : Set} → {{eqT : Eq t}} → t → t → Bool
```

This type signature says that the function *equal* takes a *Set* (type) as its first (implicit) argument t. The novelty is in the double braces which mark the function's second argument eqT of type $Eq\ t$ as an instance argument. Next, the function takes two standard arguments of type t and returns a *Bool*. In *equal*'s definition, we simply take the implicitly passed dictionary and return the *equal* function contained in it:

```
equal {{eqT}} = Eq.equal eqT
```

With this type signature, we can now use *equal* as if it were defined as the method of a Haskell type class:

$$test \;=\; equal\ 5\ 3 \vee equal\ true\ false$$

In both calls to *equal*, *equal*'s instance argument is not explicitly given, so that it is inferred by the compiler, like for Agda's existing implicit arguments. The difference with the latter is in *how* the values are resolved. We will explain our resolution algorithm further on in the text, but for the example above, *eqBool* and *eqNat* will be correctly resolved.

The fact that instance arguments closely resemble Agda's existing implicit arguments and only differ in how omitted arguments are inferred, means that we can reuse many elements of the design of implicit arguments and inherit some of their qualities. Specifically, functions with instance arguments are first class citizens and there are no limitations on the position of the implicit arguments within a function type.

2.1.8 Contributions

The contributions of this work are the proposal and study of instance arguments and an implementation in Agda. Our proposal does not introduce a separate structuring concept and ad hoc polymorphic functions are first-class citizens. Our proposal can work with less or more powerful types of instance resolution, but we choose a simple one that avoids the introduction of a separate computational model.

To the best of our knowledge, no other proposal in the literature offers equivalents to *all* of the following features: associated type families and constraint families, multi-parameter type classes, local instances, abstraction over type classes and first-class ad hoc polymorphic functions (although Coq, Haskell and Scala each have almost all of them). No other proposal has explored an alternative to type classes without introducing a separate computational model in the language. Every other proposal has required "instances" to be somehow marked eligible for implicit resolution. Finally, no other proposal has a mechanism that is equivalent to how we automatically bring the benefits of instance arguments to unmodified records.

We formally define the workings of our feature, and discuss our design choices. We demonstrate the use of monads and present (often simpler) encodings of some type class based patterns from the literature. We also present some novel patterns of our own.

2.2 Instance arguments

We have already briefly presented instance arguments in Section 2.1.7 above. Let us provide some more details.

2.2.1 Resolving instance arguments

First, let us explain how a value for an omitted instance argument is inferred. Generally, an *instance argument is resolved* by the compiler *when only a single identifier is bound to a value of the expected type in the call-site scope* (more details in Section 2.4 and Appendix 2.A.3). We do not require values to be marked in a specific way to be eligible for this resolution. We take care to limit the computational power of our instance search algorithm so that we do not unwantedly introduce an alternative computational model.

Recall the example *test* from Section 2.1.7:

$$test \;=\; equal\ 5\ 3 \vee equal\ true\ false$$

What happens underneath, for example, for the application *equal 5 3*, is that the Agda type-checker notices that in order to pass the normal argument 5 to function *equal*, it first needs to infer the implicit argument t and instance argument eqT. It will assign a new meta-variable (see Norell [161, 162]) to both, but for the second argument, a constraint will additionally be registered indicating that that meta-variable needs to be resolved as an instance argument. The argument 5 will then be passed to *equal* as the third argument, and Agda will unify the first meta-variable with value $\mathbb{N}$. Agda will now notice that there is *only one value of type* $Eq\ \mathbb{N}$ *in scope* (*eqNat*) and assign it to the second meta-variable.

Like for implicit arguments, it is also possible to provide the instance arguments explicitly, should this be necessary:

$$test_2 \;=\; equal\ \{\!\{eqNat\}\!\}\ 5\ 3 \vee equal\ \{\!\{eqBool\}\!\}\ true\ false$$

Instance argument resolution will also consider values that are bound as the arguments of a function. Consider, for example, our version of *neq*. Like *equal*, it accepts a dictionary of type $Eq\ t$ as an instance argument:

$$neq\ :\ \{t\ :\ Set\} \to \{\!\{eqT\ :\ Eq\ t\}\!\} \to t \to t \to Bool$$

We can implement *neq* by explicitly accepting the dictionary argument and passing it to the *equal* function:

$$neq\ \{\!\{eqT\}\!\}\ a\ b\ =\ \neg\ (equal\ \{\!\{eqT\}\!\}\ a\ b)$$

However, this is unnecessarily verbose. If we leave out the dictionaries in the definition, Agda will silently insert an unnamed instance argument in the left-hand side and will silently infer *equal*'s instance argument to that unnamed value:

$$neq\ a\ b\ =\ \neg\ (equal\ a\ b)$$

Notice again how the mechanism is in many respects strikingly similar to Agda's existing implicit arguments. *Only the resolution of the instance value is not done by unification but by looking up a unique value of the right type in the current scope.*

There are some finer details about instance argument resolution that we go into in Section 2.4, but let us first explain some more visible features.

2.2.2 Existing records as type classes

An important and novel feature of our proposed system is that we can automatically bring its benefits to unmodified libraries that use standard dependently-typed records. In the above example, it is the function *equal* of type

$$equal\ :\ \{t\ :\ Set\}\ \to\ \{\!\{eqT\ :\ Eq\ t\}\!\}\ \to\ t\ \to\ t\ \to\ Bool$$

which allows us to use the *Eq* record in a more convenient, type-class-like way.

It turns out that this function is almost identical to a function in the *record module Eq*,[2] which Agda auto-generates. Let us explain this by repeating the definition of record *Eq* and additionally assuming that it contains *neq* as an associated function:

$$\begin{array}{l}\textbf{record}\ Eq\ (t\ :\ Set)\ :\ Set\ \textbf{where}\\ \quad \textbf{field}\ equal\ :\ t\ \to\ t\ \to\ Bool\\ \quad neq\ :\ t\ \to\ t\ \to\ Bool\\ \quad neq\ a\ b\ =\ \neg\ (equal\ a\ b)\end{array}$$

A module in Agda is a group of definitions that can be jointly parameterised over (and applied to) arguments. The *record module* that is generated for the above record definition will contain the function *neq* as well as a generated

[2]Note that record module *Eq* and record type *Eq* can share the same name because module and term name spaces are disjoint in Agda.

field projection function *Eq.equal* [161, 4.3 pp.82–83]. It is equivalent to the following:

```
module Eq {t : Set} (eq : Eq t) where
  equal : t → t → Bool
  equal = {- extract field equal from record value eq -}
  neq : t → t → Bool
  neq a b = ¬ (equal a b)
```

Note that the module abstracts over arguments $\{t : Set\}\ (eq : Eq\ t)$, which results in all functions in the module abstracting over these arguments. As a result, *Eq.equal* is available *outside* of the record module at the following type:

$$Eq.equal : \{t : Set\} \to (eqT : Eq\ t) \to t \to t \to Bool$$

The only difference between *Eq.equal* and our function *equal* above is that the latter takes *eqT* as an instance argument instead of an explicit argument. This observation has inspired us to automatically provide definitions like our *equal*, by auto-generating new versions of the record projection functions which take the record as an instance argument instead of a standard one. It is convenient to do so by extending Agda's module mechanism.

Agda modules or sections are a general scoping mechanism and there is a mechanism called *module application* that allows us to bring in scope (part of) a module's contents and abstract or apply all functions in a module from/to arguments. For our purpose, we can use a module application like the following:

```
open module EqInst {t : Set} {{eq : Eq t}} = Eq {t} eq
```

This defines a new module *EqInst*, containing all the definitions from module *Eq*, in this case the field projector *equal* and function *neq*. The module application ensures that they are available in a form that accepts the record value *eq* as an *instance argument instead of a normal argument.* The function *equal* in the new module *EqInst* is equivalent to our custom definition above. As we have demonstrated in the introduction, such definitions allow them to be used in a *type-class-like* way. By **open**ing the module, we bring in scope the definitions in the new module directly.

For module applications like the above, we provide a short-hand notation:

```
open Eq {{...}}
```

Like for Agda's standard module applications, the modifiers **public**, **using**, **renaming** and **hiding** can be used to control precisely what is brought in scope. This new module application is equivalent to the above except for the fact that

it doesn't name the EqInst module. The new type of module application can in fact be applied to any module taking at least one argument, turning the last (normal or implicit) argument into an instance argument. Because of this, the mechanism can also be used for modules that are not directly record modules, but, for example, instances of one.

2.2.3 Subclasses

In dictionary models of type classes, a subclass dictionary typically carries a superclass dictionary as one of its fields. The Agda standard library, for example, uses such a model. In the context of a dependently typed language, there is another possible model for subclasses, known as *Pebble-style structuring* or *sharing* [**?**], recommended by Sozeau and Oury [213, §4.1]. In this style, subclass dictionaries carry superclass dictionaries as parameters instead of fields.

Both models can be expressed with our system. Each has some specific advantages and disadvantages, e.g. a requirement to explicitly bring superclass dictionaries into scope or the need for an extra implicit superclass dictionary parameter in the type of functions with a subclass constraint. In this section, we demonstrate a Pebble-style model of an *Ord* subclass of our previously defined *Eq*:

```
record Ord {A : Set} (eqA : Eq A) : Set where
  field _<_ : A → A → Bool
```

Let's now suppose that we have values $eq\mathbb{N} : Eq\ \mathbb{N}$, $ord\mathbb{N} : Ord\ eq\mathbb{N}$ and $eqBool : Eq\ Bool$ in scope, but no instance of *Ord* for *Bool*s. We can now open the *Eq* {{...}} and *Ord* {{...}} modules and use the appropriate methods on $\mathbb{N}$ and *Bool*s, with the correct dictionaries being resolved in the background.

```
open Ord {{...}}
open Eq {{...}}
test₁ = 5 < 3
test₂ = equal 5 3
test₃ = equal true false
```

An ad-hoc polymorphic function $\_\leqslant\_$ can be defined as:

```
_≤_ : {A : Set} → {eqA : Eq A} → {{ordA : Ord eqA}} →
      A → A → Bool
a ≤ b = a < b ∨ equal a b
```

Note how the Pebble-style subclass model requires us to explicitly mention a superclass constraint in the type signature of $\_\leqslant\_$. This argument $eqA : Eq\ A$

is accepted as an implicit argument, not an instance, because it can typically be inferred from the parameters of the chosen *ordA* parameter. Because we require the superclass dictionary as an argument, it is automatically in scope for resolution inside the method.

The above shows that our mechanism does not impose a choice as to how subclasses are to be modelled by the programmer. We think this demonstrates that instance arguments are a general mechanism, giving the programmer the freedom to make his or her own design choices.

2.2.4 Considerations for instance arguments in other languages

An important question about our proposed instance arguments is how Agda-specific they are. We believe that the mechanism is widely applicable, and that many variations on our design choices are possible.

Let us consider the different modifications that we have made. A first step is the introduction of a new, specially annotated type of arguments to functions, which is likely to be unproblematic in many programming languages. Clearly, in non-dependently typed languages the arguments' type must be restricted to not depend on earlier non-type arguments, but this reflects the rules for normal arguments in those languages. However, care must be taken that functions with the new type of arguments are fully first-class on the one hand and that the programmer can tightly control the resolution of instance arguments on the other hand.

To the best of our knowledge, Agda was the first language to demonstrate that these two requirements can be combined with a natural syntax through a careful balancing in the type checking rules which govern function applications, lambda expressions, and the implicit insertion of implicit lambda's. The rules in Appendices 2.A.1 and 2.A.2 for our instance arguments are simply adaptations of the corresponding rules for Agda's existing implicit arguments [161]. We expect that similar types of function argument and similar rules can be introduced for any language which has some form of partial function application and lambda expressions.

The choice of the algorithm for resolving instance arguments is orthogonal to the rest of our design. We clearly choose a relatively restricted one (more explanation in Section 2.4), but we think that other choices can also be combined with the rest of our design. This can range from our relatively restricted inference to a full-power automated proof-search like Coq's [213]. An advantage of our current algorithm is that we do not require values in scope to be specially annotated

to be eligible, but an eligibility annotation similar to Scala's **implicit** (see Section 2.1.4) can be used to limit the complexity of a more powerful inference mechanism. Another advantage of our approach is that it does not introduce a separate type-level computational model in the language.

2.2.5 Formal developments

In Appendix 2.A, we formally develop instance arguments, based on the formalism that Norell uses to present the Agda language [161]. We formally define functions with instance arguments, how values for them are type-checked, when values for instance arguments are inferred and the rules for this resolution. We discuss various technical points and present a soundness result.

2.2.6 Implementation

We have implemented the above proposal in Agda. Our implementation is surprisingly cheap, with a non-context-diff of the initial implementation of about 750 lines. A comparison of line counts is not necessarily objective and can only give a partial view, but for what it's worth: the initial diff of Sozeau and Oury's Coq type classes [213] was ∼2k lines long. Our implementation is part of Agda version 2.3.0 and later versions.

2.3 Monads case study

Instance arguments provide an alternative for type classes. Although they lift some of the limitations of type classes, our inference algorithm is less powerful than Haskell's (see Section 2.4). To demonstrate that our mechanism is at least powerful enough for common use cases of type classes, we take a look at a typical type class example: monads. In this section, we demonstrate that with our instance arguments, we can use an Agda version of *Monad*s in a way similar to Haskell.

We use a simplified version of the monads from Agda's standard library. Although instance arguments can also be used conveniently with the original version, we do not use it here because it is complicated by its support for indexed monads. We have kept the universe level argument f but it can be safely ignored by readers not familiar with Agda's universe polymorphism. The $\forall$ in the type signatures is Agda shorthand syntax for arguments whose name is given but whose type is omitted. For example the type $\forall\ \{A\} \to A \to M\ A$

is shorthand for $\{A : \_\} \to A \to M\ A$, i.e. names between the $\forall$ symbol and the first subsequent arrow are interpreted as names of arguments whose type should be inferred.

```
record Monad {f : Level} (M : Set f → Set f) : Set (suc f) where
  infixl 1 _>>=_ _>>_
  field  return : ∀ {A} → A → M A
         _>>=_  : ∀ {A B} → M A → (A → M B) → M B
  _>>_ : ∀ {A B} → M A → M B → M B
  m₁ >> m₂ = m₁ >>= λ _ → m₂
```

We see that *Monad* contains the basic monadic operators *return* and _>>=_ as fields and provides the _>>_ operation. In order to highlight correspondence with Haskell's monads, we include a syntax definition (a form of restricted macro) for a form of do-notation. This addition is orthogonal to the use of instance arguments.

```
bind : ∀ {A B} → M A → (A → M B) → M B
bind {A} {B} = _>>=_ {A} {B}
syntax bind m (λ x → c) = do x ← m then c
```

We will assume some type constructors and *Monad* instances for them (similar to those defined in the Agda standard library): a state monad with mutable state variable of fixed type $\mathbb{N}$, a monad instance for a type constructor $\_\bot : Set \to Set$ of partial computations and the list monad:

```
postulate stateMonad     : Monad (State ℕ)
          partialityMonad : Monad _⊥
          listMonad       : Monad List
```

In current Agda, the most convenient way to use these monad instances, is to apply the *Monad* module to the correct instance at the location where it is used.

```
test1 : ℕ → ℕ ⊥
test1 k = let open Monad partialityMonad in
          do x ← return k then
             if (equal x 4) then return 10 else never
```

This code does not look too bad actually. Opening a monad instance's module brings into scope just the definitions of the monadic operations we need. However, it becomes more difficult if we decide that we need to use, for example, the monadic bind operator on a list, requiring monadic operations from two different

instances. In this case, current Agda requires us to rename the operations for one of the instances:

```
postulate nToList : ℕ → List ℕ
test₂ : ℕ → (List ℕ) ⊥
test₂ k =
  let open Monad partialityMonad
      open Monad listMonad using () renaming (_>>=_ to _>>=ₗ_) in
  do x ← return [k] then
     if (equal k 4) then return (x >>=ₗ nToList) else never
```

We can improve upon this situation using instance arguments. First, we bring the definitions from the *Monad* {{ ... }} module application into scope. As explained in Section 2.2.2, the {{...}} syntax turns the *Monad* module's last non-implicit argument into an instance argument. We can then define our examples in a simpler way and let Agda infer the correct values for the instance arguments.

```
open Monad {{...}}
test₁ : ℕ → ℕ ⊥
test₁ k = do x ← return k then
             if (equal x 4) then return 10 else never
test₂ : ℕ → (List ℕ) ⊥
test₂ k = do x ← return [k] then
             if (equal k 4) then return (x >>= nToList) else never
```

In the case of $test_1$, one could argue that we don't actually gain all that much. Agda now automatically chooses the correct monad instance from the values in scope instead of requiring the programmer to make this choice. However, the second example shows that in a case where we use monadic operations from different monad instances, instance arguments effectively spare us some uninteresting bookkeeping, by inferring the instances in the background.

2.4 Instance resolution

We provide a precise definition of the resolution algorithm in Appendix 2.A.2, but in this section, we discuss and motivate the primary design choices.

2.4.1 Non-recursive

Our resolution algorithm is only a restricted analogue to Haskell's. The mechanism is designed such that a type-directed scope-based resolution will not recursively trigger further resolutions (see Appendix 2.A.3). This limitation is a deliberate choice. We thus avoid the introduction of a separate computational model through the instance search mechanism, as for Scala implicits or Haskell and Coq type classes. This decision however does unavoidably limit the functionality of our mechanism. For example, for the *Eq* type introduced in Section 2.2, we could have defined:

```
listEq : {A : Set} → Eq A → Eq (List A)
listEq {A} eqA = record {equal = eq′}
  where eq′ : List A → List A → Bool
        eq′ []       []       = true
        eq′ (a :: as) (b :: bs) = equal eqA a b ∧ eq′ as bs
        eq′ _        _        = false
```

With the *eqBool* value from Section 2.2 in scope, one might expect an instance of *Eq* (*List Bool*) to be automatically inferred as *listEq eqBool*. This is not the case for our system; we require the user to explicitly construct a value of the correct type himself. It suffices to bring this value in scope at the call site, for example, by placing it in a local **where** block.

```
test = equal (true :: false :: true :: []) (true :: false :: [])
  where listBoolEq = listEq eqBool
```

2.4.2 Resolving instances' implicit arguments

In the previous section, we have used examples that were simplified from definitions in the Agda standard library. Specifically, the types of the imported values *partialityMonad* and *listMonad* were simplified from their definitions in the Agda standard library:

```
partialityMonad : {l : Level} → Monad (_⊥ {l})
listMonad       : {l : Level} → Monad (List {l})
```

These definitions exploit Agda's universe polymorphism. The term _⊥ is not a functor of type *Set* → *Set*, but instead, for any *level l*, _⊥ {*l*} is a functor of type *Set l* → *Set l*. This means that partial computations can be defined producing values (Set 0), types (Set 1), kinds (Set 2), and for each of these types of partial computations, a monad *instance* is provided as *partialityMonad* {*l*}.

There is a fine balance in the definition of the resolution mechanism that allows us to use the same examples from the previous section with these more complex types of monads, without compromising our choice for a limited-power resolution mechanism. In the calls to the monadic operations *return* and $\_\gg\!=\_$, instance arguments would be resolved for types $Monad\ (\_\bot\ \{zero\})$ and $Monad\ (List\ \{zero\})$ even though no values of these types were in scope. The reason that this works is that our instance resolution mechanism does not only consider values that are of the correct type directly, but will also consider values that are of the correct type after the application to a number of *implicit* arguments. This choice in our resolution mechanism was made after the appearance of the conference version of our paper and attempts to strike a fine balance between our desire to limit the resolution mechanism's power but still minimise programmer work and confusion in using the feature. It is important that only values expecting *implicit* arguments are considered, not values expecting *instance* arguments, since that would make instance resolution recursive and more powerful.

2.4.3 When is a candidate instance rejected?

Another fine balance in our design is related to the fact that equality of types is harder to decide in a dependently-typed language like Agda than in Haskell. Let us reconsider the monads case study (see Section 2.3), where *return* and $\_\gg\!=\_$ have (essentially) the following types:

$$\begin{aligned} &return \;\; : \forall\ \{M\}\ \{\!\{Mon\ :\ Monad\ M\}\!\}\ \{A\} \to A \to M\ A \\ &\_\gg\!=\_ \;\; : \forall\ \{M\}\ \{\!\{Mon\ :\ Monad\ M\}\!\}\ \{A\ B\} \to \\ &\qquad\qquad M\ A \to (A \to M\ B) \to M\ B \end{aligned}$$

When type-checking a term like $return\ 3 \gg\!= nToList$, our version of Agda will silently insert implicit and instance arguments. Specifically, it will insert type functors M_1 and $M_2 : Set \to Set$, monad instances $m_1 : Monad\ M_1$, $m_2 : Monad\ M_2$ and result types A_1, A_2 and $A_3 : Set$:

$$\_\gg\!=\_\ \{M_1\}\ \{\!\{m_1\}\!\}\ \{A_1\}\ \{A_2\}\ (return\ \{M_2\}\ \{\!\{m_2\}\!\}\ \{A_3\}\ 3)\ nToList$$

From the function applications and the types of literal $3 : \mathbb{N}$ and function $nToList : \mathbb{N} \to List\ \mathbb{N}$, Agda can determine the following equations:

$$\begin{aligned} A_3 &= \mathbb{N} \\ M_2\ A_3 &= M_1\ A_1 \\ A_2 &= \mathbb{N} \\ M_1\ A_2 &= List\ \mathbb{N} \end{aligned}$$

In a similar situation, a Haskell compiler would be able to exploit properties of Haskell type-constructor application to deduce $M_1 = \mathit{List}$, $A_1 = \mathbb{N}$ and $M_2 = \mathit{List}$, but the corresponding properties do not hold in more complex dependent type-systems like Agda's.

The reason that our modified version of Agda is able to type-check the term $\mathit{return}\ 3 \gg\!= \mathit{nToList}$ is that it will try to instantiate instance arguments m_1 and m_2 with the monad instances in scope and check that they do not (immediately) invalidate constraints. When instantiating m_1 with, for example, *partialityMonad*, this will imply $M_1 = \_\bot$, violating the equality constraint $M_1\ A_2 = \mathit{List}\,\mathbb{N}$. As a result, *partialityMonad* will be rejected as a candidate value for m_1, leaving only *listMonad* as a candidate. That unique candidate will be chosen, it will imply a solution for M_1 and type-checking can continue.

This example shows that even though instance resolution is inherently more difficult in Agda, because of the inherently more complex notion of type equality, the check that candidate values do not violate constraints during instance resolution suffices to solve the problem for examples like our monads case study.

2.4.4 Why limit the computational power?

In the above example concerning list equality, our resolution algorithm was not smart enough to automatically infer instance arguments that one might expect it to. Help from the programmer is required to make it find the correct value, although explicitly passing the instance arguments wherever they are used is not necessary: it suffices to place the required value in scope.

First, we believe that instance arguments can be combined with a smarter resolution algorithm if desired. Extensions can be imagined where functions like *listEq* are annotated somehow to make the resolution consider them. Such an extension would be largely orthogonal to the rest of our design.

However, introducing such an extension makes the instance search recursive. Even if it can still be kept decidable with restrictions on the functions considered, it inevitably exposes an additional computational model, similar to Haskell's, Scala's or Coq's instance resolution. In these other systems, the unification-based Prolog-like computational models were added as a natural component of the ad hoc polymorphism primitive. The fact that this added a separate type-level computation primitive was a byproduct rather than a design goal. Functional dependencies, an extension of Haskell's type class resolution model (see Jones [111] and our Section 2.5.2), at first seemed natural too, but many now consider it preferable to use the more functional type families (see Schrijvers et al. [204]) instead.

The computational model exposed by a recursive instance search is programmer-extensible, type-level and supports (open) pattern matching on types. We see such a primitive as an important and powerful part of a programming language. It is, for example, essential for datatype-generic programming. As such, we argue that the design of such a feature should be made consciously and well thought through. The fact that a Prolog-like model seems natural for an ad hoc polymorphism feature, should not blind us to the fact that there could be alternatives that can also be combined with an ad hoc polymorphism feature. Foundational calculi like Girard's System F_ω [80] but also dependently-typed calculi and languages (e.g. Martin-Löf's type theory [140] or Coquand's calculus of constructions [44]) suggest that a suitable type-level computation model is not necessarily Prolog-like, but can very well be functional (with possible advantages like a better understood meta-theory, simpler implementations and more consistency in the language).

Especially in the dependently-typed language Agda, where a powerful and well-understood form of type-level computation is a core part of the language, it would be preferable if we could combine an ad hoc polymorphism primitive with this existing model instead of introducing a special-purpose additional one. However, to support standard use cases of parametric instances, the model would need to be extended with a way to pattern match on types. This implies a form of typed meta-programming, a topic that is the subject of a lot of recent research. We point the interested reader to some related work in this area by Chapman et al. [33], Danielsson [48], Chapman [32], McBride [144] and ourselves (see Chapter 4).

Because of this reasoning, our choice in this paper to not introduce a powerful computational model as part of our resolution algorithm is a conservative one. In summary, the main reasons for our choice are the following:

- We believe there are possible alternatives to the choice made in other proposals (parametric instances with a (decidable or not) Prolog-like computation model). By not committing to a Prolog-like model now, we maintain the possibility of combining with better, alternative models later.
- Our design and implementation shows that coupling ad hoc polymorphism to a powerful type-level computation primitive is *not* essential; a useful form of ad hoc polymorphism is possible without it (although it is more limited).
- Our low-power resolution scheme has disadvantages (e.g. not automatically producing instances like *eqList eqNat*) but also benefits, as we discuss in the next section.

2.4.5 Advantages and disadvantages

Note also that our simple resolution scheme has some advantages of its own. We have used it for all of the examples in Section 2.3 and 2.5 and have found the resolution practical, predictable, intuitive and sufficient. We do not need to limit resolution complexity by requiring candidate values to be annotated specially, but instead we consider all values in scope. This lowers the impact of our feature on users' code and makes, for example, the ellipsis in Section 2.5 more widely usable. Its intuitive meaning changes from "Fill in this value from an *annotated* value in scope" to "Fill in this value from the scope", which feels more natural to us.

Note that because the entire scope is considered for resolution, it is up to the programmer to make sure that only a single value of a correct type is in scope. Instance arguments should only be used on types which are informative enough so that they typically identify values uniquely. If there still is a conflict, existing features in Agda's module system (e.g. **hiding** and **using** modifiers) can be used to control the scope. Values that can only be named using a qualified reference are not considered for instance resolution. Finally, the programmer must also remember that values may be implicitly used because of instance resolution, despite the fact that they are never referenced from the actual code. In our experiments, we find that instance arguments provide a solution (ad hoc overloading) for many of the name conflicts that arise in typical use of Agda's standard library (e.g. $\_ \stackrel{?}{=} \_$ in *Data.Nat*, *Data.Bool* etc.) and that type conflicts for reasonably typed instance arguments occur seldom.

In some cases, it is useful to limit the set of eligible values for instances to be resolved. For example, Scala will in some contexts implicitly convert a value of type A to type B if a value of type $A \rightarrow B$ is in scope and has been marked as eligible for implicit resolution. Because we do not require an explicit eligibility annotation, this would not work well in our system because there may be functions of type $A \rightarrow B$ in scope that should not be used in such implicit conversions. Nevertheless, an alternative for our system would be to do such implicit conversion based on a value of type *ImplicitConversion A B*, a type that simply wraps the actual conversion function. Instead of specially marking a conversion function as eligible for instance resolution, we would then just wrap it into a value of type *ImplicitConversion A B* to obtain the same effect. This has the additional advantage that such functions are then not eligible for situations where a function is needed for a purpose other than an implicit conversion. Generally, this simple pattern can be used instead of eligibility annotations when they are needed.

2.5 Some advanced patterns

It turns out that our relatively simple extension of Agda can support analogues or variants of many features which have required non-trivial implementation efforts in Haskell, as well as some new patterns of its own. In this section, we discuss a selection of such topics.

2.5.1 Standing on the shoulders of records

We discussed in the introduction how modelling type classes using an existing powerful record mechanism such as Agda's dependently typed records makes certain features available "for free" where they require separate extensions for Haskell type classes. Sozeau and Oury and Oliveira et al. have previously demonstrated this observation for Coq type classes (which are Coq dependently typed records underneath [213]) and Scala implicits (where type classes are typically modelled as Scala traits [167]).

One such feature is the equivalent of Haskell's associated type families [204]. An associated type family is essentially a type class member that is a type or type functor. Using a dictionary model of a type class in a dependently typed language, there is nothing special about records with members that are not just values and we essentially get associated type families for free. We can, for example, model the generic finite maps discussed by Chakravarty [31] after Hinze [95] and Hinze et al. [96] as follows:

$$
\begin{aligned}
&\textbf{record}\ \mathit{GMapKey}\ (K\ :\ \mathit{Set})\ \textbf{where}\\
&\quad \textbf{field}\ \mathit{GMap}\ :\ \mathit{Set} \to \mathit{Set}\\
&\qquad\quad \mathit{empty}\ :\ \{\, V\ :\ \mathit{Set}\,\} \to \mathit{GMap}\ V\\
&\qquad\quad \mathit{lookup}\ :\ \{\, V\ :\ \mathit{Set}\,\} \to K \to \mathit{GMap}\ V \to \mathit{Maybe}\ V\\
&\qquad\quad \mathit{insert}\ :\ \{\, V\ :\ \mathit{Set}\,\} \to K \to V \to \mathit{GMap}\ V \to \mathit{GMap}\ V\\
&\mathit{natGMapKey}\ :\ \mathit{GMapKey}\ \mathbb{N}\\
&\mathit{natGMapKey}\ =\ \textbf{record}\ \{\, \mathit{GMap}\ =\ \mathit{NatMap}\\
&\qquad\qquad\qquad\qquad\quad \mathit{empty}\ =\ \mathit{NatMap.empty}\\
&\qquad\qquad\qquad\qquad\quad \mathit{lookup}\ =\ \mathit{NatMap.lookup}\ k\ m\\
&\qquad\qquad\qquad\qquad\quad \mathit{insert}\ =\ \mathit{NatMap.insert}\,\}
\end{aligned}
$$

Another feature which we get for free has been described for Haskell by Orchard and Schrijvers as *constraint families* [174]. A constraint (synonym) family in Haskell is a member of a type class that represents a class constraint on a type class's parameters and/or other types. Using a dictionary model of type classes, this concept actually reduces to type families. Orchard and Schrijvers' example

of constrained functors (functors whose *fmap* function is restricted to types in a certain type class) can be modelled as follows:

```
record ConstrainedFunctor (F : Set → Set) : Set₁ where
  field Constraint : Set → Set
        fmap : {A B : Set} → {{ctA : Constraint A}} →
               {{ctB : Constraint B}} → (A → B) → F A → F B

listConstrainedFunctor : ConstrainedFunctor List
listConstrainedFunctor = record {Constraint = λ _ → ⊤
                                 ; fmap      = List.map}

postulate TreeSet : Set → Set
          Ord : Set → Set
          mapTreeSet : {A B : Set} {{ordA : Ord A}} {{ordB : Ord B}} →
                       (A → B) → (TreeSet A → TreeSet B)

treeSetConstrainedFunctor : ConstrainedFunctor TreeSet
treeSetConstrainedFunctor = record {Constraint = Ord
                                    ; fmap      = mapTreeSet}
```

2.5.2 Multi-parameter type classes and functional dependencies

A multi-parameter type class in Haskell is a type class with more than one parameter. The equivalent in our approach would be an instance argument of a record type with more than one parameter, something which is clearly allowed in our system. Functional dependencies in a multi-parameter type class are annotations which indicate that certain parameters of a type class can be deduced from (a subset of) the others [111]. Such an annotation cannot directly be provided in our framework. However, in this section, we highlight certain behaviour of our system that is reminiscent of using functional dependencies, even though it works differently under the hood.

Consider the following code, which uses the *IsDecEquivalence* record from module *Relation.Binary* in the Agda standard library. We use explicit **using** declarations to avoid certain name clashes, but also to make it more clear what is happening implicitly.

```
open import Relation.Binary
  using (module DecSetoid; module IsDecEquivalence)
open import Data.Bool using (false; true; decSetoid)
open DecSetoid decSetoid using (isDecEquivalence)
```

```
open IsDecEquivalence {{...}} using (_≟_)
test = false ≟ true
```

The *IsDecEquivalence* t _ ≈ _ record is semantically a more developed version of the record *Eq* from Section 2.1. It contains essentially an equality decision procedure _ ≟ _ for a binary predicate _ ≈ _ on type t (as well as proof that _ ≈ _ is an equivalence relation). The field _ ≟ _ has the following type:

```
_≟_ : (a : t) → (b : t) → Dec (a ≈ b)
```

A value of type *Dec* $(a \approx b)$ contains either a proof that $a \approx b$ or a proof that $a \not\approx b$. We can bring a value of type *IsDecEquivalence Bool* _ ≡ _ in scope by importing *Data.Bool* and opening the *decSetoid* record (this would be more convenient if *isDecEquivalence* were exported directly by the *Data.Bool* module). Finally, we bring the new record field projection operator (taking the record as an instance argument) into scope by importing it from the *IsDecEquivalence* {{...}} module application (see Section 2.2.2). From that point on, we can transparently use the function _ ≟ _ on *Bool*s, as demonstrated in the definition of *test*.

A first thing to explain is that the *IsDecEquivalence* record takes two arguments, making it the equivalent of a multi-parameter type class. It is interesting to consider what happens when type-checking the definition of *test*. The function _ ≟ _ has the following type (ignoring universe polymorphism):

```
_≟_ : {A : Set} → {_≈_ : A → A → Set} →
      {{isDE : IsDecEquivalence A _≈_}} →
      (a : A) → (b : B) → Dec (a ≈ b)
```

When *false* ≟ *true* is type checked, Agda infers that A = *Bool* from the arguments of _ ≟ _. It then infers the instance argument *isDE* from the local scope. The only candidate value in scope is *isEquivalence*, typed *IsDecEquivalence Bool* _ ≡ _. From unifying the type of this value with the expected type of *isDE*, Agda infers that the implicit argument _ ≈ _ must be the binary predicate _ ≡ _.

In this case, we see that one argument of the *IsDecEquivalence* type constructor already uniquely determines the value to be used from the scope. Its other arguments can then be inferred from this value, producing an effect similar to a hypothetical situation where *IsDecEquivalence* were a multi-parameter type class with a functional dependency from type A to binary predicate _ ≈ _.

Nevertheless, our mechanism works very differently from Haskell type classes with functional dependencies. First of all, nowhere have we declared any

functional dependencies between arguments of the *IsDecEquivalence* record type, and these dependencies were not checked when we brought values of type *IsDecEquivalence* into scope. Only when we actually needed to infer an instance argument, was it checked that only a single suitably-typed value was in scope.

Declaring the equivalent of a functional dependency on the *IsDecEquivalence* record type's arguments would correspond to an assertion that only one decidable equality predicate can exist for any given type A. Such an assertion would be wrong here and would cause problems in scenarios where multiple such predicates are used together. Our system manages to infer the value of the $\_ \approx \_$ predicate without such a dependency, because only one value of type *IsDecEquivalence Bool* $\_ \approx \_$ is *in scope* at the call site of $\_ \stackrel{?}{=} \_$, which is a much weaker requirement.

Note finally that it is a value, not a type, that is being inferred in a functional dependencies-like way. In fact, our mechanism does not make any fundamental distinction between types or values, which is what one might expect in a dependently-typed language like Agda. The mechanism will even happily infer types from values, which is not possible in Haskell.

2.5.3 Implicit configurations

One pattern implemented in the context of type classes which is rendered almost trivial in the context of our proposal is Kiselyov and Shan's implicit configurations [118]. The authors discuss a solution to what they call the *configurations problem*: propagating run-time preferences throughout a program, allowing multiple concurrent configuration sets to coexist safely under statically guaranteed separation. Their main example concerns modular arithmetic: they want to be able to build expressions in modular arithmetic which are parameterised over a concrete modulus but without the need to pass the modulus around explicitly. They also want static assurance that the same modulus is used for all operations in such an expression.

Kiselyov and Shan's solution is based on a mix of phantom types, type classes and type-level computation. We demonstrate that a simpler encoding is possible in our system, and that we can fully avoid one of the main difficulties in their work: the reflection at type-level of run-time values. Let us suppose that we have a signature like the following: we assume an *Integral* dictionary record and *add*, *mul* and *mod* operations taking such a dictionary as an instance argument. We also assume we have a type N containing values *zero*, *one*, *two* and *three* and a dictionary *nInt* of type *Integral N*.

```
postulate Integral : Set → Set
          add : ∀ {A} ⦃intA : Integral A⦄ → A → A → A
          mul : ∀ {A} ⦃intA : Integral A⦄ → A → A → A
          mod : ∀ {A} ⦃intA : Integral A⦄ → A → A → A
          N : Set
          zero one two three : N
          nInt : Integral N
```

Like Kiselyov and Shan, we define a wrapper data type *M s A* parameterised by phantom token *s* (in our case not a type but a value of opaque type *Token*) and type *A*. This wrapper represents a value of type A that is being considered under an unspecified modulus. We also define a dictionary record *Modulus s A* (also parameterised by token s and type *A*) representing a modulus of type *A*.

```
private postulate Token : Set
record Modulus (s : Token) (A : Set) : Set where
  field modulus : A
data M (s : Token) (A : Set) : Set where
  MkM : A → M s A
unMkM : ∀ {s A} → M s A → A
unMkM (MkM a) = a
```

The *withModulus* function, which instantiates a value *M s A* for a specified modulus, is simpler in our setting than Kiselyov and Shan's because we don't have to bother with constructing a type for which the Modulus instance returns a certain value, but instead just pass the desired dictionary explicitly:

```
private postulate theOnlyToken : Token
withModulus : ∀ {A} → ⦃intA : Integral A⦄ → (modulus : A) →
                (∀ {s} → ⦃mod : Modulus s A⦄ → M s A) → A
withModulus m f =
  unMkM (f {theOnlyToken} ⦃record {modulus = m}⦄)
```

Similar to Kiselyov and Shan's, the addition and multiplication functions unwrap the values, apply the respective operation, apply the modulus *m* and re-wrap the result:

```
normalise : ∀ {s A} ⦃intA : Integral A⦄ ⦃mod : Modulus s A⦄ →
              A → M s A
normalise a = MkM (mod modulus a)
_+_ : ∀ {s A} ⦃intA : Integral A⦄ ⦃mod : Modulus s A⦄ →
```

```
        M s A → M s A → M s A
(MkM a) + (MkM b) = normalise (add a b)
_*_ : ∀ {s A} ⦃intA : Integral A⦄ ⦃mod : Modulus s A⦄ →
        M s A → M s A → M s A
(MkM a) * (MkM b) = normalise (mul a b)
```

These operators are used similarly to Kiselyov and Shan's:

```
test₁ : N
test₁ = withModulus two (let o = MkM one in (o + o) * (o + o))
testExpr : ∀ {s} → ⦃mod : Modulus s N⦄ → M s N
testExpr = let o = MkM one
               t = MkM two
           in (o + t) * t
test₂ : N
test₂ = withModulus three testExpr
```

With this, our encoding of Kiselyov and Shan's implicit configurations is complete. We believe that we achieve the same goals as Kiselyov and Shan, but we avoid their threading of values into types (through an involved type-level reflection of values) and back again (through a form of type-level computation), which seems unneeded, very complex and possibly inefficient (depending on what optimisations the compiler can perform). Interestingly, the fact that we don't need to reflect values at the type level is not (as one might expect) a consequence of Agda's dependently typed nature. Instead, it is the value-level representation of dictionaries which allows this greater simplicity. More concretely, in the definition of *withModulus* above, we can construct the dictionary as a value and pass it explicitly to the computation, whereas Kiselyov and Shan need to jump through a lot of hoops to construct a type for which the correct instance will be inferred. Kiselyov and Shan realise that this is non-ideal and propose adding a restricted form of local instances to Haskell, which we support in a more general form (see Section 2.5.5).

2.5.4 Implicit proof obligations

In the context of Agda, we believe that instance arguments are useful for a pattern which is (to the best of our knowledge) novel: implicit proof obligations. Consider the integer division operator in module *Data.Nat.DivMod* in Agda's standard library:

```
_div_ : (dividend divisor : ℕ) {≢0 : False (divisor ≟ 0)} → ℕ
```

This division operator requires a guarantee that the provided divisor is non-zero. However, instead of requiring a normal argument of type $divisor \not\equiv 0$, the $\_div\_$ operator cleverly accepts a value of type $False\ (divisor \stackrel{?}{=} 0)$. This type contains a single value if and only if *divisor* is non-zero, but additionally, this value can be automatically inferred if Agda knows that *divisor* is of the form $suc\ n$ for some n. For example, if we write $5\ div\ 3$, then Agda will infer the non-zeroness proof obligation. This pattern has been described by Norell [161, §3.7.1 p.71], and critically depends on the fact that the property in question (non-zeroness) can be decided. Proof obligations modelled using this pattern are not passed on implicitly to other methods that require it. Finally, the $\_div\_$ operator becomes somewhat clumsy to use in a situation where only a "normal" proof, i.e. a value of type $divisor \not\equiv 0$ is available.

We propose an additional operator $\_div'\_$ which takes the proof obligation as an instance argument (we omit the definition in terms of the above $\_div\_$). This operator does not have the limitations of the $\_div\_$ operator discussed above, but does have some limitations of its own: for example, in the call $5\ div'\ 3$, Agda can only infer the implicit argument of our operator if a value of type $3 \not\equiv 0$ is in scope.

```
_divMod'_ : (dividend divisor : ℕ) ⦃ _ : divisor ≢ 0 ⦄ → ℕ × ℕ
_divMod'_ =  -- omitted
_div'_ : (dividend divisor : ℕ) ⦃ _ : divisor ≢ 0 ⦄ → ℕ
a div' b with a divMod' b
a div' b | (q , _) = q
postulate d : ℕ
          d≢0 : d ≢ 0
test : ℕ
test = 5 div' d
```

Note how in the definition of $\_div'\_$, the proof obligation is implicitly passed on to the $\_divMod'\_$ function, which also requires it. We believe that this example shows that our proposed instance arguments have uses that go beyond those of type classes. *Not only dictionary records can be usefully passed around implicitly but also other values that are uniquely identified by their type in call-site scopes.* In a dependently typed language like Agda, implicit proof obligations are a clear example of such values.

2.5.5 Local instances

A feature that is not supported by Haskell type classes are local type class instances. Consider the following two equality functions on *String*s: the first

represents standard equality and the second only compares the lengths. The first definition uses the standard string equality decision procedure and the second applies the *EqInst.equal* operator after first applying a string length function to its two arguments. Note that we assume a single, standard value of type $Eq\ \mathbb{N}$ in scope.

$$
\begin{aligned}
&eqString_1 \ :\ String \to String \to Bool\\
&eqString_1\ s_1\ s_2\ =\ \lfloor\ s_1 \stackrel{?}{=} s_2\ \rfloor\\
&eqString_2\ :\ String \to String \to Bool\\
&eqString_2\ =\ equal\ on\ length
\end{aligned}
$$

Now suppose that we have a function whose behaviour depends on a configuration argument, determining which type of equality it should use throughout a series of calculations. We can support this by defining the equivalent of a *local instance eqLocal* of the *Eq type class*, which uses the correct string equality operator, depending on the configuration parameter.

$$
\begin{aligned}
&test\ :\ Bool \to Bool\\
&test\ lengthEq\ =\ if\ equal\ \texttt{"abcd"}\ \texttt{"dcba"}\ then\ ...\ else\ ...\\
&\quad \textbf{where}\ eqLocal\ =\ \textbf{record}\ \{\\
&\qquad\qquad equal\ =\ if\ lengthEq\ then\ eqString_2\ else\ eqString_1\ \}
\end{aligned}
$$

The value *eqLocal* functions as a local type class instance, something which is also supported by Scala implicits, but not by Haskell or Coq type classes, where type class instances are always global.

It is interesting to note that Wadler and Blott already considered local instances when they defined type classes in 1989. In fact, the simplified language in their appendix, for which they present typing judgments (with an embedded dictionary translation) supports local instances. However, Wadler and Blott show that this calculus does not in general have the principal types property. This property states that any expression e that can be typed in a given context, must have a principal type t, i.e. a type t such that if e has any type t', it must be an instance of t.

In a hypothetical notation for local instances and using the Haskell *Eq* type class from our introduction, this is Wadler and Blott's counter-example e_1:

$$
\begin{aligned}
e_1\ =\ &\textbf{let}\ instance\ Eq\ Int\quad \textbf{where}\ ...\\
&\qquad instance\ Eq\ Char\ \textbf{where}\ ...\\
&\ \ \textbf{in}\ equal
\end{aligned}
$$

They show that (with no other instances available) e_1 has types $Int \to Int \to Bool$ and $Char \to Char \to Bool$ but no type generalising both. Wadler and

Blott conjecture that the problem does not exist if no local instances or local type class definitions are allowed, and this restriction has been adopted in the Haskell language.

In their work about implicit configurations, Kiselyov and Shan recognise that a major part of their development consists of "type system hackery" that works around the lack of local instances in Haskell. As a remedy, they propose to allow local instances under the restriction that their types must mention a simultaneously introduced fresh type variable, a solution which they claim salvages the principal types property, yet suffices for the purposes of their implicit configurations paper.

Note, that a dependently-typed language like Agda typically does not have the principal types property to begin with. An example is the term $(3, x)$ with x of type $Fin\ 3$. Since Agda has dependent sums, this term can be given the type $\Sigma\ \mathbb{N}\ (\lambda\ n\ \rightarrow\ Fin\ n)$ as well as $\Sigma\ \mathbb{N}\ (\lambda\ n\ \rightarrow\ Fin\ 3)$. Agda will refuse to type-check such terms unless the user provides additional information.

The above problem does not exist in our design because the analogue of the above term e_1 does not have any type at all in our system, so that it would even (vacuously) satisfy the principal types property:

$$
\begin{aligned}
e_1\ =\ &\mathbf{let}\ eqInt\ :\ Eq\ Int\\
&\quad\ \ eqInt\ =\ ...\\
&\quad\ \ eqChar\ :\ Eq\ Char\\
&\quad\ \ eqChar\ =\ ...\\
&\mathbf{in}\ equal
\end{aligned}
$$

This term e_1 does not have any type, because there is no *unique* type-correct value in scope for *equal*'s instance argument and Agda will produce a type error. The term definitely does not have the type $\{t\ :\ Set\}\ \rightarrow\ \{\!\{eqT\ :\ Eq\ t\}\!\}\ \rightarrow\ t\ \rightarrow\ t\ \rightarrow\ Bool$, because in our system we do not infer instance arguments when an instance argument cannot be filled in. We can however change the term so that it does have this type by preventing the instance argument from being resolved as follows:

$$e_1\ =\ \mathbf{let}\ ...\ \mathbf{in}\ \lambda\ \{t\}\ \{\!\{eqT\}\!\}\ \rightarrow\ equal\ \{t\}\ \{\!\{eqT\}\!\}$$

For this expression, Agda will not attempt to resolve the instance argument.

2.5.6 Overloaded literals

Another Haskell feature related to type classes that we have not yet discussed are *overloaded literals*. This term refers to the fact that, for example, the integer

literal *3* in Haskell source code does not have type *Integer* as one might expect, but instead it has type $\forall\ a.\ Num\ a\ \Rightarrow\ a$. More specifically, the *Num* class has a method $fromInteger :: Num\ a\ \Rightarrow\ Integer\ \rightarrow\ a$ and integer literals are interpreted to stand for an application of this function to the corresponding *Integer* value and a similar system exists for rational literals. Numeric literals are defined in this indirect way so that they may be interpreted as values of any appropriate numeric type.

Let us consider an analogous system for Agda, based on instance arguments. The Agda compiler currently parses four kinds of literals: positive integers (naturals), (positive) floating points, characters and strings and they are considered of builtin types $\mathbb{N}$, *Float*, *Char* and *String* respectively.[3] Like for Haskell, it seems useful to define a system such that literals can be interpreted in any suitable numeric type.

However, there is a problem with Haskell's idea of overloaded literals that we have to tackle first. Consider the example of natural literals. When such a literal is interpreted in a numeric type a, a user might expect the compiler to check that the literal value actually fits inside the type a. However, this is not the case, as the reader can verify by executing the Haskell expression $print\ (4294967296 :: Int)$ or $print\ (18446744073709551616 :: Int)$ on a 32- or 64-bit system respectively. Indeed, it compiles without any warning and because of a machine integer overflow produces output *0*. The problem appears hard to solve in general: for an arbitrary user-defined type, it is unclear how the compiler should figure out whether a given value is reasonable.

In Agda, this problem is arguably worse, because Agda has a richer type system. For example, Agda's standard library defines a family of types $Fin : \mathbb{N}\ \rightarrow\ Set$ such that $Fin\ n$ models the set of numbers between *0* and $n - 1$. It would be useful to be able to interpret integer literals at this type, but then we would even more like to receive a compiler warning if, for example, the literal *5* is interpreted in $Fin\ 3$.

Luckily, there is an Agda programming idiom that we can use. We have already encountered the technique in Section 2.5.4, where the *_div_* function was defined in such a way that the non-zeroness of its divisor argument was statically checked. Similarly, the Agda standard library defines a function #_ that statically checks that a natural m is strictly smaller than n before converting it to a value of type $Fin\ n$.

$$\#\_ : \forall\ m\ \{n\}\ \{m < n\ :\ True\ (m\ N <?\ n)\}\ \rightarrow\ Fin\ n$$
$$\#\_\ \_\ \{m < n\ =\ m < n\}\ =\ from\mathbb{N}\leqslant\ (toWitness\ m < n)$$

[3] These types need to be defined and specially marked in the user's code or an imported file. Agda's standard library currently defines suitable versions of $\mathbb{N}$, *Char* and *String* but not *Float*.

This type signature of #_ ensures that when we type-check, for example, $\#\ 2\ :\ \mathit{Fin}\ 3$, Agda will infer a value of type $\mathit{True}\ (2\ N{<}?\ 3)$. This works because the type reduces to $\mathit{True}\ (\mathit{yes}\ \mathit{prf})$ for some $\mathit{prf}\ :\ 3 \leqslant 3$ which reduces further to the unit type $\top$. For this type, Agda infers the single value $\mathit{tt}\ :\ \top$. Conversely, if we type-check $\#\ 5\ :\ \mathit{Fin}\ 3$, then the type $\mathit{True}\ (5\ N{<}?\ 3)$ will reduce to the bottom type $\bot$, and the compiler will report that it cannot infer a value for it.

This technique can be generalised to a type-class for types that support natural literals as follows:

```
record HasNatLiterals (t : Set) : Set₁ where
  field ValidLiteral : ℕ → Set
        decideValid  : (n : ℕ) → Dec (ValidLiteral n)
        fromℕ        : (n : ℕ) → {valid : True (decideValid n)} → t
```

This record models a type class of types t that support natural literals. It is required to contain a predicate *ValidLiteral* identifying what values can legally be used as literals, a decision procedure *decideValid* for this predicate (a function that returns, for any n, a value of $\mathit{Dec}\ (\mathit{ValidLiteral}\ n)$, i.e. a proof of either $\mathit{ValidLiteral}\ n$ or its negation $\neg\ \mathit{ValidLiteral}\ n$) and a function *from*ℕ that takes a natural n, a proof that n satisfies the predicate and returns a value of type t.

With this design, the Agda compiler can give a natural literal the type

```
{t : Set} ⦃hnl : HasNatLiterals t⦄
  {_ : True (HasNatLiterals.decideValid hnl n)} → t
```

The literal n would then be equivalent to $\lambda\ \{t\}\ ⦃\ \mathit{hnl}\ ⦄\ \{\mathit{prf}\}\ \rightarrow\ \mathit{from}\mathbb{N}\ \{t\}\ ⦃\ \mathit{hnl}\ ⦄\ n\ \{\mathit{prf}\}$. This would require users to define or bring into scope a value of type $\mathit{HasNatLiterals}\ t$ for any type t for which they use natural literals. We give two examples of such values for types ℕ and $\mathit{Fin}\ n$. The instance for *Fin* uses the previously defined function #_.

```
wnlℕ : HasNatLiterals ℕ
wnlℕ = record { ValidLiteral = λ _ → ⊤;
                decideValid  = λ _ → yes tt;
                fromℕ        = λ x → x }
wnlFin : {k : ℕ} → HasNatLiterals (Fin k)
wnlFin {k} = record { ValidLiteral = λ n → n < k;
                      decideValid  = λ n → n <? k;
                      fromℕ        = λ n {prf} → #_ n {k} {prf} }
```

We believe that this design for natural literals can be generalised to other kinds of literals (floating point and strings). Note that Haskell strings are not normally

overloaded but there is a GHC extension that changes this. It is interesting to point out that for strings the technique allows statically parsed and checked string literals, something which might be a fruitful avenue for the modelling of deep DSLs within Agda. This could form the basis for a nice alternative to GHC's quasi-quotes [136].

We have implemented the above design of overloaded natural literals for Agda but it has not been adopted in the released version of Agda. In a discussion on the Agda mailing list [57], it was noted that *HasNatLiterals* could be used without compiler modifications by writing $\mathit{fromℕ}\ \mathit{3}$ instead of $\mathit{3}$ and there was doubt about whether the additional advantage of being able to write $\mathit{3}$ instead was important enough to warrant a compiler modification.

In Haskell, numeric literals are also overloaded during pattern matching, and we think our design can be adapted to support that as well.

2.5.7 Two final examples

As a small encore in this section, we can't resist discussing two code snippets using instance arguments. The first is an example of a function abstracting over functions with implicit arguments. It demonstrates the first-class nature of our new type of arguments: functions with instance arguments can be abstracted over, their types can be written out etc.

$$\begin{aligned}&\mathit{explicitise} : \forall\ \{A : \mathit{Set}\}\ \{B : A \to \mathit{Set}\} \to \\ &\qquad\qquad (\{\!\{x : A\}\!\} \to B\ x) \to (x : A) \to B\ x \\ &\mathit{explicitise}\ f\ x\ =\ f\ \{\!\{x\}\!\}\end{aligned}$$

Our final example is small, but very useful: it is an analogue of Agda's standard underscore construct for instance arguments, similar to Scala's *implicitly* or ?. Like in Scala, we don't need to introduce special syntax for this: the following definition suffices. This ellipsis can be used as a shorthand in any situation where only a single type-correct value is in scope. Because our resolution algorithm does not require candidates to be specially annotated to be eligible, our ellipsis is more generally useful than Scala's *implicitly*.

$$\begin{aligned}&\cdots : \{A : \mathit{Set}\} \to \{\!\{a : A\}\!\} \to A \\ &\cdots\ \{\!\{a\}\!\}\ =\ a\end{aligned}$$

2.6 Related work

There exists a lot of literature about type classes, extensions of them and alternatives to them [235, 112, 204, 174, 111, 237, 65, 181, 167, 63, 213, 90, 223, 61, 211, 212, 168]. We have already discussed Haskell type classes and Scala implicits in Section 2.1 and later and we do not repeat that information here. We also do not further discuss Lewis et al.'s implicit parameters, implemented in Hugs and GHC [127], as they use a name-based resolution, instead of the type-based resolution of our design and are thus not suited for our use cases.

2.6.1 Scala implicits

We have already discussed Scala implicits in Section 2.1.4, so we only provide some more details here.

As mentioned before, functions that take implicit arguments are restricted in Scala. To be more specific, Scala implicits have the following restrictions. In the first place, a function can accept several implicit arguments, but they are required to occur after the conventional arguments. Abstracting over functions taking implicit arguments is not syntactically possible but requires encoding such functions as objects with an *apply* method taking an implicit argument. There is no user syntax for the type of a function accepting implicit arguments. Anonymous functions cannot accept implicit arguments.[4] Full and tight control on the insertion of implicit arguments does not seem to be available and it seems impossible to partially apply a given function with implicit arguments to any chosen subset of its (implicit and ordinary) arguments (while keeping the implicit arguments implicit).

More details about Scala's resolution algorithm, like the termination criterion for implicit resolution or the precise set of candidates for resolving a certain implicit argument can be found in the Scala specification [164].

2.6.2 Coq type classes

Sozeau and Oury have recently presented Coq type classes [213]. Like Agda, Coq is a dependently typed, purely functional programming language/proof assistant. Compared to Agda, Coq has a longer history, a larger user base, and

[4] The Scala syntax (*implicit* $x \Rightarrow x$) defines an anonymous function that takes a *normal* argument x, but makes x eligible for implicit resolution in the function body [164, §6.23].

a principled core language and associated type-checker.[5] On top of that, Coq offers a variety of language features and meta-programming/proof automation facilities.

Sozeau and Oury's type classes are a new form of dependently typed record types. If a function has an implicit argument of such a record type, and its value cannot be inferred through Coq's standard unification, then Coq will try to infer a value through an instance search. This instance search is implemented as an automated proof search using a special-purpose port of the `eauto` tactic. This tactic performs a bounded breadth- or depth-first search using the type class's instances as lemmas. Both direct instances (objects of the record type) and parameterised instances (functions which take certain arguments and return such an object) are supported.

Sozeau and Oury further discuss syntax extensions and models of superclasses and substructures and then provide a discussion of various aspects of their system, most importantly their instance search tactic. They think their current instance search tactic is not sufficient in the context of multi-parameter type classes and arbitrary instances (which their system currently allows). They state the algorithm's non-determinism and unpredictability as problems which they hope to address in the future by restricting the shape of allowed instances and using a more predictable algorithm.

Sozeau and Oury's instance search is currently at least as powerful as Haskell's or Scala's instance/implicit search: it can be used as a separate computational model (see Section 2.4). Sozeau and Oury's mechanism is limited to *record types* that were *defined as a type class*, so existing libraries need to be adapted to benefit from it. Type class instances can be defined locally, but it seems that the local instance will not be considered for automatic resolution. In comparison, instance arguments provide a weaker computational model, can be used with unmodified existing libraries and provides an equivalent of local instances, as we discussed previously.

2.6.3 Coq canonical structures

Coq features another type system concept which can be exploited as an alternative to type classes: *canonical structures* [13, 223]. This feature allows certain values of a record type to be marked as *canonical structures*. These are then automatically considered when the Coq type inferencer tries to infer a value of the record type from the value of one of its fields. Canonical structures

[5] Contrary to Coq, Agda does not at the moment have a clearly demarcated and formalised core language.

have existed for some time in Coq, but have recently attracted the attention of authors looking to provide easy to use libraries of complex concepts, exploiting canonical structures as a powerful meta-programming feature that implicitly resolves values in the background. There are some similarities in the design to ours, as it does not introduce a separate type of structure and does a form of implicit resolution from call-site scope.

The precise workings of the feature are tightly coupled to the inner workings of Coq's type inferencer. A precise formal description does not appear to be available but Gonthier et al. have reverse-engineered an operational semantics for the mechanism in Coq 8.3 [85]. It is not clear if any termination guarantees are offered/required regarding canonical structure resolution.

Coq's instance resolution presents a powerful computational model. The best illustration of this fact is presented by Gonthier et al. [86] who present an assortment of *design patterns* that allows one to exploit the model in a logic programming style, with support for recursion, backtracking and syntactic pattern matching on types. This makes the resolution very powerful, and the authors demonstrate how it enables a form of proof automation ("overloaded lemmas") that is a compelling alternative to Coq's tactics.

The canonical structures resolution mechanism and Gonthier et al.'s patterns for it are powerful, but they seem to require expertise and understanding of Coq's type inferencer to use. The computational model is Prolog-like: unification-based with support for backtracking, not functional like Coq's underlying logic.

2.6.4 Isabelle type classes

The Isabelle/HOL proof assistant also features a form of type classes which have been described by Haftmann and Wenzel [90]. Haftmann and Wenzel explain these type classes in terms of existing Isabelle concepts like locales and top-level polymorphic functions, together with a form of extra-logical constraints checking, thus avoiding an extension of the logic. Isabelle type classes are restricted to a single parameter and this parameter must be a type (not a type constructor, not a value). The resolution mechanism is comparable in computational power to Haskell's, so more powerful than that of instance arguments.

2.6.5 Modular type classes

Dreyer et al. discuss an alternative to type classes in the context of ML [63]. They share our view that Haskell type classes duplicate functionality by introducing a separate structuring concept, and they argue that ML modules

already provide functionality akin to associated type families and type class inheritance (like we do for ADTs). They propose to model single-parameter type classes as *class signatures*: module signatures with a single abstract type named t. Instances become modules and functors. Their primitive **overload** *fun* **from** *sig* returns a version of function *fun* from class signature *sig* that will resolve the appropriate module implementing *sig* from call-site scope. Another primitive *canon* (*sig*) resolves and returns the module as a whole.

Resolution of such a module takes into account modules and functors that have been annotated in the current scope with a **using** declaration. Since functors are considered, the instance search is recursive. They do not support the equivalent of multi-parameter type classes. It is not clear to us if and how their type class modules can be abstracted over.

2.6.6 Explicit Haskell

In an unpublished technical report, Dijkstra and Swierstra describe an implicit arguments system which they have implemented in a Haskell variant called *Explicit Haskell* [61]. Their motivation is that Haskell does not provide a way to override the automatic resolution of instances (dictionaries) for functions with a type class constraint. They extend Haskell with named instances, local instances, and a way to explicitly provide an instance to a function with a type class constraint, either by naming the instance or by lifting a value of a record type corresponding to the type class. They also allow type class constraints to appear anywhere in a type, not just at the beginning. For resolving type class constraints, they use a resolution close to Haskell's (their design is a conservative extension of Haskell). The only difference is that instances can be annotated to not take part in this resolution (in which case they can only be used by name). In the same text, Dijkstra and Swierstra discuss a system for partial type signatures, with independent value. The system allows the user to partially specify types for values and leave the rest to be inferred.

This design has many similarities to our system. Their extensions to the concept of type class constraints effectively transform them into a special form of function arguments similar to our instance arguments. Their design offers some of the same benefits as ours (e.g. local instances, named instances), and they discuss some of the same examples as we do in Section 2.5. However, they make some different choices: their constraints remain limited to arguments whose type was defined as a type class instance and their resolution is similar to Haskell's. They do not fully unify type classes with their associated record types, so that some of the advantages we can offer are not available (e.g. abstracting over a type class).

2.6.7 Concepts in C++

C++ features a form of polymorphic functions known as template functions. For a long time, templates were only type-checked after they were instantiated with concrete (type and/or compile-time value) arguments. When the community started looking for a type system that could type-check templates before instantiation, they set out to add language support for the semi-formal *concepts* that were used in the documentation of the Standard Template Library. Over the years, a variety of proposals was defined but due to an ongoing disagreement about elements of the design, concepts have not been included in the latest version of the C++ standard [211]. We do not try to discuss all proposals and their differences here, let alone the details of the disagreements, but we only discuss one of the most principled designs that originates from this effort: Siek and Lumsdaine's $\mathcal{G}$ programming language [212].

The $\mathcal{G}$ language is a variation of C++, with concepts as the most significant addition. Concepts are closely related to type classes and $\mathcal{G}$ supports a wide variety of features including analogues of type families, multi-parameter type classes, equality constraints and local instances as well as some non-standard features like *concept-based overloading*; the equivalent of allowing parametric instances to overlap if they have different constraints and resolving them by choosing the most specific applicable instance.

In $\mathcal{G}$, it is not possible to abstract over constraints as described in Section 2.1.2. Siek and Lumsdaine chose not to restrict the shape of parameterised instances, so that termination of constraint resolution is not guaranteed and the computational model is comparable to that of Haskell with the `UndecidableInstances` extension [210].

2.6.8 The implicit calculus

Oliveira et al. present the implicit calculus, intended as a minimal and general core calculus for studying and informing implementations of type-class like mechanisms in other languages [168]. The authors and the calculus focus on the two questions of instance (or in their terms *rule*) scope and the resolution mechanism. Our system does not seem fully supported by their calculus. For example, our system (but also e.g. Scala) allows values in the implicit context to be referred to by name, while in their calculus it is only possible to refer to them by type. Regarding resolution, they support higher-order rules and partial resolution of class constraints. They present a theorem that connects resolution explicitly to a form of Prolog-style logic programming, a connection that we have strived to avoid. An extended version of the paper discusses a

possible restriction on instances that ensures termination of the search [171], which would lead to a computational power similar to that of Haskell's type class resolution.

2.6.9 Agda Prelude

Instance arguments have been used in Kettelhoit's master thesis to construct a Prelude for Agda [116]. His aims were to solve the problem of name clashes in Agda's standard library, to construct an analogue of Haskell's Prelude for Agda and to try everything out on a number of IO-based programs. He uses instance arguments as an essential tool to solve name clashes. In the conclusions, he disagrees with our choice for a non-recursive instance search, because they are needed to automatically derive *Show* instances and expresses the hope that this will be changed in Agda in the future.

2.7 Acknowledgements

This research is partially funded by the Research Foundation - Flanders (FWO), and by the Research Fund KU Leuven. Dominique Devriese holds a PhD fellowship of the Research Foundation — Flanders (FWO). The authors thank Adriaan Moors (EPFL) for providing valuable feedback on a draft of this text, as well as clarifying various points about Scala implicits. We thank our editors Kenichi Asai and Olivier Danvy and anonymous reviewers for many valuable comments.

2.A Under the hood

In this appendix, we discuss the precise changes we make in more detail. The definitions in this section are extensions and adaptations of Norell's rules for Agda's type system and standard implicit arguments [161, 3.5 pp. 69–70, 5.1.5 pp. 99–100]. They should be read in the context of Norell's developments and may not be fully clear without them.

2.A.1 Implicit lambdas

We add another function space $\{\!\{ x : A \}\!\} \to B$ in addition to the existing $\{ x : A \} \to B$ and $(x : A) \to B$. Like the existing implicit functions, the new

functions are semantically equivalent to the corresponding ordinary functions. Values of type $\{\!\{x : A\}\!\} \to B$ can be introduced as (typed or untyped) lambda values $\lambda\,\{\!\{x\}\!\} \to e$ or $\lambda\,\{\!\{x : A\}\!\} \to e$ or they can be defined as constants (at the top-level or in where-clauses).

For type-checking values of this type, we extend the rules for Agda's standard implicit arguments [161, 3.5 pp. 69–70] as follows. If a value does not explicitly mention an instance argument from the type it is checked against, rule (2.2) infers implicit lambdas, like for normal implicit arguments.

$$\frac{\Gamma, x : A \vdash e \uparrow B \rightsquigarrow s}{\Gamma \vdash \lambda\{\!\{x\}\!\}.e \uparrow \{\!\{x : A\}\!\} \to B \rightsquigarrow \lambda\{\!\{x\}\!\}.s} \tag{2.1}$$

$$\frac{\Gamma, x : A \vdash e \uparrow B \rightsquigarrow s \qquad e \neq \lambda\{\!\{x\}\!\}.e'}{\Gamma \vdash e \uparrow \{\!\{x : A\}\!\} \to B \rightsquigarrow \lambda\{\!\{x\}\!\}.s} \tag{2.2}$$

2.A.2 Instance arguments

Next, we need to determine when instance arguments of a function are not provided explicitly and should be inferred. This mechanism is governed by the inference rules for argument checking judgements of the form $\Gamma \vdash A \mathbin{@} \bar{e} \downarrow B \rightsquigarrow \bar{s}$. Such a judgment means that the values $\bar{e}$ can be passed as arguments to a value of type A, producing a value of type B. The full list of arguments to be applied to the function (including implicitly inserted ones) is "returned" in $\bar{s}$.

We extend the corresponding rules for implicit arguments [161, 3.5 p. 70] as follows. For a non-provided instance argument, we do not just insert a meta-variable α, but we additionally add a constraint $\mathrm{FindInScope}\,\alpha$. This is a special kind of constraint that indicates that α should be resolved as an instance argument. To do this, we need to extend the form of argument checking judgements to additionally return a set of constraints $\mathcal{C}$: $\Gamma \vdash A \mathbin{@} \bar{e} \downarrow B \rightsquigarrow \bar{s}, \mathcal{C}$. This adapted form actually corresponds more closely to Agda's existing implementation of the rules. Existing rules in the old form of the judgment should now be read as producing no constraints or passing through generated constraints if they recurse.

$$\frac{\Gamma \vdash e \uparrow A \rightsquigarrow s \qquad \Gamma \vdash B[x := s] \mathbin{@} \bar{e} \downarrow B' \rightsquigarrow \bar{s}, \mathcal{C}}{\Gamma \vdash \{\!\{x : A\}\!\} \to B \mathbin{@} \{\!\{e\}\!\}; \bar{e} \downarrow B' \rightsquigarrow s; \bar{s}, \mathcal{C}} \tag{2.3}$$

$$\frac{\begin{array}{c}\mathrm{AddMeta}(\alpha : \Gamma \to A) \qquad \bar{e} \neq \{\!\{e\}\!\}; \bar{e}' \\ \Gamma \vdash \{\!\{x : A\}\!\} \to B \mathbin{@} \{\!\{\alpha\}\!\}; \bar{e} \downarrow B' \rightsquigarrow \bar{s}, \mathcal{C}\end{array}}{\Gamma \vdash \{\!\{x : A\}\!\} \to B \mathbin{@} \bar{e} \downarrow B' \rightsquigarrow \bar{s}, \mathcal{C} \cup \{\mathrm{FindInScope}\,\alpha\}} \tag{2.4}$$

We change the last rule on page 70 of [161,] to the following:

$$\frac{A \neq \{\!\{x : A_1\}\!\} \to A_2 \qquad A \neq \{x : A_1\} \to A_2}{\Gamma \vdash A \mathbin{@} \epsilon \downarrow A \rightsquigarrow \epsilon, \{\}} \tag{2.5}$$

A somewhat technical point here is that at the moment, we do not allow meta-variables introduced for instance arguments to be η-expanded, as this is done for Agda's normal implicit arguments. We take a conservative approach because we currently do not have a good understanding of possible interactions between η-expansion and instance resolution. During our experiments, we have established that all of them (see Section 2.3 and 2.5) have worked well without η-expansion. It is future work to get a better understanding of the issues involved.

2.A.3 Resolution algorithm

The resolution of a constraint $\mathrm{FindInScope}\,\alpha$ in context Γ and scope S with $\Gamma \vdash \alpha : A$ tries to infer a value from either the values in the current context Γ or the constants in scope S. If only one candidate is found in both sets, it is selected. If there is more than one candidate, resolution of the constraint is postponed in the hope that more type information will become available further on, reducing the set of candidates further. If the constraint is not resolved when type checking finishes, this is reported to the user. If there are no candidates, then the constraint cannot be solved and this is also reported.

To formalise these rules, we need some extra information about meta variables introduced through inference rule (2.4) above: the context and scopes at the point where they were defined. We do not make this change explicit because the context is actually already implicitly being maintained throughout Norell's development [161], and because both the scope and the context are already being kept in the Agda implementation. For a meta variable α, we write $\mathrm{MScp}\,\alpha$ and $\mathrm{MCtx}\,\alpha$ for the scope resp. the context in which a meta variable α was introduced.

With these nuances, we can formally define how we solve constraints $\mathrm{FindInScope}\,\alpha$ as follows:

$$\frac{\begin{array}{c}\mathrm{Lookup}(\alpha : A) \\ \mathrm{Candidates}(\mathrm{MCtx}(\alpha), \mathrm{MScp}(\alpha), \alpha, A) = \{(n, A_n)\} \\ \Gamma \vdash A \simeq A_n \rightsquigarrow \mathcal{C} \qquad\qquad \alpha := n\end{array}}{\Gamma \vdash \mathrm{FindInScope}\,\alpha \rightsquigarrow \mathcal{C}} \tag{2.6}$$

This definition says that if we have a meta-variable α typed A, that is to be inferred as an instance argument, then it is resolved if there is a unique solution.

In this case, we require convertibility of the types and assign the value to the meta-variable. The set of candidates in context Γ and scope S, for meta-variable α, of type A is defined by predicate Candidates:

$$\text{Candidates}(\Gamma, S, \alpha, A) = \{(n', A_n) \mid \text{Cand}(\Gamma, S, n, A_n) \text{ and } \text{ValidCand}(\alpha, A, n, n', A_n)\} \quad (2.7)$$

For terms n, of type A_n, that are potential candidates in the current context and scope (predicate Cand below), the candidates are the terms n' that result from validity checking n with respect to the current meta-variable and its type. This last property is defined by the ValidCand predicate.

$$\frac{\langle\Sigma\rangle\, \text{CheckCand}(\alpha, A, n, n', A') \rightsquigarrow \mathcal{C} \Longrightarrow \langle\Sigma'\rangle}{\langle\Sigma\rangle\, \text{ValidCand}(\alpha, A, n, n', A') \Longrightarrow \langle\Sigma\rangle} \quad (2.8)$$

In the definition of this predicate, we perform a check, but if this check makes changes to the current signature, we do not yet carry them through here. This is formalised using the explicit notation of the signatures in the judgements [161, 3.3.1 p. 54].

The check that a certain term is valid for a certain meta-variable consists of two parts:

$$\frac{\begin{array}{c}\Gamma \vdash A' \;@_{\text{NI}}\; \varepsilon \downarrow A \rightsquigarrow \bar{s}, \mathcal{C} \qquad \alpha := n\ \bar{s} \qquad \text{CurrentConstraints}(\mathcal{C}) \\ \forall C \in \mathcal{C} : C \neq \text{FindInScope}\, \alpha' \Rightarrow \neg(\nvdash C \rightsquigarrow \mathcal{C}')\end{array}}{\text{CheckCand}(\alpha, A, n', A') \rightsquigarrow \mathcal{C}} \quad (2.9)$$

First, we use the judgment $\Gamma \vdash A' \;@_{\text{NI}}\; \varepsilon \downarrow A \rightsquigarrow \bar{s}, \mathcal{C}$ to require that the type A' of n is convertible to type A after potentially applying a number of implicit (but not instance) arguments. The rules for judgment $\Gamma \vdash A \;@_{\text{NI}}\; \bar{e} \downarrow B \rightsquigarrow \bar{s}, \mathcal{C}$ are identical to the ones for judgment $\Gamma \vdash A \;@\; \bar{e} \downarrow B \rightsquigarrow \bar{s}, \mathcal{C}$ except that our additional rule (2.4) is not allowed and rule (2.5) is changed back to its original version.

Second, if we assign the resulting value to the meta-variable, no other constraint must be immediately invalidated. For this last check, we do not recursively consider other FindInScope constraints, since this would introduce recursion in the instance search. This check is necessarily incomplete: in Norell's words, the type checker will give one of three answers [161, Note p. 65]: "yes it is type correct", "no it is not correct" or "it might be correct if the meta-variables are instantiated properly". Only if we get the second answer, we reject the candidate under scrutiny.

Rule (2.9) strikes a fine balance. On the one hand, the resolution algorithm needs to be powerful enough to be usable, but we avoid making it too powerful

(see discussion in Section 2.4). The intuition behind the criterion above is that we consider any value that is type-correct in the sense that it has the correct type, but also in the sense that it does not immediately invalidate constraints. The criterion has proven sufficient for all use cases discussed in this text, *but also necessary: without the check for invalidated constraints, monad instances, for example, are often not resolved.* Note that we have used a new CurrentConstraints operation, which works on the signature that is implicit in the typing judgements:

$$\langle \Sigma \rangle \;\; \text{CurrentConstraints}(\mathcal{C}) \Longrightarrow \langle \Sigma \rangle \text{ where } \mathcal{C} = \{C \mid C \in \Sigma\} \tag{2.10}$$

We still need to define the potential candidates in a given context and scope. The Cand property formalises this:

$$\frac{\Gamma = \Gamma_1; n : A; \Gamma_2}{\text{Cand}(\Gamma, S, n, A)} \tag{2.11}$$

$$\frac{\text{Visible}_{\text{pri}}(n, S) \qquad \text{Lookup}(n : A)}{\text{Cand}(\Gamma, S, n, A)} \tag{2.12}$$

The somewhat technical predicate $\text{Visible}_p(n, S)$ asserts that name n is publicly or privately (defined by p) in scope stack S. The name must either be available at visibility p or *pub* in the scope at the top of the stack S or publicly visible in one of the scopes below the top. The predicate $\text{VisibleNS}_\alpha(n, \sigma)$ asserts that name n is available in the α-visible namespace (ns_α) of scope σ. Its definition requires that n is in the domain of the first part of that namespace, i.e. the part containing values (the second part contains modules).

$$\frac{\text{VisibleNS}_\alpha(n, \sigma) \vee \text{VisibleNS}_{\text{pub}}(n, \sigma)}{\text{Visible}_\alpha(n, S \blacktriangleright \sigma)} \tag{2.13}$$

$$\frac{\text{Visible}_{\text{pub}}(n, S)}{\text{Visible}_\alpha(n, S \blacktriangleright \sigma)} \tag{2.14}$$

$$\frac{n \in \text{dom}(\text{fst}(ns_\alpha))}{\text{VisibleNS}_\alpha(n, \langle M, ns_{\text{pub}}, ns_{\text{pri}} \rangle)} \tag{2.15}$$

From rules (2.6) and (2.9) above, it is clear that resolution of constraints FindInScope α only compares types that have already been type checked, and does not trigger extra type checking. Therefore, only one constraint FindInScope α will be resolved per occurrence in the user's code of a function taking an instance argument without a value being provided explicitly. This means that, contrary to other proposals, the computational power of *our resolution algorithm is fundamentally limited, in the sense that it does not allow any form of recursion, looping or backtracking. It therefore does not introduce a separate computational model in the language* (see Section 2.4).

2.A.4 Soundness

Intuitively, soundness of the rules above is easily guaranteed, because all we do is assign terms of the correct type to meta-variables. The following lemma reflects this intuition, supplementing Norell's Lemma 3.5.13:

Lemma 2.A.1 (Instance resolution preserves consistency)**.** *If* $\Gamma \vdash_{|\Sigma|}$ **valid**, Σ *is consistent and*

$$\langle \Sigma \rangle\ \Gamma \vdash \mathrm{FindInScope}\, \alpha \Longrightarrow \langle \Sigma' \rangle$$

then Σ' *is consistent.*

Proof. A consequence of Norell's Lemma 3.5.12 (Refinement preserves consistent signatures), together with the observation that rule (2.6) will only ever perform a type correct assignment of a meta-variable (a signature refinement). □

This lemma suffices to establish that Norell's Lemma 3.5.14 (Constraint solving is sound) stays valid in the context of our new kind of constraints, as well as the main result, Theorem 3.5.18 (Soundness of type checking).

Like Norell for normal implicit arguments, we provide formal rules for the insertion of instance arguments and the insertion of instance lambdas, but do not prove any formal results about them.

Some of the rules above may give the impression that this resolution algorithm is sensitive to the order in which type-checking is interleaved with constraint solving. However, this sensitivity actually only exists for error reporting. Remember that during type-checking, constraints will only be added and solved (after a correct meta-variable assignment), but they cannot otherwise be removed. As a consequence of this, the candidates set for a given instance argument meta-variable α, defined by rule (2.7) above, form a descending series during type-checking: later sets are subsets of previous ones. Furthermore, if a value in scope can be assigned to α such that the entire Agda expression successfully type checks, then this value will be contained in all of these candidate sets. All non-valid candidates will eventually be removed. Therefore, if no other valid candidates are available, the valid value will inevitably be chosen.

Note that this notion of candidates set is also useful for efficiently implementing instance argument resolution. In fact, the current Agda implementation maintains this set explicitly during type checking such that instance argument candidates that have already been discarded are not reconsidered afterwards.

For erroneous programs, the order of constraint solving may determine the kind of error message that is generated. Depending on whether a constraint

C is registered after a certain instance argument is already resolved, or before, an error will be reported for the FindInScope constraint or the constraint C. This influence of type-checking on error reporting also exists for standard Hindley-Milner type inferencing [93], so we consider it acceptable.

One possible extension of the current resolution scheme that we have considered in detail is based on a prioritisation of candidates, e.g. by giving precedence to values defined closer to the call site. However, contrary to our current resolution algorithm, such a prioritisation does make the result of instance resolution depend on the order of constraint resolution. Suppose there is a value in the highest priority set which is valid except for a constraint produced late during type checking and suppose this is the only candidate at the highest priority, but a lower priority candidate is also valid, and does not invalidate the late constraint. Since we don't know upfront which constraints will be produced during the rest of type-checking, we have to decide at some point which value to use. If the late constraint has then not yet been produced, the highest priority candidate will be selected and a type error will be reported when the late constraint is finally encountered. However, if the resolution occurs after the production of the late constraint, the valid low-priority candidate is chosen instead of the invalid high-priority one, and all goes well.

We currently do not see a solution for this problem, so we keep the introduction of a prioritised resolution algorithm as future work. Our experiments (see Section 2.3 and 2.5) show that the current non-prioritised resolution scheme suffices for real use.

Chapter 3

Finally Tagless Observable Recursion for an Abstract Grammar Model

Publication data

Dominique Devriese and Frank Piessens. Finally tagless observable recursion for an abstract grammar model. *Journal of Functional Programming*, 22(6):757–796, November 2012.

Dominique Devriese and Frank Piessens. Explicitly recursive grammar combinators — the implementation of some grammar algorithms - technical report. CW Report CW594, Department of Computer Science, K.U.Leuven, September 2010.

Dominique Devriese and Frank Piessens. Explicitly recursive grammar combinators — a better model for shallow parser DSLs. In *Practical Aspects of Declarative Languages (PADL)*, volume 6539 of *Lecture Notes in Computer Science*, pages 84–98. Springer, January 2011.

Abstract

We define shallow embedding of a typed grammar language in Carette et al.'s finally tagless style. In order to avoid the limitations of traditional parser combinator libraries (no bottom-up parsing, no full grammar analysis or transformation), we need object-language recursion to be observable in the meta-language. Since existing proposals for recursive constructs are not fully satisfactory, we propose new finally tagless primitive recursive constructs to solve the problem. To do this in a well-typed way, we require considerable infrastructure, for which we reuse techniques from the multirec generic programming library. Our infrastructure allows a precise model of the complex interaction between a grammar, a parsing algorithm and a set of semantic actions. On the flip side, our approach requires the grammar author to provide a type- and value-level encoding of the grammar's domain and we can provide only a limited form of constructs like *many*.

We demonstrate five meta-language grammar algorithms exploiting our model, including a grammar pretty-printer, a reachability analysis, a translation of quantified recursive constructs to the standard one, and an implementation of the left-corner grammar transform. The work we present forms the basis of the `grammar-combinators` parsing library,[1] which is the first to work with a precise, shallow model of abstract context-free grammars in a classical (not dependently-typed) functional language and which supports a wide range of grammar manipulation primitives.

From a more general point of view, our work shows a solution to the well-studied problem of observable sharing in shallowly embedded domain specific languages and specifically in finally tagless DSLs.

[1] `http://projects.haskell.org/grammar-combinators`

3.1 Introduction

Parser combinator libraries are a prime example of using functional languages to embed a Domain-Specific Language (DSL) in a shallow way, i.e. reusing many facilities from the host language. Nevertheless, despite their advantages, current mainstream purely functional parser combinator libraries are not satisfactory from a parsing theory point of view. While many other parsing tools employ more powerful bottom-up parsing algorithms, parser combinators are naturally restricted to top-down algorithms. Unlike other tools, they do not employ much grammar analysis or precalculate tables; grammar authors are not provided with standard implementations of well-known grammar analysis, transformation or visualization techniques.

In this paper, we work towards the aim of functional parsing libraries which combine the advantages of parser combinators with the power that is standard in parser generators and associated tooling. In this paper, we focus on a major problem that needs to be solved for this to happen: removing the limitations of the grammar model currently used by parser combinators.

It turns out that many of the limitations of purely functional parser combinator libraries are caused by the direct encoding of recursion in the grammar object language using meta-language recursion. We show that we can do better with a new encoding of recursive constructs. Our constructs are parametric in the interpretation of the grammar's recursion.

Our object-language recursive constructs use the finally tagless style as described by Carette et al. [30].[2] This style of modelling a DSL allows grammar algorithms to interpret a grammar's production rules in the way they need to and to distinguish regular, context-free and extended context-free grammars in a natural way. We use an alternative to the *fix* construct of Carette et al., that seems better suited for the parsing domain. The main technical challenge we face is ensuring that our constructs remain well-typed, for which we employ techniques from the multirec generic programming library [197]. We show that our infrastructure allows for a precise and modular modelling of the complex interaction between grammar, parsing algorithm and semantic actions.

A limitation of our representation of recursion is that it crucially depends on the grammar author providing a type- and value-level encoding of the grammar's domain. Additionally, the encoding of some constructs becomes more difficult: we will discuss how we need to restrict the standard *many* operator (corresponding to Kleene-*) to non-terminal references to allow algorithms to

[2]The word *finally* refers to the *final style* of the approach, in which a DSL is quantified over all possible interpretations of its primitives. This contrasts to other, initial encodings.

interpret the constructs in the way they need. We have no detailed measurements of parsing performance for our grammars, but with current compilers, our additional indirection and our use of generic programming techniques introduces significant performance costs.

We show that our approach does bring important additional expressivity by demonstrating five well-known grammar algorithms from the parsing literature, including a grammar pretty-printer, a reachability analysis, a translation of quantified recursive constructs to the standard one, and an implementation of the left-corner grammar transform. In our `grammar-combinators` library, we provide implementations of a range of other algorithms. These include an implementation of the packrat parsing algorithm [73] and a grammar transformation which induces a bottom-up matching order on the original grammar using a top-down parsing of the transformed grammar.

More generally, the problem of observing recursion and sharing in Embedded DSL (EDSL) terms has triggered a lot of research. Our approach presents a novel solution applied to a parsing DSL. Our solution does not compromise referential transparency or unnecessarily force the user to resort to models of code with side-effects. We keep the validation of our approach in other domains (like the typical example of hardware description DSLs) as future work. Our design also extends the set of known programming patterns for finally tagless models of DSLs.

3.1.1 Contributions

In this chapter, we make the following contributions:

- Evidence that observably recursive constructs are needed for full power, purely functional parsing DSLs.
- Primitive recursive constructs enabling a shallow embedding of a purely functional grammar DSL.
- A proper typing for our constructs using techniques from the multirec generic programming library.
- A deforestation, enabled by our infrastructure, that precisely models the complex interaction between grammar, parsing algorithm and semantic actions, independent from the matching order.
- Five grammar algorithms, including the left-corner grammar transformation, showing that our encoding provides significant and important additional expressivity over traditional parser combinator libraries.

An earlier account of some of the results in this chapter was presented at the 2011 PADL conference [59]. This chapter is based on a different presentation of the material that appeared in the Journal of Functional Programming [60], which highlights the relation with finally tagless encodings, uses a more standard notation and type classes for applicative functor operations and presents a rationale for our techniques. The content of Section 3.5 was not presented at PADL. It is partly new and partly appeared in a technical report [58].

3.1.2 Outlook

In Section 3.2, we introduce an example grammar, and a standard encoding of abstract parser combinators in a finally tagless style. We take a brief look at the problem of left-recursion, and then explain the problem with the standard modelling of object-language recursion using direct meta-language recursion.

We introduce new recursive constructs in Section 3.3 and we show how they can be properly typed in Section 3.4. We present this through gradual (initially untyped) transformations of the first definition of our example grammar. This allows us to tackle technical problems one at a time and to show the rationale of our encoding.

In Section 3.5, we demonstrate the increased power of our grammar model with the definition of five grammar algorithms:

- In Section 3.5.1, we show that the recursive structure can be observed in our model by implementing a *grammar pretty-printer*.
- In Section 3.5.2, we show the equivalence between our final recursive constructs and an alternative encountered in Section 3.3. The resulting algorithm is also a useful technical aid for what follows.
- We show that our recursive constructs permit complex analyses, by implementing a *reachability analysis* in Section 3.5.3.
- In order to prove that our grammar model supports simple grammar transformations introducing new non-terminals, and provides tight control of the constructs allowed in a grammar, we define a *translation of quantified recursive constructs into the standard one* in Section 3.5.4.
- Finally, in Section 3.5.5, we show that our technique supports complex, realistic grammar transformations with the implementation of the standard *left-corner grammar transform*.

We discuss related work in Section 3.6.

Much of the Haskell code in this text relies on a set of Haskell extensions that is currently only supported by the GHC Haskell compiler.[3] However, these are all well-accepted extensions that do not make type-checking undecidable. Our library optionally supports the use of Template Haskell [208] for performing grammar transformations at compile-time.

3.2 Finally tagless parser combinators

In this section, we will present an example grammar and encode it in a finally tagless parser combinator library. Using this example, we will discuss in detail why the standard representation of recursion is unsatisfactory.

3.2.1 Arithmetic expressions

We start our presentation with a standard example from the parser literature: a simple grammar describing arithmetic expressions of the form "$(6 * (4 + 2)) + 6$", in a formalism similar to (E)BNF (see [3, Section 2.2]).

Line	$\rightarrow$	Expr EOF
Expr	$\rightarrow$	Expr '+' Term
	$\rightarrow$	Term
Term	$\rightarrow$	Term '$*$' Factor
	$\rightarrow$	Factor
Factor	$\rightarrow$	'(' Expr ')'
	$\rightarrow$	Digit+
Digit	$\rightarrow$	'0' | '1' | '2' | ... | '8' | '9'

The definitions of Expr and Term are such that "$a+b*c$" can only be interpreted as "$a + (b * c)$" and "$a + b + c$" only as "$(a + b) + c$". This modelling of operator precedence and left-associativity is idiomatic for LR-style grammars, but fundamentally relies on left-recursion: one of the productions of non-terminal Expr, for example, refers back to Expr in the first position.

In order to obtain a parser for this grammar (without manually writing it ourselves), parser generators like Yacc [109] and ANTLR [176] are typically used to translate the grammar (provided in an EBNF-like formalism) into source code in the developer's programming language. This technique has proven successful in practice, but suffers from various downsides: little assurance for syntax- and type-correctness of generated code, little reuse of the developer's existing

[3] TypeFamilies, GADTs, MultiParamTypeClasses, FunctionalDependencies, FlexibleContexts and RankNTypes.

programming environment (editor, type-checker, build system etc.), limited support for abstraction, no support for grammars constructed at runtime etc.

3.2.2 Finally tagless parser combinators

Parser combinator libraries provide an elegant alternative, modelling the grammar directly in a general-purpose programming language. These libraries treat parsers as first-class values that can be combined, extended, reused, abstracted from etc. Many of these libraries only provide a single parsing algorithm, even though there is no immediate technical reason for this. In this section, we define a more abstract grammar model, without this coupling. This abstraction also saves us the trouble of actually explaining a concrete parsing algorithm; our abstract model can be used with many of the well-known parser combinator libraries (e.g. uu-parsinglib [219] or Parsec [125]).

The technique we use to achieve this decoupling was already used by Swierstra and Duponcheel [220] and has been described and popularised by Carette et al. [30] as the finally tagless modelling of domain-specific languages. In this style, we define our grammars abstractly over a parsing algorithm with parser types $p\ a$, where p has instances for a set of type classes containing primitive parsing operators. The type constructor p is parameterised by the type of parsing results.

We define the necessary primitive parser operators in a type class called *CharProductionRule*. Since parser combinators were a motivating example for the development of the concept of applicative functors [145] and the type classes *Applicative* and *Alternative* (repeated below), it is no coincidence that they map perfectly to our needs.[4] Standard applicative functor laws apply, but they may only be valid *morally* in some of our examples (e.g. for the pretty-printer in Section 3.5.1: equivalent expressions might be pretty-printed in different but equivalent ways).

$$\begin{aligned}
&\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \textbf{where}\\
&\quad \mathit{pure} :: a \to f\ a\\
&\quad (\circledast) :: f\ (a \to b) \to f\ a \to f\ b\\
&\textbf{class}\ \mathit{Applicative}\ f \Rightarrow \mathit{Alternative}\ f\ \textbf{where}\\
&\quad \mathit{empty} :: f\ a\\
&\quad (\circleddash) :: f\ a \to f\ a \to f\ a
\end{aligned}$$

[4] In the `grammar-combinators` library, we can unfortunately not use the *Applicative* or *Alternative* type classes, due to a technical reason related to an advanced feature that we do not discuss in this text (Template Haskell lifting of grammars). The library also uses a different notation for the applicative operators, for historical reasons.

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
(<*) :: Applicative f => f a -> f b -> f a
ma <* mb = const <$> ma <*> mb
(*>) :: Applicative f => f a -> f b -> f b
ma *> mb = flip const <$> ma <*> mb
(<$ ) :: Functor f => a -> f b -> f a
f <$ m = const f <$> m
```

Figure 3.1: Definitions of standard related and derived *Applicative* operators `<$>`, `<*`, `*>` and `<$ `.

```
class Alternative p => CharProductionRule p where
  endOfInput :: p ()
  token      :: Char -> p Char
```

In our setting, the *Applicative* operator `<*>` consecutively applies two given parsers. It produces a parsing result by applying the first parser's result to the second parser's. The *pure* primitive parser matches the empty string, producing the value provided as argument as its parsing result. The *Alternative* disjunction operator `<|>` models a choice between two parsers producing the same result type and returns the result of the parser that matched. The *Alternative empty* primitive parser never matches anything, and can therefore return an arbitrary result type.

We will use only two operators that are specific to the parsing domain: the *endOfInput* and *token* parsers. The first matches the end of the input string, returning a unit result and the latter matches a single, specified character in the input stream and returns it on success.

Figure 3.1 shows the definitions of standard related and derived applicative operators `<$>`, `<*`, `*>` and `<$ `, which respectively apply a given function to the result of a rule, ignore a sequenced rule's result, and replace a rule's result with a given value. We temporarily omit operators *many* and *some* which apply a given parser any (resp. any non-zero) number of times, but we come back to them in Section 3.3.3. Note that `<$>` is a synonym for *fmap* in the *Functor* type class and for *Applicative* functors, it is required to satisfy the (defining) property *fmap f m* = *pure f* `<*>` *m*. In the rest of the paper, we will consistently regard it as a derived operator and not discuss its instances.

The gist of the finally tagless technique is visible in the type signature of the parsing functions:[5]

$$line, expr, term, factor :: CharProductionRule\ p \Rightarrow p\ Integer$$
$$digit :: CharProductionRule\ p \Rightarrow p\ Char$$

The functions are defined abstractly over any type constructor p which is an instance of the class *CharProductionRule*. Because of this constrained universal quantification over p, parametricity ensures that these functions can only construct values of type $p\ a$ through the primitive operators defined in class *CharProductionRule* and its parent classes. The finally tagless style allows us to extend or restrict the primitive constructs which a grammar definition has access to. This allows us to write, for example, regular expressions and context-free grammars using the same primitives, and still be able to distinguish them at the type level. Later on, we will similarly make the distinction between extended and normal context-free grammars.

The functions can be defined as follows in terms of the primitive constructs:

$$\begin{aligned}
line &= expr \mathbin{\circledast} endOfInput \\
expr &= (+) \mathbin{\circledS} expr \mathbin{\circledast} token\ \texttt{'+'} \mathbin{\circledast} term \\
&\quad \oplus\ term \\
term &= (*) \mathbin{\circledS} term \mathbin{\circledast} token\ \texttt{'*'} \mathbin{\circledast} factor \\
&\quad \oplus\ factor \\
factor &= read \mathbin{\circledS} some\ digit \\
&\quad \oplus\ token\ \texttt{'('} \mathbin{\circledR} expr \mathbin{\circledast} token\ \texttt{')'} \\
digit &= token\ \texttt{'0'} \oplus token\ \texttt{'1'} \oplus \ldots \oplus token\ \texttt{'9'}
\end{aligned}$$

For every non-terminal, a grammar function is defined directly as a Haskell value using the primitive parsing and combinator operators from the *CharProductionRule* type class and its parents. The definitions look fairly standard for an applicative parser combinator library, even though they are in fact abstract over the parsing algorithm used. The code is fairly concise and reasonably close to the original grammar. Note that we use the $some :: CharProductionRule\ p \Rightarrow p\ a \to p\ [a]$ combinator that will try to match the argument rule one or more times and return the list of results ($some\ r$ is often written as r^+ in grammar definitions) . We will discuss the *some* combinator in more detail in Section 3.3.3.

Note that the above definitions of the parser functions incorporate semantic actions; all parsers return the semantic value of the non-terminal they represent: the integer or char value of the matched string. We consider this coupling of grammar and semantics non-ideal and we will come back to this in Section 3.4.4.

[5]The parsing functions all return the calculated *Integer* value of matches, except for *digit* which just returns a *Char*. It would be slightly cleaner to make *digit* also return a numeric value, but this would be a bit more verbose throughout the text.

3.2.3 Left-recursion

Readers familiar with parser combinator libraries will however have noticed an important problem in the above code. With a mainstream applicative parser combinator library like uu-parsinglib, it does not actually work. The problem is caused by the left-recursion in the definition: *expr*, *term* and *factor* all immediately refer to themselves in the leftmost position of one of their alternatives. A simple top-down parsing algorithm asked to parse an *expr*, would at some point try to match the first alternative for *expr*. The first thing it then needs is a parse of *expr* at the location where it just started looking for an *expr*. Less naive parser combinator libraries exist that can handle left recursion to a certain extent during top-down parsing [76, 53, 150]. However, other libraries like uu-parsinglib and Parsec require the programmer to manually transform the grammar to a non-left-recursive form, such as:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{line}, \mathit{expr}, \mathit{term}, \mathit{factor} :: \mathit{CharProductionRule}\ p \Rightarrow p\ \mathit{Integer}} \\
\multicolumn{3}{l}{\mathit{exprTail}, \mathit{termTail} :: \mathit{CharProductionRule}\ p \Rightarrow p\ (\mathit{Integer} \to \mathit{Integer})} \\
\multicolumn{3}{l}{\mathit{digit} :: \mathit{CharProductionRule}\ p \Rightarrow p\ \mathit{Char}} \\
\mathit{line} & = & \mathit{expr} \mathbin{\circledast} \mathit{endOfInput} \\
\mathit{expr} & = \mathit{foldr}\ (\$) \mathbin{\circledS} & \mathit{term} \circledast \mathit{many}\ \mathit{exprTail} \\
\mathit{exprTail} & = (+) \mathbin{\circledS} & \mathit{token}\ \texttt{'+'} \circledast \mathit{term} \\
\mathit{term} & = \mathit{foldr}\ (\$) \mathbin{\circledS} & \mathit{factor} \circledast \mathit{many}\ \mathit{termTail} \\
\mathit{termTail} & = (*) \mathbin{\circledS} & \mathit{token}\ \texttt{'*'} \circledast \mathit{factor} \\
\mathit{factor} & = \mathit{read} \mathbin{\circledS} & \mathit{some}\ \mathit{digit} \\
 & \oplus & \mathit{token}\ \texttt{'('} \circledcirc \mathit{expr} \mathbin{\circledast} \mathit{token}\ \texttt{')'} \\
\mathit{digit} & = & \mathit{token}\ \texttt{'0'} \oplus \mathit{token}\ \texttt{'1'} \oplus \ldots \oplus \mathit{token}\ \texttt{'9'}
\end{array}
$$

This transformed version of the grammar uses an alternative modelling of operator precedence and associativity which does not rely on left-recursion and can be used with naive top-down parsing algorithms. In fact, standard combinators exist (e.g. *pChainL* in uu-parsinglib) which implement this pattern generically. But even with these combinators, properly identifying and dealing with left recursion remains the responsibility of the programmer.

There is however also a more fundamental problem with the grammar model we have defined so far.

3.2.4 ω-regular grammars considered harmful

The problem lies in the modelling of recursion between non-terminals using recursively defined Haskell values. Haskell supports this thanks to its call-by-need (lazy) evaluation strategy. At first sight, it seems that this allows a faithful representation of the recursive structure of the original grammar. However, closer inspection reveals that what the Haskell values represent is in fact not so much a graph as an infinite tree. We can see this by considering, for example, the most recent definition of the *expr* parser function. Because of Haskell's purely functional nature [201], *expr* is observationally equivalent to what we get if we expand it to its definition, and likewise if we expand subexpressions to their definitions (highlighting the term being expanded in each step):

$$
\begin{aligned}
\mathit{expr} &\equiv \mathit{foldr}\ (\$)\ \text{\textcircled{\$}}\ \boldsymbol{\mathit{term}} && \circledast\ \mathit{many}\ \mathit{exprTail} \\
&\equiv \mathit{foldr}\ (\$)\ \text{\textcircled{\$}}\ (\mathit{foldr}\ (\$)\ \text{\textcircled{\$}}\ \boldsymbol{\mathit{factor}} \circledast \mathit{many}\ \mathit{termTail}) \circledast \mathit{many}\ \mathit{exprTail} \\
&\equiv \mathit{foldr}\ (\$)\ \text{\textcircled{\$}}\ (\mathit{foldr}\ (\$)\ \text{\textcircled{\$}} \\
&\qquad (\mathit{read}\ \text{\textcircled{\$}}\ \mathit{some}\ \mathit{digit} \oplus \mathit{pSym}\ \texttt{'('} \circledcirc \boldsymbol{\mathit{expr}} \circledcirc \mathit{pSym}\ \texttt{')'}) \\
&\qquad\qquad \circledast \mathit{many}\ \mathit{termTail}) \circledast \mathit{many}\ \mathit{exprTail}
\end{aligned}
$$

Figure 3.2 shows a graphical representation of the *expr* parser after some further expansions.

In this way, we find an expansion of the definition of *expr* containing *expr* itself as a subexpression. We can continue expanding forever, obtaining an infinite number of expanded expressions, growing in size, and each indistinguishable from the original definition of *expr*. In fact, for any n, it is even possible to construct a different expression which cannot be distinguished from the original in less than n evaluation steps: take the original definition of *expr*, perform $n + 1$ expansions, and then make a change in the result of the final expansion.

This observation has very real practical consequences. A parser library working with our parser definitions (or those in most parser combinator libraries, which model recursion the same way), and respecting referential transparency (see Section 3.6.3), is fundamentally limited. It cannot, for example, print a representation of the grammar in any finite number of evaluation steps n, because it might be looking at another grammar that can only be distinguished from the original after more than n computation steps. Similarly, no parsing library using this grammar model can calculate parsing tables completely upfront, fully execute a grammar transformation, or ever perform a sanity check for LL(1)-ness.

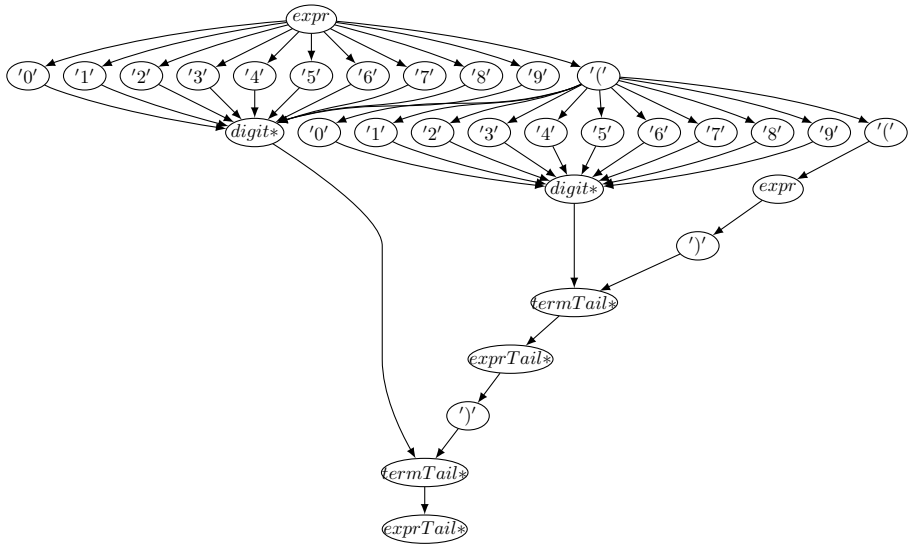

Figure 3.2: A graphical representation of the *expr* parser after some expansions of its definition (see Section 3.2.4). The *expr* node at the right (as well as some of the other nodes), can be expanded further, arbitrarily deep.

Because of the similarity of these "infinite-tree" grammar definitions to what one might see as infinite regular grammars, we will refer to this grammar model as ω-regular, rather than context-free.[6]

3.3 A different modelling of recursion

These fundamental limitations are in fact an instance of a more general problem. For many DSLs, object language terms feature a mutually recursive structure and it is often advantageous to be able to observe this structure in the metalanguage. For example, Sheard [206] cites the problem as one of the main reasons to build a special-purpose hardware design language instead of embedding the DSL in a general-purpose programming language.

[6]Our usage of the term ω-regular grammars is related to, but not the same as other usages in the literature. For some insights from language theory, we refer to Park [175], who proves a relation between functions using $*$ and $\dagger$ operators and minimal and maximal fixpoints. In those terms, what we call ω-regular grammars correspond to expressions generated by the regular operators with $*$ replaced by $\dagger$.

So, what could be a better way to represent recursion? In this section and the next, we first consider the *fix* construct which Carette et al. use in their finally tagless modelling of a typed lambda calculus [30], and show that it is not perfect for our needs. After that, we present our approach, which we introduce step by step. We incrementally transform the parser combinators grammar introduced before, at first postponing well-typedness concerns until we are ready to show the solution for this. We will clearly mark all untyped pseudo-code as such in what follows.

3.3.1 Fixing recursion?

In their finally tagless model of a typed lambda calculus, Carette et al. use a *fix* construct to model recursion. In our setting, such a construct would resemble the following:

```
class FixProductionRule p where
  fix :: (p v → p v) → p v
```

Recursive production rules could then be defined as follows:

```
expr :: (CharProductionRule p, FixProductionRule p) ⇒ p Integer
expr = fix $ λself →      (+) <$> self <* token '+' <*> term
                     <|> term
```

This *fix* operator employs a finally tagless style and effectively makes object language recursion observable in the meta-language, allowing meta-language algorithms to interpret the recursion in the way that they need to. However, if we consider the above definition more closely, it turns out that we missed a recursive occurence of *expr*. Indeed, the grammar is *mutually* recursive, with *term* referencing *factor*, and *factor* referencing *expr* again. Indeed, what we require is an encoding of this mutual recursion, allowing us to model the combined fixpoint of the following functions. The example uses an omitted primitive *fix3* which can be defined in terms of *fix*:

```
expr, term, factor :: CharProductionRule p ⇒
  p Integer → p Integer → p Integer → p Integer
digit :: CharProductionRule p ⇒ p Char
expr   e t f = (+) <$> e <* token '+' <*> t
           <|> term e t f
term   e t f = (*) <$> t <* token '*' <*> f
           <|> factor e t f
factor e t f = read <$> some digit
           <|> token '(' *> e <* token ')'
```

$$
\begin{aligned}
&\mathit{digit} \qquad = \mathit{token}\ \texttt{'0'} \oslash \mathit{token}\ \texttt{'1'} \oslash \ldots \oslash \mathit{token}\ \texttt{'9'} \\
&\mathit{line}' :: (\mathit{FixProductionRule}\ p, \mathit{CharProductionRule}\ p) \Rightarrow p\ \mathit{Integer} \\
&\mathit{line}' = e \circledast \mathit{endOfInput}\ \textbf{where}\ (e, t, f) = \mathit{fix}_3\ \mathit{expr}\ \mathit{term}\ \mathit{factor}
\end{aligned}
$$

This approach seems successful, although the syntax is somewhat verbose. In a typical lambda calculus, we could make it more concise by defining a object-language record type containing three fields *expr*, *term* and *factor*. We could then construct a value of that record type as the fixpoint of a single function based on the above functions. However, to do this, we need record types in our object language, and adding them to our parsing DSL purely for this technical reason is not our preferred solution. We conjecture that the above syntax can also be made more concise with a superficial Haskell extension similar to the recursive do notation by Erkok and Launchbury [70]. In fact, we think that our *FixProductionRule* type class can be seen more generally as analogous to their *MonadRec* (later renamed to *MonadFix*) for applicative functors.

However, we can also choose a different way to define a mutual recursion construct: we model the record suggested above as a function from a finite domain to production rules:

warning: untyped pseudo-code...

$$
\begin{aligned}
&\textbf{data}\ \mathit{Domain} = \mathit{Line} \mid \mathit{Expr} \mid \mathit{Term} \mid \mathit{Factor} \mid \mathit{Digit} \\
&\textbf{type}\ \mathit{Grammar} = \mathit{Domain} \to p\ \textbf{?} \\
&\textbf{class}\ \mathit{FixGramProductionRule}\ p\ \textbf{where} \\
&\quad \mathit{fixG} :: ((\mathit{dom} \to p\ \textbf{?}) \to (\mathit{dom} \to p\ \textbf{?})) \to \mathit{dom} \to p\ \textbf{?}
\end{aligned}
$$

It turns out that this idea can be elaborated to a workable solution; there are ways to properly type this *fixG* construct (we will encounter such techniques further on). However, this *fixG* construct deviates from standard practice in grammar definitions, since it can be used multiple times in different locations in the same grammar. In standard context-free grammar (CFG) formalisms, all the recursion occurs at the top level. Our proposal therefore does not introduce an actual *fix* construct, but instead, we model the grammar as the function of which it is the fixed point.

3.3.2 Toward context-free grammars

The idea to model the grammar in that way corresponds to a classic technique from the functional programmer's bag of tricks: defining the grammar with open recursion. We factor out all recursive calls in the definition by calls to a *self* function that it receives as an argument.

warning: untyped pseudo-code...

$$
\begin{aligned}
&g_{arith} :: \mathit{CharProductionRule}\ p \Rightarrow (\mathit{Domain} \to p\ \textbf{\textit{?}}) \to \mathit{Domain} \to p\ \textbf{\textit{?}} \\
&g_{arith}\ \mathit{self}\ \mathit{Line} = \mathit{self}\ \mathit{Expr} \circledast \mathit{endOfInput} \\
&g_{arith}\ \mathit{self}\ \mathit{Expr} = \mathit{self}\ \mathit{Expr} \circledast \mathit{token}\ \texttt{'+'} \circledast \mathit{self}\ \mathit{Term} \\
&\qquad\qquad \oslash\ \mathit{self}\ \mathit{Term} \\
&\ldots
\end{aligned}
$$

Even though it is not clear how to type this solution, this model does effectively solve the problem of unobservable recursion. Algorithms working with the grammar can provide custom interpretations of recursion to suit their needs. However, the *self* parameter obscures the definition while its role is fairly technical and as a model of a primitive object language recursion primitive, it stylistically differs from the other object language primitives which are defined in type classes.

A more *finally tagless* recursion primitive can be defined by replacing the self parameter by a primitive $\langle\cdot\rangle$[7], defined in an additional type class *RecProductionRule*. Production rule types p become linked to the domain for which recursive calls are allowed, which we reflect in the *RecProductionRule* class's parameters and its functional dependencies:

warning: untyped pseudo-code...

$$
\begin{aligned}
&\textbf{class}\ \mathit{CharProductionRule}\ p \Rightarrow \mathit{RecProductionRule}\ p\ \mathit{dom} \mid p \to \mathit{dom}\ \textbf{where} \\
&\quad \langle\cdot\rangle :: \mathit{dom} \to p\ \textbf{\textit{?}} \\
&g_{arith} :: \mathit{RecProductionRule}\ p\ \mathit{Domain} \Rightarrow \mathit{Domain} \to p\ \textbf{\textit{?}} \\
&g_{arith}\ \mathit{Line} = \langle \mathit{Expr} \rangle \circledast \mathit{endOfInput} \\
&g_{arith}\ \mathit{Expr} = \langle \mathit{Expr} \rangle \circledast \mathit{token}\ \texttt{'+'} \circledast \langle \mathit{Term} \rangle \\
&\qquad\qquad \oslash\ \langle \mathit{Term} \rangle \\
&\ldots
\end{aligned}
$$

This modelling is equivalent to the one using open recursion using the *self* parameter. In fact, we will define a translation algorithm in Section 3.5.2, turning the representation using *RecProductionRule* into the one using a *self* parameter. This algorithm will be useful for technical reasons.

3.3.3 Extended context-free grammars

There is actually one thing still missing in this definition: we have craftily hidden the use of the *some* operator in the production rule for *Factor* by including it in the ellipsis above:

[7] We use $\langle\cdot\rangle$ as a notation for a one-argument function with the dot as the placeholder for the argument. This means that $\langle \mathit{Term} \rangle$ denotes $\langle\cdot\rangle$ applied to the argument *Term*.

```
                                    warning: untyped pseudo-code...
g_arith Factor = token '(' ⊛ ⟨Expr⟩ ⊛ token ')'
               ⓛ some ⟨Digit⟩
```

The *some* operator is defined as follows (together with its sibling *many*):

```
many, some :: CharProductionRule p ⇒ p a → p [a]
many p = pure [] ⓛ some p
some p = (:) ⓢ p ⊛ many p
```

These definitions essentially rely on unobservable meta-language recursion, which we need to replace with an observable form of recursion as well. In addition to the $\langle\cdot\rangle$ primitive recursion operator *RecProductionRule* type class, we define restricted versions of *many* and *some* in the *LoopProductionRule* type class, as follows:

```
                                    warning: untyped pseudo-code...
class RecProductionRule p dom ⇒
  LoopProductionRule p dom | p → dom where
  ⟨·⟩* :: dom → p ?
  ⟨·⟩+ :: dom → p ?
```

The grammar type and the production rules for factor now become:

```
                                    warning: untyped pseudo-code...
g_arith :: LoopProductionRule p Domain ⇒ Domain → p ?
...
g_arith Factor = token '(' ⊛ ⟨Expr⟩ ⊛ token ')'
               ⓛ ⟨Digit⟩+
...
```

Note that the operators $\langle\cdot\rangle^*$ and $\langle\cdot\rangle^+$ are less powerful than the *many* and *some* operators which allow any production rule (not just recursive references) to be quantified. This restriction is needed to make it possible for grammar algorithms to interpret these object-language constructs appropriately. However, we think the new constructs are still general enough for most purposes. Grammar authors may sometimes need to split out a production rule to be quantified into an additional non-terminal.

Note also that we are in fact replacing library algorithms (*many* and *some*) by what are essentially new builtin operators in our object language. This is unfortunate, but it is part of the cost we pay in our approach to rule out ω-regular grammars. We will show in Section 3.5.4 that $\langle\cdot\rangle^*$ and $\langle\cdot\rangle^+$ support standard grammar transformations.

3.3.4 Typing context-free grammars

So, this last representation is promising in the sense that algorithms can instantiate the abstract grammar with their own interpretations of recursion and the primitive parser combinator operations. Unfortunately, it is not clear how to properly define the value that each of the production rules produce. In a typical implementation, our example grammar above produces result values of the following Abstract Syntax Tree (AST) type:

```
newtype Line   = SExpr Expr
data Expr      = Sum Expr Term
               | STerm Term
data Term      = Product Term Factor
               | SFactor Factor
data Factor    = Paren Expr
               | Number [Digit]
newtype Digit = MkDigit Char
```

When we now try to define the type of g_{arith}, we run into another problem. It turns out that our modelling of the grammar as a function from the grammar domain to production rules forces all production rules to produce the same type of values:

```
                                  warning: untyped pseudo-code...
g_arith :: CharProductionRule p ⇒ Domain → p ?
```

Similarly, if we try to define the type of the $\langle\cdot\rangle$ operator, we cannot express that the parser result of $\langle idx \rangle$ should vary based on the value of idx.

```
                                  warning: untyped pseudo-code...
class CharProductionRule p ⇒ RecProductionRule p dom | p → dom where
  ⟨·⟩ :: dom → p ?
g_arith Line = SExpr <$> ⟨Expr⟩ <* endOfInput
g_arith Expr = Sum   <$> ⟨Expr⟩ <* token '+' <* ⟨Term⟩
             <|> STerm <$> ⟨Term⟩
...
```

The essential problem here is that all our non-terminals are of type *Domain*, so that all references $\langle idx \rangle$ must share a single result type (because Haskell is not dependently typed, see Section 3.6.2). Therefore, we cannot express that non-terminal *Line* corresponds to a different type of semantic values than non-terminal *Expr*.

3.4 Typing our recursion model

It turns out that we can define precise types for the untyped pseudo-code in the previous section by using a representation of non-terminals not sharing a single type.

3.4.1 Representing non-terminals

We model the set of non-terminals (the *domain*) as a "subkind" with witnesses, using the technique employed by Rodriguez et al. [197] to model indices into a set of mutually recursive data types in multirec. The Generalized Algebraic Data Type (GADT) [207] ϕ_{arith} is a "subkind"; a family of singleton types that represents the domain of our arithmetic expressions grammar. Note that Haskell's separation between type and value name spaces allows the data constructor *Expr* and the type *Expr* to share the same name.

$$\begin{aligned}\textbf{data}\ \phi_{arith}\ ix\ \textbf{where}\ \mathit{Line} &:: \phi_{arith}\ \mathit{Line}\\ \mathit{Expr} &:: \phi_{arith}\ \mathit{Expr}\\ \mathit{Term} &:: \phi_{arith}\ \mathit{Term}\\ \mathit{Factor} &:: \phi_{arith}\ \mathit{Factor}\\ \mathit{Digit} &:: \phi_{arith}\ \mathit{Digit}\end{aligned}$$

We use the previously defined AST types *Line*, *Expr*, *Term*, *Factor* and *Digit* to represent the non-terminals at the type-level. The GADT ϕ_{arith} introduces, for every non-terminal ix, a term of type $\phi_{arith}\ ix$, serving as a proof that ix is part of the domain ϕ_{arith}. With this "subkind" representation, the compiler will guarantee that a function f typed $\forall ix.\phi\ ix \to \ldots$ is polymorphic over precisely the five non-terminal types in the domain.

3.4.2 A first typing of our grammars and the recursion operator $\langle\cdot\rangle$

This representation of our domain as a subkind with witnesses allows us to present a first proper typing of our grammars and the recursion operator $\langle\cdot\rangle$, which we introduced as untyped pseudo-code before.

We first consider the primitive recursion construct $\langle\cdot\rangle$, defined in the *RecProductionRule* type class. Within a grammar with domain ϕ (e.g. type constructor ϕ_{arith} above), we can now declare that $\langle\cdot\rangle$ can be invoked on any value idx of type $\phi\ ix$ for some ix. The expression $\langle idx\rangle$ represents a parser for that non-terminal, and returns a value of type ix: the AST type

of the non-terminal. The constructs $\langle\cdot\rangle^*$ and $\langle\cdot\rangle^+$ are defined analogously in the *LoopProductionRule* type class. We use functional dependencies to couple production rules with the domain for which recursive references can be made.

$$
\begin{aligned}
&\textbf{class } \mathit{CharProductionRule}\ p \Rightarrow \mathit{RecProductionRule}\ p\ \phi \mid p \to \phi \textbf{ where}\\
&\quad \langle\cdot\rangle \quad :: \phi\ ix \to p\ ix\\
&\textbf{class } \mathit{RecProductionRule}\ p\ \phi \Rightarrow \mathit{LoopProductionRule}\ p\ \phi \mid p \to \phi \textbf{ where}\\
&\quad \langle\cdot\rangle^* :: \phi\ ix \to p\ [ix]\\
&\quad \langle\cdot\rangle^+ :: \phi\ ix \to p\ [ix]
\end{aligned}
$$

$$
\begin{aligned}
&g_{arith} :: \mathit{LoopProductionRule}\ p\ \phi_{arith} \Rightarrow \phi_{arith}\ ix \to p\ ix\\
&g_{arith}\ \mathit{Line} \quad = \mathit{SExpr} \quad \$\ \langle \mathit{Expr}\rangle \circledast \mathit{endOfInput}\\
&g_{arith}\ \mathit{Expr} \quad = \mathit{Sum} \quad \$\ \langle \mathit{Expr}\rangle \circledast \mathit{token}\ \texttt{'+'} \circledast \langle \mathit{Term}\rangle\\
&\qquad\qquad \oplus\ \mathit{STerm} \quad \$\ \langle \mathit{Term}\rangle\\
&g_{arith}\ \mathit{Term} \quad = \mathit{Product} \quad \$\ \langle \mathit{Term}\rangle \circledast \mathit{token}\ \texttt{'*'} \circledast \langle \mathit{Factor}\rangle\\
&\qquad\qquad \oplus\ \mathit{SFactor} \quad \$\ \langle \mathit{Factor}\rangle\\
&g_{arith}\ \mathit{Factor} = \mathit{Paren} \quad \mathbb{S}\ \mathit{token}\ \texttt{'('} \circledast \langle \mathit{Expr}\rangle \circledast \mathit{token}\ \texttt{')'}\\
&\qquad\qquad \oplus\ \mathit{Number} \quad \$\ \langle \mathit{Digit}\rangle^+\\
&g_{arith}\ \mathit{Digit} \quad = \mathit{MkDigit}\ \$\ (\mathit{token}\ \texttt{'0'} \oplus \mathit{token}\ \texttt{'1'} \oplus \ldots \oplus \mathit{token}\ \texttt{'9'})
\end{aligned}
$$

3.4.3 Semantic value families

With this typed version of our grammars, we are making good progress, but this representation of recursion in our object language is still not fully satisfactory. The problem is in the result types of the recursive calls. Algorithms are now free to plug in their own interpretation of object-language recursion, but they are still forced to work with the full AST types as result types of the recursive calls. In many cases, we want to be able to plug in different representation types, often a different type for every non-terminal.

We can make this more concrete for the example of our grammar language. There, the AST result types might at first sight seem satisfactory, since conceptually, the AST is part of the definition of the grammar and practically, we can apply any set of semantic actions once we have the AST, by implementing them as a structural fold (a *catamorphism*) over the AST. However, this approach also allows semantics that are not formulated as such catamorphisms. For example, the following semantics negates all literals that are inside an uneven number of parentheses:

$$
\begin{aligned}
&\mathit{weirdSem} :: \phi_{arith}\ ix \to ix \to \mathit{Int}\\
&\mathit{weirdSem}\ idx\ v = \mathit{go}\ \mathit{False}\ idx\ v
\end{aligned}
$$

```
  where unMkDigit (MkDigit c) = c
        neg :: Bool → Int → Int
        neg True x = −x
        neg False x = x
        go :: Bool → φ_arith ix → ix → Int
        go inv Line (SExpr e) = go inv Expr e
        go inv Expr (STerm t) = go inv Term t
        go inv Expr (Sum e t) = go inv Expr e + go inv Term t
        go inv Term (SFactor f) = go inv Factor f
        go inv Term (Product t f) = go inv Term t ∗ go inv Factor f
        go inv Factor (Paren e)  = go (¬ inv) Expr e
        go inv Factor (Number n) = neg inv (read (map unMkDigit n))
        go _ Digit d = read (unMkDigit d : [ ])
```

A disadvantage of semantics like *weirdSem* (which are not formulated as catamorphisms) is that they are inherently coupled to a top-down matching order: the semantics has to be applied to the top AST node once it is available. A bottom-up parser already reduces production rules before it is sure at which depth the production will fit in the final AST and it might want to force the semantics to already be applied at such times during parsing, e.g. for optimization purposes.[8] However, this is inherently not possible for semantics like *weirdSem*, whose behaviour depends on the depth of the match in the final AST. Also for semantic reasons, we find it preferable to define grammar semantics as catamorphisms over the abstract syntax trees and exclude definitions like *weirdSem*.

We can achieve this by abstracting our model even further, this time over *semantic value families*. These are data families [204] indexed by the non-terminal types we've seen before. A semantic value family r associates each non-terminal type ix with the type of its semantic value $r\ ix$. We define one such family for the ϕ_{arith} domain, written $[\![]\!]^{value}_{\cdot}$. For clarity, in the notation $[\![]\!]^{value}_{\cdot}$, the dot is a placeholder for the type argument ix and, for example, in the constructor $[\![\cdot]\!]^{value}_{Line}$, the dot is a placeholder for the integer argument of the constructor, so that for example $[\![3]\!]^{value}_{Line}$ is a value of type $[\![]\!]^{value}_{Line}$.

```
data family [[]]^value_. ix
newtype instance [[]]^value_. Line    = [[·]]^value_Line Integer
newtype instance [[]]^value_. Expr    = [[·]]^value_Expr Integer
newtype instance [[]]^value_. Term    = [[·]]^value_Term Integer
newtype instance [[]]^value_. Factor  = [[·]]^value_Factor Integer
newtype instance [[]]^value_. Digit   = [[·]]^value_Decimal Char
```

[8] Such a parser would typically use Haskell's *seq* function to force the semantics to actually be evaluated at such moments.

This semantic value family specifies that all of our non-terminals have *Integer* semantic values (their calculated value), except for *Digit*, which has a character as its semantic value.

We can now redefine the primitive recursion operator to return values of some semantic value family r, which (like the domain ϕ) is required to be the same throughout the grammar by the *RecProductionRule* type class's functional dependencies. Note that we provide default definitions of operators $\langle\cdot\rangle^*$ and $\langle\cdot\rangle^+$ in terms of each other and the $\langle\cdot\rangle$ operator. However, we expect instances of *LoopProductionRule* to provide custom definitions of at least one of both operators, otherwise they will behave as their ω-regular analogs.

```
class CharProductionRule p ⇒
        RecProductionRule p φ r | p → φ, p → r where
  ⟨·⟩ :: φ ix → p (r ix)
class RecProductionRule p φ r ⇒
        LoopProductionRule p φ r | p → φ, p → r where
  ⟨·⟩* :: φ ix → p [r ix]
  ⟨idx⟩* = pure [] ⓛ ⟨idx⟩+
  ⟨·⟩+ :: φ ix → p [r ix]
  ⟨idx⟩+ = (:) <$> ⟨idx⟩ <*> ⟨idx⟩*
```

In the next section, we will show how this definition allows us to decouple the grammar from a semantic value family. Here we can already show how we can use the new definition of $\langle\cdot\rangle$ to make the grammar work for the family $[\![]\!]^{value}_{\cdot}$. Like before, the mixing of semantic values in the grammar hampers the grammar's readability:

$$
\begin{array}{l}
g_{arith} :: \mathit{LoopProductionRule}\ p\ \phi_{arith}\ [\![]\!]^{value}_{\cdot} \Rightarrow \phi_{arith}\ ix \to p\ [\![]\!]^{value}_{ix} \\
g_{arith} = \mathit{gram} \\
\quad \textbf{where}\ \mathit{gram}\ \mathit{Line} \quad = sa_{Expr} \quad \$\ \langle \mathit{Expr} \rangle \circledast \mathit{endOfInput} \\
\qquad \mathit{gram}\ \mathit{Expr} \quad = sa_{+} \quad \$\ \langle \mathit{Expr} \rangle \circledast \mathit{token}\ \texttt{'+'} \circledast \langle \mathit{Term} \rangle \\
\qquad\qquad\qquad \oplus\ sa_{Term} \quad \$\ \langle \mathit{Term} \rangle \\
\qquad \mathit{gram}\ \mathit{Term} \quad = sa_{*} \quad \$\ \langle \mathit{Term} \rangle \circledast \mathit{token}\ \texttt{'*'} \circledast \langle \mathit{Factor} \rangle \\
\qquad\qquad\qquad \oplus\ sa_{Factor}\ \$\ \langle \mathit{Factor} \rangle \\
\qquad \mathit{gram}\ \mathit{Factor} = sa_{()} \quad \mathbin{<\$} \mathit{token}\ \texttt{'('} \circledast \langle \mathit{Expr} \rangle \circledast \mathit{token}\ \texttt{')'} \\
\qquad\qquad\qquad \oplus\ sa_{\texttt{lit}} \quad \$\ \langle \mathit{Digit} \rangle^{+} \\
\qquad \mathit{gram}\ \mathit{Digit} \quad = [\![\cdot]\!]^{value}_{Decimal}\ \$ \\
\qquad\qquad\qquad\qquad (\mathit{token}\ \texttt{'0'} \oplus \mathit{token}\ \texttt{'1'} \oplus \ldots \oplus \mathit{token}\ \texttt{'9'}) \\
\quad sa_{Expr}\ [\![v]\!]^{value}_{Expr} = [\![v]\!]^{value}_{Line} \\
\quad sa_{+}\ [\![v_1]\!]^{value}_{Expr}\ [\![v_2]\!]^{value}_{Term} = [\![v_1 + v_2]\!]^{value}_{Expr} \\
\quad sa_{Term}\ [\![v]\!]^{value}_{Term} = [\![v]\!]^{value}_{Expr} \\
\quad sa_{*}\ [\![v_1]\!]^{value}_{Term}\ [\![v_2]\!]^{value}_{Factor} = [\![v_1 * v_2]\!]^{value}_{Term} \\
\quad sa_{Factor}\ [\![v]\!]^{value}_{Factor} = [\![v]\!]^{value}_{Term} \\
\quad sa_{()}\ [\![v]\!]^{value}_{Expr} = [\![v]\!]^{value}_{Factor} \\
\quad sa_{\texttt{lit}} = [\![\cdot]\!]^{value}_{Factor} \circ \mathit{read} \circ \mathit{map}\ (\lambda [\![c]\!]^{value}_{Decimal} \to c)
\end{array}
$$

3.4.4 Semantic value family polymorphism

So, the next question is: can we decouple the grammar from its semantics? Clearly, there are reasons other than aesthetic ones for doing this. For our arithmetic expressions, we have already seen a grammar producing AST values and a grammar calculating integer values for them. Other useful semantic processors (a set of semantic actions for the non-terminals in a grammar) transform the same expressions into reverse polish notation, construct an AST or perform some form of side effects in a *Monad*. It is clear that we can improve the modularity of our grammar language by decoupling a grammar from sets of semantic actions.

Our solution for this decoupling here uses (again) techniques from the multirec generic programming library [197], which uses a representation of mutually recursive data types as the fixed point of a *pattern functor* to manipulate them in generic algorithms. The AST data types shown previously are an example of such a family of mutually recursive data types. To understand the pattern functor for this family, it is instructive to define it in two steps. First, we define a single indexed data type that represents the entire family of AST types:

$$
\begin{array}{l}
\textbf{data}\ AST_{arith}\ ix\ \textbf{where} \\
\quad SExprAST \ :: AST_{arith}\ Expr \rightarrow \qquad\qquad\qquad\quad \rightarrow AST_{arith}\ Line \\
\quad SumAST \ :: AST_{arith}\ Expr \rightarrow AST_{arith}\ Term \rightarrow AST_{arith}\ Expr \\
\quad STermAST \ :: AST_{arith}\ Term \rightarrow \qquad\qquad\qquad\quad \rightarrow AST_{arith}\ Expr \\
\quad ProductAST :: AST_{arith}\ Term \rightarrow AST_{arith}\ Factor \rightarrow AST_{arith}\ Term \\
\quad SFactorAST :: AST_{arith}\ Factor \rightarrow \qquad\qquad\qquad\quad \rightarrow AST_{arith}\ Term \\
\quad ParenAST \ :: AST_{arith}\ Expr \rightarrow \qquad\qquad\qquad\quad \rightarrow AST_{arith}\ Factor \\
\quad NumberAST :: [AST_{arith}\ Digit] \rightarrow \qquad\qquad\qquad\quad \rightarrow AST_{arith}\ Factor \\
\quad MkDigitAST :: Char \qquad\qquad\qquad\quad \rightarrow \qquad\qquad\qquad\quad \rightarrow AST_{arith}\ Digit
\end{array}
$$

This AST_{arith} data type has constructor analogous to those of all the AST data types (see Section 3.3.4 p. 95). It is indexed by the type of terms represented, so that we can still distinguish a *Expr* from a *Term* value through their type.

For generic programming purposes, it is then useful to be able to work with a version of this data type that does not represent an arithmetic term as a full tree of sub-terms, but rather only represent its top-level structure where sub-trees are represented in a parametrisable way. We can obtain such a data type from the AST_{arith} type above, by replacing recursive positions of type $AST_{arith}\ ix$ with values $r\ ix$ of an argument semantic value family r. This amounts to rewriting AST_{arith} in an *open recursive* manner, and gives us the family's *pattern functor*:

$$
\begin{array}{l}
\textbf{data}\ PF_{arith}\ r\ ix\ \textbf{where} \\
\quad SExprF \ :: r\ Expr \rightarrow \qquad\qquad\quad \rightarrow PF_{arith}\ r\ Line \\
\quad SumF \ :: r\ Expr \rightarrow r\ Term \rightarrow PF_{arith}\ r\ Expr \\
\quad STermF \ :: r\ Term \rightarrow \qquad\qquad\quad \rightarrow PF_{arith}\ r\ Expr \\
\quad ProductF :: r\ Term \rightarrow r\ Factor \rightarrow PF_{arith}\ r\ Term \\
\quad SFactorF :: r\ Factor \rightarrow \qquad\qquad\quad \rightarrow PF_{arith}\ r\ Term \\
\quad ParenF \ :: r\ Expr \rightarrow \qquad\qquad\quad \rightarrow PF_{arith}\ r\ Factor \\
\quad NumberF :: [r\ Digit] \rightarrow \qquad\qquad\quad \rightarrow PF_{arith}\ r\ Factor \\
\quad MkDigitF :: Char \qquad\quad \rightarrow \qquad\qquad\quad \rightarrow PF_{arith}\ r\ Digit
\end{array}
$$

In this definition, the semantic value family r defines what values to keep for subtrees of AST nodes. We will therefore sometimes refer to it as the *subtree representation functor* (our terminology). The PF_{arith} pattern functor values are still tagged with the AST node type they represent. For example, the constructor *SumF* constructs a pattern functor value corresponding to the first constructor of the *Expr* AST type, so the result of *SumF* is tagged with the *Expr* type.

Note that we do not use the type functor combinators that Rodriguez et al. [197] define to build pattern functors. This is because we do not require the generic operations which can be derived over these combinators and because we think our direct presentation of the pattern functor is clearer.

They also define a type family PF mapping domains ϕ to their pattern functor $PF\ \phi$. The following type family instance registers PF_{arith} as the pattern functor for domain ϕ_{arith}:

type instance $PF\ \phi_{arith} = PF_{arith}$

Like for simply recursive types, data types isomorphic to our original AST data types can be recovered from this pattern functor by taking its fixed point using a type-level fixpoint combinator. But the pattern functor allows doing more with the AST values. Rodriguez et al. demonstrate how to go back and forth between a type ix in a domain ϕ and its *one-level unfolding* of type $PF\ \phi\ I_*\ ix$ (with I_* a wrapping identity functor: $I_*\ ix \sim ix$). In this way, a value of the AST type *Expr* can be converted into an unfolded value of type $PF_{arith}\ I_*\ Expr$, exposing the top-level of its structure (similar to the unfold operation for iso-recursive types, see [187, pp. 276–277]). Generic operations on instances of the pattern functor can then be used to implement various generic algorithms. All of this gives impressive, elegant and powerful generic programming machinery, but for our purposes, the pattern functor is useful in another way.

A powerful feature of the pattern functor is that it abstracts over the subtree representation functor r, allowing subtrees to be represented differently than as full subtrees. If we take our semantic value family $[\![]\!]_{\cdot}^{value}$ as this subtree representation functor (instead of the wrapping identity functor I_*), then subtrees in the one-level unfolding of an AST are represented just by their calculated value (instead of a full sub-AST). For example, the value $(SumF\ [\![15]\!]_{Expr}^{value}\ [\![3]\!]_{Term}^{value})$ of type $(PF_{arith}\ [\![]\!]_{\cdot}^{value}\ Expr)$ represents an *Expr* value, constructed as the sum of another *Expr* and a *Term*, where we only know that the arithmetic value of the left hand side *Expr* and the right hand side *Term* are respectively 15 and 3. In general, the pattern functor PF_{arith} allows us to represent an AST where subtrees have already been processed into a semantic value, and this turns out to be precisely the vehicle we need for modelling the collaboration between a grammar, a parsing algorithm and a semantic processor.

Let us consider production rule Expr → Expr '+' Term as an example. Figure 3.3 shows a graphical illustration of this collaboration (for a semantic processor working with a semantic value family r). In Figure 3.3a, the parser has matched the right-hand side elements of the production rule and has obtained their semantic values, typed $r\ Expr$, *Char* and $r\ Term$. In Figure 3.3b, the grammar specifies how to combine these three values to the single-layer top of an AST, constructing a value of type $PF_{arith}\ r\ Expr$. For this production rule, the *SumF* constructor is used, throwing away the parse result for the token '+'. Note that the grammar does not make any assumptions about the semantic value family r. In Figure 3.3c, the semantic processor accepts the

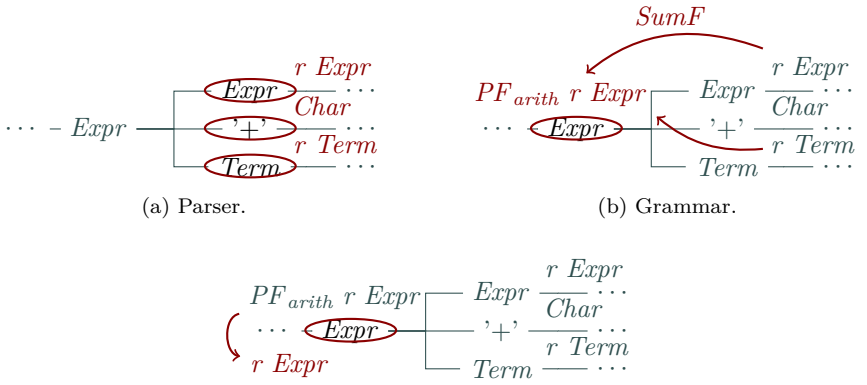

Figure 3.3: A graphical representation of the collaboration between parser, grammar and semantic processor, using ϕ_{arith}'s pattern functor over a semantic value family r as an intermediate representation. The parser matches the right-hand side elements of a production rule and obtains their semantic values. The grammar specifies how to combine these three values to the single-layer top of an AST, constructing a value of type $PF_{arith}\ r\ Expr$. It does not make any assumptions about the semantic value family r. The semantic processor accepts the constructed $PF_{arith}\ r\ Expr$ value, calculates the combined semantic value and returns a processed value of type $r\ Expr$ to the parser for use in subsequent matches.

constructed $PF_{arith}\ r\ Expr$ value, calculates the combined semantic value and returns a processed value of type $r\ Expr$ to the parser for use in subsequent matches. Note that nothing here assumes any specific matching order (top-down vs. bottom-up).

For readers who are familiar with the terminology [147], it is interesting to note that the grammar's action on the semantic values is an anamorphism from concrete parsing trees to our mutually recursive data types. Correspondingly, the semantic processor specifies a catamorphism for the mutually recursive data types, and multirec's pattern functor machinery allows the parser to explicitly fuse the two together according to its own matching order.

With this machinery, we can effectively decouple grammars from their semantic processors and vice versa. In the next section, we take a look at the resulting code to see how it all fits together.

3.4.5 So what do we get?

So, our finally tagless model of observable recursion is completed; we know how to abstract from the representation of return values of recursive calls, and we can even model the interaction between a grammar and its semantic processors, and abstract the grammar from the processors. We finally show the resulting definition of our running example grammar:

$$
\begin{aligned}
&\textbf{type } \mathit{ExtendedCFG}\ \phi = \forall p\ r\ ix.\mathit{LoopProductionRule}\ p\ \phi\ r \Rightarrow \\
&\qquad\qquad\qquad\qquad \phi\ ix \to p\ (\mathit{PF}\ \phi\ r\ ix) \\
&g_{arith} :: \mathit{ExtendedCFG}\ \phi_{arith} \\
&g_{arith}\ \mathit{Line} \quad = \mathit{SExprF} \quad \$\ \langle \mathit{Expr} \rangle \circledast \mathit{endOfInput} \\
&g_{arith}\ \mathit{Expr} \quad = \mathit{STermF} \quad \$\ \langle \mathit{Term} \rangle \\
&\qquad\qquad \oplus\ \mathit{SumF} \quad \$\ \langle \mathit{Expr} \rangle \circledast \mathit{token}\ \texttt{'+'} \circledast \langle \mathit{Term} \rangle \\
&g_{arith}\ \mathit{Term} \quad = \mathit{SFactorF}\ \$\ \langle \mathit{Factor} \rangle \\
&\qquad\qquad \oplus\ \mathit{ProductF}\ \$\ \langle \mathit{Term} \rangle \circledast \mathit{token}\ \texttt{'*'} \circledast \langle \mathit{Factor} \rangle \\
&g_{arith}\ \mathit{Factor} = \mathit{NumberF}\ \$\ \langle \mathit{Digit} \rangle^{+} \\
&\qquad\qquad \oplus\ \mathit{ParenF} \quad \circledast\ \mathit{token}\ \texttt{'('} \circledast \langle \mathit{Expr} \rangle \circledast \mathit{token}\ \texttt{')'} \\
&g_{arith}\ \mathit{Digit} \quad = \mathit{MkDigitF}\ \$\ (\mathit{token}\ \texttt{'0'} \oplus \mathit{token}\ \texttt{'1'} \oplus \ldots \oplus \mathit{token}\ \texttt{'9'})
\end{aligned}
$$

We first define a general *ExtendedCFG* type synonym (CFG for context-free grammar), expressing that an extended context-free grammar is a function returning a production rule for every non-terminal. The $\forall\cdot$ quantification expresses that it must be defined for any production rule interpretation type p supporting the context-free grammar operations of type class *LoopProductionRule* (and its parents *Applicative*, *Alternative*, *CharProductionRule*, *RecProductionRule*). It must also work for any semantic value family r, producing values of the pattern functor $PF\ \phi$ with r as the subtree representation type.

Our grammar g_{arith} is an extended context-free grammar for the domain ϕ_{arith}. Its production rules are defined using the combinators we saw before, and values of $PF_{arith}\ r$ are produced using the pattern functor's constructors. Stylistically, the pattern functor constructors end up at the beginning of each production rule, giving a nice visual tagging of the rules, and defining for each production rule what kind of AST node it corresponds to. This final definition of our grammar is not linked to any parsing algorithm, matching order or set of semantic actions. As such, it is about as close as it gets to the formal definition of the grammar in Section 3.2.1.

Our semantic processors are algebra's over the pattern functor. In fact, our type synonym *Processor* is identical to multirec's *Algebra* as defined by Rodriguez et al. [197]. Note also that syntactically, they look remarkably similar to

syntax-directed definitions traditionally used with parser generators (see [3, pp. 303–323]):

$$
\begin{array}{l}
\textbf{type } \mathit{Processor}\ \phi\ r = \forall ix.\phi\ ix \to \mathit{PF}\ \phi\ r\ ix \to r\ ix \\
\mathit{calc}_{arith} :: \mathit{Processor}\ \phi_{arith}\ [\![]\!]^{value}_{.} \\
\mathit{calc}_{arith}\ \mathit{Line}\ \ (\mathit{SExprF}\ [\![e]\!]^{value}_{Expr}) = [\![e]\!]^{value}_{Line} \\
\mathit{calc}_{arith}\ \mathit{Expr}\ \ (\mathit{SumF}\ [\![e]\!]^{value}_{Expr}\ [\![t]\!]^{value}_{Term}) = [\![e + t]\!]^{value}_{Expr} \\
\mathit{calc}_{arith}\ \mathit{Expr}\ \ (\mathit{STermF}\ [\![t]\!]^{value}_{Term}) = [\![t]\!]^{value}_{Expr} \\
\mathit{calc}_{arith}\ \mathit{Term}\ \ (\mathit{ProductF}\ [\![e]\!]^{value}_{Term}\ [\![t]\!]^{value}_{Factor}) = [\![e * t]\!]^{value}_{Term} \\
\mathit{calc}_{arith}\ \mathit{Term}\ \ (\mathit{SFactorF}\ [\![t]\!]^{value}_{Factor}) = [\![t]\!]^{value}_{Term} \\
\mathit{calc}_{arith}\ \mathit{Factor}\ (\mathit{ParenF}\ [\![e]\!]^{value}_{Expr}) = [\![e]\!]^{value}_{Factor} \\
\mathit{calc}_{arith}\ \mathit{Factor}\ (\mathit{NumberF}\ ds) = \\
\qquad\qquad [\![\mathit{read}\ (\mathit{map}\ (\lambda [\![d]\!]^{value}_{Decimal} \to d)\ ds)]\!]^{value}_{Factor} \\
\mathit{calc}_{arith}\ \mathit{Digit}\ \ (\mathit{MkDigitF}\ c) = [\![c]\!]^{value}_{Decimal}
\end{array}
$$

This processor implements the direct calculation of *Integer* values for subexpressions that we have previously described. Its type expresses that it is a processor for domain ϕ_{arith}, producing semantic values of family $[\![]\!]^{value}_{.}$. Like in traditional parser combinator libraries, a semantic processor can also produce side effects, simply by working with monadic calculations as semantic values instead of simple values.

Another example of a semantic processor, for which we do not need to provide any code, has been defined by Rodriguez et al. [197]. Their function $to :: \phi\ ix \to \mathit{PF}\ \phi\ I_*\ ix \to ix$ in the *Fam* type class transforms a single-level unfolding of an AST (as described earlier) back into the traditional AST data type. Serendipitously, composing *to* with the I_* constructor yields a ready-to-use and important semantic processor for our grammars. The function $(I_* \circ) \circ to$ (applying I_* to the result of applying *to* to two arguments) is precisely the semantic processor that produces a wrapped version of the AST as its semantic value. This direct correspondence illustrates that our use of multirec pattern functors to abstract semantic actions is a natural and powerful fit.

A processor and a grammar can be combined using the following function. It takes an extended context-free grammar for domain ϕ, and a processor for domain ϕ and semantic value family r and turns it into an extended context-free grammar which produces values of semantic value family r.

$$
\begin{array}{l}
\textbf{type } \mathit{ProcessingExtendedCFG}\ \phi\ r = \\
\quad \forall p\ ix.\mathit{LoopProductionRule}\ p\ \phi\ r \Rightarrow \phi\ ix \to p\ (r\ ix) \\
\mathit{applyProcessor} :: \mathit{Processor}\ \phi\ r \to \\
\qquad\qquad \mathit{ExtendedCFG}\ \phi \to \mathit{ProcessingExtendedCFG}\ \phi\ r \\
\mathit{applyProcessor}\ proc\ g\ idx = proc\ idx\ \$\$\ g\ idx
\end{array}
$$

Some of the algorithms we define in Section 3.5 will be able to work on grammars of types *ExtendedCFG and ProcessingExtendedCFG*. It is therefore useful to define a more general type of grammars as follows:

$$\begin{aligned}&\textbf{type } \mathit{GeneralExtendedCFG}\ \phi\ r\ rr = \\ &\quad \forall p\ ix.\mathit{LoopProductionRule}\ p\ \phi\ r \Rightarrow \phi\ ix \to p\ (rr\ ix)\end{aligned}$$

Note that $\mathit{ProcessingExtendedCFG}\ \phi\ r$ and $\mathit{GeneralExtendedCFG}\ \phi\ r\ r$ designate the same type and $\mathit{ExtendedCFG}\ \phi$ can also be written as $\forall r.\mathit{GeneralExtendedCFG}\ \phi\ r\ (\mathit{PF}\ \phi\ r)$. For non-extended and regular CFGs, we introduce analogous type synonyms:

$$\begin{aligned}&\textbf{type } \mathit{GeneralCFG}\ \phi\ r\ rr = \forall p\ ix.\mathit{RecProductionRule}\ p\ \phi\ r \Rightarrow \\ &\qquad\qquad\qquad\qquad \phi\ ix \to p\ (rr\ ix) \\ &\textbf{type } \mathit{ProcessingCFG}\ \phi\ r = \mathit{GeneralCFG}\ \phi\ r\ r \\ &\textbf{type } \mathit{CFG}\ \phi = \forall r.\mathit{GeneralCFG}\ \phi\ r\ (\mathit{PF}\ \phi\ r)\end{aligned}$$

There are ways to abstract this even further to remove the duplication between the extended and non-extended type synonyms. We do not go into that here, but they can be studied in our online library.

3.4.6 Grammar ingredients

In summary, our approach requires the grammar author to provide five things.

1. The standard AST data types from Section 3.3.4 (the types *Line*, *Expr* etc. for our example).
2. The domain "subkind" with the witness constructors as in Section 3.4.1 (ϕ_{arith} for our example), defining the collection of non-terminals for the grammar. Various grammar algorithms require extra information about the domain, which needs to be provided through instances of the type classes *ShowFam*, *FoldFam* and *EqFam* that we will encounter further on.
3. The pattern functor from Section 3.4.4 (PF_{arith} for our example), defining the recursive structure of the relations between the non-terminals, and the corresponding instance of the multirec *PF* type family. As discussed in Section 3.4.4, it can also be useful to implement multirec's *Fam* type class which defines the link between the domain, the pattern functor and the AST types.
4. A grammar for the domain (g_{arith} in our example), defining the concrete syntactic structure. Various algorithms allow the programmer to analyse and/or transform the grammar. Multiple grammars can even be defined for the same domain.

5. If the programmer wants to create a parser, he probably also requires one or more semantic processors as defined in Section 3.4.4 (e.g. $calc_{arith}$ for our example). These define how to combine parsed non-terminals to a value needed. Standard processors exist (e.g. a constant processor which leads to a recognizer for the grammar or the AST constructing processor $(I_* \circ) \circ to$ we encountered in Section 3.4.4).

However, of these five, the second and third consist of boilerplate code, which could be mechanically derived from the definition of the AST data types. In fact, the multirec library provides Template Haskell functions which mechanize this translation. The concepts we defined in addition to multirec (like the instances for the *ShowFam*, *FoldFam* and *EqFam* type classes) could be generated in a similar way.

Finally, we note again that we do not use Rodriguez et al.'s [197] type functor combinators to define the pattern functor. These combinators allow them to derive certain generic operations over it, reducing the amount of boilerplate code required. We avoid them for presentation reasons: we find they make pattern functor and semantic processor definitions more difficult to read and we do not need the automatically derived generic operations.

3.5 The proof of the pudding

Carette et al. show how a finally tagless encoding allows them to interpret a DSL for a simple higher order typed object language in different ways [30]. They demonstrate an evaluator, a compiler, a partial evaluator and call-by-name and call-by-value continuation-passing style transforms. In Sections 3.3 and 3.4, we have extended their approach with a model of recursion in the object language such that it is observable in the meta-language.

We will now demonstrate that we can define different interpretations for the recursive constructs. In fact, these interpretations will work similarly to Carette et al.'s different interpretations of object language primitives: a suitable production rule interpretation type is defined, and the behaviour of primitive parsing and recursion constructs supported by the algorithm is defined in the instances of the *Applicative*, *Alternative*, *CharProductionRule*, *RecProductionRule* and/or *LoopProductionRule* type classes. Transformations are possible using a production rule interpretation parametric in an abstract underlying interpretation type p. In this section, we demonstrate this approach with a couple of such algorithms, both analyses and transformations.

The algorithms we discuss will have varying requirements on the grammars they work for (and for transformations: the grammars they produce), either for fundamental reasons (e.g. *foldLoops* and *transformLeftCorner* will be defined only for *processing* grammars and cannot straightforwardly be extended to abstract grammars) as for reasons of conciseness (e.g. *isReachable* and *foldReachable* will be defined for normal grammars only but can trivially be generalised to *extended* context free grammars).

3.5.1 Pretty-printing grammars

A first grammar algorithm that requires a custom interpretation of recursion is pretty-printing. The implementation is not terribly difficult but it is instructive as a first demonstration of how to work with our recursion model. Furthermore, as a first test bed, it will also motivate some further infrastructure we need to put in place. This algorithm is a simplified version of the one in our `grammar-combinators` library.

To compute textual representations, we use a custom production rule interpretation type *PrintRuleInterp*, containing simply a *String* representation of the rule. It needs to carry the domain type ϕ and semantic value family r along in its type for a technical reason related to the functional dependencies of the production rule interpretation type classes.

$$\textbf{newtype}\ \mathit{PrintRuleInterp}\ (\phi :: * \to *)\ (r :: * \to *)\ v = \mathit{MkPRI}\ \{\mathit{printRule} :: \mathit{String}\}$$

We implement the *ProductionRule* operations by simply constructing a proper *String* representation of the rule. Note that this is in fact the first time in this chapter that we provide instances for these classes.

```
instance Applicative (PrintRuleInterp φ r) where
  pure _      = MkPRI "pure"
  a ⊛ b       = MkPRI (printRule a ++ " " ++ printRule b)
instance Alternative (PrintRuleInterp φ r) where
  empty       = MkPRI "empty"
  a ⓛ b       = MkPRI ("(" ++ printRule a ++ " | " ++ printRule b ++ ")")
instance CharProductionRule (PrintRuleInterp φ r) where
  endOfInput = MkPRI "endOfInput"
  token t     = MkPRI (show t)
```

For the *RecProductionRule* instance, we need to know how to represent a non-terminal as a *String*. We therefore require our domain ϕ to be an instance of a new type class called *ShowFam*, telling us how to convert a domain witness into a *String*.

```
class ShowFam φ where showIdx :: φ ix → String
instance ShowFam φ ⇒
  RecProductionRule (PrintRuleInterp φ r) φ r where
  ⟨idx⟩ = MkPRI ("<" ++ showIdx idx ++ ">")
instance ShowFam φ ⇒
  LoopProductionRule (PrintRuleInterp φ r) φ r where
  ⟨idx⟩* = MkPRI ("<" ++ showIdx idx ++ ">*")
  ⟨idx⟩+ = MkPRI ("<" ++ showIdx idx ++ ">+")
```

Given this interpretation for production rules, we can define how to print the production rules for a single non-terminal:

```
printNT :: ShowFam φ ⇒ GeneralExtendedCFG φ r rr → φ ix → String
printNT gram idx =
  "<" ++ showIdx idx ++ ">" ++ "  ::=  " ++ printRule (gram idx)
```

This *printNT* function takes a grammar, a non-terminal witness, and produces a string representation of the grammar's production rules for that non-terminal. Note that it takes our most general form of grammar *GeneralExtendedCFG*.

To print a full grammar, all that is left to do, is to consecutively apply this *printNT* function to all non-terminals in a grammar. To do this, we again need information from the domain, and we define this as another general requirement for domains in the *FoldFam* type class. Since we can't require that there exist a list of all non-terminals (because their witnesses all have a different type), the *FoldFam* class contains a function *foldFam*, which folds a given function over all non-terminals in the domain.

```
class FoldFam (φ :: ∗ → ∗) where foldFam :: (∀ix.φ ix → b → b) → b → b
printGrammar :: (FoldFam φ, ShowFam φ) ⇒
  GeneralExtendedCFG φ r rr → String
printGrammar gram = foldFam ((++) ∘ (++"\n") ∘ printNT gram) ""
```

One might have the impression that we are defining ad hoc *XFam* classes for all of our algorithms, but this impression would be false. The type classes *FoldFam* and *ShowFam* (and *EqFam* which we will encounter further on) express general requirements for domains. Only for presentation purposes, we have chosen

```
ghci> putStr (printGrammar g_arith)

<Line>   ::= <Expr> EOI
<Expr>   ::= <Term> | (<Expr> '+' <Term>)
<Term>   ::= <Factor> | (<Term> '*' <Factor>)
<Factor> ::= <Digit>+ | ('(' <Expr> ')')
<Digit>  ::= '0' | '1' | '2' | '3' | ... | '8' | '9'
```

Figure 3.4: Printing out an (E)BNF-like representation of the arithmetic expressions grammar with the library grammar printing algorithm (result manually indented).

to define them when we first encountered the need. The instances for our domain ϕ_{arith} are trivial:

$$\begin{aligned}
&\textbf{instance } \mathit{ShowFam}\ \phi_{arith} \textbf{ where } \mathit{showIdx}\ \mathit{Line} &&= \texttt{"Line"}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{showIdx}\ \mathit{Expr} &&= \texttt{"Expr"}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{showIdx}\ \mathit{Term} &&= \texttt{"Term"}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{showIdx}\ \mathit{Factor} &&= \texttt{"Factor"}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{showIdx}\ \mathit{Digit} &&= \texttt{"Digit"}\\
&\textbf{instance } \mathit{FoldFam}\ \phi_{arith} \textbf{ where}\\
&\quad \mathit{foldFam}\ f = f\ \mathit{Line} \circ f\ \mathit{Expr} \circ f\ \mathit{Term} \circ f\ \mathit{Factor} \circ f\ \mathit{Digit}
\end{aligned}$$

A more polished version of this algorithm produces output as given in Figure 3.4.

3.5.2 Open recursion

In Section 3.3.2, our first attempt at a better representation of recursion used a form of open recursion, different from the $\langle\cdot\rangle$ construct which we introduced later. It is in fact easy to formalize the equivalence between these two representations of grammars. In this section, we define a function called *openRecursion* that will turn out to be a useful technical aid in the implementation of other algorithms. It has the following type signature:

$$\begin{aligned}
&\mathit{openRecursion} :: \mathit{CharProductionRule}\ p \Rightarrow\\
&\quad \mathit{GeneralCFG}\ \phi\ r\ rr \rightarrow (\forall ix.\phi\ ix \rightarrow p\ (r\ ix)) \rightarrow \phi\ ix \rightarrow p\ (rr\ ix)
\end{aligned}$$

This function turns a grammar using the *RecProductionRule* type class' $\langle\cdot\rangle$ construct into a grammar taking a *self* parameter instead. To implement this,

we define a production rule type *ORRule* which wraps a production rule taking a *self* parameter.

```
newtype ORRuleInterp p φ r v =
  MkORR { unORR :: (∀ix.φ ix → p (r ix)) → p v }
```

We omit instances for classes *Applicative*, *Alternative* and *CharProductionRule* for this type. They simply pass through the *self* parameter to their components (if any). The *RecProductionRule* instance replaces calls $\langle idx \rangle$ with calls to *self idx*.

```
instance CharProductionRule p ⇒
  RecProductionRule (ORRuleInterp p φ r) φ r where
  ⟨idx⟩ = MkORR (λself → self idx)
```

In the implementation of *openRecursion*, we construct the production rule for non-terminal *idx* in the new grammar by interpreting the grammar with our *ORRuleInterp* production rule type and unwrapping the result.

```
openRecursion g self idx = unORR (g idx) self
```

Note, by the way, that the reverse transformation is even easier to define:

```
closeRecursion :: RecProductionRule p φ r ⇒
  (∀p. CharProductionRule p ⇒
    (∀ix.φ ix → p (r ix)) → φ ix → p (rr ix)) →
  φ ix → p (rr ix)
closeRecursion g idx = g ⟨·⟩ idx
```

3.5.3 Reachability

The previous transformation is sometimes a useful technical tool in the implementation of other algorithms. In this section, we implement a simple non-terminal reachability analysis. We perform a depth-first search while keeping track of an environment of non-terminals already encountered, which we represent as a function from non-terminals to *Bool*s. The environment *nothingSeen* represents the empty set:

```
newtype SeenEnv φ = MkSG { seenIdx :: ∀ix.φ ix → Bool }
nothingSeen :: SeenEnv φ
nothingSeen = MkSG (\_ → False)
```

To mark a non-terminal as seen in an environment, we need to be able to override the wrapped function for a single non-terminal and leave it unmodified for the others. In fact, overriding polymorphic functions in this way is another general requirement on domains, which we model in the *EqFam* type class:

```
class EqFam φ where
  overrideIdx :: (∀ix.φ ix → r ix) → φ oix → r oix → (∀ix.φ ix → r ix)
```

This type class models a general notion of domains with a decidable equality between non-terminals. However, unlike a simpler equality test (like the derived *eqIdx* below), the *overrideIdx* function allows us to exploit this decidable equality to override a polymorphic function over a domain ϕ for one of the non-terminals ϕ *oix*.

We need to instantiate the *EqFam* type class for all of our domains:

```
instance EqFam φ_arith where overrideIdx _ Line v Line = v
                             overrideIdx _ Expr v Expr = v
                             overrideIdx _ Term v Term = v
                             overrideIdx _ Factor v Factor = v
                             overrideIdx _ Digit v Digit = v
                             overrideIdx f _ _ idx = f idx
```

Using the general *overrideIdx* function, we can define a specialisation *overrideIdxK* for functions returning values of a constant type. We use a standard constant type functor K_* from multirec. We can also use it to define equality of non-terminal witnesses.

```
overrideIdxK :: EqFam φ ⇒ (∀ix'.φ ix' → v) → φ oix → v → φ ix → v
overrideIdxK f idx v = unK_* ∘ overrideIdx (K_* ∘ f) idx (K_* v)
eqIdx :: EqFam φ ⇒ φ ix1 → φ ix2 → Bool
eqIdx idx1 = overrideIdxK (const False) idx1 True
```

With this additional infrastructure, we can update our sets of non-terminals as follows:

```
setSeen :: EqFam φ ⇒ φ ix → SeenEnv φ → SeenEnv φ
setSeen idx s = MkSG (overrideIdxK (seenIdx s) idx True)
```

We implement our reachability analysis as a *foldReachable* function. Like the *foldFam* we've seen before, this function folds a function over a set of non-terminals. However, unlike that function, it does not fold the function over all non-terminals in the domain. The folding is restricted to the non-terminals reachable from a given start non-terminal in a given grammar:

```
type Folder φ n = ∀ix.φ ix → n → n
foldReachable :: ∀φ ix n r rr.EqFam φ ⇒
  GeneralCFG φ r rr → φ ix → Folder φ n → n → n
```

The implementation of this function uses an interpretation type wrapping an algorithm with the set of encountered non-terminals *SeenEnv* ϕ as a mutable state variable. The wrapped algorithm takes the function to be folded and the start value and its return type is the result type of the fold.

```
newtype FoldReachableRuleInterp φ n v = MkFRRI {
  foldRule :: Folder φ n → n → State (SeenEnv φ) n}
putSeen :: EqFam φ ⇒ φ ix → State (SeenEnv φ) ()
putSeen idx = modify (setSeen idx)
```

The algorithm is simple. For leaf rules in the grammar, the algorithm doesn't need to do anything, and for branch nodes, it simply iterates over subnodes. We omit the instances for *Applicative*, *Alternative* and *CharProductionRule*, which simply translate all operations into the following *foldLeaf* or *foldBranch* as appropriate:

```
foldLeaf :: FoldReachableRuleInterp φ n v
foldLeaf = MkFRRI (λ_ n → return n)
foldBranch :: FoldReachableRuleInterp φ n v →
  FoldReachableRuleInterp φ n v' → FoldReachableRuleInterp φ n v''
foldBranch ra rb = MkFRRI (λf n → do n' ← foldRule ra f n
                                     foldRule rb f n')
```

The only magic of the algorithm is in the handling of references to non-terminals. For a reference to a non-terminal *idx*, we need to check if we have encountered the non-terminal *idx* already and if so, terminate the recursive search. If not, we fold the fold function over the non-terminal and subsequently recurse over the production rules of this non-terminal:

```
foldRef :: EqFam φ ⇒ φ ix → FoldReachableRuleInterp φ n v →
           FoldReachableRuleInterp φ n v'
foldRef idx r = MkFRRI (λf n →
  do seen ← gets (λseenSet → seenIdx seenSet idx)
     if seen then return n else putSeen idx ≫ foldRule r f (f idx n))
```

In order to define an instance of the type class *RecProductionRule* for our *FoldReachableRuleInterp* using *foldRef*, we need to modify that type to carry the rules for the entire grammar along. By using the previously introduced

openRecursion function as a technical aid, we avoid a bit of this verbiage. The *self* parameter we provide to that algorithm is constructed using the *foldRef* function. We then evaluate the fold in the start non-terminal's production rule with an initially empty set of seen non-terminals:

$$
\begin{aligned}
&\mathit{foldReachable}\ g\ \mathit{idx}\ f\ n = \\
&\quad \textbf{let}\ g' :: \forall \mathit{ix}.\phi\ \mathit{ix} \to \mathit{FoldReachableRuleInterp}\ \phi\ n\ (\mathit{rr}\ \mathit{ix}) \\
&\qquad g' = \mathit{openRecursion}\ g\ (\lambda \mathit{idx} \to \mathit{foldRef}\ \mathit{idx}\ (g'\ \mathit{idx})) \\
&\quad \textbf{in}\ \mathit{evalState}\ (\mathit{foldRule}\ (g'\ \mathit{idx})\ f\ n)\ \mathit{nothingSeen}
\end{aligned}
$$

Checking whether a non-terminal *end* is reachable from a non-terminal *start* is easy to implement in terms of *foldReachable*:

$$
\begin{aligned}
&\mathit{isReachable} :: \forall \phi\ r\ \mathit{rr}\ \mathit{ix}\ \mathit{ix}'.\mathit{EqFam}\ \phi \Rightarrow \\
&\quad \mathit{GeneralCFG}\ \phi\ r\ \mathit{rr} \to \phi\ \mathit{ix} \to \phi\ \mathit{ix}' \to \mathit{Bool} \\
&\mathit{isReachable}\ g\ \mathit{start}\ \mathit{end} = \mathit{foldReachable}\ g\ \mathit{start}\ ((\vee) \circ \mathit{eqIdx}\ \mathit{end})\ \mathit{False}
\end{aligned}
$$

3.5.4 Production rule origami

In Section 3.3.3, we introduced the $\langle\cdot\rangle^*$ and $\langle\cdot\rangle^+$ operators, and defined their types in Section 3.4.4. Formally, these operators are very similar to the other recursive operator $\langle\cdot\rangle$, and our modelling of them in the *LoopProductionRule* type class allows us to define different interpretations, like for the $\langle\cdot\rangle$ operator.

From a parsing point of view, these operators are less fundamental than $\langle\cdot\rangle$. In this section, we implement a standard transformation, known as the recursive interpretation of *regular right part* grammars (see Grune and Jacobs [88, §2.3.2.4]), which transforms grammars using $\langle\cdot\rangle^*$ and $\langle\cdot\rangle^+$ into grammars which only use $\langle\cdot\rangle$.

The grammar transformation works by replacing calls to $\langle \mathit{idx} \rangle^*$ for non-terminal *idx* with normal references $\langle \mathit{idx}^* \rangle$ to newly introduced non-terminals idx^*. We define suitable production rules for these new non-terminals in the transformed grammar. This transformation is implemented in our library as the *foldLoops* algorithm.

In a first step, we define the domain of the transformed grammar:

$$
\begin{aligned}
&\textbf{data}\ \cdot^1\ \mathit{ix} \\
&\textbf{data}\ \cdot^*\ \mathit{ix} \\
&\textbf{data}\ \cdot_{fl}\ \phi\ \mathit{ix}\ \textbf{where}\ \cdot^1 :: \phi\ \mathit{ix} \to \phi_{fl}\ \mathit{ix}^1 \\
&\qquad\qquad\qquad\qquad\qquad\quad \cdot^* :: \phi\ \mathit{ix} \to \phi_{fl}\ \mathit{ix}^*
\end{aligned}
$$

We introduce new non-terminal types ix^1 and ix^*, parameterised over an underlying non-terminal type ix. The new non-terminal ix^1 represents the base non-terminal ix, and ix^* its quantified *-variant.[9] Both types ix^1 and ix^* are strictly needed on the type-level and they do not contain any values. The new domain ϕ_{fl}, parameterised over an underlying domain ϕ (the dot is a placeholder for ϕ in the definition), contains witnesses for ix^1 and ix^* given a proof that ix is a non-terminal in the underlying domain ϕ. We use the names $\cdot^1$ and $\cdot^*$ in Haskell's namespace for types and values as respectively the non-terminal types and the constructors of the witnesses.

All necessary type classes (like *FoldFam* and *ShowFam*, and others which we haven't encountered yet) can be implemented for this new domain. As an example, we show the *FoldFam* instance, which simply uses the underlying domain ϕ's *foldFam* function to iterate over both types of non-terminals in the domain ϕ_{fl}:

$$\begin{aligned}&\textbf{instance } \mathit{FoldFam}\ \phi \Rightarrow \mathit{FoldFam}\ \phi_{fl}\ \textbf{where}\\ &\quad \mathit{foldFam}\ f\ n = \mathit{foldFam}\ (\lambda idx \to f\ idx^*)\ (\mathit{foldFam}\ (\lambda idx \to f\ idx^1)\ n)\end{aligned}$$

For representing semantic values for the new domain, we introduce a semantic value family adapter r_{flv}, parameterised over an underlying semantic value family r. As you might expect, r_{flv} wraps a value of type $r\ ix$ for the new non-terminal ix^1 and a value of type $[r\ ix]$ for the non-terminal ix^*.

$$\begin{aligned}&\textbf{data family } \cdot_{flv}\ (r :: * \to *)\ ix\\ &\textbf{newtype instance } \cdot_{flv}\ r\ ix^1 = \mathit{FLBV}\ \{\mathit{unFLBV} :: r\ ix\}\\ &\textbf{newtype instance } \cdot_{flv}\ r\ ix^* = \mathit{FLMV}\ \{\mathit{unFLMV} :: [r\ ix]\}\\ &\mathit{consFLV} :: r_{flv}\ ix^1 \to r_{flv}\ ix^* \to r_{flv}\ ix^*\\ &\mathit{consFLV}\ (\mathit{FLBV}\ v)\ (\mathit{FLMV}\ vs) = \mathit{FLMV}\ (v : vs)\end{aligned}$$

In a second step, we define the *foldLoops* algorithm, which turns an extended context-free grammar over a domain ϕ into the equivalent non-extended context-free grammar over the larger domain ϕ_{fl}. The algorithm only supports grammars which have already been combined with a semantic processor, i.e. grammars of type *ProcessingExtendedCFG*. This leads to the following type signature:

$$\mathit{foldLoops} :: \mathit{ProcessingExtendedCFG}\ \phi\ r \to \mathit{ProcessingCFG}\ \phi_{fl}\ r_{flv}$$

The transformed grammar is defined by the production rules for both types of non-terminals in domain ϕ_{fl}. The production rules for a non-terminal idx^* are

[9] We do not require values for the ix^1 and ix^* types, so we define them using the EmptyDataTypes GHC Haskell extension.

straightforward. Such a non-terminal must either be the corresponding base non-terminal idx^1 followed by another instance of non-terminal idx^* itself, or it must be empty. In both cases, we make sure to produce the correct semantic value.

$$\begin{aligned} \mathit{foldLoops}\ \mathit{bgram}\ \mathit{idx}^* = {} & \mathit{consFLV} \mathbin{\$} \left\langle \mathit{idx}^1 \right\rangle \circledast \langle \mathit{idx}^* \rangle \\ & \oplus\ \mathit{pure}\ (\mathit{FLMV}\ [\,]) \end{aligned}$$

The production rules for a base non-terminal idx^1 are obtained by taking the production rules of the unmodified grammar and replacing all references to $\langle idx \rangle^*$ with calls to $\langle idx^* \rangle$. We perform this substitution by instantiating the original grammar's production rules with a special production rule interpretation type *FLW*. The type *FLW* implements a production rule for the original context-free grammar over domain ϕ, in terms of an underlying production rule for the transformed context-free grammar over the extended domain ϕ_{fl}. The classes *Applicative*, *Alternative* and *CharProductionRule* are implemented by just passing the call through to the underlying production rules and wrapping/unwrapping the results as appropriate (not shown for brevity). The *RecProductionRule* instance transforms a reference $\langle idx \rangle$ into a reference $\left\langle idx^1 \right\rangle$ and the *LoopProductionRule* instance transforms a quantified reference $\langle idx \rangle^*$ into the desired normal reference $\langle idx^* \rangle$. The default definition of $\langle \cdot \rangle^+$ transforms $\langle idx \rangle^+$ into $(:) \mathbin{\$} \langle idx \rangle \circledast \langle idx \rangle^*$, which is perfect for our purposes.

$$\begin{aligned} & \textbf{data}\ \mathit{FLRuleInterp}\ p\ (\phi :: * \to *)\ (r :: * \to *)\ v = \mathit{MkFLRI}\ \{\mathit{unFLRI} :: p\ v\} \\ & \textbf{instance}\ \mathit{RecProductionRule}\ p\ \phi_{fl}\ r_{flv} \Rightarrow \\ & \quad \mathit{RecProductionRule}\ (\mathit{FLRuleInterp}\ p\ \phi\ r)\ \phi\ r\ \textbf{where} \\ & \quad \langle \mathit{idx} \rangle = \mathit{MkFLRI}\ (\mathit{unFLBV} \mathbin{\$} \left\langle \mathit{idx}^1 \right\rangle) \\ & \textbf{instance}\ \mathit{RecProductionRule}\ p\ \phi_{fl}\ r_{flv} \Rightarrow \\ & \quad \mathit{LoopProductionRule}\ (\mathit{FLRuleInterp}\ p\ \phi\ r)\ \phi\ r\ \textbf{where} \\ & \quad \langle \mathit{idx} \rangle^* = \mathit{MkFLRI}\ (\mathit{unFLMV} \mathbin{\$} \langle \mathit{idx}^* \rangle) \end{aligned}$$

We can now finish our algorithm with the definition of the transformed grammar's production rules for non-terminals idx^1. These simply unwrap the *FLRuleInterp* production rule interpretation type:

$$\mathit{foldLoops}\ \mathit{bgram}\ \mathit{idx}^1 = \mathit{FLBV} \mathbin{\$} \mathit{unFLRI}\ (\mathit{bgram}\ \mathit{idx})$$

In Figure 3.5, we show the result of applying the foldLoops algorithm to the arithmetic expressions grammar. We omit the base rules which are identical to the ones in Figure 3.4.

```
ghci> putStr (printGrammar (foldLoops (applyProcessor calc_arith g_arith)))
```

```
<Line*> ::= (<Line> <Line*>) | pure
<Expr*> ::= (<Expr> <Expr*>) | pure
<Term*> ::= (<Term> <Term*>) | pure
<Factor*> ::= (<Factor> <Factor*>) | pure
<Digit*> ::= (<Digit> <Digit*>) | pure
...
<Factor> ::= (<Digit> <Digit*>) | '(' <Expr> ')'
...
```

Figure 3.5: A printed version of the added production rules for $\cdot^*$ non-terminals added by the *foldLoops* algorithm. We omit the $\cdot^1$ production rules which are identical to Figure 3.4.

3.5.5 The left-corner transform, declaratively...

As a final demonstration of a special interpretation of recursive constructs, we show by example that our framework allows the definition of non-trivial general grammar transformations. We develop an implementation of the left-corner transform as defined (among others) by Moore [157]. It removes left-recursion from a grammar, solving the problem we saw in Section 3.2.3.

Figure 3.6 partially shows the result of applying the left-corner transformation to the arithmetic expressions grammar. What happens is that for, for example, the *Expr* non-terminal, the transformation has analysed the set of terminals and the set of non-terminals that a match of *Expr* can possibly start with. The second set is called the set of *left corners* of *Expr*. New non-terminals define what remains of the *Expr* non-terminal after one of these terminals (e.g. *Expr* − `'('`) or non-terminals (e.g. *Expr* − *Factor*) has been matched. The new rules are not (directly or mutually) left-recursive but they define the same language as the original grammar.

In the literature, the left-corner transform is typically presented in an algorithmic style (e.g. [157, 16, 9]): an initial grammar is analysed, and step-by-step, new rules are added to obtain a final transformed grammar. We conjecture that such an implementation can be supported in our framework using techniques similar to Baars et al. [9]. They define transformation arrows to generate new type-level identifiers as well as keep track of a modifiable typing environment for non-terminal references.

```
ghci> putStr (printReachableGrammar (filterDiesE
        (transformLeftCornerE calcGrammarArith)) (LCBase Expr))
```

```
(...)
<Expr>         ::= ('(' <'('\Expr>) | ('0' <'0'\Expr>)
                 | ('1' <'1'\Expr>) | ('2' <'2'\Expr>) | ...
                 | ('7' <'7'\Expr>) | ('8' <'8'\Expr>)
                 | ('9' <'9'\Expr>)
<'('\Expr>     ::= <Expr> ')' <Factor\Expr>
<Factor\Expr>  ::= <Term\Expr>
<Term\Expr>    ::= (('*' <Factor>) <Term\Expr>) | <Expr\Expr> | pure
<Expr\Expr>    ::= '+' <Term> (<Expr\Expr> | pure) | (EOI <Line\Expr>)
<Line\Expr>    ::= empty
<'9'\Expr>     ::= <Digit\Expr>
<Digit\Expr>   ::= <Digit>* <Factor\Expr>
(...)
```

Figure 3.6: Some rules from the printed version of the arithmetic expressions grammar after applying the left-corner transform and dead-branch removal. Output reformatted, reordered and selected.

However, we prefer to give a more declarative account of the transformation. It turns out that by analysing the algorithmic description, we can identify the three different forms of production rules that will be generated, and the production rules for all three can be derived from the rules in the original grammar. For any given domain ϕ, we define a new domain ϕ_{lc}, containing three types of non-terminals: for given non-terminals a and b and terminal t, we have non-terminals a_1 (representing the base non-terminal a), $b \setminus_{NT} a$ (matching the remainder of non-terminal a when non-terminal b has already been matched) and $t \setminus_T a$ (matching the remainder of non-terminal a when character t has been matched). Note that the latter is related to Brzozowski's derivatives of regular expressions [27]. Note again the $\cdot$ as placeholder in the notations.

$$
\begin{aligned}
&\textbf{data}\ \cdot_1\ ix \\
&\textbf{data}\ (\cdot \setminus_{NT} \cdot)\ ix'\ ix \\
&\textbf{data}\ (\setminus_T \cdot)\ ix \\
&\textbf{data}\ \cdot_{lc}\ \phi\ ix\ \textbf{where}\ \cdot_1 \quad\ \ :: \phi\ ix \to \phi_{lc}\ ix_1 \\
&\qquad\qquad\qquad\qquad\ \ \cdot \setminus_{NT} \cdot\ :: \phi\ ix' \to \phi\ ix \to \phi_{lc}\ (ix' \setminus_{NT} ix) \\
&\qquad\qquad\qquad\qquad\ \ \cdot \setminus_T \cdot\ \ :: Char \to \phi\ ix \to \phi_{lc}\ (\setminus_T ix)
\end{aligned}
$$

For a semantic value family r for the underlying domain ϕ, we define a new semantic value family r_{lc} for our new domain ϕ_{lc}, with appropriate semantic values for the newly introduced non-terminals. For example, since a non-terminal $b \setminus_{NT} a$ represents the remainder of a non-terminal a starting with a non-terminal b that has already been parsed, we define the type of its semantic value as $r\ b \to r\ a$: a function that returns the semantic value of non-terminal a when given the value of the already parsed non-terminal b.

$$
\begin{array}{l}
\textbf{data family}\ \cdot_{lc}\ (r :: * \to *)\ ix \\
\textbf{newtype instance}\ r_{lc}\ ix_1 \qquad\qquad = LCV_1 \qquad \{\, unLCV_1 :: r\ ix \,\} \\
\textbf{newtype instance}\ r_{lc}\ (ix' \setminus_{NT} ix) = LCV_{\cdot \setminus_{NT} \cdot}\ \{\, unLCV_{\cdot \setminus_{NT} \cdot} :: r\ ix' \to r\ ix \,\} \\
\textbf{newtype instance}\ r_{lc}\ (\setminus_T ix) \qquad = LCV_{\cdot \setminus_T \cdot}\ \{\, unLCV_{\cdot \setminus_T \cdot} :: Char \to r\ ix \,\}
\end{array}
$$

In order to construct the production rules for these new non-terminals, we need to analyse the existing rules in the grammar. The information we need is collected in the four fields of production rule interpretation type *TLCRuleInterp*:

$$
\begin{array}{l}
\textbf{data}\ TLCRuleInterp\ p\ \phi\ r\ v = \\
\quad MkTLCRI\ \{\, tlcEmpty \quad\quad :: Maybe\ v, \\
\qquad\qquad\ \ tlcFull \quad\quad\ :: p\ v, \\
\qquad\qquad\ \ tlcNTMinNT :: \forall ix'.\phi\ ix' \to p\ (r\ ix' \to v), \\
\qquad\qquad\ \ tlcNTMinT \ :: Char \to p\ (Char \to v)\,\}
\end{array}
$$

The field *tlcEmpty* keeps track of whether the production rule can (directly) match the empty string and if so, what value it produces. Under *tlcFull*, we keep an unmodified version of the original production rule. Under *tlcNTMinNT*, we keep the original production rule with leading (direct) references to a given base non-terminal removed (or, in the absence of such a leading reference, a never-matching *empty* rule) and *tlcNTMinT* provides the original production rule with leading (direct) references to a given terminal removed.

We omit the instances for the *Applicative*, *Alternative* and *CharProductionRule* type classes for brevity. In the *Applicative* instance, we need to make sure to properly handle empty and non-empty left hand sides in the sequencing operator (to make sure we properly detect *leading* tokens and references). In the *CharProductionRule* instance, we interpret a call to *token tt* specially under the *tlcNTMinT* interpretation, replacing it with a *pure id* rule, that simply passes through the already matched token.

The *RecProductionRule* instance is the most interesting one. Under the *tlcNTMinNT* interpretation of the base production rule (where the current rule has to consume a given already matched non-terminal), we need to interpret a call to a base non-terminal $\langle idx \rangle$ as a pure rule that simply passes through the

already matched semantic value, but only if the already matched non-terminal is the requested non-terminal *idx*. Otherwise, the *tlcNTMinNT* interpretation must fail. To do this in a well typed way, we use the function *overrideIdx*, defined in Section 3.5.3:

$$\begin{aligned}&\mathit{overrideIdx} :: \mathit{EqFam}\ \phi \Rightarrow \\ &\quad (\forall \mathit{ix}.\phi\ \mathit{ix} \to r\ \mathit{ix}) \to \phi\ \mathit{oix} \to r\ \mathit{oix} \to (\forall \mathit{ix}.\phi\ \mathit{ix} \to r\ \mathit{ix})\end{aligned}$$

The *RecProductionRule* instance above defines the *tlcNTMinNT* interpretation of an underlying production rule as a function that will fail for all non-terminals except for the requested non-terminal, in which case it is an empty rule passing through the already matched result. A technical problem is that the *overrideIdx* function requires the result type of the overridden function to be directly parametric in the non-terminal type *ix*, requiring us to wrap and unwrap the returned rules in a wrapper type *WrapNTMinNTP*. The other interpretations are straightforward.

$$\begin{aligned}&\textbf{instance}\ (\mathit{RecProductionRule}\ p\ \phi_{lc}\ r_{lc}, \mathit{EqFam}\ \phi) \Rightarrow \\ &\quad \mathit{RecProductionRule}\ (\mathit{TLCRuleInterp}\ p\ \phi\ r)\ \phi\ r\ \textbf{where} \\ &\quad \langle \mathit{idx} :: \phi\ \mathit{ix} \rangle = \mathit{MkTLCRI}\ \{\mathit{tlcEmpty} = \mathit{Nothing}, \\ &\qquad\qquad\qquad\qquad \mathit{tlcFull} = \mathit{unLCV}_1\ \$\ \langle \mathit{idx}_1 \rangle, \\ &\qquad\qquad\qquad\qquad \mathit{tlcNTMinNT} = \mathit{rNTMinNT}, \\ &\qquad\qquad\qquad\qquad \mathit{tlcNTMinT} = \mathit{const}\ \mathit{empty}\} \\ &\qquad \textbf{where}\ \mathit{rNTMinNT} :: \forall \mathit{ix}'.\phi\ \mathit{ix}' \to p\ (r\ \mathit{ix}' \to r\ \mathit{ix}) \\ &\qquad\qquad \mathit{rNTMinNT}\ \mathit{idxm} = \mathit{unWNMNP} \\ &\qquad\qquad\quad (\mathit{overrideIdx}\ (\backslash\_ \to \mathit{WNMNP}\ \mathit{empty}) \\ &\qquad\qquad\qquad \mathit{idx}\ (\mathit{WNMNP}\ (\mathit{pure}\ \mathit{id}))\ \mathit{idxm}) \\ &\textbf{newtype}\ \mathit{WrapNTMinNTP}\ p\ r\ \mathit{ix}\ \mathit{ix}' = \\ &\quad \mathit{WNMNP}\ \{\mathit{unWNMNP} :: p\ (r\ \mathit{ix}' \to r\ \mathit{ix})\}\end{aligned}$$

With these instances, we have the machinery we need to analyse a grammar's production rules, and we can proceed to the actual transformation of the grammar in the function *transformLeftCorner*. This function is restricted to processing grammars because the left-corner transform inherently mixes transformed versions of rules from the original grammar and new rules of standard forms, making it difficult to work with non-processing grammars.

$$\begin{aligned}&\mathit{transformLeftCorner} :: \\ &\quad (\mathit{FoldFam}\ \phi, \mathit{EqFam}\ \phi) \Rightarrow \mathit{ProcessingCFG}\ \phi\ r \to \mathit{ProcessingCFG}\ \phi_{lc}\ r_{lc}\end{aligned}$$

To define the production rules of the transformed grammar, we need to know the FIRST sets of the non-terminals (see [3, pp.188–189]): the set of terminals that

a match of a given non-terminal can start with. To obtain this information, we make use of a general algorithm *calcFirst*, which performs the standard FIRST-set analysis. We omit its implementation, which is relatively straightforward (~70 LOC in the library). With this extra information, we call another function *transformLeftCorner′* which will generate the actual production rules for our new non-terminals.

```
transformLeftCorner gram idx =
  transformLeftCorner′ gram (calcFirst gram) idx
```

The production rules for non-terminals idx_1 are of the following form: they first expect to see one of the tokens of the FIRST set of the non-terminal *idx* and then pass on the work to the non-terminal $t \setminus_T idx$, properly wrapping and unwrapping values along the way:

```
transformLeftCorner′ bgram firstSet idx₁ =
  let ruleT tt = flip ($) <$> token tt <*> (unLCV_.\T. <$> ⟨tt \T idx⟩)
  in  LCV₁ <$> Set.fold ((<|>) ∘ ruleT) empty (firstSet idx)
```

Omitting the production rules for non-terminals $ix' \setminus_{NT} ix$ (which are technically similar to those that follow), all that is still required for the left-corner grammar transformation are the rules for non-terminals $t \setminus_T idx$. These rules come in two forms, because the non-terminal *idx* can start with character *t* in two ways. Either one of the original production rules for the non-terminal *idx* starts with character *t* directly, and in that case, the remainder of that production rule becomes the production rule for $t \setminus_T idx$. This remainder of the original production rule is precisely what is represented by its interpretation under *tlcNTMinT t*.

The other possibility is that a production rule of *idx* starts with a (direct or indirect) reference to another non-terminal *idxB*, and that non-terminal directly starts with character *t*. This is captured by a production rule for non-terminal $t \setminus_T idx$ that starts with the remainder of the production rules for non-terminal *idxB* starting with character *t* (which we again get using that production rule's interpretation under *tlcNTMinT*) and then references non-terminal $idxB \setminus_{NT} idx$. Because non-terminal $idxB \setminus_{NT} idx$ represents the remainder of a base non-terminal *idx* after a non-terminal *idxB* has been matched, its production rules will properly match the remainder of non-terminal *idx*.

```
transformLeftCorner′ bgram _ (t \T (idx :: ϕ ix1)) =
  let bMinT :: ϕ ix2 → p (Char → r ix1)
      bMinT idxB = flip (∘) <$> tlcNTMinT (bgram idxB) t <*>
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad (unLCV_{.\setminus_{NT}.} \mathbin{\$} \langle idxB \setminus_{NT} idx \rangle) \\
\quad bMinTs = foldFam\ ((\oplus) \circ bMinT)\ empty \\
\textbf{in} \quad LCV_{.\setminus_{T}.} \mathbin{\$} bMinTs \\
\quad \oplus\ LCV_{.\setminus_{T}.} \mathbin{\$} tlcNTMinT\ (bgram\ idx)\ t
\end{array}
$$

Note that we don't actually check whether character t is in the FIRST set of non-terminal *idxB*, nor that *idxB* is a left corner of *idx*. These would both be worthwhile optimisations, but they are not necessary, because in those cases, subsequent parts of the production rule in question become *empty* rules and can be removed using general postprocessing algorithms (dead-branch removal and dead non-terminal unfolding).

3.5.6 The grammar-combinators library

The above algorithms show that our grammar model adds useful expressiveness to parser combinator libraries in a finally tagless style: we can do more grammar analyses and transformations. The limitation of parser combinator libraries to top-down parsing algorithms is also lifted: after the left-corner transform, a top-down matching order in the transformed grammar corresponds to a left-corner matching order for the original grammar [198]. Our semantic processors can be applied during parsing, independently of the matching order.

In addition to what we've shown, we have implemented an elaborate grammar analysis, transformation and parsing library called `grammar-combinators`. This library is designed as a collection of independently usable grammar algorithms. The library provides a combination of various features that, to the best of our knowledge, are unavailable in any existing parser EDSL library.

Practical features include a powerful transformation library (including the left-corner transform and a uniform version of Paull's left-recursion removal (see [3, p. 177]), support for performing grammar transformations at compile time using Template Haskell [208]), a generic packrat parser [73] and basic interfaces to uu-parsinglib [219] and Parsec [125] as *backend parsers*. The library is open source and available online.

3.5.7 Limitations

Notwithstanding its advantages, the typing of recursive constructs entails a certain overhead in defining concepts such as the domain, its witnesses and pattern functor, and semantic value families, when compared to standard parser combinators (see Section 3.4.6). On top of that, some limitations need to be taken into account.

Our recursive constructs are clearly more verbose than the almost trivial recursion in typical parser combinator libraries. However, we believe that a certain verbosity is unavoidable if we wish to support a wide range of standard algorithms from the parsing literature, many of which require observable recursion. Also, supporting additional recursive constructs requires quite a bit of work. In this chapter, we can see that supporting quantified recursive constructs on top of normal recursive constructs (which is again almost trivial in normal parser combinators) required an extra type class, a translation algorithm involving a model of the modified domain and semantic value families etc.

A compelling feature of parser combinators that we have not looked at, is the ease with which you can combine unrelated parsers into new ones. An example is the definition of grammar patterns like typical comment styles or standard number notations. We require a full view of grammars, and this makes us lose some of the simple compositionality of parser combinators. We are experimenting with a grammar combination primitive that partly recovers this, but it is not ready for inclusion in the library.

Another limitation is that the added abstraction unfortunately has a performance cost. In some initial tests, we have effectively noticed an important performance impact, even though general optimizations for generic code [133] appear to reduce it considerably. The performance impact could also be reduced by performing grammar transformations at compile-time using Template Haskell. A more detailed performance analysis remains future work, but we expect that compiler improvements are needed (like better inlining heuristics and more control over partial evaluation) to improve performance of generic code in general and our code in particular.

3.6 Related work

For background material on context-free grammars, parsing and grammar transformations, we refer to Aho et al. [3].

3.6.1 Finally tagless DSLs

The finally tagless style for modelling a typed object language in a meta-language was identified and popularised by Carette, Kiselyov and Shan [30]. They demonstrate a model of a higher-order typed lambda calculus in a typed functional metalanguage (they use both Haskell and ML). Their model is parameterised by an interpretation of the primitive operations of their object

language. It uses metalanguage typing to statically ensure type-correctness of the modeled object language terms. Carette et al. demonstrate a set of different interpretations of their lambda calculus variant: an evaluator, a staged interpreter, a partial evaluator, and call-by-name and call-by-value continuation passing style transformations.

In this text, we have described why the standard *fix* operator is not a perfect fit for our requirement for metalanguage-observable recursion in our parsing DSL and we have defined an alternative model. It is interesting however, that for any grammar AST, we can also consider the AST as a representation type for terms of a separate embedded typed object language, representing the semantics of the grammar. In Carette et al.'s terminology, the standard AST types from Section 3.4 is an "initial" embedding of this language, but we could have used a "finally tagless" model for it as well. In this model, the *Sum* constructor would, for example, correspond to a *sum* function in a grammar-specific *ArithSemantics* type class. Such a finally tagless encoding is more extensible than the naive AST representation, but our representation using the multirec pattern functor actually features this extensibility as well; because the pattern functor is parametric in the representation of recursive sub-data, we can apply the same technique as Swierstra [221]. In this sense, the pattern functor offers an alternative "initial" modelling, less naive than the standard GADT representation, and lacking its inextensibility. Additionally, it offers some benefits of its own that seem unavailable in a finally tagless style (e.g. it supports generic algorithms using multirec [197]). A more detailed study of this correspondence is interesting future work.

3.6.2 Parser combinators

Parser combinators have a long history (see [125] for references), but most work employs an ω-regular representation of grammars, with the associated downsides that we have discussed in Section 3.1. Here, we limit ourselves to work that uses a representation of grammars in which recursion is observable. Even then, almost all libraries are tied to a single parsing algorithm.

TTTAS

Baars et al. [9, 8] implement the left-corner grammar transform [157] using type-level naturals as the representation of non-terminals. They ensure type-safety using a type environment encoded as a list of types. They propose a transformation library based on the arrows abstraction, which they use essentially for the generation of fresh type-level identifiers. Like ours, their

grammar representation explicitly represents the grammar's recursion in a well-typed way and allows them to implement the left-corner transform and support left-recursive grammars.

Nevertheless, we believe our work provides advantages over theirs. Our representation of non-terminals as a "subkind with proof terms" [197] and type environments as data families is less complex. We provide semantic value family polymorphism, which they do not. They use stateful *Trafo* transformation arrows to allow for generation of fresh non-terminal identifiers. This is more powerful than the way we introduce new non-terminals (e.g. in Section 3.5.5), since algorithms in our approach need to define statically (potentially in terms of an argument domain) the non-terminals that will be used, while their algorithms can decide at run-time how many non-terminals are needed. This allows them to implement the standard, imperative-style descriptions of grammar transformations and imperatively extend domains step by step during the transformation. Our algorithms work with *fixed* domains, which we found beneficial in the sense that it has forced us to formulate the algorithms in a more functional style. However, there may be algorithms that do not lend themselves well to such a reformulation (although we have not encountered them in the parsing domain), in which case more complex techniques like Baars and Swierstra's extensible domains are required.

Finally, Baars and Swierstra's grammars seem designed to be generated by the compiler in Viera et al.'s alternative for the standard Haskell *read*-function [227]. They are less easily human-readable than our grammars. In the parsing domain, Baars and Swierstra only discuss an implementation of the left-corner grammar transform, while we show the importance of our approach for a wider parsing library, discussing implementations of a variety of useful algorithms for grammar analysis, transformation and parsing.

Dependently typed parser combinators

Brink, Holdermans and Löh describe a dependently typed parser combinator library [24], implemented in the Agda programming language [161]. Agda's dependently typed nature strongly simplifies the requirements on the representation of non-terminals (types of production rules can more simply depend on non-terminals). They implement the left-corner transformation in their formalism, and they provide a machine-checkable proof of a language-inclusion property for the transformation.

The proof of correctness properties beyond type-safety is out of range in our Haskell implementation. In addition to making such proofs possible, the power of dependent types also lets the authors get away with very simple models of

grammars (a list of production rules) and production rules (a LHS non-terminal and a list of RHS symbols). Types are simply recalculated from these simple models when needed instead of going through a lot of trouble to model them and carry them around. Our use of Haskell limits us to a more restricted formalism, but this does make our ideas more portable and our approach more disciplined. Our use of a finally tagless model allows us to define different sets of primitives that can be mixed and matched (keeping, for example, extended CFGs separate from normal CFGs), whereas Brink et al. restrict themselves to standard CFGs.

Danielsson and Norell [53] use Agda to define a provably terminating parser combinator library of *total parser combinators*. They use unobservable (co-)recursion,[10] limiting them to a top-down parser algorithm. They manage to support left-recursion (although their approach does not seem suited for online parsing) with an algorithm based on Brzozowski derivatives and they provide a static termination guarantee using dependent types and a mixture of induction and coinduction. It is interesting that in an unpublished draft, seemingly a pre-cursor of their total parser combinators, Danielsson and Norell investigate a model of grammars with observable recursion, using an operator _! similar to our $\langle\cdot\rangle$. They discuss it as one of two alternative modellings of grammars which solves certain technical modularity problems of a more standard parser combinator model. Brink et al. authors do not discuss the fact that their "grammar-based" model makes recursion observable or the additional power this provides to the model.

3.6.3 Observable recursion

In order to model and work with recursive structures in a pure language like Haskell, several approaches have been explored in the literature. One branch of research has focused on introducing a varying amount of impurity, ranging from observing sharing within the *IO* monad [79] to adding referential identity as a fundamental language feature [40]. We do not go into these approaches in detail, as it is our goal to model the recursion in the parsing EDSL with a representation that is observable in Haskell without compromising purity. Much of this research focuses on the application domain of hardware description languages and we would be interested to see if our approach can be successfully applied in this field as well.

Carette et al. [30] provide a form of observable recursion through the *fix* primitive which we have discussed in detail in Section 3.3, so we do not go into that further. Another interesting proposal is the recursive do notation as proposed

[10] What we've been calling recursion throughout this paper is actually corecursion in their terminology.

by Erkok and Launchbury [70], who add a primitive recursive operator for monads in a type class *MonadRec* (later renamed to *MonadFix*):

$$\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadRec}\ m\ \textbf{where}$$
$$\quad \mathit{mfix} :: (\alpha \to m\ \alpha) \to m\ \alpha$$

Instances are supposed to obey three laws (strictness, purity and left-shrinking). Analogous to the translation of Haskell's **do**-notation to pure code involving the monadic operators, Erkök and Launchbury define a recursive **mdo**-notation and a translation to pure code involving monadic operators *and* the *mfix* primitive. Erkök and Launchbury's proposal could be used to provide observable recursion in a monadic parser EDSL, but unfortunately, a monadic parser EDSL is more difficult to analyse for other reasons (see e.g. Swierstra and Duponcheel [220, §5.2]). As discussed in Section 3.3, we believe that the *FixProductionRule* type class and its *fix* method (based on the *fix* method defined by Carette et al. [30]) are analogous to *MonadRec* and *mfix* for applicative functors and we think it is interesting future work to extend the bracket notation for applicative code by McBride and Paterson [145] with a notation for recursion.

The **do**-notation for arrows by Paterson [178] also translates recursion to a recursion primitive in the *ArrowLoop* class, which seems to support observable recursion. Allowing for the (limited-depth) LL(1) analysis performed by uu-parsinglib [220] was a motivation for the development of arrows [102], so together with the observable recursion primitive provided by the *ArrowLoop* type class and the recursive **do**-notation, arrows may allow for the more elaborate kinds of grammar analysis and transformation that we perform, but we are not aware of any work that investigates this in more detail.

3.7 Acknowledgements

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the Research Foundation - Flanders (FWO), and by the Research Fund K.U.Leuven. Dominique Devriese holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO). We thank Arie Middelkoop, Tom Schrijvers, Adriaan Moors, Doaitse Swierstra, Nils Anders Danielsson and anonymous reviewers for their comments.

Chapter 4

Typed Syntactic Meta-programming

Publication data

Dominique Devriese and Frank Piessens. Typed syntactic meta-programming. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, September 2013.

Abstract

We present a novel set of meta-programming primitives for use in a dependently-typed functional language. The types of our meta-programs provide strong and precise guarantees about their termination, correctness and completeness. Our system supports type-safe construction and analysis of terms, types and typing contexts. Unlike alternative approaches, they are written in the same style as normal programs and use the language's standard functional computational model. We formalise the new meta-programming primitives, implement them as an extension of Agda, and provide evidence of usefulness by means of two compelling applications in the fields of datatype-generic programming and proof tactics.

4.1 Introduction

Meta-programming means writing programs that write or manipulate other programs. It is an important software engineering technique that is widely used in practice. The term covers a wide variety of techniques and applications, including parser generators [109], reflection and byte-code generation in Java-like languages [173, 25], macro's in Lisp-like languages [224], eval primitives in languages like JavaScript [196], special-purpose meta-programming or generic programming primitives [34, 208, 222, 132, 36, 86, 20], tactics in proof assistants [56, 214, 215] and term representations in advanced type systems [72, 48, 32, 144]. Meta-programming jargon distinguishes between the *meta-language*, which meta-programs are written in, and the *object language*, which the programs being manipulated are in.

Meta-programming can often be used to implement features in a library that would otherwise require ad hoc compiler support. This ranges from meta-programs that generate small amounts of boilerplate code to give libraries a more native feel (e.g. [120, 137, 197]) to languages built from the ground up using meta-programming [224].

In many applications, meta-programs must not only be able to produce new code but also analyse existing terms, types or type contexts. Applications in, for example, datatype-generic programming or tactics for proof assistants involve meta-programs that analyse the syntactic structure of object language data types [132, 36], types [86, 214], types and contexts [56]. Some systems allow analysing terms [34], terms and types [208] or all three [215, 48, 32, 144].

Type-safety in the context of meta-programming can mean different things. In some approaches, generated code is type-checked upon completion of the meta-program, either at compile-time or run-time [56, 208, 25]. This can be sufficient to guarantee type-correctness of the resulting program. In this text, we are interested in a stronger form of type-safety, in which a meta-program's type can guarantee type-correctness of all programs it will ever generate (e.g. [222, 132, 34, 214, 215, 86]). This stronger form of type-safety provides meta-program authors and users with greater correctness assurance. Sometimes, it also enables additional applications. For example, MetaML runs meta-programs and compiles the generated code at run-time, but type errors during this run-time compilation are ruled out by its strong type-safety [222]. In the context of a dependently-typed proof assistant, where proofs and programs are equated, Chlipala argues that the stronger form of type-safety has a performance advantage because proofs generated by meta-programs do not need to be calculated as long as they can be trusted to exist [37]. Note that for this last application, the meta-program must be guaranteed to terminate, as well as produce well-typed code.

However, this stronger form of type-safety puts a high demand on representations of object code and the meta-language type system, especially for object languages with strong (e.g. dependent) type systems and if meta-programs can construct and analyse both terms, types and typing contexts. Most approaches use an explicit syntactic representation of object language terms and/or types. To achieve strong type safety, they employ advanced type-system features of the meta-language, including GADTs [222, 34], strong type systems with powerful type-level languages [72, 36, 214, 215] and an advanced feature of dependent type systems called induction-recursion [48, 32, 144]. However, even the most powerful approaches have to make certain compromises, simplifying the resulting system at the cost of expressivity. For example, many approaches provide syntactic models of only types [132] or only terms [222, 34] or types and terms but not typing contexts [214].

Particularly technically ambitious are those meta-programming systems that use a syntactic model of a dependently-typed language within another one [48, 32, 144]. Their term encoding represents type-correctness *internally*, i.e. only well-typed terms are represented. To support this, they require an advanced dependent type system with support for induction-recursion in the meta-language and even then have trouble fitting the interpretation function (which translates encoded object terms to the values they represent) in it [48, 32]. McBride presents a model that is accepted by Agda but has to significantly limit the dependent nature of the object language's type system in the process [144]. The objective of these meta-programming systems is generally to prove meta-theory for the object language and the authors work hard to fit their encodings into the advanced, but general and previously studied schema of inductive-recursive definitions.

However, beside their meta-theoretical value, syntactic models of a typed object language with a well-typed interpretation function are also promising for meta-programming applications. Unfortunately, the full potential of this has not been explored or demonstrated so far because researchers have not yet managed to build syntactic models of dependently-typed programming languages that support a big enough subset of a dependently-typed language and still have provably sound interpretation functions. In this paper, we ignore the aim of building meta-theory for dependent type theories within themselves and instead focus on applying such techniques to meta-programming. We will explain this in more detail further on and show that this approach has some very compelling qualities.

We use Agda [161], a pure functional dependently-typed language, as both the meta- and object language and we start from a conventional representation of the object language based on de Bruijn-encoded lambda terms and an external typing judgement. *We make a set of interpretation functions available as*

new meta-programming primitives. This puts us on shakier ground, because the soundness of the primitive is not guaranteed by existing meta-theory, but it allows us to side-step the unsolved problem of syntactically representing a dependent type theory within itself with a provably sound interpretation function. As such, we gain the ability to explore and demonstrate our approach's potential for meta-programming and present novel techniques for it.

Our choice to keep the meta- and object language the same (known as *homogeneous* meta-programming [208, 222]) contrasts with systems where meta-programs use a different computational model than object programs. Often this is an imperative model [56, 214, 215, 208], but some systems even use a logic programming-like model derived from the meta-programs' interaction with type inference [132, 86]. Our meta-programs use the same functional model as normal programs and dependent pattern matching [82] for syntactically analysing terms, types and typing contexts. This choice keeps the system smaller, makes techniques, tools and knowledge for normal programming directly reusable in meta-programs and it allows meta-programs to use other meta-programs to do their work. It does not exclude imperative, generally recursive, non-deterministic or unification-based reasoning in meta-programs. Research has demonstrated functional models of such algorithms [28, 50, 119, 124] and such ideas could be combined with our work.

In the dependently typed meta-language, meta-programs have strong and precise types that guarantee termination and correctness. Termination is standard for Agda functions (Agda is total). For strong type safety, our primitives require meta-programs to provide type-correctness proofs together with generated code and they can exploit type-correctness proofs for the code they analyse.

Some homogeneous meta-programming systems couple meta-programming with *multi-stage programming* [222, 20, 208], which allows object code programs to explicitly invoke meta-programs and use the generated expressions as if they were hand-written (*splicing* or *evaluation*) and allows meta-programs to include references to existing terms in generated code (*quoting*, *cross-stage persistence*). A linear hierarchy of staging levels exists when meta-programs may unquote expressions generated by other meta-programs. The bottom stage is the program executed at run-time, while other stages execute at compile-time or run-time, depending on the system. Our set of interpretation functions for encoded terms is analogous to an evaluation primitive and we will demonstrate how object-level terms can be referenced in generated code. The question of when meta-programs are executed becomes a matter of choice and a special case of partial evaluation.

We demonstrate the properties of our system by applying it to two important application domains: *datatype-generic programming* and *proof tactics*. For the

first, we define a syntactic representation *SimpleDT* of inductive data types that can be used to write general datatype-generic meta-programs. As proof-of-concept, we present a meta-program *deriveShow* that syntactically derives a serialisation function $show : A \to String$ for a data type A.

$$deriveShow : (A : Set) \to SimpleDT\ A \to A \to String$$

SimpleDT and *deriveShow* do not require compiler support beyond our (general) meta-programming primitives, although the value of type $SimpleDT\ A$ could be provided by the compiler for additional convenience. The type of *deriveShow* guarantees its correct termination and well-typedness of generated programs (modulo the primitives' soundness). To the best of our knowledge, this is the first demonstration of strongly typed, general datatype-generic meta-programs, with support for syntactic analysis of terms and types and using the language's standard computational model.

The second application domain is proof tactics. A tactic is a meta-program that analyses the type of a proof obligation and produces a proof term (possibly including remaining proof obligations) using general or domain-specific reasoning. Several proof assistants provide special-purpose languages for writing custom tactics [56, 214, 215]. These are often imperative and only guarantee weak type-safety (generated code is checked after execution of meta-program) or partial strong type-safety (generated code is guaranteed type-correct but meta-programs may not terminate). Gonthier et al. argue that tactics without strong type-safety can be hard to maintain and compose [86]. Chlipala discusses a performance advantage of precisely-typed and terminating meta-programs since generated proofs do not need to be calculated if they are known to exist [37]. We present an account of Coq's *assumption* tactic with a very precise type, guaranteeing that it will always terminate and produce a guaranteed type-correct term under a precise condition. The tactic uses a functional computational model and dependent pattern matching for syntactic analysis of terms, types and typing contexts.

We have implemented our primitives in Agda and our example meta-programs are accepted by Agda's type-checker.[1] Unfortunately, this does not mean our work is readily usable. The practicality of our implementation is currently hampered by long compilation times. However, we will argue that this problem is not intrinsic, but caused by the inefficient evaluation strategy of Agda's compile-time evaluator. *The soundness of our approach depends on the soundness of our primitives, which we can currently not provide guarantees about.* We believe that our work gives a strong motivation to investigate both of these aspects further, since we provide strong evidence for the additional power that the system offers

[1] Code available on `http://people.cs.kuleuven.be/dominique.devriese/permanent/tsmp.zip`.

for meta-programming in general and the hard problems of datatype-generic programming and well-typed tactics in particular.

4.1.1 Contributions

Our first contribution in this work is the definition of novel meta-programming primitives in a dependently-typed language, starting from a partial formalisation of the language's meta-theory. We also contribute the (to our knowledge first) demonstration of using such a formalisation for meta-programming, with compelling examples in two important application domains: datatype-generic programming and proof tactics. Our meta-programming model works with the language's standard functional computational model, and meta-programs are written in the same way as normal programs. Modulo the soundness of our primitives, meta-programs can be given strong and precise guarantees of termination and correctness of the generated code. Finally, our proof-of-concept applications in these two application domains are interesting in their own right. For both datatype-generic programming and proof tactics, the prospect of writing general meta-programs with strong and precise guarantees about termination, correctness and completeness and using the language's standard computational model is compelling and novel.

4.1.2 Outlook

We present the representation of our object language in Section 4.2. In Section 4.3, we show how the represented terms and types are brought to life in the meta-language using our meta-programming primitives. In Section 4.4, we present applications to the fields of datatype-generic programming and proof tactics. We discuss issues like soundness and performance in Section 4.5, related work in Section 4.6 and we conclude in Section 4.7.

4.2 Self-representation

As discussed, we start from a representation of Agda terms in Agda using a notion of lambda expression representing terms as well as their types and a typing judgement linking terms and types together.

Terms Figure 4.1 shows the definition of *Expr*, our representation of Agda terms and types as lambda terms, using de Bruijn indices. We represent

```
data Constant : (arity : ℕ) → Set where  (empty for now)
data Binder : Set where Π : Binder
                        Λ : Binder
data Expr (n : ℕ) : Set where
  set      : Expr n
  var      : Fin n → Expr n
  appl     : Expr n → Expr n → Expr n
  constant : {arity : ℕ} → Constant arity → Vec (Expr n) arity → Expr n
  bind     : Binder → Expr n → Expr (suc n) → Expr n
pi     : {n : ℕ} → Expr n → Expr (suc n) → Expr n
pi     = bind Π
lambda : {n : ℕ} → Expr n → Expr (suc n) → Expr n
lambda = bind Λ
```

Figure 4.1: The representation of terms.

de Bruijn indices as integers between 0 and $n - 1$ using the Agda standard library type *Fin n* [51]. The type *Expr* is parameterised by the number of free variables in scope. It is defined as a standard inductive data type [66], with an enumeration of its constructors and their types. The *set* constructor represents the type of types in the object language and free variables are embedded through *var*. There is a standard function application constructor *appl* and constants applied to a fixed number of arguments (as determined by the constant's *arity*) through term constructor *constant*. *Vec A n* is another Agda standard library type representing a vector of precisely n values of type A. In what follows, we use [] for the empty vector and, for example, $[x, y]$ for the vector with elements x and y. Similarly, we write literal *Fin*s as numbers.

The final *Expr* constructor in Figure 4.1, *bind*, is a common representation of two separate binding constructs: lambda expressions $\lambda(x : T) \to b$[2] and dependent function types $(x : T) \to T'$, constructed as *bind Λ* and *bind Π* respectively. They take two arguments: the type T of the bound variable and the body of the construct (b or T' respectively) with the bound variable additionally in scope in the body. Note by the way that a standard non-dependent function type $s \to t$ can be represented as dependent function $(\_ : s) \to t$. Finally, note in the type of *constant*, *pi* and *lambda* that some arguments are bound using curly brackets, indicating that they can be omitted in calls. Agda will then infer their value from the types of the remaining arguments.

[2] We use Agda notation for lambdas $\lambda(x : T) \to b$, not the more standard $\lambda x : T.b$.

$$
\begin{array}{ll}
\mathit{Sub} & : \mathbb{N} \to \mathbb{N} \to \mathit{Set} \\
\_\,/\,\_ & : \{m\ n : \mathbb{N}\} \to \mathit{Expr}\ m \to \mathit{Sub}\ m\ n \to \mathit{Expr}\ n \\
\mathit{weaken} & : \{n : \mathbb{N}\} \to \mathit{Expr}\ n \to \mathit{Expr}\ (\mathit{suc}\ n) \\
\_\,[\,\_\,] & : \{n : \mathbb{N}\} \to \mathit{Expr}\ (\mathit{suc}\ n) \to \mathit{Expr}\ n \to \mathit{Expr}\ n
\end{array}
$$

Figure 4.2: Substitutions (implementations omitted).

$$
\begin{array}{l}
\textbf{data}\ \_\rightsquigarrow_0\_\ \{n\} : \mathit{Expr}\ n \to \mathit{Expr}\ n \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{reduceApplication} : \forall\ \{s\}\ \ b\ \mathit{val} \to \quad \mathit{appl}\ (\mathit{lambda}\ s\ b)\ \mathit{val}\ \rightsquigarrow_0\ \ b\,[\,\mathit{val}\,] \\
\textbf{data}\ \_\rightsquigarrow\_\ \{n\} : \mathit{Expr}\ n \to \mathit{Expr}\ n \to \mathit{Set}\ \textbf{where} \\
\quad \cdots\ \text{(congruence closure of } \_\rightsquigarrow_0\_\text{)} \\
\_\rightsquigarrow_*\_\ :\ \{n : \mathbb{N}\} \to \mathit{Expr}\ n \to \mathit{Expr}\ n \to \mathit{Set} \\
\_\rightsquigarrow_*\_ = \cdots\ \text{(transitive-reflexive closure of } \_\rightsquigarrow\_\text{)} \\
\_\approx\_\ :\ \{n : \mathbb{N}\} \to \mathit{Expr}\ n \to \mathit{Expr}\ n \to \mathit{Set} \\
x \approx y\ \ = \exists\,(\lambda\ z \to x\ \rightsquigarrow_*\ z \times y\ \rightsquigarrow_*\ z)
\end{array}
$$

Figure 4.3: Full β-reduction and β-equivalence for untyped terms.

Substitutions We use a library of substitutions that is part of the Agda standard library [51], based on a technique by McBride [143]. Figure 4.2 shows a type of substitutions *Sub m n* that will substitute terms with n free variables for all m free variables of other terms. More concretely, the function $\_/\_$ applies a substitution ϕ of type *Sub m n* to a term t typed *Expr m* to obtain term $t\,/\,\phi$, typed *Expr n*. Note that, for example, $\_/\_$ is Agda notation for a *mixfix* operator that is applied to two arguments t and ϕ in the form $t\,/\,\phi$ [52]. The function *weaken* uses the substitution infrastructure to increase free de Bruijn indices by one and $\_\,[\,\_\,]$ substitutes term v for de Bruijn variable 0 in term t, to obtain term $t\,[v]$, shifting other free de Bruijn indices downward in the process.

Convertibility The next thing we define is an untyped notion of strong β-reduction and β-equivalence of terms in Figure 4.3. It is technically convenient to define primitive reductions in judgement $\_\rightsquigarrow_0\_$, a congruence closure of it in $\_\rightsquigarrow\_$ and a transitive-reflexive closure of that in $\_\rightsquigarrow_*\_$. The *reduceApplication* rule uses the substitution function $\_\,[\,\_\,]$ we saw before. In the type of *reduceApplication* we use Agda's $\forall$ shorthand notation, which desugars to a normal dependent type. For example, $\forall\ \{n\} \to \cdots$ or $\forall\ n \to \cdots$ is short for $\{n : \_\} \to \cdots$ and $(n : \_) \to \cdots$ respectively, i.e. an implicit or normal argument

```
data Telescope (i : ℕ) : ℕ → Set where
  ε       : Telescope i i
  _◁_ : {n : ℕ} →  Expr n  →  Telescope i n → Telescope i (suc n)
Context : (n : ℕ) → Set
Context = Telescope 0
lookup : ∀ {n} → Fin n → Context n → Expr n
lookup zero    (t ◁ _)  = weaken t
lookup (suc n) (_ ◁ Γ)  = weaken (lookup n Γ)
```

Figure 4.4: Telescopes and Contexts

n whose type is inferred by Agda. One $\forall$ symbol can apply to more than one argument. In $\_ \approx \_$, we use the $\exists$ and $\times$ types: for a type A and predicate P typed $A \to Set$, $\exists P$ represents a dependent sum type containing tuples (v, pv) with v of type A and pv of type $P\ v$. For types A and B, $A \times B$ represents the cartesian product type of A and B (containing (a, b) with a of type A and b of type B). Two terms t and t' are defined to be convertible ($t \approx t'$) iff there exists a third term n that both t and t' reduce to.

Typing Contexts Figure 4.4 contains the definition of typing contexts and the more general notion of telescopes. A telescope is a sequence of expressions, each representing the type of a bound variable. The entries may refer to a number of free variables, assumed to be bound outside the telescope. The first index i of the *Telescope* type indicates how many such *initial* variables are assumed. Telescopes are dependent: subsequent types can mention variables bound earlier in the telescope. This allows us to represent e.g. the telescope $(n : \mathbb{N})\ (t : Expr\ n)$, where the type of t depends on the value of n. As a consequence of this dependence, each additional entry in a telescope has an additional variable in scope. The second index n of the *Telescope* type is the number of *final* variables: if i variables are initially bound, and we add the bindings of a *Telescope* i n, then in total n variables will be bound, so the telescope contains precisely $n - i$ entries. A typing context *Context* n is a telescope with zero initial and n final bound variables. The *lookup* function looks up the type of a variable in a context. *lookup*'s dependent type ensures that only de Bruijn variables lower than the length of the context can be looked up.

$$
\begin{aligned}
&\textbf{data}\ \_\vdash\_:\_\ \ \{n\}\ (\Gamma : \mathit{Context}\ n) : \mathit{Expr}\ n \to \mathit{Expr}\ n \to \mathit{Set}\ \textbf{where}\\
&\quad \mathit{typeSet} : \Gamma \vdash \mathit{set} : \mathit{set}\\
&\quad \mathit{typeVar} : \forall\ \{i\} \to\ \ \Gamma \vdash \mathit{var}\ i : \mathit{lookup}\ i\ \Gamma\\
&\quad \mathit{typePi} : \forall\ \{s\ t\} \to\ \ \Gamma \vdash s : \mathit{set}\ \ \to\ \ (s \lhd\ \Gamma) \vdash t : \mathit{set}\ \ \to\ \ \Gamma \vdash \mathit{pi}\ s\ t : \mathit{set}\\
&\quad \mathit{typeLam} : \forall\ \{s\ b\ t\} \to\\
&\qquad \Gamma \vdash s : \mathit{set}\ \ \to\ \ (s \lhd\ \Gamma) \vdash b : t\ \ \to\ \ \Gamma \vdash \mathit{lambda}\ s\ b : \mathit{pi}\ s\ t\\
&\quad \mathit{typeAppl} : \forall\ \{s\ f\ t\ \mathit{val}\} \to\ \ (s \lhd\ \Gamma) \vdash t : \mathit{set}\ \ \to\\
&\qquad \Gamma \vdash f : \mathit{pi}\ s\ t\ \ \to\ \ \Gamma \vdash \mathit{val} : s\ \ \to\ \ \Gamma \vdash \mathit{appl}\ f\ \mathit{val} : \mathit{appl}\ (\mathit{lambda}\ s\ t)\ \mathit{val}\\
&\quad \mathit{typeConv} : \forall\ \{e\ t\ t'\} \to\ \ t \approx t'\ \ \to\ \ \Gamma \vdash e : t'\ \ \to\ \ \Gamma \vdash t : \mathit{set}\ \ \to\ \ \Gamma \vdash e : t
\end{aligned}
$$

Figure 4.5: Typing Judgements.

Typing Judgements In Figure 4.5, we show the typing judgement $\Gamma \vdash v : t$ stating that term v has type t in typing context Γ. The typing judgement models a fairly standard dependent type system, except for the first rule *typeSet*. This rule expresses that *set* has type *set* in any context, a rule which is known as type-in-type and a known source of paradox in dependent type theories [103]. However, we use this rule only for ease of presentation. Our full code avoids type-in-type using a predicative hierarchy of universes similar to Agda's [161]. It uses a level-indexed set_l, the typing rule that $\mathit{set}_l : \mathit{set}_{\mathit{suc}\ l}$ for all l, and a level-indexed typing judgement $\Gamma \vdash_l v : t$ with l such that $\Gamma \vdash_{\mathit{suc}\ l} t : \mathit{set}\ l$ must hold.

In the remaining typing rules in Figure 4.5 we have *typeVar*, stating that the type of a variable is given by the corresponding entry in the typing context and *typePi*, stating that $(x : S) \to T$ is a type if S and T are types, with $x : S$ added to the context for T. For lambda expressions, *typeLam* says that $\lambda\ (x : S) \to b$ is typed $(x : S) \to T$ if b has type T in a context extended with $x : S$. According to *typeAppl*, a function application $f\ \mathit{val}$ has type $((\lambda\ (x : S) \to T)\ \mathit{val})$ if f has type $(x : S) \to T$ and val has type S. Note that we could equivalently have given such an application the type $T\ [\mathit{val}]$. Finally, the rule *typeConv* states that a type t can be substituted for a convertible type t' in any typing judgement.

In the full version of our code, we extend the calculus with built-in dependent sum types (like the $\exists$ type we have already seen), identity types $x \equiv y : A$ (which contain proofs that x and y of type A are definitionally equal) and the empty type $\bot$ (which does not contain any value). These are modelled by adding suitable constructors for the types, their constructors and eliminators to the *Constant* data type, together with appropriate typing and reduction rules.

```
data ⊢_ : ∀ {n} → Context n → Set where
  ty_ϵ : ⊢ ε
  ty_◁ : ∀ {n e} {Γ : Context n} → ⊢Γ → Γ ⊢ e : set → ⊢(e ◁ Γ)
data _⊢_ : _ {n} (Γ : Context n) :
  {m : ℕ} (ρ : Sub m n) (tel : Context m) → Set where
  ··· (omitted)
```

Figure 4.6: Well-typed Contexts

More typing judgements In addition to the typing judgement for terms above, we also define typing judgements for contexts and for substitutions. Figure 4.6 shows the judgement $\vdash \Gamma$ expressing that context Γ is well-typed, i.e. that all context entries are sets. Its rule ty_ϵ states that the empty context is always well-typed and $ty_\triangleleft$ says that subsequent entries should be types in their preceding context. We omit the definition of judgement $\Gamma \vdash \phi : tel$,[3] which expresses that the terms substituted by substitution ϕ satisfy the type requirements of telescope tel in context Γ.

Meta-theory and helper functions We have proved quite some meta-theory about the reduction, convertibility and typing judgements. For full detail we refer to the full version of our code, but to give you an idea of what is there, Figure 4.7 shows the types of the most important results. $weaken - inj- \approx$ shows that weakening is injective with respect to convertibility. $\approx$*–trans* shows that convertibility is transitive. $\approx -trans$ is a consequence of the Church-Rosser-property for our reduction rules, which we have proved using a technique for untyped lambda calculi by Tait, described by Martin-Löf [139]. Theorem $\approx$–/ states that convertibility is invariant under substitutions. Theorems *weakenJudgementTop*, *substJudgementTop*, *substContext* and $\vdash -/$ state roughly that typings are preserved under weakening, instantiating a variable in the context, replacing a type in the context by a convertible one and applying a substitution to term and type. $\vdash$*–var* is a simple proof that entries in a well-typed context must be sets. By theorem *typesAreSets*, the type of a judgement in a well-typed context must in fact be a type. Finally, *substJudgementType* is not a theorem but a simple helper function that replaces a judgement's type by a provably equal type (it is a special case of *subst*, the standard eliminator of Agda's singleton type $t \equiv t'$).

[3] For ease of presentation, we overload the notation $\_\vdash\_ : \_$ in this text.

$$
\begin{aligned}
&\mathit{weaken{-}inj{-}}{\approx} : \forall\ \{n\}\ \{x\ y : \mathit{Expr}\ n\} \to \quad \mathit{weaken}\ x \approx \mathit{weaken}\ y \quad \to \quad x \approx y \\
&{\approx}\mathit{{-}trans} : \forall\ \{n\}\ \{x\ y\ z : \mathit{Expr}\ n\} \to \quad x \approx y \quad \to \quad y \approx z \quad \to \quad x \approx z \\
&{\approx}{-}/ : \forall\ \{n\}\ \{x\ y\}\ \{m\}\ (\rho : \mathit{Sub}\ \mathit{Expr}\ n\ m) \to \quad x \approx y \quad \to \quad x \,/\, \rho \approx y \,/\, \rho \\
&\mathit{weakenJudgementTop} : \forall\ \{n\}\ \{\Gamma : \mathit{Context}\ n\}\ \{v\ t\ t'\} \to \\
&\qquad \Gamma \vdash v : t' \quad \to \quad t \lhd\ \Gamma \vdash \mathit{weaken}\ v : \mathit{weaken}\ t' \\
&\mathit{substJudgementTop} : \forall\ \{n\}\ \{\Gamma : \mathit{Context}\ n\}\ \{t'\ e\ t\ v\} \to \\
&\qquad t' \lhd\ \Gamma \vdash e : t \quad \to \quad \Gamma \vdash v : t' \quad \to \quad \Gamma \vdash e\,[v] : t\,[v] \\
&\mathit{substContext} : \forall\ \{n\}\ \{\Gamma : \mathit{Context}\ n\}\ \{e\ t\}\ \{t'\ t''\} \to \\
&\qquad t' \approx t'' \quad \to \quad \Gamma \vdash t' : \mathit{set} \quad \to \quad t' \lhd\ \Gamma \vdash e : t \quad \to \quad t'' \lhd\ \Gamma \vdash e : t \\
&\vdash -/ : \forall\ \{m\ n\}\ \{e\ t\}\ \Gamma_1\ \Gamma_2 \to \quad (\phi : \mathit{Sub}\ m\ n) \quad \to \\
&\qquad \Gamma_2 \vdash \phi : \Gamma_1 \quad \to \quad \Gamma_1 \vdash e : t \quad \to \quad \Gamma_2 \vdash e \,/\, \phi : t \,/\, \phi \\
&\vdash \mathit{{-}var} : \forall\ \{n\}\ \{\Gamma : \mathit{Context}\ n\} \to \\
&\qquad \vdash \Gamma \quad \to \quad (i : \mathit{Fin}\ n) \quad \to \quad \Gamma \vdash \mathit{lookup}\ i\ \Gamma : \mathit{set} \\
&\mathit{typesAreSets} : \forall\ \{n\}\ \{\Gamma : \mathit{Context}\ n\}\ \{e\ t\}\ \{l\} \to \\
&\qquad \vdash \Gamma \quad \to \quad \Gamma \vdash e : t \quad \to \quad \Gamma \vdash t : \mathit{set} \\
&\mathit{substJudgementType} : \forall\ \{n\}\ \{\Gamma : \mathit{Context}\ n\}\ \{e\ t\ t'\} \to \\
&\qquad t \equiv t' \quad \to \quad \Gamma \vdash e : t \quad \to \quad \Gamma \vdash e : t'
\end{aligned}
$$

Figure 4.7: Meta-theoretic properties of our typing judgements.

Some example terms Let us consider the encoding of a simple example term: the following polymorphic identity function:

$$
\begin{aligned}
&\mathit{id} \ : \ \forall\ (A : \mathit{Set}) \to A \to A \\
&\mathit{id} = \lambda\ (A : \mathit{Set}) \to \lambda\ (v : A) \to v
\end{aligned}
$$

The definition and type of this function are given by closed expressions *idTm* and *idTyTm*.

$$
\begin{aligned}
&\mathit{idTm} \quad \ : \ \mathit{Expr}\ 0 \\
&\mathit{idTm} \quad \ = \mathit{lambda}\ \mathit{set}\ (\mathit{lambda}\ (\mathit{var}\ 0)\ (\mathit{var}\ 0)) \\
&\mathit{idTyTm} \ : \ \mathit{Expr}\ 0 \\
&\mathit{idTyTm} = \mathit{pi}\ \mathit{set}\ (\mathit{fun}\ (\mathit{var}\ 0)\ (\mathit{var}\ 0))
\end{aligned}
$$

We can prove that the term *idTm* satisfies type *idTyTm* using the typing rules from Figure 4.5.

$$
\begin{aligned}
&\mathit{ty}_{\mathit{idTm}} \ : \ \varepsilon \vdash \mathit{idTm} : \mathit{idTyTm} \\
&\mathit{ty}_{\mathit{idTm}} = \mathit{typeLam}\ \mathit{typeSet}\ (\mathit{typeLam}\ \mathit{typeVar}\ \mathit{typeVar})
\end{aligned}
$$

By the *typesAreSets* theorem, it follows that *idTyTm* is a type.

$$ty_{idTyTm} \ : \ \varepsilon \vdash idTyTm : set$$
$$ty_{idTyTm} = typesAreSets\ ty_{\epsilon}\ ty_{idTm}$$

4.3 Bringing terms to life

With this infrastructure in place, we can define our meta-programming primitive *interp* together with auxiliary primitives *interpCtx* and *interpSet*. Their types are:

$$interpCtx : \{n : \mathbb{N}\}\ \{\Gamma : Context\ n\} \to \vdash \Gamma \ \to \ Set$$
$$interpSet : \{n : \mathbb{N}\}\ \{\Gamma : Context\ n\}\ \{A : Expr\ n\} \to$$
$$\Gamma \vdash A : set \ \to \ (ty_{\Gamma} : \vdash \Gamma) \ \to \ interpCtx\ ty_{\Gamma} \ \to Set$$
$$interp : \{n : \mathbb{N}\}\ \{\Gamma : Context\ n\}\ \{v\ t : Expr\ n\} \to$$
$$(ty_v : \Gamma \vdash v : t) \ \to \ (ty_{\Gamma} : \vdash \Gamma) \ \to \ (asmpts : interpCtx\ ty_{\Gamma}) \ \to$$
$$interpSet\ (typesAreSets\ ty_{\Gamma}\ ty_v)\ ty_{\Gamma}\ asmpts$$

interpCtx turns the types in a well-typed context into a dependent sum type of the context entries' interpretations. It is used by the two other judgements to require values for all of a context's assumptions. *interpSet* interprets an encoded type, yielding a *Set*, and *interp* interprets a term v typed t. In the result type of *interp* for a proof ty_v of judgement $\Gamma \vdash v : t$, we use the previously mentioned theorem *typesAreSets* to calculate $typesAreSets\ ty_{\Gamma}\ ty_v$, a proof that $\Gamma \vdash t : set$. The result of *interp* is then of type t, interpreted using *interpSet* and this derived judgement.

Interpreting examples Before we go into more details, consider again the previously encoded polymorphic identity function. Remember that the closed terms *idTm* and *idTyTm* encode the function and its type and the proofs ty_{idTm} and ty_{idTyTm} witness the typing judgements $\varepsilon \vdash idTm : idTyTm$ and $\varepsilon \vdash idTyTm : set$. Both proofs assume only an empty context, which is always well typed according to the rule ty_{ϵ} in Figure 4.6. We will discuss the reduction behaviour of our primitives further, but $interpCtx\ ty_{\epsilon}$ (the assumptions in the empty context) reduces to unit type $\top$ (with canonical inhabitant *tt*). With all of this, we can interpret the encoded type *idTyTm* to obtain the type $intrp_{idTyTm}$:

$$intrp_{idTyTm} = interpSet\ ty_{idTyTm}\ ty_{\epsilon}\ tt$$

More details follow, but $intrp_{idTyTm}$ reduces to $(x : Set)\ (x_1 : x) \to x$, which is alpha-equivalent to the intended type $(A : Set) \to A \to A$. Similarly, we can

interpret term *idTm* and its typing proof ty_{idTm} to obtain $intrp_{idTm}$ of type $intrp_{idTyTm}$.

$$intrp_{idTm} = interp\ ty_{idTm}\ ty_{\epsilon}\ tt$$

As intended, $intrp_{idTm}$ reduces to $\lambda\ (x : Set) \to (x_1 : x) \to x_1$, alpha-equivalent to our intended $\lambda\ (A : Set) \to \lambda\ (x : A) \to x$.

Interfacing with the real world In real examples, generated code needs to interface with existing types and values. In staging meta-programming systems, this is supported with a built-in *quoting* primitive, but we use an alternative approach. Suppose, for example, that we want a meta-program to construct the term *suc* 2 from the pre-existing value 2 and function *suc*. To do this, the meta-program clearly needs to refer to the type $\mathbb{N}$, the function *suc* and the value 2 in the generated object code, but our term encoding does not provide a way to refer to such outside definitions. One solution would be to build natural numbers into our calculus as primitives, but this is not a scalable approach, since we cannot expect to do this for all types we will ever need, let alone a user's custom types.

A better solution lets the meta-program construct the object term in a suitable context, postulating values of the correct types. Real values can then be provided in the interpretation of this context. For our example, we need the context Γ_{ex}:

$$\Gamma_{ex} = (pi\ (var\ 1)\ (var\ 2)) \triangleleft\ (var\ 0) \triangleleft\ set \triangleleft\ \varepsilon$$

This definition should be read right-to-left: $\triangleleft$ is right-associative and the left-most context entries are added last and may refer to the values of entries to their right. It starts with the empty context ε and lists the types for which we want to postulate values. In order, these are a type (of type *set*), a value of this type (of type *var* 0) and a function from this type to itself (of type *pi* (*var* 1) (*var* 2)). The context is intended to be instantiated to values $\mathbb{N}$, 2 and *suc* respectively. Note that the de Bruijn variables *var* 0, *var* 1 and *var* 2 in the context all refer to the value of the rightmost context entry of type *set*; subsequent context entries have an additional variable in scope and the body of a *pi* as well. Proof $ty_{\Gamma ex}$ of judgement $\vdash \Gamma_{ex}$ shows that context Γ_{ex} is well-typed, i.e. all entries are in fact sets:

$$ty_{\Gamma ex} = ty_{\triangleleft}\ (ty_{\triangleleft}\ (ty_{\triangleleft}\ ty_{\epsilon}\ typeSet)\ typeVar)\ (typePi\ typeVar\ typeVar)$$

We will fill in the appropriate values for this context's assumptions with the value $asmpts_{\Gamma ex}$ of type $interpCtx\ ty_{\Gamma ex}$:

$$asmpts_{\Gamma ex} = ((tt, \mathbb{N}), 1), suc$$

In context Γ_{ex}, we can now construct the value *suc* 2 as a term *ex*. It is an *Expr* 3, since it may refer to Γ_{ex}'s three assumptions, and applies the postulated *suc* function to the postulated value 2.

$$ex = appl\ (var\ 0)\ (var\ 1)$$

We construct a proof ty_{ex} of judgement $\Gamma_{ex} \vdash ex : var\ 2$, i.e. that the constructed term *ex* has the first postulated value ($\mathbb{N}$) as its type, in three steps. First typing rules *typeAppl* and *typeVar* give us proof ty'_{ex}, showing that *ex* has a more complicated type. We then prove this type convertible to *var* 2 in (partly omitted) proof $conv_{ex}$. ty_{ex} then uses typing rule *typeConv* to replace the convertible type.

$$\begin{aligned}
&ty'_{ex}\ :\ \Gamma_{ex} \vdash ex : appl\ (lambda\ (var\ 2)\ (var\ 3))\ (var\ 1)\\
&ty'_{ex} = typeAppl\ typeVar\ typeVar\ typeVar\\
&conv_{ex}\ :\ appl\ (lambda\ (var\ 2)\ (var\ 3))\ (var\ 1) \approx var\ 2\\
&conv_{ex} = \cdots (reduceApplication\ (var\ 3)\ (var\ 1))\\
&ty_{ex}\ :\ \Gamma_{ex} \vdash ex : var\ 2\\
&ty_{ex} = typeConv\ conv_{ex}\ ty'_{ex}\ typeVar
\end{aligned}$$

We can then interpret the object program *ex* to obtain a value of type $interpSet\ (typesAreSets\ ty_{\Gamma ex}\ ty_{ex})\ ty_{\Gamma ex}\ asmpts_{\Gamma ex}$:

$$exInt = interp\ ty_{ex}\ ty_{\Gamma ex}\ asmpts_{\Gamma ex}$$

The reduction behaviour of our primitives that we will talk about next ensures that *exInt*'s type and *exInt* itself reduce to $\mathbb{N}$ and *suc* 2 respectively, precisely as we intended.

Sometimes, a meta-program does not just need to refer to an external function *f* in generated code, but also depends on information about such a function's reduction behaviour to prove well-typedness of the generated code. Without going into much detail, the ideas of this section can support this if we add singleton types to the object calculus. Concretely, a context could postulate the external function *f* together with proofs of its reduction behaviour. Such proofs could then be used in the typing of generated programs and the invocation of the interpretation primitive would require actual proofs of the reduction behaviour in the context interpretation.

Reduction behaviour The reduction behaviour of our primitives is an important part of their definition and crucial for the functioning of the previous examples. We present the reduction rules in Figure 4.8. In general, these rules interpret encoded types, terms and contexts, but only when the well-typedness of the result can be guaranteed. To achieve the latter, we need to ascertain that

$$
\begin{array}{llllll}
\mathit{interp} & \mathit{typeSet} & \mathit{ty}_\Gamma & \mathit{asmpts} & = & \mathit{Set} \\
\mathit{interp} & (\mathit{typeVar}\ \{i = i\}) & \mathit{ty}_\Gamma & \mathit{asmpts} & = & \mathit{interpVar}\ i\ \mathit{ty}_\Gamma\ \mathit{asmpts} \\
\mathit{interp} & (\mathit{typePi}\ \mathit{ty}_s\ \mathit{ty}_t) & \mathit{ty}_\Gamma & \mathit{asmpts} & = & (x : \mathit{interpSet}\ \mathit{ty}_s\ \mathit{ty}_\Gamma\ \mathit{asmpts}) \to \mathit{interpSet}\ \mathit{ty}_t\ (\mathit{ty}_\triangleleft\ \mathit{ty}_\Gamma\ \mathit{ty}_s)\ (\mathit{asmpts}, x) \\
\mathit{interp} & (\mathit{typeLam}\ \mathit{ty}_s\ \mathit{ty}_b) & \mathit{ty}_\Gamma & \mathit{asmpts} & = & \lambda\,(x : \mathit{interpSet}\ \mathit{ty}_s\ \mathit{ty}_\Gamma\ \mathit{asmpts}) \to \mathit{interp}\ (\mathit{ty}_\triangleleft\ \mathit{ty}_\Gamma\ \mathit{ty}_s)\ \mathit{ty}_b\ (\mathit{asmpts}, x) \\
\mathit{interp} & (\mathit{typeAppl}\ \mathtt{ty_t}\ \mathtt{ty_f}\ \mathtt{ty_{val}}) & \mathit{ty}_\Gamma & \mathit{asmpts} & = & \mathit{interp}\ \mathit{ty}_\Gamma\ \mathit{ty}_f\ \mathit{asmpts}\ (\mathit{interp}\ \mathit{ty}_\Gamma\ \mathit{ty}_{val}\ \mathit{asmpts}) \\
\mathit{interp} & (\mathit{typeConv}\ \mathtt{t{\sim}t'}\ \mathtt{ty_e}\ \mathtt{ty_t}) & \mathit{ty}_\Gamma & \mathit{asmpts} & = & \mathit{interp}\ \mathit{ty}_\Gamma\ \mathit{ty}_e\ \mathit{asmpts}
\end{array}
$$

$$
\begin{array}{llll}
\mathit{interpCtx} & \mathit{ty}_\epsilon & = & \top \\
\mathit{interpCtx} & (\mathit{ty}_\triangleleft\ \mathit{ty}_\Gamma\ \mathit{ty}_t) & = & \exists\,\lambda\,(\mathit{asmpts} : \mathit{interpCtx}\ \mathit{ty}_\Gamma) \to \mathit{interpSet}\ \mathit{ty}_t\ \mathit{ty}_\Gamma\ \mathit{asmpts}
\end{array}
$$

$$
\begin{array}{l}
\mathit{interpVar} : \forall\ \{n\}\ \{\Gamma : \mathit{Context}\ n\}\ i \to (\mathit{ty}_\Gamma : \vdash \Gamma) \to (\mathit{asmpts} : \mathit{interpCtx}\ \mathit{ty}_\Gamma) \to \mathit{interpSet}\ (\vdash\!\!-\mathit{var}\ \mathit{ty}_\Gamma\ i)\ \mathit{ty}_\Gamma\ \mathit{asmpts} \\
\mathit{interpVar}\ \ \mathit{zero}\ \ (\mathit{ty}_\triangleleft\ \mathit{ty}_\Gamma\ \mathtt{ty_t})\ \ (\_, \mathit{asmpt}) = \mathit{asmpt} \\
\mathit{interpVar}\ \ (\mathit{suc}\ i)\ \ (\mathit{ty}_\triangleleft\ \mathit{ty}_\Gamma\ \mathit{ty}_t)\ \ (\mathit{asmpts}, \_) = \mathit{interpVar}\ \mathit{ty}_\Gamma\ i\ \mathit{asmpts} \\
\mathit{interpSet}\ \mathit{ty}_t\ \mathit{ty}_\Gamma\ \mathit{asmpts} = \mathit{interpSet}'\ \mathit{ty}_t \approx\!\!-\mathit{refl}\ \mathit{ty}_\Gamma\ \mathit{asmpts}
\end{array}
$$

$$
\begin{array}{llllll}
\mathit{interpSet}' : \forall\ \{n\}\ \{\Gamma : \mathit{Context}\ n\}\ \{A\ t\} \to & \Gamma \vdash A : t & \to & t \approx \mathit{set} & \to & (\mathit{ty}_\Gamma : \vdash \Gamma) \to \mathit{interpCtx}\ \mathit{ty}_\Gamma \to \mathit{Set} \\
\mathit{interpSet}' & \mathit{typeSet} & \mathit{eq} & \mathit{ty}_\Gamma & \mathit{asmpts} & = \mathit{Set} \\
\mathit{interpSet}' & (\mathit{typeVar}\ \{i = i\}) & \mathit{eq} & \mathit{ty}_\Gamma & \mathit{asmpts} & = \mathit{interpVarSet}\ i\ \mathit{eq}\ \mathit{ty}_\Gamma\ \mathit{asmpts} \\
\mathit{interpSet}' & (\mathit{typePi}\ \mathit{ty}_s\ \mathit{ty}_t) & \mathit{eq} & \mathit{ty}_\Gamma & \mathit{asmpts} & = \\
\multicolumn{6}{l}{\quad (x : \mathit{interpSet}\ \mathit{ty}_s\ \mathit{ty}_\Gamma\ \mathit{asmpts}) \to \mathit{interpSet}\ \mathit{ty}_t\ (\mathit{ty}_\triangleleft\ \mathit{ty}_\Gamma\ \mathit{ty}_s)\ (\mathit{asmpts}, x)} \\
\mathit{interpSet}' & (\mathit{typeAppl}\ \mathtt{ty_t}\ \mathtt{ty_f}\ \mathit{ty}_{val}) & \mathtt{eq} & \mathit{ty}_\Gamma & \mathit{asmpts} & = \mathit{interp}\ \mathit{ty}_f\ \mathit{ty}_\Gamma\ \mathit{asmpts}\ (\mathit{interp}\ \mathit{ty}_{val}\ \mathit{ty}_\Gamma\ \mathit{asmpts}) \\
\mathit{interpSet}' & (\mathit{typeConv}\ t{\sim}t'\ \mathit{ty}'_A\ \mathit{ty}_t) & \mathit{eq} & \mathit{ty}_\Gamma & \mathit{asmpts} & = \mathit{interpSet}'\ \mathit{ty}'_A\ (\approx\!\!-\mathit{trans}\ (\approx\!\!-\mathit{sym}\ t{\sim}t')\ \mathit{eq})\ \mathit{ty}_\Gamma\ \mathit{asmpt}
\end{array}
$$

$$
\begin{array}{l}
\mathit{interpVarSet} : \forall\ \{n\}\ \{\Gamma : \mathit{Context}\ n\}\ \{l\}\ i \to \mathit{lookup}\ i\ \Gamma \approx \mathit{set} \to (\mathit{ty}_\Gamma : \vdash \Gamma) \to \mathit{interpCtx}\ \mathit{ty}_\Gamma \to \mathit{Set} \\
\mathit{interpVarSet}\ \ \mathit{zero}\ \ \mathtt{eq}\ \ (\mathit{ty}_\triangleleft\ \mathit{ty}_\Gamma\ \mathtt{ty_t})\ \ (\_, \mathit{asmpt}) = \mathit{asmpt} \\
\mathit{interpVarSet}\ \ (\mathit{suc}\ i)\ \ \mathit{eq}\ \ (\mathit{ty}_\triangleleft\ \mathit{ty}_\Gamma\ \mathit{ty}_t)\ \ (\mathit{asmpts}, \_) = \mathit{interpVarSet}\ i\ (\mathit{weaken}{-}\mathit{inj}{-}{\approx}\ (\mathit{lookup}\ i\ \Gamma)\ \mathit{set}\ \mathit{eq})\ \mathit{ty}_\Gamma\ \mathit{asmpts}
\end{array}
$$

Figure 4.8: Reduction behaviour of our primitives. Patterns in `red typewriter` font are required to be values.

the provided well-typedness proofs are valid and do not rely on assumptions that might not hold. This is non-trivial because a language like Agda applies strong reductions during type-checking, i.e. reductions can be applied to open terms as well as closed. Non-closed proofs are not necessarily valid, since they may rely on invalid assumptions. We will provide more insight further on and discuss our solution based on the *value patterns* in Figure 4.8. These are the patterns written in `typewriter` font in the left-hand sides of some reduction rules. Such a value pattern indicates that the rule must only be applied if the corresponding argument is a *value*. The types of these arguments are conversion or typing judgements and their values are finite trees of constructor applications (see Figures 4.3 and 4.5). As such, the property of value-ness can easily be checked in the primitives' implementation. But before we discuss the role of the value patterns further, let us take a better look at the reduction rules.

Recall the type of our most important primitive *interp*.

$$\begin{aligned} &\mathit{interp} : \{n : \mathbb{N}\}\ \{\Gamma : \mathit{Context}\ n\}\ \{v\ t : \mathit{Expr}\ n\} \rightarrow \\ &\quad (\mathit{ty}_v : \Gamma \vdash v : t) \ \rightarrow\ (\mathit{ty}_\Gamma : \vdash \Gamma) \ \rightarrow\ (\mathit{asmpts} : \mathit{interpCtx}\ \mathit{ty}_\Gamma) \ \rightarrow \\ &\quad \mathit{interpSet}\ (\mathit{typesAreSets}\ \mathit{ty}_\Gamma\ \mathit{ty}_v)\ \mathit{ty}_\Gamma\ \mathit{asmpts} \end{aligned}$$

The primitive takes a context Γ, a term v and a type t as hidden arguments, followed by proofs ty_v, ty_Γ of typing judgements $\Gamma \vdash v : t$ and $\vdash \Gamma$ and a value *asmpts* of the context's interpretation type $\mathit{interpCtx}\ \mathit{ty}_\Gamma$. The reduction rules in Figure 4.8 specify that for certain forms of the judgement ty_v, the primitive application reduces to appropriate right-hand sides. For $\mathit{ty}_v = \mathit{typeSet}$, which implies[4] $v = \mathit{set}$ and $t = \mathit{set}$, the first rule returns interpretation *Set*. For $\mathit{ty}_v = \mathit{typeVar}$, an interpretation of the ith context assumption is given by primitive *interpVar*, discussed below.

The rules for $\mathit{ty}_v = \mathit{typePi}\ \mathit{ty}_s\ \mathit{ty}_t$ and $\mathit{typeLam}\ \mathit{ty}_s\ \mathit{ty}_b$ interpret terms $\mathit{pi}\ s\ t$ and $\mathit{lambda}\ s\ b$ as respectively the corresponding Agda Π-type and lambda term, recursively constructed from interpretations of s, t and b. The bound variable x is made available for the interpretation of t and b by placing it in the interpretation of the extended context $s \triangleleft \Gamma$. For an application of a function to a value, we apply the interpretation of the function to the interpretation of the value. Note the value patterns on the left-hand side that we will come back to further on. Finally, the interpretation of a *typeConv* is simply the interpretation of the judgement whose type it substitutes, on the condition that the arguments are values.

Recall also the type of primitive *interpCtx*:

$$\mathit{interpCtx} : \{n : \mathbb{N}\}\ \{\Gamma : \mathit{Context}\ n\} \rightarrow \vdash \Gamma \rightarrow \mathit{Set}$$

[4]Note: pattern matches that imply equalities about other arguments are standard for dependent pattern matching [82].

The primitive takes a context Γ as a hidden argument and a well-typedness proof for it and returns its interpretation, i.e. a type that contains all the context's assumptions. We saw in *interp*'s reduction rules for *typeLam* and *typePi*, how an extended context $s \triangleleft \Gamma$ is interpreted by a tuple of the s value and the interpretation of Γ. This corresponds to *interpCtx*'s reduction behaviour, that we look at now. The first reduction rule interprets an empty context by the unit type $\top$. More interestingly, a context Γ extended with a type t is interpreted by an interpretation *asmpts* of Γ, and an interpretation of the type t. We use a dependent sum $\exists$ to specify the interpretation of t with respect to the interpretation *asmpts* of the rest of the context.

Now that we know how to interpret a context, we can define reduction rules for *interpVar*, to project out a context's ith entry. Its reduction rules are not surprising, projecting out the top assumption for variable *zero* and recursing for *suc i*. The primitive *interpSet* is a version of *interp* that works on types only. Its role is to break the circularity in the types of the primitives. It is implemented in terms of helper primitives *interpSet′* and *interpVarSet*. We do not discuss their reduction behaviour as it is similar to *interp* and *interpVar* except that we require proof that the judgement's type is convertible to *set* and that this proof is a value in some cases.

Although the reduction rules in Figure 4.8 superficially resemble a definition by dependent pattern matching [82] (as supported by Agda), they should not be understood as such a definition, because they do not satisfy several of the criteria for such a definition. A first problem is that Agda does not have value patterns, but this is a technical problem, since it is possible to simulate their behavior using pattern matching. A second, more fundamental problem is that the clauses are not structurally recursive, so that Agda's termination checker would not accept them. Furthermore, not all clauses satisfy the criteria for Agda's type-checker. One problem here is the peculiar recursion structure, where *interp*'s type mentions the *interpSet* function, which uses *interp* in its definition. Another problem is that type-safety of the clauses sometimes relies on the fact that for $ty_\Gamma : \vdash \Gamma$, $asmpts : interpCtx\ ty_\Gamma$ and $prf : \Gamma \vdash A : set$, $interpSet\ prf\ ty_\Gamma\ asmpts$ will produce the same result as $interp\ prf\ ty_\Gamma\ asmpts$. All these problems would be hard to solve, and because of them, the rules in Figure 4.8 should be interpreted as defining the reduction behavior of primitives.

Soundness in the presence of open terms To understand the value patterns in five of the reduction rules in Figure 4.8, we have to explain the powerful form of type-level computation that a dependently typed language like Agda uses. It uses a *strong* form of reductions: reductions can be applied even inside the body of lambda or pi terms. The term $\lambda\ x \to 0 + x$, for example, is considered

equal to $\lambda\ x \to x$, because $0 + x$ is reduced to x despite the open variable x. However, such strong reductions can be dangerous because, in the presence of open variables, we may be reasoning under absurd assumptions. Consider the following function:

$$absurdTerm = \lambda\ (prf : Int \equiv Bool) \to cast\ prf\ 3 \vee false$$

The function *absurdTerm* takes a proof *prf* that $Int \equiv Bool$, modelling an equality proof of types *Int* and *Bool*. This proof type is of course empty, but the type-checker is not aware of that. With *prf* and an appropriate *cast* function, we can use a value 3 as a *Bool*. However, this is not problematic, because a correct definition of the *cast* function will never reduce *cast prf* 3 to 3. Instead, it will block on the open variable *prf* until a value (i.e. *refl*) is somehow substituted for it. This mechanism effectively protects values like 3 from being used at wrong types like *Bool*.

For our primitives, similar issues arise. We can, for example, assume a proof ty_{absurd} of judgement $\varepsilon \vdash set : pi\ set\ set$ even though this type of proofs is empty. Clearly, $interp\ ty_{absurd}\ ty_{\epsilon}\ tt$ should then not reduce to *Set* at type $Set \to Set$, but instead block on the open variable ty_{absurd}. Similarly, if we assume a proof *prf* of judgement $pi\ set\ set \approx set$, and use it with *typeConv* to construct a proof ty'_{absurd} of judgement $\varepsilon \vdash set : pi\ set\ set$, then our primitives should block on open variable *prf*.

By the value patterns in Figure 4.8, some rules require that certain arguments are values. We have checked for each rule that the right-hand side's type was equal to the declared type, assuming just the information from the left-hand side patterns, similar to how dependent pattern matching can be type-checked [82]. For the five rules with value patterns, this was not the case. In, for example, the rule for $interp\ (typeConv\ \mathtt{t}{\sim}\mathtt{t'}\ \mathtt{ty_e}\ \mathtt{ty_t})\ ty_\Gamma\ asmpts$, the right-hand-side is of type

$$interpSet\ (typesAreSets\ ty_\Gamma\ ty_e)\ ty_\Gamma\ asmpts$$

i.e. the interpretation of t', not t and the convertibility assumption $t \sim t'$ is essential for returning a value of type t' as one of type t. We believe that the value patterns in Figure 4.8 solve this problem, because they prevent the clause from applying when the primitives are applied to non-closed arguments and for closed arguments, the right-hand-side's type will match the declaration. Nevertheless, this is clearly not a very formal argument and the general question of soundness remains open.

The primitives' properties In addition to the reduction behaviour of our primitives, some of our meta-programs require additional properties about them

$$
\begin{aligned}
&\mathit{castInterp{-}{\approx}'} : \forall\ \{n\}\ \{\Gamma : \mathit{Context}\ n\}\ \{A\ A'\} \to \\
&\quad (\mathit{ty}_A : \Gamma \vdash A : \mathit{set}) \ \to\ (\mathit{ty}'_A : \Gamma \vdash A' : \mathit{set}) \ \to\ A \approx A' \ \to \\
&\quad (\mathit{ty}_\Gamma : \vdash \Gamma) \ \to\ (\mathit{asmpts} : \mathit{interpCtx}\ \mathit{ty}_\Gamma) \ \to \\
&\qquad \mathit{interpSet}\ \mathit{ty}_A\ \mathit{ty}_\Gamma\ \mathit{asmpts} \equiv \mathit{interpSet}\ \mathit{ty}'_A\ \mathit{ty}_\Gamma\ \mathit{asmpts} \\
&\mathit{interpCompSubCtx} : \forall\ \{m\ n\}\ \{\Gamma_1\ \Gamma_2\}\ \{\phi : \mathit{Sub}\ \mathit{Expr}\ m\ n\} \to \\
&\quad \Gamma_2 \vdash \phi : \Gamma_1 \ \to\ (\mathit{ty}\Gamma_1 : \vdash \Gamma_1) \ \to\ (\mathit{ty}\Gamma_2 : \vdash \Gamma_2) \ \to \\
&\quad \mathit{interpCtx}\ \mathit{ty}\Gamma_2 \ \to\ \mathit{interpCtx}\ \mathit{ty}\Gamma_1 \\
&\mathit{interpCompSubSet}' : \forall\ \{m\ n\ t\}\ \{\Gamma_1\ \Gamma_2\}\ \{\phi : \mathit{Sub}\ \mathit{Expr}\ m\ n\} \to \\
&\quad (\mathit{comp} : \Gamma_2 \vdash \phi : \Gamma_1) \ \to\ (\mathit{ty}\Gamma_1 : \vdash \Gamma_1) \ \to\ (\mathit{ty}_t : \Gamma_1 \vdash t : \mathit{set}) \ \to \\
&\quad (\mathit{ty}\Gamma_2 : \vdash \Gamma_2) \ \to\ (\mathit{asmpts}_2 : \mathit{interpCtx}\ \mathit{ty}\Gamma_2) \ \to \\
&\quad \mathit{interpSet}\ (\vdash -/\mathit{comp}\ \mathit{ty}_t)\ \mathit{ty}\Gamma_2\ \mathit{asmpts}_2 \equiv \\
&\qquad \mathit{interpSet}\ \mathit{ty}_t\ \mathit{ty}\Gamma_1\ (\mathit{interpCompSubCtx}\ \mathit{comp}\ \mathit{ty}\Gamma_1\ \mathit{ty}\Gamma_2\ \mathit{asmpts}_2)
\end{aligned}
$$

Figure 4.9: Primitive properties

(Figure 4.9). Property *castInterp*$-\approx'$ states that for convertible types A and A', the interpretations under *interpSet* must be the same. The next two properties are related to the interpretation of a type after a well-typed substitution $\Gamma_2 \vdash \phi : \Gamma_1$ between well-typed contexts Γ_1 and Γ_2. *interpCompSubCtx* says that an interpretation of Γ_1 can be constructed from one of Γ_1 and *interpCompSubSet*$'$ says that the interpretation of a type t in Γ_2 is the same as that of $t\ /\ \phi$ in Γ_1 using the interpretation of Γ_1 constructed by *interpCompSubCtx*.

We are currently using stub proofs of these properties, based on an Agda primitive called *primTrustMe*. *primTrustMe* is an unsafe primitive that proves equalities $a \equiv b$ for any set A and values a, b of type A. However, during type-checking, *primTrustMe* only reduces to *refl* when a and b are definitionally equal. It is future work to ascertain that these properties follow from the reduction rules of Figure 4.8 and the proofs of theorems like $\vdash -/$.

In summary, the primitives we introduce are *interpCtx*, *interpSet* and *interp*. Their types are listed in Section 4.3 on page 141 and their reduction behavior is specified in Figure 4.8 on page 144. Additionally, we are currently using stub proofs of the primitives' properties in Figure 4.9.

4.4 Applications

Our approach allows definitions of powerful meta-programs, manipulating both code and types, in a functional style and with very precise types. In this

section, we demonstrate this for two important applications: datatype-generic programming and tactics.

4.4.1 Datatype-generic programming

The field of datatype-generic programming studies the definition of algorithms that work for a wide variety of data types. An example is Haskell's **deriving** *Show* mechanism [138, §4.3.3, §11], which allows a data type A to be annotated with the directive **deriving** *Show* to make the compiler derive an instance of the *Show* type class. Such an instance consists essentially of a function $show :: A \to String$, derived syntactically by the compiler from the data type's constructors and their types. The goal of datatype-generic programming is to allow functions like *show* to be defined in a generic way, i.e. such that they can be defined once but used with a wide variety of data types.

Representing data types To apply our techniques to the field of datatype-generic programming, we start from a syntactic representation of an inductive data type:

```
record SimpleDT (A : Set) : Set where
  constructor simpleDT
  field constructors : List (Constructor A)
        folder       : folderType A constructors
```

According to this definition, a data type A is syntactically described by a list of its constructors and a *folder* or *induction principle* (*List* is a standard type of finite lists). To keep things simple, we omit well-formedness requirements (like positivity of the definition) and proofs about the reduction behaviour of the folder function, which are required to completely describe a data type, but not needed for our example application. *Constructor* is the syntactic representation of a single constructor:

```
data Constructor (A : Set) : Set where
  mkConstructor : String → (n : ℕ) →
    (tel : Telescope 1 (n + 1)) → (ty_tel : Γset ⊢ tel) →
    let ctorT = funCtx n tel (var 0)
        ty_ctorT : Γset ⊢ ctorT : set
        ty_ctorT = typeFunCtx n ty_tel typeVar
    in interpSet ty_Γset ty_ctorT (tt, A) → Constructor A
```

We describe a constructor by its name as a *String*, its arity n and a telescope *tel* containing the types of its arguments. The telescope has one initial variable

in scope: the data type A itself, so that it can be referenced in the types of constructor arguments. The telescope *tel* must be well-typed in the context $\Gamma set = set \triangleleft \varepsilon$, i.e. with the premise that A is a set. From *tel*, we can calculate the full type $ctorT$ of the constructor as the function that takes the arguments given by *tel* and produces a value of type A (using omitted helper function *funCtx*). We prove that $ctorT$ is a set (using omitted lemma *typeFunCtx*), interpret it and require a value of it, i.e. the actual constructor. For interpreting the type, we crucially rely on the *interpSet* primitive, which provides the link between the syntactically represented types and the normal type of the actual constructor.

In addition to the list of *Constructor*s, $SimpleDT$ contains an eliminator or *folder* for the data type. Every inductive data type comes with such an induction principle, which models a general way of perform structural induction over the data type. The function *folderType* syntactically derives the type of this induction principle from the types of the constructors and their interpretations.

$$
\begin{aligned}
&folderType : (A : Set) \to List\ (Constructor\ A) \to Set \\
&folderType\ A\ constructors = (P : A \to Set) \to \\
&\quad underFolderAsmpts\ A\ P\ constructors\ ((x : A) \to P\ x)
\end{aligned}
$$

Given a set A and a list of A's constructors, *folderType* returns the type for a corresponding induction principle: it takes a predicate $P : A \to Set$ (the *motive* [82], describing what the induction principle should produce) and returns a function of type $(x : A) \to P\ x$ under a number of assumptions. For every constructor, the function *underFolderAsmpts* syntactically derives the type of an assumption from the constructor's type. This is fairly involved, but presents no fundamental difficulties and we omit it for space reasons.

Let us immediately show some data types and their representations. The simplest example is the empty type, which has zero constructors. Its definition and induction principle look as follows:

$$
\begin{aligned}
&\textbf{data}\ \bot : Set\ \textbf{where} \\
&foldBot : (P : \bot \to Set) \to (t : \bot) \to P\ t \\
&foldBot\ P\ ()
\end{aligned}
$$

Note the use of an absurd pattern () in the definition of *foldBot*. This pattern communicates to Agda that no value can ever be given for the argument of type $\bot$, so that a right-hand-side is not needed. It is easy to provide a value of $SimpleDT$ for $\bot$:

$$
\begin{aligned}
&botDT\ :\ SimpleDT\ \bot \\
&botDT = simpleDT\ [\,]\ foldBot
\end{aligned}
$$

botDT specifies that $\bot$ has no constructors and *foldBot* is its induction principle. Agda successfully type-checks *foldBot* against the folder type calculated for the empty list of constructors.

For a more complex example, consider the standard definition of natural numbers and its induction principle:

$$
\begin{aligned}
&\textbf{data}\ \mathbb{N} : \mathit{Set}\ \textbf{where}\ \mathit{zero}\ : \mathbb{N} \\
&\qquad\qquad\qquad\qquad\ \ \mathit{suc}\ \ : \mathbb{N} \to \mathbb{N} \\
&\mathit{fold}\mathbb{N} : (P : \mathbb{N} \to \mathit{Set}) \to\ \ P\ \mathit{zero}\ \ \to \\
&\quad (\forall\ n \to P\ n \to P\ (\mathit{suc}\ n))\ \ \to\ \ (n : \mathbb{N}) \to P\ n \\
&\mathit{fold}\mathbb{N}\ P\ \mathit{Pz}\ \mathit{Ps}\ \mathit{zero}\ \ \ \ = \mathit{Pz} \\
&\mathit{fold}\mathbb{N}\ P\ \mathit{Pz}\ \mathit{Ps}\ (\mathit{suc}\ n) = \mathit{Ps}\ n\ (\mathit{fold}\mathbb{N}\ P\ \mathit{Pz}\ \mathit{Ps}\ n)
\end{aligned}
$$

The constructors *zero* and *suc* of data type $\mathbb{N}$ are described by *zeroConstr* and *sucConstr* of type *Constructor* $\mathbb{N}$:

$$
\begin{aligned}
&\mathit{zeroConstr} = \mathit{mkConstructor}\ \texttt{"zero"}\ 0\ \varepsilon\ \mathit{ty}_{\epsilon}\ \mathit{zero} \\
&\mathit{sucConstr}\ \ = \mathit{mkConstructor}\ \texttt{"suc"}\ 1\ (\mathit{var}\ 0 \lhd\ \varepsilon)\ (\mathit{ty}_{\lhd}\ \mathit{ty}_{\epsilon}\ \mathit{typeVar})\ \mathit{suc}
\end{aligned}
$$

The constructor *zero* is of arity 0, with the empty telescope describing its arguments. The actual constructor *zero* is then provided and Agda checks its type against the one calculated from the syntactic description. Constructor *suc* is of arity 1, taking one value of type $\mathbb{N}$ as its argument (recall that *var* 0 in the constructor telescope refers to the data type itself). The constructor telescope is well-typed under *Γset*'s assumption that *var* 0 is a set. Again, the actual constructor is given and checked against the type calculated from the description. We can now describe $\mathbb{N}$ with $\mathit{natDT} : \mathit{SimpleDT}\ \mathbb{N}$.

$$\mathit{natDT} = \mathit{simpleDT}\ [\mathit{zeroConstr}, \mathit{sucConstr}]\ \mathit{fold}\mathbb{N}$$

natDT lists $\mathbb{N}$'s constructors and provides induction principle $\mathit{fold}\mathbb{N}$, checked against the type calculated from the constructors.

Derive Show The type *SimpleDT* is a general syntactic description of inductive data types that permits a general form of datatype-generic meta-program. As a proof-of-concept, we show the function *deriveShow* that derives a *show* function for a data type A.

$$
\begin{aligned}
&\mathit{deriveShow} : \forall\ \{A\} \to \mathit{SimpleDT}\ A \to A \to \mathit{String} \\
&\mathit{deriveShow}\ (\mathit{simpleDT}\ \mathit{constructors}\ \mathit{folder}) = \text{omitted}
\end{aligned}
$$

We omit the algorithm's implementation, which takes the description of data type A and exploits the induction principle with motive $P = \lambda\ \_ \to \mathit{String}$. It

syntactically derives arguments for the folder, specifying how values constructed using the different constructors are to be serialised. The hardest part of the code is to convince the type-checker that the folder arguments we construct for the concrete motive $\lambda\ \_ \to \mathit{String}$ correspond to their expected types for a general predicate P when P is instantiated to $\lambda\ \_ \to \mathit{String}$ through the context interpretation. This essentially uses the *interpCompSubCtx* and *interpCompSubSet* primitive properties shown in Figure 4.9.

For our example data types, *deriveShow* derives an (admittedly not very useful) *show* function for $\bot$:

$$\begin{aligned}&\mathit{showBot}\ :\ \bot \to \mathit{String}\\&\mathit{showBot} = \mathit{deriveShow}\ \mathit{botDT}\end{aligned}$$

showBot's definition reduces to $\mathit{foldBot}\ (\lambda\ \_ \to \mathit{String})$, the code that *deriveShow* syntactically generates. From *natDT*, we can derive the function *showNat* of type $\mathbb{N} \to \mathit{String}$.

$$\mathit{showNat} = \mathit{deriveShow}\ \mathit{natDT}$$

Like for *showBot*, *showNat*'s definition reduces to the generated function $\mathit{showNat}' = \mathit{fold}\mathbb{N}\ (\lambda\ \_ \to \mathit{String})$ `"zero"` (...) (final argument omitted). We can apply it to numbers with, for example, *showNat* 2 producing the string `"(suc (suc zero))"`.

Discussion This account of datatype-generic programming is rudimentary, lacking support for indices and parameters and non-recursive and more general recursive constructor arguments [66]. It does not exclude non-strictly-positive data types and does not contain proofs about the induction principle's reduction behaviour (required to construct proofs about inductive functions). However, we do not see fundamental obstacles for adding any of this.

From a methodological point of view, our account of datatype-generic programming is compelling: meta-programs are written in the language itself, using the language's standard functional computational model. The syntactic description of a data type in *SimpleDT* is general and could be automatically generated by the compiler. Modulo correctness of our primitives, the meta-programs come with strong guarantees about termination, well-typedness of the generated programs and completeness.

SimpleDT and *deriveShow* are implemented in ±1200 lines of code and can be studied in the full version of our code (see the footnote on Page 133). This is still much more than what we would like, and in Section 4.5 we discuss how this could be improved.

4.4.2 Tactics

Tactics are a form of meta-programs that solve or refine proof obligations in proof assistants. In proof assistants based on dependent type theory, solving a proof obligation is equivalent to producing a program of a specified type in a specified context. Several proof assistants provide support for writing tactics, often in the form of a special-purpose sub-language. Such tactics are generally untyped and provide little upfront guarantees about their correct operation. Even though the correctness of the generated proofs can be checked after generation, Gonthier et al. argue that untyped tactics can be hard to maintain and compose and giving them more precise types is a good approach to solve this issue [86]. There are also performance advantages to tactics that can be guaranteed to terminate correctly without running them, as argued by Chlipala [37].

Our meta-programming primitives show promise for this field, and they lend themselves to a typed form of tactics written in a standard functional style. The input for a tactic is just a syntactic representation of the proof obligation, i.e. a certain type in a certain context. By additionally requiring a typing judgement for the type and interpretations for the context's values, we can use *interpSet* to specify the expected result type of the tactic.

Consider the following analogue of Coq's *assumption* tactic, a simple tactic that solves proof obligations which appear literally in the context. Our account of it enjoys a very precise type:

$$\begin{aligned} &\mathit{assumptionTactic} : \forall\ \{n\ T\}\ \{\Gamma : \mathit{Context}\ n\} \to \\ &\quad (\mathit{ty}_T : \Gamma \vdash T : \mathit{set}) \ \to\ (\mathit{ty}_\Gamma : \vdash \Gamma) \ \to\ (\mathit{asmpts} : \mathit{interpCtx}\ \mathit{ty}_\Gamma) \to \\ &\quad \mathit{ifYes}\ (\mathit{inContext}?\ \Gamma\ T)\ (\mathit{interpSet}\ \mathit{ty}_T\ \mathit{ty}_\Gamma\ \mathit{asmpts}) \end{aligned}$$

The tactic takes a type T, a well-typed context Γ and values for its assumptions. The return type will be explained further, but it specifies exactly what the tactic will return in all cases: either a value of type T if T is present in the context or a value of the unit type otherwise. Let us explain this in more detail.

We use the Agda standard library's $\mathit{Dec}\ P$ type. It models a decision of proposition P, i.e. either a proof of P or a proof of $\neg P$:

$$\begin{aligned} \textbf{data}\ \mathit{Dec}\ (P : \mathit{Set}) : \mathit{Set}\ \textbf{where}\ \ &\mathit{yes} : P \ \ \ \ \ \ \to \mathit{Dec}\ P \\ &\mathit{no}\ : \neg\ P \ \to \mathit{Dec}\ P \end{aligned}$$

Based on a decision of some property, the *ifYes* function returns either an argument type or unit type $\top$:

```
ifYes : {P : Set} → Dec P → Set → Set
ifYes (yes _) P′ = P′
ifYes (no _)  _  = ⊤
```

The *inContext*? algorithm decides whether or not a certain type t is present in context Γ, i.e. if the ith entry in the context is equal to t for some i. It uses a general decision procedure *any*?, which simply tries all i of the bounded type *Fin n*. For a given variable i, we use a general equality decision procedure for terms $\_\stackrel{?}{=}\_$ to check whether the ith context entry is equal to t.

```
InContext : {n : ℕ} (Γ : Context n) (t : Expr n) → Set
InContext Γ t = ∃λ i → lookup i Γ ≡ t
inContext? : {n : ℕ} (Γ : Context n) (t : Expr n) → Dec (InContext Γ t)
inContext? Γ t = any? (λ i → lookup i Γ ≟ t)
```

In our *assumptionTactic*, we use a **with** pattern match to make a case distinction based on the decision from *inContext*?. If the type t is not found, we can simply return $\top$ value *tt*. If it is found at position i, we essentially want to return the ith entry in the context but we need to convince Agda that it has the desired type.

```
assumptionTactic {n} {t} {Γ} tyΓ tyt asmpts with inContext? Γ t
assumptionTactic {n} {t} {Γ} tyΓ tyt asmpts | yes (i , eqΓid) =
  let tyvari : Γ ⊢ var i : t
      tyvari = substJudgementType eqΓid typeVar
  in castInterp (typesAreSets tyΓ tyvari) tyt
       tyΓ asmpts (interp tyvari tyΓ asmpts)
assumptionTactic {n} {t} {Γ} tyΓ tyt asmpts | no _ = tt
```

The first step is to use the proof $eq_{\Gamma id}$ that *lookup i* $\Gamma \equiv t$ from *inContext*? and the *typeVar* typing rule to produce a proof ty_{vari} of judgement $\Gamma \vdash var\ i : t$. We can then obtain the interpretation of the ith variable through the value *interp* ty_{vari} ty_Γ *asmpts*. Unfortunately, that value's type is

```
interpSet (typesAreSets tyΓ tyvari) tyΓ asmpts
```

What we need is a value of type *interpSet* ty_t ty_Γ *asmpts*, i.e. an interpretation of the same type t, but for a different proof that t is a set. *castInterp*, an omitted special case of property *castInterp*–$\approx$ from Figure 4.9, is precisely what we need to cast one to the other.

Tactic usage Currently, our tactics can be manually invoked with a context and goal type and well-formedness proofs. The tactic invocation appears as an

expression in the code where the goal is needed. In future systems, compiler support can increase convenience by automatically providing the goal type, context and their typing proofs. This could, for example, extend Agda's experimental and underdocumented **quoteGoal** construct. This construct allows the invocation of a reflective solver with the compiler providing a syntactic representation of the goal type. It does not however provide a syntactic representation of the context or a guarantee about well-formedness of the provided type. Also, a more developed tactic API could support returning unsolved sub-goals and tactic combinators like Coq's ";".

4.5 Discussion

There are some more aspects of our approach that we believe deserve further discussion: the representation of the object language, the performance of our meta-programs, the overhead for writing meta-programs in our system and the soundness of our primitives.

Types and Guarantees Considering our example meta-programs *deriveShow* and *assumptionTactic*, an important feature of our meta-programming approach is the strong guarantees that the meta-programs' types provide, modulo the soundness of our primitives. First, meta-programs are strongly type-safe: any object code they generate must be well-typed, since they are required to provide a proof of well-typedness to the interpretation primitive. Second, our meta-language Agda checks termination and completeness of pattern matches for all function definitions to guarantee that all functions are total. This guarantee also applies to our meta-programs, so that additionally we automatically get a totality guarantee for our meta-programs. However, this does not completely exclude the use of general recursion in tactics, techniques like Capretta's partiality monad [28, 50] can be used to model such algorithms.

The representation Meta-programming implies the syntactic analysis and construction of source code and/or types, and we have chosen a fairly well-understood representation to support this: a lambda calculus with de Bruijn indices and a standard separate encoding of typing judgements. However, many different encodings are equally possible, such as those based on more advanced representations of binders [35]. It is future work to investigate the advantages that these alternatives might offer for our purposes. We also want to investigate merging *interpSet* and *interp*, but we cannot currently try this for technical reasons. Finally, we currently represent typing judgements externally,

i.e. as a property that can be true or not for an untyped lambda term. This corresponds to standard presentations of type theory, but it may be interesting to explore the benefits of an internal encoding (defining terms in such a way that only well-typed terms can be represented) like Danielsson, Chapman or McBride's [48, 32, 144] in our setting.

Performance We do not currently consider our implementation practical, because of performance reasons. For example, type-checking just the *deriveShow* example for the type of natural numbers currently takes about 2 minutes and 3GB of memory on our system. Such performance likely prohibits all practical applications. However, we do not think this bad performance is inherent to our approach, but rather a consequence of the inefficient call-by-name execution strategy that Agda uses during type-checking. Remember how we previously defined *showNat* using our *deriveShow* function. As we mentioned, *showNat* is definitionally equal to the generated program $showNat' = foldℕ\ (\lambda\ \_ \to String)\ \texttt{"zero"}\ (...)$. Nevertheless, applying *showNat* to the numbers 0 and 1 under Agda's evaluator (which is also used during type-checking) takes 2.5 resp. 11 minutes while for $showNat'$, it is instantaneous for numbers up to at least 100. For larger numbers, *showNat* quickly runs out of memory.

This behaviour is a consequence of Agda's call-by-name evaluation strategy, which repeats the normalisation of *showNat* for every reduction of $foldℕ$. If Agda were to use a more efficient strategy like call-by-need, then the normalisation of *showNat* to $showNat'$ would occur only once. Very likely, there is a lot more work being duplicated inside the normalisation of *showNat* and we believe the call-by-name evaluation strategy is responsible for the long execution and type-checking times there as well.

Overhead Writing meta-programs in our approach entails a certain amount of programming overhead. The full code of our datatype-generic meta-programming application *deriveShow* is ±1200 lines of code (including the *SimpleDT* encoding and some reusable parts). This is a lot more than what it would take to write a corresponding untyped meta-program. A significant part is the correctness proof of the meta-program (i.e. the proof that it generates correct code for all inputs).

However, a big part of our *deriveShow* implementation consists of a rather tedious proof specific to our meta-programming primitives. It concerns the correspondence of a type in a context with a general predicate P of type $A \to String$, with the value $\lambda\ \_ \to String$ provided through the interpretation of this context and the same type with an encoding of $\lambda\ \_ \to String$ already filled

in. We expect quite some work can be saved in this proof, but long compilation times have prevented further investigation. On the bright side, our assumption tactic is only about 50 lines in total, for a big part because it reuses general functions like the decision procedure for syntactic term equality. It is likely that additional reusable functions can reduce the meta-programming effort further. For example, a verified type-inference algorithm can be combined with our primitives to obviate the need for manual typing proofs in many cases.

Finally, we also expect that more experience with the definition of interpretation primitives could provide further opportunities to reduce meta-programming effort. For example, it would likely simplify some things to merge *interp* and *interpSet*, but we currently cannot do so for technical reasons. Additionally, the *irrelevant arguments* [1] that Agda support offer the potential to make Agda understand that the type correctness proofs that our primitives require are only required to exist but do not influence the result value of the primitives. We expect this could make a big difference for shortening tedious proofs like the one in our definition of *deriveShow*, where quite some work is spent on convincing Agda of properties that would directly follow if the functions used irrelevant arguments.

Soundness The soundness of our primitives remains an open question, at least if we consider the full version that does not have the unsound $\Gamma \vdash \mathit{set} : \mathit{set}$ rule that we discussed in Section 4.2. However, we do think there is a relation to the field of foundational logic that we will try to informally explain here. What we are essentially doing is reasoning about Agda terms within Agda itself. In foundational mathematical logic, Gödel's second incompleteness theorem has something to say about a similar situation for first-order logic [81]. An informal statement of the theorem (found on Wikipedia [241]) reads

Theorem 4.5.1 (Gödel's Second Incompleteness Theorem)**.** *For any formal effectively generated theory T including basic arithmetical truths and also certain truths about formal provability, if T includes a statement of its own consistency then T is inconsistent.*

A standard proof of this theorem constructs a proposition T in the object theory such that T asserts the unprovability of its own Gödel-encoding. In vague terms, it can be proven that such a term exists as soon as the object language is powerful enough to reason about natural numbers. Such a term leads to a contradiction in combination with the self-consistency proof of the theory.

It is fair to assume the theorem can be generalised to type theory, and applied to our object theory, perhaps after adding singleton types, an empty type and a type of natural numbers. Consistency of a dependent type theory is equivalent

with the non-existence of a closed term of type $\bot$. Using our primitives, it is not hard to construct a function of type $\forall\ \{t\} \to \varepsilon \vdash t : \mathit{constant\ bot}\ [\,] \to \bot$, which means that our meta-level primitives imply the consistency of our object theory. This begs the question whether Agda extended with our primitive must therefore necessarily be inconsistent, by the second incompleteness theorem, since it implies its own consistency. We conjecture that this implication is not there, for the reason that our object calculus does not contain the primitive itself, making it a fundamentally weaker theory. What we do is reminiscent of extending a first-order logical theory T with an axiom asserting T's consistency, to obtain a new theory T'. Such an extended theory T' does not in fact prove its own consistency, just that of T, so that the second incompleteness theorem does not apply. Another question that Gödel's result suggests is whether primitives like ours could in principle be implemented as normal functions within the bounds of a meta-language. Even with sufficient additional features like induction-recursion [67], this might not be possible as it would prove the language's own consistency within itself.

For these reasons, we expect that our primitives are not implementable in pure Agda but do not compromise consistency. Because of Gödel incompleteness, we think there are only two options to gain more confidence in them: either prove consistency of the extended calculus in a strictly stronger logical system such as Zermelo-Fraenkel set theory or implement our primitives in pure Agda, relying on axioms that are easier to trust than our primitives. A non-computational axiom asserting strong normalisation of the calculus (as used by Barras [12]) is a good candidate, but it isn't practical in our current implementation because Agda lacks a Prop universe like Coq's.

We think these logical aspects of our work deserve further attention. Nevertheless, even if our primitives were to be proven unsound, we do not think our work would be useless. Our application of interpretation primitives to meta-programming remains relevant as long as the primitives can be restricted to regain soundness. Also, in some applications of a dependently-typed language for programming (rather than proof checking), full certainty about soundness can be less important than powerful meta-programming support.

Staging As discussed in the introduction, our meta-programming primitives do not use the concept of staging like some other solutions [208, 222, 34, 20]. Nevertheless, our *interp* primitive performs the same function as an unquote primitive in such systems, allowing object programs to invoke a meta-program and use generated code as if it were normal code. The quote primitive in a staging meta-programming system allows to include references to object-level terms in generated code, something which we support in a different way, as

discussed in Section 4.3. Finally, while in these systems, code at all staging levels runs at either compile-time or run-time, but not both. In our system, the question of when to execute meta-programs is an orthogonal matter, not different from the *partial evaluation* of normal functions. Conveniently, partial evaluation is relatively cheap in total dependently-typed languages and is, for example, well supported in the language Idris [23].

We see the orthogonality of our meta-programming primitives w.r.t. staging considerations as an advantage. If desired, it is technically possible to require at compile-time that all invocations of the primitives be unfoldable (producing errors if arguments are not statically known). However, like for partial evaluation, executing a meta-program upfront is not *always* a good idea, especially if we are already sure that the generated code will be well-typed (see, for example, Chlipala's arguments about the performance advantages of reflective meta-programs [37]). It seems that annotations for partial evaluation as in Idris would combine well with our primitives to conveniently let the programmer control when meta-programs are executed. For example, a version of *deriveShow* with the $SimpleDT\ A$ argument annotated as $[static]$ would generate *show* functions at compile time instead of run-time.

4.6 Related work

In the literature, we find different forms of programming language support for meta-programming. We discuss them according to the guarantees that are provided about object programs.

Many approaches represent code in an untyped way, i.e. without guarantees that the represented source code is well-typed. These techniques have no way of providing strong type-safety of meta-programs, i.e. a guarantee that all the code a meta-program will ever produce is well-typed. In this category, we include approaches that represent code textually, like parser generators [109, 176], C macro's, eval primitives like JavaScript's [196], Java's pluggable annotation processors [54] (at least on the output side). Some approaches generate untyped bytecode [25]. Also in this category are macro approaches which receive and produce an untyped data structure representation of programs and types, like Template Haskell [208], Ltac proof tactics in Coq [56] and macro systems in Lisp-related languages (e.g. Racket [224]). Some provide specific language features for working with such representations. These systems provide the power of meta-programming at a comparatively low cost, but they make it hard to provide upfront guarantees of (strong) type-safety.

Not all meta-programming approaches are based on an explicit syntactic representation of terms or types. Some exploit type system features like Haskell type classes [132], Coq canonical structures [86] or C++ templates [2] to analyse types and produce code as part of the type inference process. These features provide (intentionally or not) a form of type-level computation with at least a notion of type analysis and structural recursion. Gonthier et al. even exploit canonical structures (non-trivially) to obtain a form of syntactic pattern matching and non-determinism with backtracking [86]. Meta-programming systems based on such primitives only support analysing types (but dependent types in Coq may contain terms). The computational model of these primitives is quite different from the underlying language's (unification-based vs. functional), so that meta-programming requires special expertise and techniques. For canonical structures, the computational model is not so well understood [85] and the resulting meta-programs are tightly coupled to the precise behaviour of the inferencer. An advantage of using primitives exposed by the type inferencer is that strong type-safety can be guaranteed comparatively easily [132, 86]. The type class instance search always terminates (with common extensions), but not so for C++ templates and Coq canonical structures. Completeness of pattern matching is not statically checked in any system. More or less in this category, we also have Chlipala's language Ur, which provides value-level folder functions for record types to support a practical form of meta-programming [36] with a form of syntactic analysis of record types, no explicit representation of object code and a functional computational model. Syntactic analysis of terms or general types is not supported.

Other approaches to meta-programming with strong type safety are based on explicit typed representations of code. This requires a powerful meta-language type system, as determined by the complexity of the object language and whether terms, types and typing contexts can all be syntactically constructed and analysed or only some of those. We discuss the related work according to the type system feature used in this representation.

Rudolph and Thiemann represent typed JVM bytecode generators in the Scala Mnemonics library [199], exploiting various features of Scala's type system. Taha and Sheard [222], Chen and Xi [34], Pašalić and Linger [177] and Sheard and Pašalić [209]'s systems are based on GADTs or explicit type equality proofs. Terms of a non-dependently-typed object language are syntactically represented as values of a data type indexed with the meta-level type of the term they represent. Without analysis of types, these techniques appear unsuitable for applications like proof tactics.

In VeriML, Stampoulis and Shao [214, 215] use a contextual type system, inspired by Beluga [186] and Delphin [189], in the meta-language to model a dependently-typed object language. They provide a syntactic model of terms

and types, with a certain level of support for parameterising over and pattern matching on typing contexts. Nevertheless, contexts do not seem first class in VeriML's type system. For example, tactics cannot have contexts as their return type, so meta-programs cannot construct them, only start from the ones they receive and extend them locally. Stampoulis and Shao use an imperative meta-language with general recursion because certain tactics use algorithms that are inherently imperative. We agree that such tactics exist, but we do not see why they cannot be modelled in a pure and/or total functional setting like ours, using models like those found in the literature [50, 124, 119]. VeriML tactics are partial: they can fail or loop forever. This has modularity disadvantages: if a tactic t_1 invokes another tactic t_2, then t_1's author cannot be sure that t_2 will actually succeed when it is invoked at t_1's run-time. Stampoulis and Shao partially solve this with a `letstatic` staging construct that forces tactic t_2 to be evaluated at t_1's compile time instead. This works under certain restrictions on t_2's arguments. Because our tactics' types imply termination guarantees by default, we do not need such a system, while potential non-termination can still be modelled, e.g. using the non-termination monad [50]. Stampoulis and Shao link a proof assistant's type checker with custom tactics to obtain the effect of a sound user-extensible conversion rule in the logic [215], allowing a term t of type A to be used at type A' if the equality decision procedure (potentially a custom tactic) can find a proof that $A = A'$. This form of automatic triggering of tactics for solving constraints is interesting and could perhaps be combined with our work as well.

In a dependently-typed meta-language, it is possible to model non-dependent object languages with standard inductively-defined universes using the technique of reflection [17, 37]. Altenkirch and McBride [5] and Chapman et al. [33] provide syntactic models of data types, together with interpretation functions. Chapman et al.'s universe even describes itself as a data type. These authors do not consider syntactic models of terms or types that are not data types. Brady and Hammond [20] provide a universe that models a non-dependent object language. Terms, types as well as contexts are modelled and can be syntactically constructed and analysed.

This universe-based approach can be extended to dependently-typed object languages using the advanced type-theoretic concept of inductive-recursive definitions [67]. This has been studied by Danielsson [48], Chapman [32] and McBride [144]. These authors provide typed syntactic models of dependently-typed calculi in dependently-typed calculi, with different objectives than ours. Where we focus on the applicability of such a model in meta-programming primitives, they aim to prove properties of the modelled language in the meta-language. They use models based on advanced type-theory features like induction-recursion and mutual induction. All three authors use a model

of the object calculus with terms indexed by encodings of their types, instead of an external typing judgement like ours. The models that they use are specifically tailored to enable proofs of deep properties like normalisation, and it is unclear if their models also fit our more practical objectives. Finally, these approaches generally try to stay within the limits of the features of an existing dependently typed language (albeit one with powerful features like inductive-recursive definitions). They try hard to fit their models and interpretation functions (more or less equivalent to the normalisation proof of the object language) in a known inductive-recursive schema, not fully successfully [48, 32]. McBride's encoding is accepted by Agda but he has to significantly limit the dependent nature of his object language [144]. As discussed in the introduction, our use of interpretation *primitives* allows us to side-step the interesting but hard problems that these authors tackle, leaving us free to study the application of related techniques to concrete meta-programming applications. It also allows us to use a more conventional encoding of the object language based on external typing judgements.

4.7 Conclusion

Our primitives present a novel meta-programming model with several desirable characteristics. Our meta-programs use the same functional style and well-understood computational model as normal programs. They can be given precise types that guarantee termination and strong type-safety. Finally, they can construct and analyse terms, types and typing contexts in a type-safe way. Our proof-of-concept applications in the two important application domains of datatype-generic programming and tactics, demonstrate the generality of our approach. Still, we feel this work is only a first exploration of a new approach to meta-programming. Quite some interesting questions remain to be answered in future work.

4.8 Acknowledgements

This research is partially funded by the Research Foundation — Flanders (FWO), and by the Research Fund KU Leuven. Dominique Devriese holds a Ph.D. fellowship of the Research Foundation — Flanders (FWO).

Chapter 5

Generalising Capability Safety to Effect Polymorphism

Functional reasoning in an object-oriented world

Publication data

Dominique Devriese, Frank Piessens. Generalising Capability Safety to Effect Polymorphism — Functional reasoning in an object-oriented world. Draft, 2014.

Abstract

Static effectful APIs and global state in object-oriented programming languages make it hard to modularly control effects. Object-capability (OC) languages solve this by enforcing that effects can only be triggered by components that hold a reference to the object representing the capability to do so. We study this encapsulation of effects through a formal translation to a typed functional calculus with higher-ranked polymorphism (we use a subset of Haskell for presentation). Based on an informal view of effect-polymorphism as the fundamental feature of OC languages, we translate an OC calculus to effect-polymorphic Haskell code, i.e. computations that are universally quantified over the monad in which they produce effects. The types of our translations assert the object-capability property and we can show and exploit this using Reynolds' parametricity theorem. This text contains a simple OC calculus, the formal

translation to effect-polymorphic Haskell code, proofs that effect parametricity generalises and strengthens capability-safety, some applications and discussions. An important new insight is that current OC languages and formalisations leave one effect implicitly available: the allocation of new mutable state; adding a capability for it has important theoretical and practical advantages. In summary, we propose effect parametricity (parametricity applied to effect-polymorphic functions) as a formal property that generalises and strengthens capability-safety. This establishes a new link between object-capability languages and the well-studied fields of functional programming and denotational semantics.

5.1 Introduction

Object-oriented (OO) programming is a paradigm that is important in theory, practice and teaching of programming and programming languages. Especially in large systems with complex interaction patterns and components from several parties, an object-oriented design can fill the need for strong modularity. A key feature for this is encapsulation, a property that restricts the parts of the code that can access and modify an object's state variables. By and large, the property asserts that such variables can only be read or written from the object's own methods, not from elsewhere.

Encapsulation can be seen as a discipline imposed on the use of store side-effects. It enables a form of modular reasoning; if we ascertain that an object's own methods treat its state variables in a certain way, we are sure that any program will access them that way since other code can access the object's state only through its methods. In some languages, this is true even in the presence of malicious code that actively tries to subvert encapsulation. Enforcing invariants about objects' state variables is a common and useful application of encapsulation.

However, in most OO languages no similar discipline is imposed for other kinds of side-effects than the interaction with object state. For example, in Java or C++, any code may read files or open network connections when executed. Even when we intend such effects to only be produced in specific parts of the code, there is no way to enforce this restriction without inspecting all components. Generally speaking, modular control over effects is not available in such languages.

A notable exception are object-capability languages [152, 148, 216, 191, 151]. These make the *capabilities* to perform certain side-effects explicit as objects. They guarantee that effects can only be produced by invoking the methods of such an object. For example, if we have a static method like the following:

```
int calculateSth(Logger l) = ...;
```

then it can only produce effects by invoking methods of its argument object l. An object-capability language has no static methods that allow the function to perform effects like reading a file or connecting to the network directly. As a result, if a component possesses the only reference to a capability (for example, because the *main* method gets the only reference to primitive capabilities as an argument), then it can control which parts of the code get to perform the effect and which do not. Code with a reference can pass on the capability to other code that it invokes and wrapper objects can be used to restrict or subdivide capabilities or enforce properties.

This modular control over effects is very useful. It can be used to enforce architectural properties of an application (e.g. all network access is encrypted, the front-end can only access the database through the business layer), to securely isolate mutually distrusting components in a web page (e.g. an ad in an online mail application cannot inspect e-mails [131]) or to enforce functional properties (e.g. all file access uses a transaction journal).

Motivated by the security applications of OC languages (specifically component isolation in JavaScript), Maffeis et al. have proposed the first formalisation of capability-safety (the key property of object-capability languages) [131]. For an abstract language with a certain form of small-step operational semantics, they define two language properties: authority-safety and capability-safety. Both require that for a given term in a given run-time heap, there should be an upper bound on the memory it can access and that this bound is preserved by evaluation steps. The second property additionally requires that such a bound is given by the set of addresses that the term (directly or indirectly) references. As a limitation, the formalisation only supports one kind of effects: read and write access to mutable heap variables. Also, the definitions are tied to an operational semantics and are themselves rather operational in nature, making it hard to derive, for example, in-language equivalences.

In this paper, we give a more denotational account of object capabilities. From a helicopter perspective, our main idea is that the use of object capabilities introduces what we call *effect polymorphism*. A function or method that can only produce effects by invoking methods of objects received as arguments, is necessarily polymorphic in its side-effects; by invoking the function with objects producing certain effects, we can make the function produce those effects (and only those).

To make our informal notion of effect polymorphism more precise, we use a model of effects well-known from the functional programming (FP) community: monads [156, 232]. We present a translation of capability-safe object-oriented code into Haskell functions whose effect polymorphism is explicit in their type. Those types are of the form $\forall m.Monad\ m \Rightarrow ... \rightarrow m\ (...)$, declaring that a function can be executed in an arbitrary monad m, producing effects in that monad. For example, the above example would be translated to a function of the following type:

$$calculateSth :: \forall m.Monad\ m \Rightarrow Logger\ m \rightarrow m\ Int$$

It turns out that such effect-polymorphic types yield a good denotational interpretation of code in an object-capability language and existing reasoning techniques make the property exploitable in a well-understood theoretical framework. Let us take a closer look.

Haskell's parametric polymorphism implies that a function inhabiting a type of the form $\forall m.Monad\ m \Rightarrow ... \rightarrow m\ (...)$ is restricted in its behaviour. The language provides no way for the function to determine the concrete monad m it is executed in. Therefore, whatever the function does, it must be able to do in an arbitrary monad m. The function implicitly has access to the standard monadic operators but these do not allow it to produce effects. Therefore, the only way for it to produce effects is to use the operations it can invoke through its arguments. Our example function *calculateSth* could, for example, call the *log* method on its logger argument. We will demonstrate that this corresponds precisely to the restrictions that exist in object-capability languages.

Under our proposal, the language-level property of all methods and functions in the object-capability language corresponds to a property implied by the type of their Haskell translations. In Haskell, powerful reasoning principles are available to exploit this. Specifically, polymorphic types imply properties about their inhabitants and this is formally stated using Reynolds' parametricity property [192, 231]. Parametricity for types universally quantified over a monad, has been studied by Voigtländer [228] and Oliveira et al. [169]. We will show that the parametricity of the translations of our object-oriented calculus is a more general and stronger formulation of the calculus' object-capability. Like Maffeis et al., we can derive results about the memory locations that a piece of code can access, but we also prove a non-memory-access control property: the preservation of object invariants.

However, the benefits of our account of object-capability do not end there. Our translation also teaches valuable lessons for the design of programming languages and APIs, both on the OO side and the FP side. On the OO side, we show how a well-known FP technique for the embedding of side-effecting code in a pure calculus can be extended to provide the notion of *local capabilities*, which can be given to a function without allowing it to store or otherwise leak them. We propose a simple OO type system extension to make this technique available in an object-capability language and highlight a relation to a technique from the field of ownership typing. On the FP side, the use of effect polymorphism is a valuable design pattern for the modular composition of effectful components. FP languages are sometimes said to be especially suitable for compilers, with characteristically simple effect-level component interactions (the collaboration between parser, type-checker, optimizer, code-generator etc. is often just sequential execution of the components). A general design pattern for the modular composition of effects could extend the application domain of FP towards more complex effectful architectures. Finally, our work can also be seen as a new kind of denotational semantics for OO languages and it is useful for the design of OO-to-FP foreign language interfaces and for the implementation of a new kind of hybrid OO-FP languages.

Contributions The contributions of this text are:

- The novel view of effect polymorphism as the crucial property of object-capability languages.
- A sound, semantics-preserving translation of an OO calculus to effect-polymorphic Haskell code, i.e. functions that are universally quantified over the monad in which they produce effects.
- Effect parametricity of the translations as the semantic property through which capability-safety can be formulated and exploited. This moves away from traditional presentations of capability safety as a property of the reference graph and generalises the property in two directions: (a) the property's formulation is not restricted to a single type of effects and (b) it allows proving properties beyond access control over memory locations.
- The identification of mutable state allocation as a remaining effect that is kept implicit in current OC languages.
- A discussion of applications and consequences of the above, such as the back-translation of a Haskell technique for local capabilities.

The above establishes a new and exciting, formal and practical link between object-capabilities and the fields of functional programming and denotational semantics.

Outline In this chapter, we will first define a simple OO calculus (Section 5.2), present our translation of the calculus to effect-polymorphic Haskell code (Section 5.3), provide some background on Reynolds' parametricity property (Section 5.4) and show formally that the parametricity of our translations implies more traditional object-capability properties for a store (Section 5.5). Next, we discuss how to treat the mutable state allocation effect in either the traditional implicit or a novel explicit way (Section 5.6), show an application of effect parametricity beyond memory access control: object invariants (Section 5.7) and discuss a back-translation of functional techniques to provide local capabilities (Section 5.8). Finally, we discuss and provide an overview of related work in Section 5.9 and conclude in Section 5.10. Proofs of the theorems are in Appendices 5.A and 5.B..

5.2 A calculus with objects

We start with a simple imperative calculus with objects. It has a Java-like syntax, a simple static type system and includes only interfaces, objects implementing them and static methods. For simplicity, we do not include all features that are required or desirable for an actual OO language (such as inheritance, classes, subtyping), but just those that we need for our presentation. In Section 5.9.2, we discuss extensions to more realistic OO languages.

The following is an example program in our model language, defining an interface *Logger* with one method called *log* that takes a *String* and returns *void*. Additionally, there is a static method called *doStuff* taking a *Logger* and returning void. The method logs a message by invoking a method on the provided logger and returns the *void* result of the *log* call.

```
interface Logger { void log(String); }
void doStuff(Logger l) = l.log("Hello world?");
```

Objects are constructed using a simple **new** construct, reminiscent of Java's anonymous inner classes:[1]

```
Logger nullLogger(void _) = new Logger { log(_) = void; };
```

We model instance state as interaction of the object's methods with a mutable store variable, which are accessed through objects implementing a State_t interface for some type t. The following example defines dynamic logger objects with an object reference of type *Logger* as instance state, which they interact with through an object implementing the primitive $\text{State}_{\text{Logger}}$ interface. This interface has *put* and *get* methods of appropriate types and it is the only way to access mutable state in our language.

```
interface DynamicLogger { void log(String);
                          void setLogger(Logger); }

DynamicLogger dynamicLogger(State_Logger logger) =
  new DynamicLogger {
    log(msg) = let lg = logger.get(void) in lg.log(msg);
    setLogger(l) = logger.put(l);
    }
```

New mutable state variables can be allocated using a primitive object *alloc* implementing the Alloc interface. This primitive interface has methods

[1] *nullLogger* has a **void** argument because static methods (as well as interface methods) must have precisely one argument for simplicity.

$$
\begin{array}{rll}
P & ::= & \overline{id}; \overline{sm}; & \text{(Programs)} \\
id & ::= & \textbf{interface}\ I_v\ \{\overline{t\ m(t);}\} & \text{(Interface Definitions)} \\
t & ::= & t_{base} \mid I & \text{(Types)} \\
t_{base} & ::= & \text{bool} \mid \text{void} \mid ... & \text{(Base types)} \\
I & ::= & \text{State}_\text{t} \mid \text{Alloc} \mid I_v & \text{(Interface names)} \\
s & ::= & \textbf{let}\ x = s\ \textbf{in}\ s \mid \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s & \text{(Statements)} \\
 & & \mid p(e) \mid e.m(e) \mid e & \\
e & ::= & x \mid v_{base} & \text{(Expressions)} \\
v & ::= & \textbf{new}\ I\ \{\ \overline{m(x) = s;}\ \} \mid \text{true} \mid \text{false} \mid \text{void} \mid ... & \text{(Base values)} \\
sm & ::= & t_r\ p(t\ x) = s & \text{(Static methods)}
\end{array}
$$

Figure 5.1: Syntax of our imperative OO calculus.

State_t $\text{alloc}_t(t\ v_0)$ which allocates a new reference cell of type t with a specified initial value v_0. We demonstrate its use in the following example:

```
DynamicLogger newDynamicLogger(Alloc alloc) =
  let nl = nullLogger(void) in
  let s = alloc.alloc_Logger(nl) in dynamicLogger(s)
```

Note that the *newDynamicLogger* method allocates a fresh mutable state variable and ensures that only the object is given access to it. This corresponds to the rules governing instance state in OO language, where instance state is often only accessible from inside an object's own methods by default.

For simplicity, our calculus does not include interface inheritance/subtyping, so *DynamicLogger* is not actually a subtype of *Logger*. However, subtyping can be modeled using explicit upcasting, as demonstrated by the following *DynamicLogger*.

```
Logger upcastDynamicLogger(DynamicLogger dl) =
  new Logger { log(msg) = dl.log(msg); }
```

Figure 5.1 presents the syntax of our calculus. Programs P contain a list of interface definitions $\overline{id}$ and static methods $\overline{sm}$. Types are base types t_{base} or interfaces I. Interfaces are either user-defined interfaces I_v or the primitive interfaces State_t and Alloc that we saw before. Pure expressions are distinguished from (potentially side-effecting) statements.

The typing rules in Figure 5.2 depend on contexts $\mathcal{I}$ and $\mathcal{P}$ for tracking defined interfaces and static methods respectively. $\mathcal{I}$ maps user-defined interface names

Expressions

$$\frac{\Gamma(x) = t}{\mathcal{I}, \mathcal{P}, \Gamma \vdash_e x : t} \qquad \frac{\vdash_{base} v_b : t_b}{\mathcal{I}, \mathcal{P}, \Gamma \vdash_e v_b : t_b}$$

$$\frac{methods_{\mathcal{I}}(I) = \overline{t_r\ m(t)} \qquad \overline{\mathcal{I}, \mathcal{P}, \Gamma[x : t] \vdash_s s : t_r}}{\mathcal{I}, \mathcal{P}, \Gamma \vdash_e \textbf{new}\ I\ \{\overline{m(x) = s}\} : I}$$

Statements

$$\frac{\mathcal{I}, \mathcal{P}, \Gamma \vdash_s s_1 : t_1 \quad \mathcal{I}, \mathcal{P}, (\Gamma, x : t_1) \vdash s_2 : t_2}{\mathcal{I}, \mathcal{P}, \Gamma \vdash_s \textbf{let}\ x = s_1\ \textbf{in}\ s_2 : t_2} \qquad \frac{\mathcal{I}, \mathcal{P}, \Gamma \vdash_e e : t}{\mathcal{I}, \mathcal{P}, \Gamma \vdash_s e : t}$$

$$\frac{\mathcal{I}, \mathcal{P}, \Gamma \vdash_e e : \text{Bool} \quad \mathcal{I}, \mathcal{P}, \Gamma \vdash_s s_1 : t \quad \mathcal{I}, \mathcal{P}, \Gamma \vdash_s s_2 : t}{\mathcal{I}, \mathcal{P}, \Gamma \vdash_s \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 : t}$$

$$\frac{\mathcal{I}, \mathcal{P}, \Gamma \vdash_e e : I \quad \mathcal{I}, \mathcal{P}, \Gamma \vdash_e e' : t' \quad (t\ m(t')) \in methods_{\mathcal{I}}(I)}{\mathcal{I}, \mathcal{P}, \Gamma \vdash_s e.m(e') : t} \qquad \frac{(t\ p(t')) \in \mathcal{P} \quad \mathcal{I}, \mathcal{P}, \Gamma \vdash_e e : t'}{\mathcal{I}, \mathcal{P}, \Gamma \vdash_s p(e) : t}$$

Static methods

$$\frac{\mathcal{I}, \mathcal{P}, [x : t] \vdash_s s : t_r}{\mathcal{I}, \mathcal{P} \vdash_{sm} t_r\ p(t\ x) = s;}$$

Programs

$$\frac{}{\mathcal{I}, \mathcal{P} \vdash_P \epsilon} \qquad \frac{\mathcal{I}, \mathcal{P} \vdash_{sm} t_r\ p(t\ x) = s; \quad \mathcal{I}, (\mathcal{P} \cup \{t_r\ p(t)\}) \vdash_P \overline{sm}}{\mathcal{I}, \mathcal{P} \vdash_P t_r\ p(t\ x) = s; \overline{sm}}$$

Figure 5.2: Typing rules for statements and programs

Evaluation states

$$
\begin{array}{rll}
ES ::= & (S, s_r) & \text{(Evaluation states)} \\
v_r \in V_r ::= & \cdots \mid o & \text{(Runtime values)} \\
e_r \in E_r ::= & \ldots & \text{(Runtime expressions)} \\
s_r \in S_r ::= & \ldots & \text{(Runtime statements)} \\
o ::= & a \mid \text{alloc} & \text{(Primitive objects)} \\
a \in & Addr & \\
S ::= & Addr \hookrightarrow V_r & \text{(The Store)}
\end{array}
$$

Statement execution

$$
\begin{array}{ll}
E ::= \ \cdot \mid \textbf{let}\ x = E\ \textbf{in}\ s & \text{(Evaluation Contexts)} \\
P ::= \{\overline{p(x) = s}\} & \text{(Static Method Implementations)}
\end{array}
$$

$$
\frac{}{P \Vdash (S, E[\textbf{if}\ v\ \textbf{then}\ s_{\text{true}}\ \textbf{else}\ s_{\text{false}}]) \to (S, E[s_v])}
$$

$$
\frac{}{P \Vdash (S, E[\textbf{let}\ x = v\ \textbf{in}\ s_2]) \to (S, E[[x \mapsto v]s_2])}
$$

$$
\frac{S(a) = v}{P \Vdash (S, E[a.\text{get}(\text{void})]) \to (S, E[v])}
$$

$$
\frac{S' = S[a \mapsto v]}{P \Vdash (S, E[a.\text{put}(v)]) \to (S', E[\text{void}])}
$$

$$
\frac{a \notin \text{dom}(S) \qquad S' = S[a \mapsto v]}{P \Vdash (S, E[\text{alloc.alloc}_t(v)]) \to (S', E[a])}
$$

$$
\frac{(m(x) = s;) \in \{\overline{md}\}}{P \Vdash (S, E[(\textbf{new}\ I\ \{\overline{md}\}).m(v)]) \to (S, E\,[[x \mapsto v]\,s])}
$$

$$
\frac{(p(x) = s;) \in P}{P \Vdash (S, E[p(v)]) \to (S, E[[x \mapsto v]s])}
$$

Figure 5.3: Execution judgements.

I to their set of methods: $\mathcal{I}(I) = \{\overline{t_r\ m(t)}\}$. $\mathcal{P}$ contains a list of the static methods seen so far, with their types: $\mathcal{P} = \{\overline{t_r\ p(t)}\}$. The context Γ contains the types of variables in scope. Before type-checking, $\mathcal{I}$ is filled with the program's interface definitions. During type-checking, $\mathcal{P}$ is filled incrementally with every static method that passes type-checking. There are typing judgements for expressions $\mathcal{I}, \mathcal{P}, \Gamma \vdash_e e : t$, statements $\mathcal{I}, \mathcal{P}, \Gamma \vdash_s s : t$, static methods $\mathcal{I}, \mathcal{P} \vdash_{sm} t_r\ p(t\ x) = s$; and programs $\mathcal{I}, \mathcal{P} \vdash_P \overline{sm}$ with standard rules. The methods defined by an interface are as follows:[2]

$$\begin{aligned} \mathit{methods}_{\mathcal{I}}(I) &= \mathcal{I}(I) \\ \mathit{methods}_{\mathcal{I}}(\mathrm{State}_t) &= \{t\ \mathrm{get}(\mathrm{void}), \mathrm{void}\ \mathrm{put}(t)\} \\ \mathit{methods}_{\mathcal{I}}(\mathrm{Alloc}) &= \{\mathrm{State}_t\ \mathrm{alloc}_t(t) \mid t\ \mathrm{type}\} \end{aligned}$$

Note that the typing rules do not allow static methods to be recursive: the typing judgement for a static method p does not have p itself in $\mathcal{P}$. This restriction allows us to translate to a calculus without general recursion later on.

In Figure 5.3 we define evaluation judgements. We use an evaluation state ES consisting of a store and a runtime statement (S, s), where a runtime statement is a statement that may additionally contain runtime-only values o. Runtime addresses a represent a mutable variable reference in the store, typed State_t for the appropriate t. Value alloc is the canonical instance of the Alloc interface through which new reference cells can be allocated. A store S is a partial map of addresses to runtime values. The small-step reduction rules for statements $P \Vdash (S, s) \to (S', s')$ use evaluation contexts. They refer to a set of static method implementations $P = \{\overline{p(x) = s}\}$, collected from the program text. The method calls on our primitive objects a and alloc form the only interactions with the store. get and put calls on a mutable variable reference a read or write the value at address a and an alloc_t call on primitive object alloc allocates and initializes a new reference cell. Method invocations $e.m(e')$ where e evaluates to a non-primitive object **new** I $\{\overline{md}\}$ evaluate to the implementation of method m in $\overline{md}$ with the method argument substituted.

The calculus is generally well-behaved and type-safe. We show progress and preservation results in Appendix 5.A.

[2] We avoid polymorphism in the calculus by giving Alloc an infinite number of methods. Note that this is not allowed for user-defined types.

5.3 Translating objects to Haskell

The most important tool in this paper is a translation of our OO calculus to a functional calculus. We use Haskell as a well-known, readable and type-inferenced notation for System Fω [80], extended with (nominal) recursive types.

Our calculus is capability-safe. Side-effects can only be triggered by invoking the methods of an object. It is lexically scoped and references cannot be forged. The key property of our translation is that we maintain these good qualities and make them explicit in the Haskell type system. Let us explain how this works.

Consider the *doStuff* method we've seen previously:

```
interface Logger { void log(String ); }
void doStuff(Logger l) = l.log("Hello world?");
```

We use monads to model the imperative code. However, it is not enough to translate our methods to the IO *Monad* like this:

$$\begin{aligned}&\textbf{data}\ \mathit{Logger} = \mathit{Logger}\ \{\mathit{log} :: \mathit{String} \to \mathit{IO}\ ()\}\\&\mathit{doStuff} :: \mathit{Logger} \to \mathit{IO}\ ()\\&\mathit{doStuff}\ l = \mathit{log}\ l\ \texttt{"Hello world?"}\end{aligned}$$

Such a translation does maintain the semantics of the code, but the Haskell type of *doStuff* becomes $\mathit{Logger} \to \mathit{IO}\ ()$, a type that allows much more effects than those that can be triggered through l. From its implementation, we can still see that *doStuff* only performs effects through l, but the type permits more. In fact, this translation could just as well support a non-capability-safe calculus, showing that the types are not very restrictive.

A better solution is to use a technique that has been described and studied by Voigtländer [228] and Oliveira et al. [169], They use types that are universally quantified over the monad in which effects are produced. Concretely, our *Logger* is translated as follows:

$$\begin{aligned}&\textbf{data}\ \mathit{Logger}\ m = \mathit{Logger}\ \{\mathit{log} :: \mathit{String} \to m\ ()\}\\&\mathit{doStuff} :: \forall m.\mathit{Monad}\ m \Rightarrow \mathit{Logger}\ m \to m\ ()\\&\mathit{doStuff}\ l = \mathit{log}\ l\ \texttt{"Hello world?"}\end{aligned}$$

doStuff's type now requires it to support an arbitrary monad m. The function is a computation in m, but can only produce effects in m through the primitives it receives as part of the object $l :: \mathit{Logger}\ m$. Effectively, this means that the

function can only produce effects through methods of argument objects, i.e. the type states the OC property we want.

This connection is not just informal: the restrictions implied by the universal quantification over m can be made formal and precise using Reynolds' parametricity theorem [192]. It states that any polymorphic type implies certain properties about its inhabitants. The properties can be derived mechanically from the type. Wadler has used the slogan *Theorems for free!* to refer to such properties [231]. Our types are complicated by the quantification over the monad m of rank $* \to *$ and the *Monad* constraint on it, but Voigtländer has explained in detail how parametricity can be applied for such types. As the polymorphism of our types is over the monad m in which the computation produces effects, we use the terms *effect polymorphism* and *effect parametricity*.

The value l of type *Logger* m corresponds to the *Logger* object in the object-oriented code. Like that object, it plays a dual role: on the one hand, it serves as a witness value proving that a certain capability is available (in OO terminology) or that the monad m supports certain primitive operations (in a more functional terminology). On the other hand, the value/object also provides the implementation of the capability, i.e. its method implementations specify how the capability is executed when it is invoked. On the functional side, we can say that the value l of type *Logger* m contains implementations of the primitive *Logger* operations for the (otherwise unknown) monad m. Understanding the object/value in this second role, it is important to notice that we can have multiple implementations of a single capability, which implement the primitive operations of the capability in different ways.

In general, it is this effect parametricity that we propose as a generalisation of capability safety. The property is not coupled to a specific type of effects and we will show in Section 5.5 that it generalises capability safety.

First, let us look at our translation, defined formally in Figure 5.4.

Interfaces and methods We translate interfaces to monad dictionaries like *Logger* above. A user-defined interface *Intf* corresponds to a data type $\lfloor \mathit{Intf} \rfloor_I^{\mathcal{I}}$ containing the methods as fields. The data type is parameterised over a monad m, in which the results of the methods are produced. Argument and result types ty are translated to Haskell types $\lfloor ty \rfloor_t^{\mathcal{I}}\, m$, in which all interface types are replaced by their translations for monad m and base types translated to their Haskell counterparts.

The primitive interfaces Alloc and State_t correspond to the following *AllocD* and *StateD* monad dictionaries:

Interfaces:

$$\begin{aligned}
\lfloor \text{Alloc} \rfloor^{\mathcal{I}}_{I}\ m &= \mathit{AllocD}\ m \\
\lfloor \text{State}_t \rfloor^{\mathcal{I}}_{I}\ m &= \mathit{StateD}\ (\lfloor t \rfloor^{\mathcal{I}}_{t}\ m)\ m \\
\lfloor I \rfloor^{\mathcal{I}}_{I}\ m &= I\ m \qquad\qquad \text{when } \mathcal{I}(I) = \overline{t_r\ md(t)} \\
&\text{where } \mathbf{data}\ I\ m = I \left\{ \overline{md :: \lfloor t \rfloor^{\mathcal{I}}_{t}\ m \mapsto m \left(\lfloor t_r \rfloor^{\mathcal{I}}_{t}\ m \right)} \right\}
\end{aligned}$$

Types:

$$\begin{aligned}
\lfloor t_{base} \rfloor^{\mathcal{I}}_{t}\ m &= t_{base} \\
\lfloor I \rfloor^{\mathcal{I}}_{t}\ m &= \lfloor I \rfloor^{\mathcal{I}}_{I}\ m
\end{aligned}$$

Static methods:

$$\lfloor t_r\ p(t\ x) = s; \rfloor^{\mathcal{I}}_{sm} = \begin{cases} p :: \forall\ m.\ \mathit{Monad}\ m \Rightarrow \lfloor t \rfloor^{\mathcal{I}}_{t}\ m \to \lfloor t_r \rfloor^{\mathcal{I}}_{t}\ m \\ p = \lambda x \to \lfloor s \rfloor^{\mathcal{I}}_{s} \end{cases}$$

Statements:

$$\begin{aligned}
\lfloor \mathbf{let}\ x = s_1\ \mathbf{in}\ s_2 \rfloor^{\mathcal{I}}_{s} &= \lfloor s_1 \rfloor^{\mathcal{I}}_{s}\ \gg\!= (\lambda x \to\ \lfloor s_2 \rfloor^{\mathcal{I}}_{s}) \\
\lfloor e \rfloor^{\mathcal{I}}_{s} &= \mathit{return}\ \lfloor e \rfloor^{\mathcal{I}}_{e} \\
\lfloor \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \rfloor^{\mathcal{I}}_{s} &= \mathbf{if}\ \lfloor e \rfloor^{\mathcal{I}}_{e}\ \mathbf{then}\ \lfloor s_1 \rfloor^{\mathcal{I}}_{s}\ \mathbf{else} \lfloor s_2 \rfloor^{\mathcal{I}}_{s} \\
\lfloor e.md(e') \rfloor^{\mathcal{I}}_{s} &= md\ \lfloor e \rfloor^{\mathcal{I}}_{e}\ \lfloor e' \rfloor^{\mathcal{I}}_{e} \\
\lfloor e.\text{put}(e') \rfloor^{\mathcal{I}}_{s} &= \mathit{putM}\ \lfloor e \rfloor^{\mathcal{I}}_{e}\ \lfloor e' \rfloor^{\mathcal{I}}_{e} \\
\lfloor e.\text{alloc}_t(e') \rfloor^{\mathcal{I}}_{s} &= \mathit{allocM}\ \lfloor e \rfloor^{\mathcal{I}}_{e}\ \lfloor e' \rfloor^{\mathcal{I}}_{e} \\
\lfloor e.\text{get}(_) \rfloor^{\mathcal{I}}_{s} &= \mathit{getM}\ \lfloor e \rfloor^{\mathcal{I}}_{e} \\
\lfloor p(e) \rfloor^{\mathcal{I}}_{s} &= p\ \lfloor e \rfloor^{\mathcal{I}}_{e}
\end{aligned}$$

Expressions:

$$\lfloor x \rfloor^{\mathcal{I}}_{e} = x \qquad\qquad \lfloor v_{base} \rfloor^{\mathcal{I}}_{e} = v_{base}$$

$$\left\lfloor \mathbf{new}\ I\ \{\overline{md(x) = s;}\} \right\rfloor^{\mathcal{I}}_{e} = I\ \{\overline{md = \lambda x \to \lfloor s \rfloor^{\mathcal{I}}_{s}}\} \qquad \text{when } \mathcal{I}(I) = \overline{t_r\ md(t)}$$

Figure 5.4: The translation of our OO calculus to Haskell.

$$\begin{aligned}&\textbf{data } \mathit{StateD}\ t\ m = \mathit{StateD}\ \{\, \mathit{getM} :: m\ t\\ &\qquad\qquad\qquad\qquad\quad ,\, \mathit{putM} :: t \to m\ ()\,\}\\ &\textbf{data } \mathit{AllocD}\ m = \mathit{AllocD}\ \{\, \mathit{allocM} :: t \to m\ (\mathit{StateD}\ t\ m)\,\}\end{aligned}$$

Static methods We translate static methods as discussed for *doStuff* above. In general, the translation $\lfloor t_r\ p(t\ x) = s; \rfloor^{\mathcal{I}}_{sm}$ of a static method p is a Haskell function whose type is universally quantified over a monad m. It receives arguments and produces a result value instantiated for m and the computation runs in m. The translation of a statement *stmt* is written as $\lfloor \mathit{stmt} \rfloor^{\mathcal{I}}_{s}$. A **let** statement **let** $x = s_1$ **in** s_2 is translated to a monadic bind ($\gg\!=$) that executes s_1, binds the result to x and continues with s_2. Statements that are (pure) expressions correspond to monadic *return*s, method calls project out the implementation of the appropriate method and static method invocations call the procedure's translation. The translation $\lfloor \mathit{exp} \rfloor^{\mathcal{I}}_{e}$ of an expression *exp* is unsurprising for variables, base values. The translation of a **new** expression constructs a value of the monad dictionary for the specified interface from translations of the method implementations.

As an example, recall the *nullLogger* function:

Logger nullLogger(**void** _) = **new** Logger { log(_) { **void** } }

This function is translated as follows:

$$\begin{aligned}&\mathit{nullLogger} :: \forall m.\mathit{Monad}\ m \Rightarrow () \to m\ (\mathit{Logger}\ m)\\ &\mathit{nullLogger} = \lambda\_.\ \mathit{return}\ (\mathit{Logger}\ \{\, \mathit{log} = \lambda\_.\ \mathit{return}\ ()\,\})\end{aligned}$$

The function's implementation is a pure expression, so the translation uses *return* to construct a pure computation in monad m. It returns an implementation of the *Logger* interface containing the translation of the *log* implementation.

Typing correspondence Our translation is sound in the sense that it will translate well-typed terms in the OO calculus to well-typed Haskell code. To make this precise, we use an additional translation function for contexts $\lfloor \Gamma \rfloor^{\mathcal{I}}_{ctx}\, m$ that will replace every type *ty* with $\lfloor \mathit{ty} \rfloor^{\mathcal{I}}_{t}\, m$. Similarly, we translate the static method context $\mathcal{P} = \{\overline{t_r\ p(t)}\}$ to a Haskell context $\lfloor \mathcal{P} \rfloor^{\mathcal{I}}_{\mathcal{P}} = \overline{p :: \forall m.\mathit{Monad}\ m \Rightarrow \lfloor t \rfloor^{\mathcal{I}}_{t}\ m \to m\ (\lfloor t_r \rfloor^{\mathcal{I}}_{t}\ m)}$.

We make the soundness claim about our translation function more precise by formulating the intended type correspondence. Our choice for using Haskell instead of a more principled calculus like System Fω is disadvantageous here, because it is unclear which type system we can use. A good candidate could be

Vytiniotis et al.'s constraint-based natural typing rules [229], but they do not support type variables of ranks beyond $*$. In this paper, we choose to ignore such technical problems, and choose to just formulate, without a proof, our intended typing relations in a hypothetical (natural) type system for Haskell where the judgement $C; \Gamma \vdash e :: t$ asserts that expression e is well-typed in context Γ under constraints C. Our translation is simple enough to make this sufficiently credible. If necessary, the translation could be elaborated to, for example, System Fω with recursive types, although that would imply a loss in legibility.

The following are the typing correspondences we postulate for our translation:

- If $\mathcal{I}, \mathcal{P}, \Gamma \vdash_e \mathit{expr} : \mathit{ty}$, then we have on the Haskell side that

$$\mathit{Monad}\ m; (\lfloor \mathcal{P} \rfloor^{\mathcal{I}}_{\mathcal{P}}; \lfloor \Gamma \rfloor^{\mathcal{I}}_{\mathit{ctx}}\ m) \vdash \lfloor \mathit{expr} \rfloor^{\mathcal{I}}_{e} :: \lfloor \mathit{ty} \rfloor^{\mathcal{I}}_{t}\ m$$

- If $\mathcal{I}, \mathcal{P}, \Gamma \vdash_s \mathit{stmt} : \mathit{ty}$, then we have that

$$\mathit{Monad}\ m; (\lfloor \mathcal{P} \rfloor^{\mathcal{I}}_{\mathcal{P}}; \lfloor \Gamma \rfloor^{\mathcal{I}}_{\mathit{ctx}}\ m) \vdash \lfloor \mathit{stmt} \rfloor^{\mathcal{I}}_{s} :: m\ (\lfloor \mathit{ty} \rfloor^{\mathcal{I}}_{t}\ m)$$

Semantics correspondence To show that our translation preserves the semantics of the calculus, we need a way to execute the translations. For simplicity, we use a model with a fixed heap (so no allocation) based on a *State* monad. That means that this part of our work does not support the primitive alloc object.

To do this, we use a store typing Σ (as used in the type preservation proof in Appendix 5.A) that assigns a type to all run-time addresses. We translate a store typing to a Haskell data type:

$$\mathbf{data}\ \lfloor \Sigma \rfloor^{\mathcal{I}}_{\Sigma}\ m = \mathit{Heap}\ \left\{ \overline{a :: \lfloor \Sigma(a) \rfloor^{\mathcal{I}}_{t}\ m} \;\middle|\; a \in \mathrm{dom}(\Sigma) \right\}$$

We then define the FS_{Σ} monad as a state monad carrying a heap of this type:

$$\mathbf{data}\ \mathit{FS}_{\Sigma}\ a = \mathit{FS}\ \{\mathit{unFS} :: \mathit{State}\ (\lfloor \Sigma \rfloor^{\mathcal{I}}_{\Sigma}\ \mathit{FS}_{\Sigma})\ a\}$$

We omit the *Monad* instance for FS_{Σ}, which simply lifts the standard instance for *State*. We can execute a computation in FS_{Σ} with *runFS*:

$$\begin{aligned} &\mathit{runFS} :: \mathit{FS}_{\Sigma}\ a \to \lfloor \Sigma \rfloor^{\mathcal{I}}_{\Sigma}\ \mathit{FS}_{\Sigma} \to (\lfloor \Sigma \rfloor^{\mathcal{I}}_{\Sigma}\ \mathit{FS}_{\Sigma}, a) \\ &\mathit{runFS}\ c\ s = \mathit{runState}\ (\mathit{unFS}\ c)\ s \end{aligned}$$

In this monad, we can provide primitive state dictionaries for all addresses a as follows:

$$
\begin{aligned}
&\mathit{state}_a :: \mathit{StateD}\ (\lfloor \Sigma(a) \rfloor_t^{\mathcal{I}}\ \mathit{FS}_\Sigma)\ \mathit{FS}_\Sigma \\
&\mathit{state}_a = \mathit{StateD} \\
&\quad \{\, \mathit{getM} = \mathit{FS}\ (\mathbf{do}\ \{\, s \leftarrow \mathit{get}; \mathit{return}\ (a\ s)\,\}) \\
&\quad ,\, \mathit{putM} = \lambda v.\, \mathit{FS}\ (\mathit{get} \gg\!= \lambda h.\, \mathit{put}\ (h\ \{\, a = v\,\}))\,\}
\end{aligned}
$$

We can then translate the stores and run-time values from the semantics to translations with monad m instantiated to FS_Σ:

$$
\begin{aligned}
\lfloor (S, s) \rfloor_{ES}^{\mathcal{I}} &= \mathit{runFS}\ \ \lfloor s_r \rfloor_s^{\mathcal{I}}\ \ \lfloor S \rfloor_S^{\mathcal{I}} \\
\lfloor S \rfloor_S^{\mathcal{I}} &= \mathit{Heap}\ \left\{ a = \lfloor S(a) \rfloor_{v_r}^{\mathcal{I}} \;\middle|\; a \in \mathrm{dom}(S) \right\} \\
\lfloor \mathrm{alloc} \rfloor_{v_r}^{\mathcal{I}} &= \text{not supported} \\
\lfloor a \rfloor_{v_r}^{\mathcal{I}} &= \mathit{state}_a
\end{aligned}
$$

We can then prove the following theorem.

Theorem 5.3.1 (Semantics correspondence)**.** *If* $ES \to ES'$ *and* ES *does not contain* alloc*, then* $\lfloor ES \rfloor_{ES}^{\mathcal{I}} \cong \lfloor ES' \rfloor_{ES}^{\mathcal{I}}$*.*

In Section 5.6, we will take a closer look at mutable state allocation which we have ignored here.

5.4 Parametricity

We have translated our object-oriented code to Haskell functions quantified over an arbitrary monad m because this corresponds to the properties of our object calculus, where effects can only be triggered through objects. Our translation makes this property evident in the types. Haskell then provides tools to make this property more precise. Specifically, we exploit the *parametricity* property of Haskell's polymorphism and the associated technique of deriving *free theorems*. In the next section, we recapitulate these tools, based on an explanation by Voigtländer [228].

5.4.1 Background

Parametricity is a language property that tells us something about the semantics of polymorphic functions, based solely on their type. Intuitively, when a function is polymorphic in a type t, then it is limited in what it can do with values of type t. The function must support t to be instantiated to any concrete type and there is no way for it to find out what type that is. This effectively leads

to a strong restriction for the implementation of the function and conversely provides strong guarantees about its behaviour. For example, a function f of type $\forall t.t \to t$ has so little information about the type t that it has no other choice than to return the value it receives, i.e. the type is only inhabited by the identity function $\lambda x.\, x$ (at least, in a total calculus).

Formally, (relational) parametricity can be exploited through an interpretation of types as relations. Voigtländer [228] explains that for a type like $\forall t.t \to t$, we would first replace all quantified type variables by quantified relation variables: $\forall \mathcal{R}.\mathcal{R} \to \mathcal{R}$. Then, there is a systematic way of reading such expressions over relations as relations themselves. In particular,

- Base types like *Int* are read as identity relations
- For relations $\mathcal{R}$ and $\mathcal{S}$, we have

$$\mathcal{R} \to \mathcal{S} = \{(f, g) \mid \forall (a, b) \in \mathcal{R}.(f\ a, g\ b) \in \mathcal{S}\}$$

- For types τ and τ' with at most one free variable, say α, and a function $\mathcal{F}$ on relations such that every relation $\mathcal{R}$ between closed types τ_1 and τ_2 (denoted $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$) is mapped to a relation $\mathcal{F}\ \mathcal{R} : \tau[\tau_1/\alpha] \Leftrightarrow \tau'[\tau_2/\alpha]$, we have

$$\forall \mathcal{R}.\ \mathcal{F}\ \mathcal{R} = \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} : \tau_1 \Leftrightarrow \tau_2\ .\ (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}\ \mathcal{R}\}$$

  ($u_{\tau_1} :: \tau[\tau_1/\alpha]$ is the instantiation of $u :: \forall \alpha.\tau$ to type τ_1 and similarly for v_{τ_2}. Elsewhere, type instantiation is left implicit.)

- Fixed type constructors are read as appropriate constructions on relations. For example, the list type constructor maps a relation $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ to the relation $[\mathcal{R}] : [\tau_1] \Leftrightarrow [\tau_2]$ defined by:

$$[\mathcal{R}] = \{([], [])\} \cup \{(a : as, b : bs) \mid (a, b) \in \mathcal{R}, (as, bs) \in [\mathcal{R}]\}$$

In his seminal paper about parametricity [192], Reynolds' insight was that any relation that can be constructed in this way as the interpretation of a closed type, must be the identity relation. He called this theorem the *abstraction theorem*, but it became better known as the *parametricity theorem*.

For example, for a function *id* inhabiting our example type $\forall a.a \to a$, this means that $(id, id) \in \forall \mathcal{R}.\mathcal{R} \to \mathcal{R}$. By the above, this means that for any τ_1, τ_2, $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$, $v_1 :: \tau_1$, $v_2 :: \tau_2$, $(v_1, v_2) \in \mathcal{R}$ implies that $(id\ v_1, id\ v_2) \in \mathcal{R}$. Now take $\tau_1 = \tau_2 = \tau$ and choose a fixed $v_0 :: \tau$. We can define a relation $\mathcal{R}_{v_0} : \tau \Leftrightarrow \tau$ as $\{(v_0, v) \mid v :: \tau\}$. If we then choose arbitrary $v_1, v_2 :: \tau$, we obtain that $v_1 = v_0$

implies that $id\ v_1 = v_0$. Because v_0 was chosen arbitrarily, id must be the identity function.

In order to study the parametricity properties of the types of our translated functions, we need to deal with two complications. In the type $\forall m.Monad\ m \Rightarrow Logger\ m \to m\ ()$, m does not represent a type (kind $*$), but a type constructor (kind $* \to *$). Additionally, it is not clear how to interpret the type class constraint $Monad\ m$. Luckily, precisely these extensions are treated by Voigtländer [228]. The first problem is solved by introducing relational actions $\mathcal{F}$. Similar to the interpretation of quantified type variables as relations between types instantiating the variable, a type constructor variable like m is interpreted as relational action $\mathcal{F} : m_1 \Leftrightarrow m_2$ between type constructors m_1 and m_2: a function mapping a relation $\mathcal{R} : a_1 \Leftrightarrow a_2$ between arbitrary types a_1 and a_2 to a relation $\mathcal{F}\ \mathcal{R} : m_1\ a_1 \Leftrightarrow m_2\ a_2$. Voigtländer mentions the example of $\mathcal{F} : Maybe \Leftrightarrow []$ which maps a relation $\mathcal{R} : a_1 \Leftrightarrow a_2$ to $\mathcal{F}\ \mathcal{R}$:

$$\mathcal{F}\ \mathcal{R} = \{(Nothing, [])\} \cup \{(Just\ a, b : bs) \mid (a, b) \in \mathcal{R}, bs :: [a_2]\}$$

For class constraints, Voigtländer extends Wadler's treatment of ML eq-type constraints to arbitrary type classes. He explains how the constraint in an expression like $\forall \mathcal{F}.Monad\ \mathcal{F} \Rightarrow ...$ can be read as a constraint on the relational action $\mathcal{F}$. For $\mathcal{F} : m_1 \Leftrightarrow m_2$, the constraint $Monad\ \mathcal{F}$ requires that there are $Monad$ instances for both m_1 and m_2 and additionally that their methods are related according to an $\mathcal{F}$-interpretation of their type. More specifically, for the $Monad$ type class's methods $return :: \forall a.a \to m\ a$ and $(\gg\!=) :: \forall a\ b.m\ a \to (a \to m\ b) \to m\ b$, we get the following requirements:

- $(return_{m_1}, return_{m_2}) \in \forall \mathcal{R}.\mathcal{R} \to \mathcal{F}\ \mathcal{R}$
- $((\gg\!=)_{m_1}, (\gg\!=)_{m_2}) \in \forall \mathcal{R}\ \mathcal{S}.\mathcal{F}\ \mathcal{R} \to (\mathcal{R} \to \mathcal{F}\ \mathcal{S}) \to \mathcal{F}\ \mathcal{S}$

Voigtländer defines a relational action $\mathcal{F} : m_1 \Leftrightarrow m_2$ to be a *Monad-action* when the above conditions are satisfied.

It turns out that Voigtländer's example above is not a *Monad*-action as $(\gg\!=_{Maybe}, \gg\!=_{[]})$ are not in $\forall \mathcal{R}.\forall \mathcal{S}.\mathcal{F}\ \mathcal{R} \to (\mathcal{R} \to \mathcal{F}\ \mathcal{S}) \to \mathcal{F}\ \mathcal{S}$. As an exercise, convince yourself of his counterexample: $\mathcal{R} = \mathcal{S} = id_{Int}$, $m_1 = Just\ 1$, $m_2 = [1, 2]$, $k_1 = \lambda i.$ **if** $i > 1$ **then** $Just\ i$ **else** $Nothing$ and $k_2 = \lambda i.reverse\ [2..i]$. He mentions a different relational action $\mathcal{F}'$ that *is* a *Monad*-action, defined by

$$\mathcal{F}'\ \mathcal{R} = \{(Nothing, [])\} \cup \{(Just\ a, [b]) \mid (a, b) \in \mathcal{R}\}$$

5.4.2 Effect parametricity

Voigtländer shows how to apply the above techniques to two applications, one of which is reasoning about monadic programs. For our purposes, it turns out that we can use very similar reasoning for deriving results about programs in our calculus.

Consider, for example, a static method that takes only a non-object argument:

bool isEven(**int** x) = ...

The method takes an int and returns a bool. Since it only gets access to an int, a primitive value through which no effects can be produced, the object capability of our calculus informally tells us that *isEven* must be a pure function. Using effect parametricity, we can make this property more precise.

We know that *isEven* translates to a Haskell function of type $\forall m.\mathit{Monad}\ m \Rightarrow \mathit{Int} \to m\ \mathit{Bool}$ and Theorem 5.3.1 tells us that $(S, \mathit{isEven}(n))$ evaluates to (S', v) if $\mathit{runFS}\ (\mathit{isEven}\ n)\ \lfloor S \rfloor_S^{\mathcal{I}}$ equals $\mathit{runFS}\ (\mathit{return}\ v)\ \lfloor S' \rfloor_S^{\mathcal{I}}$, i.e. $(v, \lfloor S' \rfloor_S^{\mathcal{I}})$.

Parametricity now allows us to prove that computation *isEven* must equal a pure computation $\lambda n.\ \mathit{return}\ (f\ n)$. The proof works similarly to that of Voigtländer's Theorem 1. It uses the *Id* monad:

$$\begin{aligned}&\textbf{data}\ \mathit{Id}\ a = \mathit{Id}\ \{\mathit{runId} :: a\}\\ &\textbf{instance}\ \mathit{Monad}\ \mathit{Id}\ \textbf{where}\ \mathit{return} = \mathit{Id}\\ &\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Id}\ a) \gg\!= k = k\ a\end{aligned}$$

We can then use the *Monad*-action $\mathcal{F} : FS_\Sigma \Leftrightarrow \mathit{Identity}$ between our monad FS_Σ and the *Id* monad, defined as

$$\mathcal{F}\ \mathcal{R} = \mathit{return}_{FS_\Sigma}^{-1}; \mathcal{R}; \mathit{Id}$$

where the semicolon denotes (forward) relation composition and f^{-1} gives the inverse of f's function graph. We do not repeat Voigtländer's proof that this is a *Monad*-action. It uses the left identity law for monad *FS*. Parametricity now gives us that $(\mathit{isEven}_{FS_\Sigma}, \mathit{isEven}_{Id}) \in \mathit{id}_{Int} \to \mathcal{F}\ \mathit{id}_{Bool}$, from which we can derive that $\mathit{isEven}_{FS_\Sigma}$ equals $\lambda n.\ \mathit{return}\ (\mathit{runId}\ (\mathit{isEven}_{Id}\ n))$. Therefore, we have that $\mathit{runFS}\ (\mathit{isEven}\ n)\ \lfloor S \rfloor_S^{\mathcal{I}}$ must be equal to $\mathit{runId}\ (\mathit{isEven}_{Id}\ n, \lfloor S \rfloor_S^{\mathcal{I}})$ which is equal to $(v, \lfloor S \rfloor_S^{\mathcal{I}})$. Therefore, $v = \mathit{runId}\ (\mathit{isEven}_{Id}\ n)$ and $\lfloor S \rfloor_S^{\mathcal{I}} = \lfloor S' \rfloor_S^{\mathcal{I}}$.

So we know that a static method whose argument and return value are not objects must be pure. But parametricity has more to say about our calculus.

5.5 Object capability for a store

We have already informally discussed object capability languages in the introduction, but here we take a closer look. In general, a language that uses the object-capability model only allows the production of an effect through references to objects that represent the capability to perform the effect. However, this statement is a simplified version of the property. Let's make it more precise.

In the object-capability literature, one often considers the *reference graph* of an evaluation state. This graph represents objects in an application as nodes and the references that they have to one another as edges. One then defines what it means for a language to be capability-safe as a restriction on the way that the graph can change during execution. The *authority* of a run-time term is defined as the set of objects that it has a direct or indirect reference to.

Before explaining the restriction that is imposed, we need to explain a way for the authority of a term to increase during execution that does *not* violate capability-safety. Consider the following example:

$$
\begin{array}{l}
\textbf{void}\ \text{pushcap}(\text{Alloc alloc}) = \\
\quad \textbf{let}\ l_1 = \text{alloc.alloc}_{\text{Bool}}(\textbf{true})\ \textbf{in} \\
\quad \textbf{let}\ l_2 = \text{alloc.alloc}_{\text{Bool}}(\textbf{true})\ \textbf{in} \\
\quad \textbf{let}\ l_{indirect} = \text{alloc.alloc}_{\text{State}_{\text{Bool}}}(l_1)\ \textbf{in} \\
\quad \textbf{let}\ \text{comp} = \text{buildComponent}(l_{indirect})\ \textbf{in} \\
\quad l_{indirect}.\text{put}(l_2)
\end{array}
$$

This example features two boolean state variables l_1 and l_2 and a third variable $l_{indirect}$ containing a reference to one of the state variables, initially l_1. A component *comp* is constructed with only a reference to $l_{indirect}$, so that l_2 is initially outside of *comp*'s authority. However, the execution of the last statement makes $l_{indirect}$ reference l_2, so that *comp*'s authority changes to include l_2.

Standard presentations of object-capability are formulated in terms of the reference graph, focusing on heap-based effects. These presentations generally formalise terms' authorities, and require them to not increase except in the way that we just explained. The following properties are often taken as defining:

Only connectivity begets connectivity A subject can influence the authority of only those subjects whose authority influence its own authority.

No authority amplification The change in authority of a subject due to actions performed by another subject is bounded by the authority of the acting subject.

Maffeis et al. [131] formalise these definitions in terms of a heap-passing operational semantics. We will see that similar properties follow from effect parametricity, when we specialise it to a heap-passing semantics.

5.5.1 Heap-locality

Consider our store-passing semantics from Section 5.3. We will define a notion of heap-locality for run-time terms and stores on the Haskell-side of the translation.

We assume a fixed store typing Σ. For any $A \subseteq \mathrm{dom}(\Sigma)$, we write $\Sigma\,|_A$ for the limitation of Σ to domain A and we define the following Haskell function:

$$\begin{aligned}&\mathit{projectStore}_A :: \lfloor \Sigma \rfloor^{\mathcal{I}}_{\Sigma}\ m \to \lfloor \Sigma\,|_A \rfloor^{\mathcal{I}}_{\Sigma}\ m\\ &\mathit{projectStore}_A\ \mathit{store} = \mathit{Heap}\ \{\, a = a\ \mathit{store} \mid a \in A \,\}\end{aligned}$$

Definition 5.5.1. For a set of addresses $A \subseteq \mathrm{dom}(\Sigma)$, we define that a computation $\mathit{cmp} :: \mathit{FS}_\Sigma\ t$ is A-local iff for any monad action $\mathcal{F} : \mathit{FS}_\Sigma \Leftrightarrow \mathit{FS}_\Sigma$, the following implication holds: if $(\mathit{state}_a, \mathit{state}_a)$ is in $\mathit{StateD}\ (\lfloor \Sigma(a) \rfloor^{\mathcal{I}}_t\ \ \mathcal{F})\ \mathcal{F}$ for all $a \in A$, then $(\mathit{cmp}, \mathit{cmp})$ is in $\mathcal{F}\ t$.

We define that a store $S :: \lfloor \Sigma \rfloor^{\mathcal{I}}_{\Sigma}\ \mathit{FS}_\Sigma$ is A-local iff for any monad action $\mathcal{F} : \mathit{FS}_\Sigma \Leftrightarrow \mathit{FS}_\Sigma$, the following implication holds: if $(\mathit{state}_a, \mathit{state}_a)$ is in $\mathit{StateD}\ (\lfloor \Sigma(a) \rfloor^{\mathcal{I}}_t\ \ \mathcal{F})\ \mathcal{F}$ for all $a \in A$, then $(\mathit{projectStore}_A\ S, \mathit{projectStore}_A\ S)$ is in $\lfloor \Sigma\,|_A \rfloor^{\mathcal{I}}_{\Sigma}\ \mathcal{F}$.

The following lemma is useful to work with this notion.

Lemma 5.5.2. *If a computation $\mathit{cmp} :: \mathit{FS}_\Sigma\ t$ is A-local and $A \subseteq B$, then cmp is also B-local.*

If cmp can be written as $f\ \mathit{state}_{a_1}\ \ldots\ \mathit{state}_{a_n}$ for $A = \{a_1 \cdots a_n\}$, then cmp is A-local if

$$f :: \forall m.\mathit{Monad}\ m \Rightarrow \overline{\mathit{StateD}\ (\lfloor \Sigma(a_i) \rfloor^{\mathcal{I}}_t\ \ m)\ m} \to m\ (\lfloor t \rfloor^{\mathcal{I}}_t\ m)$$

If $\mathit{projectStore}_A\ S$ can be written as $f\ \mathit{state}_{a_1}\ \ldots\ \mathit{state}_{a_n}$ for $A = \{a_1 \cdots a_n\}$, then cmp is A-local if

$$f :: \forall m.\mathit{Monad}\ m \Rightarrow \overline{\mathit{StateD}\ (\lfloor \Sigma(a_i) \rfloor^{\mathcal{I}}_t\ \ m)\ m} \to \lfloor \Sigma\,|_A \rfloor^{\mathcal{I}}_{\Sigma}\ m$$

Proof. The first result follows from the definition and the other two directly from parametricity of the f in question. □

Following Maffeis et al. [131], we define $\mathrm{tCap}(s)$ for a run-time statement s (or $\mathrm{tCap}(e)$ for an expression e) as the set of all addresses that s (or e) syntactically contains. In a heap S, we define $\mathrm{hCap}(S, s)$ (or $\mathrm{hCap}(S, e)$) as the least set of addresses such that $\mathrm{tCap}(s) \subseteq \mathrm{hCap}(S, s)$ and $\mathrm{tCap}(S(\mathrm{hCap}(S, s))) \subseteq \mathrm{hCap}(S, s)$. This means that $\mathrm{hCap}(S, s)$ is the set of addresses that expression s *directly or indirectly* has a reference to.

We have the following lemmas about our translation function:

Lemma 5.5.3. *The translation of any run-time statement s such that $\mathcal{I}, \Sigma \vdash_{s_r} s : t$ is equivalent to an expression of the form $f\ state_{a_1}\ \cdots\ state_{a_n}$ for $\{a_1, \cdots, a_n\} = \mathrm{tCap}(s)$ with*

$$f :: \forall m.Monad\ m \Rightarrow \overline{StateD\ (\lfloor \Sigma(a_i) \rfloor_t^{\mathcal{I}}\ m)\ m \rightarrow}\ m\ (\lfloor t \rfloor_t^{\mathcal{I}}\ m)$$

The translation of any run-time expression e such that $\mathcal{I}, \Sigma \vdash_{e_r} e : t$ is equivalent to an expression of the form $f\ state_{a_1}\ \cdots\ state_{a_n}$ for $\{a_1, \cdots, a_n\} = \mathrm{tCap}(e)$ with

$$f :: \forall m.Monad\ m \Rightarrow \overline{StateD\ (\lfloor \Sigma(a_i) \rfloor_t^{\mathcal{I}}\ m)\ m \rightarrow}\ \lfloor t \rfloor_t^{\mathcal{I}}\ m$$

If $\mathrm{tCap}(S(A)) \subseteq A$, then there exists a function f such that

$$f :: \forall m.Monad\ m \Rightarrow \overline{StateD\ (\lfloor \Sigma(a_i) \rfloor_t^{\mathcal{I}}\ m)\ m \rightarrow}\ \lfloor \Sigma\ |_A \rfloor_t^{\mathcal{I}}\ m$$

and (with $A = \{a_1, \cdots, a_n\}$):

$$projectStore_A\ \lfloor S \rfloor_{S_r}^{\mathcal{I}} = f\ state_{a_1}\ \cdots\ state_{a_n}$$

Proof. Structural induction on s and e. For the heap value at address a, we note that $\mathrm{tCap}(S.a) \subseteq A$ and can then apply the first result and the previous lemma. □

A result of Lemmas 5.5.2 and 5.5.3 is that the translations of any S and t are A-local for $A = \mathrm{hCap}(S, t)$.

5.5.2 Capability safety results

We can show that our notion of locality implies results similar to those of Maffeis et al. Their results are rather tied to the operational semantics, but we show that those parts of the results that imply facts about the behaviour of terms

can be transferred to our setting. Their *authority safety* property essentially requires that there exist an authority map auth that assigns to any term t and a heap H an authority $\text{auth}(H, t)$. Authority safety of the language then consists of four requirements:

- auth is a *valid authority map*:
  - Any read or write action executed by term t in heap H must be in $\text{auth}(H, t)$ or apply to a freshly allocated variable.
  - The authority is preserved by an evaluation step.
- The language is *authority safe*:
  - *Only connectivity begets connectivity*: If the authority of a term t in heap H cannot write to addresses that a different term u has the authority to read, then u's authority is unchanged after a step of t.
  - *No authority amplification*: If the authority of a term t in heap H *can* write to addresses that a different term u has the authority to read, then u's authority after an evaluation step of t is bounded by the combined authority of both terms and freshly allocated variables.

We will prove an analogous result to the first two properties with an additional semantics that makes the read and write actions of a computation explicit, so that we can reason about them. The result says that executing an A-local term in an A-local heap will only produce read or write actions to addresses in A, after any number of evaluation steps. We define the alternative semantics and prove the theorem in Section 5.5.3.

For the last two properties, we prove two analogous results:

- If a term t and heap H are A-local, then the heap after executing term t is equal to the original heap for addresses not in A.
- If a term t_1 and heap H are A_1-local and term t_2 and H are A_2-local, then they are all $(A_1 \cup A_2)$-local and so is the resulting heap after executing the second.

This is explained in Section 5.5.4.

5.5.3 From locality to authority

To formalise and prove the first result, we need a semantics that makes it clear what it means to read from or write to a memory location. This is difficult under

the semantics from Section 5.3 because for the execution of a term in a heap, it only considers the final result and the final heap, not how the term got there. For example, it does not distinguish the terms $runFS\ (getM\ a_1)\ \{a_1 = 2, \cdots\}$ and $runFS\ (return\ 2)\ \{a_1 = 2, \cdots\}$. In this section, we define an alternative semantics that fills this gap and use parametricity to prove its correspondence to the previous semantics.

We write $\mathbb{A}$ for the set of possible actions. It contains tuples $(a, \mathtt{r})$ and $(a, \mathtt{w})$ representing the actions of reading from and writing to address a. For practicality, we implicitly overload Haskell values like $state_a$ for use in both monads FS_Σ and the new FS^{expl}_Σ.

$$
\begin{aligned}
&\textbf{data}\ FS^{expl}_\Sigma\ a = \\
&\quad FS\ \{unFS :: StateT\ (\lfloor\Sigma\rfloor^{\mathcal{I}}_\Sigma\ FS^{expl}_\Sigma)\ (Writer\ [\mathbb{A}])\ a\} \\
&\textbf{instance}\ Monad\ FS^{expl}_\Sigma\ \textbf{where}\ ... \\
&runFS^{expl} :: FS^{expl}_\Sigma\ a \to \lfloor\Sigma\rfloor^{\mathcal{I}}_\Sigma\ FS^{expl}_\Sigma \to ([\mathbb{A}], (\lfloor\Sigma\rfloor^{\mathcal{I}}_\Sigma\ FS^{expl}_\Sigma, a)) \\
&runFS^{expl}\ c\ s = runStateT\ (runWriter\ (unFS\ c))\ s \\
&state_a :: StateD\ (\lfloor\Sigma(a)\rfloor^{\mathcal{I}}_t\ FS^{expl}_\Sigma)\ FS^{expl}_\Sigma \\
&state_a = StateD\ \{getM = \quad FS\ (\textbf{do}\ s \leftarrow get \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad tell\ (a, \mathtt{r}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ (a\ s)) \\
&\qquad\qquad , putM = \lambda v.\, FS\ (\textbf{do}\ x \leftarrow get \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad put\ (x\ \{a = v\}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad tell\ (a, \mathtt{w}))\}
\end{aligned}
$$

The monad FS^{expl}_Σ uses a monad state transformer over a writer monad to represent computations. If you do not know monad transformers, it suffices to understand that $StateT\ s\ (Writer\ [l])\ a$ represents a computation that carries an implicit state variable of type s and can add values of type l to a log. The list of all logged values is returned upon execution of the computation. The new implementation of $state_a$ reads and writes to state variables as before, but additionally logs the actions performed using the *Writer* monad's *tell* primitive. Let us first prove that this new semantics corresponds to the old one. To do that, we define a monad action $\mathcal{F}^{expl} : FS_\Sigma \Leftrightarrow FS^{expl}_\Sigma$ as follows. Take $\mathcal{R} : a1 \Leftrightarrow a2$. (cmp_1, cmp_2) is in $\mathcal{F}^{expl}\mathcal{R}$ if and only if for all (H_1, H_2) in $\lfloor\Sigma\rfloor^{\mathcal{I}}_\Sigma\ \mathcal{F}^{expl}$, we have that $(runFS\ cmp_1\ H_1, snd\ (runFS^{expl}\ cmp_2\ H_2))$ is in $\mathcal{R} \times \lfloor\Sigma\rfloor^{\mathcal{I}}_\Sigma\ \mathcal{F}^{expl}$.

To use this relational action, we need to prove the following lemmas. Proofs are in Appendix 5.B.

Lemma 5.5.4. *$\mathcal{F}^{expl}$ is a monad action.*

Lemma 5.5.5. *For any a, $(state_a, state_a)$ is in the relation $\mathcal{F}^{expl}$.*

From these lemmas, we can obtain the following theorem.

Theorem 5.5.6. *For any t, e and H, we have that*

- $(\lfloor t \rfloor^{\mathcal{I}}_{s_r}, \lfloor t \rfloor^{\mathcal{I}}_{s_r})$ *is in* $\mathcal{F}^{expl}$ $(\lfloor t \rfloor^{\mathcal{I}}_{t}\ \mathcal{F}^{expl})$.
- $(\lfloor t \rfloor^{\mathcal{I}}_{s_r}, \lfloor t \rfloor^{\mathcal{I}}_{s_r})$ *is in* $\lfloor t \rfloor^{\mathcal{I}}_{t}\ \mathcal{F}^{expl}$
- $(\lfloor H \rfloor^{\mathcal{I}}_{S_r}, \lfloor H \rfloor^{\mathcal{I}}_{S_r})$ *is in* $\lfloor \Sigma \rfloor^{\mathcal{I}}_{\Sigma}\ \mathcal{F}^{expl}$

This seems like a rather abstract result, but it is more useful than it may appear. Consider, for example, the following consequence:

Consequence 1. *If $\mathcal{I}, \mathcal{P}, \Gamma_0 \vdash t : ty$ and ty is a base type (e.g.* **Bool***), then for any H with $\mathcal{I}, \mathcal{P}, \Gamma_0 \vdash H : \Sigma$, we have that*

$$snd\ (runFS\ \lfloor t \rfloor^{\mathcal{I}}_{s_r}\ \lfloor H \rfloor^{\mathcal{I}}_{S_r}) = snd\ (snd\ (runFS^{expl}\ \lfloor t \rfloor^{\mathcal{I}}_{s_r}\ \lfloor H \rfloor^{\mathcal{I}}_{S_r}))$$

Rephrasing this consequence: a statement with a base result type produces the same result value under both semantics. Note that this consequence also implies something about statements t with a non-base result type or about the contents of the final heap: we can extend any statement t with a test statement t_{test} to **let** $r = t$ **in** t_{test} and apply the previous consequence to deduce that the test result must be the same. As such, the consequence shows the power of the previous, more general result.

This new semantics, shown to correspond to our original semantics, allows us to reason about the actions produced by a computation, as they are explicitly logged. We can now show that the actions produced by an A-local computation in an A-local heap are restricted to the addresses in A.

Theorem 5.5.7. *For $cmp :: FS^{expl}_{\Sigma}\ t$ and $S :: \lfloor \Sigma \rfloor^{\mathcal{I}}_{\Sigma}\ FS^{expl}_{\Sigma}$, if both cmp and S are A-local, then for $(as, (v, S')) = runFS^{expl}\ cmp\ S$, we have that $(a, \_)$ in as implies that $a \in A$.*

The proof is based on the definition of another monad action, but for space reasons, we defer it to Appendix 5.B.

5.5.4 Capability safety

In this section, we show results akin to Maffeis et al.'s authority safety with proofs in Appendix 5.B. The first states that a computation cannot write to addresses it does not have access to.

Theorem 5.5.8. *For* $cmp :: FS_{\Sigma}^{expl}\ t$ *and* $S :: \lfloor\Sigma\rfloor_{\Sigma}^{\mathcal{I}}\ FS_{\Sigma}^{expl}$*, if both cmp and S are A-local, then for* $(as, (v, S')) = runFS^{expl}\ cmp\ S$*, we have that* $(a\ S')$ *is equal to* $(a\ S)$ *for all* $a \notin A$*.*

This first theorem in this section implies that an A-local computation cmp_1 will be unaffected by executing other actions with disjunct authorities. Note that it follows from the theorem that if we have an A' disjunct from A and S is A'-local, then so is S'.

Theorem 5.5.9. *If a computation* cmp_1 *and heap H are* A_1*-local and* cmp_2 *and the same heap H are* A_2*-local, then* cmp_1*,* cmp_2 *and H are all* $(A_1 \cup A_2)$*-local.*

This theorem can be applied when we have two computations cmp_1 and cmp_2 with non-disjunct authorities (over addresses in A_1 and A_2), in the same heap H. This theorem tells us, for example, that if we sequence the computations in some order (e.g. $cmp_1 \gg cmp_2$), then the result will still be $(A_1 \cup A_2)$-local and we can draw conclusions about, for example, the read and write operations they can perform using the result from the previous section.

5.6 The pervasiveness of state allocation

Our account of store capabilities above shows that we can derive powerful properties from the translation to effect-polymorphic monadic computations. However, compared to other accounts of such properties, our results are suspiciously simple. This is because of the fact that in the previous section we have for simplicity omitted mutable state allocation effects.

In this section, we take a look at memory allocation and show how it can be modelled in our model. Importantly, we point out that while other accounts of object capability all choose to make allocation effects implicitly available throughout all code, it is possible to treat this allocation in a more object-capability-based way, using an *allocation capability*. There are two important advantages to this: (a) the formulation of effect parametricity is simpler, and (b) we obtain additional guarantees about code that is not given access to the *alloc* object.

We can model mutable state allocation in our Haskell code using an allocator method of the following type:

$$\textbf{type}\ Allocator\ m = \forall s.s \to m\ (StateD\ s\ m)\}$$

This type synonym defines an allocator method (of type $Allocator\ m$ for a monad m) as a method that allocates a new mutable reference cell of type s

with a given initial value. The mutable reference cell is returned as an object of type *StateD s m*.

Such an allocator method can be easily given for monads like *IO* in terms of more traditional APIs:

$$
\begin{aligned}
&\mathit{ioRefStateD} :: \mathit{IORef}\ s \to \mathit{StateD}\ s\ \mathit{IO} \\
&\mathit{ioRefStateD}\ r = \mathit{StateD}\ (\mathit{writeIORef}\ r)\ (\mathit{readIORef}\ r) \\
&\mathit{allocIO} :: \mathit{Allocator}\ \mathit{IO} \\
&\mathit{allocIO}\ v = \mathit{ioRefStateD}\ \$\ \mathit{newIORef}\ v
\end{aligned}
$$

In other accounts of object capability, allocating new mutable state is implicitly allowed in all code. This corresponds to standard OO practice, where classes can define a set of mutable state variables (*instance state* or *instance variables*) and allocating new instances of a class does not require an explicit capability to do so. We can encode this design using a *MonadAlloc* type class:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadAlloc}\ m\ \textbf{where} \\
&\quad \mathit{alloc} :: \mathit{Allocator}\ m
\end{aligned}
$$

Previously, we have translated our OO code to monadic computations that can run in an arbitrary monad, i.e. values of type $\forall m.\mathit{Monad}\ m \Rightarrow ....$ If we want to implicitly allow the allocation effect, we can instead use types of the form $\forall m.\mathit{MonadAlloc}\ m \Rightarrow ...$, so that any functions can allocate mutable state.

The downside of this choice is that parametricity is weakened. Whereas we were previously able to instantiate the parametricity with any monad action $\mathcal{F}$, we are now more restricted. For types with a *MonadAlloc* constraint on m, we need to additionally require that $\mathcal{F}$ be a *monad action with allocation*, a new notion that we define as follows. A monad action $\mathcal{F} : m_1 \Leftrightarrow m_2$, with instances of *MonadAlloc* available for type constructors m_1 and m_2, is a monad action with allocation iff $(\mathit{alloc}, \mathit{alloc})$ is in $\mathit{Allocator}\ \mathcal{F}$, i.e. $\forall \mathcal{R}.\mathcal{R} \to \mathcal{F}\ (\mathit{StateD}\ \mathcal{R}\ \mathcal{F})$. Intuitively, this additional requirement means that for a relational action $\mathcal{F}$ between computations to be preserved by a function of type $\forall m.\mathit{MonadAlloc}\ m \Rightarrow ...$, it must consider as related any allocations of mutable state with related initial values. The produced *StateD* instances must also respect the relationship.

Our calculus uses an alternative, more object-capability-style design that has (to the best of our knowledge) not been explored before. The basic idea is to make the allocation effect controlable using an explicit capability. That is: code is only allowed to allocate additional mutable state if it has a reference to an *alloc* object that represents the capability to do so. This is also easy to encode in our system: the idea is to continue working with standard effect-polymorphic

code of type $\forall m.Monad\ m \Rightarrow ...$ and translate the *alloc* object to an object of the following dictionary type:

$$\textbf{data}\ AllocD\ m = AllocD\ \{\, allocM :: Allocator\ m \,\}$$

An advantage of this alternative approach is that we can continue using standard monadic actions with the effect parametricity property. When we apply this to a function of type $\forall m.Monad\ m \Rightarrow AllocD\ m \rightarrow ...$, the parametricity property applied to monad action $\mathcal{F}$ we employ will automatically get an assumption that $(allocM\ alloc_1, allocM\ alloc_2)$ be in $Allocator\ \mathcal{F}$, so that for such functions we obtain the same results. The parametricity property is thus less specialised to mutable state allocation without losing expressiveness.

Another advantage of this alternative choice is that we obtain additional properties for our code. For example, for a calculation that does *not* have access to the *alloc* object, we can prove that it *cannot* allocate additional mutable state. Similarly, explicit allocators would allow code to produce its own custom instances of the *AllocD* interface, to run external code with alternative allocators, which could, for example, allocate transactional reference cells or reference cells backed by files instead of the memory. Finally, we can also combine an explicit allocation capability with the technique for temporary capabilities that we will explain in Section 5.8 to obtain a form of regions, that can be implemented using Haskell's *ST* monad [124] (we cannot provide more details because of space constraints).

5.7 Beyond memory access: invariants

A major advantage of the use of effect-parametricity w.r.t. object-capability à la Maffeis et al. is that the former is more general. The latter deals purely with what are essentially memory access properties; it essentially formalises the notion that a piece of code has (read and/or write) access to a memory location if it has a direct or indirect reference to the capability for this access (typically, an appropriate reference to the memory cell).

However, object capability languages are intended to cover much more ground than just memory access. To demonstrate this, this section applies effect-parametricity to prove a property of OO code beyond memory access: the preservation of object state invariants.

An object with mutable instance variables may enforce invariants on the values of those variables. Such invariants are a form of modular reasoning: to ascertain

that the invariant will always hold, it suffices to inspect the source code of only the class's own methods, and verify that they maintain the invariant.

As an example, consider a simple interface *Counter* and *doubleCounter*: an implementation of it in terms of a mutable state variable of type *Int*.

$$
\begin{aligned}
&\textbf{data}\ \mathit{CounterD}\ m = \mathit{CounterD}\ \{\mathit{countM} \quad :: m\ () \\
&\qquad\qquad\qquad\qquad\qquad\qquad , \mathit{getCountM} :: m\ \mathit{Int}\} \\
&\mathit{modifyM} :: \mathit{Monad}\ m \Rightarrow \mathit{StateD}\ s\ m \to (s \to s) \to m\ () \\
&\mathit{modifyM}\ \mathit{ref}\ f = \textbf{do}\ v \leftarrow \mathit{getM}\ \mathit{ref} \\
&\qquad\qquad\qquad\qquad\quad \mathit{putM}\ \mathit{ref}\ (f\ v) \\
&\mathit{dblCounter} :: \mathit{Monad}\ m \Rightarrow \mathit{StateD}\ \mathit{Int}\ m \to \mathit{CounterD}\ m \\
&\mathit{dblCounter}\ \mathit{ref} = \mathit{CounterD}\ \{\mathit{countM} \quad\quad = \mathit{modifyM}\ \mathit{ref}\ (\lambda x.\, x + 2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad , \mathit{getCountM} \quad = \mathit{getM}\ \mathit{ref}\} \\
&\mathit{nDblCounter} :: \mathit{Monad}\ m \Rightarrow \mathit{AllocD}\ m \to m\ (\mathit{CounterD}\ m) \\
&\mathit{nDblCounter}\ \mathit{alloc} = \textbf{do}\ r \leftarrow \mathit{allocM}\ \mathit{alloc}\ 0 \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{dblCounter}\ r)
\end{aligned}
$$

The *dblCounter* methods maintain an invariant on its instance variable of type *Int*, namely that if it is initially 0, its value will always remain even.

We can prove the preservation of this invariant from effect parametricity:

Theorem 5.7.1. *For a base return type r and some base types s_i, consider a function $f :: \mathit{Monad}\ m \Rightarrow \mathit{CounterD}\ m \to \overline{\mathit{StateD}\ s_i\ m \to}\ m\ r$. Then define*

$$\textbf{data}\ H = H\ \{a_1 :: s_1, ..., a_n :: s_n, a :: \mathit{Int}\}$$

Then for any $h :: H$ such that $a\ h$ is even, we have that

$$a\ (\mathit{execState}\ (f\ (\mathit{dblCounter}\ \mathit{state}_a)\ \mathit{state}_{a_1}\ ...\ \mathit{state}_{a_n})\ h)$$

is still even.

Proof. Define a monad action $\mathcal{F}_a^{even} : \mathit{State}\ H \Leftrightarrow \mathit{State}\ H$ such that for $\mathcal{R} : a_1 \Leftrightarrow a_2$, $(\mathit{cmp}_1, \mathit{cmp}_2)$ is in $\mathcal{F}_a^{even}\ \mathcal{R}$ iff $(\mathit{runState}\ \mathit{cmp}_1, \mathit{runState}\ \mathit{cmp}_2)$ is in $H \to (\mathcal{R}, H)$ and additionally, for any h such that $a\ h$ is even, $a\ (\mathit{execState}\ \mathit{cmp}_1\ h)$ and $a\ (\mathit{execState}\ \mathit{cmp}_2\ h)$ are both even.

$\mathcal{F}_a^{even}$ can be easily checked to be a monad action and we can easily check that $(\mathit{state}_{a_i}, \mathit{state}_{a_i})$ is in $\mathit{StateD}\ s_i\ \mathcal{F}_a^{even}$ for all i. Furthermore, while $(\mathit{state}_a, \mathit{state}_a)$ is not in $\mathit{StateD}\ \mathit{id}_{\mathit{Int}}\ \mathcal{F}_a^{even}$, $(\mathit{dblCounter}\ \mathit{state}_a, \mathit{dblCounter}\ \mathit{state}_a)$ *is* in $\mathit{CounterD}\ \mathcal{F}_a^{even}$ (as can be easily checked).

We can then use f's parametricity to conclude that

$$(f\ (dblCounter\ state_a)\ state_{a_1}\ ...\ state_{a_n},\\ f\ (dblCounter\ state_a)\ state_{a_1}\ ...\ state_{a_n})$$

must be in $\mathcal{F}_a^{even}\ r$ and the result follows. □

5.8 Local capabilities

Our proposal to use effect parametricity as the fundamental property of languages with an object capability model is interesting for theoretical reasons and for reasoning purposes. However, there are also some more practical applications. One such application is a design pattern on the Haskell side that can be translated back to the object-oriented world. More specifically, our Haskell translation suggests a way to introduce type-system-enforced *local capabilities* to typed, capability-safe OO languages. Such a local capability is provided to a function under the type-system enforced proviso that the capability can be used only during the execution of the function and must not leak to the store or the function's return value. The technique is based on the ST monad: a well-known technique by Launchbury and Peyton Jones that provides mutable state primitives that can be used inside pure Haskell expressions [124]. The technique is related to existing proposals from the field of ownership types [41].

In this section, we first briefly introduce Launchbury and Peyton Jones' technique and then explain our *local capabilities*.

5.8.1 The ST monad

Launchbury and Peyton Jones [124] consider the use of algorithms that rely on mutable store variables from inside pure code. They start from the observation that some algorithms inherently rely on mutable heap variables. Their work allows the invocation of such algorithms from pure code in a safe way, by modelling the algorithm in a specific monad and using a type-level guarantee that the result of the algorithm contains no reference to the mutable variables that have been allocated. For consistency with the rest of the paper, we will describe Launchbury and Peyton Jones' design in a modified but equivalent form that uses our state and allocator dictionaries.

Launchbury and Peyton Jones introduce a monad ST in which mutable state variables can be allocated and used through a primitive allocator dictionary. The monad is parameterised by an opaque type token s:

$$allocST :: AllocD\ (ST\ s)$$

Launchbury and Peyton Jones' provide a way to invoke an ST computation from pure code. A naive API to allow this would look as follows:

$$runST :: ST\ s\ a \to a$$

When $runST$ is invoked, it executes the provided computation as a sequential program, allocating the necessary mutable state on the heap. When the computation is finished, the mutable state is deallocated and the result value is returned.

The problem with this naive API is that the result value of a computation cmp can very well contain references to the mutable state that was allocated:

$$\begin{aligned} crash = {} & \textbf{let}\ badref = runST\ (alloc\ allocST\ ()) \\ & \textbf{in}\ \ runST\ (readM\ badref) \end{aligned}$$

Problematically, the mutable reference from the first computation escapes and is used in the second. Not only can the mutable variable already have been deallocated by then (since it is not supposed to be referenced any more), but Haskell's lazy semantics could even execute the second computation before the first. Such problems are consequences from the fact that our naive $runST$ primitive violates Haskell's purity.

Launchbury and Peyton Jones propose to solve this by giving $runST$ a type that prevents mutable references from escaping their scope by exploiting the opaque type parameter s of the ST monad and change $runST$'s type:

$$runST :: \forall a.(\forall s.ST\ s\ a) \to a$$

Intuitively, this type means that $runST$ will only execute a computation if the computation can support any concrete type s that $runST$ may choose to give it. If the computation's result contains references allocated during the computation, then these references' types will mention the variable s that $runST$ gave it. However, the type of s does not allow the result type a to mention this s because that would violate the scope of s's quantification. Such scope violations are caught and reported by the type-checker. Effectively, this prevents the leaking of references allocated by the computation.

5.8.2 Generalising the ST monad idea

The ST monad uses a rank-2 type to enforce a functional property. In our setting, it makes sense to change the API further:

$$\begin{aligned} & withHeap :: (\forall m.Monad\ m \Rightarrow AllocD\ m \to m\ a) \to a \\ & withHeap\ cmp = runST\ (cmp\ allocST) \end{aligned}$$

That is, the *runST* API can be reformulated elegantly as an application of effect parametricity of the argument computation. The *AllocD m* argument can be interpreted as an allocation capability given to the callback *cmp* and *withHeap*'s type then enforces that the capability is provided as a *local capability*. This means that the capability or the references allocated through it, can be used within *cmp* but are guaranteed not to leak outside of the computation.

Local capabilities readily extend to other settings. Consider the following API function from Haskell's standard IO library.

$$\mathit{withFile} :: \mathit{FilePath} \to \mathit{IOMode} \to (\mathit{Handle} \to \mathit{IO}\ r) \to \mathit{IO}\ r$$

Ignoring the *IOMode* argument for brevity, this API opens a specified file, calls the user-specified callback with a handle to the opened file and closes the file when the callback returns, returning the callback's result. The following example correctly uses *withFile* to read the first line of a file:

$$\begin{aligned}&\mathit{correct}\ f = \mathbf{do}\ \mathit{firstLine} \leftarrow \mathit{withFile}\ f\ (\lambda h.\ \mathit{hGetLine}\ h)\\ &\qquad\qquad\quad \ldots\end{aligned}$$

However, it is not statically enforced that the handle is used correctly, i.e. only when the file is open. Consider the following problematic code which leaks the handle outside *withFile*:

$$\begin{aligned}&\mathit{crash}\ f = \mathbf{do}\ h \leftarrow \mathit{withFile}\ f\ \mathit{return}\\ &\qquad\qquad\quad \mathit{hGetLine}\ h\end{aligned}$$

The callback even has another way to leak the handle, by storing it in a mutable reference cell, and fetching it there afterwards.

$$\begin{aligned}&\mathit{crash}_2\ f = \mathbf{do}\ \mathit{ref} \leftarrow \mathit{alloc}\ \mathit{allocIO}\ \mathit{Nothing}\\ &\qquad\qquad\quad \mathit{withFile}\ f\ (\lambda h.\ \mathit{putM}\ \mathit{ref}\ (\mathit{Just}\ h))\\ &\qquad\qquad\quad \mathit{Just}\ h \leftarrow \mathit{getM}\ \mathit{ref}\\ &\qquad\qquad\quad \mathit{hGetLine}\ h\end{aligned}$$

Local capabilities can statically prevent this problem. Concretely, we can give *withFile* and *hGetLine* the following types:

$$\begin{aligned}&\mathit{hGetLine} :: \forall n.\mathit{Handle}\ n \to n\ \mathit{String}\\ &\mathit{withFile}\ \ :: \mathit{FilePath} \to (\forall n.\mathit{Monad}\ n \Rightarrow\\ &\qquad\qquad\qquad\qquad \mathit{Handle}\ n \to n\ a) \to \mathit{IO}\ a\end{aligned}$$

With these new types, *correct* will still work, but *crash* and crash_2 both produce an error that the quantified monad n escapes its scope.

The above types strongly restrict the callback. It cannot leak the handle, but also cannot perform *IO* actions that are unrelated to the file handle. Without going into the details, we can additionally allow the latter by changing *withFile*'s type to the following:

$$\begin{aligned}&\textbf{data}\ \mathit{LiftD}\ m\ n = \mathit{LiftD}\ \{\mathit{liftM} :: m\ a \to n\ a\}\\&\mathit{withFile}_2 :: \mathit{FilePath} \to (\forall n.\mathit{Monad}\ n \Rightarrow\\&\qquad\qquad \mathit{LiftD}\ \mathit{IO}\ n \to \mathit{Handle}\ n \to n\ a) \to \mathit{IO}\ a\end{aligned}$$

5.8.3 Closed types

Inspired by these techniques from the Haskell side, we can add language support to make local capabilities available in the OO calculus. For inspiration, we discuss *closed types*: a simple type construct and associated rules that could be used.

For a given type t and base type (e.g. **Bool**) t_r, we can define a type closed$[t \to t_r]$ and a form of lambda with the following additional typing rule. A function can accept an argument of such a type when it wishes to give a callback access to only a set of capabilities and not let it leak them. The following typing rules allow defining and using values of such types.

$$\frac{\mathcal{I}, \mathcal{P}, [x : t] \vdash_s s : t_r}{\mathcal{I}, \mathcal{P}, \Gamma \vdash_e \lambda x.s : \text{closed}[t \to t_r]}$$

$$\frac{\mathcal{I}, \mathcal{P}, \Gamma \vdash_e f : \text{closed}[t \to t_r] \qquad \mathcal{I}, \mathcal{P}, \Gamma \vdash_e x : t}{\mathcal{I}, \mathcal{P}, \Gamma \vdash_s f(x) : t_r}$$

Values of a *closed* type can be constructed as lambda functions according to the first rule. The body of the lambda has access to its argument, but not the outer context (note that the typing derivation in the premise of this rule needs to hold for a context containing only the argument x; the context Γ is not made available). This corresponds to a requirement that the callback is implemented without any of the local capabilities that it would normally have access to.

Soundness of such a feature is guaranteed because we can adapt our translation to accomodate the new feature:

$$\begin{aligned}\lfloor \text{closed}[t \to t_r] \rfloor_t^{\mathcal{I}}\ m &= \forall n.\mathit{Monad}\ n \Rightarrow \lfloor t \rfloor_t^{\mathcal{I}}\ n \to \lfloor t_r \rfloor_t^{\mathcal{I}}\ n\\ \lfloor \lambda x.s \rfloor_e^{\mathcal{I}} &= \lambda x.\ \lfloor s \rfloor_s^{\mathcal{I}}\\ \lfloor f(x) \rfloor_s^{\mathcal{I}} &= \lfloor f \rfloor_e^{\mathcal{I}}\ \ \lfloor x \rfloor_e^{\mathcal{I}}\end{aligned}$$

Closed types allow the definition and use of APIs like *withFile* above, taking callbacks that have access *only* to the local capabilities that we give them. If we want to be able to define APIs like $withFile_2$, whose callbacks have access to all their own capabilities plus the additional local capability we give them, then we need something like Clarke and Wrigstad's borrowed references [41]. Their borrow construct takes essentially a callback that is universally polymorphic in an owner type parameter. Such a construct would combine well with OC languages and translate to functions like our $withFile_2$ above.

5.9 Discussion and related work

In this section, we provide some more background information and discuss related work about four topics: parametricity (Section 5.9.1), extensions to more realistic OO languages (Section 5.9.2), some lessons that our work offers for the design of OO and FP languages (Section 5.9.3) and object-capability languages (Section 5.9.4).

5.9.1 Parametricity

The calculus we have presented in Section 5.2 is very limited as an OO language. Most crucially, we have omitted interface inheritance and subtyping and the only way to implement an interface is using the *new* construct. These choices keep the presentation small and limit the features that we exploit on the Haskell side of our translation. In this section, we take a closer look at the features that we use on the Haskell side and how parametricity applies to them.

We have used Haskell as a type-inferenced notation for System Fω [80], extended with only a limited set of additional features. Most importantly, we use recursive types for the translation of interfaces and the implementation of the monads in our heap-passing semantics. We have avoided the use of some features, although they would be needed for extensions of our work. Specifically, to accomodate more realistic OO calculi (see below), we will need general recursion and for a semantics that treats allocation, we need a realistic model of a store with allocation.

There is quite some work by different authors on the foundational study of extensions of System F and System Fω and the study of parametricity in them. We list some papers as a starting point for interested readers without striving for completeness. Birkedal et al. study a parametric model of impredicative polymorphism with general recursion and recursive types based on domain

theory [14]. Atkey shows parametricity of System Fω using a model in the impredicative calculus of inductive constructions [6]. Birkedal and other authors study a model of impredicative polymorphism with general recursion, general references and recursive types [15]. Vytiniotis et al. prove parametricity for an extension of System Fω with type equality proofs [230].

5.9.2 Extending the calculus

Even though we have kept our calculus small, our ideas apply to more realistic OO languages too. In order to properly support object-oriented programming, we should add at least a form of interface inheritance, and an interface implementation construct with *late binding*, such as classes. However, these do not pose fundamental problems, except that they may increase the features required in the subset of Haskell that our translation targets. Specifically, the translation of an account of classes would probably require general recursion. Interface inheritance can be translated without additional requirements by translating implicit upcasts to explicit dictionary projections.

Composition primitives like classes, prototype-based inheritance or aspects can all be modelled as building blocks that can be combined to obtain functions that construct objects. Schrijvers et al. have already shown how this can be done for aspects and advice in an effect-polymorphic model of aspects that would combine well with our development [172]. In a draft of the current chapter, we used similar techniques to model a form of classes with single inheritance and private instance state. Other forms of inheritance are also encodable, such as prototype-based inheritance or family polymorphism (using techniques like Oliveira et al.'s [170]).

5.9.3 Lessons for language design

Both our new account of object capability and our translation of an object-oriented calculus to effect-polymorphic Haskell functions present interesting lessons for OO and FP language designers‘.

First, our work shows that object capability languages are similar in nature to pure functional languages like Haskell, in the sense that they enforce a strict discipline for the use of effects. A good way to exploit the discipline to obtain formal results is through our translation to effect-polymorphic code in a pure calculus. Techniques from the functional world can be translated back to OC languages, as we have explained in Section 5.8.

On the functional side, APIs in pure programming languages like Haskell are still lacking a general modular treatment of effects, while this is a solved problem in the OO world. Our translation shows how this knowledge from the OO world can be reused on the FP side with an effect-polymorphic model. Oliveira et al. [169] have studied a similar style without looking at the connection to object capabilities.

Our translation also presents interesting possibilities as the basis for foreign language interfaces between object-oriented and pure functional languages or for adding OO support to pure functional languages. In previous work (e.g. Nordlander's O'Haskell [160]) all OO code was either pure or only usable from the *IO* monad, but we can allow more precise types for effectful code that respects the object-capability model.

5.9.4 Object capability

Object capability languages have their roots in the research on operating systems and their security models. A thorough overview of the field is outside the scope of this text, and we refer to Miller's Ph.D. thesis instead [152]. Object-capability languages exist in several forms, for example, Miller's E[3] (a subset of Java), Joe-E [148] (idem), Emily [216] (a subset of OCaml), W7 [191] (based on Scheme) and Caja [151] (a subset of JavaScript).

We are aware of only one formalisation of object capability as a language feature that can be used to reason about code written in the language: Maffeis et al.'s formulation of authority safety and capability safety as language properties for an abstract language with an operational semantics and store-based effects [131]. We have discussed and contrasted their approach to ours throughout this text and we will not repeat this here.

5.10 Conclusion

In this chapter, we identify effect parametricity as a formal property that generalises object capability. We formalise it with a translational semantics of an object-oriented calculus with effects to Haskell, which we take as a convenient notation for System Fω. We no longer see object capability as a property of the reference graph, but a form of effect polymorphism. In the process, we identify mutable state allocation as an effect that is still left implicit in existing OC treatments and discuss advantages of making it controllable. This establishes

[3] `http://erights.org/elang/index.html`

an exciting new link between OO and the fields of functional programming and denotational semantics and presents a large potential for applications.

5.A Calculus properties: progress and preservation

We define a store typing Σ as a mapping of addresses to types.

Evaluation state correctness: extra typing judgements for runtime values. Note: only for closed values.

$$\frac{\mathcal{I}, \Sigma \vdash_{e_r} \Sigma(a) = t}{\mathcal{I}, \Sigma \vdash_{e_r} a : \mathrm{State}_t} \qquad \frac{}{\mathcal{I}, \Sigma \vdash_{e_r} \mathrm{alloc} : \mathrm{Alloc}}$$

We define $\mathcal{I}, \Sigma \vdash_{S_r} S : \Sigma$ iff $\mathcal{I}, \Sigma \vdash_{v_r} S(a) : \Sigma(a)$ for all $a \in \mathrm{dom}(\Sigma)$.

$$\frac{\mathcal{I}, \Sigma \vdash_{S_r} S : \Sigma \qquad \mathcal{I}, \Sigma \vdash_{s_r} s : \mathrm{void}}{\mathcal{I}, \Sigma \vdash_{exec} (S, s)}$$

Theorem 5.A.1 (Initial state typing)**.** *Assume* $\mathcal{I}, [\mathrm{alloc} : \mathrm{Alloc}] \vdash_p s : \mathrm{void}$. *Then* $\mathcal{I}, \Sigma_0 \vdash_{exec} (S_0, s)$.

Proof. By its typing judgement, statement s may only contain a reference to runtime value alloc so it is well-typed in the context of the empty store typing Σ_0. □

Theorem 5.A.2 (Type preservation)**.** *If* $\mathcal{I}, \Sigma \vdash_{exec} ES$ *and* $ES \to^* ES'$ *then there exists a* Σ' *that extends* Σ *such that* $\mathcal{I}, \Sigma' \vdash_{exec} ES'$.

Proof. Induction on evaluation judgement. □

Theorem 5.A.3 (Progress)**.** *Assume* $ES = (S, s)$ *with* $\mathcal{I}, \Sigma \vdash_{exec} ES$. *Then either* s *is a pure expression or* $ES \to ES'$ *for some* ES'.

Proof. Case analysis. □

5.B Proofs

Proof of Theorem 5.3.1. Case analysis on evaluation rules. Uses iota reduction for if+Bool, iota for data type projectors, monadic Left Identity law, standard equational reasoning, some properties of the *State* monad:

$$\begin{array}{l} \mathit{runState}\ (\mathit{getM}\ \mathit{state}_a \ggg\!= f)\ S \cong \mathit{runState}\ (f\ (a\ S))\ S \\ \mathit{runState}\ (\mathit{putM}\ \mathit{state}_a\ e \ggg\!= f)\ S \cong \\ \quad \mathit{runState}\ (f\ ())\ (S\{a \mapsto e\}) \end{array}$$

□

Proof of lemma 5.5.4.

- For any $\mathcal{R} : a_1 \Leftrightarrow a_2$, $(\mathit{return}, \mathit{return})$ is in $\mathcal{R} \to \mathcal{F}^{\mathit{expl}}\mathcal{R}$.

  It suffices to note that $\mathit{runFS}\ (\mathit{return}\ v_1)\ H_1$ is equal to (v_1, H_1) and $\mathit{runFS}^{\mathit{expl}}\ (\mathit{return}\ v_2)\ H_2)$ is equal to $([\,], (v_2, H_2))$.

- For any $\mathcal{R}_a : a_1 \Leftrightarrow a_2$, $\mathcal{R}_b : b_1 \Leftrightarrow b_2$, $((\ggg\!=), (\ggg\!=))$ is in $\mathcal{F}^{\mathit{expl}}\mathcal{R}_a \to (\mathcal{R}_a \to \mathcal{F}^{\mathit{expl}}\mathcal{R}_b) \to \mathcal{F}^{\mathit{expl}}\mathcal{R}_b$.

  If $\mathit{runFS}\ \mathit{cmp}_1\ H_1 = (v_1, H_1')$, $\mathit{runFS}^{\mathit{expl}}\ \mathit{cmp}_2\ H_2 = (as, v_2, H_2')$, $\mathit{runFS}\ (\mathit{f1}\ v_1)\ H_1' = (\mathit{v1}', \mathit{H1}'')$ and $\mathit{runFS}^{\mathit{expl}}\ (\mathit{f2}\ v_2)\ H_2' = (as', \mathit{v2}', \mathit{H2}'')$, then we know that $\mathit{runFS}\ (\mathit{cmp}_1 \ggg\!= \mathit{f1})\ H_1 = (\mathit{v1}', \mathit{H1}'')$ and $\mathit{runFS}^{\mathit{expl}}\ (\mathit{cmp}_2 \ggg\!= \mathit{f2})\ H_2 = (as +\!\!+ as', \mathit{v2}', \mathit{H2}'')$. The required result then follows easily from the relations between the values involved.

□

Proof of lemma 5.5.5.

- $(\mathit{getM}\ \mathit{state}_a, \mathit{getM}\ \mathit{state}_a)$ is in $\mathcal{F}^{\mathit{expl}}\ (\lfloor \Sigma(a) \rfloor_t^{\mathcal{I}}\ \mathcal{F}^{\mathit{expl}})$.

  For (H_1, H_2) in $\lfloor \Sigma \rfloor_{\Sigma}^{\mathcal{I}}\ \mathcal{F}^{\mathit{expl}}$, we know that $(a\ H_1, a\ H_2)$ is in $\lfloor \Sigma(a) \rfloor_t^{\mathcal{I}}\ \mathcal{F}^{\mathit{expl}}$. The result follows after noting that $\mathit{runFS}\ (\mathit{getM}\ \mathit{state}_a)\ H_1$ is equal to $(a\ H_1, H_1)$ and $\mathit{runFS}^{\mathit{expl}}\ (\mathit{getM}\ \mathit{state}_a)\ H_2)$ is equal to $([(a, \mathtt{r})], (a\ H_2, H_2))$.

- $(\mathit{putM}\ \mathit{state}_a, \mathit{putM}\ \mathit{state}_a)$ is in $\lfloor \Sigma(a) \rfloor_t^{\mathcal{I}}\ \mathcal{F}^{\mathit{expl}} \to \mathcal{F}^{\mathit{expl}}\ ()$.

  Assume that (v_1, v_2) is in $\lfloor \Sigma(a) \rfloor_t^{\mathcal{I}}\ \mathcal{F}^{\mathit{expl}}$ and (H_1, H_2) is in $\lfloor \Sigma \rfloor_{\Sigma}^{\mathcal{I}}\ \mathcal{F}^{\mathit{expl}}$. Then $(H_1\ \{\, a = v_1 \,\}, H_2\ \{\, a = v_2 \,\})$ is still in $\lfloor \Sigma \rfloor_{\Sigma}^{\mathcal{I}}\ \mathcal{F}^{\mathit{expl}}$. The result follows after noting that $\mathit{runFS}\ (\mathit{putM}\ \mathit{state}_a\ v_1)\ H_1$ is equal to $((), H_1\ \{\, a = v_1 \,\})$ and $\mathit{runFS}^{\mathit{expl}}\ (\mathit{putM}\ \mathit{state}_a\ v_2)\ H_2)$ is equal to $([(a, \mathtt{w})], ((), H_2\ \{\, a = v_2 \,\}))$.

□

Proof of Theorem 5.5.6. With $A = \mathrm{dom}(H)$, we get from lemma 5.5.3 functions f and f_H such that for all a:

$$f \quad :: \forall m.Monad\ m \Rightarrow \overline{StateD_{\lfloor \Sigma(a_i) \rfloor_t^{\mathcal{I}}\ m \rightarrow}}\ m\ (\lfloor t \rfloor_t^{\mathcal{I}}\ m)$$
$$f_H :: \forall m.Monad\ m \Rightarrow \overline{StateD_{\lfloor \Sigma(a_i) \rfloor_t^{\mathcal{I}}\ m \rightarrow}}\ \lfloor \Sigma \rfloor_{\Sigma}^{\mathcal{I}}\ m$$

and

$$\lfloor t \rfloor_{s_r}^{\mathcal{I}} = f\ state_{a_1}\ \cdots\ state_{a_n}$$
$$\lfloor projectStore_A H \rfloor_{e_r}^{\mathcal{I}} = f_H\ state_{a_1}\ \cdots\ state_{a_n}$$

From parametricity of f and f_a, and because of lemmas 5.5.4 and 5.5.5, we get that

$$(\lfloor t \rfloor_{s_r}^{\mathcal{I}}, \lfloor t \rfloor_{s_r}^{\mathcal{I}}) = (f\ state_{a_1}\ \cdots\ state_{a_n}, f\ state_{a_1}\ \cdots\ state_{a_n})$$

is in $\mathcal{F}^{expl}(\lfloor t \rfloor_t^{\mathcal{I}}\ \mathcal{F}^{expl})$.

The result for expressions follows similarly and the result for heaps follows from that for each of its values. □

Proof of consequence 1. This follows from lemma 5.5.3, the definition of $\mathcal{F}^{expl}$, the fact that the translation of a base type does not involve the monad and the fact that the relational interpretation of such a translated base type is the identity relation on that type. □

We define a relational action on FS_{Σ}^{expl} for use in the proof of theorems 5.5.7 and 5.5.8.

Definition 5.B.1. We define monad action $\mathcal{F}_A^{restr}$ on monad FS_{Σ}^{expl} as follows. Given a relation $\mathcal{R} : a_1 \Leftrightarrow a_2$ and $cmp_i :: FS_{\Sigma}^{expl}\ a_i$, we define that (cmp_1, cmp_2) is in $\mathcal{F}_A^{restr}\ \mathcal{R}$ iff for any heaps H_1, H_2 with $(projectStore_A\ H_1, projectStore_A\ H_2)$ in $\lfloor \Sigma \mid_A \rfloor_{\Sigma}^{\mathcal{I}}\ \mathcal{F}_A^{restr}$, we have that if $runFS^{expl}\ cmp_1\ H_1 = (as_1, (v_1, H_1'))$ and $runFS^{expl}\ cmp_2\ H_2 = (as_2, (v_2, H_2'))$, then (v_1, v_2) is in $\mathcal{R}$ and $(projectStore_A\ H_1', projectStore_A\ H_2')$ is in $\lfloor \Sigma \mid_A \rfloor_{\Sigma}^{\mathcal{I}}\ \mathcal{F}_A^{restr}$. Additionally, for equal-length finite prefixes as_1' and as_2' of as_1 and as_2, we require that $as_1' = as_2'$ and for all $(a, \_) \in as_1'$, a is in A. Finally, for $B = \mathrm{dom}(\Sigma) \setminus A$, we have that $projectStore_B\ H_1' \equiv projectStore_B\ H_1$ and $projectStore_B\ H_2' \equiv projectStore_B\ H_2$.

For this relational action, we have the following lemma's:

Lemma 5.B.2. *$\mathcal{F}_A^{restr}$ is a monad action.*

Proof.
- If (v_1, v_2) is in $\mathcal{R}$, then $(return\ v_1, return\ v_2)$ is in $\mathcal{F}_A^{restr}\ \mathcal{R}$: the conditions from the definition of $\mathcal{F}_A^{restr}$ are easy to check.
- If (m_1, m_2) is in $\mathcal{F}_A^{restr}\mathcal{R}_a$ and (k_1, k_2) is in $\mathcal{R}_a \to \mathcal{F}_A^{restr}\ \mathcal{R}_b$ then $(m_1 \gg\!= k_1, m_2 \gg\!= k_2)$ is in $\mathcal{F}_A^{restr}\ \mathcal{R}_b$: the conditions from the definition of $\mathcal{F}_A^{restr}$ are easy to check.

□

Lemma 5.B.3. *$(state_a, state_a)$ is in $\mathcal{F}_A^{restr}$ for any $a \in A$.*

Proof.
- $(getM\ state_a, getM\ state_a)$ is in $\mathcal{F}_A^{restr}\ (\lfloor\Sigma(a)\rfloor_t^{\mathcal{I}}\ \mathcal{F}_A^{restr})$. Conditions easily checked.
- $(putM\ state_a, putM\ state_a)$ is in $\lfloor\Sigma(a)\rfloor_t^{\mathcal{I}}\ \mathcal{F}_A^{restr} \to \mathcal{F}_A^{restr}\ ()$. Conditions easily checked.

□

Based on these lemmas, we have:

Proof of Theorem 5.5.7. If *cmp* and S are A-local, then lemmas 5.B.2, 5.B.3 and the definition tell us that (cmp, cmp) is in $\mathcal{F}_A^{restr}\ t$ and (S, S) is in $\lfloor\Sigma\rfloor_\Sigma^{\mathcal{I}}\ \mathcal{F}_A^{restr}$. By the definition of $\mathcal{F}_A^{restr}$, we then get the result about *as*. □

Proof of theorem 5.5.8. If *cmp* and S are A-local, then lemmas 5.B.2, 5.B.3 and the definition tell us that (cmp, cmp) is in $\mathcal{F}_A^{restr}\ t$. The result follows from the definition of $\mathcal{F}_A^{restr}$. □

Proof of theorem 5.5.9. The result for cmp_1 and cmp_2 follows easily from lemma 5.5.2. For the heap, take a monad action $\mathcal{F} : FS_\Sigma \Leftrightarrow FS_\Sigma$, and assume that $(state_a, state_a)$ is in $StateD\ (\lfloor\Sigma(a)\rfloor_t^{\mathcal{I}}\ \mathcal{F})\ \mathcal{F}$ for all a in $A_1 \cup A_2$. Then the A_1- and A_2-locality of H give us that $(projectStore_{A_1}\ S, projectStore_{A_1}\ S)$ is in $\lfloor\Sigma \mid_{A_1}\rfloor_\Sigma^{\mathcal{I}}\mathcal{F}$ and $(projectStore_{A_2}\ S, projectStore_{A_2}\ S)$ is in $\lfloor\Sigma \mid_{A_2}\rfloor_\Sigma^{\mathcal{I}}\mathcal{F}$. From the definition of the translation function, the definition of $projectStore_A$ and the relational interpretation of the heap data type, we then get that $(projectStore_{A_1\cup A_2}\ S, projectStore_{A_1\cup A_2}\ S)$ is in $\lfloor\Sigma \mid_{A_1\cup A_2}\rfloor_\Sigma^{\mathcal{I}}\mathcal{F}$. □

Chapter 6

Conclusion

In this final section, I want to take a step back to consider the results of the work in this thesis and look forward on directions for future research and practice. I present them separately for the four domains considered in the text.

Future work In my opinion, there is interesting further research to be done in all four domains that I discussed in this text. I will discuss some ideas for each of the four fields covered in this text in the following sections.

Instance arguments My work on instance arguments fits in a tradition started by eqtype variables in Standard ML [155] and Haskell's type classes [235]. In this approach, ad hoc polymorphism is treated in a type system by adding a sort of phantom arguments, whose value can be inferred from the type at which the ad hoc polymorphic function is used. Like many of the related work, instance arguments build further on this tradition, extending the basic idea to new types of programming languages and exploring alternatives to some of the initial design choices.

Let's take a look at some important aspects of the design of instance arguments. First, the values that ad hoc polymorphic functions can implicitly accept (which we called *phantom arguments* above) can be values of arbitrary types. Unlike type classes, they are not restricted to structures that were specially declared for use with ad hoc polymorphism. A similar choice was previously made in the design of Scala's implicit arguments, Coq's canonical structures and Dreyer et al.'s modular type classes. The design choice allows reusing existing structuring mechanisms and our work confirms previous evidence that it brings clear benefits

and few downsides. I think the evidence has by now become sufficiently strong to make Haskell follow this direction as well. This can be done concretely by making dictionary records for all type classes available to the user as data types and by providing some form of *local instances*. The incoherence problems with local instances that were described among others by Kiselyov et al. [118] can be avoided by requiring sufficient type information from the programmer. I am currently mentoring a student who is investigating a practical design of such a Haskell extension for his master thesis.

A second important design choice of instance arguments is more controversial. Most related work exposes a logic programming (Prolog-like) computational model on the type-level through equivalents of *parametric instances*. The use of a Prolog-like form of type-level computation is not a good design choice, because it differs from established practice in formal calculi (including System F_C, underlying GHC's type system [244, 217, 238]) and from the value-level model of functional languages. Prolog-like type-level computation is less well understood, as indicated e.g. by the fact that current formalisations of the Haskell type system keep the type-level computation of parametric instances (and type families) out of the underlying formal language System F_C, but instead model them using equality axioms that are added to the code by the type inferencer. I expect this will become increasingly problematic as the type system of Haskell gains more power (as it has in recent years).

A result of my work in this context is evidence that this logic programming-based form of type-level computation is not essential for typing ad hoc polymorphism.

Context-free grammars Grammars are a classic application of higher-level programming; the use of a higher-level grammar language for writing parsers is almost universally accepted. External DSLs (where an external tool compiles the grammar to a parser) are most often used, but they suffer from limitations like poor integration with tools, specification techniques etc. in the host language and often a lack of features (like abstraction) in the grammar language. Embedded DSLs offer an alternative, but a good representation for the recursion in grammars has proved hard to come by. The goal is to define a DSL such that its semantics closely corresponds to that of context-free languages, but as we have explained, the host language recursion semantically does not correspond to recursion in context-free grammars. The approach presented in Chapter 3, but also the approach I studied together with Ilya Sergey, Frank Piessens and Dave Clarke and the approach studied by Oliveira et al. [166], are all based on a functional approach and start from a fixpoint primitive and Baars and Swierstra have explored typed references for use with imperative grammar algorithms [9].

Despite the advances in this line of work, it is my impression that we have not yet found the perfect representation of grammars and recursion. The above approaches all succeed in the sense that the EDSLs' semantics correspond to that of grammars or grammars coupled with semantic actions, but grammars remain unnecessarily hard to work with or hard to combine with one another. To this date, the `grammar-combinators` approach reported on in this text is the first one to decouple grammars from semantic actions, without necessarily passing through an abstract syntax tree representation. Perhaps it is worth further investigating whether its overhead can be reduced and grammars made easier to combine, perhaps reusing Baars and Swierstra's idea of using type-level lists of types [9] and modernising it to use recent extensions of the GHC type system (specifically data kinds [244] and closed type families [68]).

Typed meta-programming As explained in the introduction, meta-programs inherently feature a very complex two-level semantics, making it very hard to properly specify their properties. Perhaps the only specification technique that is up to the task are the very powerful dependent type systems. Meta-programs are not imperative by nature, so in order to not complicate matters further, it is advantageous to represent them as functional programs . As such, the starting point for the work presented in Chapter 4 is quite natural: applying dependent types to the specification of functional meta-programs.

On the other hand, the goal of the work is rather ambitious. Rather than writing meta-programs that manipulate programs in a comparatively simple language like Haskell, it reaches directly for higher-hanging fruit: a dependently-typed object language. As discussed in Chapter 4, this choice implies representing terms in a very complex type system and even finding a way around Gödel's second incompleteness theorem. Nevertheless, this work achieves just that, albeit by side-stepping the important soundness problem. By introducing meta-programming support through primitives in the language, soundness remains an open question but we can already explore the potential of the approach.

And its potential has turned out substantial. The demonstration meta-programs in the fields of datatype-generic programming and proof tactics improve significantly over the state of the art in their respective domains. They are strongly typed and functional and do not feature any obvious fundamental limitations. Unfortunately, that does not mean they are readily applicable for practical programming or computer-assisted proof engineering: their performance is currently prohibitive.

All in all, I think this work shows great potential for applying dependently-typed languages to writing well-typed meta-programs. Even for the very complex object language considered, the approach is successful and in future work, I

want to try to make it perform acceptably in a dependently-typed language with efficient compile-time evaluation. The soundness problem appears trickier. As explained in the introduction, the fundamental logical limitations of Gödel's second incompleteness theorem imply that one shouldn't expect to construct powerful meta-programming primitives within a language itself. My approach of using primitives therefore seems warranted. However, it would be better to use primitives whose soundness is more credible, like the axiom of strong normalisation for typed terms in the object language suggested by a reviewer and used by Barras [12]. Implementing meta-programming primitives by relying on such a normalisation axiom is therefore important future work.

Another topic that I believe deserves further study is applying this approach to meta-programming for simpler languages. Strongly typed meta-programs written in Agda for an object language like Haskell or Java seem a very compelling and novel aim. For such a simpler language, the meta-programs themselves can probably be kept simpler and the overhead of correctness proofs significantly reduced. There is possibly a link to the work on instance arguments as such meta-programming could perhaps be integrated into Haskell as a replacement for the logic-programming based type-level computation primitives currently exposed by parametric type class instances..

Effect polymorphism Although the work in Chapter 5 has not yet been published and the presentation could still be significantly improved, I see it as the most significant contribution in this thesis. The object-capability approach is in my opinion currently one of the most promising techniques for improving software quality in industrial applications. The scientific literature mainly sees it as a security technique (see e.g. [64, 216, 191, 151, 148]), and a contribution of the work presented in Chapter 5 is already that we consider it as a much more general semantic approach for reasoning about side-effecting imperative code. It is hard to overestimate the importance of better control over the side-effecting behaviour of components, as this behaviour is crucial for architectural properties (e.g. all network access is encrypted, the front-end can only access the database through the business layer), inter-component security assumptions (e.g. an ad in an online mail application cannot inspect e-mails) or functional properties of components or applications (e.g. all file access uses a transaction journal).

Such properties are generally hard to guarantee in current programming languages, as most specification techniques and those that do remain largely academical (see section 1.2.2). The object-capability approach on the other hand can be used in the existing object-oriented languages that are widely used in the industry, simply by avoiding certain features (e.g. static mutable state and other static effectful APIs) and respecting certain rules. For security applications,

language modifications can enforce the respect of such rules [148, 216, 191, 151] and such limitations have even been added (optionally) to the most recent version of JavaScript [106].

Despite the fact that the importance of object-capability systems seems well understood in computer security circles, it is still poorly understood. One generally starts from the notion of the reference graph: the directed graph consisting of all objects in a system, linked by the references that they hold to one another. Object capability is then understood as a restriction on how a currently executing statement is allowed to change the reference graph, based on the references it holds into the graph. The best current formalisation of this idea is by Maffeis et al. [131]. In chapter 5, we have proposed the novel view that we should instead use the notion of effect polymorphism. The basic idea is that code respecting object capabilities is restricted to the effects it can perform through methods of the objects it has access to. As such, by providing the code only with access to objects producing certain effects, it can produce those effects and only those. This novel view upgrades object-capabilities from a security-related technique to a general approach for managing the side-effects of imperative code.

To support this view, we have developed a translation from a simple object-capability calculus to monadic functions in Haskell, where we use Haskell as a type-inferenced notation for a formal calculus close to System F_ω. This puts us in line with a well-established tradition of using monads to reason about arbitrary side-effects [156]. It also puts forward a new link between the fields of formal security (where object-capabilities are generally studied) and the foundations and formal semantics of (functional) programming languages (where System F_ω is an important technical tool). A tool from the second field (the formal property of parametricity) then allows us to derive properties of effect-polymorphic code that we have shown in Chapter 5 to generalise previous characterisations of object-capability languages based on the reference graph.

The identification of effect polymorphism as the defining property of object-capability languages yields a very general and useful characterisation. It provides a clear criterion to decide if a language feature satisfies the object-capability model: can it be translated to our effect-polymorphic computations? The characterisation is useful in the sense that the language property can more easily be exploited for concrete components than previous characterisations, by applying the parametricity property.

Effect polymorphism also establishes a novel link between the field of object-oriented and object-capability programming languages on the one hand and functional programming, polymorphism and denotational semantics on the other hand. This link provides an opportunity for both fields to learn from each other.

In functional programming, a modular treatment of side-effects is still the topic of ongoing research [203, 22, 113, 169] and effect polymorphic functions (such as the results of our translation) provide an unexplored track that builds on the experience of OO programming. Conversely, the field of object-oriented programming and object capabilities would benefit from functional programming ideas; an example are the *local capabilities* that we present in Section 5.8.

A limitation of the presentation in Chapter 5 is that it focuses on effect polymorphism and its relation to previous characterisations of object-capability languages and, as a result, the translation only covers a minimal object-oriented calculus, without features like interface extension and subtyping, classes and inheritance. However, the translation can be extended to support features like the above, and an earlier version of the work actually does this for interface inheritance, subtyping with explicit upcasts and classes. With such extensions, one obtains a novel denotational semantics for object-oriented languages. It looks promising for studying the interaction of instance state and other effects with various forms of inheritance like family polymorphism, multiple inheritance, aspects etc. Despite the fact that dealing with mutable instance state is known to be problematic in many situations, effects are disregarded in models like Featherweight Java [104]. Nevertheless, a detailed comparison with related research remains future work. Another direction for future work comes from the observation that my semantics of OC languages as effect-polymorphic functions would also work in languages other than Haskell. For example, I intend to study effect polymorphic functions in dependently-typed languages like Agda, which I expect will produce object-capability systems with richer type systems that allow for verifiable specifications of objects' contracts.

Practically, I want to emphasize the importance of the object-capability model for object-oriented programming. Unlike other techniques for managing side-effects (see Section 1.2.2), the approach is ready for real-world use. For example, the latest version of the JavaScript programming language [106] has already been strongly influenced by industry practice and research around capability-safe subset languages for security applications (specifically Google Caja [151]), although it remains a bit unclear if the resulting language (in its *strict mode*) is actually capability-safe. Bringing the object-capability model to mainstream languages has important benefits, not just for security applications (e.g. [148, 151]), but also to make architectural properties realistically reviewable [148], for explicitising inter-component assumptions about side-effects (see e.g. Bracha's story about re-entrancy of javac [19]) and for informing compiler and middleware of side-effect-related properties of the code (enabling applications like composable transactions in Haskell's STM monad [92]).

Finally, I want to put forward the opinion that the object-capability model should not be regarded as an optional feature that builds on the object oriented programming paradigm. Rather, it should be seen as one of the defining characteristics of object-oriented programming[1], replacing the more specific notion of encapsulation. This is because the approach is essentially a generalisation of the object-oriented principle of encapsulation. From a discipline on producing side-effects involving objects' instance states, the principle becomes a more general discipline on the production of arbitrary side-effects. In a simplified setting, one can imagine that the outside world is made the instance state of a new `World` object, and other code is not allowed to touch it unless (directly or indirectly) invoking that object's methods. Language features, APIs, designs and design patterns that are in conflict with effect polymorphism, such as mutable static state, static effectful APIs (e.g. Java's `System.out.println`) and patterns like Gamma et al.'s *Singleton* [78] are remnants of other paradigms (particularly *structured programming*) that should be avoided. I realise that this view is currently not sufficiently supported by the work presented in this thesis, but in future work I intend to develop this argument further.

It is worth noting though, that many people in the object-oriented community seem to have reached similar conclusions already, recommending object-capability-based designs as best practice. Their recommendations are sometimes influenced by object-capability research (e.g. Bracha [19]) and sometimes independent and using other terminology (e.g. Fowler's *dependency injection* [75]). They are typically motivated by specific applications of the object-capability treatment of side-effects. Such motivations include the fact that effect polymorphism enables independent testing of side-effecting components, because these components only have access to objects whose side-effects can be controlled [75]. Another motivation is that the interaction of components with mutable state variables is important to control in concurrent or distributed settings [19]. Finally, another reason is that dependencies between components that interact using side-effects are often hard to detect and such dependencies can be made explicit using an object-capability-based design [75].

[1] Together with the orthogonal features of *subtyping* for interfaces and *inheritance* with open recursion.

Bibliography

[1] A. Abel. Irrelevance in type theory with a heterogeneous equality judgement. In *Foundations of Software Science and Computational Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 57–71. Springer Berlin Heidelberg, 2011. pages 157

[2] D. Abrahams and A. Gurtovoy. *C++ template metaprogramming: Concepts, tools, and techniques from Boost and beyond*. Addison-Wesley, 2004. pages 160

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 2nd edition edition, 2006. pages 21, 84, 105, 120, 122, 123

[4] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):365, 1996. pages 5

[5] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Generic Programming*, pages 1–20. Kluwer, 2003. pages 40, 161

[6] R. Atkey. Relational parametricity for higher kinds. In *Computer Science and Logic*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 46–61. Schloss Dagstuhl Leibniz Center for Informatics, 2012. pages 13, 198

[7] L. Augustsson. Cayenne — a language with dependent types. In S. Swierstra, J. Oliveira, and P. Henriques, editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 240–267. Springer Berlin Heidelberg, 1999. pages 18

[8] A. I. Baars and D. S. Swierstra. Type-safe, self inspecting code. In *Haskell Workshop*, pages 69–79. ACM, 2004. pages 124

[9] A. I. Baars, D. S. Swierstra, and M. Viera. Typed transformations of typed abstract syntax. In *Types in Language Design and Implementation*, pages 15–26. ACM, 2009. pages 117, 124, 205, 206

[10] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978. pages 4, 6

[11] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. *Information Processing*, 1959. pages 20

[12] B. Barras and G. Huet. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université de Paris 07, 1999. pages 158, 207

[13] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 86–101. Springer Berlin Heidelberg, 2008. pages 68

[14] L. Birkedal, R. E. Møgelberg, and R. L. Petersen. Domain-theoretical models of parametric polymorphism. *Theor. Comput. Sci.*, 388(1-3):152–172, December 2007. pages 13, 198

[15] L. Birkedal, K. Støvring, and J. Thamsborg. Relational parametricity for references and recursive types. In *Types in Language Design and Implementation*, pages 91–104. ACM, 2009. pages 13, 198

[16] N. Blum and R. Koch. Greibach normal form transformation revisited. *Information and Computation*, 150(1):112–118, 1999. pages 117

[17] S. Boutin. Using reflection to build efficient and certified decision procedures. In *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer Berlin Heidelberg, 1997. pages 161

[18] J. Bovet and T. Parr. ANTLRWorks: an ANTLR grammar development environment. *Software: Practice and Experience*, 38(12):1305–1332, 2008. pages 22

[19] G. Bracha. Cutting out static. `http://gbracha.blogspot.be/2008/02/cutting-out-static.html`, February 2007. pages 209, 210

[20] E. Brady and K. Hammond. A verified staged interpreter is a verified compiler. In *Generative Programming and Component Engineering*, pages 111–120. ACM, 2006. pages 130, 132, 158, 161

[21] E. C. Brady. IDRIS: systems programming meets full dependent types. In *Programming Languages meets Program Verification*, pages 43–54. ACM, 2011. pages 18

[22] E. C. Brady. Programming and reasoning with algebraic effects and dependent types. In *International Conference on Functional Programming*. ACM, 2013. pages 209

[23] E. C. Brady and K. Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *International Conference on Functional Programming*, pages 297–308. ACM, 2010. pages 159

[24] K. Brink, S. Holdermans, and A. Löh. Dependently typed grammars. In *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*, pages 58–79. Springer Berlin Heidelberg, 2010. pages 125

[25] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 2002. pages 14, 23, 130, 159

[26] W. C. Bruno Barral. Machine analytique de charles babbage, exposée au science museum de londres. `https://en.wikipedia.org/wiki/File:AnalyticalMachine_Babbage_London.jpg`, 2009. pages 3

[27] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964. pages 118

[28] V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–28, July 2005. pages 14, 18, 132, 155

[29] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985. pages 12

[30] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009. pages 81, 85, 91, 107, 123, 126, 127

[31] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 1–13. ACM, 2005. pages 55

[32] J. Chapman. Type theory should eat itself. In *Logical Frameworks and Metalanguages: Theory and Practice*, volume 228 of *Electronic Notes in Theoretical Computer Science*, pages 21–36. Elsevier, 2009. pages 53, 130, 131, 156, 161, 162

[33] J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *International Conference on Functional Programming*, pages 3–14. ACM, 2010. pages 53, 161

[34] C. Chen and H. Xi. Meta-programming through typeful code representation. In *International Conference on Functional Programming*, pages 275–286. ACM, 2003. pages 130, 131, 158, 160

[35] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming*, pages 143–156. ACM, 2008. pages 155

[36] A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *Programming Languages Design and Implementation*, pages 122–133. ACM, 2010. pages 15, 18, 130, 131, 160

[37] A. Chlipala. Certified programming with dependent types. `http://adam.chlipala.net/cpdt/`, 2012. pages 130, 133, 153, 159, 161

[38] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956. pages 20

[39] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940. pages 11

[40] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Asian Workshop on Programming Languages and Systems*, volume 1742 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1999. pages 126

[41] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer Berlin Heidelberg, 2003. pages 27, 193, 197

[42] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64. ACM, 1998. pages 10, 15

[43] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler,

P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the NuPRL proof development system.* Prentice-Hall, Inc., 1986. pages 18

[44] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988. pages 17, 18, 53

[45] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium - Volume 7*, volume 81, pages 346–355. USENIX Association, 1998. pages 5

[46] H. Curry and R. Feys. *Combinatory Logic.* North-Holland, 1958. pages 11

[47] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *Programming Languages Design and Implementation*, pages 411–422. ACM, 2013. pages 15

[48] N. A. Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer Berlin Heidelberg, 2007. pages 53, 130, 131, 156, 161, 162

[49] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Principles of Programming Languages*, pages 133–144. ACM, 2008. pages 14

[50] N. A. Danielsson. Operational semantics using the partiality monad. In *International Conference on Functional Programming*, pages 127–138. ACM, 2012. pages 14, 18, 132, 155, 161

[51] N. A. Danielsson and many others. The Agda standard library. `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.StandardLibrary`, 2009. pages 40, 135, 136

[52] N. A. Danielsson and U. Norell. Parsing mixfix operators. In *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 80–99. Springer Berlin Heidelberg, 2008. pages 136

[53] N. A. Danielsson and U. Norell. Total parser combinators. In *International Conference on Functional Programming.* ACM, 2010. pages 88, 126

[54] J. Darcy. JSR 269: Pluggable annotation processing API. `http://jcp.org/en/jsr/detail?id=269`, 2011. pages 159

[55] N. G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on automatic demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer Berlin Heidelberg, 1970. pages 17

[56] D. Delahaye. A tactic language for the system Coq. In *Logic Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 85–95. Springer Berlin Heidelberg, 2000. pages 130, 132, 133, 159

[57] D. Devriese, D. Peebles, and N. A. Danielsson. Overloaded literals. discussion on the Agda mailing list, `https://lists.chalmers.se/pipermail/agda/2011/003177.html`, June 2011. pages 66

[58] D. Devriese and F. Piessens. Explicitly recursive grammar combinators - Implemention of some grammar algorithms. Technical Report CW594, KULeuven CS, 2010. pages 83

[59] D. Devriese and F. Piessens. Explicitly recursive grammar combinators. In *Practical Applications of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*, pages 84–98. Springer Berlin Heidelberg, 2011. pages 83

[60] D. Devriese and F. Piessens. Finally tagless observable recursion for an abstract grammar model. *J. Funct. Programming*, 22(06):757–796, 2012. pages 83

[61] A. Dijkstra and S. D. Swierstra. Making implicit parameters explicit. Technical Report UU-CS-2005-032, Dept. ICS, Utrecht University, 2005. pages 67, 70

[62] E. Dijkstra. A case against the GOTO statement. *Communications of the ACM*, 11:147, 1968. pages 2

[63] D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller. Modular type classes. In *Principles of Programming Languages*. ACM, 2007. pages 67, 69

[64] S. Drossopoulou and J. Noble. The need for capability policies. In *Formal Techniques for Java-like Programs*, page 6. ACM, 2013. pages 207

[65] D. Duggan and J. Ophel. Type-checking multi-parameter type classes. *J. Funct. Program.*, 12(2):133–158, March 2002. pages 67

[66] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994. pages 135, 152

[67] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer Berlin Heidelberg, 1999. pages 18, 158, 161

[68] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Principles of Programming Languages*. ACM, 2014. pages 18, 206

[69] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273. ACM, 1997. pages 16, 17

[70] L. Erkök and J. Launchbury. A recursive do for Haskell. In *Haskell Workshop*. ACM, 2002. pages 92, 127

[71] C. Flanagan and M. Abadi. Types for safe locking. In *Programming Languages and Systems*, volume 6539 of *Lecture Notes in Computer Science*, pages 91–108. Springer Berlin Heidelberg, 1999. pages 15

[72] S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: indexed types now! In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 112–121. ACM, 2007. pages 130, 131

[73] B. Ford. Packrat parsing: simple, powerful, lazy, linear time: Functional pearl. In *International Conference on Functional Programming*. ACM, 2002. pages 82, 122

[74] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *International Conference on Functional Programming*, pages 279–291. ACM, 2011. pages 17

[75] M. Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, January 2004. pages 210

[76] R. Frost, R. Hafiz, and P. Callaghan. Parser combinators for ambiguous left-recursive grammars. In *Practical Applications of Declarative Languages*, volume 4902 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008. pages 88

[77] J. Fuegi and J. Francis. Lovelace, Babbage and the creation of the 1843 'notes'. *Annals of the History of Computing, IEEE*, 25(4):16–26, 2003. pages 2

[78] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. pages 210

[79] A. Gill. Type-safe observable sharing in Haskell. In *Haskell Symposium*, pages 117–128, 2009. pages 126

[80] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. pages 12, 53, 174, 197

[81] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik*, 38(1):173–198, 1931. pages 157

[82] H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer Berlin Heidelberg, 2006. pages 18, 132, 145, 146, 147, 150

[83] G. Gonthier. The four colour theorem: Engineering of a formal proof. In D. Kapur, editor, *Computer Mathematics*, volume 5081 of *Lecture Notes in Computer Science*, pages 333–333. Springer Berlin Heidelberg, 2008. pages 18

[84] G. Gonthier. Engineering mathematics: the odd order theorem proof. In *Principles of Programming Languages*, pages 1–2. ACM, 2013. pages 18

[85] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. technical appendix, 2011. pages 69, 160

[86] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *International Conference on Functional Programming*, pages 163–175. ACM, 2011. pages 69, 130, 132, 133, 153, 160

[87] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley Professional, 2005. pages 11, 15

[88] D. Grune and C. Jacobs. *Parsing techniques: a practical guide*. Springer-Verlag, second edition edition, 2008. pages 114

[89] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150. Springer Berlin Heidelberg, 2010. pages 25

[90] F. Haftmann and M. Wenzel. Constructive type classes in isabelle. In *International Conference on Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 160–174. Springer-Verlag, 2007. pages 67, 69

[91] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993. pages 18

[92] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Principles and practice of parallel programming*, pages 48–60. ACM, 2005. pages 15, 209

[93] B. Heeren, J. Hage, and S. Swierstra. Scripting the type inference process. In *ACM SIGPLAN Notices*, volume 38, 2003. pages 78

[94] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. pages 13

[95] R. Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10:327–351, 7 2000. pages 55

[96] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. In *Mathematics of Program Construction*, volume 2386 of *Lecture Notes in Computer Science*, pages 148–174. Springer Berlin Heidelberg, 2002. pages 55

[97] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. pages 9

[98] T. Hoare and P. O'Hearn. Separation logic semantics for communicating processes. In *International Conference on Foundations of Informatics, Computing and Software*, volume 212 of *Electronic Notes in Theoretical Computer Science*, pages 3 – 25. Elsevier, 2008. pages 15

[99] W. A. Howard. The formulae-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980. pages 11, 17

[100] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. pages 15, 34

[101] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming

language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992. pages 11, 12

[102] J. Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111, 2000. pages 127

[103] A. J. C. Hurkens. A simplification of Girard's paradox. In *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 266–278. Springer Berlin Heidelberg, 1995. pages 138

[104] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. pages 12, 209

[105] D. H. Ingalls. The Smalltalk-76 programming system design and implementation. In *Principles of Programming Languages*, pages 9–16. ACM, 1978. pages 24

[106] E. International. ECMAScript language specification. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`, June 2011. ECMA-262 Standard. pages 208, 209

[107] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *Programming Languages and Systems*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer Berlin Heidelberg, 2010. pages 8, 15

[108] A. Jeffrey and J. Rathke. Java Jr.: Fully abstract trace semantics for a core java language. In *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer Berlin Heidelberg, 2005. pages 25

[109] S. C. Johnson. YACC. *UNIX Programmer's Manual*, 2b, 1979. pages 4, 21, 23, 84, 130, 159

[110] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Functional Programming Languages and Computer Architecture*, pages 52–61. ACM, 1993. pages 15

[111] M. P. Jones. Type classes with functional dependencies. In *European Symposium on Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 2000. pages 38, 52, 56, 67

[112] W. Kahl and J. Scheffczyk. Named instances for Haskell type classes. In *Haskell Workshop*, pages 71–99. ACM, 2001. pages 37, 67

[113] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *International Conference on Functional Programming*. ACM, 2013. pages 209

[114] A. C. Kay. The early history of Smalltalk. In *History of programming languages—II*, chapter The early history of Smalltalk, pages 511–598. ACM, 1996. pages 6

[115] B. W. Kernighan, D. M. Ritchie, and P. Ejeklint. *The C programming language*, volume 2. Prentice-Hall, 1988. pages 11

[116] F. Kettelhoit. Towards a prelude for agda. Master's thesis, Ludwig–Maximilians–Universität München, 2012. pages 72

[117] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es), December 1996. pages 6, 25

[118] O. Kiselyov and C.-c. Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *Haskell Workshop*, pages 33–44. ACM, 2004. pages 39, 58, 205

[119] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: functional pearl. In *International Conference on Functional Programming*, pages 192–203. ACM, 2005. pages 132, 161

[120] R. Lämmel and S. Peyton-Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types in Languages Design and Implementation*, pages 26–37. ACM, 2003. pages 35, 130

[121] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *International Conference on Functional Programming*, pages 204–215. ACM, 2005. pages 35

[122] L. Lamport. *LaTeX: a document preparation system*. Addison-Wesley, 1986. pages 5

[123] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. pages 11

[124] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, pages 24–35. ACM, 1994. pages 15, 132, 161, 191, 193

[125] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Universiteit Utrecht CS, 2001. pages 85, 122, 124

[126] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. pages 17

[127] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: dynamic scoping with static types. In *Principles of Programming Languages*, pages 108–118. ACM, 2000. pages 67

[128] A. Lovelace. Translator's notes to an article on Babbage's Analytical Engine. *Scientific Memoirs*, 3:691–731, 1842. pages 2

[129] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Principles of programming languages*, pages 47–57. ACM, 1988. pages 15

[130] Z. Luo. ECC, an extended calculus of constructions. In *Logic in Computer Science*, pages 386–395. IEEE, 1989. pages 17

[131] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *S&P*, pages 125–140. IEEE, 2010. pages 14, 26, 166, 184, 185, 199, 208

[132] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for Haskell. In *Haskell Symposium*, pages 37–48. ACM, 2010. pages 130, 131, 132, 160

[133] J. Magalhães, S. Holdermans, J. Jeuring, and A. Löh. Optimizing generics is easy! In *Partial Evaluation and Program Manipulation*, pages 33–42. ACM, 2010. pages 123

[134] L. Magnusson and B. Nordström. The Alf proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer Berlin Heidelberg, 1994. pages 18

[135] I. Maier and M. Odersky. Deprecating the observer pattern with Scala.React. Technical report, École polytechnique fédérale de Lausanne, 2012. pages 17

[136] G. Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Haskell Workshop*, pages 73–82. ACM, 2007. pages 66

[137] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Haskell Symposium*, pages 67–78. ACM, 2010. pages 130

[138] S. Marlow. Haskell 2010 language report. `http://www.haskell.org/onlinereport/haskell2010/`, 2010. pages 149

[139] P. Martin-Löf. An intuitionistic theory of types. draft, 1972. pages 139

[140] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73 – 118. Elsevier, 1975. pages 17, 53

[141] C. McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 12(4–5):375–392, 2002. pages 18, 39

[142] C. McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer Berlin Heidelberg, 2005. pages 18

[143] C. McBride. Type-preserving renaming and substitution. draft, 2005. pages 136

[144] C. McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In *Workshop on Generic Programming*, pages 1–12. ACM, 2010. pages 53, 130, 131, 156, 161, 162

[145] C. McBride and R. Paterson. Functional pearl: Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008. pages 85, 127

[146] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960. pages 11

[147] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*. Springer, 1991. pages 103

[148] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed System Security*. The Internet Society, 2010. pages 26, 165, 199, 207, 208, 209

[149] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Security and Privacy*, pages 481–496. IEEE, 2010. pages 5

[150] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: a Functional Pearl. In *International Conference on Functional Programming*, pages 189–195. ACM, 2011. pages 88

[151] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Safe active content in sanitized JavaScript. Technical report, Google, Inc, 2008. pages 25, 26, 165, 199, 207, 208, 209

[152] M. S. Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006. pages 25, 165, 199

[153] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978. pages 13

[154] R. Milner. A proposal for Standard ML. In *ACM Symposium on LISP and functional programming*, pages 184–197. ACM, 1984. pages 12

[155] R. Milner. *The definition of Standard ML: revised*. The MIT press, 1997. pages 19, 204

[156] E. Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991. pages 14, 166, 208

[157] R. Moore. Removing left recursion from context-free grammars. In *Conference of the North American Chapter of the Association for Computer Linguistics*, 2000. pages 117, 124

[158] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *Partial Evaluation and Program Manipulation*, pages 117–120. ACM, 2012. pages 5

[159] J. H. Morris. *Lambda-calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, 1968. pages 11

[160] J. Nordlander, M. P. Jones, M. Carlsson, R. B. Kieburtz, and A. Black. Reactive objects. In *Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 155–158. IEEE, 2002. pages 199

[161] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, 2007. pages 18, 39, 42, 44, 46, 47, 61, 72, 73, 74, 75, 125, 131, 138

[162] U. Norell and C. Coquand. Type checking in the presence of meta-variables. Unpublished draft., 2007. pages 42

[163] K. Nygaard and O.-J. Dahl. The development of the SIMULA languages. In *History of programming languages I*, pages 439–480. ACM, 1981. pages 6, 24

[164] M. Odersky. The Scala language specification, version 2.8. `http://www.scala-lang.org/docu/files/ScalaReference.pdf`, 2010. pages 37, 67

[165] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer Berlin Heidelberg, 2003. pages 18

[166] B. C. Oliveira and A. Löh. Abstract syntax graphs for domain specific languages. In *Partial Evaluation and Program Manipulation*. ACM, 2013. pages 205

[167] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Object Oriented Programming Systems, Languages and Applications*, pages 341–360. ACM, 2010. pages 19, 37, 39, 55, 67

[168] B. C. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. The implicit calculus: a new foundation for generic programming. In *Programming Language Design and Implementation*, pages 35–44. ACM, 2012. pages 67, 71

[169] B. C. Oliveira, T. Schrijvers, and W. R. Cook. MRI: Modular reasoning about interference in incremental programming. *J. Funct. Program.*, 22(6):797–852, 2012. pages 167, 174, 199, 209

[170] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses. In *European Conference on Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin Heidelberg, 2012. pages 25, 198

[171] B. C. d. S. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. Extended report: The implicit calculus. Technical Report arXiv:1203.4499, arXiv, 2012. pages 72

[172] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. Effectiveadvice: disciplined advice with explicit effects. In *Aspect-Oriented Software Development*, pages 109–120. ACM, 2010. pages 198

[173] Oracle. Java core reflection. `http://docs.oracle.com/javase/1.5.0/docs/guide/reflection/spec/java-reflectionTOC.doc.html`, 1996. pages 23, 130

[174] D. Orchard and T. Schrijvers. Haskell type constraints unleashed. In *International Conference on Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 56–71. Springer-Verlag, 2010. pages 35, 55, 67

[175] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981. pages 90

[176] T. Parr and R. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. pages 4, 21, 23, 84, 159

[177] E. Pašalić and N. Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 136–167. Springer Berlin Heidelberg, 2004. pages 160

[178] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM, 2001. pages 127

[179] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989. pages 18

[180] S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering theories of software construction*, 180:47, 2001. pages 15

[181] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*. ACM, 1997. pages 67

[182] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, 2007. pages 35

[183] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming*, pages 50–61. ACM, 2006. pages 23, 35

[184] F. Pfenning. Elf: A meta-language for deductive systems. In *Automated Deduction — CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 811–815. Springer Berlin Heidelberg, 1994. pages 18

[185] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Automated Deduction — CADE-16*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer Berlin Heidelberg, 1999. pages 18

[186] B. Pientka and J. Dunfield. Programming with proofs and explicit contexts. In *Principles and Practice of Declarative Programming*, pages 163–173. ACM, 2008. pages 160

[187] B. Pierce. *Types and programming languages*. MIT Press, 2002. pages 3, 24, 102

[188] G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer Berlin Heidelberg, 2009. pages 16

[189] A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 93–107. Springer Berlin Heidelberg, 2008. pages 160

[190] F. P. Ramsey. The foundations of mathematics. *Proceedings of the London Mathematical Society*, 2(1):338–384, 1926. pages 11

[191] J. A. Rees. *A security kernel based on the lambda-calculus*. PhD thesis, Massachusetts Institute of Technology, 1995. pages 165, 199, 207, 208

[192] J. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing*, pages 513–523. North Holland, 1983. pages 12, 167, 175, 180

[193] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974. pages 12

[194] J. C. Reynolds. Polymorphism is not set-theoretic. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer Berlin Heidelberg, 1984. pages 12

[195] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE, 2002. pages 8, 9

[196] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *European Conference on Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 52–78. Springer Berlin Heidelberg, 2011. pages 14, 23, 130, 159

[197] A. Rodriguez, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *International Conference on Functional Programming*. ACM, 2009. pages 81, 96, 100, 101, 104, 105, 107, 124, 125, 130

[198] D. J. Rosenkrantz and P. M. Lewis. Deterministic left corner parsing. In *Switching and Automata Theory*. IEEE, 1970. pages 122

[199] J. Rudolph and P. Thiemann. Mnemonics: type-safe bytecode generation at run time. *Higher-Order and Symbolic Computation*, 23(3):371–407, 2010. pages 160

[200] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003. pages 10

[201] A. Sabry. What is a purely functional language? *J. Funct. Program.*, 8(1):1–22, 1998. pages 89

[202] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer Berlin Heidelberg, 2003. pages 25

[203] T. Schrijvers and B. C. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *International Conference on Functional Programming*, pages 32–44. ACM, 2011. pages 209

[204] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *International Conference on Functional Programming*, pages 51–62. ACM, 2008. pages 23, 35, 38, 52, 55, 67, 98

[205] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Computers and Automata*, volume XXI, pages 19–46. Polytechnic Press, April 1971. pages 3

[206] T. Sheard. Another look at hardware design languages. `http://www.cs.pdx.edu/~sheard/`, December 2005. pages 90

[207] T. Sheard. Putting Curry-Howard to work. In *Haskell Workshop*, pages 74–85. ACM, 2005. pages 18, 96

[208] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Haskell Workshop*, pages 1–16. ACM, 2002. pages 84, 122, 130, 132, 158, 159

[209] T. Sheard and E. Pasalic. Meta-programming with built-in type equality. In *Logical Frameworks and Meta-Languages*, 2004. pages 160

[210] J. Siek and W. Taha. A semantic analysis of C++ templates. In *European conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 304–327. Springer-Verlag, 2006. pages 71

[211] J. G. Siek. The C++0x "concepts" effort. In *Generic and Indexed Programming*, volume 7470 of *Lecture Notes in Computer Science*, pages 175–216. Springer-Verlag, 2012. pages 19, 67, 71

[212] J. G. Siek and A. Lumsdaine. A language for generic programming in the large. *Sci. Comput. Program.*, 76(5):423–465, May 2011. pages 19, 67, 71

[213] M. Sozeau and N. Oury. First-class type classes. In *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer-Verlag, 2008. pages 19, 45, 46, 47, 55, 67

[214] A. Stampoulis and Z. Shao. VeriML: typed computation of logical terms inside a language with effects. In *International Conference on Functional Programming*, pages 333–344. ACM, 2010. pages 130, 131, 132, 133, 160

[215] A. Stampoulis and Z. Shao. Static and user-extensible proof checking. In *Principles of Programming Languages*, pages 273–284. ACM, 2012. pages 130, 131, 132, 133, 160, 161

[216] M. Stiegler. Emily: A high performance language for enabling secure cooperation. In *Conference on Creating, Connecting and Collaborating through Computing*, pages 163–169. IEEE, 2007. pages 165, 199, 207, 208

[217] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System F with type equality coercions. In *Types in Language Design and Implementation*, pages 53–66. ACM, 2007. pages 12, 205

[218] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming*, pages 266–278. ACM, 2011. pages 18

[219] D. Swierstra. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development*, volume 5520 of *Lecture Notes in Computer Science*, pages 252–300. Springer Berlin Heidelberg, 2009. pages 85, 122

[220] S. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1996. pages 85, 127

[221] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18:423–436, July 2008. pages 124

[222] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. In *Theoretical Computer Science*, volume 248, pages 211–242. Elsevier, 2000. pages 130, 131, 132, 158, 160

[223] The Coq development team. The Coq reference manual. `http://coq.inria.fr/refman/`, 2012. pages 18, 67, 68

[224] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Programming Languages Design and Implementation*, pages 132–141. ACM, 2011. pages 5, 130, 159

[225] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997. pages 15

[226] M. van Dooren, D. Clarke, and B. Jacobs. Subobject-oriented programming. In *Formal Methods for Components and Objects*, volume 7866 of *Lecture Notes in Computer Science*, pages 38–82. Springer Berlin Heidelberg, 2013. pages 25

[227] M. Viera, S. Swierstra, and E. Lempsink. Haskell, do you read me? constructing and composing efficient top-down parsers at runtime. In *Haskell Symposium*. ACM, 2008. pages 125

[228] J. Voigtländer. Free theorems involving type constructor classes: Functional pearl. In *International Conference on Functional Programming*, pages 173–184. ACM, 2009. pages 13, 167, 174, 179, 180, 181

[229] D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. OutsideIn(X): Modular type inference with local assumptions. *J. Funct. Program.*, 21(4–5):333–412, 2011. pages 13, 178

[230] D. Vytiniotis and S. Weirich. Parametricity, type equality, and higher-order polymorphism. *J. Funct. Program.*, 20(2):175, 2010. pages 13, 198

[231] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989. pages 13, 167, 175

[232] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 12 1992. pages 14, 166

[233] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer Berlin Heidelberg, 1995. pages 14

[234] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming*, pages 63–74. ACM, 1998. pages 15

[235] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages*, pages 60–76. ACM, 1989. pages 19, 36, 67, 204

[236] D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog - the language and its implementation compared with Lisp. In *Symposium on Artificial intelligence and programming languages*, pages 109–115. ACM, 1977. pages 6

[237] S. Weirich. Replib: a library for derivable type classes. In *Haskell Workshop*, pages 1–12. ACM, 2006. pages 35, 67

[238] S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation. In *Principles of programming languages*, pages 227–240. ACM, 2011. pages 205

[239] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Logic in Computer Science*, pages 176–185. IEEE, 1994. pages 13

[240] A. N. Whitehead and B. Russell. *Principia mathematica*, volume 2. Cambridge University Press, 1912. pages 11

[241] Wikipedia. Gödel's incompleteness theorems, November 2012. pages 157

[242] H. Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17:215, March 2007. pages 18

[243] H. Xi and R. Harper. A dependently typed assembly language. In *International Conference on Functional Programming*, pages 169–180. ACM, 2001. pages 18

[244] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Types in Language Design and Implementation*, pages 53–66. ACM, 2012. pages 18, 36, 205, 206

[245] Y. Younan, W. Joosen, and F. Piessens. A methodology for designing countermeasures against current and future code injection attacks. In *Information Assurance*, pages 3–20. IEEE, 2005. pages 14

List of Publications

Articles in International Reviewed Journals

[1] Dominique Devriese and Frank Piessens. Instance arguments in Agda. *Higher-Order and Symbolic Computation*, 2014. Accepted.

[2] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal on Computer Security*, 2014. Accepted.

[3] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic abstract interpreters. *ACM SIGPLAN Notices*, 48(6):399–409, June 2013.

[4] Dominique Devriese and Frank Piessens. Finally tagless observable recursion for an abstract grammar model. *Journal of Functional Programming*, 22(6):757–796, November 2012.

[5] Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in Agda. *ACM SIGPLAN Notices*, 46(9):143–155, September 2011.

Contributions at International Conferences and Workshops, Published in Proceedings

[1] Jesper Cockx, Frank Piessens, and Dominique Devriese. Overlapping and order-independent patterns - definitional equality for all. In *European Symposium on Programming (ESOP)*. Springer-Verlag, 2014. Accepted.

[2] Thomas Winant, Dominique Devriese, Frank Piessens, and Tom Schrijvers. Partial type signatures for Haskell. International Symposium on Practical Applications of Declarative Languages, 2014. Accepted.

[3] Dominique Devriese and Frank Piessens. Typed syntactic meta-programming. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, September 2013.

[4] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic abstract interpreters. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI).*, pages 399–410. ACM, June 2013.

[5] Dominique Devriese, Ilya Sergey, Dave Clarke, and Frank Piessens. Fixing idioms: a recursion primitive for applicative DSLs. In *ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 97–106. ACM, January 2013.

[6] Tom Reynaert, Willem De Groef, Dominique Devriese, Lieven Desmet, and Frank Piessens. PESAP: a privacy enhanced social application platform. In *International Workshop on Security and Privacy in Social Networks (SPSN 2012)*, September 2012.

[7] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. Secure multi-execution through static program transformation. In *International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE)*, volume 7273 of *Lecture Notes in Computer Science*, pages 186–202. Springer Berlin Heidelberg, June 2012.

[8] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security (CCS)*, pages 748–759. ACM, 2012.

[9] Willem De Groef, Dominique Devriese, and Frank Piessens. Better security and privacy for web browsers: A survey of techniques, and a new implementation. In *International Workshop on Formal Aspects of Security and Trust (FAST)*, volume 7140 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2012.

[10] Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in Agda. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 143–155. ACM, September 2011.

[11] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. Reactive non-interference for a browser model. In *International Conference on Network and System Security (NSS)*, pages 97–104, September 2011.

[12] Dominique Devriese and Frank Piessens. Information flow enforcement in monadic libraries. In *ACM SIGPLAN workshop on Types in Language Design and Implementation (TLDI)*, pages 59–72, January 2011.

[13] Dominique Devriese and Frank Piessens. Explicitly recursive grammar combinators - a better model for shallow parser DSLs. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 6539 of *Lecture Notes in Computer Science*, pages 84–98. Springer, January 2011.

[14] Dominique Devriese and Frank Piessens. Non-interference through secure multi-execution. In *IEEE Symposium on Security and Privacy (SP)*, pages 109–124. IEEE Computer Society, May 2010.

Technical Reports

[1] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. Secure multi-execution through static program transformation: extended version. CW Reports CW620, Department of Computer Science, KU Leuven, April 2012.

[2] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. Reactive non-interference for the browser: extended version. CW Reports CW602, Department of Computer Science, K.U.Leuven, February 2011.

[3] Dominique Devriese and Frank Piessens. Explicitly recursive grammar combinators - the implementation of some grammar algorithms - technical report. CW Report CW594, Department of Computer Science, K.U.Leuven, September 2010.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMINDS - DISTRINET
Celestijnenlaan 200A bus 2402
B-3001 Heverlee
http://www.cs.kuleuven.be