Warsaw University
Faculty of Mathematics, Informatics and Mechanics

Mikołaj Konarski

Application of Category-Theory Methods to the Design of a System of Modules for a Functional Programming Language

Supervisor
Prof. dr hab. Andrzej Tarlecki
Institute of Informatics
Warsaw University

Author's Declaration

I declare that I have composed this dissertation myself.

Supervisor's Declaration

The dissertation is ready for review.

Abstract

Module systems embed elements of methodologies, formalisms and tools for programming-in-the-large into programming languages. In our thesis we design a module system, its categorical model and a constructive semantics of the system in the model. The categorical model is simple and general, built using elementary categorical notions, admitting many variants and extensions of the module system. Our module system features programming mechanisms inspired by category theory and enables fine-grained modular programming, as witnessed by the case studies presented.

We show how to extend our categorical model with data types expressible as adjunctions, as well as with (co)inductive constructions and with fully parameterized exponents. We analyze and compare alternative axiomatizations of the extensions and demonstrate a series of properties concerning rewriting of terms denoting morphisms of 2-categories; in particular, the terms expressing general adjunctions.

On the basis of the extended model and without introducing recursive types we define the mechanism of mutual dependency among modules, in the form of (co)inductive module construction. The extended module system contains the essential mechanisms of modular programming, such as type sharing, transparent as well as non-transparent functor application and grouping of related modules. Conventional modular idioms are expressible in new, original ways, enabling modular programming with no headers, no explicit module applications and no global modular errors.

ACM Classification

Categories and subject descriptors: D.3.3 [Programming Languages]: Language Constructs and Features—Modules, packages; D.2.2 [Software Engineering]: Design Tools and Techniques—Modules and interfaces; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—Lambda calculus and related systems; D.1.1 [Programming Techniques]: Applicative (Functional) Programming.
Additional key words and phrases: category theory, adjunctions, inductive types, recursive modules, inductive modules, type sharing.

Streszczenie

Systemy modułów wbudowują do wewnątrz samych języków programowania elementy metodologii, formalizmów i narzędzi, służących programowaniu w dużej skali. W naszej rozprawie projektujemy taki system modułów, jego kategoryjny model i konstruktywną semantykę owego systemu w tym modelu. Nasz kategoryjny model jest prosty i ogólny, zbudowany używając elementarnych pojęć teorii kategorii, dopuszczający wiele wariantów i rozszerzeń systemu modułów. Zaprojektowany system modułów charakteryzuje się mechanizmami programistycznymi inspirowanymi przez teorię kategorii i umożliwia drobnoziarniste programowanie modularne, jak świadczą zaprezentowane eksperymenty z jego użyciem.

Pokazujemy, jak rozszerzyć nasz kategoryjny model o typy danych wyrażalne jako sprzężenia w 2-kategoriach, jak również o konstrukcje (ko)indukcyjne i w pełni sparametryzowane obiekty wykładnicze. Analizujemy i porównujemy alternatywne aksjomatyzacje tych rozszerzeń i pokazujemy szereg własności przepisywania termów oznaczających morfizmy 2-kategorii; w szczególności termów wyrażających ogólne sprzężenia.

Na podstawie rozszerzonego modelu i bez wprowadzania rekurencyjnych typów definiujemy mechanizm wzajemnej zależności pomiędzy modułami w postaci konstrukcji (ko)indukcyjnego modułu. Rozszerzony system modułów zawiera zasadnicze mechanizmy modularnego programowania, takie jak dzielenie typów, przezroczystą jak i nieprzezroczystą aplikację funktorów oraz grupowanie powiązanych modułów. Popularne modułowe konstrukcje są wyrażalne na nowe i oryginalne sposoby, umożliwiając modularne programowanie bez nagłówków, bez aplikacji modułów i bez globalnych modułowych błędów.

Contents

I Introduction 11

1 Overview 12

1.1 Aim and Results 12

1.2 Previous work 17

1.2.1 Module systems 17

1.2.2 Models of module systems 19

1.2.3 Categorical models of core language 19

1.2.4 Term rewriting 21

1.3 Summary of the thesis 22

2 Dule tutorial 30

2.1 Core language mechanisms 30

2.1.1 Identity and types 32

2.1.2 Basic value operations 34

2.1.3 Functions and derived forms 45

2.2 Simple modular programming 51

2.2.1 Base modules 51

2.2.2 Linking 55

2.3 Advanced modular programming 59

2.3.1 Specifications 59

2.3.2 Module operations 61

2.3.3 Recursive modules 71

2.4 Conclusion 77

2.4.1 Absence of standard mechanisms 78

2.4.2 Additional ways of expressing 80

2.4.3 New expressible entities 83

3 Preliminaries 86

3.1 Prerequisites 86

3.2 Formal notation 87

3.3 Typical identifiers 94

II Semantics of Dule 97

4 Base categorical model 98

4.1 *LC* — Labeled Cartesian category . . . 98
4.1.1 Definition . . . 98
4.1.2 Language . . . 99
4.2 *2-LC* — 2-Labeled Cartesian category . . . 102
4.2.1 Definition based on *LC* . . . 102
4.2.2 Language . . . 103
4.3 *2-LC-lc* — *2-LC* with labeled products . . . 105
4.3.1 Definition based on *2-LC* . . . 105
4.3.2 Language . . . 107
4.4 Conclusion . . . 109

5 Semantics of the module system 112

5.1 Simple Category of Modules . . . 113
5.1.1 Objects and morphisms . . . 113
5.1.2 Identity and composition . . . 118
5.1.3 Products . . . 121
5.1.4 Equalizers . . . 123
5.1.5 Conclusion . . . 130
5.2 W-Dule — module system with specialization . . . 130
5.2.1 Language . . . 131
5.2.2 Algebra of modules . . . 134
5.2.3 Incorporating core language . . . 137
5.2.4 Cartesian structure . . . 144
5.2.5 Specialization, instantiation and trimming . . . 158
5.2.6 Conclusion . . . 171
5.3 L-Dule — module system with linking . . . 172
5.3.1 Language . . . 172
5.3.2 Grouping . . . 174
5.3.3 Linking . . . 177
5.3.4 Conclusion . . . 180

6 Instances of the base categorical model 182

6.1 p-Core — core language with product type . . . 183
6.1.1 Syntactic sugar for indexed lists . . . 183
6.1.2 Rewriting in *LC* . . . 185
6.1.3 Equality testing in *LC* . . . 189
6.1.4 Rewriting in *2-LC* . . . 193
6.1.5 Rewriting in *2-LC-lc* . . . 199
6.1.6 Conclusion . . . 203

6.2 s-Core — core language with sum type 204
6.2.1 Definition based on p-Core 204
6.2.2 Language 205
6.2.3 Rewriting 207
6.2.4 Conclusion 213
6.3 i-Core — core language with (co)inductive types 213
6.3.1 Definition based on s-Core 214
6.3.2 Language 214
6.3.3 Rewriting 218
6.3.4 Conclusion 228

III Variants and extensions 229

7 Variants of the base categorical model 230
7.1 Adjunctions 230
7.1.1 *2-LC-A* — *2-LC* with adjunctions 230
7.1.2 *2-LC-F* — *2-LC-A* by factorizers 235
7.1.3 Example adjunctions 237
7.1.4 Reduction in *2-LC-F* 244
7.1.5 Conclusion 250
7.2 Opposite categories 250
7.2.1 *2-LCO* — *2-LC* with opposite categories 251
7.2.2 Why exponent is contravariant 254
7.2.3 Attempt to accommodate parameterized adjoints 256
7.2.4 *2-LCX* — *2-LCO* with self-dualities 257
7.2.5 Conclusion 270
7.3 Parameterized adjunctions 271
7.3.1 *2-LCO-A* — *2-LCO* with opposite adjunctions 271
7.3.2 *2-LCX-A* — *2-LCX* with adjunctions 274
7.3.3 *2-LCX-F* — *2-LCX-A* by factorizers 287
7.3.4 Reduction in *2-LCX-F* 291
7.3.5 Conclusion 297

8 Extensions to the core language 298
8.1 e-Core — core language with parameterized exponent 298
8.1.1 Definition based on i-Core and *2-LCX-F* 298
8.1.2 Language 299
8.1.3 Construction using *2-LCX-F* 300
8.1.4 Moving back to *2-LC-lc* 306
8.1.5 Conclusion 308
8.2 f-Core — core language with general recursion 309

8.2.1 Language . . . 309
8.2.2 Rewriting . . . 310
8.2.3 Conclusion . . . 313
8.3 Extended notation for the core language . . . 315
8.3.1 Bare core language . . . 315
8.3.2 User core language . . . 331
8.3.3 Conclusion . . . 335

9 Extensions to the module language 336
9.1 I-Dule — module system with inductive modules . . . 336
9.1.1 Language . . . 337
9.1.2 Simple inductive modules . . . 339
9.1.3 Conclusion . . . 348
9.2 Extended notation for modules . . . 349
9.2.1 Bare module language . . . 349
9.2.2 Completeness of module typing . . . 361
9.2.3 Environment-dependent module language . . . 367
9.2.4 User module language . . . 374
9.2.5 Conclusion . . . 377

IV Conclusion 379

10 Final remarks 380
10.1 Theory . . . 380
10.2 Pragmatics . . . 382
10.3 Implementation . . . 384
10.4 Type reconstruction algorithms . . . 385

V Appendices 387

A Language summaries 388
A.1 Internal core language . . . 388
A.1.1 Syntax . . . 388
A.1.2 Semantics . . . 391
A.1.3 Typing rules . . . 396
A.1.4 Reduction . . . 400
A.1.5 Equality testing . . . 404
A.2 Internal module language . . . 406
A.2.1 Syntax . . . 406
A.2.2 Semantics . . . 407

A.2.3 Typing rules 409
A.2.4 Simulation of multiplication 411
A.2.5 Specification stripping 413
A.3 Syntax of the full user language 414

B Longer proofs from Chapter 6 and Chapter 7 417
B.1 Proofs for Chapter 6 417
B.2 Proofs for Chapter 7 433

C Program examples in Dule 442
C.1 Standard prelude 442
C.1.1 Module of natural numbers 443
C.1.2 Module of character strings 445
C.2 Dule compiler written in Dule 449
C.2.1 Assigning categories to functors 449
C.2.2 Implementation of SCM 452
C.2.3 Implementation of W-Dule and L-Dule 454
C.2.4 Implementation of I-Dule 466

Bibliography 475

Part I

Introduction

Chapter 1

Overview

1.1 Aim and Results

The goal of our research has been to find a good mathematical model for a module system in the style of Standard ML [120]. The model should lead to an intuitive semantics and consequently enable the programmer to readily understand and predict the operation of the module system. It should enable fine-grained modularization without prohibitive programming overhead. In particular, managing type sharing requirements [102] should be easy and debugging modular code ought to be fast.

The mathematical model should capture all the important aspects of modular programming: the construction of modules from the entities of the core language and various forms of grouping, instantiating and accessing defined modules. The semantics should be simple enough [129] to prevent confusion and paradoxes in reasoning about modules, especially those with many interdependent parameters. The set of operations should grant full control over the fine details of module hierarchy, at the same time enabling succinct notation for the most common idioms. The properties of the model should ensure soundness, abstraction, separate compilation and well localized error reporting [72].

The model of the module system should also grant insight into the nature of the fundamental modular mechanisms, such as the transparent and non-transparent functor application [103], grouping of related modules and mutual dependency of modules [32]. The shallow, computational understanding of common module operations should be enriched and systematized by recovering the operations inside an abstract semantic model. The model should not be a syntactic calculus imitating the computational behavior of conventional module operations, but an independent, abstract, mathematical construction. On the other hand, the model should be clear, simple and equipped with a natural computational interpretation, so that reasoning about modular programs and designing specialized compilation techniques for them are easy.

Importance

A move towards modular programming is necessary due to the growing size, complication and the requirement of modifiability of computer programs. The most useful approach appears to be the integration of rigorous module systems into high level programming languages, as in Standard ML or OCaml [105]. However, strictly modular programming style can be sustained only in small programming projects written in these languages. In large projects, managing the many layers of abstraction, introduced with the module hierarchy, turns out to be harder even than manual tracing of each particular dependency in unstructured code [138].

In our opinion, to have chances of usability in any scale, a module system must have a well designed and compact model. The model should be very simple so that it doesn't accumulate the complexity of module dependencies in a growing program. At the same time, the model should be strong enough that the simplifying abstraction can be sidestepped in a controlled way, if necessary. Having good syntactic sugar, default conventions and programming tools is not enough for a module system to be usable. A palatable and suggestive model is necessary so that literate programmers can think in terms of the model, which is crucial — at least when debugging code.

In order to advance the art of programming, we also need to experiment with new modular programming operations, with new schemas of compiling module collections, with new notions, methodologies and programming tools. Possibly, it is time that the programming at the level of modules be liberated from the functional style (a λ-calculus of modules) and moved to another style, weaker but safer and easier to use. Perhaps, this could be the categorical style of modular programming proposed here.

Obstacles

Programming inside a module is a well known and usually easy task. Yet, abstract and fine-grained modular programming is cumbersome, because the headers to be written are complicated and module applications often trigger global type sharing errors. When a module has n parameters, $O(n^2)$ potential typing conflicts await at each application, making not only creation of modules, but also their use and maintenance very costly.

Another problem is the tension between abstraction, expressiveness and applicability of a module. In simplification: the more abstract a specification of a module result is, the easier it is to implement the module, but the harder it is to use the module. Without specialized module system mechanisms, this global tension can only be solved by making each module excessively powerful and general.

However, the usual practical approach is to gradually sacrifice abstraction as the program grows, whereas, especially for ensuring correctness of large programs, the abstraction is crucial [17].

Huge collections of interdependent modules themselves require modularization, lest managing the modules becomes as tedious as tending the mass of individual entities in non-modular programs. Grouping of modules performed using the mechanism of submodules often overloads the programmer with type sharing bureaucracy or requires a violation of the abstraction guarantees. Other solutions, some of them using external tools, are usually very crude and incur the risk of name-space clashes or ignore abstraction.

Results

The construction of the main result of our thesis — the mathematical model of a modular programming language — is accomplished in two stages. First, we propose an abstraction of a core language (a language without modules) using the elementary notions of product and 2-category. Then, on top of the abstraction (the 2-categorical model of the core language) we define the model of the module system: the Simple Category of Modules. We prove the properties of the category and give the semantics of basic module operations in it. Such setup causes our module system to benefit from any subsequent studies of the core language model, its computational meaning and its extensions.

In the first stage, we develop the concept of a 2-cartesian category, as a basis of our categorical framework. This notion underlies every model of a core programming language and a module system we develop in our thesis. Using products alone we model type variables and value variables, that is: the dependency of programming language types on types, values on types and values on values.

In the second stage, we express programming modules in our underlying 2-categorical model and show that the set of modules with module composition forms a category — the Simple Category of Modules (SCM). Then we notice that SCM is cartesian and so we can capture dependencies among multiple modules. Moreover, there is enough equalizers (limits) in SCM to model type sharing specifications. For our module system we choose a type sharing mechanism that, together with the overall simplicity of the categorical module operations, eliminates the type sharing bureaucracy.

The compositionality of the semantics of our module system in SCM ensures separate compilability of modules. The lack of any references to environments in our semantic definitions results in complete abstraction of module parameters. However, when a module is composed with its arguments, the abstraction can be overcome with the use of adequate modular operations. The construction of the SCM upon a very general class of 2-categories shows that our module system has set-theoretic models [127] and ensures that the construction of our module system is independent of the core language, in particular it does not require function types nor second-order polymorphism [10].

In our thesis we construct equational theories and design confluent and strongly normalizing typed reduction systems for an array of extensions of our underlying 2-categorical model. In particular we construct and analyze reduction systems for our 2-categories extended with sum types [39] and (co)inductive types [67] with structured recursion combinators [22], which can serve as the main computational mechanism of the core language. Extension with a function type (exponent parameterizable by types on both the covariant and contravariant positions) is more difficult [48], so first we develop the notion of parameterized adjunction that subsumes and generalizes the operations of our underlying 2-categorical model. With the most sophisticated of our adjunction formalisms we manage to capture fully parameterized exponents, as well as pinpoint and isolate the places where its contravariance interferes [87] with (co)inductive types causing problems in reduction of the mapping combinator [45].

We use the parameterized adjunctions also for systematization and analysis of our equational theories and reduction systems. In particular, we propose rules related to η-equivalence and derive a set of rules for reducing multiplications by adjoint functors. The algebraic signature of the extended 2-categorical model (viewed as an algebra) together with the reduction system based on its equational theory is itself an interesting core programming language. To ease programming in this language we propose some syntactic sugar and notational conventions, provide syntactic typing rules and prove their properties; for instance, that each type inferred using the typing rules is expressible by the programmer as a closed type expression.

On the basis of our extended 2-categorical model we add mutually dependent modules to SCM. In their construction we manage to avoid the mechanism of recursive types, using (co)inductive types to recursively close the type part of modules. The use of (co)inductive types complicates the construction, but enables its verification in a relatively safe typing framework and opens way to structured recursion [114] over modules' types. To make the module system modeled by SCM usable by the programmer, we add derived operations to the language of SCM, enable pseudo-higher-order notation, introduce recursive signatures and provide signature inference, proving that all inferred signatures are expressible by the programmer.

The modular programming language Dule emerging from our research turns out to be quite rich and quite different from other programming languages. Due to its originality, Dule still has many rough edges. However, we are convinced we have overcome what we perceive the main obstacles to practical modular programming. The operations available in our module system provide for modular programming style with no mandatory module headers, no explicit module applications and no hard to localize module errors. The model of our module system facilitates viewing the same module with different levels of abstraction. A new methodology resulting from our study of inductive types grants precise control over the concreteness of module interfaces. The grouping of modules can be done in several powerful and specialized ways, without verbose notation and without sacrificing abstraction.

The semantics of Dule turned out to be simple and constructive enough that the design of a Dule compiler straightforwardly implementing the semantics was possible; see Appendix C.2 and Dule homepage at `http://www.mimuw.edu.pl/~mikon/dule.html`. Consequently, our thesis may be seen as a proof of correctness of the compiler (type-correctness of the generated code and other fundamental properties). On the other hand, the Dule language proved powerful enough to facilitate writing, in an extremely modular and yet concise manner, various example Dule programs, including the main parts of Dule compiler itself. Further large-scale case studies of fine-grained modular programming, made possible by our module system, should uncover any remaining problems emerging from our methodology and suggest improvements in practical usability. The flexibility and extensibility of our module system give hope that one of its future variants will mature being used in practical large-scale projects.

The extensive summary of the thesis is given in Section 1.3. In Chapter 10 one can find discussion of future work, both theoretical and practical.

Acknowledgments

I would like to thank every person, who asked me about my work during these many years. I am grateful to everyone who helped me with advice, criticism or coding, especially to my Ph.D. advisor, Andrzej Tarlecki, but also to Patryk Czarnik, Michał Gajda, Marek Zawadowski, Stasiek Skowron, Jacek Chrząszcz, Andrzej Gąsienica-Samek, Piotr Roszkowski, Paweł Findeisen and Aleksander Nałęczyński.

I would like to thank the authors of Charity for their brave experiments with category theory in programming:

`http://pll.cpsc.ucalgary.ca/charity1/www/home.html`

and the authors of OCaml:

`http://caml.inria.fr`

for a practical, powerful, free programing language.

1.2 Previous work

1.2.1 Module systems

Module systems implement and extend the idea of abstract data types [144] and some other less fundamental notions and methodologies of algebraic specifications [134]. An implementation of the simplest kind of abstract data types appears, among the first serious programming languages adopting it, in Modula-2 [145]. Modula-2 modules are first-order, in the sense that parameters (imports) of a module cannot be regarded as parameterized modules themselves and, for instance, applied to different argument modules within the body, before being used. In fact, in Modula-2 there is even no syntactic construct of module application. The initial module system of OCaml [105] (in recent versions governing only the linking of modules identified with program files) is one of many module systems similar to the Modula-2 system. Such systems assure type abstraction themselves, but depend on the features of a file system and a smart linker for the other modular programming tasks.

From theoretical point of view, the main weakness of the Modula-2 module system may seem to be the lack of functional completeness — the first-order nature of the system. When the module system of Standard ML [119] was introduced, its main novelty was the introduction of functors, that is, named parameterized modules that can be explicitly applied to different arguments, though not taken as parameters. The full OCaml module system [103], as well as other contemporary proposals [41] feature higher-order modules — functors can be both taken as arguments and returned as results of modules. However, regardless their power, functors are considered unfit to be used as a basis of large-scale modular programming [138] as seen in their scarcity in the code of the OCaml [106] and Standard ML of New Jersey [111] compilers themselves.

The Standard ML functors are used both to express dependencies among modules (mostly to limit those dependencies by ensuring abstraction) and to enable code reuse [47]. The code reuse situations do not occur often, compared, e.g., to program modification by hand, caused by bug fixing. Moreover, from the very nature of the program design process, nontrivial code reuse usually requires an additional rewrite of important parts of the code [9], regardless of the helper language mechanism used. This may partly explain why standard tools for checking correctness of completed, rewritten code are essential in practice, while functors and other specialized mechanisms are not. For rewriting a program, precise indication of the area of impact for each modification is crucial. Functors help in such analysis, but in case of large programs the cost of creating and updating the functor description of dependencies is prohibitive. The problem lies mainly in the extensive bureaucracy of sharing equations used to indicate interdependencies of functor's arguments. The introduction of module systems employing a particularly simple form of sharing equations — type abbreviations [102] — helped in readability of module headers, but we are not aware of any efforts succeeding in reducing the sheer volume of module bureaucracy required.

What we perceive to be the main shortcoming of Modula-2 and similar module systems is not their simple first-order nature or lack of built-in code reuse features, but its lack of adequate module isolating and relating mechanisms. This prevents fine-grained modular programming, because there is no point in subdividing modules when the components cannot be well isolated and when their interrelations are not expressed using specialized mechanisms of the module language, but remain at the core language level. The identification of modules with files results in cluttered name-space, which may be remedied using file directories, but this brings in its own complications. This problem is related to the lack of module grouping operations and mechanisms for hiding modules or their portions in some parts of the program and revealing them in others. Yet, the biggest problem is that every module depends on a global pool of modules, their types, their sharing, and therefore every local change necessitates an inspection of all implementations in the pool. Although Modula-2-like modules are already analyzed and improved for decades [143], we don't know of any module system providing benefits comparable to that of Standard ML, but avoiding its conceptual complexity and practical limitations.

For code reuse, interesting and promising approaches seem to be the Haskell class/monads system [108], strongly typed object oriented systems [121] and mixin module systems [2]. However, they are very complex for the quality of modularization they provide and, indeed, they have different goals than aiding creation and maintenance of modularized programs. Other aspects of modular programming not touched upon in our thesis are first-class modules [128] and effects (imperative features, generativeness, module initialization) [40]. Such language mechanisms, while not indispensable, are expected to gravely complicate semantic models, especially categorical models and especially as elementary as ours.

Recently, an area of active research is mutual dependency among modules [32]. Most of the proposed systems involve higher-order modules and are therefore inherently complicated. Moreover, they require, implicitly or explicitly, the powerful mechanism of recursive types [117] in the core language and/or type reconstruction. As far as we know, ours is the first attempt to base mutually dependent modules on the well-understood, fundamental categorical notion of (co)algebras or (co)inductive types, instead of recursive types. In connection with recursive modules, there is also ongoing work on understanding and implementing arbitrary fixpoint values that emerge when the modules are compiled [74]. The results of this ongoing work are well ahead of the naive approach sketched in our thesis and, in time, they will surely be used to improve our experimental implementation.

1.2.2 Models of module systems

Theoretical underpinnings of algebraic specifications usually focus on describing classes of modules and operations on those classes. Individual modules are often treated as points, without accounting for their construction, hierarchy, interaction and use, which are either glossed over or described informally as elements of methodology or unimportant technical details. CASL [26] architectural specifications are an exception in that the process of module construction is captured within the specification language, though still very abstractly and generally [17].

The module system of Modula-2 has never been described formally. The whole Standard ML is formally defined [119] using Natural Semantics — a kind of operational semantics. There is an ongoing work on describing modules in less operational manner: type abstraction is modeled by existential types [122], phase distinction [72] analyzed and related to dependent types. However, dependent types in their full generality are now widely known to be nonessential for modelling module languages [129] and to introduce additional complexity to semantics and type reconstruction of core languages, though some advocate their use in both roles [6].

On the basis of the work on modelling module systems, OCaml modules [102] and Standard ML modules [71] have been formalized by translation into some, more or less standard, type theory systems. The basic semantics of type theory systems is operational, but there is a long history of denotational semantics, with models ranging from those of classical domain theory to various elaborate categorical constructions. All these efforts together provide mathematical models of module systems, though indirect and complex.

Another approach to modelling programming languages is by proposing direct denotational semantics into a simple categorical model. The approach has been successful with respect to (co)inductive datatypes [67] leading to the creation of the categorical programming language Charity [25]. However, among many extensions to simple categorical models for the core language we have not found any encompassing a module system.

1.2.3 Categorical models of core language

The categorical model of Charity, based on the notion of strong categorical datatypes [24] is relatively simple and fundamental, though not elementary, which incurs the risk of missing the potentially most important audience — that of programmers that are literate but not versed in category theory. The basic construction generalizes both (co)algebras and (a form of) adjunctions, and the framework is described in terms of strength and fibrations. Initially without polymorphism and function types, throughout the years the language has been extended with some forms of these and other useful features. However, the extensions involving the categorical model are difficult technically and restricted in generality in order to fit the fixed categorical framework imposed to make the initial kernel constructions as comprehensive and powerful, as they are.

One of the first successful approaches going in the direction from an already existing polymorphic type system to a categorical construction, models the whole system F using dinatural transformations [10]. Despite the complications incurred by the type quantification in system F, the core of the resulting categorical construction is very similar to our *2-LCX* with morphisms of mixed variance (Section 7.2.4). This is surprising considering that our framework was developed using the reverse approach, that is, starting with an elementary categorical model and then designing typing rules for it.

A more recent work in a similar spirit [48], while preparing ground for categorical modelling of recursive types, captures the framework for functors of mixed variance using the notion of involutory categories, symmetric functors and their generalizations. Again, despite our different goals and starting point, our restriction of carriers of *2-LCX* in Definition 7.2.4 turns out to determine exactly the set of symmetric functors between the categories of *2-LCO*. However, we are not aware of any attempts in the literature to recover the (nearly) 2-categorical structure in the category of involutory categories and symmetric functors (not to be confused with pseudo involutory 2-categories [19], which are involutory categories themselves). Consequently, we have not encountered in the literature any generalization of the adjunction construction to a framework of functors with mixed variance nor any analysis of rewriting of transformations between such functors.

In contrast to the fixed framework of dinatural transformations and involutory categories, the mode of our development makes it constructive; we extend elementary categorical concepts with explicit new operations in a systematic way. This results in a general framework, so that for instance, function types emerge as a special case of parameterized adjunctions. The constructivity also creates an immediate link between theory, pragmatics and implementation of our models of programming language concepts. Another advantage is that the elementary starting notions remain a valid approximation of the whole model, so that category theory can be learned alongside the learning of the programming language and, given creative tutorials, even from the programming language itself — a characteristic of which the Charity programming language is a pioneering and unrivalled example [22] and which currently also seems to apply to Haskell monads [93].

The Squiggol style of programming [114] is more a methodology for deriving programs from specifications than a programming language, and more a set of algebraic laws than a model. It has been given a strict categorical semantics in order to extend it from finite lists to other data types. Then the categorical model was refined [115], using the notion of dinatural transformations, to allow function types. While excellent in advocating the use of structured recursion, the

Squiggol approach lacks generality. The used languages can only express a small part of the model and various restrictions have to be used [45] to maintain the validity of algebraic laws whenever the expressiveness is increased.

Covariant types [87] with semantics in data categories [85] are yet another approach to modeling a programming language. They constitute a very interesting subsystem of system F — one that can be modeled categorically in particularly easy way, despite admitting a (restricted) use of function types. Then the subsystem is easily extended with structured recursion and no collision with the function types emerges. The system provides an interesting extension to a conventional functional programming language (Functorial ML [88]) and there are convincing examples of the use of similar ideas for mixed functional-imperative programming with arrays [84]. However, the restrictions on function types are probably too severe for practical applicative programming using just the covariant types. We were inspired and encouraged by the original notion of covariant types while designing our pseudo-higher-order specifications for modules. Viewed as functions on types our specifications are covariant, despite their unrestricted syntax that helps in expressing type sharing.

1.2.4 Term rewriting

The reduction system for categorical combinators [34] is an untyped rewriting system for the language of cartesian closed categories. The terms have no variables, so the system is easily implemented by low-level abstract machines [31]. Confluence [68] and strong normalization [70] of interesting subsystems have been proved and our work is influenced by these results. The whole untyped system of categorical combinators is known to be not normalizing and not confluent.

Simply typed λ-calculi corresponding to the cartesian closed categories, which entails various forms of extensionality rules, have been studied [89], proved confluent and strongly normalizing even when enriched with (non-extensional) sums [39] and shown decidable with added extensionality rules for coproducts [58]. These calculi are not based on combinators, unlike our system, but they are typed, just as our system. This allows us to draw ideas from proofs of their properties, especially with respect to termination.

As far as we know, there are no studies of rewriting systems for any languages of 2-categories, for any presentation of general adjunction operations nor for any instances of generalized adjunction mechanisms in the presence of functors of mixed variance. The technique of adjoint rewriting [57] is successfully used to analyze rewriting of interesting instances of standard adjunctions, but the resulting systems, while complete, are untyped and their reduction is restricted by contexts.

1.3 Summary of the thesis

Before a serious study of our thesis the reader is invited to enjoy a taste of the resulting programming language Dule by browsing the informal tutorial in Chapter 2. We also invite the reader to experiment with our implemented Dule compiler, which is freely available at its homepage. A version of the compiler code [97] tested against Dule examples included in our thesis is kept at Dule homepage in directory `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/` and a snapshot of the whole Dule homepage is enclosed on CD.

We do not describe the Dule compiler in our thesis, but most of its code is a straightforward encoding of the constructive semantics of Dule in an abstract categorical model, developed in the thesis. Our compiler, after type and specification reconstruction, computes the semantics of modular programs in the categorical model and verifies its type-correctness, almost literally following formal semantic definitions. All example programs given in our thesis compile successfully to the language of our base 2-categorical model and yield the expected results through combinator reduction developed here and implemented in the compiler. The immediate practical use of our theoretical results in such a framework is fascinating. Moreover, such an encoding in a programming language offers possibilities to test, experiment with, simplify and fine-tune both the categorical models and the language semantics developed in the thesis.

In Chapter 3 we sketch the notational conventions adopted in our thesis and explain how to read our meta-notation for semantic definitions. Returning to this chapter as well as advancing to Appendix A containing language summaries may be helpful whenever notational difficulties arise. In Appendix B there are some longer proofs omitted in the main body of our thesis. Appendix C contains an assortment of example Dule program fragments, including some of the main parts of the Dule compiler written in Dule.

Summary of Chapter 4

Part II contains the categorical foundations and the principal semantic descriptions of Dule. Chapter 4 brings in the concept of a 2-cartesian category that underlies every model of a programming language or a module system we develop in our thesis. Within a 2-cartesian category, just as in any 2-category, there are three kinds of categories: the underlying category (of 1-morphisms and objects), the categories of 2-morphisms (as morphisms) and 1-morphisms (as objects) with the vertical compositions and the category of 2-morphisms and objects with the horizontal composition of 2-morphisms. The richest of our constructed categorical models, *2-LC-lc*, has products in all three kinds of categories. The products model the mechanism of a core programming language identifier or variable (both type variables and value variables). Having all the products enables us to capture the notions of dependency of programming language types on types, values on types and values on values.

Even more importantly, the products in the core language are crucial for our module system. We want a hierarchy of modules to capture the dependency relationship of program fragments. Moreover, due to type abstraction, the dependency relation cannot be expressed in a shallow way, such as by compilation order or file inclusion. We need modules parameterized on other modules and this, even in the weak categorical form we promote, requires the comprehensive product machinery provided by our underlying 2-categorical model *2-LC-lc*.

Summary of Chapter 5

In Chapter 5 we construct and start using the Simple Category of Modules (SCM). In Section 5.1 we model a system of modules as a category — the SCM. Signatures are objects of the SCM, modules are morphisms, parameters of a module are described in its domain, the result in its codomain and composition of modules is modeled as the composition in the category. We demonstrate that SCM is cartesian. Therefore, similarly as with the core language, we are able to model dependency of a module on multiple named modules and we can express module variables as projections. Moreover there is enough equalizers (hence limits) in SCM to model type sharing specifications. As building blocks for the SCM we use solely the morphisms of an arbitrary but fixed underlying 2-categorical model *2-LC-lc*. Because objects and morphisms of the SCM are just (tuples of) morphisms of the *2-LC-lc*, it is easy to extend the SCM with additional operations defined using the language of *2-LC-lc*, provided that proper semantic invariants are maintained, so that we stay within the category.

In Section 5.2 we build a basic, but quite comprehensive module system W-Dule (named after the Dule keyword `with` denoting instantiation and specialization) by extending Simple Category of Modules with several new operations. We argue that each case of partiality of any of the new operations corresponds to a class of modular programming errors. We prove that the semantics of the operations is well defined; in particular, their results belong to the set of objects and morphisms of the SCM. We check that the declared parameter and result signatures of W-Dule modules coincide with domains and codomains of the morphisms the modules denote in the SCM.

The semantics of W-Dule in SCM is compositional and environment-free, which implies that the modules are separately compilable. The correctness and result of a module operation depend only on the target code (SCM morphisms) obtained from the operands, so the original W-Dule source code of the operands can be compiled only once and then forgotten. The lack of any environments ensures that module parameters are abstract. Upon supplying arguments the abstraction can be retained or overcome, depending on the operations used.

The modules of W-Dule are first-order and not recursive. This simplicity makes it possible to entirely embed them, via SCM, into the *2-LC-lc* that is a model of their core language. The fact that the *2-LC-lc* underlies the entire module system allows W-Dule to benefit from theorems about *2-LC-lc* and tools developed for it. Moreover, the fact that *2-LC-lc* has **Set**-based models implies that the module language W-Dule has a set-theoretic semantics. Interestingly, *2-LC-lc* does not feature exponents (function types) and so our module system, in principle, does not depend on their presence.

In Section 5.3 we construct an extended module system L-Dule (named after the operation of module linking). Its operations are expressed using the operations of W-Dule, which witnesses the power of the W-Dule module system, as well as greatly simplifies proving properties of L-Dule. In particular the proof of well-definedness of the new operations of L-Dule and the adequacy of their declared domains and codomains is trivial. As before, the Simple Category of Modules underlies the extended module system. The categorical structure ensures that every module of the richer system still has exactly one domain and one codomain signature and there is no need for the complicated operations of matching and enrichment. Moreover, categorical composition together with categorical product are still the most used module operations (although in this system they are used implicitly, most of the time).

The new operations of L-Dule ease the grouping of modules, performed in W-Dule only with the primitive operation of module record. One of the new operations — linking — additionally automates supplying implementations of parameters, by implicitly performing multiple compositions. The compatibility of the implicit composition used in linking, on one hand, and our sharing conventions, on the other hand, eliminates the need to artificially introduce submodules for the purpose of specifying sharing between modules. The linking operation facilitates naming parameterized modules and applying named modules, but the language remains first-order and avoids copying of code.

Summary of Chapter 6

Chapter 6 is devoted to an analysis of the language of our 2-cartesian categories as a rewriting system. The result of this analysis is the basic execution model for our programs. In Section 6.1 we construct the equational theory of *2-LC-lc* and a confluent and strongly normalizing typed reduction system based on the theory and agreeing with programming intuitions. Together with an equality testing procedure for the 1-morphisms of the *2-LC-lc* constructed from terms, this forms a basis for a programming language. Equipped with a module system and amended with some syntactic sugar the language is definitely a high-level programming language, despite its fine-grained control over substitution and type manipulation.

In Section 6.2 we prove the equivalence of two axiomatizations for *2-LC-lc* with sums (coproducts distributive over products). Distributivity makes the language's values parameterizable and so suitable for inclusion into modules. Distributivity is also essential for the expressiveness of the core language itself. One of the two equivalent axiomatizations is an extension of the standard presentation of non-distributive coproducts, the other captures the distributive coproduct operation from a programming perspective and is a basis for our reduction system. We discuss the role of η-equations in rewriting coproducts and prove the reduction system to be confluent and strongly normalizing.

In Section 6.3 we prove several important properties of the mapping operation and, guided by them, we construct a reduction system for the (co)inductive operations in the spirit of Charity [54]. The system has considerable computational strength and is used as a basis of our core programming language. Our core language should not resort to a meta-theory for mechanisms such as substitution or instantiation; everything should be modeled internally and explicitly. By building the equational theory of (co)inductive types on the basis of the multiplication and mapping operations we adhere to this principle. We capture polytypism (generic structured recursion) purely at the semantic level, without any meta-programming mechanisms.

While a programming language with such a degree of explicitness is not usable by humans without some syntactic sugar, it is perfectly amenable to modularization. The notions of dependency needed by a module language are here neither vague theoretical concepts nor mechanisms idiosyncratic to a favored execution model, but are concrete parts of the core language itself, general but necessarily compatible with whatever may be expressed in the language.

Summary of Chapter 7

In Part III we describe those extensions and variants of our programming language Dule that are significantly useful or interesting, yet not considered the essence of Dule. In Chapter 7 we generalize our base 2-categorical model using the notion of adjoint and proposing the notion of contravariance morphism and parameterized adjunction. The generalization will not be implemented in the user-accessible language, but it offers insight and systematization into construction of equational theories and reduction systems for our categorical models. The developed theoretical tools can be used both to carry on with the extension of our core language and to improve the execution mechanism.

In Section 7.1 we prove the equivalence of the two equational presentations of adjunctions we use in our thesis: the standard presentation, based on units and co-units, and an alternative presentation, focused on factorizers. Both equational theories are constructed in an equational framework designed to make them as strong as possible, despite partiality of adjunction operations that hinders the use of transitivity rule of equational reasoning. The strength of the theories makes it easy to instantiate them to particular categories with fixed sets of defined adjunctions.

We demonstrate on several examples how the factorizers together with other operations generalize the basic mechanisms of a core programming language. We show how the theory of adjunctions provides correctness criteria for reduction systems and helps in assessing their completeness and comparing their subsystems. In particular, we conjecture that in the context of the theory of adjunctions, the β-rule for exponents is weaker than the β-rules for products and coproducts. The axioms of the theory of adjunctions can also suggest new reduction rules for core language constructs, as well as additional language mechanisms such as exceptions treated as dual to variables, etc.

We offer analysis and partial solutions to the rewriting problem for general adjunctions. We propose a general reduction rule for rewriting multiplication by the adjoint functor. We discuss rules corresponding to η-expansion and prove that using exclusively weak η-rules for right adjoints as well as using exclusively weak η-rules for left adjoints results in a confluent reduction system, while using both leads to unsolvable critical pairs.

Categorical frameworks, capable of accommodating exponents fully parameterized by types, tend to be complicated because of contravariance. In Section 7.2 we prove several properties of such a framework we propose, showing that it can be used in most cases in the same manner as our simple categorical models. For other cases we provide operations and equations allowing one to express and use contravariance. In our framework it is possible to express source and target categories of any generalized adjoints parameterized by types, in particular of the fully parameterized exponent, as well as domain and codomain functors of the units and co-units in the respective adjunctions.

In the very technical Section 7.3 we develop the language, theory and reduction system analogous to that of Section 7.1 but for adjoints to functors that may be instantiated even after the adjoints are applied. Just as for non-parameterized adjoints, after introducing units and co-units we extend the language with factorizers and provide an equivalent alternative equational theory based on them, proposing reduction rules for multiplication by adjoint functors and η-equivalence. The weak η-rules turn out to be moderate generalizations of their counterparts for non-parameterized adjunctions. The rules for multiplication are considerably more complicated, just as the parts of equational theory used in their proof.

Summary of Chapter 8

In Chapter 8 we extend the core language of Dule. In Section 8.1 the core language with (co)inductive types is extended with exponents parameterizable on both the covariant and contravariant positions. The exponent construction is an instance of our general mechanism of parameterized adjunctions, which itself is not a part of the core language. We identify the operation in which the exponent shows its contravariant nature, to be the multiplication by exponent functor from the right. Using the properties of parameterized adjunctions we manage to express the typing of the operation in the core language. The construction of the rewriting rules for parameterized exponents benefits from our results about contravariance and parameterized adjunctions, but at the same time uses solely the syntax of the core language.

In the presence of inductive types, problems arise with rewriting terms that involve exponent constructors. Our reduction system makes it possible to tackle the problems in various ways by providing reduction rules for the operation of mapping (a distributive analogue to multiplication by a functor). It facilitates easy experimentation with various sets of such reduction rules, while precluding adverse effects on the rewriting of other combinators.

We base our reduction system for parameterized exponents on the theory of the core language with (co)inductive types extended with several most computationally important equations pertaining to parameterized adjunctions instantiated for exponents and expressed in the syntax of the internal core language. In particular, we manage to translate the weak η-equation for parameterized right adjunctions and to sidestep the preconditions of the general reduction rule for multiplication by a right adjoint. The resulting language is adequate for use as a basis for our module system, since all operations including exponent can be fully parameterized and, at the same time, the model closely resembles *2-LC-lc*.

In Section 8.2 we add general recursion to the core language. The internal core language extended with exponents and fixpoints is the final step in the series of extensions of *2-LC-lc* developed in our thesis. Its reduction system is no longer normalizing, but it remains locally confluent and we conjecture it is confluent as well, guaranteeing unique normal forms for normalizing terms. The language is the richest core language instance for our base module system and the starting point for the module system extended with (co)inductive modules (Section 9.1), which require inductive types and a fixpoint operation on values.

In Section 8.3 we develop the bare and the user languages that make programming in our internal core language more comfortable. We provide typing of the bare language terms and prove it is compatible with the typing of their semantics in the internal language. We show that the inferred type of every value is expressible, moreover it is expressible as a closed type expression.

Similarly as with the internal language, the typing of the bare language values assigns to a term not only its codomain type but also its domain type. The domain models the types of values in the environment. In this setting we model type variables and value variables as labeled projections of the internal language. We also design a uniform way of supplying operands to coproduct, (co)inductive and mapping combinators. There is a lot of substitution notation in the typing of these combinators but there is no need for a syntactic substitution in our languages. We prove that the composition in our categories faithfully and semantically implements the type substitution or, in other words, the language of types is referentially transparent. We also show referential transparency of the language of values, demonstrate that our language has no principal typing property and discuss the complexity and practical usability of our type reconstruction algorithm that infers minimal types for values.

Summary of Chapter 9

Chapter 9 describes extensions to the module language. In Section 9.1 we describe inductive and coinductive modules defined using inductive and coinductive types, respectively. The new constructions enable our module language to express mutual dependencies among modules, which is crucial for our programming style with numerous small modules. With the help of (co)inductive modules, the mutual dependencies among core language entities scattered in separate small modules can be dealt with entirely at the module language level.

The main (co)inductive module operation performs an (up to observational indistinguishability) fixpoint closure of a single module. Every step of the recursive closure of a single module is expressed in the internal core language and so is fully typed. By careful construction we ensure that the resulting module satisfies the module language invariants. The approach with full typing enables us to pinpoint the problems caused by the interaction between (co)inductive closure of types and multiplication of values by an arbitrary functor from the right (for example by the exponent functor, which is not needed for our construction but is present in the core language and adds to the usability of modules). We overcome the problems without resorting to recursive types and any other forms of identifying similar types or bypassing type verification.

The more general module closure operation, designed for multiple mutually dependent modules, is reduced to the module fixpoint using operations of the basic module language W-Dule. The general (co)inductive module operations have the same operands with their typing relations as the operation of linking modules and, similarly, preserve separate compilation of the operand modules. There is no need to extend the signature inference for the cases of (co)inductive module operations, because the typing is built into the category of modules and not defined ad hoc on the set of module terms. The simplicity and strictness of the categorical model provide perfect ground for making fundamental design choices, such as the semantics of mutually dependent modules operations. However, a programmer needs an extensive array of operations, default conventions, syntactic sugar, programming tools, and these are best designed in an outer denotational layer of the semantics, not in the categorical kernel.

In Section 9.2 we define three extended module languages and their semantics using a relaxed, denotational approach with semantic domains built from those of the internal module language. With the first of the extended languages, the bare module language, we introduce the notion of a module specification. A specification is a joint notation for the domain and the codomain of a module. It expresses, in clearer and more suggestive way, the usual dependency of values of the codomain on the types of the domain. Surprisingly, specifications can be nested, despite the lack of exponential object in our module category. We provide a translation, called saturation, which flattens the pseudo-higher-order specifications, while preserving their type dependencies. Using the saturation we also manage to resolve mutual dependencies of recursive specifications by enriching each of them with types of all others.

We discuss the notion of the least detailed specification of a module and our specification reconstruction algorithm, which finds (if successful) such specifications. We show that every specification derivable for a module is saturated and expressible in the bare module language. We demonstrate that any specification inferred for a bare language module agrees with the domain and codomain of the internal language semantics of the module. We prove that due to not including module composition in the bare language, almost everything that type-checks in the bare language has an internal module language semantics.

The environment-dependent module language allows the programmer to name specifications and libraries, providing a limited form of code reuse. The user module language provides some more syntactic sugar. Both these languages are easily translatable to the bare module language without performing typing of modules. The introduced shorthands and default assignment of specifications, together with the linking operation and implicit type sharing, almost totally eliminate module headers, which in our preferred fine-grained modularization style might easily constitute more than a half of the program code.

Chapter 2

Dule tutorial

The variant of Dule we present in the tutorial and in the example programs of Appendix C is quite a usable functional programming language with a very rich module system. Some characteristic properties of this programming language are discussed in the conclusion of this chapter. The core language is described technically throughout the thesis concluding in Section 8.2 and is summarized in Appendix A.1. The module language is described formally in Chapter 5 and Section 9.1 and is summarized in Appendix A.2. For both languages we use extended notations as described in Section 8.3 and Section 9.2. Here we give a lighter, informal presentation based on the grammar of the user language (the language accessible to the programmer) presented in Appendix A.3.

The main highlight of Dule is its module system, especially when used to manage large collections of very small modules. However, to analyze any longer modular examples, such as those discussed in Appendix C.2, we have to understand the meaning of particular modules. The basic modules, in turn, are constructed using the core language of Dule, which is unconventional both in its choice of mechanisms and in their relative semantic significance, and so requires a detailed description. For example, despite being a functional language, the core language features function types as an extension, while categorical composition is considered the fundamental computational engine. The reader is asked to bear with us whenever we inspect the main mechanisms of the Dule language in the order of their semantic significance and not necessarily in the order of their appearance in the example language phrases.

2.1 Core language mechanisms

Remark. Every example given in this section can be successfully type-checked and compiled using the Dule compiler. The archive with the compiler code can be obtained from `http://www.mimuw.edu.pl/~mikon/Dule/download/dule-src-phd.tgz` and compiled according to the instructions in the `README` file. Another file in the archive, also available separately at `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/test/tutorial_core.dul`, contains the examples of this section. The file can be processed by issuing "`./dule -c ../test/tutorial_core.dul`" or "`make test-core`". The compilation is performed with the help of the standard prelude, described in Appendix C.1.

The Dule core language makes it possible to express values in a style reminiscent of modern general-purpose functional programming languages. We present two conventional examples, in which there are only minor notational novelties, already known from OCaml and described in subsequent sections, such as tilde (as in `~n`) that marks labels of function parameters (however, in function bodies the parameters are referred to without `~`, as in `n`). The phrase `Nat.leq ~n:n ~it:1`, abbreviated to `Nat.leq ~n ~it:1`, is an application of a function `leq` from the module `Nat` to the arguments `n` and `1` at positions labeled `n` and `it`, respectively. Label `it` appears in the typing of many built-in operations, but the label has no special status and can be freely used by the programmer — in fact, there are no reserved labels nor variant names in Dule.

The Dule programmer can use general recursion, just as in most functional languages. Note that module-level identifiers are written uppercase, while core-level identifiers are written lowercase.

```
let rec fib = fun ~n ->
  if Nat.leq ~n ~it:1 then 1
  else Nat.add ~n:(fib ~n:(Nat.sub ~n ~it:1))
              ~it:(fib ~n:(Nat.sub ~n ~it:2))
in
fib ~n:4
```

The following example illustrates the use of the usual case analysis, where variant names are marked, just like in OCaml, by back-quote (and are usually uppercase, as in `` `True ``). The case analysis, denoted in Dule by square braces, is triggered by keywords `match` and `with`, used here to deconstruct the value of the parameter `it`.

```
let disj = fun ~b ~it ->
  match it with [`True -> `True
                |`False -> b]
in
disj ~b:`True ~it:`False
```

However, the Dule language also enables and encourages the use of other styles of notation and definition. These new possibilities are discussed in sections to follow. For example, the disjunction operation may be written down in the following way:

```
let disj =
  [‘True fun ~b ~it -> ‘True
  |‘False fun ~b ~it -> b]
in
disj ~b:‘True ~it:‘False
```

and the `fib` function may be defined using structured recursion (`fold`, induction) rather than general recursion:

```
let fib = fun ~n ->
  let {this; next} = (* n and n+1 Fibonacci numbers *)
    match n with
    fold [‘Zero -> {this = 1; next = 1}
         |‘Succ {this; next} ->
             {this = next;
              next = Nat.add ~n:this ~it:next}]
  in this
in
fib ~n:7
```

The example above contains a comment, which is written like in the ML family of languages, that is, enclosed in parenthesis with stars. Note that the semicolons separate record fields and we will never use them in our thesis to denote any kind of composition or sequential execution. Neither the labels nor the variant names have scopes or types assigned within a scope. All the types — function types, sum types, record types, (co)inductive types — are anonymous, that is, they have no identity other than given by semantics of their construction. The labels and variant names have no semantics outside types and their related operations, and they can appear in many types at once. Type reconstruction resolves any emerging ambiguities.

2.1.1 Identity and types

Types

The Dule core language is strongly and statically typed. The types of correct values are automatically reconstructed and no type annotations, either in the core phrases or supplied in module interfaces, are essential for the reconstruction. Moreover, all the reconstructed types are expressible in the core language, as stated in Section 8.3.1. This strong property follows from the anonymous nature of product, sum, inductive and coinductive types and from the lack of polymorphism at the core language level. In Dule, values depending on types are expressed using either polytypism [91] or module parameterization.

The polytypism will be illustrated later on with the examples of mappings and (co)inductive constructions. Here we concentrate on typing monomorphic values. For example, the expression

```
Bool.conj ~b:`True ~it:`False
```

where module `Bool` is a part of the standard prelude, will be assigned type `` [`True|`False] ``, which is the type of boolean values. Another expression

```
fun ~b -> if b then 7 else 5
```

will be typed as a function with a type

```
    ~b:[`True|`False] -> ind t: [`Zero|`Succ t]
```

where the type expression

```
ind t: [`Zero|`Succ t]
```

represents inductively defined natural numbers (decimal numbers are an abbreviation for the unary inductive representation of natural numbers, described under **Inductive constructions** below). A given type can be written in different ways. For example, the type of boolean values can also be written as `` [`False|`True] `` and the type of natural numbers as `` ind nat: [`Succ nat|`Zero {}] ``.

More details about particular families of types will be given when the value constructions associated to them are discussed. Here we only describe type projection and the composition of types. A type projection denotes a type obtained from a context. By context we mean either the implicit environment of an expression or an explicitly given module. In the later case the access to the module type components is expressed by a type composition, which is written using a dot.

For example, the type expression

```
Nat . t
```

consists of two composed type projections. The second `nat` in

```
coind stream: {head : ind nat: [`Zero|`Succ nat];
               tail : stream}
```

is a type projection, too. In this restricted language of types that we make available to the programmer, projection and type composition don't serve any other purpose than extracting types from (nested) contexts.

Identity

Type annotations can be placed inside function formal parameters and, more generally, in any place where an element of the pattern syntactic domain may appear, as described in the grammar of Dule. The only other way of asserting the type of a value in the core language is an indirect way of composing with another value. The identity operation makes this method viable — in fact universally applicable. Consider the following example, where dot denotes value composition and type preceded by a colon denotes identity value on that type (and underscore is a dummy identifier).

```
let _ = Bool.conj ~b:‘True ~it:(‘False . : [‘True|‘False]) in
let seven = 7 . : ind t: [‘Zero|‘Succ t] in
fun ~b ->
  let _ = b . : [‘True|‘False] in
  if b then seven else 0
```

A stand-alone identity could be used to assert the type of values contained in a current environment, but the environment is usually too large to make this feasible. A more practical use of an identity can be in records with many identical fields, as in:

```
(if Nat.is_zero ~it:1 then ‘Z else ‘N)
  . {z1 = : [‘N|‘Z]; z2 = : [‘N|‘Z]; z3 = ‘N;
     zr = {z1 = ‘Z; z2 = : [‘N|‘Z]}}
```

where the values at fields `z1`, `z2` and the nested field `zr.z2` are taken from the fist operand of composition. However a `let`-expression with suggestive value names can be used here to good effect as well, resulting in the same outcome value:

```
let the_same_z = if Nat.is_zero ~it:1 then ‘Z else ‘N in
    {z1 = the_same_z; z2 = the_same_z; z3 = ‘N;
     zr = {z1 = ‘Z; z2 = the_same_z}}
```

2.1.2 Basic value operations

Value composition

The composition is denoted by an infix dot (written with spaces or without). Identity is a neutral element of composition. Nevertheless, asserting types using identity may change the semantics of an expression. For example, although

```
‘True . : [‘True]
```

has the same semantics as

```
‘True
```

and the same semantics as

```
‘True . (: [‘True]).(: [‘True]) . : [‘True]
```

one can easily verify that the expression

```
‘True . : [‘True|‘False]
```

has a different semantics than the three above. Yet, if an expression occurs in a context that fully determines its type, then composing with identity does not change the expression’s semantics.

Extracting a component from a record is always done by composing with a projection corresponding to the field label. For example

```
{ready = ‘True; speed = 3; color = ‘Red} . speed
```

and similarly for accessing an operation from a module

```
Nat.add ~n:(Nat . pred ~n:1) ~it:5
```

This latter use stems from the fact that the value part of a module is visible from the core language as a record of values. This allows one to “open” modules using composition:

```
let plus_one = Nat . fun ~n -> add ~n:(succ ~n:zero) ~it:n in
plus_one ~n:4
```

But beware this not only opens a module but also renders all the rest of the environment invisible. This restricts the area of the idiom’s use, but also prevents most accidental name clashes.

Injections and constructors are associated with their values using composition, too. In the following example the injection `‘True` is associated with the trivial value, while `‘OK` injects the whole record into a sum type. The injection `‘Nil` is implicitly composed with `{}` and the result is made into a value of an inductive type (presumably the type of lists) by composing with the constructor.

```
{strong = {} .‘True; args = ‘Nil . con} .‘OK
```

Composition may as well be used to destruct values (using the destructors `de` or `unde` that will be described later on):

```
let ones = (unfold tail -> {head = 1; tail}) ~it:{} in
ones . unde . tail . unde . head
```

The result of the above expression is the number `1` that is the second element of the infinite stream created with `unfold`. The various destruction mechanisms used above will be explained in more detail later on.

Value projection

Product types may serve to type user defined records of values. They are written between braces, just as records are:

```
{b = ‘True; t = {}} . : {b : [‘True|‘False]; t : {}}
```

Product types are also used to model dependency on the environments, either implicitly created by modules, function abstractions, etc., or explicitly given by the programmer. Both cases can be seen in the following example where the first three of the four identities may be omitted, but no other phrase can be, if we are to obtain the same result.

```
{} . (: {})
  . {t = ‘C} . (: {t : [‘C]})
    . fun ~(x:[‘A]) ->
        (: {t : [‘C]; x : [‘A]})
          . (let w = t in
            : {w : [‘C]; t : [‘C]; x : [‘A]})
```

The composition of the third identity with the `let`-expression is correct, because the environment of a whole `let`-expression is smaller than the environment of the phrase after `in`.

Projection is a value with a product type domain. In the following expression

```
Nat . is_zero
```

the `Nat` projection has domain equal to the type of the entire environment. The `Nat` codomain is the product of values' types of module `Nat`, and the same product is also the domain of the `is_zero` projection. The codomain of `is_zero` is

```
~it:(ind t: [‘Zero|‘Succ t]) -> [‘True|‘False]
```

Record

Records are constructors of the product type values. In particular a trivial value — a trivial record `{}` — is of a trivial product type `{}`.

If a field of a record contains a projection with the same label as the label of the field, one can employ an abbreviation, writing only the field label. This abbreviation can be seen in the third and fourth component of the following record. Each of the four components has the same semantics as the first one.

```
{r1 = {a = 1; b = 2; c = 3};
 r2 = {a = 1; b = 2; c = 3} . {a = a; b = b; c = c};
 r3 = {a = 1; b = 2; c = 3} . {a = a; b; c};
 r4 = {a = 1; b = 2; c = 3} . {a; b; c}}
```

Another example shows nested records:

```
fun ~a ~b ~c ~d ->
  {a; b; c; g = 1} . {a; h = {a; g}; z = `OK}
```

As everywhere in Dule, only the assignment of labels to record components matters, while their relative order is unimportant. The labels have to be unique within the same level of a record but needn't be unique in any other respect. The type reconstruction ensures that the semantics of the record below is unambiguous; the outcome of the function will be the value of its argument at label `~a`.

```
fun ~a ~b ->
  let a = {a = {b = {a}}; b = a} in
  a . a . b . a
```

Injection

The type of boolean values

```
[`True|`False]
```

is an example of a sum type. In a verbose notation, this type expression looks more complex

```
[`True {}|`False {}]
```

The additional phrases denote the types of variants (the argument types of the variant constructors). If they are trivial, as above, they need not be written.

Here is an example of a natural number injected into a sum type

```
1 .`OK
```

and the same expression, but with a type explicitly assigned

```
1 .`OK . : [`OK ind t: [`Zero|`Succ t]|`Error]
```

and another expression, with a different type, and so with a different semantics

```
1 .`OK . : [`OK ind t: [`Zero|`Succ t]]
```

The injections are the constructors of the sum type. They are associated with their values using a composition, as above. If the values are of a trivial type the composition may be omitted. So for example

```
`Error . : [`OK ind t: [`Zero|`Succ t]|`Error]
```

is equivalent to

```
{} .‘Error . : [‘OK ind t: [‘Zero|‘Succ t]|‘Error]
```

Note that the composition is associative and so the typing may be seen as pertaining to the whole expression, as well as to the injection alone.

The injections could also occur without an immediate composition, but due to the abbreviations they would be misunderstood for injections of trivial values. To overcome this, composition with an identity is required, as in the following example

```
1 . {ok = (: ind t: [‘Zero|‘Succ t]) .‘OK}
```

which is equivalent to

```
{ok = 1 .‘OK}
```

Case analysis

Values of sum type are transformed using case analysis. For example, the `case_exp` below is a function that when applied to a value of sum type gives a natural number:

```
let case_exp =
  [‘OK n -> n
  |‘Error -> 0]
in
case_exp ~it:‘Error
```

The phrase to the right of the variant label is called an embedding. Normally an embedding is just a function with a parameter labeled `it`. In the example above, the basic form of embedding is not present. Instead, two other possible forms of embeddings appear. Their meaning may be clarified by rewriting the embeddings into the basic form:

```
let case_exp =
  [‘OK fun ~it:n -> n
  |‘Error fun ~it:_ -> 0]
in
case_exp ~it:‘Error
```

A recent extension to Dule case analysis is the default variant, defining the outcome of case analysis for all unlisted variants. The following example

```
[‘A -> ‘True |‘B -> ‘True |‘C -> ‘True
|_ -> ‘False ]
```

is an abbreviation for, e.g., the following ordinary case analysis:

```
[‘A -> ‘True |‘B -> ‘True |‘C -> ‘True
|‘a -> ‘False|‘b -> ‘False|‘c -> ‘False]
```

Note that the underscore that denotes the default variant has a totally different semantics than the underscore in

```
fun ~it:_ -> 0
```

or in

```
[‘Nil -> 0
|‘Cons _ -> 1]
```

which is just a variable inaccessible to the programmer.

Another recent extension of Dule, related to pattern matching, is record pattern matching (marked by curly braces below)

```
[‘Nil -> 0
|‘Cons {head; tail} -> head]
```

which may appear in any form of embedding, including standard functions

```
fun ~it:{head; tail} -> head
```

and which is just an abbreviation for value projections

```
fun ~it -> it.head
```

The semantics of the two recent extensions is not formally described in our thesis. In the future, it will be analyzed separately, as a part of a more general and comprehensive pattern-matching mechanism for Dule. However these two extensions make example programs distinctively more readable, so we will use them in the tutorial, as well as in the Dule program examples in Appendix C.

The conditional expression is a simple derived form of case analysis, so that for instance the following code

```
if b then 1 else 0
```

has the same semantics as

```
[‘True -> 1|‘False -> 0] ~it:b
```

A precise account of this abbreviation is given in Section 8.3.2.

Mapping over a value

The keyword `map` signals the operation of traversing a complex value, applying the given function to all sub-values on the way. This is a very broad generalization of the map combinator on lists, well known in functional programming. The list mapping combinator operating with a function `f` may be recast as simply `map f`. In the following example we obtain a single element list containing the uppercase letter `A` (`Char` is the library module implementing operations on character literals represented as their ASCII numbers, `Char.tcon ~it:‘a` is the internal representation of character `a`, `Char.upper` converts a letter to uppercase, `CharList` is the library module of character lists):

```
let list_map = fun ~f ~l ->
  (map f) ~it:l
in
list_map
  ~f:Char.upper
  ~l:(CharList.cons ~head:(Char.tcon ~it:‘a) ~tail:CharList.nil)
```

A map combinator on binary trees operating with a function `f` is written in the same way: `map f`. In the following example we construct a tree and then apply the mapping combinator, obtaining a tree with a single value `‘True`:

```
let tree_map = fun ~f ~t ->
  (map f) ~it:t
in
let t =
  {valu = 0;
   left = ‘Empty . con;
   right = ‘Empty . con} .‘Node . con
in
tree_map ~f:Nat.is_zero ~t
```

The phrase after the keyword `map` is an embedding, the same as in case analysis, with all the forms of syntactic sugar permitted. Type reconstruction copes with deriving a type for any correct mapping, despite the many possible ambiguities.

One can map not only over values of inductive types, but over a value of every type (except a function type, see Section 8.1). Below we define a function negating every value on a coinductive stream. Then the function is tested and the resulting stream is destructed to obtain its first element, which turns out to be `‘True`.

```
let negate_stream =
      map [‘True -> ‘False
```

```
          |‘False -> ‘True]
    falsities =
      (unfold tail -> {head = ‘False; tail}) ~it:{}
in
let truths = negate_stream ~it:falsities in
truths . unde . head
```

The expression below checks and notes whether a number is zero, for every natural number value contained in the record:

```
match {b = ‘False; n = 0;
       r = {rn = 5; ro = ‘OK}} with
map Nat.is_zero
```

The result will be a similar record but with `‘True` in place of `0` and `‘False` in place of `5`. The type-checker will reconstruct such a type for the `map` combinator so that the application of the combinator will retain the old values at the `b` and `ro` fields of the record. In the following example we increment three natural numbers (the parenthesis in the applications of `incr` are superfluous):

```
let incr =
  (map n -> Nat.succ ~n)
in
{n1n2 = incr ~it:({n1 = 5; n2 = 7, b = ‘True} .‘Two);
 n = incr ~it:13 .‘One}
```

In the resulting record there will be numbers 6, 8 and 14, respectively.

The astute reader may ask “what if I want only one of the numbers `n1`, `n2` above to be incremented or, in another application, only attributes of an AVL tree, and not its elements?”. This could be answered by providing an additional syntax for directing the type reconstruction for mappings more precisely. Instead we suggest that the desired functions be written by hand, using case analysis, folding, etc. A formal account of how typing determines sub-values to be changed is provided in Section 8.3.1.

Inductive constructions

In Dule core language, inductive types are anonymous, just as any other types. For example

```
ind t: [‘Zero|‘Succ t]
```

is a type of inductive natural numbers. Number 0 is defined as

```
‘Zero . con
```

where `con` is a constructor of an inductive type. The precise semantics of the constructor operation is given in Section 6.3.1. Note that the expression

```
‘Zero
```

is not of an inductive type of natural numbers: it is of a sum type. Number 1 can be written and typed using composition

```
‘Zero . con .‘Succ . con . : ind t: [‘Zero|‘Succ t]
```

and number 2 can be written similarly

```
‘Zero . con .‘Succ . con .‘Succ . con
```

The expression

```
‘Zero . con .‘Succ .‘Succ
```

does not have the type of natural numbers, but a sum type. One of its shorter possible types is

```
[‘Succ [‘Succ ind t: [‘Zero|‘Succ t]]]
```

The type of binary trees of boolean values, which we have already used in an example above, can be defined as follows:

```
ind tree: [‘Empty
          |‘Node {valu : [‘True|‘False];
                  left : tree;
                  right : tree}]
```

The values of inductive types can be traversed using mappings or general recursion with the help of the destructor, or structured recursion with the `fold` keyword. The destructor, written `de`, is an inverse of the constructor `con` with respect to the composition:

```
let pred = fun ~n ->
  match n . de with
  [‘Succ n_1 -> n_1
  |‘Zero -> n]
in
pred ~n:‘Zero . con
```

Using the destructor and general recursion we can define the addition function on natural numbers (notice the keyword `rec`):

```
let rec add = fun ~n ~it ->
  match n . de with
  [‘Zero -> it
  |‘Succ n_1 -> add ~n:n_1 ~it .‘Succ . con]
in
add ~n:3 ~it:7
```

But the same function can be defined using structured recursion (induction) as follows (notice that the destructor is not used here):

```
let add = fun ~n ~it ->
  match n with
  fold [‘Zero -> it
       |‘Succ nn -> nn .‘Succ . con]
in
add ~n:3 ~it:7
```

The phrase after the keyword `fold` is a case analysis expression and so an embedding, that is, a function with a parameter named `it`. Other forms of embeddings can be used as well. For example, when the fold expression in the following example is applied to a binary tree, it traverses the tree.

```
fun ~tree ->
  (fold -> 3) ~it:tree
```

All the intermediate results during the traversal as well as the final outcome are the number 3. The embedding in this example is "`-> 3`", which denotes the constant function "`fun ~it -> 3`".

The destructor can be itself defined in Dule core language in terms of induction. For example, the following function destructs a list `l` producing the same value of a sum type that results from the composition `l . de`:

```
fun ~l ->
  match l with
  fold [‘Nil -> ‘Nil
       |‘Cons ht -> ht . {head; tail = tail . con} .‘Cons]
```

(where the composition `ht . {head; tail = tail . con}` is equivalent to record `{head = ht . head; tail = ht . tail . con}`). Similar definitions are possible for other inductive types. However, the built-in destructor combinator is polytypic (one for all datatypes), usually more readable and resulting in faster code.

The last example in this section is more complicated. It shows an induction with case expression and embeddings that have a function result type. The embedding after `fold` is an anonymous case expression. The embedding after `‘Cons` is named `non_nil_eq` and is a function with a single parameter labeled `it`. In the example we define an equality on strings (lists of characters). The presented solution performs one traversal on each of the two parameters (named `it` and `l`) of the main function. The first traversal is over the `it` list and builds a function. The function is then applied to the `l` list and traverses it checking equality of individual characters. The auxiliary function `CharListTool.is_nil` with a single parameter of string type (labeled `l`) detects if the string is empty. `Char.eq` is the equality predicate on character literals.

```
let non_nil_eq = fun ~it -> fun ~l ->
  match l . de with
  [‘Nil -> Bool.ff
  |‘Cons lht ->
      let head_matches = Char.eq ~c:it.head ~it:lht.head
          tail_equal = it.tail ~l:lht.tail in
      Bool.conj ~b:head_matches ~it:tail_equal]
in
fun ~it ~l ->
  ((fold [‘Nil -> CharListOps.is_nil
         |‘Cons non_nil_eq]) ~it) ~l
```

Coinductive constructions

Coinductive types are defined similarly as inductive types. The difference is the pattern of the structured recursion. The coinductive counterpart of inductive natural numbers is

```
coind t: [‘coZero|‘coSucc t]
```

The constructors and destructors behave just as their inductive counterparts. A coinductive zero is

```
‘coZero . uncon
```

A coinductive number one would be:

```
‘coZero . uncon .‘coSucc . uncon
```

There is also the “infinity” number [22], defined using co-induction (`unfold`) [114]:

```
match {} with
unfold cn -> cn .‘coSucc
```

Addition of coinductive numbers can be defined without problems using `unfold` and the coinductive destructor `unde`, which is the inverse of `uncon`.

```
fun ~n ~it ->
  match {n; it} with
  unfold {n; it} ->
    match n . unde with
    [`coZero ->
       match it . unde with
       [`coZero -> `coZero
       |`coSucc it -> {n; it} .`coSucc]
    |`coSucc n -> {n; it} .`coSucc]
```

This definition copes well with the possibility of the infinity number as one of its arguments. The computation of an `unfold` expression is triggered only by composing it with the destructor and so is, in a way, lazy. The coinductive structured recursion is guaranteed to converge, just as the inductive one.

On the other hand, the solution based on the destructor and general recursion, as given below, diverges when applied to infinity at label `n`.

```
let rec add = fun ~n ~it ->
  match n . unde with
  [`coZero -> it
  |`coSucc n_1 -> add ~n:n_1 ~it .`coSucc . uncon]
in
add
```

In fact, when using only constructors and destructors there is no observable difference between the inductive and coinductive constructions. What makes the difference is the structured recursion, that is induction (`fold`) and co-induction (`unfold`). By mixing inductive and coinductive structured recursion one can express a wide range of functions. Moreover, this can be done in an efficient way and always assuring termination. Many interesting examples are given in tutorials related to the Charity programming language ([22], [23]). This surprising expressiveness is possible even without a function type, although higher-order functions add a lot of flexibility to the language.

2.1.3 Functions and derived forms

Functions

Function types in Dule have labeled parameters. The addition function on coinductive numbers, defined above, has a type:

```
~n:coind t: [`coZero|`coSucc t]
  ~it:coind t: [`coZero|`coSucc t]
    -> coind t: [`coZero|`coSucc t]
```

Inside the following expression there is an example of a longer function type that has some more informative labels:

```
fun ~sublist ->
  let third = fun ~l ->
    let _ =
      sublist
        . : ~l:ind t: [‘Nil|‘Cons {head : [‘T|‘F]; tail : t}]
            ~pos:ind t: [‘Zero|‘Succ t]
            ~len:ind t: [‘Zero|‘Succ t]
              -> ind t: [‘Nil|‘Cons {head : [‘T|‘F]; tail : t}]
    in
    sublist ~l ~pos:3 ~len:1
  in
    third ~l:‘Nil . con
```

Although the order of labels is unimportant, there is no α-conversion of function parameter labels, just as there is no α-conversion of record labels. Therefore, the following piece of code is wrong

```
fun ~add ->
  add ~n:5 ~it:(add ~n’:5 ~itself:5)
```

because the function parameter `add` should be assigned the same function type in every place it is used.

On the other hand, labels in a function definition can be followed by a pattern, and any identifiers occurring in the pattern admit α-conversion. They are the true bound identifiers of the function definition. For example, the following function

```
fun ~add ~n1:_ ~n2:important_n ~n3:_ ->
  add ~n:important_n ~it:important_n
```

is equivalent to

```
fun ~add ~n1:_ ~n2:nnn ~n3:_ ->
  add ~n:nnn ~it:nnn
```

If there is a pattern, then the label itself is not visible in the body of the function. The case of no pattern can be thought of as a pattern consisting just of the label’s name. The underscores are (internally distinct from each other) identifiers that can be written only inside a pattern.

Labels may not be repeated within a function header. The list of labeled parameters may be of arbitrary length, even of zero length, for example:

```
let thunk = fun -> 1 in
let _ = thunk . : (-> ind t: [‘Zero|‘Succ t]) in
thunk ~
```

where the function followed by a sole tilde is an application of the function to the empty list of arguments. When a function is applied to an argument that is a projection at label identical to the argument's label, the projection may be omitted:

```
fun ~n ~add ->
  let it = 3 in
  add ~n ~it:(add ~it ~n:2)
```

The lack of application by juxtaposition, that is by grouping expressions one after another, greatly reduces ambiguity in the grammar. In our opinion it helps considerably in catching typographic errors, especially omissions of keywords and separators. For example omitting keyword `in` in the above expression results in parsing error, while with juxtaposition the attempt to apply `3` to `add` would be grammatically correct. Moreover, if instead of `3` there was an identifier, the type-error message could be hard to understand, or even the expression could type-check at this stage.

Partial application

A nested function type is never equivalent to a flat function type with all its labels. In particular fully applying a nested function requires a lot of parenthesis:

```
let f = fun ~x -> fun ~y1 ~y2 -> fun ~z -> x in
((f ~x:{}) ~y2:2 ~y1:1) ~z:3
```

Partially applying a flat function type involves parenthesis, too. This time they are a part of a special syntactic abbreviation. The partial application in the following example

```
let f = fun ~a ~b1 ~b2 ~d -> a in
f(~d:3 ~b1:1)
```

is a shorthand for the abstraction and normal application below

```
let f = fun ~a ~b1 ~b2 ~d -> a in
fun ~a ~b2 -> f ~a ~b2 ~b1:1 ~d:3
```

The separate syntax for partial application is convenient, because partial applications are relatively rare. The syntactic distinction between partial and full application helps humans in reading the code and helps the parser in catching omitted arguments.

The form of the abbreviation shows that partial application to zero arguments would be equal to the function itself, so such application is not syntacticly supported. On the other hand multiple partial applications and mixed partial and full applications are possible:

```
let f = fun ~a ~b1 ~b2 ~d -> a in
let fb1 = f(~b1:1) in
let r = fb1 (~a:{}) (~d:3) ~b2:2 in
r . : {}
```

Note that a partial application with the full list of arguments is not equivalent to a full application:

```
let f = fun ~a ~b1 ~b2 ~d -> a in
let r = f(~a:{} ~d:3 ~b2:2 ~b1:1) in
r . : (-> {})
```

A full application to an empty list of arguments is needed to obtain the final result:

```
let f = fun ~a ~b1 ~b2 ~d -> a in
let r = f(~a:{} ~d:3 ~b2:2 ~b1:1) in
r ~ . : {}
```

Other shorthands

The most frequently occurring kind of application is that with a single label `it`. This application appears, in particular, when supplying arguments to case analysis or structured recursion. For these cases there is an additional syntactic form for application, written using keywords `match` and `with`, which denotes an application of a single argument at label `it`. For example

```
(unfold -> ‘True) ~it:5
```

is equivalent to

```
match 5 with (unfold -> ‘True)
```

The syntax is especially handy for delimiting large arguments:

```
match (Nat.pred ~n:(Nat.add ~n:1 ~it:2)) . de with
[‘Zero -> ‘True
|‘Succ -> ‘False]
```

Another common derived form is the `let`-expression. The expression

```
let one = 1
    two = 2
in
Nat.add ~n:one ~it:two
```

is equivalent both in typing and in semantics to the expression

```
(fun ~one ~two ->
  Nat.add ~n:one ~it:two)
~one:1 ~two:2
```

Arbitrary patterns may be used in place of simple names of locally defined values:

```
let b : [‘True|‘False] = ‘True
    _ = fun ~unimportant -> unimportant
    {t; f} = {} . {t = : {}; f = {}}
in
{b; t}
```

The Dule language is based on flexible structured recursion mechanisms in the core language and powerful fixpoint capabilities trough mutually dependent modules. However, the simplest cases of general recursion are expressible not only through module language, but in the core language as well. For this purpose a fixpoint operation is introduced as an extension to the `let`-notation:

```
let rec exhaust = fun ~cn ->
  match cn . unde with
  [‘coZero -> Nat.zero
  |‘coSucc cn -> Nat.succ ~n:(exhaust ~cn)]
in
exhaust ~cn:‘coZero . uncon .‘coSucc . uncon
```

Recursion can be used to define values of arbitrary type, in particular of a record type. For example, the following expression leads to non-terminating computation

```
let rec r = {a = r.a} in r.a
```

while the expression below results in the value `‘c`

```
let rec rab = {a = rab.b; b = ‘b} in
let rec rf = {f = fun ~x -> rf.g ~x;
              g = fun ~x -> ‘c} in
rf.f ~x:rab.a
```

In this indirect way one can express mutually recursive definitions in the core language, but there is no separate syntax for such an idiom. As a rule, mutually recursive definitions are complex enough to warrant embedding into separate mutually recursive modules.

Assertion and failure

Assertions (`assert` *property* `in` *code*) serve to express and possibly test some important assumed properties of programs. The official semantics of an assertion is that of the expression after the keyword `in`. But depending on the command line options, the Dule compiler may insert some code that checks the condition after `assert`. As soon as the condition is decided to be not satisfied, the code aborts computation.

Here is an example of the use of assertions:

```
fun ~n ->
  assert Nat.leq ~n ~it:9 in
  let m = Nat.succ ~n in
  assert Bool.neg ~it:(Nat.is_zero ~it:m) in
  Nat.add ~n ~it:m
```

A longer example defines a constructor for lists of unique elements:

```
let cons__avoid_duplicates =
  let is_in = fun ~n ~l ->
    let in_tail_or_check_head = fun ~it:{head; tail} ->
      match tail with
      [‘True -> ‘True
      |‘False -> Nat.eq ~n ~it:head]
    in
      match l with
      fold [‘Nil -> ‘False
           |‘Cons in_tail_or_check_head]
  in
    fun ~n ~l ->
      assert Bool.neg ~it:(is_in ~n ~l) in
      {head = n; tail = l} .‘Cons . con
in
cons__avoid_duplicates
```

The `fail` expression is used to mark cases that cannot occur:

```
fun ~b ->
  match Bool.disj ~b ~it:‘True with
  [‘True -> 1
  |‘False fail]
```

The semantics of the expression `fail` is declared to be the same as of:

```
assert ‘False in let rec it = it in it
```

However, to ensure that the effect of a failure expression is independent of evaluation strategy and compilation options, it is preferable if the compiler directly generates the aborting code. Whenever such a code is involved in a computation, the program immediately stops running.

2.2 Simple modular programming

Remark. All the examples of modular Dule code given in this version of the tutorial are gathered in file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/test/tutorial_modules.dul` and can be compiled using the `dule` compiler without additional options. The examples take advantage of the standard prelude, described in Appendix C.1.

2.2.1 Base modules

The core language is used to define components of Dule base modules. A base module, also called a structure, groups related entities — values, types, comments, etc. — between keywords `struct...end`. The values are declared using the keyword `value` and a core language expression:

```
struct
  value tt = ‘True
  value ff = ‘False
  value neg = [‘True -> ‘False
              |‘False -> ‘True]
  value conj = [‘True fun ~b ~it -> b
               |‘False fun ~b ~it -> ‘False]
  value disj = [‘True fun ~b ~it -> ‘True
               |‘False fun ~b ~it -> b]
end
```

Base specifications of modules, called also signatures, describe base modules. For each value in a structure there is a corresponding core language type expression in the signature. There may also be a comment describing the value in more detail either formally or using some formalism chosen by the programmer. A specification describing the module above may look as follows:

```
sig
  value tt : [‘True|‘False]
  value ff : [‘True|‘False]
  value neg : ~it:[‘True|‘False] -> [‘True|‘False]
  value conj :
    ~b:[‘True|‘False] ~it:[‘True|‘False] -> [‘True|‘False]
  value disj :
    ~b:[‘True|‘False] ~it:[‘True|‘False] -> [‘True|‘False]
end
```

A structure may contain types. Let the following example module be called `Nat`:

```
struct
  type t = ind nat: [‘Zero|‘Succ nat]
  value t2ind = fun ~it -> it (* see the typing below *)
  value tde = fun ~it -> it . de
  value zero = ‘Zero . con
  value succ = fun ~n -> n .‘Succ . con
end
```

The values are not allowed to refer to one another. Similarly, type definitions mustn't mention other type names of the same structure. If there is a significant logical dependency among the values or among the types, then the module should be split and the dependency expressed using the module language. Although the types and values in a module are logically interconnected, in a value definition one cannot use a name of a type as an alias for the type expression itself. There is no deep reason for this last restriction, but it simplifies the technicalities of type reconstruction.

The relation between the values and types of a module is captured only at the level of a signature, where types of values of the module can refer to names of types defined in the module. Let the following specification of module `Nat` be called `Nat` as well (all the various name spaces in Dule — types, values, modules, specifications, libraries — are disjoint):

```
sig
  type t
  value t2ind : ~it:t -> ind nat: [‘Zero|‘Succ nat]
  value tde : ~it:t -> [‘Zero|‘Succ t]
  value zero : t
  value succ : ~n:t -> t
end
```

Note that at this level the names of types are not interchangeable with the type definitions from the module — they are semantically different and they play utterly different roles. For instance, the result of function `t2ind` could be specified as `t`, but then the function would cease to provide access to the inductive nature of type `t` to users of the module.

A specification of a structure that refers to other modules is no longer a base specification, but a parameterized specification, describing both the (nontrivial in this case) domain and the codomain of a module. The following example is parameterized by a module `Nat`.

```
~Nat:Nat ->
  sig
    value pred : ~n:Nat.t -> Nat.t
    value add : Nat . ~n:t ~it:t -> t
    value is_zero : ~it:Nat.t -> [‘True|‘False]
  end
```

In the type of the value `add` we have moved the type composition to the front to avoid writing the name of the module three times. By doing this we have also emphasized that the type depends solely on the types contained in the module `Nat` (by the properties of type composition). However, in this particular case there are no other types that could possibly be referred to, because Dule modules as well as their specifications are closed.

If we know that the following structure has the above specification we can refer in it to both types and values of the module `Nat`:

```
struct
  value pred = fun ~(n : Nat.t) ->
    match Nat.tde ~it:n with
    [‘Zero -> n
    |‘Succ nn -> nn]
  value add = fun ~n ~it ->
    match Nat.t2ind ~it with
    fold [‘Zero -> n
         |‘Succ n -> Nat.succ ~n]
  value is_zero = fun ~it ->
    match Nat.tde ~it with
    [‘Zero -> ‘True
    |‘Succ -> ‘False]
end
```

But again, the name of a type or of a value is not an alias for its implementation in the module. We know about module `Nat` only as much as its specification tells us: that certain abstract types exist and that values have certain types. So for example we couldn't replace "`Nat.succ ~n`" in the above example by "`n .‘Succ . con`", even though the implementation of the value `succ` is indeed of this form. As a result, the semantics of a module with a parameterized specification can always be computed independently of the semantics of its argument modules.

Note how we use functions `t2ind` and `tde` to recover the inductive nature of type `Nat.t`, which has been forgotten by depending on an abstract parameter, rather than the concrete argument module. (We discuss the `t2ind` technique in detail and illustrate it by examples in [96].) Let us imagine that the actual implementation of the type of natural numbers was, for example, decimal instead of the current unary. Under such circumstances the module above would still work as expected. Moreover, if the compiler supports some degree of laziness and the `t2ind` and `tde` functions are written efficiently, then the above code of `pred` and `is_zero` would be as fast as the code written with access to primitive operations of positional implementation.

One last bit of complexity is encountered when a base module that depends on another module, defines its own types. Now both the local types and the types from parameters can appear in the specification. This is illustrated by the following specification of lists. In this specification as well as in most other places, we name all the types just `t` and differentiate them by the modules they come from. This coincides well with our emphasis on small modules and our lack of any conventional module opening construct.

```
~Elem:sig type t end ->
  sig
    type t
    value t2ind :
      ~it:t -> ind list: [‘Nil|‘Cons {head : Elem.t;
                                      tail : list}]
    value ind2t :
      ~it:ind list: [‘Nil|‘Cons {head : Elem.t;
                                 tail : list}] -> t
    value tde :
      ~it:t -> [‘Nil|‘Cons {head : Elem.t; tail : t}]
    value nil : t
    value cons : ~head:Elem.t ~tail:t -> t
  end
```

The type definition in the module may refer to the type from the parameter module. This is the case in the definition of the module of lists:

```
struct
  type t = ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
  value t2ind = fun ~it -> it
  value ind2t = fun ~it -> it
  value tde = fun ~it -> it . de
  value nil = ‘Nil . con
  value cons = fun ~head ~tail -> {head; tail} .‘Cons . con
end
```

The module of lists above is a generic parameterized module, which is likely to be instantiated often and with arguments that are much richer than the signature of parameter `Elem` requires. In our programming practice we find such modules to be the least common, but often the most useful. Managing such modules is quite troublesome in Dule, which is a language aimed at automating the most common module manipulation tasks, not the least common. Normally, in Dule, defining modules one after another is enough to ensure that each of them will be automatically composed with the arguments it needs. But this is not the case with a generic module, because its dependencies can be satisfied in many incompatible ways using the modules of the program at hand. There are no defaults and there should be no defaults when instantiating generic modules.

To fully utilize the potential of the module above, it should be defined in a special way (described in detail in the next section), that is as a library. Then each time the library is used, the programmer has to define which module should be taken as the `Elem` argument and how to fit the potentially rich module's specification to the minimal requirements of `Elem`.

Let's suppose the library is called `List` and we want to obtain a module of lists of natural numbers. The instantiated module would be denoted by the following expression:

```
load List with {Elem = Nat}
```

where five new module operations appear: loading (`load`), instantiation (`with`), record (curly braces), projection (`Nat`) and trimming (implicit, as part of instantiation). We will describe these and other module operations after, in the next section, we finish describing the most common and straightforward aspects of modular programming.

2.2.2 Linking

The modules may be put together in various ways. The most common way is by linking, that is gathering several module, specification and library definitions between keywords `link` and `end`. The result of the linking operation is a record of all the modules, with their dependencies satisfied by composing.

Let us recast the collection of example modules from the previous section as constituents of two linking expressions. First, the specification and module of boolean operations, where we write the `link...end` brackets explicitly:

```
link
  spec Bool =
    sig
      value tt : [`True|`False]
      value ff : [`True|`False]
      value neg : ~it:[`True|`False] -> [`True|`False]
      value conj :
        ~b:[‘True|‘False] ~it:[‘True|‘False] -> [‘True|‘False]
      value disj :
        ~b:[‘True|‘False] ~it:[‘True|‘False] -> [‘True|‘False]
    end

  module Bool =
    :: Bool
    struct
      value tt = ‘True
      value ff = ‘False
      value neg = [‘True -> ‘False
                  |‘False -> ‘True]
      value conj = [‘True fun ~b ~it -> b
                   |‘False fun ~b ~it -> ‘False]
      value disj = [‘True fun ~b ~it -> ‘True
                   |‘False fun ~b ~it -> b]
  end
end
```

There is only one module to link, so the process of satisfying inter-module dependencies is trivial. The keyword `module` is optional and is usually omitted. The construction denoted by a double colon, specification name and a module is an explicitly specified module. In the example above it asserts that the module `Bool` fulfills the specification `Bool`. In case when a module has the same name as a previously defined specification the double colon construction is implicitly inserted into the definition of a module. This convention usually relieves us of writing explicitly specified modules.

Every file of modular Dule code is considered to be embedded inside an implicit linking operation. In the examples to follow we will usually omit the `link...end` brackets. Here is a longer and more interesting linking expression:

```
spec Nat =
  sig
    type t
    value t2ind : ~it:t -> ind nat: [‘Zero|‘Succ nat]
    value tde : ~it:t -> [‘Zero|‘Succ t]
    value zero : t
    value succ : ~n:t -> t
  end
Nat =
  struct
    type t = ind nat: [‘Zero|‘Succ nat]
    value t2ind = fun ~it -> it
    value tde = fun ~it -> it . de
    value zero = `Zero . con
    value succ = fun ~n -> n .`Succ . con
  end
spec NatAdd =
~Nat:Nat ->
  sig
    value pred : ~n:Nat.t -> Nat.t
    value add : Nat . ~n:t ~it:t -> t
    value is_zero : ~it:Nat.t -> [`True|`False]
  end
NatAdd =
  struct
    value pred = fun ~(n : Nat.t) ->
      match Nat.tde ~it:n with
      [`Zero -> n
      |`Succ nn -> nn]
    value add = fun ~n ~it ->
      match Nat.t2ind ~it with
      fold [`Zero -> n
           |`Succ n -> Nat.succ ~n]
    value is_zero = fun ~it ->
      match Nat.tde ~it with
      [`Zero -> `True
      |`Succ -> `False]
  end
spec List =
~Elem:sig type t end ->
  sig
    type t
    value t2ind :
      ~it:t -> ind list: [`Nil|`Cons {head : Elem.t;
                                      tail : list}]
    value ind2t :
      ~it:ind list: [`Nil|`Cons {head : Elem.t;
                                 tail : list}] -> t
    value tde :
      ~it:t -> [`Nil|`Cons {head : Elem.t; tail : t}]
    value nil : t
    value cons : ~head:Elem.t ~tail:t -> t
  end
library List =
  struct
    type t = ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value t2ind = fun ~it -> it
    value ind2t = fun ~it -> it
    value tde = fun ~it -> it . de
    value nil = ‘Nil . con
    value cons = fun ~head ~tail -> {head; tail} .‘Cons . con
  end
spec NatList = List with {Elem = Nat}
NatList = load List with {Elem = Nat}
spec Test =
~NatList ->
  sig
    value five_zeros : NatList.t
  end
Test =
  :: ~NatAdd -> Test
  struct
    value five_zeros =
      let copy = fun ~elem ~count ->
        match Nat.t2ind ~it:count with
        fold [‘Zero -> NatList.nil
             |‘Succ tail -> NatList.cons ~head:elem ~tail]
      in
      let five_ones = copy ~elem:1 ~count:5 in
      NatList.ind2t ~it:
        match NatList.t2ind ~it:five_ones with
        map n -> NatAdd.pred ~n
  end
```

The specification and module `Nat` are self-contained, similarly as in the example describing boolean values. On the other hand, specifications `NatAdd`, `List` and `Test` are all parameterized. In `List` the parameter is called `Elem` and its specification is given after a colon. In `NatAdd` the specification of the parameter `Nat` is the same as the already defined specification of the same name. In such a case one can use a shorthand omitting the colon and the repeated name, as we do in the parameter `NatAdd` of specification `Test`.

According to the semantics of linking, the module `NatAdd`, which is implicitly specified by specification `NatAdd`, will be composed with module `Nat`. To simplify inspection of code, the argument modules are required to appear before the module that depends on them. Library List is parameterized too, but as a library it does not take part in linking, until included using the keyword load. Therefore, although there is no module Elem defined in the above example, no error occurs.

The use of named libraries is similar to the use of named specifications — they both do not take part in linking, until explicitly included (libraries by loading and specifications by recalling them by name). Both specifications and libraries can be "consumed" many times. On the other hand, ordinarily declared modules are used only once, and only in their immediately enclosing linking expression.

While List is a library, NatList is a module, so during linking, it will be composed with its argument, which is the module Nat. The module operation of instantiation and the specialization operation on specifications (both denoted by with) will be described in Section 2.3.2. In short, the effect of the presented specialization is that specification NatList is parameterized not by Elem, but by Nat. Consequently, the specification Test is parameterized by two entities: NatList and its parameter Nat.

The module Test depends on one more parameter, called NatAdd. Without exposing this dependency the NatAdd.pred value couldn't be used in the structure Test. The semantics ensures that the Nat parameter of specification NatList and of specification NatAdd will be shared. As a result, the Test module will be composed with a total of three (not four, with two copies of Nat) module arguments during linking.

2.3 Advanced modular programming

2.3.1 Specifications

The two specification forms that are not associated with any module operation are the parameterized specification and the referring to (recalling, copying) a specification. As seen before, the recalling is denoted simply by the specification name and its effect is similar to quoting the specification's definition. Here we will discuss in more detail the semantics of parameterized specifications, which is less obvious. Remaining specification forms will be described together with their related module operations.

Parameterized specification

The syntax of a parameterized specification is similar to the syntax of core language function types. For example, inside the longer linking expression of Section 2.2.2 we could write;

```
~NatList:NatList ~NatAdd -> Test
```

The difference with respect to types is that uppercase labels are used (as always when naming modules) and that if a specification of a parameter is omitted, then a specification of the same name is recalled from the environment. Thus an equivalent version of the above parameterized specification is

```
~NatList ~NatAdd:NatAdd -> Test
```

The semantics of parameterized specifications is quite different from the semantics of core language function types. In fact, the modules of Dule are first-order, so nested parameterized specifications, if taken seriously, are not implementable. For this reason they are always flattened in the first step of computing their semantics. The specification above is equivalent to a flattened parameterized specification that is not expressible in the user-accessible language. Below is an expressible, but only partially flattened semantically equivalent version of this parameterized specification. We cannot remove the nested `Nat` in the specification of `NatAdd`, because of the references to type `Nat.t`.

```
~NatList
~Nat
~NatAdd:(~Nat ->
         sig
           value pred : ~n:Nat.t -> Nat.t
           value add : Nat . ~n:t ~it:t -> t
           value is_zero : ~it:Nat.t -> [‘True|‘False]
         end)
  -> Test
```

Consider the following example:

```
~Nat:(~Bool -> Nat) ~Nat2:Nat -> ~Bool2:Bool -> Bool
```

The fully flattened equivalent specification is

```
~Nat ~Bool ~Nat2:Nat ~Bool2:Bool -> Bool
```

There are four parameters. Their interdependencies are trivial, because named specifications are isolated from the influence of others in the same list of module parameters (see Section 9.2 for the discussion of the isolation operation). The specifications `Bool` and `Nat` have been compiled in the context of their definition, with neither `Nat2` nor the other specifications given as parameters. Being trivial, the interdependencies are fully expressible in the user language without the use of nested parameterization constructs, as so the fully flattened specification is expressible as well.

Explicitly specified module

An explicitly specified module, as in the following example:

```
:: sig type t end
struct type t = {} end
```

has the same semantics as without the explicit specification, but is required to fulfill the specification. The match has to be exact, as in the above example. If a module is richer than the specification states, then trimming is necessary, before the explicit specification can be assigned:

```
:: sig type t end
((:: sig type t type u value v : u end
  struct type t = {} type u = {} value v = {} end)
   :> sig type t end)
```

On the other hand, the set of parameters of the specified module is required only to be a superset of the specification's parameters. This is visible in case of the module `Test` of Section 2.2.2. The module is explicitly specified and implicitly specified, as well, because there is a `Test` specification defined beforehand. We can rewrite the doubly specified module to the following equivalent form:

```
:: Test
  (:: ~NatAdd -> Test
   struct
     value five_zeros = ...
   end)
```

We see that the module has one more parameter than the outer specification states. This is perfectly correct.

2.3.2 Module operations

Module projection

Module projection, written as a solitary module name, projects its arguments onto the chosen one. For example, the arguments may be gathered in a record and the projection may be applied using the instantiation operation, here written with an infix bar:

```
{Trivial = {T = {}}; Void = struct end} | Void
```

The result is equivalent to:

```
struct end
```

The specification of a projection is reconstructed from the way it is used. Additionally, if there is a previously defined specification of the same name as the name of the projection, then the projection gets implicitly specified. Hence, the specification reconstructed for the projection `TreeSort` in the following example:

```
spec TreeSort = ~TreeOps ~Balance -> Sort
MySort = TreeSort
```

is the same as below:

```
MySort = :: ~TreeOps ~Balance -> Sort
         TreeSort
```

which is the same as:

```
MySort = :: ~TreeOps ~Balance ~TreeSort:Sort -> Sort
         TreeSort
```

because a projection at a given name must have a component of the same name among its parameters. The last step of the inference has been possible due to the lax treatment of parameters in explicitly specified modules.

The convention of implicit specification of projections based on their names is useful when instantiating libraries. In the following example we use the operation of double record, denoted by double curly braces and described in section **Double record** below. The double record consisting of a projection is equivalent to a normal record of projections, one for each of the parameters of the original projection.

```
link
  spec NatAny = ~Nat -> sig value any : Nat.t end
  NatAny = struct value any = Nat.zero end
  library Test = :: ~NatAny -> sig value v : Nat.t end
                 struct value v = NatAny.any end
  LoadedTest = load Test with {{NatAny}}
end
```

The projection `NatAny`, through specification `NatAny`, has `Nat` among its parameters. In the result, the semantics of double record ensures that the library `Test` gets instantiated not only with `NatAny`, but also with `Nat` as required.

The same projection convention also causes some inconveniences. For instance, consider the module `NatWhole` inside the nested linking expression of the following example and the projection `NatWhole` applied to the linking expression using the instantiation operation. The existence of the specification `Nat` precludes changing the two occurrences of the name `NatWhole` to a more naturally looking name `Nat`.

```
spec Bool =
  sig
    type t
    value tt : t
    value ff : t
  end
spec Nat =
~Bool ->
  sig
    type t
    value t2ind : ~it:t -> ind nat: [‘Zero|‘Succ nat]
    value tde : ~it:t -> [‘Zero|‘Succ t]
    value zero : t
    value succ : ~n:t -> t
    value pred : ~n:t -> t
    value add : ~n:t ~it:t -> t
    value is_zero : ~it:t -> Bool.t
 end
Nat =
link
  spec NatBasic =
    sig
      type t
      value t2ind : ~it:t -> ind nat: [‘Zero|‘Succ nat]
      value tde : ~it:t -> [‘Zero|‘Succ t]
      value zero : t
      value succ : ~n:t -> t
    end
  NatBasic =
    struct
      type t = ind nat: [‘Zero|‘Succ nat]
      value t2ind = fun ~it -> it
      value tde = fun ~it -> it . de
      value zero = ‘Zero . con
      value succ = fun ~n -> n .‘Succ . con
    end
  spec NatAdd =
  ~Bool ~NatBasic ->
    sig
      value pred : ~n:NatBasic.t -> NatBasic.t
      value add : NatBasic . ~n:t ~it:t -> t
      value is_zero : ~it:NatBasic.t -> Bool.t
    end
  NatAdd =
    struct
      value pred = fun ~n ->
        match NatBasic.tde ~it:n with
        [‘Zero -> n
        |‘Succ nn -> nn]
      value add = fun ~n ~it ->
        match NatBasic.t2ind ~it with
        fold [‘Zero -> n
             |‘Succ n -> NatBasic.succ ~n]
      value is_zero = fun ~it ->
        match NatBasic.tde ~it with
        [‘Zero -> Bool.tt
        |‘Succ -> Bool.ff]
    end
  NatWhole = :: ~NatBasic ~NatAdd -> Nat
    struct
      type t = NatBasic.t
      value t2ind = NatBasic.t2ind
      value tde = NatBasic.tde
      value zero = NatBasic.zero
      value succ = NatBasic.succ
      value pred = NatAdd.pred
      value add = NatAdd.add
      value is_zero =  NatAdd.is_zero
    end
end | NatWhole
```

The above use of projection to extract a linked module is possible because the result of the linking operation is a record of all its, sufficiently composed, components. If in the specification `Nat` there was no dependency on `Bool`, the extracted module and the projection could be named `Nat`. As it is, the projection convention imposes that one of the parameters of the projection `Nat` is `Bool`. Yet, there is no component `Bool` in the record resulting from the linking expression, because `Bool` is not defined inside the expression, but in the preceding code. As a result, instantiating a `Nat` projection with the linking expression would be incorrect. In fact, already the specification reconstruction would fail, because the information from the specification `Nat` and the list of components of the linking expression are inconsistent.

Module record

Module records are written between braces, and so are product specifications:

```
:: {Nat : Nat; NatList : NatList; MyBool : Bool}
{Nat = Nat; NatList = load NatList; MyBool = Bool}
```

The record notation admits an abbreviation analogous to the one of the core language record. If the field of a record is a projection on the same label as the name of the field, then the equal sign and the projection may be omitted. Note that the omitted field must not be a library loading but a module projection. With the use of this shorthand, the record above can be rewritten to:

```
{Nat; NatList = load NatList; MyBool = Bool}
```

The product specifications can be abbreviated similarly as parameterized specifications. If a specification at a given label is a reference to a previously defined specification of the same name as the label, then the colon and the reference may be omitted. Thus, the product of specifications from the beginning of this section may be written as follows:

```
{Nat; NatList; MyBool : Bool}
```

The linking operation, described in Section 2.2.2, is a generalization of the record operation. If there is no need of implicit composition, named specifications, inductive modules, etc., ordinary record suffices. It suffices, in particular, for supplying arguments to libraries, provided that each argument is to be explicitly listed. If there are many arguments, the double record operation can be more convenient.

Double record

The operation denoted by double curly braces is the module double record. It differs from the module record in that there are some additional implicitly inserted components. The additional components are projections; one for each parameter of the outcome module. For instance the double record

```
{{Trivial = {};
  Dependent = :: ~Parm:sig end -> sig end struct end;
  K2 = :: ~M1:sig end -> sig end  M2}}
```

is equivalent to the following ordinary record of modules

```
 {Trivial = {};
  Dependent = :: ~Parm:sig end -> sig end struct end;
  Parm = Parm;
  K2 = :: ~M1:sig end -> sig end  M2;
  M1 = M1;
  M2 = M2}
```

A double record operation with projections as its only operands “flattens” its arguments similarly as a parameterized specification flattens its parameters. This is illustrated in the following example, where we assume some of the specifications are already defined. Whenever there is a library with a parameterized specification, it can be instantiated with a double record of projections of the same names, as seen in the last line of the example. If we used the ordinary module record, we would have to list all nine projections instead of the three.

```
link
  spec M3 = ~M1 ~M2 -> sig ... end
  M3 = struct ... end
  spec M5 = ~M4 -> sig ... end
  M5 = struct ... end
  spec M9 = ~M6 ~M7 ~M8 -> sig ... end
  M9 = struct ... end
  library L = :: ~M3 ~M5 ~DifferentName:M9 ->
                   sig value c : M1.t ... end
              struct value c = M1.c ... end
  LInst = load L with {{M3; M5; DifferentName = M9}}
end
```

Instantiation

The normal module composition, not available to the programmer, but implicitly performed during linking, fills in implementation details in parameterized modules. The module instantiation not only does that, but also allows the argument module to influence the result specification of the whole operation.

The longest example of Section 2.2.2 illustrates the module instantiation operation and the corresponding specification specialization construct, both denoted by an infix `with`. Here are two most relevant lines:

```
spec NatList = List with {Elem = Nat}
NatList = load List with {Elem = Nat}
```

The records in the two lines are the same, while the `List` identifiers denote: the first — a specification, the second — a library to be loaded.

Let us add the following specification to the example code:

```
spec Singelton =
~NatList ->
```

```
  sig
    value singleton : ~elem:Nat.t -> NatList.t
  end
```

Notice that the type of the parameter `elem` is written `Nat.t`. If the programmer wrote `NatList.Elem.t` or `NatList.Nat.t` this would be wrong, because `NatList` is not a product specification. On the other hand, `Elem.t` in the same place would be wrong, because the specification specialization operation replaces the parameter `Elem` by `Nat` and `Elem` disappears from the result.

The specification of the instantiated module could always be written anew by hand, at least as long as our specification do not contain axioms and proof obligations. But the laborious manual rewrite would result in confusing code duplication. For example, the specification `NatList` above could have been written in the following way (compare with specification `List` in Section 2.2.2):

```
spec NatList =
~Nat ->
  sig
    type t
    value t2ind :
      ~it:t -> ind list: [‘Nil|‘Cons {head : Nat.t;
                                      tail : list}]
    value ind2t :
      ~it:ind list: [‘Nil|‘Cons {head : Nat.t;
                                 tail : list}] -> t
    value tde :
      ~it:t -> [‘Nil|‘Cons {head : Nat.t; tail : t}]
    value nil : t
    value cons : ~head:Nat.t ~tail:t -> t
  end
```

The version of the module instantiation operation denoted by the keyword `with` employs implicit trimming (the explicit trimming denoted by “`:>`” is described in the next section). If the module used as argument has too rich a specification, like `Nat` in the example above, it is trimmed down to match the parameter specification. The same implicit trimming happens in the specification specialization operation.

The operation of module instantiation denoted by a vertical bar does no trimming. It requires that the modules have strictly compatible specifications, as in:

```
TrivElem = struct type t = {} end
NatList = {Elem = TrivElem} | load List
```

In the following example the trimming is needed, because the type `Nat.t` from the specification `Nat` is recorded as part of the specification `NatElem`, while `List` expects only one type.

```
spec NatElem = ~Nat -> sig type t end
NatElem = struct type t = Nat.t end
NatList = {Elem = NatElem :> Elem} | load List
```

On the other hand the following form of declarations allows one to do without trimming, because the type names of `Nat` are not present at the time when specification `Elem` is built. When we define specification `NatElem`, specification `Elem` is already defined and isolated (see section **Parameterized specification** above) and consequently module `NatElem` contains only one type: `t`.

```
spec Elem = sig type t end
spec NatElem = ~Nat -> Elem
NatElem = struct type t = Nat.t end
NatList = {Elem = NatElem} | load List
```

The (reverse) order of operands of the "`with`" instantiation notation is chosen to match the syntax of specification specialization operation. The order of operands of the vertical bar notation matches the order in the categorical composition of modules, which is not available to the user, but denoted by a dot in the internal module language. While in the categorical composition `M1 . M2` the module `M2` is given a tool `M1` to operate with, in `M1 | M2` module `M2` is instantiated to a new area `M1` to operate on, and the specification of `M2` is specialized by `M1`.

We have already seen the module instantiation written with vertical bar in our examples involving module projections, for example:

```
{Trivial = {T = {}}; Void = struct end} | Void
```

In Dule, components of a module record are accessed by "composing" (instantiating) the record with a projection module, hence the same syntax in the two roles indeed denotes the same semantics. See Section 5.2.5 and 9.2.2 for further technical discussion.

Trimming

The trimming is denoted by the "`:>`" symbol. Most of the time trimming happens as part of the instantiation and specialization operations. For example the instantiation of modules defined in Section 2.2.2

```
load List with {Elem = Nat}
```

is equivalent to

```
{Elem = Nat :> sig type t end} | load List
```

There are situations when explicit trimming is useful to avoid writing interface modules by hand. In the following example we no longer refer to the specifications and libraries of Section 2.2.2 and subsequent sections, but to the library `Nat` of the standard prelude.

```
spec Nat' =
  sig
    type t
    value t2ind : ~it:t -> ind nat: [‘Zero|‘Succ nat]
    value tde : ~it:t -> [‘Zero|‘Succ t]
    value zero : t
    value succ : ~n:t -> t
    value eq : ~n:t ~it:t -> [‘True|‘False]
  end
Nat' = load Nat :> Nat' (* the prelude version is too rich *)
spec Stamp =
  sig
    type t
    value zero : t
    value succ : ~n:t -> t
    value eq : ~n:t ~it:t -> [‘True|‘False]
  end
Stamp = Nat' :> Stamp
spec Expressions =
~Nat' ->
  sig
    ...
  end
Expressions = struct ... end
spec HashedExpressions =
~Stamp ~Expressions ->
  sig
    ...
  end
HashedExpressions = struct ... end
```

The specification `Stamp` protects the user from an erroneous usage of stamp in `HashedExpressions`. For example, an attempt to add a stamp to a value of an `Expressions` constant expression would be rejected by the compiler. The trimming of the projection `Nat'` is a short way of building the `Stamp` module out of the module of natural numbers. (The first trimming was of the library `Nat` from the standard prelude that has a richer specification than needed in this example.)

A module may be trimmed with a parameterized specification, as well. However the parameters are ignored when cutting down the module. They are only checked against the parameters of the module, in the same way as in the case of explicitly specified module (see Section 2.3.1).

Trimming can be used not only to remove some components from a module but more generally to coerce a module to a given specification:

```
Coerce =
  (:: sig type t value v : t end
   struct type t = {} value v = {} end) :>
     sig value v : {} end
```

The following extended example shows that trimming, just as explicit specification of a module, is translucent. In other words a module is not made abstract by trimming.

```
StillCoerce =
  (:: sig type t value v : t end
   struct type t = {} value v = {} end) :>
     sig type t value v : t end :>
       sig value v : {} end
```

However, when the coerced module is a projection, the effect of the coercion is always only a removal of some type and value components. Consider the following example.

```
TV = :: sig type t value v : t end
     struct type t = {} value v = {} end
WeakCoerce = TV :> sig type t (* value v : {} *) end
```

Here the part enclosed in a comment would cause an error, because module projections are always, in a way, abstract. Since the trimming operation is usually applied to projections, the name “trimming” has a practical justification.

The only way to treat a module that is not a projection as an abstract module, is to perform an analogue of λ-abstraction at the module level. In this way a module can be locally seen as abstract, for instance inside an explicitly specified base module (where outside modules are seen through the parameters of the specification as abstract entities) or inside a whole linking expression (where all the parameters of the expression as well as all the modules being named are seen as abstract). Every such local abstraction of a module can still be reversed from outside by global coercion.

2.3.3 Recursive modules

Recursive specifications

Specifications in Dule can be mutually recursive. The following example of recursive specifications describes the mutually recursive nature of the syntax of Dule module language itself:

```
spec rec Sp = (* module specifications *)
  sig
    type t
    ...
    (* specification specialization: *)
    value s_Ww : ~m1:Dule.t ~s2:t -> t
  end
and Dule = (* modules *)
~IdIndex ->
  sig
    type t
    ...
    (* module projection: *)
    value m_Pr : ~i:IdIndex.t -> t
    (* module instantiation: *)
    value m_Inst : ~m1:t ~m2:t -> t
    (* module trimming: *)
    value m_Trim : ~m1:t ~r2:Sp.t -> t
  end
```

Simplifying, this definition has the effect of prefixing the individual specification definitions `Sp` and `Dule` with a list of parameters containing both `Sp` and `Dule`. This is a simplification: the specifications in the list of parameters would not be themselves well-defined, unless they are also prefixed, etc. For this reason the recursive specifications are not, in general, expressible using the other operations.

Due to the flattening of parameterized specifications, having a recursive specification among parameters of a module entails the inclusion of all the other mutually dependent specifications and their respective parameters. This may result in a handy abbreviation, or a dangerous cluttering of the module's name space. In the latter case, there is a possibility of writing a smaller specification that would be weaker than the original recursive one. If fact, recursive specifications can always be built in this simple way from such small specifications. For example, the above definitions can be recast in the following way:

```
spec SpAlone =
~Dule:sig type t end ->
```

```
  sig
    type t
    ...
    value s_Ww : ~m1:Dule.t ~s2:t -> t
  end
spec DuleAlone =
~IdIndex ~Sp:sig type t end ->
  sig
    type t
    ...
    value m_Pr : ~i:IdIndex.t -> t
    value m_Inst : ~m1:t ~m2:t -> t
    value m_Trim : ~m1:t ~r2:Sp.t -> t
  end
spec rec Sp = SpAlone with {Dule}
and Dule = DuleAlone with {IdIndex; Sp}
```

Then the both the non-recursive and recursive specification definitions can be used in definitions of modules. We only provide a sketch — a detailed discussion of inductive modules is carried out in the next section.

```
module ind Sp = struct ... end
and Dule = struct ... end
spec PrintSpAlone =
~Sp:SpAlone ~String ->
  sig
    value print : ~s:Sp.t -> CharList.t
  end
library PrintSpAlone = struct ... end
PrintSp = load PrintSpAlone with {{Sp; String}}
```

Thus instantiated module `PrintSp` will now successfully link with the inductive modules `Sp` and `Dule`. The instantiation itself is correct, because specification `SpAlone` is weaker than specification `Sp`, which means that implicit trimming in the instantiation succeeds.

If there is no mutual dependency among the types of values listed in specifications, the recursive specifications can be obtained from non-recursive prototypes in the most straightforward way. This is illustrated with specifications of error-checking procedures for `Sp` and `Dule`. The modules `CheckSp` and `CheckDule` should obviously be mutually recursive, as seen in their extended code in the next section. In particular they should be available as module parameters to one another, which is easily expressible using recursive specifications.

```
spec ChSp =
~Sp ->
  sig
    value check_s : ~s:Sp.t -> [‘OK|‘Error]
  end
spec ChDule =
~Dule ->
  sig
    value check_m : ~m:Dule.t -> [‘OK|‘Error]
  end
spec rec CheckSp = ChSp
and CheckDule = ChDule
module ind CheckSp = struct ... end
and CheckDule = struct ... end
spec PrintWarning =
~CheckDule:ChDule ~String ->
  sig
    value warn : ~m:Dule.t -> CharList.t
  end
PrintWarning = struct ... end
```

Now the module `PrintWarning`, which uses `CheckDule` but doesn't know about `CheckSp`, is ready for linking without further modifications.

Inductive modules

In Dule, the mutual dependency among modules can be expressed using inductive modules or using coinductive modules. The difference lies in the model of the mutually dependent type parts of the modules. These parts are realized either as one huge inductive type, or an analogous coinductive type. The dependencies on values are always modeled by general recursion. As explained in Section 9.1, in the current variant of the Dule core language structured recursion does not work on the types of inductive or coinductive modules. The types can be manually equipped with coercions into core language (co)inductive types, but this is done with general recursion and (co)inductive constructors, which behave the same in the inductive and coinductive setting. For this reason the inductive and coinductive modules are currently behaviorally indistinguishable.

A standard example of mutually dependent datatypes, naturally divisible into separate modules, is the syntax of a sufficiently complex programming language. Now we fill up some of the holes in the definitions of specifications and modules from the previous section, describing the syntax of the Dule module language. The sum type constructors, marked by back-quotes below, represent constructors of the abstract syntax trees; in particular, the constructor of specification specialization `‘S_Ww`, module projection `‘M_Pr`, module instantiation `‘M_Inst` and trimming `‘M_Trim`. As the form of specifications `Sp` and `Dule` suggests, there are nontrivial dependencies between the type parts of inductive modules `Sp` and `Dule`. On the other hand, there are no dependencies among the values:

```
spec SpAlone =
~Dule:sig type t end ->
  sig
    type t
    value tde : ~s:t ->
             [...
             |‘S_Ww {m1 : Dule.t; s2 : t}]
    ...
    value s_Ww : ~m1:Dule.t ~s2:t -> t
  end
spec DuleAlone =
~IdIndex ~Sp:sig type t end ->
  sig
    type t
    value tde : ~m:t ->
             [...
             |‘M_Pr IdIndex.t
             |‘M_Inst {m1 : t; m2 : t}
             |‘M_Trim {m1 : t; r2 : Sp.t}]
    ...
    value m_Pr : ~i:IdIndex.t -> t
    value m_Inst : ~m1:t ~m2:t -> t
    value m_Trim : ~m1:t ~r2:Sp.t -> t
  end
spec rec Sp = SpAlone with {Dule}
and Dule = DuleAlone with {IdIndex; Sp}
module ind Sp =
  struct
    type t =
      ind t: [...
             |‘S_Ww {m1 : Dule.t; s2 : t}]
    value tde = fun ~s -> s . de
    ...
    value s_Ww = fun ~m1 ~s2 -> {m1; s2} .‘S_Ww . con
  end
and Dule =
  struct
    type t =
      ind t: [...
             |‘M_Pr IdIndex.t
             |‘M_Inst {m1 : t; m2 : t}
             |‘M_Trim {m1 : t; r2 : Sp.t}]
    value tde = fun ~m -> m . de
    ...
    value m_Pr = fun ~i -> i .‘M_Pr . con
    value m_Inst = fun ~m1 ~m2 -> {m1; m2} .‘M_Inst . con
    value m_Trim = fun ~m1 ~r2 -> {m1; r2} .‘M_Trim . con
  end
```

We continue the definition of type-checking procedures for `Sp` and `Dule`. As the specifications themselves hint at, there are no dependencies between the type parts of `CheckSp` and `CheckDule`. Yet the value parts depend heavily on one another, so the modules have to be mutually dependent.

```
spec ChSp =
~Sp ->
  sig
    value check_s : ~s:Sp.t -> [‘OK|‘Error]
  end
spec ChDule =
~Dule ->
  sig
    value check_m : ~m:Dule.t -> [‘OK|‘Error]
  end
spec rec CheckSp = ChSp
and CheckDule = ChDule
module ind CheckSp =
  struct
    value check_s = fun ~s ->
      match Sp.tde ~s with
      [...
      |‘S_Ww {m1; s2} ->
          match CheckDule.check_m ~m:m1 with
          [‘OK m1 ->
              match CheckSp.check_s ~s:s2 with
              [‘OK s2 ->
                  ...
              |‘Error er -> er .‘Error]
          |‘Error er -> er .‘Error]]
  end
and CheckDule =
  struct
    value check_m = fun ~m ->
      match Dule.tde ~m with
      [...
      |‘M_Pr i -> ‘OK
      |‘M_Inst {m1; m2} ->
          match CheckDule.check_m ~m:m1 with
          [‘OK m1 ->
              match CheckDule.check_m ~m:m2 with
              [‘OK m2 ->
                  ...
              |‘Error er -> er .‘Error]
          |‘Error er -> er .‘Error]
      |‘M_Trim {m1; r2} ->
          match CheckDule.check_m ~m:m1 with
          [‘OK m1 ->
              match CheckSp.check_s ~s:r2 with
              [‘OK r2 ->
                  ...
              |‘Error er -> er .‘Error]
          |‘Error er -> er .‘Error]]
  end
```

Notice how the recursive references of `check_s` value to itself and of `check_m` value to itself are captured using the same construction that models the mutually recursive references. There is no keyword `rec` in the function definitions, because the functions are determined to be recursive by the mechanism of inductive modules alone.

Here is another example of inductive modules with a nontrivial mutual dependency among the type parts:

```
spec IList =
~IdIndex ~Value ->
  sig
    type t
    value nil : t
    value cons : ~i:IdIndex.t ~v:Value.t ~l:t -> t
  end
library IList =
  struct
    type t =
      ind t: [‘Nil|‘Cons {i : IdIndex.t; v : Value.t; l : t}]
    value nil = ‘Nil . con
    value cons = fun ~i ~v ~l -> {i; v; l} .‘Cons . con
  end
spec rec CatIList = IList with {Value = Cat; IdIndex}
and Cat =
  sig
    type t
    value tde : ~c:t ->
            [‘C_PP CatIList.t
            |‘C_BB]
    value c_PP : ~lc:CatIList.t -> t
    value c_BB : t
  end
module ind CatIList = load IList with {{Value = Cat}}
and Cat =
  struct
    type t =
      ind t: [‘C_PP CatIList.t
            |‘C_BB]
    value tde = fun ~c -> c . de
    value c_PP = fun ~lc -> lc .‘C_PP . con
    value c_BB = ‘C_BB . con
  end
```

In the above example there is no dependency between the values in different modules, and the mutual dependency between the types is of the kind that in standard functional languages is often captured using a parameterized type (`'a IList.t`) embedded into a recursive (but not mutually recursive) definition of another type (`Cat.t`).

2.4 Conclusion

Dule is a statically and strictly typed applicative language with both structured and general recursion. Its module system features applicative module composition with both transparent and non-transparent versions (see Section 5.2.5), various module grouping mechanisms and (co)inductive modules. Let us compare the Dule programming language to a conventional language with signatures and functors, such as Standard ML [119].

2.4.1 Absence of standard mechanisms

One kind of the distinguishing properties of Dule is the absence of a standard programming mechanism, intended to obligate the programmer to write code in a different and (we hope) better style.

No sharing equations

There are no sharing equations (or "`where`" clauses or type abbreviations). Those mechanisms are the most common cause for large and unreadable module headers. In the Dule module system, sharing is the default behavior, so there is no need to require sharing explicitly. Thus module operations become more light-weight for the programmer both in the size of textual representation and the frequency of compiler complaints (assuming the programmer adheres to the one-name-one-meaning principle).

The categorical record together with module instantiation allows for renaming of module parameters. The default sharing behavior is based on module names, so the renaming is useful when a sharing needed by the programmer happens not to coincide with the naming of modules at hand. Here are three examples of renaming, the first two may additionally coerce renamed parameters.

```
spec IntQueue = Queue with {Order = Int}
IntQueue = load Queue with {Order = Int}
module IntSet =
  {Order = Int; Queue = IntQueue} | load Set
```

No submodules

There are no substructures or submodules. Substructures are the second cause for verbose modular notation in some modular programming styles. In a very modularized program, where sharing is specified using substructures and sharing equations, the modular book-keeping may take as much as half of the program text size. In Dule, again, including arguments of a structure as its "substructures" is usually the default (more strictly, only the type parts of argument structures are inherited and only of the arguments mentioned in the signature).

There are no submodules, but when a real grouping of modules is needed, a categorical labeled record of modules serves the purpose. There are also other related operations, such as double record and linking, specialized for different patterns of dependency among the modules and their parameters. Strict scoping rules for the linking operation prevent name-space pollution, but allow the use of external libraries.

No polymorphism

There is no polymorphism at the core language level. Any dependency on types in Dule is expressed modularly. This enforces better awareness of multiple type instantiations of entities appearing in the program, usually making the interfaces narrower and proving polymorphism unnecessary for such cases. Another result can be the early elimination of potential bugs, when the more strict module dependency mechanism explicitly imposes requirements on the types to be used in instantiations. For example, if an ordering on a type is needed only in a rarely used operation, other polymorphic operations of the same module can be successfully applied to values without ordering, leading to a ground-up rewrite when the ordering is finally found necessary.

Sometimes the polymorphic behavior can be recovered (with a completely different flavor but staying at the core language level) by polytypism [91], using the mapping and iteration combinators (even across modules, if enabled by conversions such as `t2ind` on page 54). The latter would be especially powerful in the variant of Dule with direct iteration over types resulting from (co)inductive modules (see Section 9.1). The polymorphic, more so than polytypic, nature of the mapping combinator is seen in the following example.

```
let intlist = {h = 1;
               t = {h = 2;
                    t = ‘Nil . con} .‘Cons . con} .‘Cons . con
    boollist = {h = ‘True;
                t = {h = ‘False;
                     t = ‘Nil . con} .‘Cons . con} .‘Cons . con
    monomorphic_map =
      map fun ~it -> {x = it; y = it}
in
  {intmap =
     (map fun ~it -> {x = it; y = it}) ~it:intlist
  ;boolmap =
     (map fun ~it -> {x = it; y = it}) ~it:boollist
  ;intmap2 =
     monomorphic_map ~it:intlist

(* this would be incorrect:

  ;boolmap2 =
    monomorphic_map ~it:boollist
 *)
  }
```

If `map` was not “polymorphic”, different versions would have to be used for integer lists and boolean lists. Note, however, that as soon as the combinator is named (as with `monomorphic_map` above) it becomes monomorphic, even if the exact monomorphic type is not determined until the name is used (under `intmap2` above). Consequently, two different uses of the named mapping can lead to inconsistent typings, as would be the case with `boolmap2` in the example above.

No dependency

In core-level definitions there is no way to refer to other entities from the same module. Generally, any dependencies between components of Dule modules are expressed through the module system, be it components from different modules or the same module. In particular, the code of a value cannot refer to another value from the same module. This results in smaller modules representing levels of abstraction, each with a set of (usually few) orthogonal operations.

2.4.2 Additional ways of expressing

Dule provides alternative ways of expressing many common modular idioms. We will discuss three such new mechanisms.

Implicit composition

Module composition may be performed not only explicitly, but also implicitly. This cuts down on the last source of module bureaucracy. The implicit composition performed during linking corresponds to non-transparent functor application. The less abstract but more purposeful transparent application is modeled by the module instantiation. It always has to be explicit, but occurs rarely in ordinary programs. The implicit composition also adds more importance to naming of modules. In the fragment of a program where all module applications are implicit, identical names always correspond to identical modules.

Solving conflicts

Dule facilitates many ways of solving module requirement conflicts, without ad hoc module hierarchy modifications (and without module stamps or other semantic tools compromising safety). When a program is very abstractly modularized, the same module is often seen from various points of view, with different specifications, derived from what other modules expect of it. The various views of a single module may cause inconsistencies.

If a module `Set` has two parameters `Queue` and `Order`, where specification `Queue` depends on parameter `Order`, then there are two main kinds of potential conflicts. The first is that the actual argument module `Queue` expects more of

`Order` than `Set` expects of `Order`, and the second that `Set` expects more of `Order` than `Queue` does. Both the kinds of conflicts have elegant and original solutions in Dule, without rewriting `Set`, `Queue` or `Order`.

Here is an example of the conflicts and their solutions. Notice that neither `Order` nor `Int` is implemented in this (implicit) linking expression and so `Int` becomes a parameter of the resulting module. The libraries are used only for readability. The main tool used here is the module instantiation operation.

```
spec Order =
  sig
    type t
    value leq : ~n:t ~it:t -> [‘True|‘False]
  end
spec Int =
  sig
    type t
    value leq : ~n:t ~it:t -> [‘True|‘False]
    value zero : t
  end
spec Queue =
~Order ->
  sig
    type t
    value empty : t
    value append : ~e:Order.t ~q:t -> t
  end
spec Set =
~Queue ->
  sig
    type t
    value empty : t
    value add : ~e:Order.t ~s:t -> t
    value set2queue : ~s:t -> Queue.t
  end

spec IntQueue = Queue with {Order = Int}

(* case 1: [IntQueue] has stronger [Order] than [Set] *)

module IntQueue =
  struct
    type t = ind l: [‘Nil|‘Cons {head : Int.t; tail : l}]
    value empty = ‘Nil . con
    value append = fun ~e:_ ~q -> (* contrived *)
      {head = Int.zero; tail = q} .‘Cons . con
  end

library Set =
  struct
    type t = Queue.t
    value empty = Queue.empty
    value add = fun ~e ~s -> Queue.append ~e ~q:s
    value set2queue = fun ~s -> s
  end

(* we would like to just apply [Set] to [IntQueue],
   but there is a conflict, because [IntQueue] expects
   more of the module of its elements
   than [Set] expects of [Order] *)

module IntSet1 = (* conflict solved: *)
  load Set with {Order = Int; Queue = IntQueue}

(* case 2: [IntSet] has stronger [Order] than [Queue] *)

library Queue =
  struct
    type t = ind l: [‘Nil|‘Cons {head : Order.t; tail : l}]
    value empty = ‘Nil . con
    value append = fun ~e ~q -> (* contrived *)
      {head = e; tail = q} .‘Cons . con
  end

spec IntSet = Set with {Order = Int; Queue = IntQueue}

library IntSet =
  struct
    type t = IntQueue.t
    value empty = IntQueue.append ~e:Int.zero ~q:IntQueue.empty
    value add = fun ~e ~s -> IntQueue.append ~e ~q:s
    value set2queue = fun ~s -> s
  end
```

```
(* we would like to just apply [IntSet] to [Queue],
   but there is a conflict, because [IntSet] expects
   more of the module of its elements
   than [Queue] expects of [Order] *)

module IntSet2 = (* conflict solved: *)
  load IntSet with {{IntQueue = load Queue with {Order = Int}}}
```

Pseudo-higher-order notation

A specification in Dule may be written using a pseudo-higher-order notation for parameters. This should be useful for thinking about modules as if they were higher-order. At the same time Dule stays first-order and avoids the typical pitfalls and subtleties of higher-order modular programming, such as sharing with formal parameters, sharing with not fully applied modules, contravariance of signature subsumption, mixed variance type dependencies, and similar problems analyzed in literature [128]. An example of flattening the pseudo-higher-order specifications is given on page 60.

2.4.3 New expressible entities

There are a few situations where it is possible to capture in Dule some modular phenomena not expressible in other languages.

Specifications of transparent functor applications

The result specifications of transparent functor applications (modeled by the instantiation operation) are always expressible in Dule (using the specification specialization operation denoted by `with`). There is no restriction as to the form of the operand modules in the application. Similarly expressive but different specifications have been proposed in [104].

Recursive specifications

Specifications in Dule can be defined recursively and mutually recursively. The main use of the mechanism is for specifying mutually dependent modules, but the conciseness of the notation of recursive specifications can be useful in any case where the specifications of the result and one of the parameters coincide. For example:

```
spec rec Bootstrap_heuristics =
~StateSet ~Evaluation ->
  sig
    value restrict : ~s:StateSet.t -> StateSet.t
      (* picks a subset of [Bootstrap_heuristics.restrict ~s]
         so as to maximize [Evaluation.grade ~s] *)
  end
```

Inductive modules

Dule features the novel mechanism of (co)inductive modules, which provide, up to observational indistinguishability, a fixpoint construction on modules. In the presence of module products, the mechanism easily extends to mutually dependent families of modules. Both the operands of the (co)inductive construction and the modules resulting from it are first-class citizens of the module system and their specifications are easily expressible. The correctness of the construction depends solely on specifications of modules, similarly as in the case of value fixpoint and its typing.

The following example shows how one could define inductive types (such as the type of unary natural numbers) implicitly, by using the induction on modules instead of on types and values. This is an odd thing to do, but the possibility of such an application proves that the mutually recursive modules of Dule are constructed by a real recursion on modules rather than some ad hoc mixing transformations:

```
spec rec Nat =
  sig
    type nat
    value t2ind : ~it:nat -> ind nat: [‘Zero|‘Succ nat]
    value tde : ~it:nat -> [‘Zero|‘Succ Nat.nat]
    value zero : nat
    value succ : ~it:Nat.nat -> nat
    value is_zero : ~it:nat -> [‘True|‘False]
    value plus : ~n:Nat.nat ~m:Nat.nat -> Nat.nat
    value two : Nat.nat
  end
module ind Nat =
  struct
    type nat = [‘Zero|‘Succ Nat.nat]
    value t2ind = fun ~it ->
      match it with
      [‘Zero -> ‘Zero . con
      |‘Succ n -> (Nat.t2ind ~it:n) .‘Succ . con]
    value tde = fun ~it -> it
    value zero = ‘Zero
    value succ = fun ~it -> it .‘Succ
    value is_zero = [‘Zero -> ‘True|‘Succ -> ‘False]
    value plus = fun ~m ~n ->
      match Nat.tde ~it:n with
      [‘Zero -> m
      |‘Succ nn -> Nat.succ ~it:(Nat.plus ~n:nn ~m)]
    value two =
      let one = Nat.succ ~it:Nat.zero in
      Nat.plus ~m:one ~n:one
  end
spec Result =
~Nat ->
  sig
    value three : Nat.nat
  end
Result =
  struct
    value three =
      let one = Nat.succ ~it:Nat.zero in
      Nat.plus ~m:Nat.two ~n:one
  end
```

Notice that there is no keyword `ind` in the definition of type `nat`. There is also no keywords `fold` or `rec` in the definition of `t2ind` and `plus`. The definition of `tde` is surprisingly simple — it doesn't even contain any inductive type destructor.

Like any other result of the recursive closure, the specification `Nat` is a parameterized specification, with one of the parameters (and in this case, the only one) named `Nat` as well. When the parameterized specification `Nat` found in the specification `Result` is flattened, then its result and its parameter of the same name are blended together. Outside of the recursive specification definition, when the recursion has already taken place, this blending is semantically correct. On the other hand, inside the definition of the inductive module `Nat` the types `Nat.nat` and `nat` are distinct. Any violation of this distinction would result in a typing error.

Chapter 3

Preliminaries

3.1 Prerequisites

Our thesis uses extensively some basic notions of category theory. The focus is on 2-categories (see for example Chapter 3 of [82]), in particular on the notions of horizontal and vertical composition as well as multiplication by a functor (see for example [133]). Another basic concept extensively used and somewhat generalized here is that of categorical adjunction (see for example [51]). In some sections we also mention exponential object, defined in our thesis through adjunctions, just as in Chapter 7 of [82].

Almost all our categorical and logical terminology is standard, though the notions are sometimes generalized. Each of the notions is briefly described in our thesis and the adopted variant is rigorously defined, usually by means of a complete axiomatization. However, a source of independent examples, diagrams and intuition, such as a good textbook on category theory [12] would make an invaluable companion for the reader with insufficient categorical background.

The standard axiomatization of a categorical binary product is

$$<f_1\,,\ f_2>\ .\ \pi_1 \ = \ f_1 \tag{1}$$

$$<f_1\,,\ f_2>\ .\ \pi_2 \ = \ f_2 \tag{2}$$

$$<f\ .\ \pi_1\,,\ f\ .\ \pi_2> \ = \ f \tag{3}$$

where the dot is the categorical composition written in diagrammatical order, π_1 and π_2 are projections and $<\_,\_>$ is the factorizer operation, also known as transpose, adjunct, tuple or record. We call the first and the second equation the existence requirements, because they require that the factorizer of each arbitrary pair of morphisms exists. We call the third equation the uniqueness requirement, because it encodes the uniqueness of the factorizer of any given two morphisms.

A categorical diagram is a directed graph with vertices labeled by objects and edges by morphisms, so that each edge comes from a vertex labeled by the domain of its morphism to a vertex labeled by the codomain of its morphism. A categorical cone over a diagram is a family of morphisms with a common domain and with codomains in all objects of the diagram, such that each triangle formed by a morphism of the diagram and two morphisms of the cone is a commuting diagram. A limiting cone of a diagram D is such a cone over D, with the common domain A, that for each cone with a domain B there is exactly one morphism from B to A, such that the sum of both cones and the morphism is a commuting diagram. The common domain A of the limiting cone is called the limit object or just the limit of the diagram.

The product is a special case of a limit; it is a limit of a diagram containing only objects. A limit of a diagram containing only two morphisms with equal domains and codomains is called equalizer. A limit of a diagram consisting of morphisms with a common codomain is called a pullback. There are various constructions of general limits from pullbacks or from products and equalizers [110]. In Chapter 5 we will refer to the construction given in [131].

For a given set of labels S, a limit of a diagram with vertices uniquely labeled by elements of S is called a labeled limit (with labels from set S). Then the morphisms of the limiting cone can be referred to by the labels of their codomain vertices. Labeled product is a special case of a labeled limit. For example a labeled product with labels `1`, `2` and `A` is a labeled limit of a diagram consisting of three objects labeled `1`, `2` and `A`. The three morphisms in the limiting code of this labeled product are called projections at labels `1`, `2` and `A`, respectively. We reformulate the product axiomatization above for labeled product in Sections 6.1.2, 5.2.4 and others. The axioms of products are generalized to arbitrary adjunctions, retaining their names (the existence requirement and the uniqueness requirement) in Section 7.1.2.

3.2 Formal notation

Among the propositions in this thesis, those marked **Observation** can be proved in a way we deem uninteresting: obvious, routine or similar to other propositions in the thesis. Such proofs are omitted. Details of nontrivial but long proofs from Chapters 6 and 7 are usually moved to Appendix B. The examples and remarks throughout the thesis are labeled with **Remark** and **Example** headings and set in a smaller font. These may be skipped at the first reading.

Some observations, lemmas, theorems and their proofs are not formal transcripts of mathematical facts, but rather informal statements helpful in constructing and analyzing the semantics and compiler of Dule. Their complete formalization is beyond the scope of our thesis, because the systems we chose to analyze are not small kernel languages but complex constructions necessary for large-scale realistic experiments. Nevertheless, the presented examples and descriptions of constructions should be sufficient and the given proofs, or rather arguments and sketches of reasoning, should be convincing enough to form a basis for possible complete formalization, which however does not seem necessary to reach the main goal of our thesis.

Most of the time, we use standard mathematical and logical notation for describing such objects as equations, rewrite rules, conditional equations, derivation rules. We gloss over foundational difficulties and write “set” for collections of morphisms of large categories, such as **Cat**, etc. To describe concrete syntax and derived forms, we employ a variant of the conventions used in the definition of CASL [27]. We give a detailed description of these conventions in Section 8.3.1.

The abstract syntax of our languages and most of the semantic functions are defined using our own meta-language, which is based on the functional programming language OCaml [105]. The meaning of this meta-language and its common idioms is discussed in the next sections. The choice of the meta-language follows from the fact that functional languages are very convenient for expressing (semantic) function definitions. Moreover, OCaml is a practical programming language with an efficient compiler, so such formal definitions can immediately undergo extensive test.

OCaml is one of the family of ML languages, another member of which is Standard ML [120]. While Standard ML is rigorously defined and its syntax virtually unchanged for over a decade, OCaml evolves rapidly. In particular, the syntax of OCaml has recently seen interesting extensions, like labels [55] and CamlP4, as well as inspiring variants, such as the CamlP4 revised syntax [37]. The advantage of using OCaml rather than any other functional notation is that OCaml and Dule are reasonably close, especially syntacticly. In particular getting acquainted with one of them should help understanding the other. Even an occasional stumble upon a syntactic difference between Dule and OCaml may lead to some insight into the semantical differences and thus improve understanding of the properties of Dule.

Abstract syntax definitions

We use OCaml type declarations to define grammars of formal languages or, equivalently, algebraic signatures for the (often syntactic) algebras we study. For example, an OCaml datatype `foo` with constructors `BAR` and `EMPTY` is defined as follows:

```
type foo =
  | BAR of foo * foo
  | EMPTY
```

Read according to our convention, it determines an algebraic signature (to be distinguished from Dule module language signatures) with one sort `foo` and two operations. The application of the algebraic operations introduced in such a way to variables or terms is denoted using standard mathematical notation:

$$\texttt{BAR}(x, \texttt{BAR}(\texttt{BAR}(z, \texttt{EMPTY}), y))$$

By `foo`-terms we mean expressions built according to the grammar given as a definition of the OCaml type `foo`. When a formal semantics is rigorously defined in a preceding text, we may sometimes refrain from distinguishing typographically a `foo`-term (such as `EMPTY`) and the semantic entity it denotes (the "empty" distinguished element of the semantic algebra defined by the preceding text), though usually we will mark the difference, using semantic combinators (see section **Semantic combinators** below) for the latter role.

When describing a semantic function or an algorithm in OCaml, we often operate on terms of the languages that have been presented using OCaml type declarations. While normally we use mathematical notation, in the OCaml expressions we take the liberty to manipulate terms using techniques specific to functional programming, such as pattern matching. These techniques and overall conventions that we adopt in our source code are described in detail in the next sections.

How to read function declarations

We use OCaml mechanisms of function declarations to define our semantic functions. The notation of OCaml is based on λ-calculus [11] with λ-abstraction denoted using the keyword `fun`. If x is a variable and e is an expression (possibly containing the variable) then

$$\texttt{fun}\ x\ \texttt{->}\ e$$

is a function that given x produces the value of e. OCaml introduces local definitions with the keyword `let`, for example

$$\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ e\ \texttt{in}\ f\ 1$$

where f 1 is the application of f to 1. There is also a shorthand allowing one to express abstraction together with declaration. For example, the function f can be defined as in the following expression

$$\texttt{let}\ f\ x\ \texttt{=}\ e\ \texttt{in}\ f\ 1$$

Recursive definitions are often used and are denoted with the prefix `let rec`.

In denotational semantics, semantic functions are defined traditionally using λ-calculus with minor extensions. Here, when defining semantics we will use not only λ-calculus written in OCaml syntax, but also some other mechanisms of OCaml (but only the purely functional ones). In particular, we will use pattern matching, beginning with the keyword `match`, as in the following expression.

```
fun nok ->
  (match nok with
  |‘OK n -> n + 1
  |‘Error er -> 0)
```

The expression denotes a function that applied to an “erroneous” argument produces zero and otherwise increments the argument. The type of the function’s parameter is an anonymous sum type

```
[‘OK of int|‘Error of string]
```

similar to the conventional OCaml datatypes, but not requiring declaration prior to use and with constructors marked by back-quotes. The pattern matching in OCaml works the same on both kinds of sum types.

Semantic combinators

In agreement with the methodology of denotational semantics we interpret program grammars as algebraic signatures and construct semantic algebras by defining their carriers (semantic domains) and operations. Traditionally, the definitions of the operations are grouped together into a semantic function definition presented using semantic equations. We will rather define the interpretations of operations, often called semantic combinators in the chapters to follow, gradually; one definition for each operation specified in the algebraic signature. We will always list the names of the semantic combinators to be defined, together with their full typing, shortly after the corresponding algebraic signatures are defined (using the OCaml datatype meta-language). The first two listings of semantic combinators are in Section 4.4 and in Section 5.2.1.

Example. Among the combinators of Section 5.2.1 there is one representing the operation of instantiation of modules.

```
val m_Inst : Dule.t -> Dule.t ->
               [‘OK of Dule.t|‘Error of string]
```

This combinator corresponds to the term constructor of module instantiation.

```
type dule =
  | M_Inst of dule * dule
  ...
```

The typing of the combinator differs slightly from what the algebraic signature suggests. First, the operands of the combinator are in the “curried” form. Second, the result is of the variant form which means that the operation is partial and that in case of failure there is a message detailing an error cause bound to the constructor `‘Error`. The semantic combinators always have the same names as the corresponding term constructors, but with the first letter written lowercase.

There is a prototype compiler of Dule written in OCaml. Most of it is a straightforward encoding of all the formal semantics. When a semantic combinator is defined in our thesis using OCaml, its definition is identical to the one in the source code of the compiler. The compiler works correctly and is even quite efficient, considering the emphasis on semantic clarity. The complete source code of our compiler can be found in directory `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code`. File names with extension `.ml` mark OCaml source code files, other files are auxiliary. The most important part of the Dule compiler has also a version written in Dule itself, see Appendix C.2. Other examples of Dule programs, in files with names ending with `.dul`, are available in directory `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/test`.

In the original source code, the OCaml functions are defined inside modules. For this reason, when they are used, sometimes their names should be prefixed with module names, depending on the details of the module organization. We took care that there are no unintended cases of functions of the same name provided by different modules, so the reader can safely ignore the prefixes when reading the code.

We will cite the source code together with comments, which in OCaml are written between symbols "`(*`" and "`*)`". There is a convention that source code quoted inside comments should be enclosed in square brackets. Comments intermingled with the code are sparse, so their presence should alert reader to an important or difficult detail. On the other hand, there are numerous comments describing general properties of operations, which usually precede main semantic combinators definitions. These will be cited in the thesis together with the definitions, and sometimes even cited separately and proved, just as any other propositions.

Indexed list operations

Indexed lists, that is lists of elements indexed by labels, are the main data structure used in the definition of the semantic combinators in this thesis. They start appearing when we begin defining the core language semantics in Section 4.1.1 and they are heavily used everywhere, especially in the definition of the semantics of modules. Outside of the source code we use lots of syntactic sugar for the indexed lists, especially once the abbreviations are carefully defined in Section 6.1.1.

In the source code we express and transform indexed lists using specialized operations. The OCaml indexed lists datatype is defined in module `IList` contained in file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/tools.ml`. The signature of the module is cited below and a detailed description follows. The module provides a considerable number of operations for programmer's convenience. However, most of them are instances of several general ones, as the names indicate.

```
type 'a t (* indexed lists *)

val nil : 'a t
val cons : (Index.t * 'a) -> 'a t -> 'a t

val is_nil : 'a t -> bool
val is_in : Index.t -> 'a t -> bool
val not_in : Index.t -> 'a t -> bool
val bforall : (Index.t * 'a -> bool) -> 'a t -> bool
val vforall : ('a -> bool) -> 'a t -> bool
val iforall : (Index.t -> bool) -> 'a t -> bool
val vexists : ('a -> bool) -> 'a t -> bool
val subset : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
val eqset : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool

val bfold : 'a -> (Index.t * 'b -> 'a -> 'a) -> 'b t -> 'a
val vfold : 'a -> ('b -> 'a -> 'a) -> 'b t -> 'a
val ifold : 'a -> (Index.t -> 'a -> 'a) -> 'b t -> 'a
val bmap : (Index.t * 'a -> 'b) -> 'a t -> 'b t
val vmap : ('a -> 'b) -> 'a t -> 'b t
val imap : (Index.t -> 'b) -> 'a t -> 'b t
val rmap : (Index.t -> Index.t) -> 'a t -> 'a t
val bfilter : (Index.t * 'a -> bool) -> 'a t -> 'a t
val vfilter : ('a -> bool) -> 'a t -> 'a t
val ifilter : (Index.t -> bool) -> 'a t -> 'a t
val remove : Index.t -> 'a t -> 'a t
val subtract : 'a t -> 'b t -> 'a t
val diff : 'a t -> 'a t -> 'a t (* common values [==] *)
val inter : 'a t -> 'a t -> 'a t (* common values [==] *)
val (@@) : 'a t -> 'a t -> 'a t (* labels must be unique *)
val find : Index.t -> 'a t -> 'a
val find_ok : Index.t -> 'a t -> [`OK of 'a|`Error of string]
val bchoose : 'a t -> [`OK of Index.t * 'a|`Error of string]
val vchoose : 'a t -> [`OK of 'a|`Error of string]
val ichoose : 'a t -> [`OK of Index.t|`Error of string]
val append_eq : ('a -> 'a -> bool) ->
  'a t -> 'a t -> [`OK of 'a t|`Error of string]
val bfold1ok :
  'a -> (Index.t * 'b -> 'a -> [`OK of 'a|`Error of 's]) ->
```

```
    'b t -> [`OK of 'a|`Error of 's]
val vfold1ok :
  'a -> ('b -> 'a -> [`OK of 'a|`Error of 's]) ->
    'b t -> [`OK of 'a|`Error of 's]
val ifold1ok :
  'a -> (Index.t -> 'a -> [`OK of 'a|`Error of 's]) ->
    'b t -> [`OK of 'a|`Error of 's]
val bmap1ok : (Index.t * 'a -> [`OK of 'b|`Error of 's]) ->
  'a t -> [`OK of 'b t|`Error of 's]
val vmap1ok : ('a -> [`OK of 'b|`Error of 's]) ->
  'a t -> [`OK of 'b t|`Error of 's]
val imap1ok : (Index.t -> [`OK of 'b|`Error of 's]) ->
  'a t -> [`OK of 'b t|`Error of 's]
```

The OCaml type of indexed lists `IList.t` is a parameterized type, therefore "`cat IList.t`" denotes the type of indexed lists of values of type `cat`, etc. The type of labels (indexes) `Index.t` is not parameterized and is accompanied by a set of operations. There are two views on the data type of indexes, used in different parts of the semantics. The variant `AtIndex.t` provides several constants, for example `AtIndex.atd` (see the next section). The variant `IdIndex.t` is equipped with an equality predicate.

The operations `nil` and `cons` are the constructors of the type of indexed lists `IList.t`. An invariant that no label can index two values on the same indexed list is maintained by all the operations. In particular the operation `cons` throws an exception if the label to be added is already on the list. Whenever we use the operations to define semantics we ensure that no exceptions can be raised.

The operation `is_nil` tests for list emptiness, while `is_in` and `not_in` test whether there is a value indexed by a given label on the list. The universal quantifier `bforall` checks if all the pairs of value and its index on the list satisfy a given predicate. A variant of the quantifier, called `vforall`, checks a predicate that does not depend on the index, while the variant `iforall` considers only indexes. The existential quantifier `vexists` has only one variant, which ignores indexes. The operation `subset`, when supplied with an equality function on values, tests whether the set of indexed values on the first operand list is contained in the set of indexed values on the second. The operation `eqset` checks equality of the sets.

The operation `bfold` is the most general iterator over the indexed lists, while `vfold` and `ifold` consider only the values or the indexes on the list, respectively. The operation `bmap` is a mapping operation, which preserves the indexes on the list, with variants `vmap`, `imap` that construct the values on the outcome list using respectively restricted information from the operand list. Differently than with `bmap`, the operation `rmap` preserves values on the list, while renaming their indexes.

The operation `bfilter` removes all the values that do not satisfy a predicate, while `vfilter` and `ifilter` are its less general variants. The operation `remove` gets rid of an element of the list with the given index, if there is one. The operation `subtract` removes these elements of the first list that have indexes from the second list. The operation `diff` not only subtracts lists but also checks that the values at common indexes are equal (throwing an exception otherwise). Similarly, operation `inter` produces the intersection of two lists, verifying the equality of values at common indexes. The infix operation `@@` concatenates two lists, failing if there are common indexes.

The operations `find` gives a value at a given index. If there is a possibility that there is no such index, the operations `find_ok` should be used instead. The operation `bchoose` produces one element of the given list, unless the list is empty. The variants `vchoose` and `ichoose` give only a value or an index, respectively. The operation `append_eq`, supplied with an equality function, concatenates two lists, checking that the elements at common indexes are equal. The iterators `bfold1ok`, `vfold1ok` and `ifold1ok`, as well as the mapping combinators `bmap1ok`, `vmap1ok` and `imap1ok` accept potentially failing actions and, if the failure occurs, propagate it to the overall result.

3.3 Typical identifiers

Throughout the thesis, in particular in every equation, we adopt a uniform convention for naming identifiers that range over given syntactic and semantic domains. We also adhere to the convention in all the source code of the Dule compiler written in Dule as well as the one written in OCaml. The convention will be presented here, while definitions of the domains of the identifiers are scattered through different parts of the thesis. It is a good idea to browse through the list below when beginning to read trough a longer body of formal notation.

- c and also d, e, a, b stand for categories (objects of LC, *2-LC* and others, see Section 4.1),
- f and also g, h range over functors (for example, morphisms of LC) as well as types (in bare and user languages, see Section 8.3),
- t and also u stand for transformations (for example, 2-morphisms of *2-LC*, see Section 4.2), and for values (in bare and user languages),
- numerical and other subscripts and postfixes are used whenever required,

- l is a prefix for indexed lists, so for example lc stands for indexed list of categories and lt ranges over indexed list of transformations,
- i, j, k range over labels that index components of indexed lists,
- in the context of indexed lists such as lc or lf, subscripted values like c_i or f_j denote elements of these lists, in this case the i-th element of lc and the j-th element of lf, respectively; then we may present lf as $(j : f_j; k : f_k; \ldots)$, where different symbols may take the place of the colon in different kinds of lists,
- Greek letters denote constant distinguished labels and here the notations used in the thesis and in the source code differ: δ, ϵ, ι and κ in mathematical formulae play the role of `AtIndex.atd`, `AtIndex.ate`, `AtIndex.atj` and `AtIndex.atk` in the compiler code, non-Greek constant labels are usually written in quotation marks,
- s and also r range over signatures (objects in module categories, such as W-Dule of Section 8.1),
- m ranges over modules (for example, over the morphisms of L-Dule in Section 5.3),
- p, q and sometimes a range over specifications (of bare, environment-dependent and user module languages in Section 9.2).

Part II

Semantics of Dule

Chapter 4

Base categorical model

We present here the bare minimum of our base categorical model, so that the reader is quickly able to follow the construction of the Dule module system (Chapter 5), which is the central topic of this work. More on our base categorical model — in particular, the complete equational theories of the defined categories — may be found in Chapter 6. In that chapter we also present the model as a programming language by declaring some syntactic sugar for terms and defining the mechanism of evaluation.

The base categorical model will be specified gradually, starting with the notion of labeled cartesian category, then adding the 2-cartesian structure and concluding with a category that has products at all levels. The products are important as they embody the notion of parameterization or dependency, whether on types or on values, which is crucial for small-scale programming as well as for modular programming.

4.1 LC — Labeled Cartesian category

4.1.1 Definition

Some basic mechanisms of a functional programming language may be modeled as simply typed λ-calculus with products [11]. An alternative, attractive way of formalizing the same concept is given by cartesian closed categories (CCC) [98]. As a starting point for a model of Dule programming language we choose a simpler notion of cartesian categories, differing from CCC in that they do not have to be closed under exponential objects. The closeness property is not needed for our module system but will reappear later as an important extension to the rudimentary programming language based on our categorical models.

Cartesian category is a category with all finite products. One may imagine that components of a finite product are labeled (indexed) using consecutive natural numbers. A more general construction, which we use, is the finite labeled product with labels from arbitrary (countable, well-ordered) sets. The labels are assigned without any restrictions, except that no label may occur twice on the same indexed list. The components are identified solely by their labels, and the order in which they appear on indexed lists of operands is unimportant for product operations.

We name and implement the collections of product operands as indexed lists rather than finite maps or dictionaries, because while the order of elements is not meaningful for the semantics, it may be used during compilation to optimize code, generate warnings or decide the meaning of shorthands. In a syntax the elements are always ordered and it turns out to be practical to use the same data structure for the syntax and for its semantics. However, in formal arguments a semantic indexed list should be read as an abstraction class of all indexed lists with the same labeled components.

Formally, labeled product is defined as a limit of a finite labeled collection of objects (as opposed to an ordered collection of anonymous objects in the case of finite products). Another (equivalent) definition, using adjunctions, is given in Section 7.1.3. Any category with all labeled products will be called "labeled cartesian category" (LC). Such category, as a mathematical structure, behaves much the same as a cartesian category. In fact, every cartesian category is also labeled cartesian, because any labeled product can be uniquely represented as a finite product by sorting its labels (obviously, every LC is cartesian, too). This observation provides us with plenty of examples of LC. Another example, the initial model among the class of all LC considered as partial algebras, is known to exist, because of the simple pattern of dependency between the definedness of operations in its various sorts [126]. The pattern is similarly simple in every other partial algebra we define in our thesis, and we will quietly assume the existence of initial models from now on.

The language of LC is much more convenient than the language of standard cartesian categories for tasks such as defining programming constructs or describing compiler transformations. The main asset is that it relieves one of the need to use large numbers of projections to change the order of components or just to access them, which is a drawback of combinator languages based on finite and especially binary products.

4.1.2 Language

Syntax

We present here a syntax of a language we will use to denote objects and morphisms of LC. There are two sorts of terms: `cat`-terms and `funct`-terms. This is not readily visible, until we provide a detailed account in Section 4.3.1, but the `cat`-terms are meant to denote kinds and `funct`-terms to denote types of a programming language. A third syntactic domain of `trans`-terms will be introduced in the next section and will correspond to values of a programming language.

As explained in Section 3.2, the phrase "`cat IList.t`" below denotes an indexed list of `cat`-terms, "`funct IList.t`" denotes an indexed list of `funct`-terms, "`IdIndex.t`" is the set of labels and the two OCaml datatypes define the syntax of LC (or the algebraic signature of LC seen as an algebra).

```
type cat =
  | C_PP of cat IList.t
  | C_BB
type funct =
  | F_ID of cat
  | F_COMP of funct * funct
  | F_PR of cat IList.t * IdIndex.t
  | F_RECORD of cat * funct IList.t
```

But this is a raw syntax only. A raw `funct`-term belongs to the language of LC, if it is type-correct, as described alongside the semantics below. All the type assignments defined in this chapter are very simple, moreover the terms carry a lot of types (for example the `funct`-term `F_ID`(c) carries in itself the `cat`-term c, which is both its domain and codomain), and therefore the type-checking is always easily decidable.

Note that there are no variables in the syntax, since references to previously defined items in our programming language are modeled using projections. The (implicitly typed) meta-variables used in equations, etc., do not interfere with the distinction between raw syntax and final (type-checked) language. The entities to which meta-variables refer are always assumed to be defined and typed.

Semantics

The objects of LC will be called categories and the morphisms will be called functors. The names are forged by analogy to **Cat** — the category of all (small) categories. This analogy will be made clearer once 2-morphisms, called transformations, appear in the next sections.

Let us fix an arbitrary LC. We start by assigning semantics in the LC to `cat`-terms, which are always type-correct and denote categories (objects of the LC).

- `C_PP`(lc) is a (chosen) labeled product of an indexed list of categories lc,
- `C_BB` is a distinguished constant, called the base category.

The category `C_BB` will represent the kind of basic types. This is the most important kind, as all user datatypes live there (see for instance Section 8.3.1). Note, however, that from the formal point of view, the requirement that there is a distinguished constant in LC has only syntactic consequences, up to Section 4.3. No properties are here associated with the object `C_BB` and the set of objects is guaranteed to be non-empty anyway, because LC does have a terminal object `C_PP(nil)`, where `nil` is the empty indexed list.

Every type-correct `funct`-term has a well defined semantics and denotes a morphism of the LC — a functor. While presenting the semantics we accompany each `funct`-term with `cat`-terms denoting its domain and codomain. Then we provide the description of the semantics that may explicitly or implicitly impose some typing side conditions (for example when a category meta-variable occurs twice in the description). If any of the typing side conditions is not met, the term is said to have no semantics; its value is undefined. Checking a single typing side condition does not involve the semantics of `funct`-terms, but depends on the semantics of `cat`-terms (for example to decide the equality of `cat`-terms representing domains of operands of `F_RECORD`). For the sake of readability we join the typing and the semantics of `funct`-terms in the same presentation, but see Appendix A.1.3 for a separate and purely syntactic presentation of the typing rules.

- `F_ID`$(c) : c \to c$
  is the identity functor on category c,
- `F_COMP`$(f, g) : c \to e$
  is the composition of functors $f : c \to d$ and $g : d \to e$,
- `F_PR`(lc, i) : `C_PP`$(lc) \to c_i$
  is the projection at label i, where the i-th element of lc (called c_i) is required to be present,
- `F_RECORD`$(c, lf) : c \to$ `C_PP`(le)
  is the record (labeled tuple, universal morphism of the product) of an indexed list of functors lf, where the elements of lf are $f_i : c \to e_i$ and the indexed list of categories le contains all and only labels from the list of functors lf.

Example. An example of a correct term is `F_RECORD(C_BB, nil)`. This term represents the unique morphism from `C_BB` to the terminal object. On the other hand `F_PR(nil,`i`)` does not belong to the language (and hence has no semantics) as the label i is not present on the empty indexed list `nil`. In the initial LC the term

```
F_COMP(F_ID(C_BB), F_ID(C_PP(nil)))
```

has no semantics, while in the trivial LC (one object, one morphism) the term is type-correct and denotes the only morphism of the category.

4.2 *2-LC* — 2-Labeled Cartesian category

4.2.1 Definition based on *LC*

In our semantic model of a programming language we want to talk about both values and types. We do not model higher-order polymorphism, only simple parameterization of a value by some types, where the types are treated as “locally” abstract types within the value. In simply typed λ-calculus values depending on types are expressed using type variables and (in Church style notation) type instantiation. We capture the notion categorically by demanding LC structures for both values and types joined inside a 2-categorical framework. In this section we lay out the 2-categorical structure and the “outer” cartesian structure on values, corresponding to the cartesian structure on types.

Each 2-category [82] comprises of two ordinary categories and a family of categories $C(c, e)$. The first of the two categories is the underlying category, that is, the category of objects and 1-morphisms with the composition of 1-morphisms. Then, for each pair of objects c, e, there is a category $C(c, e)$ of all 1-morphisms with source c and target e as objects and all 2-morphisms between them as morphisms with their vertical composition. Horizontal composition yields the category of objects and 2-morphisms, with the identity on object c equal to the 2-identity on the 1-identity on object c.

In category theory, a functor is said to preserve products, when it maps product objects to product objects and projection morphisms to projection morphisms. In a setting where the product operation is a part of the algebraic structure, there is a possibility to speak about the distinguished products. We have distinguished products in LC with its language defined in the previous section, because we have set `C_PP`(lc) to denote a specific object and not a class of isomorphic objects. A functor is said to preserve products “on the nose” if it maps distinguished product objects to distinguished product objects and distinguished projection morphisms to distinguished projection morphisms.

A *2-LC* (pronounced “2-labeled cartesian category”, compare with [29]) is a 2-category such that:

- its underlying category, that is the category of objects and 1-morphisms is an LC,
- the category of objects and 2-morphisms with horizontal composition is an LC (the products in this category are called 2-products in [29]),
- these two LC structures are compatible, that is the 2-identity operation determines a functor that preserves products “on the nose” from the underlying category to the category of objects and 2-morphisms. The categories $C(c, e)$ do not have to be labeled cartesian, yet.

The essence of the compatibility condition may be captured by the following equation:

$$\texttt{T\_id}(\texttt{F\_PR}(lc, i)) \quad = \quad \texttt{T\_PR}(lc, i)$$

where `T_id` is the 2-identity operation and `T_PR` is the distinguished projection in the category of objects and 2-morphisms. Note that the type-correctness of the equation implies that the labeled product objects in the two categories coincide. See Section 6.1.4 for a complete equational theory of 2-labeled cartesian categories.

An example of *2-LC* is the category of all categories **Cat**. The labeled cartesian structure in both the underlying category and the category of objects and 2-morphisms should be defined point-wise using the same arrangement of labels, which ensures that they are compatible. A $C(c, e)$ in this 2-category is just the functor category from c to e (not to be confused with the underlying category from the definition of *2-LC*, where functors are morphisms, not objects).

An example of category with incompatible products is again **Cat** with point-wise definition of product operations, but with different choices of the product of categories for the underlying category and for the category of objects and 2-morphisms. For example, the labeled tuples constituting product objects in the category of objects and 2-morphisms may be sorted in reverse order than the tuples in the underlying category. Then the 2-identity functor preserves products, but not "on the nose".

A more interesting example would be a category with both the *LC* structures, but where the 2-identity functor does not even preserve products. An easy argument shows that such a category may not be a *2-LC* under any choice of products, so **Cat** is not a good example for this situation.

4.2.2 Language

Syntax

The syntax of *2-LC* is an extension of the syntax of *LC*. The `cat`-terms and the `funct`-terms are the same. The `trans`-terms are entirely new, although they resemble the `funct`-terms, which is understandable considering that the cartesian structure should be expressible in both.

The syntax of raw (not type-checked) *2-LC* terms looks as follows.

```
type cat =
  | C_PP of cat IList.t
  | C_BB
type funct =
  | F_ID of cat
```

```
    | F_COMP of funct * funct
    | F_PR of cat IList.t * IdIndex.t
    | F_RECORD of cat * funct IList.t
  type trans =
    | T_ID of cat
    | T_COMP of trans * trans
    | T_PR of cat IList.t * IdIndex.t
    | T_RECORD of cat * trans IList.t
    | T_FT of funct * trans
    | T_TF of trans * funct
    | T_id of funct
    | T_comp of trans * trans
```

Semantics

Let us fix an arbitrary *2-LC*. The `cat`-terms and `funct`-terms have the same meaning as in *LC*. They denote the objects (categories) and morphisms (functors) in the underlying category of the *2-LC*. The semantics of `trans`-terms, which denote 2-morphisms (transformations), is the following. As before, the type-checking of transformations is independent of their semantics, but depends on the equality of categories and functors.

- `T_ID`(c) : `F_ID`$(c) \to$ `F_ID`(c)
  is the identity in the category of objects and 2-morphisms,

- `T_COMP`(t_1, t_2) : `F_COMP`$(f_1, f_2) \to$ `F_COMP`(h_1, h_2)
  is the composition in the category of objects and 2-morphisms (horizontal composition) of $t_1 : f_1 \to h_1$ and $t_2 : f_2 \to h_2$, where $f_1, h_1 : c \to d$ and $f_2, h_2 : d \to e$,

- `T_PR`(lc, i) : `F_PR`$(lc, i) \to$ `F_PR`(lc, i)
  is the projection in the category of objects and 2-morphisms,

- `T_RECORD`(c, lt) : `F_RECORD`$(c, lf) \to$ `F_RECORD`(c, lh)
  is the record (labeled tuple) in the category of objects and 2-morphisms of an indexed list of transformations lt, where $t_i : f_i \to h_i$, $f_i, h_i : c \to e_i$ and lf, lh have the same labels (indexes) as lt,

- `T_FT`(f_1, t_2) : `F_COMP`$(f_1, f_2) \to$ `F_COMP`(f_1, h_2)
  is the multiplication of the transformation $t_2 : f_2 \to h_2$ by the functor f_1 from the left, where $f_1 : c \to d$ and $f_2, h_2 : d \to e$,

- `T_TF`(t_1, f_2) : `F_COMP`$(f_1, f_2) \to$ `F_COMP`(h_1, f_2)
  is the multiplication of the transformation $t_1 : f_1 \to h_1$ by the functor f_2 from the right, where $f_1, h_1 : c \to d$ and $f_2 : d \to e$,

- `T_id`$(g) : g \to g$
  is the 2-identity on $g : c \to e$, that is the identity on object g in the category $C(c, e)$,
- `T_comp`$(t, u) : f \to h$
  is the composition in $C(c, e)$ (vertical composition) of $t : f \to g$ and $u : g \to h$, where $f, g, h : c \to e$.

The multiplications by a functor may be expressed in terms of horizontal composition as follows.

$$\begin{aligned}\texttt{T\_FT}(f_1, t_2) &= \texttt{T\_COMP}(\texttt{T\_id}(f_1), t_2)\\ \texttt{T\_TF}(t_1, f_2) &= \texttt{T\_COMP}(t_1, \texttt{T\_id}(f_2))\end{aligned}$$

However, the additional term constructors provide for shorter notation, and the separation of the three notions will be convenient for term rewriting.

Example. When interpreted in **Cat**, the `cat`-terms denote categories, for example `C_PP(nil)` is the terminal, trivial (one object and one morphism) category. The `funct`-terms denote functors, for example `F_RECORD`(c, lf) acts on an object of the category c by applying each of the functors in lf and then tupling the results. The `trans`-terms denote natural transformations. For example: `T_ID(C_BB)`, given an object of the category denoted by `C_BB`, produces the identity morphism on this object. A similarly looking, but in general different transformation is `T_id`(f), where f : `C_BB` $\to$ `C_BB` is a functor. The transformation `T_id`(f) given an object produces the identity morphism on the result of applying f to the object. Therefore `T_ID(C_BB)` is equal to `T_id(F_ID(C_BB))`, which is also true in any *2-LC*, as stated in one of the next chapters.

In the initial *2-LC*, as well as in the *2-LC* based on **Cat** where `C_BB` is **Set** (the category of sets and functions), the term

```
T_comp(T_ID(C_BB), T_ID(C_PP(nil)))
```

has no semantics. In the *2-LC* based on **Cat** where `C_BB` is the trivial category, as well as in the trivial *2-LC*, the term is type-correct.

4.3 *2-LC-lc* — *2-LC* with labeled products

4.3.1 Definition based on *2-LC*

We proceed with specifying the semantic model of a programming language. Now that we have both the cartesian structure to be used for modeling types dependent on types and the cartesian structure for values dependent on types we may install the "inner" cartesian structure to model values referring to values inside our framework.

A *2-LC-lc* (pronounced "2-labeled cartesian category with labeled products") is a *2-LC* such that:

- for each object c, the category $C(c, \texttt{C\_BB})$ is an LC.

Remark. Notice that this induces the cartesian structure in all $C(c, e)$, where e is a category expressible as a `cat`-term. The product operations in $C(c, \texttt{C\_PP}(le))$ can be recovered as records (`T_RECORD`, not `T_record`) of product operations from each of $C(c, e_i)$, where e_i is the i-th element of the indexed list le (note that $C(c, \texttt{C\_PP}(\texttt{nil}))$ has the trivial cartesian structure, since `C_PP(nil)` is terminal). This suggests an extension of *2-LC-lc* that will have the same raw syntax and semantics, but a wider class of type-correct terms. The extension would result in more concise notation and faster rewriting of the complicated product terms.

A *2-LC-lc* with explicit compound products is a *2-LC* such that:

- for each pair of objects c, e, if e is expressible as a `cat`-term then the category $C(c, e)$ is an LC,
- the two kinds of labeled products on 2-morphisms satisfy product interchange law, that is the product operations in each $C(c, \texttt{C\_PP}(le))$ are chosen to be the records (`T_RECORD`) of the product operations from $C(c, e_i)$.

To see the equational cast of the product interchange law consult Section 6.1.5. The explicit compound products are not needed for our implemented variant of a module system. Only some specific future extensions will benefit from the generalized notation, so we'll admit it only as far as it sheds some light on the plain *2-LC-lc*.

Examples of *2-LC-lc* include **Cat**, with **Set** as the category `C_BB`. This implies that any type system with a semantics in arbitrary *2-LC-lc* has, among others, a set-theoretic semantics. The full subcategory of **Cat** consisting of labeled cartesian categories is a *2-LC-lc* with explicit compound products, regardless of the choice of `C_BB`. There are also many not so abstract settings. Any (functional subset of a) formally or informally defined polymorphic functional programming language, such Standard ML or Haskell, determines a *2-LC-lc*. The exceptions would be languages that already at the level of language definitions have explicit and possibly deferred substitutions. Yet, most of the descriptions of functional subsets of programming languages treat substitution as a meta mechanism and thus respect all the mathematical laws we depend on.

The *2-LC-lc* built from a polymorphic programming language is the following. The 2-morphisms are the values of the programming language, the 1-morphisms are the types while the objects are the kinds and are usually trivial. The vertical composition is the value substitution and the multiplication by a functor from the left is the type instantiation. The multiplication by a functor from the right can be recovered using higher-order functions if present in the language or the `map` combinator if the language is (co)induction oriented. Products are given by record types and by tupling of kinds. In the same way any model of simply typed λ-calculus with type variables gives rise to a *2-LC-lc*.

On the other hand, compilers or interpreters of functional programming languages usually do not have any natural *2-LC-lc* structure as they model value substitution in strange ways. For example, by using environments they may hold off any intermediate computations until the value is of a ground type (such as `int`). In this way most of the time they honor almost no equalities, but to the user they try to appear as conforming to a simple semantic model, by readily printing values such as integers, while refusing to print any values of a function type. However, when we quotient the set of states of an interpreter by observational indistinguishability [99], again we get a *2-LC-lc*.

4.3.2 Language

Syntax

To the grammar of *2-LC* we append one `funct`-term constructor (`F_pp`) and three `trans`-term constructors. The raw syntax of the whole *2-LC-lc* language looks as follows.

```
type cat =
  | C_PP of cat IList.t
  | C_BB
type funct =
  | F_ID of cat
  | F_COMP of funct * funct
  | F_PR of cat IList.t * IdIndex.t
  | F_RECORD of cat * funct IList.t
  | F_pp of cat * funct IList.t
type trans =
  | T_ID of cat
  | T_COMP of trans * trans
  | T_PR of cat IList.t * IdIndex.t
  | T_RECORD of cat * trans IList.t
  | T_FT of funct * trans
  | T_TF of trans * funct
  | T_id of funct
  | T_comp of trans * trans
  | T_pp of cat * trans IList.t
  | T_pr of funct IList.t * IdIndex.t
  | T_record of funct * trans IList.t
```

Semantics

Let us fix an arbitrary *2-LC-lc*. Here we will describe the semantics of new constructs with respect to *2-LC*, only. We present the semantics in a form that, when read without restrictions, matches the *2-LC-lc* with explicit compound products. Knowing that our *2-LC-lc* may have no explicit compound products we restrict the class of type-correct terms of the new forms to contain only those of target `C_BB`. In other words, we require that the category e below is equal to (the semantics of) `C_BB`.

- $\texttt{F\_pp}(c, lf) : c \to e$
  is a (chosen) labeled product of an indexed list of functors lf (in the category $C(c, e)$, where functors are objects), where $f_i : c \to e$,

- $\texttt{T\_pp}(c, lt) : \texttt{F\_pp}(c, lf) \to \texttt{F\_pp}(c, lh)$
  is the action of the labeled product functor on morphisms lt of $C(c, e)$, where lt is an indexed list of transformations such that $t_i : f_i \to h_i$, $f_i, h_i : c \to e$ and lf, lh have the same labels as lt,

- $\texttt{T\_pr}(lf, i) : \texttt{F\_pp}(c, lf) \to f_i$
  is the projection in $C(c, e)$, where $f_k : c \to e$,

- $\texttt{T\_record}(f, lt) : f \to \texttt{F\_pp}(c, lh)$
  is the record (labeled tuple) in $C(c, e)$ of an indexed list of transformations lt, where $t_i : f \to h_i$, $f, h_i : c \to e$ and lh has the same labels as lt.

The constructor `F_pp` may be thought of as the action of the product functor on objects and `T_pp` as the action on morphisms. In fact, `T_pp` is expressible as a multiplication by the labeled product functor from the right as follows. (See Section 6.1.5 for further discussion.)

$$\texttt{T\_pp}(c, lt) \quad = \quad \texttt{T\_TF}(\texttt{T\_RECORD}(c, lt), \texttt{F\_pp}(\texttt{C\_PP}(le),\ i : \texttt{F\_PR}(le, i);\ \ldots))$$

Example. Let us look closer at the *2-LC-lc* built from **Cat** with `C_BB` denoting the category **Set**. Let us focus on the category $B = C(\texttt{C\_PP}(\texttt{nil}), \texttt{C\_BB})$ that represents **Set**. Then if lf is a labeled list of objects of B, and so representations of sets, then $\texttt{F\_pp}(\texttt{C\_PP}(\texttt{nil}), lf)$ is the representation of the cartesian product of the sets. If lt is a labeled list of morphisms of B, and so representations of functions, then $\texttt{T\_pp}(\texttt{C\_PP}(\texttt{nil}), lt)$ represents the function that given a tuple of elements, applies each of lt to the corresponding element of the tuple, and then tuples the results. If, moreover, lt represents functions from a common domain f then $\texttt{T\_record}(f, lt)$ is a representation of the function that given an element of the set f, applies each of the functions lt to it and then tuples the results.

4.4 Conclusion

We have defined the base categorical model that underlies every model of a programming language or a module system we develop in our thesis. The last of the defined categories, *2-LC-lc*, has products at all levels. The products model the mechanism of a programming language identifier or variable (type variable and value variable). Having products at all levels enables us to capture the notions of dependency of programming language types on types, values on types and values on values. This connection will be made evident in Section 8.3.

Even more importantly, the products in the core language are crucial for our module system. We want a hierarchy of modules to capture the dependency relationship of program fragments. Moreover, due to abstraction, the dependency relation cannot be expressed in a shallow way, such as by compilation order or file inclusion. We need modules parameterized on other modules and this, even in the weak categorical form we promote, requires the comprehensive product machinery provided by *2-LC-lc*.

The languages of categories defined in this chapter, as well as some of their extensions and variants, will be later called internal core languages. The languages are internal in that their terms are not intended to be written directly by the programmer, but rather generated by the compiler. The languages do not include any module operations and hence they are named core languages. A summary of the complete main variant of the internal core language of Dule is given in Appendix A.1. In particular, there is a separate, more conventional presentation of the typing rules for terms of the internal core language.

Semantic combinators

We need to present the operations of our categorical model as semantic combinators, to use them as building blocks in construction of other models. Let us fix an arbitrary *2-LC-lc*. The algebraic signature of *2-LC-lc* seen as an algebra (see Section 3.2) has three sorts named `cat`, `funct` and `trans`. The same three names are also used to denote the corresponding OCaml datatypes or the sets of terms of corresponding sorts. To differentiate the syntactic notions from the semantic ones, let us call the carriers of the fixed *2-LC-lc* `Cat.t`, `Funct.t` and `Trans.t`, respectively. The names are taken from the module structure of the Dule compiler.

We present the profiles of semantic combinators (see Section 3.2) corresponding to the term constructors of *2-LC-lc*. Usage of these combinators in formal definitions (like the definition of the operations of our module system) results in semantics in an arbitrary fixed *2-LC-lc*. The same definitions, but using term constructors, could be misinterpreted as translations, that is semantics into the term algebra of the language of *2-LC-lc*. Not every *2-LC-lc* has carriers constructed from terms and even then the operations can be more complex than just term constructors. For example, in the Dule compiler the operations reduce terms to some normal forms. Moreover, the carrier of the *2-LC-lc* in the Dule compiler is represented by an abstract datatype, so using term constructors instead of the semantic combinators provided with the datatype would lead to typing errors.

Although we present the OCaml typing of the semantic combinators, we will not provide their OCaml definitions. Their general meaning in any *2-LC-lc* (in particular in the implemented one) is completely determined by the semantics of term constructors given in the previous section. We do not want to go into the implementation details (which can be found in the source files of the Dule compiler, see `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/core_back.ml`), so we only admit that the terms of the language of *2-LC-lc* (sufficiently adorned with stamps for efficient storage in memory and considered up to the equivalence induced by the theory of *2-LC-lc*) are indeed an important part of the construction of the carriers of the implemented *2-LC-lc*. The OCaml definitions of the combinators represent and implement the reduction of the terms according to the rewrite system for *2-LC-lc* that is developed later in the thesis, starting in Section 6.1.

There are two combinators corresponding to the syntactic domain `cat`.

```
val c_PP : Cat.t IList.t -> Cat.t
val c_BB : Cat.t
```

As we see, the names of the combinators are the same as the names of the term constructors but with the first letter turned lowercase. There are also semantic combinators for all possible forms of `funct`-terms.

```
val f_ID : Cat.t -> Funct.t
val f_COMP : Funct.t -> Funct.t -> Funct.t
val f_PR : Cat.t IList.t -> IdIndex.t -> Funct.t
val f_RECORD : Cat.t -> Funct.t IList.t -> Funct.t
val f_pp : Cat.t -> Funct.t IList.t -> Funct.t
```

As expected, the multi-operand combinators are in the curried form. The syntactic domain of `trans`-terms has a full set of semantic combinators, as well.

```
val t_ID : Cat.t -> Trans.t
val t_COMP : Trans.t -> Trans.t -> Trans.t
val t_PR : Cat.t IList.t -> IdIndex.t -> Trans.t
val t_RECORD : Cat.t -> Trans.t IList.t -> Trans.t
val t_FT : Funct.t -> Trans.t -> Trans.t
val t_TF : Trans.t -> Funct.t -> Trans.t
val t_id : Funct.t -> Trans.t
val t_comp : Trans.t -> Trans.t -> Trans.t
val t_pp : Cat.t -> Trans.t IList.t -> Trans.t
val t_pr : Funct.t IList.t -> IdIndex.t -> Trans.t
val t_record : Funct.t -> Trans.t IList.t -> Trans.t
```

There are no partial operations in *2-LC-lc*, therefore the result type of each semantic combinator is just `Trans.t`, `Funct.t` or `Cat.t`.

Chapter 5

Semantics of the module system

The notion of *2-LC-lc* developed in Chapter 4 serves as a model of a core programming language and as the framework for the construction of Simple Category of Modules (SCM). SCM is the model of our module system, allowing us to capture a variety of mechanisms for modularization of programs written in any core language based on *2-LC-lc*.

After constructing the SCM we will develop two module systems, that is, programming-oriented languages for SCM. First, we will define the semantics of a very small, yet expressive language W-Dule into SCM, using all the features of *2-LC-lc*. Then we will enrich the language with three important operations for putting modules together, obtaining a comprehensive module system L-Dule. This second time we refer to *2-LC-lc* only indirectly, using essentially only W-Dule as a basis for our definition.

The definition of the semantics is written in a functional language. A complete version written in Dule itself is described in Appendix C.2 and available online. Here we will base our presentation on an OCaml version, that is a part of a working Dule compiler. We will strive to make the presentation in this chapter self-contained, assuming knowledge of Section 3.2, which describes how to read the OCaml notation. We will either describe or cite every auxiliary function that appears in the code of the semantic definitions.

The language W-Dule, as well as its basic extensions described in this chapter and in Section 9.1, are called internal module languages. These languages are not intended to be used for writing Dule programs by hand. However, each user language program can be easily translated to the internal language in a manner described in Section 9.2. A summary of the implemented variant of the internal module language is given in Appendix A.2.

5.1 Simple Category of Modules

The construction of Simple Category of Modules (SCM) should look quite intuitive to a programmer with a minimal categorical background. The objects of this category are module signatures and the morphisms are modules themselves. The domain of a module is the signature of its parameters and the codomain is the result signature of the module. Finally, the identity morphism of SCM is the identity module and the composition is the operation of supplying implementation of parameters to a module.

Now we will proceed to the formal definition of the modular programming notions used above, starting with signatures. For the rest of the chapter let us fix an arbitrary base categorical model *2-LC-lc*. Objects and morphisms of the SCM will be built from the morphisms of the fixed *2-LC-lc*. Since, in particular, *2-LC-lc* can be based on **Cat** with `C_BB` equal to **Set**, SCM has a set-theoretic semantics and the signatures and modules can be thought of as (quite complex but not higher-order) functions.

5.1.1 Objects and morphisms

Objects of SCM are called signatures and are defined as follows.

Definition 5.1.1. *A 2-LC-lc functor* `f` *is a signature if it can be presented as* `F_pp(C_PP(lc),lf)`*, where* `lc` *is an indexed list of categories and* `lf` *is an indexed list of functors. The indexed list* `lc` *is called the type part of (the signature represented by)* `f`*, while the indexed list* `lf` *is called the value part of* `f`*.*

The OCaml code of the semantics of our module system expresses as assertions a number of properties of the framework. From now on, while defining formally the objects (signatures), morphisms (modules), domains and codomains of SCM we will cite the assertions straight out of the source code (see file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/mod_back.ml`), hence the font and formatting.

We cite the condition defining signatures from a comment in the OCaml module of signatures `Sign`. Let `f` be a functor, then

```
the application [f2s f] results in a signature
iff [f] is of the form [f_pp (c_PP lc) lf]
```

The auxiliary partial function `f2s` : `Funct.t` $\to$ `Sign.t` mentioned above is identity on functors that are signatures (according to Definition 5.1.1) and is undefined elsewhere.

We see that a signature is a labeled product functor with a labeled product source. This simple form of signatures resembles the syntax of simple Standard

ML or OCaml signatures (or the syntax of Dule base signatures, to be defined later), where `lc` would correspond to names of types and `lf` to types of values in module signature. In general, beside the names of types, `lc` will usually contain names and kinds of parameter modules with nested names of types. Also, when the signature is a product object in SCM, `lf` is an indexed list of types of value parts of modules. Still, the *2-LC-lc* labeled product operations are enough to capture the diversity.

The code of the auxiliary function `f2s` is given below. The defining properties of a signature are expressed in OCaml and tested, as an OCaml assertion, every time a functor is brought to the role of a signature in the source code. The expression `src f` denotes the source category of functor `f`. The auxiliary function `unpp_ok` chooses (in an arbitrary but deterministic way) the indexed list of components of the chosen product functor, so that `f_pp (src f) (unpp_ok f) = f` (note that this equality is exact, not just up to isomorphism). If the SCM is built upon the initial *2-LC-lc*, such lists are uniquely determined. Similarly, auxiliary function `unPPok` chooses the indexed list of components of the chosen product category operation. If the operands cannot be presented as the chosen products, the two functions produce error messages. As explained in Section 3.2, the auxiliary functions may appear prefixed with module names. For example, the function `src` is here prefixed with the module name `SrcFCore`, while `unPPok` and `unpp_ok` are taken from the modules `Cat` and `Funct`, respectively, implementing two of the carriers of the *2-LC-lc*.

```
let f2s f =
  assert
    (match Funct.unpp_ok f with
    |‘OK lf ->
        (match Cat.unPPok (SrcFCore.src f) with
        |‘OK lc -> true
        |‘Error er -> false)
    |‘Error er -> false
    );
  f
```

The OCaml type representing objects of SCM is called `Sign.t`. Profiles of the operations contained in the module `Sign` are the following:

```
val f2s : Funct.t -> t
val s2f : t -> Funct.t
val type_part : t -> Cat.t IList.t
val value_part : t -> Funct.t IList.t
```

The function `Sign.f2s` has an inverse called `Sign.s2f`, which is the inclusion of the set of signatures into the set of functors. The function `Sign.s2f` facilitates applying *2-LC-lc* operations to representations of signatures. However, in the definition below, as well as occasionally throughout the chapter, we will omit occurrences of `Sign.s2f`, to shorten notation. The two operations for accessing signature parts are called `type_part` and `value_part`; their choice of product components is the same as in `Funct.unpp_ok` and `Cat.unPPok`, respectively.

Definition 5.1.2. *A module is a triple of a functor, a transformation and a signature satisfying certain conditions that we cite below directly from the code. The auxiliary partial function* `pack` *takes a triple into the set of modules. Let* `f` *be a functor,* `t` *be a transformation and* `s` *be a signature. Then we state that*

```
the application [pack (f, t, s)] results in a module
iff there is a signature [r] such that
1. f : src r -> src s
2. t : r -> f_COMP f s
```

The (uniquely determined) signature `r` *is the categorical domain of the above module seen as morphism of an SCM and the signature* `s` *is the categorical codomain (also uniquely determined, because given in the triple). The functor* `f` *is called the type part of the module and* `t` *is called the value part.*

We will sometimes use the notation `m` : `r` → `s` to mark the categorical domain and the categorical codomain of module `m` in SCM. Concrete examples of triples constituting modules are given in the proof of a simple lemma below. Less trivial examples are used as illustrations for module composition in the next section.

The auxiliary function `pack` is defined as follows. The assumptions about a module are checked every time a module is constructed from its type and value parts, with a given codomain. The expression `dom t` denotes the domain functor and `cod t` the codomain functor of a transformation `t`. The predicates `EqFCat.eq` and `EqFFunct.eq` test the equality of categories and functors, respectively. The expression `trg f` denotes the target category of a functor `f`.

```
let pack (f, t, s) =
  assert
    (let g = DomFCore.dom t in
    let r = Sign.f2s g in
    let h = Sign.s2f s in
    EqFCat.eq (SrcFCore.src f) (SrcFCore.src g) &&
    EqFCat.eq (SrcFCore.trg f) (SrcFCore.src h) &&
    EqFFunct.eq (DomFCore.cod t) (Funct.f_COMP f h)
    );
  (f, t, s)
```

The auxiliary function `pack` resides in the OCaml module `Dule`. The OCaml type representing morphisms of SCM (i.e., modules) is called `Dule.t`. In addition to the constructor function `pack` there are also four operations for retrieving information from a module. The names and profiles of the operations should suffice to guess their role, according to Definition 5.1.2.

```
val pack : Funct.t * Trans.t * Sign.t -> t
val domain : t -> Sign.t
val codomain : t -> Sign.t
val type_part : t -> Funct.t
val value_part : t -> Trans.t
```

Lemma 5.1.3. *Categorical domain of a module is uniquely determined by the type and value parts of a module, while categorical codomain is not. In other words: modules with the same type parts and value parts cannot differ in the domain signature and can differ in the codomain signature.*

Proof. Let `t` be a *2-LC-lc* transformation. By Definition 5.1.2, all modules with value part `t` have the same domain, which is

```
Sign.f2s (DomFCore.dom t)
```

This property becomes understandable once we recall that our *2-LC-lc* categorical combinators are fully typed and consequently carry the type of their environment inside them. Therefore the value part of a module, which is such a combinator, cannot be embedded in modules with less, or even more parameters than listed in the carried environment type. Otherwise, its domain functor would not agree with the domain signature of the module.

On the other hand, the module's codomain relates to the codomains of values in a more flexible way. Let us consider the following module. For the sake of readability we first present the module in a concrete syntax (this is the syntax of the language W-Dule, to be defined later in this chapter). The curly braces denote (trivial) products and records.

```
:: {} -> sig
           type u
           value triv : {}
         end
struct
  type u = {}
  value triv = {}
end
```

According to the semantics of W-Dule base signatures, defined precisely in Section 5.2.3, the module expression above represents the module:

```
pack (F_RECORD(C_PP(nil), "u" : F_pp(C_PP(nil), nil)),
      T_record(mdom, "triv" = T_record(mdom, nil)),
      f2s (mcod))
```

where the functor `F_pp(C_PP(nil), nil)` is abbreviated to `mdom` and `mcod` is the functor

```
F_pp(C_PP("u" : C_BB), "triv" : F_pp(C_PP("u" : C_BB), nil))
```

The module is written using the auxiliary functions `pack`, `f2s` and the abstract syntax of *2-LC-lc*. Empty indexed lists are denoted by `nil` and non-empty by sequences of pairs separated by semicolons, as exhaustively described in Section 6.1.1.

The categorical domain of the above module is represented by the functor `mdom` and the categorical codomain by `mcod`. We see that condition 2 of Definition 5.1.2 holds, because the composition of the type part and codomain signature equals to the codomain of the value part:

```
F_RECORD(C_PP(nil), "u" : F_pp(C_PP(nil), nil))
. F_pp(C_PP("u" : C_BB),
    "triv" : F_pp(C_PP("u" : C_BB), nil))
= F_pp(C_PP(nil), "triv" : F_pp(C_PP(nil), nil))
```

However, there exists another module with the same type and value parts but with different codomain signature:

```
:: {} -> sig
           type u
           value triv : u (* was "{}" *)
         end
struct
  type u = {}
  value triv = {}
end
```

The codomain of this module can be represented by a functor `mcod` of the following form:

```
F_pp(C_PP("u" : C_BB), "triv" : F_PR("u" : C_BB, "u"))
```

Condition 2 of Definition 5.1.2 holds here as well:

```
F_RECORD(C_PP(nil), "u" : F_pp(C_PP(nil), nil))
. F_pp(C_PP("u" : C_BB), "triv" : F_PR("u" : C_BB, "u"))
= F_pp(C_PP(nil), "triv" : F_pp(C_PP(nil), nil))
```

Therefore, we see that codomains of modules are not uniquely determined by type and value parts. □

Remark. Modules could be represented in OCaml just as `trans`-terms. One could encode both the type part and the value part of a module as a single transformation; the codomain signature part could be ignored. Such version could possibly be more efficient, especially taking into account memory usage. However, representing modules as triples of OCaml values is safer due to the continuous checking of the assertion related to the module codomain. Moreover, the link between theoretical model and implementation is then straightforward.

Working without the codomains of modules would require some extra care. In particular, a strict account of domains and codomains is necessary during specification reconstruction (see Section 10.4) and signature checking (see Appendix A.2.3). However, after the modules are type-checked in some way, most of the domains and codomains of modules could be forgotten. We could assume that the module operands of the combinators have been type-checked and therefore always have implicitly assigned unique categorical codomain signatures.

In the semantic definitions of module operations the `trans`-term approach would free us from explicitly supplying the codomain as part of the result of every module semantic combinator. Most of the time the codomain would not have to be passed around, either. Nevertheless, we would be forced to introduce some additional operands for the combinators to supply codomain signatures, where necessary. In particular, all the record-like module operations would require an additional indexed list of codomains of their module operands. In the current implementation the list is still needed, as well, if only to obtain the codomain signature of the overall result. However, the list is now successfully computed on the spot, thanks to the codomains of the individual modules being readily available.

5.1.2 Identity and composition

Let us look at the code of combinators `m_Id` and `m_Comp` defining the identity and composition of SCM. Some of the auxiliary functions used in the definition appear prefixed with module names, as anticipated in Section 3.2. For example, the function `pack`, which we have described as residing in the module called `Dule`, is written here in this way. The assertion "`s1 = r2`" below comes from the usual categorical assumptions on the domains and codomains of composed morphisms.

```
let m_Id s = (* : s -> s *)
  let h = Sign.s2f s in
  let c = src h in
  Dule.pack (f_ID c, t_id h, s)

let m_Comp m1 m2 = (* : r1 -> s2 *)
  let f1 = Dule.type_part m1 (* : r1 -> s1 *) in
```

```
    let t1 = Dule.value_part m1 in
    let f2 = Dule.type_part m2 (* : r2 -> s2 *) in
    let t2 = Dule.value_part m2 in
    let f = f_COMP f1 f2 in (* s1 = r2 *)
    let it2 = t_FT f1 t2 in
    let t = t_comp t1 it2 in
    let s2 = Dule.codomain m2 in
    Dule.pack (f, t, s2)
```

The following drawing shows the domains and codomains of the transformations appearing in the definition of `m_Comp`, according to Definition 5.1.2. Transformations are here depicted by arrows. Compositions in the underlying category are denoted by horizontal bars (double minuses).

```
       t1    f1         t2    f2         t    f1
  r1 ------> --    r2 ------> --    r1 -----> --
             s1               s2              f2
                                              --
                                              s2
```

The drawing below illustrates the value part of the result of module composition. Horizontal composition of transformations is here represented by placing the first transformation above the second. Vertical composition is represented by sharing a common domain/codomain. Transformation marked `t_id` is the identity transformation.

```
                 t_id
       t1    f1 ------> f1
  r1 ------> --         --
                        f2
             s1 ------> --       s1 = r2
                  t2    s2
```

Modularly speaking: `m_Comp` instantiates the values of `m2` with the concrete implementation of the types given in `m1` and then applies the instantiated procedure to the values of `m1`. This is illustrated in much detail in the long example below, which may be safely skipped on the first reading.

Example. Consider the following composition written in the concrete syntax of W-Dule (to be defined shortly).

```
  {M = :: {} -> sig type t1 value v1 : t1 end
      struct type t1 = {} value v1 = {} end}
  .
  :: {M : sig type t1 value v1 : t1 end} ->
       sig value v2 : M.t1 end
  struct value v2 = M.v1 end
```

The module represented by the above expression can be written using the language of *2-LC-lc* with the following abbreviations:

```
  fpp = F_pp(C_PP(nil), nil)
  cmt = C_PP("M" : C_PP("t1" : C_BB))
  M.t1 = F_COMP(F_PR("M" : C_PP("t1" : C_BB), "M"),
                F_PR("t1" : C_BB, "t1"))
  M.v1 = T_comp (T_pr ("M" : F_pp(cmt, "v1" : M.t1), "M"),
                 T_pr ("v1" : M.t1, "v1"))
```

The module is expressible as the following application of the combinator `m_Comp` to modules expressed using the auxiliary functions `pack`, `f2s` and the abbreviations.

```
m_Comp
 (pack (F_RECORD(C_PP(nil),
          "M" : F_RECORD(C_PP(nil), "t1" : fpp)),
        T_record(fpp,
          "M" : T_record(fpp, "v1" = T_record(fpp, nil))),
        Sign.f2s (F_pp(cmt,
          "M" : F_pp(cmt, "v1" : M.t1)))))
 (pack (F_ID(cmt),
        T_record(F_pp(cmt, "M" : F_pp(cmt, "v1" : M.t1)),
          "v2" = M.v1),
        Sign.f2s (F_pp(cmt,
          "v2" : M.t1))))
```

When computing the expression, the implementation of value `v2`, which is the functor `M.v1`, is multiplied from the left by the implementation of the types of the first module, that is the functor

```
  F_RECORD(C_PP(nil), "M" : F_RECORD(C_PP(nil), "t1" : fpp))
```

resulting in the composition of projections

```
  T_comp (T_pr ("M" : F_pp(C_PP(nil), "v1" : fpp), "M"),
          T_pr ("v1" : fpp, "v1"))
```

The composition is then vertically composed (as the second operand) with the implementation of values of the first module, that is transformation

```
  T_record(fpp, "M" : T_record(fpp, "v1" = T_record(fpp, nil)))
```

resulting in the transformation

```
T_record(fpp, nil)
```

The whole result module can be expressed as

```
pack (F_RECORD(C_PP(nil),
        "M" : F_RECORD(C_PP(nil), "t1" : fpp)),
      T_record(fpp,
        "v2" = T_record(fpp, nil)),
      Sign.f2s (F_pp(cmt,
        "v2" : M.t1)))
```

Notice that the codomain of the module refers to type `t1` from signature `M`, which is no longer present anywhere in the result module. This peculiar property of categorical composition of modules is discussed in more detail in Section 9.2.2. The result module is expressible in W-Dule using the mechanism of context signatures (see next sections) to retain the signature `M`

```
:: {} ->
     sig {M : sig type t1 value v1 : t1 end}
       value v2 : M.t1
     end
struct value v2 = M.v1 end
```

Theorem 5.1.4. *The above definitions result in a category, where signatures are objects, modules are morphisms,* `m_Id` *is the identity and* `m_Comp` *is the composition.*

Proof. The domains and codomains of modules are well defined. The equalities about identity as the neutral element of composition follow promptly from the properties of identities in *2-LC-lc*. The associativity of composition is easy to establish, again using the properties of *2-LC-lc*. □

5.1.3 Products

Products are necessary in our model to give semantics to modules that depend on many arguments. It turns out that SCM has (labeled) products.

Theorem 5.1.5. *Simple Category of Modules is cartesian.*

Proof. We construct the (labeled) cartesian structure by defining relevant semantic combinators in OCaml. For illustration of the inner workings of the structure see Section 5.2.4 where its generalization is described in detail and with examples. Consider the following definition of product operations, where `s_Pp_ordinary` stands for the labeled product signature, `m_Pr_ordinary` for the labeled projection and `m_Record_ordinary` for the labeled record operation.

```
let typesPp ls =
  let capture_type_part s =
    let la = Sign.type_part s in
    c_PP la
  in
  vmap capture_type_part ls

let footPp lc i s =
  f_PR lc i

let s_Pp_ordinary ls =
  let lc = typesPp ls in
  let legPp (i, s) =
    let foot = footPp lc i s in
    f_COMP foot (Sign.s2f s)
  in
  let legs = bmap legPp ls in
  let c = c_PP lc in
  let body = f_pp c legs in
  Sign.f2s body

let m_Pr_ordinary lr i = (* : S_Pp_ordinary lr -> s *)
  let s = find i lr in
  let r = s_Pp_ordinary lr in
  let lc = Sign.type_part r in
  let foot_i = footPp lc i s in
  let legs = Sign.value_part r in
  let t = t_pr legs i in
  Dule.pack (foot_i, t, s)

let m_Record_ordinary r lm = (* : r -> S_Pp_ordinary ls *)
  let lf = vmap Dule.type_part lm (* : r -> s_i *) in
  let lt = vmap Dule.value_part lm in
  let ls = vmap Dule.codomain lm in
  let g = Sign.s2f r in
  let c = SrcFCore.src g in
  let f = f_RECORD c lf in
  let t = t_record g lt in
  let s = s_Pp_ordinary ls in
  Dule.pack (f, t, s)
```

The three combinators are well defined; in particular when given correct operands they produce signatures and modules as defined in Definition 5.1.1 and Definition 5.1.2. The two module operations, projection and record, result in modules with domains and codomains as required (and stated in the comments). They also satisfy the two equations required of a product (quoted in Theorem 5.2.7). Therefore the three operations represent the cartesian structure of SCM. A detailed proof of the two product equalities in a more general case, where also sharing of types is allowed, is given in the proof of Theorem 5.2.7. □

Once we have products, we can easily model in SCM a module system similar to the one of Standard ML but with no sharing requirements. Sharing requirements are used in modular programming to ensure that certain types, possibly appearing in distant modules, are equal. In a framework enabling abstraction, such as ours, guarantees of type equality are crucial to enable inter-operation between modules. However, sharing can be difficult to model. In particular, the ordinary labeled products of SCM do not suffice for this task.

5.1.4 Equalizers

Binary pullback of signatures is a good model of a signature of a pair of modules with some sharing between the two. Consider the following example, in which we use the language of W-Dule (defined in the next section). Additionally we write "`sharing type`" to mark an ad hoc notation for sharing equations, which in this case specifies that the two product types are equal (and if the product operation is injective in the *2-LC-lc*, then `M1.t` is equal to `M2.t` and `M2.u` is equal to `{}`).

```
{
 M1 : sig type t end;
 M2 : sig type t type u end;
 sharing type
   {t : M1.t; u : {}}
   = {t : M2.t; u : M2.u}
}
```

Such a signature can be interpreted as a pullback of two morphisms (modules) from the signatures `M1` and `M2`, respectively, to a common target (e.g., signature `sig type c end`). The morphisms determine the types to be identified. The first of the morphisms could look as follows:

```
struct
  type c = {t : M1.t; u : {}}
end
```

and the second as follows:

```
struct
  type c = {t : M2.t; u : M2.u}
end
```

However, neither finite nor labeled pullbacks (labeled limits of diagrams consisting of morphisms with a common codomain, see Section 3.1) are adequate for a similar task in case of many signatures. A sharing requirement may refer to components that are absent in some of the (possibly numerous) signatures, while the pullback construction is based on diagrams with morphisms from all the objects. We could overcome the problem by considering multiple sharing equations and adding a trivial equation for each signature absent from the main equation, but there are more elegant and efficient solutions.

A pullback of given morphisms can be constructed as an equalizer (generalized from two morphisms to a family of morphisms) of the morphisms prefixed with projections [133]. The projections come from the product of sources of the morphisms. This representation is valid for the labeled pullbacks as well, and involves labeled products and equalizers. If we allow the equalizer to be taken of a smaller family than the one used in the product, we can capture the sharing of components of only the chosen signatures. In fact, this construction is just the construction of the general (labeled) limit of a diagram ([110], Theorem V.2.1). It turns out that a product of signatures with some sharing is just the limit of the diagram formed by all the signatures and the morphisms representing the sharing requirement.

Let's suppose we are to represent categorically a collection of five signatures with a type shared among three of them.

```
{
 M1 : sig value v : {} end;
 M2 : sig type t end;
 M3 : sig type t end;
 M4 : sig type u type w end;
 M5 : sig type t end;
 sharing type
   M2.t = M3.t = M5.t
}
```

First, we can represent the types to be shared as three morphisms (`t2`, `t3`, `t5`, in this case just identities) into a common target `T`. Then, to construct the limit, we take the product of the five signatures and compose the respective projections (`pi2`, `pi3`, `pi5`) with the three morphisms. The equalizer of the family of the three compositions is the sought limit signature `P`.

```
                     pi1
                M1  -----> M1
                 x   pi2        t2
                M2  -----> M2 ----____
   equalizer     x   pi3              \
P ----------->  M3  -----> M3 ---------> T
                 x   pi4       t3   __/
                M4  -----> M4    __/
                 x   pi5      __/   t5
                M5  -----> M5
```

If there are several sharing equations, the sought signature is again the (labeled) limit of the diagram, this time containing several families of morphisms, each sharing a codomain. In the construction using product and equalizer, the equalizers representing consecutive sharing equations have to be composed. Let us formalize the two main concepts. We will gloss over the distinction between the labeled and standard notions.

Definition 5.1.6. *A diagram of sharing requirement (sharing diagram) is a graph with vertices labeled by signatures and edges by modules (a categorical diagram in SCM). If all the modules have empty value parts, the diagram is called a diagram of type sharing requirement (type sharing diagram).*

Definition 5.1.7. *The labeled limit of a diagram of sharing requirement, if it exists, is called the product signature (of the diagram) with sharing (represented by the diagram).*

Definition 5.1.8. *Let D be a diagram of sharing requirement. Let C be a family of modules with a common domain: for each signature R in the graph D, a module with codomain R. We say that family C satisfies the sharing requirement represented by D, if C forms a categorical cone (see Section 3.1) over D.*

Theorem 5.1.9. *Let S be a product signature with sharing represented by diagram D. Then the set of modules with the codomain signature S is in bijection with the set of families of modules satisfying the sharing requirement represented by D.*

Proof. This is a standard fact about categorical limits [49]. In fact, this is a reformulation of the definition of categorical limits using the formalism of adjunctions. The construction of the bijection is as follows. Let M be a module with the limit signature S as its codomain. The family of modules satisfying the sharing requirement represented by D is constructed by composing M with each of the morphisms of the limiting cone of S. □

The construction of the reverse bijection, producing the unique morphism into the product, later written as `m_Record`, would be of more interest to us. In the most general case it provides a way to implement the operations of signature product with arbitrary sharing. Unfortunately, the construction is different for different categories and classes of limits. We will describe our construction in SCM for our chosen class of limits in Section 5.2.4. Here we will try to answer a prior question: which sharing diagrams have limits in SCM and how to construct the limits.

As described above, a limit can be constructed using products and equalizers. We know that all labeled products in SCM exist and are easy to construct. On the other hand, the equalizers of morphisms (modules) with non-empty value parts are hard to construct. Our modules are not generative and so their equality is decided by comparing components. Therefore, to work with arbitrary equalizers we would have to compare values, which in practical core languages is undecidable.

In the initial *2-LC-lc* (and any of its extensions described in our thesis) finding the most general unifier, modulo *2-LC-lc* equalities, of a pair of `funct`-terms is decidable. Consequently, in the SCM built upon the initial *2-LC-lc* we can automatically verify if an equalizer of a given (finite) family of modules with empty value parts exists. We can also construct the relevant equalizer morphisms, although the construction is complicated and requires thorough inspection of `funct`-terms representing type parts of the modules.

Remark. A proof of existence of the most general unifier and of the decidability is beyond the scope of our thesis. It follows from the existence of a complete [38] rewriting system for `funct`-terms of *2-LC-lc* and its extensions (no such system is presented in our thesis) and from the general theorems about first-order unification (not modulo equalities). When unifying `funct`-terms, the role of variables would be played by some of the projections appearing in terms. See also our remarks about type reconstruction in Section 10.4. Note that if our categories modeled second-order polymorphism, the unification would be undecidable [66].

The details of the procedure for constructing the equalizer object and the related morphisms are beyond the scope of this thesis, too. The procedure is a moderate generalization of the abstract construction informally described in the proof of Lemma 5.1.11, but the unification restricts validity of the procedure only to term models. The correspondence between projections and variables, established for the sake of unification in the procedure, could also constitute the basis of the formal account of the construction of equalizer morphisms in Lemma 5.1.11.

An example of a non-existing equalizer of modules with empty value parts may be the representation of sharing between product types with two fields and one field, given below. In the SCM build upon the initial *2-LC-lc* such product signature with sharing does not exist.

```
{
 M1 : sig type t end;
 sharing type
   {t : M1.t; u : M1.t}
   = {t : M1.t}
}
```

To precisely link our informal syntactic presentations of sharing requirements with our categorical notions we introduce the following definition. The definition will only be used in discussion about the strength of the presented sharing framework. It will not be needed anywhere else in our thesis, because in our module system we use implicit sharing instead of any explicit sharing equations.

Definition 5.1.10. *Let $E_1, E_2, \ldots, E_n$ be type expressions (*`funct`*-terms) with sources represented by type parts of signatures $S_1, S_2, \ldots, S_n$, respectively, and targets represented by the type part of a common signature S. The representation of the type sharing equation $E_1 = E_2 = \ldots = E_n$ is the sharing diagram consisting of n modules with empty value parts, where the k-th module has domain S_k, codomain S and type part E_k. The representation of a set of equations is the sum of the diagrams representing individual equations.*

In the type sharing diagram from the example given above:

```
M1
     t2
M2 ----____
           \
M3 ---------> T
    t3   __/
M4    __/
   __/   t5
M5
```

the sub-diagram consisting of the three identity modules `t2`, `t3` and `t5` represents the type sharing equation `M2.t = M3.t = M5.t`.

Labeled limits of families of modules with empty value parts represent fully general type sharing equations, where the equated entities are arbitrary type expressions (as in the first example in this section). Each form of the somewhat restricted type sharing typical for the SML'97 [120] module system can be represented by limits of modules with empty value parts. The restrictions are necessary for SML, because their core language with polymorphism would render general type sharings undecidable [118].

Type abbreviations, that is, sharing equations of core language types where among equated types only one is permitted to be non-abstract (only one module not an identity or a projection), are much easier to verify and construct. In particular, if a type abbreviation mentions only types present in its signature and there is no self-nesting (adding `M3.t = M1.t` to the sharing in the example below would incur one) the corresponding equalizer is guaranteed to exist. Here is an example of a type abbreviation for which the equalizer signature exists:

```
{
 M1 : sig type t end;
 M2 : sig type t end;
 M3 : sig type t end;
 sharing type M1.t = M2.t = {t1 : M3.t; t2 : M3.t}
}
```

Another class of sharing specifications has an even simpler construction of the equalizer. These are sharing requirements between whole type parts of modules.

```
{
 M1 : sig type t type u end;
 M2 : sig type t type u end;
 M3 : sig type t end;
 M4 : sig type ttt type uuu end;
 sharing type M1 = M2
}
```

In this case, all of the shared types are abstract and their names are reflected in the kinds of the whole type parts. If the kinds do not agree (as would be with `M1 = M3` or `M1 = M4`, unless the underlying *2-LC-lc* is degenerated enough to equate them), then the sharing clearly cannot be represented as an equalizer, because the family of morphisms corresponding to the equated types does not have a common target. Otherwise, if only the modules with type parts to be shared are present (`M1 = M7` not allowed), the equalizer is expressible and easy to construct (no self-nesting expressible here).

Lemma 5.1.11. *Let us fix an SCM over an arbitrary 2-LC-lc. For each sharing diagram in the SCM representing sharing requirement between whole type parts of modules specified with equal kinds, there always exists a product signature with sharing represented by the diagram.*

Proof. Type parts of such sharing diagrams are just compositions of projections at module names. It is enough to show that equalizers of diagrams consisting of compositions of projections exist.

The source of the constructed equalizer signature has less components than the original signature S. The removed components are all the equated whole type parts, except for one representative. The construction of the equating morphism (module) from the equalizer signature to S is straightforward — the type part is just a record of projection functors. For each of the omitted whole type part names there is a projection at the name of the remaining representative.

The operation of trimming a morphism with codomain S, so as to obtain a morphism into the equalizer, is simple as well. It is enough to compose the morphism with a record of only those projections that have their labels in the equalizer signature. The construction has all the properties of the categorical equalizer, which follows from the properties of products in the underlying category of the *2-LC-lc*. □

In our module system we will apply a variant of the whole type parts sharing requirements defined precisely in Section 5.2.4. In this variant the modules to be equated are determined by names of nested signatures contained in a product. The sharing should take place between type parts of module signatures having the same labels. For instance, the following signature (featuring context signatures, as described in the next section) requires the values `M1.v1` and `M2.v2` to have the same type.

```
{
 M1 : sig {M : sig type c end} type t1 value v1 : M.c end;
 M2 : sig {M : sig type c end} value v2 : M.c end
}
```

In the following example, under the semantics of Section 5.2.4, type `M.c` occurring in three context signatures and one main signature of the product will be shared in all four main signatures.

```
{
 M1 : sig {M : sig type c end} type t1 end;
 M2 : sig {M : sig type c end} end;
 M3 : sig {M : sig type c end} value v : M.c end;
 M : sig type c end
}
```

The context signature of `M1` indicates that `M` is imported into `M1`, or rather that `M1` is built from `M`. The name `M`, assigned to one of the main signatures of the product, is interpreted as marking the same module that was used in construction of `M1`. The module `M` is, in this particular case, required to be provided separately, perhaps to be used for building other modules later on.

Our operation, equating whole type parts of modules with the same names, has the same syntax as for the ordinary labeled product, as no explicit sharing specifications need to be written (notice the absence of explicit "`sharing type`" requirements in the above example). The operation will be called product with name-driven sharing or (ambiguously but succinctly) just product. Every product with name-driven sharing is a categorical limit of a simple sharing diagram and can be constructed as a composition of a number of equalizers of whole type parts of modules (slightly generalized to allow nesting), taken on the product of signatures. The operations of projection with name-driven sharing and record with name-driven sharing are expressible in an analogous way using the operations of the ordinary product and the equalizer. The complete definition in Section 5.2.4 does not mention equalizers of SCM, expressing the limit construction directly in the language of *2-LC-lc*.

5.1.5 Conclusion

We have modeled a system of modules as a category — the Simple Category of Modules (SCM). Signatures are objects of the SCM, modules are morphisms, parameters of a module are described in its domain, the result in its codomain and composition of modules is modeled as the composition in the category.

We have shown that SCM is cartesian (Theorem 5.1.5). Therefore, similarly as with the core language, we are able to model dependency of a module on multiple named modules and we can express module variables as projections. Moreover there is enough equalizers (limits) in SCM to model type sharing specifications. We show that the kind of equalizers to be used for the semantics of our module language has a simple construction in SCM (Lemma 5.1.11).

As building blocks for the SCM we use solely the morphisms of an arbitrary but fixed *2-LC-lc*. Because objects and morphisms of the SCM are just (tuples of) morphisms of the fixed *2-LC-lc*, it is easy to extend the SCM with additional operations defined using the language of *2-LC-lc*, provided that proper semantic invariants are maintained, so that we stay within the category.

5.2 W-Dule — module system with specialization

The module system W-Dule (named after the Dule keyword `with` denoting instantiation and specialization) provides a language for the Simple Category of Modules. Moreover, it is possible to express products with name-driven sharing and three additional operations: signature specialization, module instantiation and module trimming. The modules of W-Dule are first-order and not recursive. Despite this simplicity, W-Dule constitutes a powerful engine for a variety of module languages to be described throughout the thesis.

The code of the semantics of W-Dule uses the semantic combinators of *2-LC-lc* (see Section 4.4). Whenever we fix a *2-LC-lc*, the W-Dule semantics maps `sign-`

terms and dule-terms of W-Dule to signatures and modules of the uniquely determined SCM, and the signatures and modules are in turn constructed from functors and transformations of the *2-LC-lc*. Thus, whenever we consider a *2-LC-lc* with additional structure, we obtain a W-Dule with modules constructed using the extended core language and module operations acting on such modules. This property will allow us to blend our module system with richer instances of the base categorical model (and, consequently, stronger core languages) we develop in the next chapters.

5.2.1 Language

Syntax

The cat, funct and trans-terms come from the language of *2-LC-lc* and represent the core language. The sign-terms represent signatures, that is, interfaces of modules, while the dule-terms represent programming modules themselves.

Below we present the raw syntax of the two new syntactic domains of W-Dule. The raw sign-terms are always type-correct, if only their embedded raw dule-terms are. The conditions on the interfaces of modules, for the raw dule-terms to belong to the language, are described alongside the semantics in the next section. Unlike in the core language, a raw term may belong to the language and still be incorrect in some more subtle sense. This will be discussed in more detail when the combinators corresponding to term constructors are introduced.

As explained in Section 3.2, the phrase sign IList.t represents an indexed list of sign-terms, etc.

```
type sign =
  | S_Pp of sign IList.t
  | S_Bb of sign * cat IList.t * funct IList.t
  | S_Ww of dule * sign
and dule =
  | M_Id of sign
  | M_Comp of dule * dule
  | M_Pr of sign IList.t * IdIndex.t
  | M_Record of sign * dule IList.t
  | M_Base of sign * sign * funct IList.t * trans IList.t
  | M_Inst of dule * dule
  | M_Trim of dule * sign
```

Note the keyword type followed by the keyword and. This signals that the definitions of types sign and dule are mutually recursive. This is caused by the reference to type dule in the profile of signature specialization operation S_Ww (the name comes from the fist letter of the user language keyword `with`, while `S_Bb` is for "base" and `S_Pp` for "product").

Semantics

Now we give an informal overview of the semantics of signature and module operations. What is precise and formal here, are the conditions under which raw terms are considered terms of the language, and the assignment of domain and codomain `sign`-terms to `dule`-terms (presented in a more traditional manner in Appendix A.2.3). We will refer to these syntactic domains and codomains in the next sections, where the complete formal semantics of all the operations is given. In those sections we will also try to illustrate the semantics with commented examples, as well as explain in detail the more subtle points.

For each term constructor that we describe, an abstract syntax is given as well as a concrete, more understandable and succinct one, where indexed lists are denoted by sequences of pairs separated by semicolons, similarly as discussed in detail in Section 6.1.1. The two syntactic forms are used interchangeably, since the arguments to syntactic constructors that are omitted in the concrete syntax are usually unimportant or easy to reconstruct from the context. Recall that at the beginning of this chapter we have fixed an arbitrary *2-LC-lc*.

- `S_Pp`(ls)
  written "`{`i_1 `:` s_1`;` i_2 `:` s_2`;` $\ldots$`;` i_n `:` s_n`}`"
  is a (chosen) labeled product with name-driven sharing in the SCM,

- `S_Bb`(r, la, lf)
  written "`sig` r `type` i $\ldots$ `value` j `:` f $\ldots$ `end`"
  or just "`sig type` i $\ldots$ `value` j `:` f $\ldots$ `end`"
  when the context signature r is easy to recover from the form of lf and from the context
  is the signature specifying core language types and values of a base module with a domain signature r or richer, where the indexed list of type specifications is la = (`cons` (i, `C_BB`) $\ldots$) and the indexed list of value specifications is lf = (`cons` (j, f) $\ldots$),

- `S_Ww`(m_1, s_2)
  written "m_1 `|` s_2"
  is the signature s_2 specialized by a module m_1, as explained in detail in the next sections.

Each `dule`-term constructor is given with its syntactic domain and codomain `sign`-terms as well as the concrete syntax. We provide typing conditions, as well as informally describe the semantics, and hint at the additional definedness side conditions; formal definitions of both are given in subsequent sections. Whether the raw `dule`-term is type-correct and thus belongs to the language is decided by checking the typing conditions (for example, that codomain of m_1 and domain of m_2 are equal in case of `M_Comp`(m_1, m_2)) using the equality of `sign`-terms determined by the formal semantics of `sign`-terms (in the fixed *2-LC-lc* at hand). The semantics of `sign`-terms is mutually recursive with the semantics of `dule`-terms, but only the type part of modules is needed for the type-checking, which therefore remains decidable, even with Turing-complete core languages.

- `M_Id`$(s) : s \rightarrow s$
  written "`:` s"
  is the identity module — the identity morphism of the SCM,

- `M_Comp`$(m_1, m_2) : r_1 \rightarrow s_2$
  written "m_1 `.` m_2"
  is the composition (in the SCM) of morphisms $m_1 : r_1 \rightarrow s$ and $m_2 : s \rightarrow s_2$,

- `M_Pr`(lr, i) : `S_Pp`$(lr) \rightarrow r_i$
  written "i"
  is the module projection (with name-driven sharing) at label i, where the i-th element of lr (called r_i) is required to be present,

- `M_Record`$(r, lm) : r \rightarrow$ `S_Pp`(ls)
  written "`{`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}`"
  is the record (with name-driven sharing) of modules, where the elements of lm are $m_i : r \rightarrow s_i$ and the list of signatures ls has the same labels as lm,

- `M_Base`$(r, s, lg, lt) : r \rightarrow s$
  written "`::` r `->` s `struct type` i `=` g $\ldots$ `value` j `=` t $\ldots$ `end`"
  is the base module that uses elements available from an argument satisfying signature r to define core language types and values as specified in signature s,

- `M_Inst`$(m_1, m_2) : r_1 \rightarrow$ `S_Ww`(m_1, s_2)
  written "m_1 `|` m_2"
  is the instantiation of a module $m_2 : s \rightarrow s_2$ by a module $m_1 : r_1 \rightarrow s$,

- `M_Trim`$(m_1, r_2) : r_1 \rightarrow r_2$
  written "m_1 `:>` r_2"
  is the module $m_1 : r_1 \rightarrow s_1$ with the codomain changed to r_2 and some subcomponents removed, if necessary.

As mentioned above, the complete formal semantics of W-Dule into SCM, using the formalism of *2-LC-lc*, is given by means of semantic combinator definitions expressed in OCaml. More precisely with each term constructor we associate an OCaml combinator of the appropriate type. The definition of such combinator, given in one of the subsequent sections, provides the semantics of the term constructor. The profiles of all the semantic combinators are as follows.

```
val s_Pp : Sign.t IList.t -> [‘OK of Sign.t|‘Error of string]
val s_Bb : Sign.t -> Cat.t IList.t -> Funct.t IList.t ->
             [‘OK of Sign.t|‘Error of string]
val s_Ww : Dule.t -> Sign.t ->
             [‘OK of Sign.t|‘Error of string]

val m_Id : Sign.t -> Dule.t
val m_Comp : Dule.t -> Dule.t -> Dule.t
val m_Pr : Sign.t IList.t -> IdIndex.t ->
             [‘OK of Dule.t|‘Error of string]
val m_Record : Sign.t -> Dule.t IList.t ->
                 [‘OK of Dule.t|‘Error of string]
val m_Base : Sign.t -> Sign.t ->
               Funct.t IList.t -> Trans.t IList.t ->
                 [‘OK of Dule.t|‘Error of string]
val m_Inst : Dule.t -> Dule.t ->
               [‘OK of Dule.t|‘Error of string]
val m_Trim : Dule.t -> Sign.t ->
               [‘OK of Dule.t|‘Error of string]
```

Notice that W-Dule sorts `sign` and `dule` are here mapped to `Sign.t` and `Dule.t`, which are, in turn, realized using the carriers of *2-LC-lc* (capturing SCM morphisms and objects, respectively). In this way W-Dule operations can be defined in terms of *2-LC-lc* operations. Most of the combinators produce results that are either marked as correct or erroneous. If a result is erroneous, it carries a report detailing why the semantics is not defined for the given operands.

The definitions of W-Dule operations do not describe their computational behavior; we give only a semantics in *2-LC-lc* that determines a mathematical model of the modules. On the other hand, the semantics is constructive and consequently when in the chapters to follow we develop a notion of computation for our base categorical model *2-LC-lc*, all modules will have indirectly ascribed a computational meaning: that of the *2-LC-lc* morphism they denote.

5.2.2 Algebra of modules

W-Dule can be seen as a five-sorted partial algebra. While describing the language of W-Dule we have constructed an (infinite, because of labels) algebraic signature.

The algebraic signature is well defined; the indexed lists in the type definitions may be translated into finite products and the distinction between raw and normal terms can be handled easily, for example using the notions of partial algebra. W-Dule is five-sorted; the three sorts of *2-LC-lc* are inherited and two new sorts are introduced. The sorts of the core language (`cat`, `funct` and `trans`) serve only to express components of base signatures (built by `S_Bb`) and base modules (built by `M_Base`). The two main sorts are `sign` and `dule`. The `sign` carrier consists of objects of Simple Category of Modules and the `dule` carrier consists of morphisms of the category.

W-Dule has to be a partial algebra, because the operations refuse to produce a value when their operands are not compatible with each other (in any of several senses). Well-definedness of signature operations of W-Dule will be established in Lemma 5.2.2 and of module operations in Lemma 5.2.3. Since the identity and composition operations are taken straight from the SCM, the algebra W-Dule is a category with some additional structure. Theorem 5.2.1, which is the most important property of W-Dule, shows that the syntactic domains and codomains coincide with the categorical ones.

Theorem 5.2.1. *Let* `m` *be a* `dule`*-term and* `r`*,* `s` *be* `sign`*-terms. Let*

$$\mathtt{m} : \mathtt{r} \to \mathtt{s}$$

be the result of a derivation of syntactic domain `r` *and syntactic codomain* `s` *for the* `dule`*-term* `m`*, according to the rules given in the overview of the semantics in Section 5.2.1 (and written explicitly as derivation rules in Appendix A.2.3). Then the computation of the semantics of* `m`*, as described in the source code definition of semantic combinators below, terminates and yields a result that is either a value in the algebra W-Dule of the term* `m`*, or an error message.*

If the result does not carry an error message but a value `u`*, then the computation of the semantics of* `r` *and* `s` *produces values* `g` *and* `h`*, respectively, such that the 2-LC-lc transformation* `u` *is a module (according to Definition 5.1.2) with the categorical module domain* `g` *and the categorical module codomain* `h` *(which implies that* `g` *and* `h` *are signatures in the sense of Definition 5.1.1).*

Proof. The proof is by induction over the derivation of

$$\mathtt{m} : \mathtt{r} \to \mathtt{s}$$

from the rules described in the overview of the semantics. The inductive steps are easily performed using the following two lemmas. The proofs of the lemmas are constructed in subsequent sections, separately for each signature or module operation. The lemmas are stronger than what is required for the inductive steps, in that they talk about all signatures and modules, while only the expressible ones need to be considered. □

Lemma 5.2.2 (signature invariant). *Each signature operation of W-Dule is well defined by the source code of the corresponding semantic combinator. More precisely, the code of the combinator is type-correct in the underlying 2-LC-lc and the combinator, given operands of the SCM carriers indicated by its profile, yields a value belonging to the carrier of signatures or a value containing an error message.*

Lemma 5.2.3 (module invariant). *Each module operation of W-Dule is well defined by the source code of the corresponding semantic combinator and its result module has proper categorical domain and codomain. More precisely, the code of the combinator is type-correct in the underlying 2-LC-lc and the combinator, given operands of the SCM carriers indicated by its profile, yields a value belonging to the carrier of modules or a value containing an error message. If the result is not an error message then the categorical domain and codomain of the outcome module is as required by the typing in the overview of the semantics.*

The proofs of the two lemmas are constructed gradually in subsequent sections, based on complete precise definitions of semantic combinators. The definitions of semantic combinators and, consequently, the implementation of the Dule compiler are guided by the lemmas and their proofs. In fact, the implementation and the proof of its correctness were developed simultaneously.

Both lemmas admit semantic combinators that are functions constantly giving error results (with the exception of combinators `m_Id` and `m_Comp`, as implied by their profiles). When defining the combinators we will state precisely conditions under which their results are allowed to produce error messages and we will show how each such situation corresponds to an erroneous use of modules as seen from a programming perspective. We will also provide examples of correct modules constructed using the combinators.

Now we provide proofs of Lemma 5.2.3 for the simple cases of the identity and composition operations, as defined formally by the combinators `m_Id` and `m_Comp` in Section 5.1.2. We mark each proof with a header describing required categorical domain and codomain of the resulting module. We begin by assuring type-correctness of the resulting terms and verifying the carrier profile of the combinator. Then we inspect the source code of the type part of the resulting module (called f in Definition 5.1.2) to prove condition 1 of Definition 5.1.2 and the source code of the value part (called t) to prove condition 2.

Proof of module invariant.

- `m_Id` $s : s \to s$
  The application of the combinator obviously always results in type-correct *2-LC-lc* terms and the outcome is of the type `Dule.t`, as required.
  $f =$ `f_ID` c, where f is the type part of the identity module, $c =$ `src h` and

`h` is the representation of s, so condition 1 holds.
t = `t_id h`, where t is the value part of the module, so domain of t is as required in condition 2. The codomain is correct because `f_COMP (f_ID c) h` = `h`. □

- `m_Comp` m_1 $m_2 : r_1 \to s_2$
  Since m_1 and m_2 are from the carrier of modules, the retrieval of module parts is successful. Let f_1, f_2 be the type parts and t_1, t_2 be the value parts of modules m_1, m_2, respectively. The three *2-LC-lc* operations in the source code are type-correct, as illustrated by the diagrams in Section 5.1.2. The last line of the source code definition ensures the proper type of the result.
  f = `f_COMP` f_1 f_2, therefore, because m_1 and m_2 are modules, source and target of f are as required and condition 1 holds.
  t = `t_comp` t_1 (`t_FT` f_1 t_2), so domain agrees with condition 2 by the assumptions about m_1 and m_2. Codomain of t_2 is `f_COMP` f_2 s_2 so the horizontal composition has codomain `f_COMP` f_1 (`f_COMP` f_2 s_2), which is right by associativity of composition in the underlying category. □

5.2.3 Incorporating core language

We now finish the general discussion about semantics and we present in detail how the core language entities are incorporated into signatures and modules. Presentation of each of the respective semantic combinators is concluded with a sketch of a proof for the corresponding case of the signature or module invariant.

Base signature

A simple base signature may be written in the concrete syntax as follows.

```
sig {}
  type t_1
  type t_2
  ...
  type t_n
  value v_1 : f_1
  value v_2 : f_2
  ...
  value v_k : f_k
end
```

where `t_i` and `v_i` are names and `f_i` are core language types (functors of the *2-LC-lc*). The semantics of a base signature is determined by the combinator `s_Bb`, which is defined by the following code (`EqFCat.eq` represents the equality of categories in the fixed *2-LC-lc* and the auxiliary function `typesBb` will be given later on.

```
let s_Bb r la lf =
  (* [la] are local types of this signature,
     [lf] is the value part of this signature
   *)
  (match TypesWSign.typesBb r la with
  |‘OK lc -> (* type part of the signature *)
      let c = c_PP lc in
      let ld = vmap SrcFCore.src lf in
      if vforall (EqFCat.eq c) ld then
        let le = vmap SrcFCore.trg lf in
        if vforall (EqFCat.eq c_BB) le then
          ‘OK (Sign.f2s (f_pp c lf))
        else ‘Error "types are of compound kinds"
      else ‘Error "types depend on wrong modules"
  |‘Error er -> ‘Error er)
```

The operand `r` represents a context signature on which types of values may depend (in the example above the signature is `{}`). The operand `la` represents the user defined type names ($\mathtt{t}_i$ above) of the base signature and the operand `lf` describes types of values (indexed by $\mathtt{v}_i$ in the example above). The code essentially only calls the function `typesBb` and verifies the outcome.

The auxiliary function `typesBb` computes the type part of the base signature, again mainly verifying the arguments, but also performing subtraction and concatenation of indexed lists of kinds (categories). Local, that is user provided types, must have base kinds (with category `C_BB` as their codomains). Otherwise local types could sometimes seem to be context types in signature specialization operation, and then some type-correct user language specialized signatures would be causing internal language errors. Similarly, context types obtained from the context signature `r` must have product kinds (kinds equal to the chosen *2-LC-lc* products of some lists of categories, this is tested with auxiliary function `Cat.isPP`), which is naturally the case when the context signature is a product representing a collection of module parameters.

The notions of local and context types of a signature are described more precisely in Definition 5.2.4 in the next section. The types from the context signature `ld` that have the same labels as the local types defined by the user `la` are hidden, hence the subtraction.

```
let typesBb r la = (* type part of [S_Bb(r, la, lf)] *)
  let ld = Sign.type_part r in (* some will be hidden *)
  if vforall Cat.isPP ld then
```

```
      let lb = subtract ld la in (* context types *)
      if vforall Cat.isBB la then
        (* we merge local and context types: *)
        ‘OK (la @@ lb)
      else ‘Error "typesBb: some local types are not C_BB"
    else ‘Error "typesBb: some context types are not C_PP"
```

In case of the simple signature given at the beginning of this section, the context signature `r` is a trivial, empty product signature. The list of local types `la` is just an indexed list consisting entirely of categories `c_BB` indexed by type names $\mathtt{t}_i$. This is so, because category `c_BB` represents the kind of basic types of the core language. Types of values are represented by the list `lf` of types $\mathtt{f}_i$ with labels $\mathtt{v}_i$.

The `s_Bb` definition says that the example signature will be represented just as a product of functors of the indexed list `lf`. Since `r` is trivial, the list of context types `lb` is empty. Therefore, from the definition of the product of functors we conclude that each $\mathtt{f}_i$ must have source `c_PP la`. Seen from a user perspective this means that the only type names to be used when writing types of values are `t_1`, `t_2`, …, `t_n`, which agrees with the intuition of closed modules programming discipline we employ in Dule.

Example. Let us consider the following signature:

```
sig {ArgumentModule : sig type t end}
  value v : ArgumentModule.t
end
```

This time the list of locally declared types `la` is empty, the list of types of values `lf` in one-element and the context `r` is a product signature with a single label `ArgumentModule`.

What semantics should the above signature have? As the definition of the combinator `s_Bb` states, the semantics should be of the form

```
Sign.f2s (f_pp (c_PP lb) ("v" : cpr))
```

that is a product with the only field labeled `v` and containing the semantics `cpr` of type expression `ArgumentModule.t`. The expression `cpr` would be a composition of projections, first at `ArgumentModule` and second at `t`.

The definition of `s_Bb` shows that the list of context types `lb` contains an element for each highest level type component of the context signature `r`. In this case there is only one such component called `ArgumentModule`, which is a product with one field called `t`. Hence, the semantics of type of a value `v` from the example signature looks as follows:

```
cpr = f_COMP (f_PR lb "ArgumentModule") (f_PR la_t "t")
```

where

```
lb = cons ("ArgumentModule", c_PP la_t) nil
la_t = cons ("t", c_BB) nil
```

Note that even if there were some value specifications in the context signature `r`, they would have been ignored in the semantics of the base signature. The reason is that the context signature is only an aid in determining context types of the main signature. There is no concept of a submodule nor of an embedded or included sub-signature in our language.

Here is the proof of Lemma 5.2.2 for the case of base signature operation.

Proof of signature invariant.

- `s_Bb` *r la lf*
  Functor *r* is a signature, so indexed list of categories `lb` is well defined. Some of the context types are hidden so that the indexed lists can be merged using the operation that fails at duplicated labels. The result is a product functor with a product source, as required. The checks performed in the code of `s_Bb` combinator assure that the product operation on functors is type-correct. □

Base module

Let us look at the definition of the semantic combinator `m_Base` that constructs a base module with a given domain and codomain out of core language types and values.

```
let m_Base r s lg lt = (* : r -> s *)
  (match OkWDule.typesBase r s lg with
  |‘OK f ->
      let g = Sign.s2f r in
      let lgt = vmap DomFCore.dom lt in
      if vforall (EqFFunct.eq g) lgt then
        let t = t_record g lt in
        let ts = DomFCore.cod t in
        let h = Sign.s2f s in
        let fs = f_COMP f h in
        if EqFFunct.eq ts fs then
          ‘OK (Dule.pack (f, t, s))
        else ‘Error
            "value definitions do not agree with codomain"
      else ‘Error "values depend on wrong modules"
  |‘Error er -> ‘Error er)
```

The most important thing to notice is that the result of `m_Base`, if defined, is equal to `pack (f, t, s)`, where the value part `t` is equal to `t_record g lt`. The type part `f` is defined in the auxiliary function `typesBase`, which will be presented later on. For now, let us give an example of the operands of `m_Base` and the value part of the resulting module.

Example. We will have a look at a concrete syntax of an example base module and we'll try to understand what the operands of `m_Base` would be in this particular case. More precisely, if the module below is written in the abstract syntax as `M_Base(r, s, lg, lt)` then we would like to know, what `r`, `s`, `lg` and `lt` are. Throughout the example we avoid writing those context signatures that are easy to deduce.

```
:: {Arg : sig
            type t
            value v : t
          end} ->
      sig (* r *)
        value m : Arg.t
      end
struct
  value m = Arg.v
end
```

First, notice that the domain signature `r` of this module is

```
{Arg : sig {}
         type t
         value v : t
       end}
```

while the codomain signature `s` is

```
sig {Arg : sig {}
             type t
             value v : t
           end}
  value m : Arg.t
end
```

Then observe that the module defines no new types and so the indexed list `lg` is empty. At last, we see that the list of transformations `lt` has only one element, indexed by label `m`. This element is the composition of projections

```
t_comp (t_pr lf_a "Arg") (t_pr lf_v "v")
```

where the indexed lists of functors `lf_a` and `lf_v` are as follows

```
lf_a = cons ("Arg", f_pp (c_PP lc_a) lf_v) nil
lf_v = cons ("v", fat) nil
fat = f_COMP (f_PR lc_a "Arg") (f_PR lc_t "t")
```

and the indexed lists of categories `lc_a` and `lc_t` are

```
lc_a = cons ("Arg", c_PP lc_t) nil
lc_t = cons ("t", c_BB) nil
```

The reader is encouraged to verify that the identified operands of the `M_Base` term constructor are type-correct and of the expected OCaml types, so the application is well defined. Now that we know what `r` and `lt` are, the value part of the semantics is just `t_record (s2f r) lt`.

Let us now study the type part of base modules. Due to the types from the context that may occur in signature `s`, the semantics is not so simple here. Below is the code of the auxiliary function `typesBase` that produces the type part of a module. Before the type definitions will be glued together by the `f_RECORD` combinator, an interpretation of the types annotated with labels from the context must be given. The code of `typesBase` shows how the types from the context are identified by the auxiliary function `partition` and how their meaning is propagated by projections.

```
let typesBase r s lg =
  let lc = Sign.type_part r in
  let c = c_PP lc in
  let ld = vmap SrcFCore.src lg in
  if vforall (EqFCat.eq c) ld then
    let (la, lb) = PartitionWSign.partition s in
    let le = vmap SrcFCore.trg lg in
    if eqset EqFCat.eq le la then
      if subset EqFCat.eq lb lc then
        let pilb = imap (f_PR lc) lb in
        let lf = lg @@ pilb in
        let f = f_RECORD c lf in
        `OK f
      else `Error "context types not provided"
    else `Error "type definitions do not agree with codomain"
  else `Error "type definitions depend on wrong modules"
```

The auxiliary function `partition` divides the type part of a signature into the local and context types lists. First, we check whether a type component is a whole type part as opposed to an individual type by inspecting its kind using the auxiliary predicate `isBB`. A type component with the basic kind is an individual type, which we always treat as local. The remaining types are whole type parts, and whether they are really context types is decided by inspecting the value part of the signature. If the value part indicates the operand is a signature product, the corresponding whole type parts are considered local, otherwise they are context types.

Definition 5.2.4. *Let* `s` *be a signature. By local types and context types of* `s` *we mean the two portions (*`la` *and* `lb`*, respectively) of the type part of signature* `s` *defined in the following auxiliary function.*

```
let partition s =
  let le = Sign.type_part s in
  let lty = vfilter Cat.isBB le in
  let lep = diff le lty in
  (* if [s] has been [S_Pp] (perhaps inside [S_Ww]) *)
  (* then [lty] is [nil] else [lePp] is [nil]: *)
  let lh = Sign.value_part s in
  let lePp = ifilter (fun i -> is_in i lh) lep in
  let la = lty @@ lePp in (* local types of [s] *)
  let lb = diff lep lePp in (* context types of [s] *)
  (la, lb)
```

Example. Once again we return to the example module above. In our example the `f_RECORD` from the definition of `typesBase` consists entirely of projections, because the base module does not define any new types and so `lg` is empty. Moreover, the record is equal to an identity, because `lb` is equal to `lc`, since the context signature of `s` is equal to the domain signature of the base module. Consequently, the only role of the resulting type part is to propagate types from the argument modules.

Let the example module be extended with a type definition:

```
:: {Arg : sig
            type t
            value v : t
          end} ->
     sig (* r *)
       type k
       value m : Arg.t
     end
  struct
    type k = Arg.t
    value m = Arg.v
end
```

Now, what happens in the type part is more interesting. Let us define some indexed lists of categories:

```
lc_t = cons ("t", c_BB) nil
lc_a = cons ("Arg", c_PP lc_t) nil
lc_k = cons ("k", c_BB) lc_a
```

The category `c` is the same as before, namely `c_PP lc_a`. On the other hand, the type part of signature `s` is now larger and equals to `lc_k`. The list `lg` has only one element, which is the functor

```
g = f_COMP (f_PR lc_a "Arg") (f_PR lc_t "t")
```

Notice that the domain of the first projection is `lc_a` and not `lc_k`, for otherwise the type definition would be self-referencing! In this example the `f_RECORD` operations is applied to the list of components consisting of the projection propagating the type `Arg.t` and the functor `g` that is the definition of the type `k`. The result of this application is the type part of the module.

Now we prove Lemma 5.2.3 for the case of the base module operation.

Proof of module invariant.

- `m_Base` $r\ s\ lg\ lt : r \to s$
  The various checks performed in the code ensure that the *2-LC-lc* operations are used correctly. If the result does not contain any of the error messages then the last application is that of function `Dule.pack`, which produces a value of the type representing the carrier of modules `Dule.t`.
  $f =$ `typesBase` $r\ s\ lg$, and `typesBase` returns a functor with the source equal to the source of the functor representing its first operand. Moreover, if `typesBase` haven't returned an error, then the check of the type definitions lg with respect to the signature must have succeeded, which means that f has the same target as the second operand of `typesBase`. Therefore condition 1 is satisfied.
  $t =$ `t_record` $g\ lt$, so domain is as required in condition 2. The correctness of the codomain is checked explicitly in the definition of `m_Base` and amounts to the verification that the values lt have the types specified in the signature. □

5.2.4 Cartesian structure

As proved in Section 5.1.3 the category of signatures and modules is labeled cartesian, which allows for parameterization by named modules. Inside a module expression with product domain the parameter modules are treated as "locally" abstract. When several module expressions are put within a module record with a product domain, they all share the same locally abstract arguments.

The form of base signatures, defined above, implies that type parts of the abstract parameters are locally treated as consisting of abstract types. However, this does not prevent specifying equalities between individual types. As discussed in Section 5.1.4 there exist enough equalizers (limits) to model arbitrary type sharing in SCM. There are numerous variants of type sharing specifications and for each of them there are numerous ways of constructing the corresponding equalizer objects and morphisms. After having investigated the properties of the model, here we make the language decisions concerning type sharing.

For the semantics of W-Dule we choose to employ implicit sharing of whole type parts of modules, determined by names of components, as illustrated on page 129. When grouping signatures in a product, all top-level context types with the same name are to be shared. In the result, top level type components of a product of signatures are the sets of types defined in the signatures themselves indexed by the signature names, together with all their context types. If a product of signatures is nested directly as a field of another product (not indirectly as a context signature) the inner fields are never required to be shared with the outer ones.

This particular merger of labeled product and equalizer, will be called product with name-driven sharing, or just product. The notation for the operations is the same as for the ordinary labeled categorical product. This setup results in concise syntax, with somewhat limited expressiveness, which is recoverable with the help of instantiation operation, described later on.

The definition will be given by means of three combinators written in OCaml. We will prove that the operation `S_Pp` represents the labeled product with name-driven sharing, with `M_Pr` as the projection and `M_Record` as the generalized tuple. We will see that whenever these operations are defined they satisfy all the equalities required of a categorical product. Moreover, whenever they are undefined their signature operands show that the programmer tried to impose a contradicting sharing requirement, or their module operands show that the programmer violated the sharing requirements he had imposed earlier. This will be explained basing on examples.

Whether any of the erroneous circumstances can occur depends on the equality of functors in *2-LC-lc*. For the next two subsections we will assume that the three cartesian structures of *2-LC-lc* are free in our chosen *2-LC-lc* (as will be the case with our initial *2-LC-lc* used for rewriting in Chapter 6). Otherwise no examples of undefined operations could be given, as for instance with trivial *2-LC-lc* (preorder categories) `S_Pp` and `M_Record` are always defined.

Example product errors

We provide two examples of the first possible cause of an error in product operations: contradicting sharing requirements in signatures. The contradiction will be caused by product signatures with inconsistent component lists. Here is the first example:

```
{S1 : sig {Arg : sig type t end}
```

```
          value v1 : Arg.t
        end;
   S2 : sig {Arg : sig end}
          value v2 : {}
        end}
```

This example might be seen as a codomain signature of a record of two modules named S1 and S2. Grouped in a record, the modules should have the same domain signature, so the two base signatures written above should have consistent context signatures. Yet the first context signature has a type t while the second one doesn't contain any type. Thus the programmer has tried to impose a contradicting sharing requirement by demanding that a whole type part with a type t and whole type part with no types were equal. Consequently, the product with name-driven sharing of the two signatures does not exist.

In the second example:

```
  {Arg : sig {} end;
   S : sig {Arg : sig type t end}
         value m : Arg.t
       end}
```

the first signature has an empty context signature. Hence, there cannot be a conflict between an element of its context and of the second signature's context. But the conflict arises from the fact that the first signature is itself named Arg, just as the signature Arg that is the context of the signature at label S. This name coincidence is interpreted as a sharing requirement, possibly hinting at a common module Arg used for composing with the module S and then additionally propagated unchanged in a record. Since the two Arg signatures differ, the sharing is inconsistent and so the product of the given signatures is undefined.

Example record errors

Here we present several (quite trivial) examples of the second cause of errors, which is sharing violation in record modules. For the rest of this section let ~PpArg be an abbreviation for the signature

```
  {Arg : sig type t end}
```

In the first example both modules of the record have the Arg module mentioned in their codomain signatures:

```
  {M1 = {Arg = :: {} -> sig {} type t end
                 struct type t = {m1 : {}} end}
             . :: ~PpArg -> sig ~PpArg end
               struct end;
   M2 = {Arg = :: {} -> sig {} type t end
                 struct type t = {m2 : {}} end}
```

```
          . :: ~PpArg -> sig ~PpArg end
            struct end}
```

The codomain signatures are empty but their context signatures `~PpArg` contain a reference to `Arg`. By the definition of name-driven sharing, having the same module reference `Arg` entails that the type propagated from the `Arg` argument should be equal in both modules of the record. The types are not equal, so the record is not defined. The phrase `{m1 : {}}` above is a product type in the underlying category with one field indexed `m1` and containing again an (empty) product. Consequently, in our examples `t` is a trivial type but indexed differently at each place and thus different.

In the second example below, only the module at label `S` is a composition. However, it is suggested that the `Arg` argument is duplicated and itself included in the record. Unfortunately the types in the two variants of `Arg`, one inside the codomain of the module at label `S` and the other being the codomain of the module at label `Arg`, are not equal:

```
{Arg = :: {} -> sig {} type t end
       struct type t = {arg : {}} end;
 S = {Arg = :: {} -> sig {} type t end
            struct type t = {} end}
       . :: ~PpArg -> sig ~PpArg end
         struct end}
```

Notice that even the following variant of the second example is not correct:

```
{Arg = :: {} -> sig {} type t end
        struct type t = {arg : {}} end;
 S = :: ~PpArg -> sig ~PpArg end
     struct end}
```

Here the type parts of `Arg` are inconsistent, because one is already defined while the other waits for the record to be composed with an `Arg` argument. The type part of the `Arg` component of the record is constructed as a product with a label `arg` and does not depend on any parameter. On the other hand, the propagated type part of the `S` component of the record is implemented as a projection on the record parameter `Arg`. The two functors are different.

Although depending on the parameters of the whole record is legal, it has to be consistent, as in the following correct example:

```
{Arg = Arg;
 S = :: ~PpArg -> sig ~PpArg end
     struct end}
```

or an overall equivalent (despite abstraction) but a more verbose one:

```
{Arg = :: ~PpArg -> sig ~PpArg type t end
        struct type t = Arg.t end;
 S = :: ~PpArg -> sig ~PpArg end
     struct end}
```

The last example, with a different semantics, is correct because in the codomain signature of S there is no dependency on a record parameter:

```
{Arg = :: {} -> sig {} type t end
        struct type t = {arg : {}} end;
 S = :: ~PpArg -> sig {} type t end
     struct type t = {arg : Arg.t} end}
```

In short, sharing requirements are violated in the erroneous examples, because the labels of types from the context are always incorporated into the signatures, regardless of whether they are used or not. Since they are incorporated, they must be correctly shared among different signatures of the record.

On the other hand the labels of values are not incorporated in any way. In the effect, the following two module records are correct, despite the difference of the value parts.

```
{M1 = {Arg = :: {} -> sig type t value v : {} end
             struct type t = {} value v = {} end}
         . :: {Arg : sig type t value v : {} end} ->
                sig {Arg : sig type t value v : {} end} end
           struct end;
 M2 = {Arg = :: {} -> sig type t value v1 : t end
             struct type t = {} value v1 = {} end}
         . :: {Arg : sig type t value v1 : t end} ->
                sig {Arg : sig type t value v1 : t end} end
           struct end}

{Arg = :: {} -> sig type t value v : {} end
       struct type t = {} value v = {} end;
 S = {Arg = :: {} -> sig type t value v : {a:{}} end
            struct type t = {} value v = {a:{}} end}
        . :: {Arg : sig type t value v : {a:{}} end} ->
               sig {Arg : sig type t value v : {a:{}} end} end
          struct end}
```

Definition of the product operation

Let us look at the code of the s_Pp combinator that defines the semantics of signature product.

```
let s_Pp ls =
  (match TypesWSign.typesPp ls with
  |‘OK lc ->
      let legPp (i, s) =
        let foot = FootWSign.footPp lc i s in
```

```
          f_COMP foot (Sign.s2f s)
        in
        let legs = bmap legPp ls in
        let c = c_PP lc in
        let body = f_pp c legs in
        ‘OK (Sign.f2s body)
    |‘Error er -> ‘Error er)
```

Among the auxiliary functions used here, the most important is `legPp`, which actually fits the signatures into their product using the auxiliary function `footPp`. The fitting is based on the source categories `lc` of the constructed product that are computed in many stages in the auxiliary function `typesPp`.

The auxiliary function `typesPp` generates the local and context components of `lc`. Then they are merged together eliminating duplicates. The combinator `s_Pp` is defined if all the duplicated categories (introduced by context signatures) are equal.

```
  let append_cat l k = append_eq EqFCat.eq l k

  let typesPp ls = (* type part of [S_Pp(ls)] *)
    let name_local (i, s) r =
      let (la, lb) = PartitionWSign.partition s in
      let l1 = cons (i, c_PP la) nil in
      (match append_cat l1 lb with
      |‘OK lc ->
          append_cat lc r
      |‘Error er -> ‘Error er)
    in
    bfold1ok nil name_local ls
```

The auxiliary function `partition` computes the local and context types contributed to the product by a single signature. Note that if the given single signature is itself a product, then the whole type parts contributed by the nested product are treated as local type components of the outer product. The definition of the `partition` function has been given in Definition 5.2.4.

The definition of the last sub-procedure `footPp` of the product operation follows. The auxiliary operation `unPP` recovers the indexed list of operand categories from a product category. The choice of the product components is the same as in the auxiliary operation `unPPok`. Since the choice needn't be unique (in some *2-LC-lc*), `li` and `la` may differ.

```
  let footPp lc i s =
    (* [lc] is the type part of a product signature
      of which [s] is the component at label [i]
     *)
    let le = Sign.type_part s in
    let li = unPP (find i lc) in (*labels of local types of [s]*)
    let lb = subtract le li in (* context types of [s] *)
    let la = diff le lb in (* local types, unPP not unique! *)
    let f_PR_lc_i = f_PR lc i in
    let pia = imap (fun j ->
      f_COMP f_PR_lc_i (f_PR la j)) la in
    let pib = imap (f_PR lc) lb in
    let c = c_PP lc in
    f_RECORD c (pia @@ pib)
```

In the code of `footPp` we take advantage of the fact that the type part `lc` comprises of local and context types of all the individual signatures. For a signature `s` at label `i`, operation `footPp` produces a functor from the source of the product signature to the source of the signature `s`. The functor is then used in `legPp` as well as in the combinator `m_Pr`.

Lemma 5.2.5. *If an indexed list of signatures ls shares no labels among the sets of their context types, then* `S_Pp`(*ls*) *is defined and is the categorical labeled product of ls.*

Proof. (Sketch.) If there are no common labels in context types of *ls* then `S_Pp`(*ls*) denotes a product with name-driven sharing where the sharing is void. Consequently, `S_Pp`(*ls*) is defined and its code coincides with that of `s_Pp_ordinary` defined in Section 5.1.3, which by Theorem 5.1.5 is the categorical product in SCM. □

To show that what we have constructed is a well defined operation of the W-Dule algebra, we have to prove Lemma 5.2.2 for the case of the signature product operation.

Proof of signature invariant.

- `s_Pp` *ls*
  The merging of lists in `partition` and `footPp` succeeds because the lists are created as disjoint using the `diff` operations. The application of the `unPP` operation is well defined, because its operand is explicitly constructed in `typesPp` as a product category.

  The projections and their compositions in `footPp` are type-correct by the definitions of the indexed lists of categories, which are used during their construction. The overall result is a product functor with a product source, and therefore belongs to the carrier of signatures. □

Definition of the projection operation

The combinator `m_Pr` representing module projection is defined when its operand signatures meet the same consistency conditions that have been given for `s_Pp`. The auxiliary functions used here are also the same as developed for the signature product combinator.

```
let m_Pr lr i = (* : S_Pp lr -> s *)
  (match find_ok i lr with
  |‘OK s ->
      (match SemWSign.s_Pp lr with
      |‘OK r ->
          let lc = Sign.type_part r in
          let foot_i = FootWSign.footPp lc i s in
          let legs = Sign.value_part r in
          let t = t_pr legs i in
          ‘OK (Dule.pack (foot_i, t, s))
      |‘Error er -> ‘Error er)
  |‘Error er -> ‘Error er)
```

The following observation shows that projections respect the sharing properties encoded in the product signatures. Recall from Section 3.1 that a categorical cone over a diagram in SCM is a family of modules (with a common domain signature and with codomains in each of the diagram's signatures) that makes the diagram commutative.

Observation 5.2.6. *The product signature with the projection modules form a categorical cone over the diagram of sharing requirement representing name-driven sharing among the product operands (see Section 5.1.4).*

Now we prove Lemma 5.2.3 for the projection operation.

Proof of module invariant.

- `m_Pr` $lr\ i$: `s_Pp` $lr \to s$
  The application of the auxiliary operation `footPp` produces type-correct *2-LC-lc* terms, analogously as in the case of the `s_Pp` operation. The `trans`-term constructed by operation `t_pr` is correct, because the functors on the operand list have identical sources (by the definition of `legPp`) and they all have target `c_BB` (by the fact that lr are signatures). The result is constructed using the auxiliary function `pack` and therefore of a correct type.
  $f =$ `footPp` $lc\ i\ s$, where lc is the type part of `s_Pp` lr. It is easy to verify that the result of `footPp` goes from `c_PP` lc to the source of s, which

concludes checking condition 1.

$t =$ `t_pr` *lf* *i*, where by definition of operation `s_Pp` the functor `f_pp` *c* *lf* is equal to `s_Pp` *lr* and so the domain is correct. The codomain should be `f_COMP` (`footPp` *lc* *i* *s*) *s*. The definition of `legPp` shows that the functors on the list *lf* are of exactly this form. Hence, the *i*-th component of *lf* is the correct codomain, satisfying condition 2. □

Definition of the record operation

The code of the `m_Record` combinator is the following:

```
let m_Record r lm = (* : r -> S_Pp ls *)
  let lf = vmap Dule.type_part lm (* : r -> s_i *) in
  let lt = vmap Dule.value_part lm in
  let ls = vmap Dule.codomain lm in
  (match OkWDule.typesRecord lf ls with
  |‘OK lf ->
      let g = Sign.s2f r in
      let c = SrcFCore.src g in
      let f = f_RECORD c lf in
      let t = t_record g lt in
      (match SemWSign.s_Pp ls with
      |‘OK s ->
          ‘OK (Dule.pack (f, t, s))
      |‘Error er -> ‘Error er)
  |‘Error er -> ‘Error er)
```

The auxiliary function `typesRecord` gathers the types being the implementations of the local and context types specified in the codomain signatures. If all these are merged without conflicts then `m_Record` is defined and the result consists of the records of types and values of the operand modules.

```
let append_funct l k = append_eq EqFFunct.eq l k

let typesRecord lf ls =
  let cut_at_i (i, f) r =
    let s = find i ls in
    let (la, lb) = PartitionWSign.partition s in
    let le = la @@ lb in
    let e = c_PP le in
    let lfib = imap (fun i -> f_COMP f (f_PR le i)) lb in
    let pia = imap (f_PR le) la in
    let fia = f_COMP f (f_RECORD e pia) in
```

```
      let l1 = cons (i, fia) nil in
      (match append_funct l1 lfib with
      |‘OK lg ->
          append_funct lg r
      |‘Error er -> ‘Error er)
    in
    bfold1ok nil cut_at_i lf
```

If the operations were not partial, the following theorem would characterize `S_Pp` as a categorical labeled product with projection `M_Pr` and factorizer `M_Record`. The operations are partial to reflect that not all sharing requirements are satisfiable and those that are needn't always be satisfied. However, when the sharing holds, the operations should behave just as the ordinary categorical product operations.

Theorem 5.2.7. *For each of the two equalities below, whenever both sides of the equality are defined and of equal domain and codomain signatures, the equality holds.*

$$\begin{aligned} \texttt{M\_Comp}(\texttt{M\_Record}(r,\, i_1 = m_1\texttt{;}\ \dots\texttt{;}\ i_n = m_n),\ \texttt{M\_Pr}(ls, i_k)) &= m_k \\ \texttt{M\_Record}(r,\, i_1 = \texttt{M\_Comp}(m,\, \texttt{M\_Pr}(ls,\, i_1))\texttt{;} & \\ \dots\texttt{;}\ i_n = \texttt{M\_Comp}(m, \texttt{M\_Pr}(ls, i_n))) &= m \end{aligned}$$

Proof. The product existence requirement

$$\texttt{M\_Comp}(\texttt{M\_Record}(r,\, i_1 = m_1\texttt{;}\ \dots\texttt{;}\ i_n = m_n),\ \texttt{M\_Pr}(ls, j)) = m_j$$

will be established first. Let $\texttt{M\_Record}(r,\, i_1 = m_1\texttt{;}\ \dots\texttt{;}\ i_n = m_n)$ be presented as $\texttt{pack}(f, t, s)$. Let m_j be presented as $\texttt{pack}(g_j, t_j, s_j)$. Then by definition of `m_Record` we have

$$f = \texttt{f\_RECORD c lf}$$

where `lf` are produced by `typesRecord`. Since the merging in `typesRecord` succeeded, `lf` contains all the local and context type implementations found in the type parts lg of the component modules. Thus, after composing with the type part of the projection at label j, the type part g_j is correctly recreated. The reconstruction of g_j is accomplished, on the basis of the target of g_j, with the help of auxiliary function `footPp` appearing in the code of the module projection combinator.

For the value part of the module record we have

$$t = \texttt{t\_record g lt}$$

by definition of `m_Record`, where `lt` are the value parts of the component modules. On the other hand, the value part of the left hand side of the existence requirement is equal to

$$\texttt{t\_comp}\ t\ (\texttt{t\_FT}\ f\ (\texttt{t\_pr}\ lh\ j))$$

for some lh, by definition of `m_Comp` and `m_Pr`. Using the uniqueness requirement of the "inner" product of *2-LC-lc* and substituting the definition of t we simplify this expression to

$$\texttt{t\_comp}\ (\texttt{t\_record g lt})\ (\texttt{t\_pr}\ lh'\ j)$$

for some lh'. This transformation is equal to t_i by the existence requirement in the appropriate cartesian category $C(c, e)$ contained in *2-LC-lc*.

Now we prove the product uniqueness requirement:

$$\begin{aligned}\texttt{M\_Record}(r,\ i_1 = \texttt{M\_Comp}(m,\ \texttt{M\_Pr}(ls,\ i_1));\\ \ldots;\ i_n = \texttt{M\_Comp}(m, \texttt{M\_Pr}(ls, i_n))) \quad = \quad m\end{aligned}$$

Let m be presented as $\texttt{pack}(f, t, s)$. Let $\texttt{M\_Comp}(m, \texttt{M\_Pr}(ls, i_k))$ be presented as $\texttt{pack}(f_k, t_k, s_k)$. Then, by definition of `m_Comp` and `m_Pr` we have

$$f_k \quad = \quad \texttt{f\_COMP}f\ (\texttt{footPp}\ lc\ k\ s_k)$$

where `footPp` ejects from `f` all the components specific to the k-th functor and then adds projections corresponding to some of types of s_k and finally creates a record. When constructing the type part of the module record all these components are released from their records using the auxiliary function `typesRecord` and then they are all merged. The definedness condition asserts that nothing is lost during merging, therefore we get back all the components of `f` as required.

For the value part we have

$$t_k \quad = \quad \texttt{t\_comp}\ t\ (\texttt{t\_FT}\ f_k\ (\texttt{t\_pr}\ lg\ i_k)) \quad = \quad \texttt{t\_comp}\ t\ (\texttt{t\_pr}\ lh\ i_k)$$

for some lg and lh. In the result, the value part of the left hand side is equal to

$$\texttt{t\_record g}\ (i_1 = \texttt{t\_comp}\ t\ (\texttt{t\_pr}\ lh\ i_1);\ \ldots;\ i_n = \texttt{t\_comp}\ t\ (\texttt{t\_pr}\ lh\ i_n))$$

and this, by uniqueness requirement in the appropriate $C(c, e)$, is equal to t.

Notice that not only the cartesian structure of $C(c, e)$, but also the existence and uniqueness equations in the underlying category are crucial for the proofs, as the properties of auxiliary functions depend on them. □

Theorem 5.2.7 together with the fact that the operations are partial does not imply that `S_Pp` is a partial product, that is, for some operands undefined, but for the rest a true categorical product. Indeed, for many signatures, `S_Pp` is their true product and indeed, as we already know, if the categories at the shared labels are unequal, `S_Pp` is undefined. However, for some signatures that have common context types, `S_Pp` is not their product, although the `S_Pp` signature is defined. The object given by `S_Pp` may be "too small" for a product; there may be not enough morphisms into it. This is caused by the operation `M_Record` being undefined for modules that have different components at the fields shown in `S_Pp` as shared. However, by Theorem 5.1.9, the product signature with sharing lacks exactly those elements that would be considered incorrect by a programmer. Furthermore, for the transformations for which the operations are defined, the operations are guaranteed to satisfy the product equalities.

Theorem 5.2.7 and Observation 5.2.6 establish that the `S_Pp` signature with the projections represents the partial operation of categorical (labeled) limit in SCM of diagrams representing name-driven sharing of the product operands. For this claim to be valid we still have to prove Lemma 5.2.3 for the case of the module record.

Proof of module invariant.

- `m_Record` r $lm : r \to$ `s_Pp` ls
  By the assumptions that lm comes from the carrier of modules and thus satisfies Definition 5.1.2, all functors in `lf` have source `c`, and all transformations of `lt` have domain `g`. Therefore the results of the record operations are well defined and the auxiliary function `Dule.pack` turns them into a value of type `Dule.t`.
  $f =$ `f_RECORD (src g) lf`, so the source is as required by condition 1. The parallel between `typesRecord` that constructs `lf` and `typesPp` that gathers the components of the source of `s_Pp` ls shows that the target of f is correct as well.
  $t =$ `t_record g lt`, so the domain is correct with respect to condition 2. The codomain of t is `f_pp` a lh where, by `lt` being value parts of modules, $h_i =$ `f_COMP` f_i s_i. The codomain should be `f_COMP` f (`s_Pp` ls), which by definition of `s_Pp` is a product of transformations. Moreover, the i-th component of the product should be

  $$\texttt{f\_COMP}\ f\ (\texttt{f\_COMP}\ (\texttt{footPp}\ lc\ i\ s)\ s_i)$$

  To conclude the case we recognize that `f_COMP` f (`footPp` lc i s) $= f_i$. A careful analysis of `typesRecord` and `footPp` substantiates this observation.

□

Remarks about product with name-driven sharing

An unusual phenomenon, not among categories, but among product types of programming languages, is that the S_Pp operation is not injective (even for the initial *2-LC-lc*). Different families of signatures can have the same product signature with sharing. This does not contradict the definition of a product (or a pullback), because the bijections known from the adjunction presentation of products are between morphisms with fixed domain and codomain objects, while S_Pp is an operation on objects. Here is an example of two products of signatures that are equal in every *2-LC-lc*, even though the families of signatures themselves are not equal. The reason for the equality is that the context signatures Arg are shared in the first product, so it doesn't matter if the context signature appears once or twice.

```
{S1 : sig {Arg : sig type t end}
        value m : Arg.t
      end;
 S2 : sig {Arg : sig type t end}
      end}

{S1 : sig {Arg : sig type t end}
        value m : Arg.t
      end;
 S2 : sig
      end}
```

The semantics of base signatures could be changed so that the context signature is ignored if not mentioned in the types of values, see the remark in Section 9.2.3. However, this change would still not make S_Pp injective and would complicate the semantics a lot and make it closely dependent on the properties of the core language. Here is a relevant example:

```
{Arg : sig {Arg : sig type t end}
         type t
         value m : Arg.t
       end}

{Arg : sig {}
         type t
         value m : t
       end}
```

The two signature products above have equal semantics, because the two Arg type parts in the first product are identified.

Note that the identification in the last example pertains only to type parts, so the value m needn't be specified in the context signature for the sharing to be consistent. The intricacies of name-driven sharing are further illustrated by the following correct composition of three modules:

```
{M = :: {} -> sig {} type t value v : t end
     struct type t = {} value v = {} end}
.
{M = :: {M : sig type t value v : t end} ->
          sig {M : sig type t value v : t end}
            type t
            value v2 : t
          end
     struct type t = M.t value v2 = M.v end}
.
:: {M : sig {M : sig type t value v : t end}
          type t
          value v2 : M.t (* ! *)
        end} ->
     sig {M : sig {M : sig type t value v : t end}
                type t
                value v2 : t (* OK, values ignored *)
              end}
       value v3 : M.t
     end
struct value v3 = M.v2 end
```

Clearly, the domain of the second module is different from its codomain. However, these two signatures share the type part and the sharing has to be satisfied by the second module. Then the third module can safely rely on the sharing, by identifying the occurrence of type `t` from the inner (context) component `M` and the outer component `M`.

While value parts are unimportant for sharing, both naming of types and naming of modules direct the sharing. Consider the following example, which differs from the previous one only in that the two `M` record components are distinguished by renaming and so the types are not automatically shared. Each of the three modules is correct but the composition is not.

```
{M1 = :: {} -> sig {} type t value v : t end
      struct type t = {} value v = {} end}
.
{M2 = :: {M1 : sig type t value v : t end} ->
           sig {M1 : sig type t value v : t end}
             type t
             value v2 : t
           end
      struct type t = M1.t value v2 = M1.v end}
.
:: {M2 : sig {M1 : sig type t value v : t end}
           type t
```

```
          value v2 : M1.t (* ! *)
        end} ->
    sig {M2 : sig {M1 : sig type t value v : t end}
                type t
                value v2 : t (* OK, values ignored *)
              end}
      value v3 : M1.t
    end
struct value v3 = M2.v2 end
```

This modular program has no semantics because the codomain of the second module is unequal to the domain of the third. Notice that changing the type of value v2 in the domain to M2.t results in an incorrect signature. On the other hand, changing the type to simple t results in an essentially different signature, where we no longer assume sharing of M1.t and M2.t.

5.2.5 Specialization, instantiation and trimming

Signature specialization

The operation S_Ww specializes a signature to a narrower field of use. Here is the code of the corresponding combinator:

```
let s_Ww m1 s2 =
  let f1 = Dule.type_part m1 in
  (match TypesWSign.typesWw f1 s2 with
  |‘OK (lc1, le1, la, lb, lc) ->
      let c = c_PP lc in
      let pic1 = imap (f_PR lc) lc1 in
      let f = f_COMP (f_RECORD c pic1) f1 in
      let pib = imap (fun i ->
        f_COMP f (f_PR le1 i)) lb in
      let pia = imap (f_PR lc) la in
      let re = f_RECORD c (pib @@ pia) in
      let h = f_COMP re (Sign.s2f s2) in
      ‘OK (Sign.f2s h)
  |‘Error er -> ‘Error er)
```

When a signature s2 is specialized by a module m1, the semantics is computed in several steps. First, in the auxiliary function typesWw, we decide which types of signature s2 are the local types and which are the context types (see Definition 5.2.4, where the partition function is introduced).

```
let typesWw f1 s2 =
(* f1 = Dule.type_part m1,
   m1 : r1 -> s1
 *)
let lc1 = unPP (SrcFCore.src f1) in (* type part of [r1] *)
let le1 = unPP (SrcFCore.trg f1) in (* type part of [s1] *)
let (la, lb) = PartitionWSign.partition s2 in
if subset EqFCat.eq lb le1 then
  (match append_cat la lc1 with
  |`OK lc -> (* type part of [S_Ww (m1, s2)] *)
      `OK (lc1, le1, la, lb, lc)
  |`Error er -> `Error er)
else `Error "lb not contained in le1"
```

Then, still within `typesWw`, we merge the local types with the type part of domain of `m1` since the resulting signature will now also depend on these. Having computed the type part, we adjust the functor `f1`, which is the type part of `m1`, to the new source, as well as prune it of spurious components. Next, the functor is extended with projections propagating the remaining types of `s2`. At last, the functor is composed with the representation of `s2`, thus instantiating the signature.

In the process of calculating the types, inconsistencies may appear, when unequal components occur at the same labels. In such cases the specialization is considered incorrect. The inconsistencies discovered by the auxiliary function `subset` inside `typesWw` are caused by the codomain signature of `m1` being incompatible with what `s2` expects.

Example. In the following incorrect specialization, signature `s2` (the last line) expects a module `Arg` to have one type, while the module `m1` (the first line) has empty type part at label `Arg`.

```
{Arg = :: sig end struct end}
|
sig {Arg : sig type t end} value m : Arg.t end
```

The second kind of inconsistencies appears during merging (in `typesWw`) the local types of `s2` with the type part of domain signature of `m1`.

Example. In the following incorrect specialization the merge between the whole type part of the signature `t` and the local type `t` clearly fails.

```
{Specialized =
   :: {t : sig {} type u1 type u2 end} ->
        sig {} type u2 end
   struct type u2 = t.u2 end}
|
sig {Specialized : sig {} type u2 end}
  type t
  value m : Specialized.u2
end
```

Neither of the two kinds of erroneous behavior occurs in programs written in Dule user language described in Section 9.2.4. Examples similar to those above are rejected already during the phase of specification reconstruction, see Corollary 9.2.15. In Dule, the context signature of `s2` is required to be smaller or equal to the codomain of `m1`. This ensures the `subset` check of types succeeds. Moreover in Dule the names of local types are lowercase and the names of signatures capitalized and therefore the types always merge successfully.

The property of distribution of signature product over signature specialization is crucial for the specification reconstruction (see Section 10.4). We formalize the property as follows.

Lemma 5.2.8. *Let m be a module and $lr = i_1 : r_1;\ \ldots;\ i_n : r_n$ be a nonempty indexed list of signatures such that no label on the list indexes a context type of any of the signatures. Then if any side of the following equality is defined, both are defined and the equality holds.*

$$\begin{aligned}&\texttt{S\_Ww}(m, \texttt{S\_Pp}(i_1 : r_1;\ \ldots;\ i_n : r_n))\\ =\ &\texttt{S\_Pp}(i_1 : \texttt{S\_Ww}(m, r_1);\ \ldots;\ i_n : \texttt{S\_Ww}(m, r_n))\end{aligned}$$

Proof. (Sketch.) The proof of the lemma is by rewriting the semantics of the left hand side as the code of the combinator `s_Ww` applied to the code of `s_Pp` and rewriting the right hand side as the code of `s_Pp` applied to a mapping of `s_Ww` to the parameter list of signatures. Then the resulting expressions are proven to be equal using the definitions of the auxiliary functions.

First, we observe that the source categories of the left hand side functor and the right hand side functor are equal, by inspecting the definitions of the auxiliary functions `typesPp` and `typesWw`. A tricky case would be when a label i is contained both in the signature product components list and in the list of context types of some component signatures. Then, if the module m would try to instantiate a context type at label i, the instantiation would be void at the left hand side, because the auxiliary function `partition` marks type i as a local type, since i is one of the signature product labels. On the other hand the instantiation of the context type i would succeed in some signature at the right hand side of the equality. Therefore the equality might fail in such a case.

The premise of the lemma excludes such cases. The implementation of specification reconstruction copes with this restriction by removing the offending type definitions from m and adding dummy (identity) types in their place. This happens whenever the signature expression is transformed from the left hand side form to the right hand side one. In the Dule user language identical names of a module and one of its arguments commonly occur during the construction of mutually dependent modules.

For the value part we notice that the left hand side functor is a composition with a product, while the right hand side functor is a product of compositions. These are equal by the laws of *2-LC-lc*, if only the corresponding operands of composition and product are equal. This is certainly true for the second operands of the functor compositions, because the representations of signatures $r_1, \ldots, r_n$ occur in the compositions unchanged. Unfortunately this appears not to be the case with the first operands of the composition. The definition of the value part of `s_Ww` suggests that the first operands of the composition at the left hand side of the equality are different than the right hand side ones that, furthermore, look as being different from one another.

However, a careful examination of the functor expressions reveals that the apparent differences might lie only in the number and names of projections propagating types. The actual type definitions of module m are present in the same form in all the operands of composition. A second inspection of the auxiliary functions, this time `typesWw` and `footPp`, shows that the differences in the adjoined projections between `s_Ww` and `s_Pp` are compensated when the functor projections of the signature product and the specializations are composed together as parts of the operands of the outermost compositions. □

Note that this proposition does not reduce `S_Ww` to the other signature operations, since expressing specialization of a base signature as a base signature would require the use of core language operations.

We conclude the proof of Lemma 5.2.2 with the case of signature specialization operation.

Proof of signature invariant.

- `s_Ww` m_1 s_2
  The checks in auxiliary function `typesWw` guarantee that the functor projections and compositions in the definition of `s_Ww` are correct. The function `typesWw` ensures also that the lists of categories `la` and `lb` are disjoint and thus the merging of lists of projections doesn't fail due to duplicated labels.

  The result functor has a product source `c_PP lc` as required. The result is a composition of functor `re` with the functor representing signature `s2`. By definition, the latter has to be a product functor. It is easy to observe that a composition of any functor with a product functor always results in a product functor. Thus the result of `s_Ww` belongs to the carrier of signatures. □

Module instantiation

The operation `M_Inst` instantiates a module so that it has more concretely defined manner of operation and more strictly specified field of operation. Here is the code of the corresponding combinator:

```
let m_Inst m1 m2 = (* : r1 -> S_Ww (m1, s2) *)
  let f1 = Dule.type_part m1 (* : r1 -> s1=r2 *) in
  let t1 = Dule.value_part m1 in
  let f2 = Dule.type_part m2 (* : s1=r2 -> s2 *) in
  let t2 = Dule.value_part m2 in
  let s2 = Dule.codomain m2 in
  (match OkWDule.typesInst f1 f2 s2 with
  |‘OK f ->
      let it2 = t_FT f1 t2 in
      let t = t_comp t1 it2 in
      (match SemWSign.s_Ww m1 s2 with
      |‘OK s ->
          ‘OK (Dule.pack (f, t, s))
      |‘Error er -> ‘Error er)
  |‘Error er -> ‘Error er)
```

and its main auxiliary function:

```
let typesInst f1 f2 s2 =
  (match TypesWSign.typesWw f1 s2 with
  |‘OK (lc1, le1, la, lb, _) ->
      let le2 = la @@ lb in (* type part of [s2] *)
      let f12 = f_COMP f1 f2 in
      let pia = imap (fun i ->
        f_COMP f12 (f_PR le2 i)) la in
      let pic1 = imap (f_PR lc1) lc1 in
      (match append_funct pia pic1 with
      |‘OK lf ->
          let pib = imap (fun i ->
            f_COMP f2 (f_PR le2 i)) lb in
          let pie1 = imap (f_PR le1) lb in
          if eqset EqFFunct.eq pib pie1 then
            ‘OK (f_RECORD (c_PP lc1) lf)
          else ‘Error "context types changed in f2"
      |‘Error er -> ‘Error er)
  |‘Error er -> ‘Error er)
```

The combinator `m_Inst` operates on the value parts of operand modules in exactly the same way as the combinator `m_Comp` does. Yet `M_Inst` is not a good candidate for an alternative composition in the category W-Dule: `M_Inst` is not always defined even if codomain of the first operand agrees with the domain of the second.

Example. In the following instantiation there is a conflict when merging the type part of the domain of the first operand of instantiation with the local types of the codomain of the second. In both lists there is the same `Arg` label, but with different kinds. The error is triggered already by the `typesWw` auxiliary function.

```
{Specialized =
       :: {Arg : sig {} end} -> sig {} end
       struct end}
|
:: {Specialized : sig {} end} ->
     sig {}
       type Arg
     end
struct
  type Arg = {}
end
```

Another example of undefined instantiation requires categorical composition of modules. Here the error will be "`context types changed in f2`". Let `~TVArg` be an abbreviation for a signature

```
{Arg : sig {} type t value v : t end}
```

in the following module

```
{Arg = :: sig {} type t value v : t end
       struct type t = {a : {}} value v = {a = {}} end}
|
(
{Arg = :: ~TVArg -> sig {} type t value v : t end
       struct type t = {b : {}} value v = {b = {}} end}
.
:: ~TVArg -> sig ~TVArg value v : Arg.t end
struct value v = Arg.v end
)
```

If the instantiation above was not rejected, the resulting module would consist of value `v` of type `{b : {}}`, while the codomain signature would specify value `v` as having type `{a : {}}`. Similarly as with signature specialization, those module instantiations that are expressible and type-correct in the Dule user language always have defined semantics.

A second phenomenon differentiating instantiation from composition is that the codomain of the whole operation may differ from the codomain of the second operand. This implies that `M_Inst` is not associative.

Example. In the following term, the parentheses cannot be moved towards the end of the expression, lest the module becomes undefined:

```
(
{T = :: {} -> sig type t value v : t end
     struct type t = {} value v = {} end}
|
:: {T : sig type t value v : t end} ->
     sig value v : T.t end
struct value v = T.v end
)
|
: sig value v : {} end
```

The last instantiation could be changed to composition, and the module would remain correct and, in fact, semantically equivalent. However, if the first instantiation is replaced by a composition, then the type `T.t` would remain (locally) abstract and would not be compatible with the type in the signature of the identity module.

The changed codomain of instantiation also makes it impossible that `M_Inst` be associative with `M_Comp` or that the product equalities for signature product, with `M_Comp` changed to `M_Inst`, would still hold. The only positive thing to say here is the following lemma.

Lemma 5.2.9. *Let s be a signature. Then the signature* `S_Ww`$(\texttt{M\_Id}(s), s)$ *is defined and equal to s.*

Proof. We will inspect the code of `s_Ww`. Let the category `c` equal to `c_PP lc` be the source of signature s. We know that the type part of the module `M_Id`(s) is equal to `f_ID c`. Now observe that

$$\texttt{typesWw (f\_ID c)}\ s \;=\; \texttt{(lc, lc, la, lb, lc)}$$

In the result, the list `lc1` from the definition of combinator `s_Ww` is equal to `lc`. Therefore the functor `f` turns out to be an identity. Then we notice that the list of projections inside `f_RECORD` makes it an identity, too. Hence, the last composition that represents the result of combinator `s_Ww`, is equal to the representation of s. □

However, if the signature inside the identity module is different from the specialized signature, bad things can happen.

Lemma 5.2.10. *There exist signatures r and s such that* `S_Ww(M_Id(`r`),`s`)` *is undefined. Moreover, there exist r and s such that* `S_Ww(M_Id(`r`),`s`)` *is defined, but its semantics is different from the semantics of s.*

Proof. The examples of undefined signatures of the above form will be variants of the already presented examples illustrating undefined specialized signatures. The first example is very trivial; failure is signalled already by the `subset` check.

```
: {Arg : sig end}
|
sig {Arg : sig type t end} value m : Arg.t end
```

In the second example, the inconsistency of components indexed by `t` is revealed during merging, as in one of the example signatures specialized by a record.

```
: {Specialized = sig {t : sig {} end} end}
|
sig {Specialized : sig end}
  type t
end
```

Now we proceed to examples with defined semantics. An example of nontrivial specialization by identity may be the following:

```
: {Specialized : sig {Arg : sig type t end}
                   type t
                   value v : Arg.t
                 end}
|
sig {Specialized : sig {} type t end}
  value m : Specialized.t
end
```

Here the original signature has no dependency on `Arg` while the result signature has `Arg` among its types. The resulting dependency on `Arg` may seem vacuous, but if there are axioms in the original signature then they would have to be specialized with the implementation of the `v` operation; and the type of `v` depends on `Arg`.

The example above cannot be expressed and type-checked in Dule user language, since in Dule the value parts of the signatures assigned to `Specialized` have to be equal. Another example is much simpler and its variant can be expressed in Dule:

```
: {Arg : sig type t end}
|
sig {} end
```

As before the original signature does not contain `Arg` and the specialized signature depends on it. In this case, however, the dependency on `Arg` is truly vacuous. The behavior of this example can be reproduced in Dule user language extended with a notation for identity, as follows:

```
spec T = sig end
spec IdWw = : {{Arg : sig type t end}}
            |
            (~Arg : sig type t end -> T)
```

□

Corollary 5.2.11. *Module identity is not a unit for the operation* `M_Inst`.

Proof. Consider a module

$$\texttt{M\_Inst}(\texttt{M\_Id}(r), m)$$

where

$$\begin{aligned} r &= \texttt{\{Arg : sig type t end\}} \\ s &= \texttt{sig \{\} end} \\ m &= \texttt{::}\ r\ \texttt{->}\ s\ \texttt{struct end} \end{aligned}$$

By the semantics of instantiation, the module has codomain

$$\texttt{S\_Ww}(\texttt{M\_Id}(r), s)$$

which is unequal to s, as demonstrated in the proof of Lemma 5.2.10. Hence the module identity cannot be the unit of instantiation. □

In fact, instantiation has neither left nor right units. The lack of a left unit can be proved by generalizing the above example, taking into account that the source of `S_Ww`(m', s) depends only on s and the domain signature of m'. If m' is to play the role of a left unit in the example, its domain has to be r. Then the source category of `S_Ww`(m', s) is the same as the source of `S_Ww(M_Id`(r), $s)$ from the above example, which is different from the source of s, just as before.

For the case of the right unit, one can prove that the identity is not a good candidate, because `M_Inst`$(m,$ `M_Id`$(s))$, has codomain `S_Ww`(m, s), which again has the source different from that of s, if m and s are as in the above example. Then the reasoning is generalized to arbitrary modules, by noticing that `S_Ww`(m, s) does not depend on the form of the candidate for the right unit.

Observation 5.2.12. *Module instantiation has no unit.*

Despite not having the pleasant properties of composition, M_Inst has a profound programming meaning. While M_Comp corresponds to non-transparent functor application [71], where arguments are used as tools only, M_Inst corresponds to transparent application [103], where arguments specialize the output signature of the functor. This is illustrated in the tutorial (Section 2.3), where composition is used in every linking expression, while the instantiation appears mainly under the name with when deploying generic library modules. The categorical composition is equivalent to a variant of the instantiation where every type of the codomain of the instantiated module is considered to be a local type (lb is empty).

Remark. The author is not able to give a simple categorical characterization of M_Inst. Since instantiation is so close to composition (the same value parts) it may well have no short and independent description. Perhaps when the formalism is extended by allowing formal axioms in signatures, and so the difference will drastically widen, an autonomous characterization will emerge (perhaps using fibrations).

Just as all the other operations, module instantiation satisfies the module invariant stated in Lemma 5.2.3.

Proof of module invariant.

- (m_Inst m_1 m_2) : $r_1 \to$ (s_Ww m_1 s_2)
  The type-correctness of the terms resulting from the auxiliary function typesInst follows from the assumption that m_1 and m_2 are modules. The correctness of the multiplication and composition is established similarly as for the module composition. The results are then merged using function Dule.pack, thus obtaining type Dule.t, as required. Let f_1, f_2 be the type parts and t_1, t_2 be the value parts of modules m_1, m_2, respectively.
  f = f_RECORD (src f_1) lf, and the source of f_1 is, by m_1 being a module, the same as the source of r_1. From the definition of s_Ww we know that the source of s_Ww m_1 s_2 consist of the sum of la and lc1. The analysis of typesInst shows that the targets of the lf are exactly these categories, which assures that condition 1 holds.
  t = t_comp t_1 (t_FT f_1 t_2), so by the assumptions concerning m_1 and m_2, the domain is as in condition 2. The codomain of t_2 is f_COMP f_2 s_2 so t has codomain f_COMP f_1 (f_COMP f_2 s_2). On the other hand condition 2 states that the codomain should be f_COMP f (s_Ww m_1 s_2). From the definition of s_Ww we know that s_Ww m_1 s_2 = f_COMP re s_2, where re is defined in the code of s_Ww. We will show that f_COMP f_1 f_2 = f_COMP f re, thus concluding the proof. When the functor f is defined in typesInst, the functors f_1 and f_2 are composed and then some of their types, the instantiated ones, are dropped. What we need to confirm is that these types can be recovered

using f_1 embedded in re. An analysis of the code of s_Ww shows that this is indeed the case. □

Trimming to a signature

The operation M_Trim performs a coercion of a module to a given signature. If necessary, some type and value components of an operand module are removed, and only those mentioned in the operand signature remain. Here is the definition of the combinator m_Trim:

```
let m_Trim m1 r2 = (* : r1 -> r2 *)
  let f1 = Dule.type_part m1 (* : r1 -> s1 *) in
  let t1 = Dule.value_part m1 in
  let g2 = Sign.s2f r2 in
  let e1 = SrcFCore.trg f1 in (* [src s1] *)
  let c2 = SrcFCore.src g2 in
  (match OkWDule.typesTrim e1 c2 with
  |‘OK scf ->
      let f = f_COMP f1 scf in
      let fcr2 = f_COMP f g2 in
      let f1s1 = DomFCore.cod t1 in (* = [f_COMP f1 s1] *)
      (match OkWDule.valuesTrim f1s1 fcr2 with
      |‘OK sct ->
          let t = t_comp t1 sct in
          ‘OK (Dule.pack (f, t, r2))
      |‘Error er -> ‘Error er)
  |‘Error er -> ‘Error er)
```

The auxiliary operations typesTrim and valuesTrim serve for cutting down functors and transformations, respectively. They produce records of projections that upon composition propagate the retained components, omitting all the spurious ones. The function typesTrim produces a functor record of functor projections.

```
let rec typesTrim e c =
  if EqFCat.eq e c then
    ‘OK (f_ID c)
  else
  (match unPPok c with
  |‘OK lc ->
      (match unPPok e with
      |‘OK le ->
          let fsu (i, c) =
            (match find_ok i le with
            |‘OK e ->
                (match typesTrim e c with
                |‘OK sf ->
                    ‘OK (f_COMP (f_PR le i) sf)
                |‘Error er -> ‘Error er)
            |‘Error er ->
                ‘Error "i not in le")
          in
          (match bmap1ok fsu lc with
          |‘OK lf ->
              ‘OK (f_RECORD e lf)
          |‘Error er -> ‘Error er)
      |‘Error er -> ‘Error "e not C_PP")
  |‘Error er -> ‘Error "e <> c")
```

The function `valuesTrim` is analogous but its result is a transformation.

```
let rec valuesTrim h f =
  if EqFFunct.eq h f then
    ‘OK (t_id f)
  else
  (match unpp_ok f with
  |‘OK lf ->
      (match unpp_ok h with
      |‘OK lh ->
          let fsu (i, f) =
            (match find_ok i lh with
            |‘OK h ->
                (match valuesTrim h f with
                |‘OK sf ->
                    ‘OK (t_comp (t_pr lh i) sf)
                |‘Error er -> ‘Error er)
            |‘Error er ->
                ‘Error ("i not in lh"))
          in
          (match bmap1ok fsu lf with
          |‘OK lt ->
              ‘OK (t_record h lt)
          |‘Error er -> ‘Error er)
      |‘Error er -> ‘Error "h not f_pp")
  |‘Error er -> ‘Error ("h <> "))
```

Example. Here is an instance of trimming, with the operation denoted by ":>":

```
  (:: {} -> sig type t value v : t end
   struct type t = {} value v = {} end) :>
    sig value v : {} end
```

In this module expression, type `t` is disposed of, while the value `v` is coerced so that it does not depend on the removed type.

In the code of `m_Trim`, the trimming of the type part is performed first. Note that the result of `typesTrim` is then used to compute operands of `valuesTrim`. By Lemma 5.2.3 and Definition 5.1.2, to obtain the required codomain of the value part we need to compose the type part with the required signature `r2`. But the type part of the operand module, denoted by `f1`, cannot be composed with the representation of `r2`. As in the example above, `f1` can define more types than `r2` specifies. However, functor `f`, which is the result type part obtained using `typesTrim`, is well suited for the composition with `g2`.

Remark. Ideally `M_Trim` should be absent from the language. But it is indispensable when we want to present an instantiated module as if it had not been instantiated. Moreover, when a larger module is used in a role of a smaller one, `M_Trim` relieves the user from writing the interface module.

Both cases occur in the following simplified fragment of a Dule standard prelude described in Appendix C.1. We will not recast this example in the internal module language, as it would be quite long and unreadable. Yet, the languages are close enough that the main points should be understandable even without perusing the tutorial in Chapter 2. In this fragment both the signature specialization and the module instantiation are denoted by keyword `with` and augmented with implicit trimming:

```
spec CharList = List with {Elem = Char}
CharList = load List with {Elem = Char}
spec CharListOps = ListOps with {Elem = Char; List = CharList}
CharListOps = load ListOps with {Elem = Char; List = CharList}
```

Here `Char` is larger than signature `Elem` suggests, so it is trimmed before `List` is instantiated with it. The trimming is implicit and always performed at the right hand side of the keyword `with`, before module instantiation or signature specialization is started.

We want the module `CharListOps` to cooperate with the module `CharList`, that is to operate on a type known to be equal to `CharList.t`, instead of some private unknown type of lists of characters, even if internally implemented as the same type. To this end we would like to instantiate `ListOps` with the record `{Elem = Char; List = CharList}`. But `CharList` no longer has the codomain `List` required by `ListOps`. Even worse `CharList` cannot be coerced alone to signature `List`. Fortunately the whole record can be trimmed down to the signature `{Elem; List}` and so the implicit trimming makes instantiation correct.

If our modules were higher-order and applicative, we would have an option to instantiate `ListOps` with the higher-order module `List` and then with `Char`. For this to work, the codomain signature of `ListOps` would have to be written with many repetitive occurrences of module application in its body.

We verify the last case needed for the proof of Lemma 5.2.3.

Proof of module invariant.

- `m_Trim` m_1 $r_2 : r_1 \to r_2$
  It is easy to verify that the checks performed in auxiliary functions `typesTrim` and `valuesTrim` guarantee that the result *2-LC-lc* terms are type-correct. On the other hand the operands of the auxiliary functions are chosen so that the compositions are correct, too. At the end, the results are glued together by the operation `Dule.pack`, to from a value of type `Dule.t`.
  f = `f_COMP` f_1 `scf`, where `scf` results from the application `typesTrim` (`src` s_1) (`src` r_2). Therefore, by m_1 being a module, the source of f satisfies condition 1. The code of `typesTrim` reveals that if the trimming is successful, the target of `scf` is `src` r_2, and so the target of f is correct, as well.
  t = `t_comp` t_1 `sct`, where `sct` is a result of `valuesTrim`. Due to the similarity of `typesTrim` and `valuesTrim`, condition 2 can be proved analogously as condition 1. □

5.2.6 Conclusion

We have built quite a comprehensive module system W-Dule by extending Simple Category of Modules with several new operations. We have argued that each case of partiality of any of the new operations corresponds to a class of modular programming errors (Observation 5.2.6). We have proved that the semantics of the operations is well defined; in particular, their results belong to the set of objects and morphisms of the SCM (Lemma 5.2.2, Lemma 5.2.3). We have checked that the declared parameter and result signatures of W-Dule modules coincide with SCM domains and codomains of the morphisms the module expressions denote (Theorem 5.2.1). The proof of well-definedness of the semantics constitutes a major part of the proof of correctness of the Dule compiler.

If we fix a *2-LC-lc* for the core language, we can construct the unique SCM built upon that *2-LC-lc*. Our module system W-Dule is an extension of the SCM (treated as a partial algebra) by several module operations, most of which are partial. The carriers are not extended and we don't need any additional assumptions on the core language, in particular we do not require function types.

The semantics of W-Dule is compositional and environment-free, which implies that the modules may be compiled separately. The correctness and the result of a module operation depend only on the target code (SCM morphisms) obtained from the operands, so the original W-Dule source code can be compiled only once and then forgotten. The lack of any environments also ensures that module parameters are abstract. Upon supplying arguments the abstraction can be retained or overcome, depending on the operations used.

The modules of W-Dule are first-order and not recursive. This simplicity makes it possible to entirely embed them, via SCM, into the *2-LC-lc* that is a model of their core language. The fact that the *2-LC-lc* underlies the entire module system allows W-Dule to benefit from theorems about *2-LC-lc* and tools developed for it. Moreover, the fact that *2-LC-lc* has **Set**-based models implies that the module language W-Dule has a set-theoretic semantics.

5.3 L-Dule — module system with linking

The module language L-Dule extends W-Dule with three operations. Each of the operations is similar to the record of modules, but much easier to use in the role it us designed for. The name L-Dule comes from the first letter of the keyword denoting the linking operations.

5.3.1 Language

Syntax

We extend the W-Dule syntactic domain of modules. Each of the operations has the same profile, which only slightly differs from the profile of the module record of W-Dule. Each of the operations has, as its first operand, a list of signatures that correspond to the components of the module record's first operand — its domain.

```
type dule =
  ...
  | M_Accord of sign IList.t * dule IList.t
  | M_Concord of sign IList.t * dule IList.t
  | M_Link of sign IList.t * dule IList.t
```

Semantics

We will describe the semantics, at the same time listing the domain and codomain signatures and providing a concrete syntax for the operations. Notice that the concrete syntax for `M_Accord` is the same as for `M_Record`. From now on, and particularly in the final user language, we will use this syntax for `M_Accord` only. These two operations have a very similar semantics, so this shouldn't be counter-intuitive. In cases when ambiguity might be dangerous, we will use the abstract syntax for both term-constructors. The operations `diff` and `@@`, used below, are the indexed list subtraction and concatenation, respectively (as described in Section 3.2).

- `M_Accord`(lr, lm) : `S_Pp`$(lr) \to$ `S_Pp`(ls)
  written "`{`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}`"
  is the record of modules that are composed with the appropriate projections so that they have the common domain `S_Pp`(lr), where the elements of lm are m_i : `S_Pp`$(lr_i) \to s_i$, each lr_i is a sublist of lr (which may be arbitrarily large, not restricted to the sum of lr_i) and ls has the same labels as lm,

- `M_Concord`(lr, lm) : `S_Pp`$(lr) \to$ `S_Pp`$(lr$ `@@ diff` ls $lr)$
  written "`{{`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}}`"
  is the record of the projections with domain `S_Pp`(lr) and modules m_i : `S_Pp`$(lr_i) \to s_i$ composed with the appropriate projections so that they have the common domain `S_Pp`(lr), where if a label j appears both on lm and lr, then s_j has to be equal to r_j and the projection at j is not included in the record of modules, lr_i is a sublist of lr, lr may be arbitrarily large and ls has the same labels as lm,

- `M_Link`(lr, lm) : `S_Pp`$(lr) \to$ `S_Pp`(ls)
  written "`link {`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}`"
  is the record of modules obtained from lm and all the projections with domain `S_Pp`(lr) by composing each module with records of the others until the resulting domain becomes `S_Pp`(lr), where the elements of lm are m_i : `S_Pp`$(lr_i) \to s_i$, each lr_i is a sublist of lr `@@` ls, lm and lr are disjoint, lr may be arbitrarily large, ls has the same labels as lm and there no cyclic dependency among lm (which ensures termination).

These are profiles of the combinators associated to the three new terms constructors.

```
val m_Accord : Sign.t IList.t -> Dule.t IList.t ->
  [‘OK of Dule.t|‘Error of string]
val m_Concord : Sign.t IList.t -> Dule.t IList.t ->
  [‘OK of Dule.t|‘Error of string]
val m_Link : Sign.t IList.t -> Dule.t IList.t ->
  [‘OK of Dule.t|‘Error of string]
```

5.3.2 Grouping

The operation `M_Accord` is useful when we want to group some modules in a record, but their domain signatures slightly differ. In a module record the domains must be equal, while here we only require them to be product signatures with components at the same labels equal in all modules. The domain of the result is then a product of an indexed list of signatures (the *lr* operand of the term constructor) consisting of the sum of all the components. The codomain is just the product of all the codomains, as in the module record.

Example. Consider the following module expression.

```
M_Accord(
(* lr = *)
  Bool : sig ... end;
  Cargo : sig type t value c1 : t value c2 : t end,
(* lm = *)
  Decision =
    :: {Bool : sig ... end;
        Cargo : sig ... end} ->
      sig {Cargo : sig ... end}
        value chosen_cargo : Cargo.t
      end
    struct
      value chosen_cargo = ...
    end;
  Cargo = M_Pr (Cargo : sig ... end, Cargo)
  )
```

The domains of the modules `Decision` and `Cargo` differ — there is no component `Bool` in the latter. Therefore the module record cannot be used to group these modules, while the operation `M_Accord` accepts such operands.

Remark. One could argue that the projection `Cargo` should be modified by the specification reconstruction algorithm to include the `Bool` component. However, our module system is designed for separate compilation of modules and the projection could have been compiled without the knowledge of its later use. Moreover, the compiled code of the projection can be used in contexts containing different variants of signatures `Bool`. In such circumstances the approach based on whole-program specification reconstruction fails, while the semantic approach of `M_Accord` produces correct results.

Here is the code of the combinator that defines the semantics of the operation. Note how a list of projections `lmf` is used to extend a module to the common domain.

```
let m_Accord lr lm = (* : S_Pp lr -> S_Pp ls *)
```

```
(match SemWSign.s_Pp lr with
|`OK r ->
    (match imap1ok (SemWDule.m_Pr lr) lr with
    |`OK lpr ->
        let prm m =
          let lf = Sign.value_part (Dule.domain m) in
          let lmf = imap (fun i -> find i lpr) lf in
          (match SemWDule.m_Record r lmf with
          |`OK re ->
              `OK (SemWDule.m_Comp re m)
          |`Error er -> `Error er)
        in
        (match vmap1ok prm lm with
        |`OK lm ->
            SemWDule.m_Record r lm
        |`Error er -> `Error er)
    |`Error er -> `Error er)
|`Error er -> `Error er)
```

Only W-Dule operations are used in the definition of the combinator, with the exception of the declaration of the list `lf` that will be explained at the end of the section, after we define the module operation `M_Concord`.

Example. The operation `M_Concord`, corresponding to the user language operation of double record, differs from `M_Accord` in that the parameters of the modules are propagated to the result. The following example doesn't contain the `Cargo` projection, but nevertheless the resulting group of modules will be richer that in the above example. The codomain will contain not only the `Cargo` component, but the `Bool` component as well.

```
M_Concord(
(* lr = *)
  Bool : sig ... end;
  Cargo : sig type t value c1 : t value c2 : t end,
(* lm = *)
  Decision =
    :: {Bool : sig ... end;
        Cargo : sig ... end} ->
      sig {Cargo : sig ... end}
        value chosen_cargo : Cargo.t
      end
    struct
      value chosen_cargo = ...
    end
  )
```

The definition of the combinator is similar to the previous one. The operation uses projections not only to unify its operand modules, but also to extend the resulting record with additional components.

```
let m_Concord lr lm = (* : S_Pp lr -> S_Pp (ls @@ diff lr ls) *)
  (match SemWSign.s_Pp lr with
  |‘OK r ->
      (match imap1ok (SemWDule.m_Pr lr) lr with
      |‘OK lpr ->
          let prm m =
            let lf = Sign.value_part (Dule.domain m) in
            let lmf = imap (fun i -> find i lpr) lf in
            (match SemWDule.m_Record r lmf with
            |‘OK re ->
                ‘OK (SemWDule.m_Comp re m)
            |‘Error er -> ‘Error er)
          in
          (match vmap1ok prm lm with
          |‘OK lm ->
              let lm = lm @@ subtract lpr lm in
              SemWDule.m_Record r lm
          |‘Error er -> ‘Error er)
      |‘Error er -> ‘Error er)
  |‘Error er -> ‘Error er)
```

As in the code of `m_Accord`, there is no access to the *2-LC-lc* other than through W-Dule operations. The only exception is the declaration of the indexed list of functors `lf`. The semantics of the three new operations of L-Dule could be written without any references to *2-LC-lc* but we have decided to use the *2-LC-lc* functors for brevity of the definition.

The problem is that the domains of the operand modules of the L-Dule operations are product signatures and we need to access their components. To allow for recovering components of the non-injective operation of signature product we would have to complicate the semantics greatly. Fortunately, here we need to know only the set of labels that index components of the signature product and not the set of signatures themselves. Therefore, in the definition we access the value part of the domain of the module and inspect the indexes. We know by Definition 5.1.1 and Lemma 5.2.2 that the value part of the signature has the required labels. Thus the access to the labels of a domain signature components is effectively simulated through the simple *2-LC-lc* operations.

The use of essentially only W-Dule operations in the code of `m_Accord` and `m_Concord` makes it easy to verify the following observation.

Observation 5.3.1. *The module operations* `m_Accord` *and* `m_Concord` *satisfy the properties stated as the module invariant in Lemma 5.2.3.*

5.3.3 Linking

The linking operation is the standard way of supplying modules with arguments. Similarly as for the other grouping operations, the domains of modules have to be product signatures and needn't be strictly equal to each other. It suffices if there are equal components at the same labels. The domain of the whole operation is equal, as before, to the product of the first operand *lr* of the term constructor.

Example. Here is an instance of the linking construction:

```
M_Link(
(* lr = *)
  Bool : sig ... end;
(* lm = *)
  Cargo =
    :: {} -> sig type t value c1 : t value c2 : t end
    struct
      ...
    end;
  Decision =
    :: {Bool : sig ... end;
        Cargo : sig ... end} ->
      sig {Cargo : sig ... end}
        value chosen_cargo : Cargo.t
      end
    struct
      value chosen_cargo = ...
    end
  )
```

Like in `m_Accord` and unlike in `M_Concord`, the codomain of `M_Link` is just the product of codomains *ls* of operand modules. Differently than in both the operations of the previous section, module domains may contain components not present in *lr*. Each of these components has to be a codomain of one of the modules, indexed by the name of the module. These components will not be propagated from the domain of the linking operation, but will be the places at which compositions with other modules occur.

The code of the combinator corresponding to the linking operation follows. As stated in the comment, for the operation `M_Link` to be defined we require that there is no circular dependency among the modules. This property is not verified in the presented version of the code — we state it in the comment and depend on the outer layers of the compiler to provide proper operands. Note how we propagate a parameter from the domain, if the find_ok operation succeeds, or compose with another module, otherwise.

```
(* here order of lm doesn't matter,
   but no circularity allowed: *)
let m_Link lr lm = (* : S_Pp lr -> Pp ls *)
  (match SemWSign.s_Pp lr with
  |‘OK r ->
      (match imap1ok (SemWDule.m_Pr lr) lr with
      |‘OK lpr ->
          let rec rlink i =
            (match find_ok i lpr with
            |‘OK pr -> ‘OK pr
            |‘Error er ->
                let m = find i lm in
                let lf = Sign.value_part (Dule.domain m) in
                (match imap1ok rlink lf with
                |‘OK lmf ->
                    (match SemWDule.m_Record r lmf with
                    |‘OK re ->
                        ‘OK (SemWDule.m_Comp re m)
                    |‘Error er -> ‘Error er)
                |‘Error er -> ‘Error er))
          in
          (match imap1ok rlink lm with
          |‘OK lm ->
              SemWDule.m_Record r lm
          |‘Error er -> ‘Error er)
      |‘Error er -> ‘Error er)
  |‘Error er -> ‘Error er)
```

There is a variant of the M_Link operation, given by the combinator called m_Link_ordered, where the modules to be composed with a given module are required to precede it on the indexed list. This is the only place of the whole semantics of Dule where the order of elements on an indexed list matters. Note that this time we have been able to express the traversal of operand modules using structural recursion rather than general recursion.

```
(* here we assume a module depends
   only on the previous ones in lm: *)
let m_Link_ordered lr lm = (* : S_Pp lr -> S_Pp ls *)
  (match SemWSign.s_Pp lr with
  |‘OK r ->
      (match imap1ok (SemWDule.m_Pr lr) lr with
      |‘OK lpr ->
          let pro = fun (i, m) lm ->
            let lf = Sign.value_part (Dule.domain m) in
            (match imap1ok (fun i -> find_ok i lm) lf with
            |‘OK lmf ->
                (match SemWDule.m_Record r lmf with
                |‘OK re ->
                    let m = SemWDule.m_Comp re m in
                    ‘OK (cons (i, m) lm)
                |‘Error er -> ‘Error er)
            |‘Error er ->
                ‘Error "i depends on an unknown module")
          in
          (match bfold1ok lpr pro lm  with
          |‘OK lprlm ->
              let lm = subtract lprlm lpr in
              SemWDule.m_Record r lm
          |‘Error er -> ‘Error er)
      |‘Error er -> ‘Error er)
  |‘Error er -> ‘Error er)
```

The original definition of `M_Link` works regardless of the order of operands. Consequently, it facilitates not only bottom-up but also top-down presentation of the module hierarchy. But the original `m_Link` combinator is ineffective in many respects. For example, topological sorting would have to be adopted to avoid exponential explosion of module compositions and to verify cycle-freedom for the definedness of the semantics. From another perspective, also a programmer might have problems detecting cyclic dependencies in his module definitions, if the original variant had been adopted. Some of these cyclic dependencies might be intentional and those should be expressed using the inductive module construction (see Section 9.1), while others might be inadvertent or unimplementable. The definition used in the Dule user module language is that given by the `m_Link_ordered` combinator.

Observation 5.3.2. *Both, the module operation* `m_Link_ordered` *and the module operation* `m_Link` *(but only applied to operands without cycles) satisfy the properties stated as the module invariant in Lemma 5.2.3.*

5.3.4 Conclusion

The operations introduced in L-Dule are expressed using the operations of W-Dule. This witnesses the power of the W-Dule module system, as well as greatly simplifies proving properties of L-Dule. In particular the proof of well-definedness of the three new operations of L-Dule and the adequacy of their declared domains and codomains is trivial (Observation 5.3.1, Observation 5.3.2).

The new operations of L-Dule ease the grouping of modules, introduced in W-Dule with the module record operation. The grouping alleviates the need for submodules, as known from conventional module systems. The linking operation fits well with the choice of sharing mechanism in W-Dule. The module product operation assumes that components with the same names are shared, and the linking operation assumes parameters are implemented by modules of the same names. Inside linking expression, supplying implementation of the parameters is automatically performed with multiple compositions and the user is free from writing the compositions by hand. There is also no need to artificially introduce submodules for the purpose of specifying their sharing with others; together with implicit composition this greatly reduces common modular bureaucracy. Linking facilitates naming parameterized modules and applying named modules, but the language remains first-order and its implementation avoids copying of code.

The three new product-related operations are capable of joining many parameterized modules into one large parameterized module. Just like in W-Dule, the parameters are treated as “locally” abstract, and they may be passed, still abstract, to the result. However, the new operations aid in making the abstraction more global, by ensuring and sharing the abstraction among their operands. Nesting of the operations can extend the area where the abstraction is safe even up to the top level of the program. When a large parameterized module is, in turn, composed with some arguments, the codomain signature of the result retains the abstract character. However, this time the abstraction can be overcome by instantiation or coercion, if needed.

The individual modules of L-Dule, often very large, are compiled into the same simple categorical internal core language that was the basis of W-Dule. The core language *2-LC-lc* is small, strict and succinct, yet close to the user language. The compilation is not a crude striping of modular decorations — the compiled modules become elements of the Simple Category of Modules and can be composed, instantiated, linked. The core language representation of modules is faithful, despite the simplicity of the core language, and this enables compositional definition of module operations (or separate compilation, in machine terms). The semantics is able to decide if operands to a module operation are correct, when the operands are already compiled to the core language code.

Each of the three new operations adjusts its operands by composing with projections at missing domain components. This shortens notation because only essential domain components have to be written in each operand module. The semantic approach to adjusting operand modules preserves separate compilation, because the operands needn't be recompiled; instead their *2-LC-lc* code is adjusted by compositions, etc.

Chapter 6

Instances of the base categorical model

In Chapter 5 we base the construction of our module system on plain *2-LC-lc*. In this chapter we develop three examples of alternative core languages. The first language fixes core language models to be initial algebras (categories with additional structure, initial among the algebras with the same set of operations and equalities; they are known to exist, as argued in Section 4.1.1) with elements represented by terms and adds an execution mechanism based on term rewriting. More precisely, since there are no identifiers and binding of identifiers in any of our internal languages, there can be no free variables and so all terms are combinators (just as `S`, `K`, `I`, `SKK`, `SII`, `S(K(SI))(S(KK)I)`, etc., are combinators of the `SKI` combinator calculus [11]) and so the rewriting has the simple form of combinator reduction.

The second language extends the first by coproducts, making the categorical model bi-cartesian, and then breaks the symmetry by requiring the coproduct to be distributive. The distributive coproduct, called a sum, is then used to model the notion of conditional execution, while maintaining the capability for parameterization given by the cartesian structure.

The last core language of this chapter introduces inductive and coinductive types, similar to those of Charity [23]. The main focus here is on designing a versatile set of basic combinators, to inspire both the programmer and the compiler designer.

Each of these three extensions to the core language is incorporated into the internal core language of Dule, as summarized in Appendix A.1 and implemented in the Dule compiler.

6.1 p-Core — core language with product type

The programming language p-Core has the same syntax that denotes objects and morphisms of *2-LC-lc* in Section 4.3. The terms of p-Core are assigned semantics in the initial *2-LC-lc*. The carrier of the initial *2-LC-lc* is the set of all *2-LC-lc* terms quotient by the equational theory of *2-LC-lc* developed in the subsequent sections. We develop an evaluation procedure for the value terms of p-Core (`trans`-terms of the initial *2-LC-lc*) and an equality testing procedure for its type terms (`funct`-terms of the initial *2-LC-lc*). Once we have the equality testing procedure for types, we can easily type-check p-Core programs, because the `trans`-terms carry enough `funct`-terms to make their typing unique up to the type equality.

To enable extensions, we construct the equational theory of all *2-LC-lc* instead of the fixed initial *2-LC-lc*. The two theories may differ, as seen in the example in Section 6.1.2. However, every ground equation (equation without variables) that holds in the initial *2-LC-lc* is derivable from the equational theory of all *2-LC-lc*, so for deciding type equality and program rewriting the theory of all *2-LC-lc* suffices.

6.1.1 Syntactic sugar for indexed lists

We will introduce some syntactic sugar to make the forthcoming numerous equations and rules more readable. The notation will eliminate the need for lengthy side conditions to equations, similar to those from the descriptions of the semantics that asserted what certain indexed lists contain and how their elements relate to the elements of other lists.

In the sugared notation an indexed list lc that has `cat`-term c_k at label i_k will be written as

$$i_1 \texttt{ - } c_1\texttt{;}\ i_2 \texttt{ - } c_2\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ - } c_n$$

and the product category $\texttt{C\_PP}(lc)$ as

$$\texttt{<}i_1 \texttt{ - } c_1\texttt{;}\ i_2 \texttt{ - } c_2\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ - } c_n\texttt{>}$$

and similarly an indexed list lf that has `funct`-term f_k at label i_k will be written as

$$i_1 \texttt{ : } f_1\texttt{;}\ i_2 \texttt{ : } f_2\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ : } f_n$$

and the record $\texttt{F\_RECORD}(c, lf)$ as

$$\texttt{<}i_1 \texttt{ : } f_1\texttt{;}\ i_2 \texttt{ : } f_2\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ : } f_n\texttt{>}$$

Note that the `cat`-term c disappears in this last expression. This means that on the rare occasions when c has to be visible in the equation, the notation cannot be used.

Indexed list lt that has `trans`-term t_k at label i_k will be written as

$$i_1 = t_1 \texttt{;}\ i_2 = t_2 \texttt{;}\ \dots \texttt{;}\ i_n = t_n$$

and the record `T_RECORD`(c, lt) as

$$\texttt{<}i_1 = t_1 \texttt{;}\ i_2 = t_2 \texttt{;}\ \dots \texttt{;}\ i_n = t_n\texttt{>}$$

by analogy to the record of functors.

The product `F_pp`(c, lf) in $C(c, e)$ of an indexed list of functors will be written as

$$\texttt{\{}i_1 \texttt{ : } f_1 \texttt{;}\ i_2 \texttt{ : } f_2 \texttt{;}\ \dots \texttt{;}\ i_n \texttt{ : } f_n\texttt{\}}$$

and the record `T_record`(f, lt) of an indexed list of transformations as

$$\texttt{\{}i_1 = t_1 \texttt{;}\ i_2 = t_2 \texttt{;}\ \dots \texttt{;}\ i_n = t_n\texttt{\}}$$

where the `funct`-term f disappears from the last term.

Furthermore, to make the presentations of equations more concise, we will usually omit all the elements of indexed lists but one. So for example the record template

$$\texttt{<}i_1 \texttt{ : } f_1 \texttt{;}\ i_2 \texttt{ : } f_2 \texttt{;}\ \dots \texttt{;}\ i_n \texttt{ : } f_n\texttt{>}$$

will in this short notation be written as

$$\texttt{<}i \texttt{ : } f \texttt{;}\ \dots\texttt{>}$$

This notation assumes that the full form is easy to reconstruct, because either the f element is typical and all the other elements play the same role in the equation, or i is the only meaningful label.

If not stated otherwise, any indexed list that is described as containing all the labels of another list, is assumed to have no other labels as its indexes. On the other hand, if the labels of a list are not specified and the description of the list contains ellipsis, the list may contain arbitrary additional elements. For example, let us consider the list le of the semantics of `F_RECORD`(c, lf) in Section 4.1.2. The typing judgment $f_i : c \to e_i$ implies that le contains all the labels of lf. Here our convention additionally ensures that le has no other labels. On the other hand, the indexed list lc of the definition of `F_PR`(lc, i), which is required only to have element c_i at label i, may be of arbitrary length and content.

In case of uncertainty: a faithful representation of all the rules of this chapter and so of most key equations, is included in our compiler source code. Complete syntactic sugar for all the core language operations is summarized together with their semantics in Appendix A.1.2.

6.1.2 Rewriting in LC

We start designing the rewriting mechanisms for the LC language, From now on we will write a lot of equations so we will often present indexed lists in the sugared form. We will also usually write f . g for the composition F_COMP(f, g) and denote C_BB by *.

Equations

A funct-equation consists of a pair of such funct-terms with variables that are type-correct in every LC (or equivalently, in this particular case, in the initial LC). The funct-variables are implicitly typed, but not with categories of some fixed LC, but with cat-terms, usually just cat-variables. For example variable f in equation (4) below is typed with variable d as its target (and, say, variable c as its source). Typing conditions (see Chapter 4) are easily extended to terms with implicitly typed variables and are easily decidable by reordering and recursive comparison of cat-terms. Additionally, both sides of an equation must have equal (in every LC) source categories and equal target categories.

An equation is said to hold in a category, if for all type-correct valuations of cat-variables and funct-variables the values of the terms are equal (the typed variables and the definition of equation guarantee that type-correct valuations result in type-correct terms). When presenting an equation, we often do not list the types of variables, but the typing should be clear from the context — usually it is the most general typing making the equation type-correct.

Let us start building the theory of labeled cartesian categories by listing equations for identity and composition.

$$f \;.\; \mathtt{F\_ID}(d) \;=\; f \tag{4}$$

$$\mathtt{F\_ID}(c) \;.\; g \;=\; g \tag{5}$$

$$f \;.\; (g_1 \;.\; g_2) \;=\; (f \;.\; g_1) \;.\; g_2 \tag{6}$$

Now we state the existence requirement for the labeled product of functors (the name of the equation is explained in Section 3.1). This axiom is similar to the β-equation of function spaces, so the reduction rules based on it will be called β-rules. See Section 7.1.3 for formal underpinnings of this analogy.

$$\mathtt{<}i : f_i ; \;\ldots\mathtt{>} \;.\; \mathtt{F\_PR}(ld, i) \;=\; f_i \tag{7}$$

We conclude by adding the axiom expressing uniqueness requirement for the labeled products. By analogy with function spaces the rules emerging from this equation will be called η-rules.

$$\mathtt{<}i_1 : f \;.\; \mathtt{F\_PR}(ld, i_1) ; \;\ldots ;\; i_n : f \;.\; \mathtt{F\_PR}(ld, i_n)\mathtt{>} \;=\; f \tag{8}$$

The equational theory determined by the above set of axioms will be named Φ_L. We state that Φ_L is just the equational theory of LC, that is, the theory of such a class of algebras with LC algebraic signature that coincides with the class of all labeled cartesian categories.

Lemma 6.1.1. *Φ_L is the theory of LC. In other words*

- *If $\phi \in \Phi_L$ then ϕ holds in $\mathbf{U}$, for an arbitrary LC $\mathbf{U}$.*
- *Let ϕ be a* `funct`*-equation. If for every LC $\mathbf{U}$, ϕ holds in $\mathbf{U}$, then $\phi \in \Phi_L$.*

Proof. For binary product case this is folklore. See for example [98]. Extending this to labeled products is trivial.

Another proof, based on the general properties of adjunctions, emerges from the discussion in Section 7.1.3. □

Remark. There exist type-correct equations with no type-correct ground instances, because some classes of elements, like the sort of functors from the category `C_PP(nil)` to the category `*`, have no inhabitants expressible in the language of LC. Among those there are such equations that do not follow from the axioms but still vacuously hold in the initial LC, for example:

$$\texttt{F\_ID(C\_PP(nil))} \;.\; f \;.\; \texttt{F\_ID(*)} \quad = \quad h$$

Such equations, however, are falsified in any LC $\mathbf{U}$ that has non-empty respective sorts and unequal `C_PP(nil)` and `*`. This fact confirms that Φ_L is indeed the theory of all LC, and is properly included in the theory of the initial LC.

Reduction

In this section we start designing reduction systems for our languages, by defining a reduction system for the language of LC, based on the theory Φ_L presented in the previous section. We assume the reader is acquainted with reduction systems [38], so we only sketch our cast of the basic notions. Each of our reduction systems will consist of a rewrite relation between the terms of a given typed language, which is given by an algebraic signature and additional type-checking rules.

In general, the rewriting relation is a multi-sorted relation, but in case of LC, only the sort `funct` of the relation is non-empty. A rewrite rule is a directed equation and, consequently, a directed `funct`-equation can be called a `funct`-rewrite rule. Note that such definition of a rewrite rule gives us the subject reduction property (reduction respects types) for all our rewriting systems for free. Rewrite relation will be defined by listing basic rewrite rules. The 1-step rewrite relation is then obtained as type-correct closure of the rules under substitution and context application. As always, the reduction relation is the reflexive, transitive closure of the 1-step rewrite relation. For a reduction system $\mathcal{R}$, by $\overline{\mathcal{R}}$ we mean the equational theory induced by the set of basic rewrite rules of $\mathcal{R}$ (viewed as equations). For an equational theory Φ, $\mathcal{R}$ is sound with respect to Φ if $\overline{\mathcal{R}}$ is contained in Φ; Φ is the theory of $\mathcal{R}$ if $\overline{\mathcal{R}}$ is equal to Φ.

The full uniqueness requirement for products, that is equation (8), is computationally expensive when treated as a contraction (directed from left to right, called η-contraction), because it is not left-linear [94], and so the recovery (using many other rules) of candidates for the term f from all of the list components is more like "exhaustive searching" than "reduction". On the other hand the rule treated as an expansion (right to left, called η-expansion) makes the system divergent and restrictions needed to ensure normalization are cumbersome.

For all these reasons we will use equation (9), called weak η-equation, as a basis for our rewrite rule. This equation is only "half" as strong as the uniqueness requirement (8) (see Section 7.1.4).

Lemma 6.1.2. *The following equation is a consequence of* Φ_L*.*

$$f \ \texttt{.} \ \texttt{<}i_1 \ \texttt{:} \ g_1\texttt{;} \ \ldots\texttt{;} \ i_n \ \texttt{:} \ g_n\texttt{>} \ = \ \texttt{<}i_1 \ \texttt{:} \ f \ \texttt{.} \ g_1\texttt{;} \ \ldots\texttt{;} \ i_n \ \texttt{:} \ f \ \texttt{.} \ g_n\texttt{>} \tag{9}$$

Proof. This is an instance of Theorem 7.1.8, but for illustration we present a short direct proof in terms of LC.

We start with the left hand side

$$f \ \texttt{.} \ \texttt{<}i_1 \ \texttt{:} \ g_1\texttt{;} \ \ldots\texttt{;} \ i_n \ \texttt{:} \ g_n\texttt{>}$$

which by axiom (8) equals

$$\begin{gathered}\texttt{<}i_1 \ \texttt{:} \ f \ \texttt{.} \ \texttt{<}i_1 \ \texttt{:} \ g_1\texttt{;} \ \ldots\texttt{;} \ i_n \ \texttt{:} \ g_n\texttt{>} \ \texttt{.} \ \texttt{F\_PR}(ld, i_1)\texttt{;} \ \ldots\texttt{;} \\ i_n \ \texttt{:} \ f \ \texttt{.} \ \texttt{<}i_1 \ \texttt{:} \ g_1\texttt{;} \ \ldots\texttt{;} \ i_n \ \texttt{:} \ g_n\texttt{>} \ \texttt{.} \ \texttt{F\_PR}(ld, i_n)\texttt{>}\end{gathered}$$

Then we rewrite by axiom (7) each component of the record (with the help of associativity of composition)

$$\texttt{<}i_1 \ \texttt{:} \ f \ \texttt{.} \ g_1\texttt{;} \ \ldots\texttt{;} \ i_n \ \texttt{:} \ f \ \texttt{.} \ g_n\texttt{>}$$

obtaining the right hand side. □

The reduction system $\mathcal{R}_L$ is based on theory Φ_L. We define $\mathcal{R}_L$ by providing the following basic rewrite rules.

$$f \ \texttt{.} \ \texttt{F\_ID}(d) \ \rightarrow \ f \tag{10}$$
$$\texttt{F\_ID}(c) \ \texttt{.} \ g \ \rightarrow \ g \tag{11}$$
$$f \ \texttt{.} \ (g_1 \ \texttt{.} \ g_2) \ \rightarrow \ (f \ \texttt{.} \ g_1) \ \texttt{.} \ g_2 \tag{12}$$
$$\texttt{<}i \ \texttt{:} \ f_i\texttt{;} \ \ldots\texttt{>} \ \texttt{.} \ \texttt{F\_PR}(ld, i) \ \rightarrow \ f_i \tag{13}$$
$$f \ \texttt{.} \ \texttt{<}i \ \texttt{:} \ g\texttt{;} \ \ldots\texttt{>} \ \rightarrow \ \texttt{<}i \ \texttt{:} \ f \ \texttt{.} \ g\texttt{;} \ \ldots\texttt{>} \tag{14}$$

Note that the rules are rather numerous, compared with the rules of common rewriting systems modelling programming, such as untyped λ-calculus [11] or system F [62]. This will be even more visible for the subsequent systems we are going to describe in the thesis. The reason is that in our systems we insist on providing a separate set of combinators with associated rules for every fundamental programming mechanism, while the conventional calculi often rely on rich meta-theory to describe certain notions (substitution, environments) keeping the set of the proper language's combinators small.

If product operations are explicitly introduced into a syntax of some variant of λ-calculus then surely a form of rule (13), which we call a β-rule for products, can be found in the calculus as well. When we consider the rules for substitution and variants of η-rules as belonging to λ-calculus or system F, instead of being a part of the meta-language, the difference in the number of rules decreases even more.

Observation 6.1.3. *$\mathcal{R}_L$ is sound with respect to Φ_L.*

We would like our languages to be strongly normalizing [38], if only possible, so that the compiler designers are free in the choice of evaluation strategies. Another reason we insist on strong normalization for basic languages is that proofs of program termination in extensions to the core languages containing, e.g., general recursion (see Section 8.2) are simpler when there are no additional sources of divergence to consider. We also aim at confluence [94] so that different evaluation orders are not only equally convergent but result in the same normal form.

Theorem 6.1.4. *$\mathcal{R}_L$ is strongly normalizing.*

Proof. This result is proved in [68] for a somewhat more complicated system with binary products. We will need to extend our proof many times — whenever our rewriting system is extended — therefore we propose another, simpler and easily extendable proof.

We define a lexicographic path ordering [38] on terms by assigning precedences to term constructors and comparing subterms lexicographically (from right to left). Such orderings are incremental, that is when we extend the precedence ordering, the lexicographic path ordering is also extended. Since the ordering is closed under substitution and context application, we are able to demonstrate that $\mathcal{R}_L$ is contained in that ordering by showing that each individual rule of $\mathcal{R}_L$ is contained in it. Since the proposed ordering is well-founded, $\mathcal{R}_L$ is strongly normalizing.

The ordering is given by the following conditions. The first five conditions (numbered with roman numerals) define our variant of lexicographic path ordering, while the remaining list of conditions (one-element; numbered with arabic numerals) sets the precedences of term constructors for $\mathcal{R}_L$ and will be extended when we extend $\mathcal{R}_L$. Together with each condition we list the names of rules validated by the condition. Term s is bigger than term t if and only if one of the following conditions holds.

I. Term t is a subterm of s (10, 11, 13).

II. The root constructor of s has higher precedence than that of t, and s is bigger than all subterms of t (14).

III. The precedences of the root constructors of s and t are equal, s is bigger than all subterms of t and the sequence of s subterms, ordered from right to left, is lexicographically bigger than the analogous sequence for term t (12).

IV. Term s is a `funct`-term and t is a `cat`-term (14, though the `cat`-term is hidden in the syntactic sugar).

V. Term s is a `trans`-term and t is a `funct`-term (this condition will be needed for most of the extensions of $\mathcal{R}_L$).

1. Functor composition has higher precedence than functor record (14).

□

Theorem 6.1.5. *$\mathcal{R}_L$ is confluent.*

Proof. By inspection of critical pairs, which are all easily proved solvable, and then by the Newman lemma [38]. □

6.1.3 Equality testing in *LC*

An instance of *LC* will be used, in particular, as the language of kinds and types for the programming language p-Core. Therefore to allow for type-checking of p-Core programs it is essential that there is an equality testing procedure for the morphisms of the initial *LC* constructed from terms.

Problems with η-rules

As mentioned earlier the rules obtained by directing the uniqueness requirement for products (8), are problematic. Moreover, both the η-rules often complicate the reduction as well as the resulting term.

We first illustrate this for η-expansion. Let lc = i `-` `*`; j `-` `*` and ld = k_1 `-` `<`lc`>;` k_2 `-` `*`. Let p be projection `F_PR`(ld, k_1). Consider the following term.

p `.` `F_ID(<`lc`>)`

This term may be simplified in one step to the projection p, but instead we will use the η-expansion and expand the identity at the right hand side of the composition, obtaining

$$p \texttt{ . F\_RECORD(<}lc\texttt{>, } i \texttt{ : F\_ID(<}lc\texttt{>) . F\_PR(}lc, i\texttt{); } j \texttt{ : F\_ID(<}lc\texttt{>) . F\_PR(}lc, j\texttt{))}$$

which may then be simplified to

$$p \texttt{ . F\_RECORD(<}lc\texttt{>, } i \texttt{ : F\_PR(}lc, i\texttt{); } j \texttt{ : F\_PR(}lc, j\texttt{))}$$

Now the term is not only bigger than the original but it is also not reducible to a projection, unless our system features not only η-expansion but η-contraction as well. The term may still be rewritten to the η-expansion of the projection p

$$\texttt{F\_RECORD(<}ld\texttt{>, } i \texttt{ : } p \texttt{ . F\_PR(}lc, i\texttt{); } j \texttt{ : } p \texttt{ . F\_PR(}lc, j\texttt{))}$$

in several steps; we have to η-expand the whole term and then β-contract both subterms.

That η-contraction is ambiguous, defeats confluence and can enlarge terms is best seen on example `funct`-terms with trivial targets such as:

```
<>
```

which can be written in a more verbose form as, for instance:

$$\texttt{F\_RECORD(<}i \texttt{ - *; } j \texttt{ - <}j_1 \texttt{ - <>; } j_2 \texttt{ - <>>>,nil)}$$

where `nil` is the empty indexed list and, this time, `<>` is a shorthand for `C_PP(nil)`. One of the many (some of them very large) η-contractions of this term is the following.

$$\texttt{F\_PR(}i \texttt{ - *; } j \texttt{ - <}j_1 \texttt{ - <>; } j_2 \texttt{ - <>>, } j\texttt{) . F\_PR(}j_1 \texttt{ - <>; } j_2 \texttt{ - <>, } j_1\texttt{)}$$

Another one is given below:

$$\texttt{F\_PR(}i \texttt{ - *; } j \texttt{ - <}j_1 \texttt{ - <>; } j_2 \texttt{ - <>>, } i\texttt{) . F\_RECORD(*,nil)}$$

And yet another one follows:

$$\texttt{<}i \texttt{ : <}i \texttt{ : <>>> . F\_PR(}i \texttt{ - <}i \texttt{ - <>>, } i\texttt{) . F\_PR(}i \texttt{ - <>, } i\texttt{)}$$

Performing equality testing

As shown in Section 2.4 of [38], a finite, normalizing and confluent system $\mathcal{R}$ can be used as an equality testing procedure for $\overline{\mathcal{R}}$. In other words such $\mathcal{R}$ simplifies terms as much as possible without violating $\overline{\mathcal{R}}$. Since the reduction system $\mathcal{R}_L$ is strongly normalizing and confluent one might expect that $\mathcal{R}_L$-rewriting is enough to decide equality. Unfortunately $\mathcal{R}_L$ lacks the other "half" of the uniqueness requirement (8); Φ_L is not the theory of $\mathcal{R}_L$.

Let Φ_H be the theory generated by the following equation.

$$\texttt{F\_RECORD}(c,\ i_1 : \texttt{F\_PR}(lc, i_1);\ \ldots;\ i_n : \texttt{F\_PR}(lc, i_n)) \quad = \quad \texttt{F\_ID}(c)$$

The instantiations of the equation involve all the finite subsets $i_1, \ldots, i_n$ of the set of labels. Theory Φ_H is obviously contained in Φ_L.

Lemma 6.1.6. $\overline{\mathcal{R}_L} \nvdash \Phi_H$*, while* $\overline{\mathcal{R}_L} \cup \Phi_H \vdash \Phi_L$ *(where the consequence relation between sets of equations is the usual equational consequence relation).*

Proof. This is a special case of Theorem 7.1.10 and Theorem 7.1.9. □

The equality testing procedure `eq_funct` given below makes up for the lack of the full uniqueness requirement (8) in $\mathcal{R}_L$ (or, more precisely, the lack of the axioms of Φ_H). When two $\mathcal{R}_L$-normal form terms to be compared are suspected of being equal due to the uniqueness requirement (8), all of their compositions with projections are compared instead, each using the `eq_pi` operation that again manages to sidestep the full uniqueness requirement.

```
(* Library functions are explained in Section 3.2.
   Arguments [f] and [g] are in normal form
   and must have the same sources and the same targets. *)
let rec eq_funct f g =
  match f, g with
  | F_ID _, F_ID _ -> true
  | F_COMP (f1, f2), F_COMP (g1, g2) ->
      eq_funct f1 g1 && eq_funct f2 g2
  | F_PR (_, i), F_PR (_, j) when IdIndex.eq i j -> true
  | F_RECORD (_, lf), F_RECORD (_, lg) -> eqset eq_funct lf lg
  | _, F_RECORD (d, lg) ->
      bforall (eq_pi d [] f) lg
  | F_RECORD (c, lf), _ ->
      bforall (eq_pi c [] g) lf
  | _ -> is_trivial (trg f)
(* [eq_pi] gives [true] if and only if
   [f] composed with projections at labels of [li]
   (in reverse order) and the projection at [i]
   (as the last one) is equal to [g] *)
and eq_pi d li f (i, g) =
  assert (EqFCat.eq d (src f) && EqFCat.eq d (src g));
  (* and [f] is not a record
     and [i] and [li] result in type-correct projections *)
  match g with
  | F_PR (_, j) when IdIndex.eq i j ->
      li = [] && eq_funct f (F_ID d)
  | F_COMP (f', F_PR (_, j)) when IdIndex.eq i j ->
      (match li with
      | [] -> eq_funct f f'
      | i1 :: r -> eq_pi d r f (i1, f'))
  | F_RECORD (_, lf) ->
      let ili = i :: li in
      bforall (eq_pi d ili f) lf
  | _ -> false
and is_trivial c =
  match c with
  | C_BB -> false
  | C_PP lc -> vforall is_trivial lc
```

Lemma 6.1.7. *The equality testing procedure* `eq_funct` *terminates.*

Proof. In the code of mutually recursive functions `eq_funct` and `eq_pi` the recursive applications are always to subterms of arguments `f` and `g` and always at least one of the subterms is proper. The exception is the application `eq_funct f (SemLFunct.f_ID c)`, which terminates in one step by either the first or the last case of `eq_funct`. This application never falls into the sixth case, where `f` is a record, because `eq_pi` is never applied to a record as its third argument: `eq_funct` does not apply `eq_pi` to a record, because if both arguments to `eq_funct` are records then fourth case would be taken, not fifth or sixth; `eq_pi` does not apply `eq_pi` to a record, because it always passes the third argument unchanged. □

Notice that we do not mention sources nor targets of `f` and `g` in our argument (and we do not even use the fact that the terms are in normal form). Consequently, we know that `eq_funct` terminates even for arguments `f` and `g` with different sources or targets (if only the assertion is removed). This implies that also an (extended) variant of `eq_funct` in Appendix A.1.5 terminates.

Observation 6.1.8. *Assuming that the recursive calls of* `eq_funct` *compute correct equality of* `funct`*-terms in the initial LC,* `eq_pi` *behaves as stated in its specification in the comment. In particular, assuming* `f` *is not a record,* `eq_pi d [] f (i, g)` *is true if and only if* `f` *composed with a projection at* `i` *is equal to* `g` *in the initial LC.*

Theorem 6.1.9. *The equality testing procedure* `eq_funct` *is sound and complete with respect to the initial LC, for arguments that are* $\mathcal{R}_L$*-normal form (ground) terms with the same source and the same target. In other words: two such terms are determined by* `eq_funct` *to be equal if and only if they are equal in the initial LC.*

Proof. (Sketch.) Structural induction over the pair $(\mathtt{f}, \mathtt{g})$ with the use of the property that subterms of normal form terms are in normal form.

The only nontrivial cases of the main `match`...`with` expression in the code of `eq_funct` are the last three. The last one needs no inductive hypotheses, because the uniqueness requirement (8) allows us to equate all terms having the targets accepted by the auxiliary function `is_trivial`. On the other hand, if the target is not trivial, the only differences of root constructors between equal terms in normal forms, allowed by the lack of the uniqueness requirement (8) in $\mathcal{R}_L$, are captured by the fifth and sixth cases (when exactly one of f and g is of the form `F_RECORD(_, _)`).

We know that the target of the compared terms in the fifth and sixth cases is a limit (the product `C_PP`). The indexed lists of terms (`lg` and `lf`, respectively) under `F_RECORD` is a cone over the diagram of the limit. The category is an initial algebra, so every morphism into the limit is the unique morphism corresponding to a cone and its cone can be recovered by composing the morphism with the limiting cone. Thus, to compare two morphisms into the limit it is enough to compare their corresponding cones, one being `lg` or `lf` and the other recovered using projections in `eq_pi`. The comparison of individual components of the cones is correctly performed in `eq_pi`, by Observation 6.1.8. □

Extending `eq_funct` to languages with datatypes such as products, sums, etc. is easy. Such extensions bring no new η-equations at the level of functors and so their reduction systems suffice to reduce `funct`-terms to unique normal forms up to the η-equation (8). Appendix A.1.5 contains the variant of the `eq_funct` procedure that accepts terms of the full internal core language, and some further remarks about the implementation of the procedure.

6.1.4 Rewriting in *2-LC*

In this section we build the equational theory and a reduction system for *2-LC*, based on those for *LC*. From now on the vertical composition of *2-LC*, denoted by $\mathtt{T\_comp}(t, u)$, will usually be written t `.` u (like the composition of functors). At the same time the horizontal composition of transformations $\mathtt{T\_COMP}(t, u)$ will be written $t * u$ to be easily distinguished from the other two compositions. Also the two multiplications by a functor; $\mathtt{T\_FT}(f_1, t_2)$ and $\mathtt{T\_TF}(t_1, f_2)$, will be written as $f_1 * t_2$ and $t_1 * f_2$, respectively. Sometimes we will also write "$: f$" for the 2-identity $\mathtt{T\_id}(f)$.

Equations

The definition of a `funct`-equation in *2-LC* is strictly analogous to the definition of a `funct`-equation in *LC*. The only difference is that the language and the typing are that of *2-LC* instead of *LC*. The definition of a `trans`-equation is similar, so we will only sketch the main difference. In addition to the requirement that both sides of a `trans`-equation are well-typed (in every *2-LC*) and have equal source and target categories, we require the domain functor of the left hand side of the equation to be equal to the domain of the right hand side (in Φ_L), and the same for the codomains. Thanks to the equality testing procedure developed in the previous section, both the type-checking of `trans`-terms and the equality of `funct`-terms can be decided (for the initial model; for extended models they have to be extended, too, and for various exotic models they would have to be modified).

We start with axioms stating that $C(c, e)$ (the categories of functors and transformations with vertical composition) are indeed categories.

$$\begin{aligned} t \;.\; \mathtt{T\_id}(g) &= t && (15)\\ \mathtt{T\_id}(f) \;.\; t &= t && (16)\\ t \;.\; (u_1 \;.\; u_2) &= (t \;.\; u_1) \;.\; u_2 && (17)\end{aligned}$$

Then we present the axioms specific to the 2-categorical structure:

$$\begin{aligned} \mathtt{T\_ID}(c) &= \mathtt{T\_id}(\mathtt{F\_ID}(c)) && (18)\\ \mathtt{T\_id}(f) * \mathtt{T\_id}(g) &= \mathtt{T\_id}(f \;.\; g) && (19)\\ (t_1 \;.\; t_2) * (u_1 \;.\; u_2) &= (t_1 * u_1) \;.\; (t_2 * u_2) && (20)\end{aligned}$$

In equation (20), called the interchange law, the most general typing of the meta-variables making both sides type-correct is: $t_1 : f_1 \to f_2$, $t_2 : f_2 \to f_3$, $u_1 : g_1 \to g_1$, $u_2 : g_2 \to g_3$, $f_i : a \to b$, $g_i : b \to c$. With such typing the equation holds in any 2-category.

The axioms defining multiplications by a functor look in the original notation as follows.

$$\begin{aligned} \mathtt{T\_FT}(f_1, t_2) &= \mathtt{T\_COMP}(\mathtt{T\_id}(f_1), t_2) && (21)\\ \mathtt{T\_TF}(t_1, f_2) &= \mathtt{T\_COMP}(t_1, \mathtt{T\_id}(f_2)) && (22)\end{aligned}$$

The same two equations can also be written in the slightly ambiguous but shorter infix notation.

$$\begin{aligned} f_1 * t_2 &= \mathtt{T\_id}(f_1) * t_2\\ t_1 * f_2 &= t_1 * \mathtt{T\_id}(f_2)\end{aligned}$$

To assert that the underlying category is an LC we hereby incorporate the `funct`-axioms 4–8. Then we state that the category of objects and 2-morphisms is an LC, too, using equations analogous to those for the underlying category.

$$\begin{aligned} t * \mathtt{T\_ID}(d) &= t & (23)\\ \mathtt{T\_ID}(c) * u &= u & (24)\\ t * (u_1 * u_2) &= (t * u_1) * u_2 & (25)\\ \mathtt{<}i \mathtt{\ =\ } t_i\mathtt{;}\ \ldots\mathtt{>} * \mathtt{T\_PR}(ld, i) &= t_i & (26)\\ \mathtt{<}i \mathtt{\ =\ } t * \mathtt{T\_PR}(ld, i)\mathtt{;}\ \ldots\mathtt{>} &= t & (27) \end{aligned}$$

A part of the product compatibility condition, namely that `T_id` preserves product objects "on the nose", follows from the choice of domains and codomains for transformations denoted by *2-LC* terms (see Section 4.2.2). The requirement that `T_id` preserves projections "on the nose", is postulated here:

$$\mathtt{T\_id}(\mathtt{F\_PR}(lc, i)) = \mathtt{T\_PR}(lc, i) \qquad (28)$$

Let Φ_2 be the theory generated by equations 4–8 and 15–28. This theory contains both `funct` and `trans`-equations that interact when the theory is being generated from the axioms. However, the `funct` equations in Φ_2 coincide with those of Φ_L, regardless of the interaction, because no new `funct`-axioms have been added and the `trans`-axioms of Φ_2 are well typed with respect to Φ_L, by definition.

Lemma 6.1.10. *Φ_2 is the theory of 2-LC.*

Proof. The argument amounts to merging the proof of Lemma 6.1.1 with a folklore knowledge about Godement Calculus [82] and an observation that axiom (28) ensures the compatibility of the products. □

Reduction

We will give a reduction system $\mathcal{R}_2$ for the theory Φ_2. This time we do not aim at deciding equality, but merely want to simplify terms. Later on we will try to assess how complete the simplifications is. Just as Φ_2 contains Φ_L, the system $\mathcal{R}_2$ contains $\mathcal{R}_L$. After introducing rules 29–43 that together with the rules of $\mathcal{R}_L$ constitute $\mathcal{R}_2$ we will observe various properties of the whole reduction system $\mathcal{R}_2$.

The following rules take care of the most complicated part of *2-LC* — the category of objects and 2-morphisms. First, we reduce the horizontal composition to the multiplications and vertical composition. We require that the `funct`-meta-variable f in the rule below is the domain of the `trans`-meta-variable t and that h is the codomain of u. In contrast to the interchange law 20, from which the rule derives, this typing is not implied by the type-correctness of the equation, while only such typing guarantees the soundness of the rule. (However, by adding spurious identities, the right typing can be imposed. Only minor modifications are then required to the rest of the reduction system, which becomes, in a sense, type-free.)

$$t * u \;\rightarrow\; (f * u)\ \texttt{.}\ (t * h) \tag{29}$$

Now we present the rules for the operation of multiplication from the left (`T_FT`).

$$\begin{aligned}
f * \texttt{<}i \texttt{ = } u\texttt{; ...>} &\;\rightarrow\; \texttt{<}i \texttt{ = } f * u\texttt{; ...>} & (30)\\
f * \texttt{T\_id}(g) &\;\rightarrow\; \texttt{T\_id}(f\ \texttt{.}\ g) & (31)\\
f * (u_1\ \texttt{.}\ u_2) &\;\rightarrow\; (f * u_1)\ \texttt{.}\ (f * u_2) & (32)
\end{aligned}$$

Below are the rules for the multiplication from the right (`T_TF`).

$$\begin{aligned}
t * \texttt{F\_ID}(d) &\;\rightarrow\; t & (33)\\
t * (f\ \texttt{.}\ g) &\;\rightarrow\; (t * f) * g & (34)\\
\texttt{<}i \texttt{ = } t_i\texttt{; ...>} * \texttt{F\_PR}(ld, i) &\;\rightarrow\; t_i & (35)\\
\texttt{T\_id}(f) * \texttt{F\_PR}(ld, i) &\;\rightarrow\; \texttt{T\_id}(f\ \texttt{.}\ \texttt{F\_PR}(ld, i)) & (36)\\
(t_1\ \texttt{.}\ t_2) * \texttt{F\_PR}(ld, i) &\;\rightarrow\; (t_1 * \texttt{F\_PR}(ld, i))\ \texttt{.}\ (t_2 * \texttt{F\_PR}(ld, i)) & (37)\\
t * \texttt{<}i \texttt{ : } g\texttt{; ...>} &\;\rightarrow\; \texttt{<}i \texttt{ = } t * g\texttt{; ...>} & (38)
\end{aligned}$$

The last two rules concerning the category of objects and 2-morphisms describe the elimination of the combinators `T_ID` and `T_PR`.

$$\begin{aligned}
\texttt{T\_ID}(c) &\;\rightarrow\; \texttt{T\_id}(\texttt{F\_ID}(c)) & (39)\\
\texttt{T\_PR}(lc, i) &\;\rightarrow\; \texttt{T\_id}(\texttt{F\_PR}(lc, i)) & (40)
\end{aligned}$$

The rules for rewriting of the morphisms of the $C(c, e)$ categories conclude our definition of the reduction system $\mathcal{R}_2$.

$$\begin{aligned}
t\ \texttt{.}\ \texttt{T\_id}(g) &\;\rightarrow\; t & (41)\\
\texttt{T\_id}(f)\ \texttt{.}\ t &\;\rightarrow\; t & (42)\\
t\ \texttt{.}\ (u_1\ \texttt{.}\ u_2) &\;\rightarrow\; (t\ \texttt{.}\ u_1)\ \texttt{.}\ u_2 & (43)
\end{aligned}$$

The following theorem, together with confluence and strong normalization for $\mathcal{R}_2$ proved later on, tells us that normal forms of $\mathcal{R}_2$ do not contain horizontal composition nor multiplications by a functor. This property will hold for all our extensions of $\mathcal{R}_2$, though we will not repeat nor extend the proofs for the extended systems. From the computational perspective, this property implies that compiled programs can be correctly executed with their types wiped out (though when the `T_map` term constructor is introduced in one of the extensions, the implication is not valid anymore).

Definition 6.1.11. *We will call a ground* `trans`*-term concrete if it can be* $\mathcal{R}_2$*-reduced to a term not containing a horizontal composition nor a multiplication by a functor.*

Theorem 6.1.12. *Every term is concrete.*

Proof. The proof is by structural induction using the two lemmas below.

Let t be a `trans`-term. If t is a multiplication we carry the inductive steps using induction hypothesis and Lemma 6.1.14 or Lemma 6.1.15. If t is a horizontal composition then using rule (29) we reduce the proof to the two proofs concerning multiplication. If t is of any other form, the inductive step follows straightforwardly from the induction hypothesis. □

Observation 6.1.13. *If there is a horizontal composition or a multiplication at the right hand side of a rule of* $\mathcal{R}_2$*, then there is one of them at the left hand side, as well. Consequently, a term without horizontal composition and multiplication will not* $\mathcal{R}_2$*-reduce to a term containing either of them.*

Lemma 6.1.14. *If* t *is concrete then so is* `T_FT`(f, t)*, for any ground term* f*.*

Proof. By induction on the size of t (not on the size of the normal form of t!) using rules 30–32.

The possible root constructors of the term t are either immediately covered by rules 30–32 or t is reducible in one step to identity. As an example we will perform an inductive step corresponding to rule (32). The transformations u_1 and u_2 are proper subterms of t and therefore we can use the induction hypothesis to conclude that the two multiplications are concrete. Therefore, by Observation 6.1.13 their vertical composition is concrete, too. Hence we deduce that t was concrete. □

Lemma 6.1.15. *If* t *is concrete then so is* `T_TF`(t, f)*, for any ground term* f*.*

Proof. By induction on the size of f and in the case of `F_PR` on the size of t. The rules to analyze are 33–38 and they cover each possible form of f. When f is a projection, then if t is `T_ID` or `T_PR`, additional one-step $\mathcal{R}_2$-reduction to `T_id` is enough to fall within the cases covered by the rules. □

Theorem 6.1.16. *System* $\mathcal{R}_2$ *is strongly normalizing.*

Proof. We extend the proof of Theorem 6.1.4 for the cases of the additional rewriting rules. The additional assignments of precedences to term constructors (together with names of affected rules) are the following.

2. Horizontal composition has the highest precedence, equal to the precedence of multiplications (29).

3. Multiplication by a functor from the left has the same highest precedence as horizontal composition (30–32).

4. Multiplication by a functor from the right has the same highest precedence as horizontal composition (36–38).

5. Both `T_ID` and `T_PR` have precedences higher than the precedence of `T_id` (39–40).

Rules (33), (35), (41) and (42) belong to the ordering by condition I from the proof of Theorem 6.1.4. Rules (34) and (43) belong to the ordering by condition III and rules (29) and 39–40 by condition V. □

Theorem 6.1.17. *System $\mathcal{R}_2$ is confluent.*

Proof. By inspection of critical pairs. There are many of them, but all are quite easily solved. For example, the critical pair caused by rules (34) and (14)

$$(t * f) * \texttt{<}i\ \texttt{:}\ g\texttt{;}\ \ldots\texttt{>} \leftarrow t * (f\ \texttt{.}\ \texttt{<}i\ \texttt{:}\ g\texttt{;}\ \ldots\texttt{>}) \rightarrow t * \texttt{<}i\ \texttt{:}\ f\ \texttt{.}\ g\texttt{;}\ \ldots\texttt{>}$$

can be solved by rewriting the left hand side using rule (38)

$$(t * f) * \texttt{<}i\ \texttt{:}\ g\texttt{;}\ \ldots\texttt{>} \rightarrow \texttt{<}i\ \texttt{=}\ (t * f) * g\texttt{;}\ \ldots\texttt{>}$$

and the right hand side using rule (38) and then rule (34)

$$t * \texttt{<}i\ \texttt{:}\ f\ \texttt{.}\ g\texttt{;}\ \ldots\texttt{>} \rightarrow \texttt{<}i\ \texttt{=}\ t * (f\ \texttt{.}\ g)\texttt{;}\ \ldots\texttt{>} \rightarrow \texttt{<}i\ \texttt{=}\ (t * f) * g\texttt{;}\ \ldots\texttt{>}$$

arriving at the same term. □

Observation 6.1.18. *System $\mathcal{R}_2$ is sound with respect to theory Φ_2.*

System $\mathcal{R}_2$ almost encompasses the whole theory Φ_2. Even the full interchange law, that is axiom (20), belongs to $\overline{\mathcal{R}_2}$. If $\mathcal{R}_2$ didn't lack (similarly as $\mathcal{R}_L$) the full uniqueness requirement for the labeled products, one could even decide equality by $\mathcal{R}_2$-reducing them to normal forms and then comparing. As it is, the following important rule is not derivable in $\mathcal{R}_2$.

$$\texttt{<}i\ \texttt{=}\ t\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ \texttt{<}i\ \texttt{=}\ u\texttt{;}\ \ldots\texttt{>} \quad\rightarrow\quad \texttt{<}i\ \texttt{=}\ t\ \texttt{.}\ u\texttt{;}\ \ldots\texttt{>} \tag{44}$$

The rule is valid in Φ_2 but not in $\overline{\mathcal{R}_2}$ and, as one might guess, the rule is one of the consequences of the uniqueness requirement.

If we consider a record in the category of objects and 2-morphisms as a tool for grouping together several layers of a single programming module (in the spirit of the remark on page 118), the meaning of rule (44) may become more understandable. The layers would be types, values, specification, optimization hints for linking with other modules, etc. The lack of rule (44) in system $\mathcal{R}_2$ means that we can not link two modules together by just composing them vertically.

However, this is not a major obstacle to using such modules, because those components that indeed require vertical composition, like the value parts, can be extracted from the two modules using a suitable projection F_PR and then composed. Next we may get the two morphisms representing optimization hints and merge them together, probably in some other way than composing, and so on. Finally, we can group the resulting layers together into a single module using the record operation. The operation of composition of such modules is no longer internal to the language of *2-LC*, but its results are still expressible in *2-LC*.

6.1.5 Rewriting in *2-LC-lc*

In this section we extend our equational theory and reduction system to the whole structure of *2-LC-lc*. Remember that the product F_pp(c, lf) is written in the sugared notation as

$$\{i_1 : f_1;\ i_2 : f_2;\ \ldots;\ i_n : f_n\}$$

and the record T_record(f, lt) as

$$\{i_1 = t_1;\ i_2 = t_2;\ \ldots;\ i_n = t_n\}$$

There is no special notation for the transformation product T_pp(c, lt); however, we can use the sugared notation for indexed lists to present its operands as follows

$$\texttt{T\_pp}(c,\ i_1 = t_1;\ i_2 = t_2;\ \ldots;\ i_n = t_n)$$

Equations

The definition of an equation is analogous to that for *2-LC*. This time the funct sort of the developed theory will not be just the theory of *LC*, because the structure of the underlying category is richer. To the axioms of *2-LC* we add an additional funct-equation analogous to equation (9), which is justified by the fact that F_pp(c, lf) can be thought of as a composition of a record F_RECORD(c, lf) and an adjoint (see Section 7.1.3).

$$f\ .\ \{i : g;\ \ldots\} \quad = \quad \{i : f\ .\ g;\ \ldots\} \tag{45}$$

In the same way projection T_pr(lg, j) can be seen as a multiplication of a record F_RECORD(c, lg) and a co-unit of an adjunction, hence the following axiom.

$$f * \texttt{T\_pr}(i : g;\ \ldots, j) \quad = \quad \texttt{T\_pr}(i : f\ .\ g;\ \ldots, j) \tag{46}$$

Next we add the trans-axiom defining the T_pp operation as the action of the product functor on morphisms.

$$\texttt{T\_pp}(c, lt) \quad = \quad \texttt{T\_RECORD}(c, lt) * \texttt{F\_pp}(\texttt{<}le\texttt{>},\ i : \texttt{F\_PR}(le, i);\ \ldots) \tag{47}$$

Last we add the existence and uniqueness requirements for the product.

$$\texttt{\{}i\texttt{ = }t_i\texttt{; }\ldots\texttt{\} . T\_pr}(lg, i) = t_i \qquad (48)$$

$$\texttt{\{}i\texttt{ = }t\texttt{ . T\_pr}(lg, i)\texttt{; }\ldots\texttt{\}} = t \qquad (49)$$

Let Φ_p denote the theory induced by Φ_2 together with axioms 45–49

Lemma 6.1.19. *Φ_p is the theory of 2-LC-lc.*

Proof. (Sketch.) The new axioms in *2-LC-lc* are analogues of equations (7) and (8) for *LC*, complemented by equations implied by the particular choice of the notation for the product adjunction (see Section 7.1.3). On the other hand *2-LC-lc* is *2-LC* with *LC* structures as the categories $C(c, e)$, so the thesis follows from the analogous theorems for *2-LC* and *LC* (Lemma 6.1.10 and Lemma 6.1.1). □

Remark. If we wanted to describe *2-LC-lc* with explicit compound products (see Section 4.3.1), we would need to express the product interchange law by a set equations. We would use the following `funct`-equation:

$$\texttt{\{}i\texttt{ : }f\texttt{; }\ldots\texttt{\} . F\_PR}(ld, j) = \texttt{\{}i\texttt{ : }f\texttt{ . F\_PR}(ld, j)\texttt{; }\ldots\texttt{\}} \qquad (50)$$

a `trans`-equation concerning both kinds of projections:

$$\texttt{T\_pr(}i\texttt{ : }f\texttt{; }\ldots, k) * \texttt{F\_PR}(ld, j) = \texttt{T\_pr(}i\texttt{ : }f\texttt{ . F\_PR}(ld, j)\texttt{; }\ldots, k) \qquad (51)$$

and a `trans`-equation about record:

$$\texttt{\{}i\texttt{ = }t\texttt{; }\ldots\texttt{\}} * \texttt{F\_PR}(ld, j) = \texttt{\{}i\texttt{ = }t * \texttt{F\_PR}(ld, j)\texttt{; }\ldots\texttt{\}} \qquad (52)$$

Note, however, that in *2-LC-lc* with explicit compound products these equations need to hold only for expressible valuations of the meta-variables!

One of the consequences of these equations shows why the semantic condition in Section 4.3.1 has been named the product interchange law.

$$\texttt{\{}i\texttt{ = <}j\texttt{ = }t\texttt{; }\ldots\texttt{>; }\ldots\texttt{\}} = \texttt{<}j\texttt{ = \{}i\texttt{ = }t\texttt{; }\ldots\texttt{\}; }\ldots\texttt{>}$$

A version without syntactic sugar is much less readable but should prevent the false impression that the left and the right hand sides are textually identical.

$$\texttt{T\_record(F\_RECORD(}c, j\texttt{ : }f_j\texttt{; }\ldots), i\texttt{ = T\_RECORD(}c, j\texttt{ = }t_{i,j}\texttt{; }\ldots\texttt{); }\ldots) =$$
$$\texttt{T\_RECORD(}c, j\texttt{ = T\_record(}f_j, i\texttt{ = }t_{i,j}\texttt{; }\ldots\texttt{); }\ldots)$$

Reduction

Operation T_pp, expressing the action of the product functor on morphisms, is not intended to appear in normal form terms. Axiom (47) presents the operation using the other term constructors. One of the operations used in the axiom is multiplication by a functor, which itself should not appear in normal form terms if we are to base our reduction system on $\mathcal{R}_2$. Since the reduction system will not be strong enough to eliminate multiplications in such contexts, we have to describe the T_pp operation in terms of the basic labeled product operations.

Lemma 6.1.20. *The following equation is a consequence of* Φ_p.

$$\texttt{T\_pp}(c,\, i \texttt{ = } t\texttt{;}\ \ldots) \;=\; \texttt{\{}i \texttt{ = } \texttt{T\_pr}(\mathit{lf}, i)\ \texttt{.}\ t\texttt{;}\ \ldots\texttt{\}} \tag{53}$$

Proof. This is a special case of Theorem 7.1.13 to be proved in Section 7.1.4. A much longer direct proof by equational reasoning in Φ_p is given in Appendix B. □

We extend the reduction system $\mathcal{R}_2$ by new rules corresponding to the axioms of Φ_p. The first rule below is a funct-rule.

$$f\ \texttt{.}\ \texttt{\{}i\ \texttt{:}\ g\texttt{;}\ \ldots\texttt{\}} \;\to\; \texttt{\{}i\ \texttt{:}\ f\ \texttt{.}\ g\texttt{;}\ \ldots\texttt{\}} \tag{54}$$

The rest are trans-rules. First, we show how to get rid of the product of transformations.

$$\texttt{T\_pp}(c,\, i \texttt{ = } t\texttt{;}\ \ldots) \;\to\; \texttt{\{}i \texttt{ = } \texttt{T\_pr}(\mathit{lf}, i)\ \texttt{.}\ t\texttt{;}\ \ldots\texttt{\}} \tag{55}$$

The existence requirement (48), is just directed from left to right to obtain a β-rule (56) for products. At the same time the uniqueness requirement (49) is weakened as in $\mathcal{R}_L$, resulting in a rule called the weak η-rule for products.

$$\texttt{\{}i \texttt{ = } t_i\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ \texttt{T\_pr}(\mathit{lg}, i) \;\to\; t_i \tag{56}$$

$$t\ \texttt{.}\ \texttt{\{}i \texttt{ = } u\texttt{;}\ \ldots\texttt{\}} \;\to\; \texttt{\{}i \texttt{ = } t\ \texttt{.}\ u\texttt{;}\ \ldots\texttt{\}} \tag{57}$$

We also need extensions to the rules concerning multiplication by a functor from the left (T_FT):

$$f * \texttt{T\_pr}(i\ \texttt{:}\ g\texttt{;}\ \ldots, j) \;\to\; \texttt{T\_pr}(i\ \texttt{:}\ f\ \texttt{.}\ g\texttt{;}\ \ldots, j) \tag{58}$$

$$f * \texttt{\{}i \texttt{ = } u\texttt{;}\ \ldots\texttt{\}} \;\to\; \texttt{\{}i \texttt{ = } f * u\texttt{;}\ \ldots\texttt{\}} \tag{59}$$

and an additional rule concerning multiplication from the right (T_TF):

$$t * \texttt{\{}i\ \texttt{:}\ g\texttt{;}\ \ldots\texttt{\}} \;\to\; \texttt{T\_pp}(c,\, i \texttt{ = } t * g\texttt{;}\ \ldots) \tag{60}$$

Multiplying from the right by a projection functor does not need extension, because the product operations (F_pp, T_pr and T_record) always have target *. The extended system will be called $\mathcal{R}_p$.

Remark. If we were rewriting in *2-LC-lc* with explicit compound products, we would need new cases for composing with projection. Then equations 50–52 from the previous section directed from left to right would do.

Lemma 6.1.21. *$\mathcal{R}_p$ is sound with respect to Φ_p.*

Proof. Soundness of rules 58–60 follow from the uniqueness requirement (49) and interchange law (20). The rest is obvious. □

This time we lack the full uniqueness requirement not only in the underlying category and analogously in the category of objects and 2-morphisms, but also in each of the $C(c, e)$ categories. Let Φ_h, contained in Φ_p, consist of the following equations.

$$\texttt{F\_RECORD}(c,\ i\ :\ \texttt{F\_PR}(lc, i);\ \ldots) \ = \ \texttt{F\_ID}(c) \tag{61}$$
$$\texttt{T\_RECORD}(c,\ i = \texttt{T\_PR}(lc, i);\ \ldots) \ = \ \texttt{T\_ID}(c) \tag{62}$$
$$\texttt{T\_record}(f,\ i = \texttt{T\_pr}(lf, i);\ \ldots) \ = \ \texttt{T\_id}(f) \tag{63}$$

Lemma 6.1.22. *None of equations 61–63 above is derivable in $\overline{\mathcal{R}_p}$, while $\overline{\mathcal{R}_p} \cup \Phi_h \vdash \Phi_p$.*

Proof. (Sketch.) For the second part it suffices to use Theorem 7.1.9 for each of the three kinds of products. For the first part using Theorem 7.1.10 three times might not be enough, because the theories of the three products interact with each other. However, theory Φ_2 almost does not touch terms of target `*`, such as those in equation (63), which allows us to consider the products inside $C(c, *)$ in isolation from the products of the outside. The only other tricky point is that equation (61) for the product in the underlying category might have followed from the theory of the product in the category of objects and 2-morphisms. But the actual symmetry of the relevant portions of $\overline{\mathcal{R}_L}$ and $\overline{\mathcal{R}_2}$ implies that if equation (61) does not follow from one of the theories (by Theorem 7.1.10), it will not follow after the addition of the other. □

Theorem 6.1.23. *System $\mathcal{R}_p$ is strongly normalizing.*

Proof. We extend the proof of Theorem 6.1.16 for the cases of the additional rewriting rules. The additional assignments of precedences to term constructors (together with names of affected rules) are the following.

6. Functor composition has higher precedence than functor product (54).

7. Term constructor `T_pp` has higher precedence than the record, the vertical composition and the projection term constructors (55).

8. Vertical composition has higher precedence than record (57).

Rule (56) belongs to the ordering by condition I from the proof of Theorem 6.1.4. Rules (58) and (59) belong to the ordering by condition 3 of Theorem 6.1.16. and rule (60) by condition 4. □

Theorem 6.1.24. *System $\mathcal{R}_p$ is confluent.*

Proof. $\mathcal{R}_p$ may be presented as a sum of two confluent systems — $\mathcal{R}_2$ and the system for the $C(c, e)$ categories that turns out to be a minor extension of $\mathcal{R}_L$ and consists of rules 54–60 above and the rules for vertical composition 41–43.

Since both the systems are syntax-driven it is easy to check that the three critical pairs spanning both systems (the first involving rules (54) and (34), the second involving rules (56) and (32), the third involving rules (57) and (32)) are easily solved using rules 58–60. For example, the critical pair between rule (56) and rule (32)

$$f * t \leftarrow f * (\texttt{\{}i \texttt{ = } t\texttt{; }\dots\texttt{\} . T\_pr(}lg, i\texttt{)}) \rightarrow (f * \texttt{\{}i \texttt{ = } t\texttt{; }\dots\texttt{\}}) \texttt{ . } (f * \texttt{T\_pr(}lg, i\texttt{)})$$

can be solved by rewriting the right hand side

$$\texttt{\{}i \texttt{ = } f * t\texttt{; }\dots\texttt{\} . } (f * \texttt{T\_pr(}lg, i\texttt{)}) \rightarrow \texttt{\{}i \texttt{ = } f * t\texttt{; }\dots\texttt{\} . T\_pr(}lh, i\texttt{)} \rightarrow f * t$$

with rules (59), (58) and (56).

Since all the inter-system critical pairs are solvable, the sum of these two confluent systems is locally confluent. Since $\mathcal{R}_p$ is also strongly normalizing, it is confluent by the Newman lemma [38]. □

6.1.6 Conclusion

We have constructed the equational theory of *2-LC-lc* (Lemma 6.1.19) and a confluent (Theorem 6.1.24) and strongly normalizing (Theorem 6.1.23) reduction system based on the theory (Lemma 6.1.21) and agreeing with programming intuitions. Together with a terminating (Lemma 6.1.7) and complete (Theorem 6.1.9) equality testing procedure for the 1-morphisms of the initial *2-LC-lc*, this forms a basis for a programming language. The language of *2-LC-lc* type-checked using the equality testing procedure `eq_funct` and evaluated by the reduction system $\mathcal{R}_p$ forms a programming language that we will call p-Core.

We have obtained a system that can be viewed as a (minimal) programming language, because it allows the user to express various values and initiate a nontrivial evaluation process. While in the language of *2-LC* we could at most require a computer to do nothing (`T_id(`f`)`), or perhaps to do nothing in a sequence (`T_id(`f`) . T_id(`f`)`), here we have a working datatype — the product (`{`i `:` f`; `$\dots$`}`) with a record constructor (`{`i `=` t`; `$\dots$`}`), an operation of accessing a field (`T_pr(`lf, i`)`) and a nontrivial computational rule (56).

Moreover, p-Core enables parameterization of types and values by types. For example, the term `F_PR`(i `- *;` j `- *`, j) represents a type parameterized by two types, first of which will always be ignored (rule (13)). Analogously, the term `T_record`(`F_PR`(i `- *;` j `- *`, j), `nil`) represents a trivial tuple parameterized by two types. The result of instantiating this term will be another trivial tuple, as seen in rule (59).

The parameterization facilitates using p-Core as a core language of our module system (Chapter 5). Equipped with a module system and amended with some syntactic sugar (see Section 8.3) p-Core is definitely a high level language, despite its fine-grained control over substitution and type manipulation. The programmer can express not only values, but also generic recipes (parameterization) and, using the module language, even the architecture of the program itself. From now on, most of the extensions to the core language will not be aimed at making it even higher level but at making it stronger and richer computationally. The goal will be to maintain strong normalization (as long as possible) and (local) confluence of the reduction system, while adding constructs and interpreting their theories. The confluence ensures that the user, after writing a program and supplying all the data, can identify the result easily. Just like in $\mathcal{R}_p$, the η-rules and others that are not essential computationally will be left out, while all the rules that really simplify terms will be included.

6.2 s-Core — core language with sum type

In this section we will extend the language p-Core to s-Core featuring a sum type, while in the next section we will extend the language with inductive and coinductive types. Unlike in Charity [54], the sum and product types will not be special cases of the inductive and coinductive constructions. In *2-LC-lc* the product is already present, but the important notion of disjoint sum is not expressible in any way. While some common examples of coinductive types base on a product only, none of the standard examples of inductive types are expressible without sums. For this reason, even though the two extensions are independent, we will start by adding the sum type, partly dual to the already present labeled product.

6.2.1 Definition based on p-Core

The name “sum” is taken from [39] to mean distributive coproduct. In the presence of exponential objects, coproducts always distribute over products. But as in this simple variant of the core language there is no exponent and because the distributivity seems to be a basic and computationally indispensable concept on its own, we impose it by introducing an auxiliary term constructor `TL_case`, called distributive case expression. The constructor may be thought of as a composition of the distributivity morphism with the more standard variant analysis operation `T_case` (the actual factorizer for coproducts, see Section 7.1.2).

The model of s-Core is the initial *2-LC-lc* with sums. The *2-LC-lc* with sums are such *2-LC-lc* that (in each category $C(c, *)$) have sums with distributivity given by the distributive case expression. The most natural example of this concept is **Cat** with the category **Set** as the distinguished $*$ category.

6.2.2 Language

Syntax

There is an additional `funct`-term constructor `F_ss`

```
type funct =
  ...
  | F_ss of cat * funct IList.t
```

and four `trans`-term constructors

```
type trans =
  ...
  | T_ss of cat * trans IList.t
  | T_in of funct IList.t * IdIndex.t
  | T_case of trans IList.t * funct
  | TL_case of funct * trans IList.t * funct
```

For the rest of the thesis we will, as a rule, denote the sum functor `F_ss`(c, lf) by

$$\texttt{[`}i_1\ \ f_1\texttt{|`}i_2\ \ f_2\texttt{|}\ \ldots\texttt{|`}i_n\ \ f_n\texttt{]}$$

and the distributive case expression `TL_case`(f, lt, h) by

$$\texttt{[`}i_1\ \ t_1\texttt{|`}i_2\ \ t_2\texttt{|}\ \ldots\texttt{|`}i_n\ \ t_n\texttt{]}$$

where the `funct`-terms f and h disappear from the term, but usually will be recoverable from the context. We will also sometimes write i for `T_pr`(lg, i) (the projection expression, introduced in the language of *2-LC-lc*) and `` `i `` for the injection `T_in`(lg, i) where the "`" sign is the back-quote.

Semantics

We will define the semantics of the new constructs of s-Core in an arbitrary fixed *2-LC-lc* with sums, instead of the initial one, to enable extensions. The semantics of old constructs is analogous as described in Section 4.3.2.

Just as for *2-LC-lc*, we are overly general in referring to categories $C(c, e)$, while the sums we intend to work with inhabit only categories $C(c, *)$. This is not harmful to the definition of semantics, but for the sake of typing conditions we have to declare that the new terms are correct only if their target categories are `*`.

- `F_ss`$(c, \mathit{lf}) : c \to e$
  is a labeled coproduct in $C(c, e)$ of an indexed list of objects (that is functors) lf, where $f_i : c \to e$,

- `T_ss`(c, lt) : `F_ss`$(c, \mathit{lf}) \to$ `F_ss`(c, lh)
  is the action of the coproduct functor on morphisms of $C(c, e)$ where lt is an indexed list of transformations, such that $t_i : f_i \to h_i$ and $f_i, h_i : c \to e$,

- `T_in`$(\mathit{lf}, i) : f_i \to$ `F_ss`(c, lf)
  is the injection morphism in $C(c, e)$, where $f_k : c \to e$,

- `T_case`(lt, h) : `F_ss`$(c, \mathit{lf}) \to h$
  is the case expression (variant analysis) of an indexed list of transformations lt, where $t_i : f_i \to h$ and $f_i, h : c \to e$.

- `TL_case`(f, lt, h) : `{`δ : `F_ss`(c, lg); ϵ : f`}` $\to h$
  is the distributive case expression of an indexed list of transformations lt, where δ and ϵ are distinguished labels, case analysis is performed over the δ components, t_i : `{`δ : g_i; ϵ : f`}` $\to h$ and $f, g_i, h : c \to e$.

Example. Here are a few sample terms. The distinguished labels δ and ϵ will be written here `AtIndex.atd` and `AtIndex.ate`, as is customary in our source code.

```
[‘True {}|‘False {}] (* boolean type *)
{} . T_in(True : {}; False : {}, True) (* truth value *)
{} . ’False (* falsity value *)
T_case (True = {} . ‘False;
        False = {} . ‘True,
        [‘True {}|‘False {}]) (* negation *)
[‘True  {} . ‘True
|‘False AtIndex.ate
] (* disjunction, 1st operand at AtIndex.atd, 2nd at AtIndex.ate *)
```

The domain of the distributive case expression and the domains of each of its immediate `trans`-term subexpressions consist of two components. This means that these terms expect to be composed with a record of two values. The ϵ component of this record would carry the “environment” of the computation or, in other words, the values for the “free variables” inside the case expression. Upon composition with the distributive case expression this component will be propagated unchanged to the subexpressions. The δ component of the record corresponds to the variant value. When the record is composed with the distributive case expression this component is analyzed and one of the values passed on to the subexpression with a matching label.

In fact, when disjoint sum type is added to λ-calculus, the case expression of λ-calculus always corresponds to `TL_case` and not to `T_case`. This is caused by the implicit assumption that case expressions must be allowed to contain free variables and that substitution may enter inside the case expressions and operate on subterms. Usually an identifier is assigned to each of the variants, which collectively correspond to the δ component of `TL_case` subterm's domain, and variables from outside of the λ-term corresponding to the ϵ component are implicitly visible inside.

By fixing the names of components in the domain of `TL_case` we restricted the distributivity of the sum to a specific family of products; the binary products with labels δ and ϵ. But distribution over multiple families is not needed for a programming language and additionally, it might be recovered up to isomorphism using the existing semantics.

6.2.3 Rewriting

Equations

To capture the theory of *2-LC-lc* with sums we extend the theory of *2-LC-lc* with the following axioms. We add a `funct`-equation that comes from the fact that `F_ss` is a composition of a record and an adjoint (see Section 7.1.3).

$$f\ \texttt{.}\ \texttt{[`}i\ \ g\texttt{|}\ \dots\texttt{]} \quad = \quad \texttt{[`}i\ \ f\ \texttt{.}\ g\texttt{|}\ \dots\texttt{]} \tag{64}$$

Injection `T_in`(lg, j) is a multiplication of a record `F_RECORD`(c, lg) and a unit of an adjunction, hence the following axiom.

$$f * \texttt{T\_in}(i\ \texttt{:}\ g\texttt{;}\ \dots, j) \quad = \quad \texttt{T\_in}(i\ \texttt{:}\ f\ \texttt{.}\ g\texttt{;}\ \dots, j) \tag{65}$$

Next we add the `trans`-equation defining `T_ss`.

$$\texttt{T\_ss}(c, lt) \quad = \quad \texttt{T\_RECORD}(c, lt) * \texttt{F\_ss}(\texttt{<}le\texttt{>},\ i\ \texttt{:}\ \texttt{F\_PR}(le, i)\texttt{;}\ \dots) \tag{66}$$

Sums may be axiomatized using either the axioms of coproduct augmented with the postulate of distributivity or by the axioms concerning directly the distributive operations. While the axioms for coproduct are simply a rewriting of the fundamental equations of adjunctions (see Section 7.1.3), the axioms for distributive combinators seem to be more programming oriented. One may expect that the programming axioms will look somewhat ad hoc, but surprisingly they are able to mimic the adjunction axioms quite accurately, and the rewriting rules based on them exhibit intriguing kinds of symmetry.

In the first presentation we focus on `T_case` as the base mechanism for operating on sums. To the axioms of *2-LC-lc* we add the existence and uniqueness requirements for the categorical coproduct.

$$\begin{aligned}\texttt{T\_in}(lg, i)\ \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ u_i\texttt{;}\ \ldots, h) &= u_i &(67)\\ \texttt{T\_case}(i\ \texttt{=}\ \texttt{T\_in}(lg, i)\ \texttt{.}\ u\texttt{;}\ \ldots, h) &= u &(68)\end{aligned}$$

Now we define some important morphisms. Let f be a functor representing the type of the additional "environment" parameter for the case analysis. Let lg be a list of functors representing the types of variants. Let $lf^i = \delta : g_i\,;\ \epsilon : f$. Then we define transformations

$$\begin{aligned}d_s(lg, f) &: \texttt{\{}\delta : \texttt{[}lg\texttt{]}\texttt{;}\ \epsilon : f\texttt{\}} \to \texttt{[`}i\ \ \texttt{\{}lf^i\texttt{\}|}\ \ldots\texttt{]}\\ v^i(lg, f) &: \texttt{\{}lf^i\texttt{\}} \to \texttt{\{}\delta : \texttt{[}lg\texttt{]}\texttt{;}\ \epsilon : f\texttt{\}}\\ p_s(lg, f) &: \texttt{[`}i\ \ \texttt{\{}lf^i\texttt{\}|}\ \ldots\texttt{]} \to \texttt{\{}\delta : \texttt{[}lg\texttt{]}\texttt{;}\ \epsilon : f\texttt{\}}\end{aligned}$$

as follows:

$$\begin{aligned}d_s(lg, f) &= \texttt{[`}i\ \ \texttt{T\_in}(i : \texttt{\{}lf^i\texttt{\};}\ j : \texttt{\{}lf^j\texttt{\};}\ \ldots, i)\texttt{|}\ \ldots\texttt{]}\\ v^i(lg, f) &= \texttt{\{}\delta\ \texttt{=}\ \texttt{T\_pr}(lf^i, \delta)\ \texttt{.}\ \texttt{T\_in}(lg, i)\texttt{;}\ \epsilon\ \texttt{=}\ \texttt{T\_pr}(lf^i, \epsilon)\texttt{\}}\\ p_s(lg, f) &= \texttt{T\_case}(i\ \texttt{=}\ v^i(lg, f)\texttt{;}\ \ldots, h)\end{aligned}$$

The next two equations say that $d_s(lg, f)$ is a distributivity morphism, that is an isomorphism between, on one hand, the product of the sum and the parameter, and on the other hand, the sum where the parameter is distributed among the individual variants.

$$\begin{aligned}p_s(lg, f)\ \texttt{.}\ d_s(lg, f) &= \texttt{T\_id}(\texttt{[`}i\ \ \texttt{\{}lf^i\texttt{\}|}\ \ldots\texttt{]}) &(69)\\ d_s(lg, f)\ \texttt{.}\ p_s(lg, f) &= \texttt{T\_id}(\texttt{\{}\delta : \texttt{[}lg\texttt{]}\texttt{;}\ \epsilon : f\texttt{\}}) &(70)\end{aligned}$$

Last we describe `TL_case` (denoted here by square brackets) as a distributive case expression.

$$\texttt{[`}i\ \ t\texttt{|}\ \ldots\texttt{]} = d_s(lg, f)\ \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ t\texttt{;}\ \ldots, h) \qquad (71)$$

The theory generated from axioms 64–71 and theory Φ_p will be called Φ_c.

Lemma 6.2.1. *Φ_c is the theory of 2-LC-lc with sums.*

Proof. The equations of Φ_p accurately describe the 2-cartesian structure by Lemma 6.1.19. The additional equations are either analogous to those for labeled products or for binary distributive coproducts, as seen in literature [20]. The correctness of the definition of distributivity morphism follows from the definition of the distributive case expression in the overview of the semantics. □

In the second presentation, instead of focusing on T_case we will treat TL_case as the base form and define others as deriving from it. To this end, first we express the standard T_case factorizer in terms of the TL_case operation.

$$\begin{aligned}&\texttt{T\_case}(i\ \texttt{=}\ u_i\texttt{;}\ \dots, h)\\ &= \texttt{\{}\delta\ \texttt{= T\_id([}lg\texttt{]);}\ \epsilon\ \texttt{=}\ t\texttt{\} . [`}i\ \ \texttt{T\_pr(}lf^i\texttt{,}\delta\texttt{) .}\ u_i\texttt{|}\ \dots\texttt{]}\end{aligned} \tag{72}$$

Here t : [lg] $\to f$ is an arbitrary transformation.

The original existence requirement (67) and uniqueness requirement (68), presented using TL_case rather than T_case, are quite weak as they cover only such distributive case expression that have operands beginning with a projection:

$$\begin{aligned}\texttt{\{}\delta\ \texttt{= T\_in(}lg, i\texttt{);}\ \epsilon\ \texttt{=}\ t\texttt{\} . [`}i\ \ \delta\ \texttt{.}\ u_i\texttt{|}\ \dots\texttt{]} &= u_i\\ \texttt{\{}\delta\ \texttt{= T\_id([}lg\texttt{]);}\ \epsilon\ \texttt{=}\ t\texttt{\} . [`}i\ \ \delta\ \texttt{. T\_in(}lg, i\texttt{) .}\ u\texttt{|}\ \dots\texttt{]} &= u\end{aligned}$$

Instead of these, we provide two strictly stronger analogues (73) and (74) of the existence and uniqueness requirements. Each of the above two equations is derivable from Φ_p and the corresponding equation below. The term $\texttt{v}^i(lg, f)$ is as defined previously.

$$\texttt{v}^i(lg, f)\ \texttt{. [`}i\ \ u_i\texttt{|}\ \dots\texttt{]} = u_i \tag{73}$$

$$\texttt{[`}i\ \ \texttt{v}^i(lg, f)\ \texttt{.}\ u\texttt{|}\ \dots\texttt{]} = u \tag{74}$$

The theory induced from the theory Φ_p of *2-LC-lc*, equations 64–66 and 72–74 will be called Φ_s.

Lemma 6.2.2. *The theory Φ_c based on* T_case *and the theory Φ_s based on* TL_case *are equal.*

Proof. Proving that Φ_c is contained in Φ_s is comparatively easy. Showing the opposite inclusion is more difficult. In particular, while deriving the axioms of Φ_s, we repeatedly use axioms of Φ_c to complicate, rather than to simplify terms, guessing many subterms. Sometimes, in the same derivation we use a single axiom of Φ_c both to complicate and to simplify subterms (different subterms, though not disjoint subterms). The derivation is carried out in Appendix B. □

Corollary 6.2.3. *Φ_s is the theory of 2-LC-lc with sums.*

Reduction

First, we have to describe the T_ss operation in terms of the basic sum constructors.

Lemma 6.2.4. *The following equation is a consequence of Φ_c.*

$$\texttt{T\_ss}(c,\ i\ \texttt{=}\ t\texttt{;}\ \ldots) \ = \ \texttt{T\_case}(i\ \texttt{=}\ t\ \texttt{.}\ \texttt{T\_in}(lg, i)\texttt{;}\ \ldots, h) \tag{75}$$

Proof. This is a special case of Theorem 7.1.13. A direct proof is dual to the proof of Lemma 6.1.20. □

The reduction system $\mathcal{R}_s$ will be based on theory Φ_s. We extend the reduction system $\mathcal{R}_p$ by the following rules. First, we describe composition with the sum functor.

$$f\ \texttt{.}\ \texttt{[`}i\ \ g\texttt{|}\ \ldots\texttt{]} \ \rightarrow \ \texttt{[`}i\ \ f\ \texttt{.}\ g\texttt{|}\ \ldots\texttt{]} \tag{76}$$

Then we deal with sum of transformations and the non-distributive case expression.

$$\texttt{T\_ss}(c,\ i\ \texttt{=}\ t\texttt{;}\ \ldots) \ \rightarrow \ \texttt{T\_case}(i\ \texttt{=}\ t\ \texttt{.}\ \texttt{T\_in}(lg, i)\texttt{;}\ \ldots, h) \tag{77}$$

$$\texttt{T\_case}(i\ \texttt{=}\ t\texttt{;}\ \ldots, h) \ \rightarrow \ \texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id}(g)\texttt{;}\ \epsilon\ \texttt{=}\ \texttt{\{\}\}}\ \texttt{.}\ \texttt{[`}i\ \ \delta\ \texttt{.}\ t\texttt{|}\ \ldots\texttt{]} \tag{78}$$

The next two rules are derived from the existence requirement (73) of Φ_s.

$$\texttt{\{}\delta\ \texttt{=}\ \texttt{`}i\texttt{;}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ \texttt{[`}i\ \ u_i\texttt{|}\ \ldots\texttt{]} \ \rightarrow \ \texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id}(g_i)\texttt{;}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ u_i \tag{79}$$

$$\texttt{\{}\delta\ \texttt{=}\ t_1\ \texttt{.}\ \texttt{`}i\texttt{;}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ \texttt{[`}i\ \ u_i\texttt{|}\ \ldots\texttt{]} \ \rightarrow \ \texttt{\{}\delta\ \texttt{=}\ t_1\texttt{;}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ u_i \tag{80}$$

We also need extensions to the rules for `T_FT`.

$$f * \texttt{T\_in}(i\ \texttt{:}\ g\texttt{;}\ \ldots, j) \ \rightarrow \ \texttt{T\_in}(i\ \texttt{:}\ f\ \texttt{.}\ g\texttt{;}\ \ldots, j) \tag{81}$$

$$f * \texttt{[`}i\ \ u\texttt{|}\ \ldots\texttt{]} \ \rightarrow \ \texttt{[`}i\ \ f * u\texttt{|}\ \ldots\texttt{]} \tag{82}$$

Finally, we add an additional rule for `T_TF`.

$$t * \texttt{[`}i\ \ g\texttt{|}\ \ldots\texttt{]} \ \rightarrow \ \texttt{T\_ss}(c,\ i\ \texttt{=}\ t * g\texttt{;}\ \ldots) \tag{83}$$

Theorem 6.2.5. *System $\mathcal{R}_s$ is strongly normalizing.*

Proof. We extend the proof of Theorem 6.1.23 for the cases of the additional rewriting rules. The additional assignments of precedences to term constructors (together with names of affected rules) are the following.

9. Functor composition has higher precedence than functor sum (76).

10. Term constructor `T_ss` has higher precedence than `T_case`, the vertical composition and the injection term constructors (77).

11. Term constructor `T_case` has higher precedence than the vertical composition, the record, the identity, the distributive case expression and the projection term constructors (78).

12. Vertical composition has higher precedence than identity (79).

Rule (79) belongs to the ordering by condition I from the proof of Theorem 6.1.4, together with condition 8 from the proof of Theorem 6.1.23 and condition 12 above. Rule (80) belongs to the ordering by condition I and 8, rules 81–82 by condition 3 and rule (83) by condition 4. □

Theorem 6.2.6. *System $\mathcal{R}_s$ is confluent.*

Proof. The proof is by inspection of critical pairs.

The newly introduced critical pairs, with respect to system $\mathcal{R}_p$, are easy to solve. For example, a critical pair emerges when a terms is vertically composed with the left hand side of rule (79)

$$t_1 \texttt{ . (\{}\delta \texttt{ = `}i\texttt{; }\epsilon \texttt{ = } t\texttt{\} . [`}i\ \ u_i\texttt{| }\ldots\texttt{])}$$

The composition may either reduce according to rule (79)

$$t_1 \texttt{ . (\{}\delta \texttt{ = `}i\texttt{; }\epsilon \texttt{ = } t\texttt{\} . [`}i\ \ u_i\texttt{| }\ldots\texttt{])} \rightarrow t_1 \texttt{ . (\{}\delta \texttt{ = T\_id(}g_i\texttt{); }\epsilon \texttt{ = } t\texttt{\} . }u_i\texttt{)}$$

or change associativity by rule (43)

$$t_1 \texttt{ . (\{}\delta \texttt{ = `}i\texttt{; }\epsilon \texttt{ = } t\texttt{\} . [`}i\ \ u_i\texttt{| }\ldots\texttt{])} \rightarrow \texttt{(}t_1 \texttt{ . \{}\delta \texttt{ = `}i\texttt{; }\epsilon \texttt{ = } t\texttt{\}) . [`}i\ \ u_i\texttt{| }\ldots\texttt{]}$$

To solve the critical pair, in the latter case

$$\texttt{(}t_1 \texttt{ . \{}\delta \texttt{ = `}i\texttt{; }\epsilon \texttt{ = } t\texttt{\}) . [`}i\ \ u_i\texttt{| }\ldots\texttt{]}$$

we may absorb t_1 into the record by rule (57)

$$\texttt{\{}\delta \texttt{ = } t_1 \texttt{ . `}i\texttt{; }\epsilon \texttt{ = } t_1 \texttt{ . } t\texttt{\} . [`}i\ \ u_i\texttt{| }\ldots\texttt{]}$$

and then use rule (80)

$$\texttt{\{}\delta \texttt{ = } t_1\texttt{; }\epsilon \texttt{ = } t_1 \texttt{ . } t\texttt{\} . }u_i$$

In the former case

$$t_1 \texttt{ . (\{}\delta \texttt{ = T\_id(}g_i\texttt{); }\epsilon \texttt{ = } t\texttt{\} . }u_i\texttt{)}$$

we change associativity by rule (43)

$$\texttt{(}t_1 \texttt{ . \{}\delta \texttt{ = T\_id(}g_i\texttt{); }\epsilon \texttt{ = } t\texttt{\}) . }u_i$$

and use rule (57)

$$\texttt{\{}\delta \texttt{ = } t_1\texttt{; }\epsilon \texttt{ = } t_1 \texttt{ . } t\texttt{\} . }u_i$$

obtaining the same term and thus solving the critical pair. □

Observation 6.2.7. *$\mathcal{R}_s$ is sound with respect to Φ_s.*

The language of *2-LC-lc* with sums, interpreted in the initial *2-LC-lc* with sums and evaluated by the reduction system $\mathcal{R}_s$, is called s-Core. There are other possible variants of the reduction system for sums. We could have added rules based on a weakened uniqueness requirement of Φ_s:

$$\begin{aligned} \texttt{[`}i\ \ t\texttt{| ...] . }u &\rightarrow \texttt{[`}i\ \ t\texttt{ . }u\texttt{| ...]} \qquad (84)\\ (t_1\texttt{ . [`}i\ \ t\texttt{| ...]) . }u &\rightarrow t_1\texttt{ . [`}i\ \ t\texttt{ . }u\texttt{| ...]} \qquad (85)\end{aligned}$$

However, the system enriched with the two rules is no longer confluent. For example the term

$$\texttt{[`}i\ \ \texttt{\{}j\texttt{ = \{\}\}] . \{}k\texttt{ = T\_pr(}j\texttt{ : \{\}, }j\texttt{)\}}$$

when rewritten according to rule (57) ends up as a record and when rewritten according to rule (84) ends up as a case expression. Any subsequent $\mathcal{R}_s$-reductions touch only the inside of the term tree, so those terms cannot be rewritten to a common form.

While eliminating `T_case` in rule (78) we have fixed `{}` in place of an arbitrary transformation appearing in equation (72). This and the lack of any form of η-rules distinguish $\overline{\mathcal{R}_s}$ from theory Φ_s. Note that the weak η-rules (84) and (85), apart from compromising confluence, are also not very convincing computationally. They do not describe how to substitute into case expression as the weak η-rule for products (57) does. Their computational meaning seems to correspond to some form of speculative evaluation (as in the microprocessor technology), since in the rules we copy and merge all the code after the case expression into the branches, instead of waiting until there is enough information to decide which branch will be taken. When there is a substantial number of variants in the case expression this seems to be quite an expensive optimization technique, especially in terms of the size of binaries.

In $\mathcal{R}_s$ the reductions corresponding to λ-calculus substitutions into case expressions are deferred until after case analysis. If we wanted to perform the substitutions immediately we would have to prove that the following equation is a consequence of Φ_s. This is easy but involves the uniqueness requirements.

$$\begin{aligned} &\texttt{\{}\delta\texttt{ = T\_pr(}lf,\delta\texttt{); }\epsilon\texttt{ = T\_pr(}lf,\epsilon\texttt{) . }t\texttt{\} . [`}i\ \ u\texttt{| ...]} = \\ &\quad\texttt{[`}i\ \ \texttt{\{}\delta\texttt{ = T\_pr(}lg,\delta\texttt{); }\epsilon\texttt{ = T\_pr(}lg,\epsilon\texttt{) . }t\texttt{\} . }u\texttt{| ...]}\end{aligned}$$

And then we would add a rewrite rule based on this equation.

$$\begin{aligned} &\texttt{\{}\delta\texttt{ = }t_1\texttt{; }\epsilon\texttt{ = }t\texttt{\} . [`}i\ \ u\texttt{| ...]} \rightarrow \\ &\texttt{\{}\delta\texttt{ = }t_1\texttt{; }\epsilon\texttt{ = T\_id(}f\texttt{)\} . [`}i\ \ \texttt{\{}\delta\texttt{ = }\delta\texttt{; }\epsilon\texttt{ = }\epsilon\texttt{ . }t\texttt{\} . }u\texttt{| ...]} \qquad (86)\end{aligned}$$

This rule alone forms a non-terminating reduction system, so the use of the rule would have to be forbidden whenever term t happens to be an identity.

A sufficiently clever compiler may choose to use the additional rules above based on a term context, on the history of computation or on user preferences. As long as the rules follow from the equational theory of sums presented in the previous section the optimizations are guaranteed to be sound. In particular the commuting conversions of [62] are admissible in our framework.

6.2.4 Conclusion

We have proven the equivalence of two axiomatizations for *2-LC-lc* with sums (Lemma 6.2.2). One of them is an extension of the standard presentation of non-distributive coproducts (Lemma 6.2.1), the other captures the distributive coproduct operation from a programming perspective and is a basis of the reduction system for the core language s-Core. We have also discussed the role of η-equations in rewriting coproducts and shown the reduction system of s-Core to be confluent (Theorem 6.2.6) and strongly normalizing (Theorem 6.2.5).

The distributivity of coproducts is not only for a programmer's convenience; it is essential for making the coproduct values parameterizable by module components as well as for expressiveness of the core language itself. For example, binary boolean operations that sometimes need to inspect both their operands, such as disjunction (see examples in Section 6.2.2), are not expressible using plain coproducts. The simplest explanation is that the disjunction should have a product as its domain (to account for the two operands) and at the same time should be a conditional (case analysis) expression. But the case expression of the plain coproduct has coproduct as its domain. A complete proof would involve a lemma that nesting the case analysis inside any of the other constructions to obtain a correct domain entails forgetting one of the operands (projection) or even both of them (empty record).

6.3 i-Core — core language with (co)inductive types

Values modeled by the transformations of the core language s-Core have the property that the length of their normal forms is linearly proportional to the length of their types. Consequently, the only programs the framework can embrace are those operating on data of a bounded size in the normal form. In this section we extend the model with mechanisms corresponding to inductive and coinductive types. With this addition, a single type may classify data of arbitrary size, for example the inductive type of lists classifies lists of all sizes and the values of the coinductive type of streams may even represent infinite lists.

6.3.1 Definition based on s-Core

The programming notions of inductive and coinductive types are modeled categorically by algebras (here we refer to algebras inside the *2-LC-lc* and not to the category itself viewed as algebra) and co-algebras [115]. *2-LC-lc* with sums and (co)algebras is a *2-LC-lc* with sums that has algebras and co-algebras for all its functors (1-morphisms). By algebras and co-algebras in the context of a *2-LC-lc* we mean the obvious generalizations of these notions from **Cat** to 2-categories, as defined by diagrams below and more formally by conditional equations in Section 6.3.3.

The programming language i-Core is an extension of the language s-Core with the operations related to inductive and (co)inductive types, interpreted in the initial *2-LC-lc* with sums and (co)algebras, which is known to exist, despite the conditional axioms, because the definedness of operations is based on the same simple categorical principles as for the ordinary *2-LC-lc*. Apart of the standard (co)inductive operations and some simple abbreviations, we provide the inductive operation `TL_fold` and the co-inductive operation `TL_unfold`, which are not so simple to define. These operations relate to the standard (co)inductive iterators `T_fold` and `T_unfold` in the same way as the distributive case operation `TL_case` relates to the ordinary case operation `T_case`. They provide a way to iterate on one component of the transformation's domain, while treating the other component as a constant parameter.

The only new operation of i-Core not directly relevant to (co)inductive types is the operation of mapping a functor over a transformation (`T_map`). This operation resembles multiplication by a functor from the right (see Corollary 6.3.4). Similarly as with the iterators, the multiplication takes place only on one component of the transformation's domain, and the other is kept as a constant parameter. The mapping is used to define equationally the distributive iterators, without referring to the simple iterators. However, the mapping operation can be quite useful by itself and has some programming meaning even in the absence of inductive and coinductive types. For example, one can map a doubling function over a list of integers obtaining a list of doubled integers. The same mapping combinator with the same doubling function can be used to map over a tree of integers, or over a record. See Section 2.1.2 for some concrete examples.

6.3.2 Language

Syntax

There are two additional `funct`-term constructors in the language of i-Core

```
type funct =
  ...
```

```
    | F_ii of funct
    | F_tt of funct
```

and eleven additional `trans`-term constructors

```
  type trans =
    ...
    | T_map of funct * trans
    | T_ii of trans
    | T_con of funct
    | T_fold of funct * trans
    | TL_fold of funct * trans
    | T_de of funct
    | T_tt of trans
    | T_uncon of funct
    | T_unfold of funct * trans
    | TL_unfold of funct * trans
    | T_unde of funct
```

Here we will start denoting the functor projections $\texttt{F\_PR}(lc, i)$ just by "i" (the same as transformation projections $\texttt{T\_pr}(lf, i)$, but the context will differentiate them). We will also use the following horizontal composition shorthand. Let c be an arbitrary category and let t and u be transformations with the following sources and targets (where ι and κ are fixed, distinguished labels):

$$\begin{aligned} t &: c \to \texttt{*} \\ u &: \texttt{<}\iota\ \texttt{-}\ \texttt{*;}\ \kappa\ \texttt{-}\ c\texttt{>} \to \texttt{*} \end{aligned}$$

We will write

$$u[t] \ : \ c \to \texttt{*}$$

to mean

$$\texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(c)\texttt{>} * u$$

This is extended to an operation on functors and transformations, by representing functors as identity transformations. So for example $u[f]$ denotes $u[\texttt{T\_id}(f)]$. If both the operands are functors then the shorthand denotes an identity transformation and we will treat the result as a functor (the domain and at the same time the codomain of the identity transformation).

Semantics

The semantics is given in an arbitrary fixed *2-LC-lc* with sums and (co)algebras. As before, we are overly general by considering types with arbitrary target categories, while elsewhere in this section only target `*` is considered. This generality is not harmful to the definition of semantics, but for simplicity we declare that the language i-Core only includes those new terms that have target categories `*`. The user of the core language will always use only types and values that have target category `*`.

On the other hand, for the construction of (co)inductive modules in Section 9.1.2 we will extend the language, allowing compound target categories of the constructor and destructor combinators. However, as long as the mapping and structured recursion combinators retain target `*`, the rules of the reduction system needn't be changed.

- $\texttt{F\_ii}(g) : d \to c$
  is an initial algebra of a functor $g : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to c$, where ι and κ are distinguished labels and the induction is over the ι component while the κ component is kept for parameters,

- $\texttt{F\_tt}(g) : d \to c$
  is a terminal co-algebra of a functor $g : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to c$, where the co-induction is over the ι component and the κ component is kept for parameters,

- $\texttt{T\_map}(g, t) : \texttt{\{}\delta : g[f_\delta]\texttt{;}\ \epsilon : f_\epsilon\texttt{\}} \to g[h]$
  is the action of the ι component of the functor $g : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to c$, on the δ component of the transformation $t : \texttt{\{}\delta : f_\delta\texttt{;}\ \epsilon : f_\epsilon\texttt{\}} \to h$ where $f_\delta, f_\epsilon, h : d \to c$,

- $\texttt{T\_ii}(t) : \texttt{F\_ii}(f) \to \texttt{F\_ii}(h)$
  is the action of the initial algebra functor on transformation $t : f \to h$, where $f, h : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to c$,

- $\texttt{T\_con}(g) : g[\texttt{F\_ii}(g)] \to \texttt{F\_ii}(g)$
  is the constructor morphism of the initial algebra for a functor $g : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to c$,

- $\texttt{T\_fold}(g, t) : \texttt{F\_ii}(g) \to h$
  is the iteration (folding) over transformation $t : g[h] \to h$ in the initial algebra for a functor $g : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to c$, where $h : d \to c$,

- $\texttt{TL\_fold}(g, t) : \texttt{\{}\delta : \texttt{F\_ii}(g)\texttt{;}\ \epsilon : f\texttt{\}} \to h$
  is the iteration (folding) over the δ component of transformation $t : \texttt{\{}\delta : g[h]\texttt{;}\ \epsilon : f\texttt{\}} \to h$, in the initial algebra for a functor $g : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to c$, where $f, h : d \to c$,

- $\texttt{T\_de}(g) : \texttt{F\_ii}(g) \to g[\texttt{F\_ii}(g)]$
  is the destructor morphism of the initial algebra for a functor $g : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to c$,

- `T_tt`(t) : `F_tt`$(f) \to$ `F_tt`(h)
  is the action of the terminal co-algebra functor on transformation $t : f \to h$, where $f, h :$ `<`ι `-` c`;` κ `-` d`>` $\to c$,
- `T_uncon`$(g) : g[$`F_tt`$(g)] \to$ `F_tt`(g)
  is the constructor morphism of the terminal co-algebra for a functor g : `<`ι `-` c`;` κ `-` d`>` $\to c$,
- `T_unfold`$(g, t) : h \to$ `F_tt`(g)
  is the co-iteration (unfolding) over transformation $t : h \to g[h]$ in the terminal co-algebra for a functor $g :$ `<`ι `-` c`;` κ `-` d`>` $\to c$, where $h : d \to c$,
- `TL_unfold`(g, t) : `{`δ : h`;` ϵ : f`}` $\to$ `F_tt`(g)
  is the co-iteration (unfolding) over the δ component of transformation t : `{`δ : h`;` ϵ : f`}` $\to g[h]$, in the terminal co-algebra for a functor $g :$ `<`ι `-` c`;` κ `-` d`>` $\to c$, where $f, h : d \to c$,
- `T_unde`(g) : `F_tt`$(g) \to g[$`F_tt`$(g)]$
  is the destructor morphism of the terminal co-algebra for a functor g : `<`ι `-` c`;` κ `-` d`>` $\to c$.

Remember that the operation $g[\_]$ appearing throughout the definitions is not a syntactic operation but an abbreviation for a certain horizontal composition.

Here are the standard diagrams that relate the main operations.

$$\begin{tikzcd}
\texttt{g[F\_ii(g)]} \arrow[r, dashed, "\texttt{g[T\_fold(g, t)]}"] \arrow[d, "\texttt{T\_con(g)}"'] & \texttt{g[h]} \arrow[d, "\texttt{t}"] \\
\texttt{F\_ii(g)} \arrow[r, dashed, "\texttt{T\_fold(g, t)}"] & \texttt{h}
\end{tikzcd}$$

$$\begin{tikzcd}
\texttt{h} \arrow[r, dashed, "\texttt{T\_unfold(g, t)}"] \arrow[d, "\texttt{t}"'] & \texttt{F\_tt(g)} \arrow[d, "\texttt{T\_unde(g)}"] \\
\texttt{g[h]} \arrow[r, dashed, "\texttt{g[T\_unfold(g, t)]}"] & \texttt{g[F\_tt(g)]}
\end{tikzcd}$$

Since the introduced algebras are initial, `T_fold(g, t)` in the first diagram is unique — any 2-morphism replacing it in the two positions in the diagram and making the diagram commutative must be equal to `T_fold(g, t)`. Similarly, our co-algebras are terminal, so `T_unfold(g, t)` is unique for every pair (`g`, `t`).

Example. Here are some expressions illustrating the inductive operations. The labels ι, κ, δ and ϵ are here written as `AtIndex.atj`, `AtIndex.atk`, `AtIndex.atd` and `AtIndex.ate`, respectively.

```
F_ii([‘Zero {}|‘Succ AtIndex.atj]) (* natural numbers type *)
‘Zero . T_con([‘Zero {}|‘Succ AtIndex.atj])
  . ‘Succ . T_con([‘Zero {}|‘Succ AtIndex.atj]) (* number one *)
T_de([‘Zero {}|‘Succ AtIndex.atj])
  . T_case (Zero = {} . ‘True;
            Succ = {} . ‘False,
            [‘True {}|‘False {}]) (* test for zero *)
T_fold([‘Zero {}|‘Succ AtIndex.atj],
       [‘Zero {} . ‘True
       |‘Succ T_case (True = {} . ‘False;
                      False = {} . ‘True,
                      [‘True {}|‘False {}])
       ]
      ) (* parity test *)
TL_fold([‘Zero {}|‘Succ AtIndex.atj],
        [‘Zero AtIndex.ate
        |‘Succ AtIndex.atd . ‘Succ
               . T_con([‘Zero {}|‘Succ AtIndex.atj])
        ]
       ) (* addition *)
```

6.3.3 Rewriting

Equations

To the axioms of Φ_c we add two additional `funct`-equations that describe instantiation of the (co)inductive types.

$$f \texttt{ . F\_ii}(g) \;=\; \texttt{F\_ii(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g\texttt{)} \qquad (87)$$
$$f \texttt{ . F\_tt}(g) \;=\; \texttt{F\_tt(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g\texttt{)} \qquad (88)$$

Type-instantiating the new transformations works in a similar way, but we need to require this only for the two basic constructors.

$$f * \texttt{T\_con}(g) \;=\; \texttt{T\_con(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g\texttt{)} \qquad (89)$$
$$f * \texttt{T\_unde}(g) \;=\; \texttt{T\_unde(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g\texttt{)} \qquad (90)$$

Then we express some of the constructors in terms of the others.

$$\begin{aligned}
\texttt{T\_ii}(t) &= \texttt{T\_fold}(f, t[\texttt{F\_ii}(h)]\ \texttt{.}\ \texttt{T\_con}(h)) && (91)\\
\texttt{T\_tt}(t) &= \texttt{T\_unfold}(h, \texttt{T\_unde}(f)\ \texttt{.}\ t[\texttt{F\_tt}(f)]) && (92)\\
\texttt{T\_de}(g) &= \texttt{T\_fold}(g, g[\texttt{T\_con}(g)]) && (93)\\
\texttt{T\_uncon}(g) &= \texttt{T\_unfold}(g, g[\texttt{T\_unde}(g)]) && (94)
\end{aligned}$$

We give the definition of $\texttt{T\_map}(g, t)$ by cases on the form of term g. Consequently, the definition does not fully determine the behavior of mapping in categories that are not reachable (viewed as algebras). We expect there is a universal categorical characterization of the mapping combinator, similarly as there is one for the multiplication by a functor from the right. If this is the case and the characterization can be expressed equationally, it would make a good axiom and the current numerous equations (and reduction rules) would be derivable from it, as is the case for multiplication.

Let t : $\texttt{\{}\delta : f_\delta\texttt{;}\ \epsilon : f_\epsilon\texttt{\}} \to h$. The operand functor g has to be of a type $g : \texttt{<}\iota\ \texttt{-}\ \texttt{*;}\ \kappa\ \texttt{-}\ d\texttt{>} \to \texttt{*}$ so it may neither be of the form $\texttt{F\_ID}(c)$ nor $\texttt{F\_RECORD}(c, lf)$. If g is a projection at label ι we have the following axiom.

$$\texttt{T\_map}(\texttt{F\_PR}(lc, \iota), t) = t \qquad (95)$$

If the projection is at label κ then to accommodate the case when d is a product category we have to compose the projection $\texttt{F\_PR}(lc, \kappa)$ with some functor f_2, so that the target is `*`, as required. The functor at label δ at the right hand side of the equation is f_2, because $(\texttt{F\_PR}(lc, \kappa)\ \texttt{.}\ f_2)[f_\delta] = f_2$.

$$\texttt{T\_map}(\texttt{F\_PR}(lc, \kappa)\ \texttt{.}\ f_2, t) = \texttt{T\_pr}(\delta : f_2\texttt{;}\ \epsilon : f_\epsilon,\ \delta) \qquad (96)$$

The cases of product and sum functors exhibit a similarity with multiplication by a functor (compare with rules (60) and (83)). Further insight into the mapping equations can be gained by consulting the general rule for rewriting multiplication in Section 7.1.4. In these axioms we use the abbreviated notation for the projections δ, i and ϵ and the injection $`i$.

$$\begin{aligned}
\texttt{T\_map}(\texttt{\{}lg\texttt{\}}, t) &= \texttt{\{}i\ \texttt{=}\ \texttt{\{}\delta\ \texttt{=}\ \delta\ \texttt{.}\ i\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}}\ \texttt{.}\ \texttt{T\_map}(g_i, t)\texttt{;}\ \ldots\texttt{\}} && (97)\\
\texttt{T\_map}(\texttt{[}lg\texttt{]}, t) &= \texttt{[}`i\ \texttt{T\_map}(g_i, t)\ \texttt{.}\ `i\texttt{|}\ \ldots\texttt{]} && (98)
\end{aligned}$$

To describe mapping by a (co)inductive functor, we define some auxiliary functors. Let g, f_δ and h be functors. Then

$$\begin{aligned}
g_f &= \texttt{<}\iota : \iota\texttt{;}\ \kappa : \texttt{<}\iota : \kappa\ \texttt{.}\ f_\delta\texttt{;}\ \kappa : \kappa\texttt{>>}\ \texttt{.}\ g\\
g_h &= \texttt{<}\iota : \iota\texttt{;}\ \kappa : \texttt{<}\iota : \kappa\ \texttt{.}\ h\texttt{;}\ \kappa : \kappa\texttt{>>}\ \texttt{.}\ g\\
g_p &= \texttt{<}\iota : \kappa\ \texttt{.}\ \iota\texttt{;}\ \kappa : \texttt{<}\iota : \iota\texttt{;}\ \kappa : \kappa\ \texttt{.}\ \kappa\texttt{>>}\ \texttt{.}\ g\\
g_a &= \texttt{<}\iota : \texttt{F\_ii}(g_h)\texttt{;}\ \kappa : \texttt{F\_ID}(c)\texttt{>}\\
g_c &= \texttt{<}\iota : \texttt{F\_tt}(g_f)\texttt{;}\ \kappa : \texttt{F\_ID}(c)\texttt{>}
\end{aligned}$$

The two axioms we give below could be made simpler, by performing the mapping on right hand sides using not the functor g_p but $g_p[g_a]$ and $g_p[g_c]$, respectively. The advantage of the current form is that it can be read as an inductive definition of T_map, as we will state in Lemma 6.3.6.

$$\begin{aligned}
\texttt{T\_map}(\texttt{F\_ii}(g), t) &= \texttt{TL\_fold}(g_f, (g_a * \texttt{T\_map}(g_p, \kappa * t)) \\
&\quad \texttt{.}\ \texttt{T\_con}(g_h)) \qquad (99) \\
\texttt{T\_map}(\texttt{F\_tt}(g), t) &= \texttt{TL\_unfold}(g_h, \texttt{\{}\delta\ \texttt{=}\ \delta\ \texttt{.}\ \texttt{T\_unde}(g_f)\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}} \\
&\quad \texttt{.}\ (g_c * \texttt{T\_map}(g_p, \kappa * t))) \qquad (100)
\end{aligned}$$

In the following equations we focus on T_fold and T_unfold as the basic (co)inductive operations. The distributive operators will be defined in terms of them later, analogously as in theory Φ_c for distributive coproducts. We start with the equations expressing the existence of the universal morphisms for a (co)algebra. They correspond to the requirement that the diagrams given above are commutative.

$$\begin{aligned}
\texttt{T\_con}(g)\ \texttt{.}\ \texttt{T\_fold}(g, t) &= g[\texttt{T\_fold}(g, t)]\ \texttt{.}\ t \qquad (101) \\
\texttt{T\_unfold}(g, t)\ \texttt{.}\ \texttt{T\_unde}(g) &= t\ \texttt{.}\ g[\texttt{T\_unfold}(g, t)] \qquad (102)
\end{aligned}$$

We conclude by stating the uniqueness requirements that (in contrast to the uniqueness requirements of products or sums) are expressed as conditional equations. They correspond to the requirement that the folding and unfolding morphisms in the diagrams are unique.

$$\begin{aligned}
\texttt{T\_con}(g)\ \texttt{.}\ u = g[u]\ \texttt{.}\ t &\Rightarrow u = \texttt{T\_fold}(g, t) \qquad (103) \\
u\ \texttt{.}\ \texttt{T\_unde}(g) = t\ \texttt{.}\ g[u] &\Rightarrow u = \texttt{T\_unfold}(g, t) \qquad (104)
\end{aligned}$$

Now we proceed to define TL_fold and TL_unfold in terms of other operations. The distributivity morphism for inductive type d_i, to be used for expressing distributivity of TL_fold below, may be defined in the following way (together with its inverse).

$$\begin{aligned}
d_i(g, f) &= \texttt{TL\_fold}(g, \texttt{T\_con}(\texttt{\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{\}})) \\
\mathrm{v}_i(g, f) &= \texttt{\{}\delta\ \texttt{=}\ \delta\ \texttt{.}\ g[\texttt{T\_pr}(\delta\ \texttt{:}\ \texttt{F\_ii}(g)\texttt{;}\ \epsilon\ \texttt{:}\ f,\ \delta)]\ \texttt{.}\ \texttt{T\_con}(g)\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}} \\
p_i(g, f) &= \texttt{T\_fold}(\texttt{\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{\}}, \mathrm{v}_i(g, f))
\end{aligned}$$

The domains and codomains are as follows.

$$\begin{aligned}
d_i(g, f) &: \texttt{\{}\delta\ \texttt{:}\ \texttt{F\_ii}(g)\texttt{;}\ \epsilon\ \texttt{:}\ f\texttt{\}} \to \texttt{F\_ii}(\texttt{\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{\}}) \\
\mathrm{v}_i(g, f) &: \texttt{\{}\delta\ \texttt{:}\ g[\texttt{\{}\delta\ \texttt{:}\ \texttt{F\_ii}(g)\texttt{;}\ \epsilon\ \texttt{:}\ f\texttt{\}}]\texttt{;}\ \epsilon\ \texttt{:}\ f\texttt{\}} \to \texttt{\{}\delta\ \texttt{:}\ \texttt{F\_ii}(g)\texttt{;}\ \epsilon\ \texttt{:}\ f\texttt{\}} \\
p_i(g, f) &: \texttt{F\_ii}(\texttt{\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{\}}) \to \texttt{\{}\delta\ \texttt{:}\ \texttt{F\_ii}(g)\texttt{;}\ \epsilon\ \texttt{:}\ f\texttt{\}}
\end{aligned}$$

We require that $d_i(g, f)$ be an isomorphism.

$$p_i(g, f) \texttt{ . } d_i(g, f) = \texttt{T\_id(F\_ii(\{}\delta \texttt{ : } g\texttt{; } \epsilon \texttt{ : } \kappa \texttt{ . } f\texttt{\}))} \tag{105}$$

$$d_i(g, f) \texttt{ . } p_i(g, f) = \texttt{T\_id(\{}\delta \texttt{ : F\_ii(}g\texttt{); } \epsilon \texttt{ : } f\texttt{\})} \tag{106}$$

The operation TL_fold is then described by the following equation.

$$\texttt{TL\_fold(}g, t\texttt{)} = d_i(g, f) \texttt{ . T\_fold(\{}\delta \texttt{ : } g\texttt{; } \epsilon \texttt{ : } \kappa \texttt{ . } f\texttt{\}}, t\texttt{)} \tag{107}$$

Let $d(g, h, f)$, to be used in the equation describing TL_unfold, be defined and typed as follows

$$\begin{aligned} d(g, h, f) &= \texttt{T\_map(}g, \texttt{T\_id(\{}\delta \texttt{ : } h\texttt{; } \epsilon \texttt{ : } f\texttt{\}))} \\ d(g, h, f) &: \texttt{\{}\delta \texttt{ : } g[h]\texttt{; } \epsilon \texttt{ : } f\texttt{\}} \to g[\texttt{\{}\delta \texttt{ : } h\texttt{; } \epsilon \texttt{ : } f\texttt{\}}] \end{aligned}$$

Note that neither the sum type distributivity morphism $d_s(lg, f)$, defined in Section 6.2.3, nor the inductive type distributivity morphism $d_i(g, f)$ are special cases of $d(g, h, f)$. Moreover $d(g, h, f)$ need not be an isomorphism. We may now proceed with defining TL_unfold in terms of the other constructors.

$$\texttt{TL\_unfold(}g, t\texttt{)} = \texttt{T\_unfold(}g, \texttt{\{}\delta \texttt{ = } t\texttt{; } \epsilon \texttt{ = } \epsilon\texttt{\} . } d(g, h, f)\texttt{)} \tag{108}$$

The theory generated by axioms 87–108 and theory Φ_c will be called Φ_i.

Observation 6.3.1. *Φ_i holds in any 2-LC-lc with sums and (co)algebras.*

We conjecture that if we had a complete axiomatization of the mapping combinator, Φ_i would be the theory of *2-LC-lc* with sums and (co)algebras. In particular, we expect the following property to be true, and so in the next section we try to validate our reduction system against Φ_i.

Conjecture 6.3.2. *Every ground equation of the theory of 2-LC-lc with sums and (co)algebras is derivable from Φ_i.*

We described mapping as analogous to multiplication by a functor, but affecting only the δ component of the domain of the transformation. In accord with this intuition, in the following lemma we show that if we ignore the other component of the domain then mapping is expressible as multiplication. The equation below is proved only for the class of reachable algebras, due to the syntax-directed definition of T_map in Φ_i. By requiring g to be a ground term, rather than a variable, we enable the use of the equation for all categories, not only reachable ones.

Lemma 6.3.3. *Let $t : f \to h$, $f, h : d \to \texttt{*}$ and let g be a ground* funct*-term such that $g : \texttt{<}\iota \texttt{ - *; } \kappa \texttt{ - } d\texttt{>} \to \texttt{*}$. Then the following equation follows from Φ_i.*

$$\texttt{T\_map(}g, \delta \texttt{ . } t\texttt{)} = \delta \texttt{ . } g[t] \tag{109}$$

Proof. The proof by induction over the normal form of g using the order on terms from the proof of Lemma 6.3.6 is given in Appendix B. The cases where this differs from a standard structural induction are those for the (co)inductive types. There we need to assume the induction hypothesis for the functor g_p (as defined for equation (99) and later), which is not a subterm of the first argument of `T_map`, but contains one less (co)inductive type constructor. $\square$

Conversely, simple cases of multiplication by a functor can be expressed by mapping, as shown in the following proposition proved in Appendix B. The premise that $g : * \to *$ is essential, because the target category of the mapping transformation must always be $*$. Multiplication by functors with product targets cannot be expressed by simple mapping.

Corollary 6.3.4. *Let* $t : f \to h$, $f, h : d \to *$ *and let* g *be a ground* `funct`*-term such that* $g : * \to *$. *Then the following equation follows from* Φ_i.

$$t * g \quad = \quad \texttt{\{}\delta \texttt{ = T\_id(}f \texttt{ . } g\texttt{); } \epsilon \texttt{ = \{\}\} . T\_map(}\iota \texttt{ . } g, \delta \texttt{ . } t\texttt{)} \qquad (110)$$

The intuition that the κ component of the functor argument and the ϵ component of the transformation argument of the mapping combinator are for parameters, is captured in the first two equations of the following observation. The third equation captures the intuition that if we include all necessary parameters in each data cell of our data structure we no longer need to use the mapping combinator — multiplication by a functor from the right suffices.

Observation 6.3.5. *Let* g *be a ground* `funct`*-term such that* $g : \texttt{<}\iota \texttt{ - *; } \kappa \texttt{ - } d\texttt{>} \to *$. *Then the following three equation follow from* Φ_i.

$$\begin{aligned} f * \texttt{T\_map(}g, u\texttt{)} &= \texttt{T\_map(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g, f * u\texttt{)} && (111)\\ \texttt{\{}\delta \texttt{ = } \delta\texttt{; } \epsilon \texttt{ = } \epsilon \texttt{ . } t\texttt{\} . T\_map(}g, u\texttt{)} &= \texttt{T\_map(}g, \texttt{\{}\delta \texttt{ = } \delta\texttt{; } \epsilon \texttt{ = } \epsilon \texttt{ . } t\texttt{\} . } u\texttt{)} && (112)\\ \texttt{T\_map(}g, u\texttt{)} &= d(g, h, f) \texttt{ . } g[u] && (113) \end{aligned}$$

Each of the three equation can be proved by induction over g, just as equation (109). Unfortunately equations (109), (111), (112) and (113) are still not enough to universally characterize mapping.

Reduction

The reduction system $\mathcal{R}_i$ is the system $\mathcal{R}_s$ extended by the following rules. The `funct`-rules and most of the rules for `T_map` closely follow the axioms of Φ_i. Two following rules are based on equations (87) and (88).

$$\begin{aligned} f \texttt{ . F\_ii(}g\texttt{)} &\to \texttt{F\_ii(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g\texttt{)} && (114)\\ f \texttt{ . F\_tt(}g\texttt{)} &\to \texttt{F\_tt(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g\texttt{)} && (115) \end{aligned}$$

Axiom (96) gives rise to a pair of rules, because if f_2 is an identity, it may be eliminated either before or after using the equation, thus threatening confluence. Hence we have to consider the case of a sole projection.

$$\texttt{T\_map}(\texttt{F\_PR}(lc, \kappa), t) \;\; \rightarrow \;\; \texttt{T\_pr}(\delta \;\texttt{:}\; \texttt{F\_ID(*)}\texttt{;}\; \epsilon \;\texttt{:}\; f_\epsilon,\, \delta) \tag{116}$$

Moreover, since the functor composition associates to the left, we cannot decide immediately whether the first term in the sequence of compositions is a projection, as required. If we fixed the strategy of evaluation to always reduce `funct`-terms before `trans`-terms the following rule would suffice.

$$\texttt{T\_map}(f_1 \;\texttt{.}\; f_2, t) \;\; \rightarrow \;\; \texttt{T\_pr}(\delta \;\texttt{:}\; (f_1 \;\texttt{.}\; f_2)[f_\delta]\texttt{;}\; \epsilon \;\texttt{:}\; f_\epsilon,\, \delta)$$

The rule, viewed as equation, does not belong to Φ_i, but if f_1 . f_2 is a normal form term then the typing ensures it is a sequence of projections starting with one at label κ, and such an equation belongs to Φ_i.

We could enforce that the first term in the sequence of compositions is a projection at label κ using rules of the following form, where the composition is meant to bind to the left, so that the functor on the left hand side is parenthesized as follows: $(((\texttt{F\_PR}(lc, \kappa) \;\texttt{.}\; f_n) \;\texttt{.}\; \ldots) \;\texttt{.}\; f_2) \;\texttt{.}\; f_1$.

$$\begin{aligned}&\texttt{T\_map}(\texttt{F\_PR}(lc, \kappa) \;\texttt{.}\; f_n \;\texttt{.}\; \ldots \;\texttt{.}\; f_2 \;\texttt{.}\; f_1, t) \;\; \rightarrow \\ &\;\; \texttt{T\_pr}(\delta \;\texttt{:}\; (f_n \;\texttt{.}\; \ldots \;\texttt{.}\; f_2 \;\texttt{.}\; f_1)[f_\delta]\texttt{;}\; \epsilon \;\texttt{:}\; f_\epsilon,\, \delta)\end{aligned}$$

The set of such rules is infinite, but this does not seem to be a new formal problem, since even without such rules our system is defined using an infinite set of basic rewrite rules, because of the rules involving arbitrary indexed lists. However, the proposed rule creates unsolvable critical pairs because the functor compositions may be reduced before the rule is applied. Then the mapping is reduced using the other rules, given below, resulting in an η-expansion of the projection, instead of the actual projection seen on the right hand side of the proposed rule. To maintain confluence, we propose the following rule, where all terms in the composition are required to be projections and so the composition is in the normal form.

$$\begin{aligned}&\texttt{T\_map}(\texttt{F\_PR}(lc, \kappa) \;\texttt{.}\; i_n \;\texttt{.}\; \ldots \;\texttt{.}\; i_2 \;\texttt{.}\; i_1, t) \;\; \rightarrow \\ &\;\; \texttt{T\_pr}(\delta \;\texttt{:}\; (i_n \;\texttt{.}\; \ldots \;\texttt{.}\; i_2 \;\texttt{.}\; i_1)[f_\delta]\texttt{;}\; \epsilon \;\texttt{:}\; f_\epsilon,\, \delta)\end{aligned} \tag{117}$$

The remaining rules for mapping are very similar to the corresponding axioms of Φ_i. First, we see a directed equation (95).

$$\texttt{T\_map}(\texttt{F\_PR}(lc, \iota), t) \;\; \rightarrow \;\; t \tag{118}$$

The rules for mapping by a product or a sum are just directed equations (97) and (98).

$$\texttt{T\_map(\{}lg\texttt{\}},t) \;\rightarrow\; \texttt{\{}i\texttt{ = \{}\delta\texttt{ = }\delta\texttt{ . }i\texttt{; }\epsilon\texttt{ = }\epsilon\texttt{\} . T\_map(}g_i,t\texttt{); }\ldots\texttt{\}} \tag{119}$$

$$\texttt{T\_map([}lg\texttt{]},t) \;\rightarrow\; \texttt{[`}i\texttt{ T\_map(}g_i,t\texttt{) . T\_in(}lh,i\texttt{)| }\ldots\texttt{]} \tag{120}$$

The auxiliary functors g, g_f, g_a, etc. in the next two rules are as defined in the previous section. These rules, too, are the directed definitional equations.

$$\begin{aligned}\texttt{T\_map(F\_ii(}g\texttt{)},t) \;\rightarrow\;& \texttt{TL\_fold(}g_f,(g_a * \texttt{T\_map(}g_p,\kappa * t))\\ & \texttt{. T\_con(}g_h))\end{aligned} \tag{121}$$

$$\begin{aligned}\texttt{T\_map(F\_tt(}g\texttt{)},t) \;\rightarrow\;& \texttt{TL\_unfold(}g_h,\texttt{\{}\delta\texttt{ = }\delta\texttt{ . T\_unde(}g_f\texttt{); }\epsilon\texttt{ = }\epsilon\texttt{\}}\\ & \texttt{. }(g_c * \texttt{T\_map(}g_p,\kappa * t)))\end{aligned} \tag{122}$$

Before we continue the definition of $\mathcal{R}_i$ let us formalize the fact that the rules for the mapping operation constitute its complete inductive definition in terms of the other combinators.

Lemma 6.3.6. *Every ground* `trans`*-term* $\mathcal{R}_i$*-reduces in a finite number of steps to a term without any* `T_map` *constructor.*

Proof. First, notice that for every possible normal form (ground) `funct`-term

$$g : \texttt{<}\iota\texttt{ - }c\texttt{; }\kappa\texttt{ - }d\texttt{>} \rightarrow c$$

there is a corresponding elimination rule for `T_map`(g,t). This means that the rules for `T_map` and the rules for the functors suffice here.

The proof is by induction on a certain partial order on `funct`-terms. The induction hypothesis for a `funct`-term g states that `T_map`(g,t) $\mathcal{R}_i$-reduces in a finite number of steps to a term with no `T_map` constructors. The order on g contains the subterm relation, but for the cases of rule (121) and (122), additionally a `funct`-term is considered smaller than any term with a larger sum of occurrences of the `F_ii` or `F_tt` constructors. With this order the mappings at the right hand sides of the rules are by strictly smaller functors than at the left hand sides. □

The action of the (co)inductive functor on morphisms will be reduced in $\mathcal{R}_i$ to folding or unfolding, just as follows from the axioms of Φ_i.

$$\texttt{T\_ii(}t) \;\rightarrow\; \texttt{T\_fold(}f,t[\texttt{F\_ii(}h)]\texttt{ . T\_con(}h)) \tag{123}$$

$$\texttt{T\_tt(}t) \;\rightarrow\; \texttt{T\_unfold(}h,\texttt{T\_unde(}f)\texttt{ . }t[\texttt{F\_tt(}f)]) \tag{124}$$

The simple folding and unfolding, in turn, will be expressed using the generalized folding and unfolding. These are the first rules which are not just directed axioms of Φ_i, in which the non-distributive operations have been considered the base ones.

$$\texttt{T\_fold}(g,t) \rightarrow \texttt{\{}\delta \texttt{ = T\_id(F\_ii(}g\texttt{)); } \epsilon \texttt{ = \{\}\} . TL\_fold(}g,\delta \texttt{ . } t\texttt{)} \tag{125}$$
$$\texttt{T\_unfold}(g,t) \rightarrow \texttt{\{}\delta \texttt{ = T\_id(}h\texttt{); } \epsilon \texttt{ = \{\}\} . TL\_unfold(}g,\delta \texttt{ . } t\texttt{)} \tag{126}$$

Here are two variants of the main reduction rule for inductive types; the first is necessary, because t_1 can be an identity and then the second rule together with rule (42) causes a critical pair. While the equation describing the reduction of simple folding expresses the result of the reduction using horizontal composition, the reduction of distributive folding results in a term with mapping. This agrees with the view of mapping as a distributive generalization of the horizontal composition, as shown before.

$$\begin{gathered}\texttt{\{}\delta \texttt{ = T\_con(}g\texttt{); } \epsilon \texttt{ = } t\texttt{\} . TL\_fold(}g,u\texttt{)} \rightarrow \\ \texttt{\{}\delta \texttt{ = \{}\delta \texttt{ = T\_id(}f\texttt{); } \epsilon \texttt{ = } t\texttt{\} . T\_map(}g,\texttt{TL\_fold(}g,u\texttt{)); } \epsilon \texttt{ = } t\texttt{\} . } u\end{gathered} \tag{127}$$
$$\begin{gathered}\texttt{\{}\delta \texttt{ = } t_1 \texttt{ . T\_con(}g\texttt{); } \epsilon \texttt{ = } t\texttt{\} . TL\_fold(}g,u\texttt{)} \rightarrow \\ \texttt{\{}\delta \texttt{ = \{}\delta \texttt{ = } t_1\texttt{; } \epsilon \texttt{ = } t\texttt{\} . T\_map(}g,\texttt{TL\_fold(}g,u\texttt{)); } \epsilon \texttt{ = } t\texttt{\} . } u\end{gathered} \tag{128}$$

Similarly, there are two variants of the main reduction rule for coinductive type. Both of them are needed, because the vertical composition is rewritten as left-associative.

$$\begin{gathered}\texttt{TL\_unfold(}g,t\texttt{) . T\_unde(}g\texttt{)} \rightarrow \\ \texttt{\{}\delta \texttt{ = } t\texttt{; } \epsilon \texttt{ = } \epsilon\texttt{\} . T\_map(}g,\texttt{TL\_unfold(}g,t\texttt{))}\end{gathered} \tag{129}$$
$$\begin{gathered}(t_1 \texttt{ . TL\_unfold(}g,t\texttt{)) . T\_unde(}g\texttt{)} \rightarrow \\ t_1 \texttt{ . \{}\delta \texttt{ = } t\texttt{; } \epsilon \texttt{ = } \epsilon\texttt{\} . T\_map(}g,\texttt{TL\_unfold(}g,t\texttt{))}\end{gathered} \tag{130}$$

We provide direct reduction rules for the auxiliary construction and destruction operations. We could have defined `T_de` and `T_uncon` in terms of other operations (as in the axioms of Φ_i), but the rewriting would then be less efficient.

$$\texttt{T\_con(}g\texttt{) . T\_de(}g\texttt{)} \rightarrow \texttt{T\_id(}g\texttt{[F\_ii(}g\texttt{)])} \tag{131}$$
$$(t_1 \texttt{ . T\_con(}g\texttt{)) . T\_de(}g\texttt{)} \rightarrow t_1 \tag{132}$$
$$\texttt{T\_uncon(}g\texttt{) . T\_unde(}g\texttt{)} \rightarrow \texttt{T\_id(}g\texttt{[F\_tt(}g\texttt{)])} \tag{133}$$
$$(t_1 \texttt{ . T\_uncon(}g\texttt{)) . T\_unde(}g\texttt{)} \rightarrow t_1 \tag{134}$$

We need a way to instantiate the new morphisms with arbitrary functors.

$$f * \texttt{T\_con(}g\texttt{)} \rightarrow \texttt{T\_con(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g\texttt{)} \tag{135}$$
$$f * \texttt{TL\_fold(}g,t\texttt{)} \rightarrow \texttt{TL\_fold(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g, f * t\texttt{)} \tag{136}$$

$$
\begin{aligned}
f * \texttt{T\_de}(g) &\to \texttt{T\_de(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g) && (137)\\
f * \texttt{T\_uncon}(g) &\to \texttt{T\_uncon(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g) && (138)\\
f * \texttt{TL\_unfold}(g, t) &\to \texttt{TL\_unfold(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g, f * t) && (139)\\
f * \texttt{T\_unde}(g) &\to \texttt{T\_unde(<}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \texttt{ . } f\texttt{> . } g) && (140)
\end{aligned}
$$

We conclude the definition of $\mathcal{R}_i$ by showing how to multiply by the (co)inductive functors from the right.

$$
\begin{aligned}
t * \texttt{F\_ii}(g) &\to \texttt{T\_ii(<}\iota \texttt{ = } \iota\texttt{; } \kappa \texttt{ = } \kappa * t\texttt{>} * g) && (141)\\
t * \texttt{F\_tt}(g) &\to \texttt{T\_tt(<}\iota \texttt{ = } \iota\texttt{; } \kappa \texttt{ = } \kappa * t\texttt{>} * g) && (142)
\end{aligned}
$$

Conjecture 6.3.7. *System $\mathcal{R}_i$ is strongly normalizing.*

We would like to prove the conjecture by extending the proof of Theorem 6.2.5 for the cases of the additional rewriting rules, but here the lexicographic path ordering method fails. The problematic rules are 127–130, as well as 121–122 in the presence of 114–115. The other rules are easy to cover using precedences, as before.

We expect we could generate a very long proof using the reducibility candidates method due to Tait and Girard, basing our efforts on proofs of termination for similar rewriting systems. However our system is both much bigger than the related systems and somewhat simpler, because we use combinators instead of variable bindings and because our calculus is first-order (it will be extended in subsequent chapters, though). Perhaps a semi-automated approach using one of the existing proof assistants for rewriting systems [61] is a better idea. However, this necessitates a laborious extension of our rewriting system with explicit operations of domain and codomain of transformation and with indexed lists operations, so we leave that for future work.

Theorem 6.3.8. *System $\mathcal{R}_i$ is locally confluent.*

Proof. We will base our proof of confluence on that for system $\mathcal{R}_s$ and analyze only the newly introduced critical pairs. There are four new critical pairs in the `funct` part of the rewriting system. Two occur if the functor f in rules (114) or (115) is an identity and two when a term is composed with the left hand side of either rule. All four are easily solved. In the `trans` part of the system, rules 114–115 form critical pairs when a `trans` term is multiplied from the right by their left hand side. Rules (141) and (142) solve these critical pairs.

Rule (117) is designed specifically so as to prevent critical pairs involving mapping. Similarly rules 118–126 and 135–142 do not incur any new nontrivial critical pairs. The critical pairs stemming from rules 127–134 are solved similarly as in $\mathcal{R}_s$. □

We conjecture that analogously as for s-Core there is an alternative, programming-oriented axiomatization of i-Core taking the distributive (co)iterators and their computational meaning as its starting point. The reduction system $\mathcal{R}_i$ suggests an outline of such an axiomatization, but we have problems in proving soundness even of this outline. The long proof of the following theorem, using Lemma 6.3.3 and Observation 6.3.5, is given in Appendix B.

Lemma 6.3.9. *Ground instances of all rules of $\mathcal{R}_i$ except rules 127–128 are sound with respect to Φ_i.*

The proof is quite difficult, because it uses extensively the conditional equations and because we use the unconditional equations both to rewrite in the usual, intuitive direction — simplifying terms similarly as in our rewriting system — and in the reverse direction — temporarily complicating terms. The same equation is sometimes needed in both roles in the same derivation. This indicates that if a complete rewriting system for the theory of *2-LC-lc* with sums and (co)algebras exists, it is probably substantially different from $\mathcal{R}_i$. We are unable to prove that rules 127–128 follow from Φ_i, even if we restrict ourselves to their ground instances. Nevertheless, we are convinced the following proposition holds.

Conjecture 6.3.10. *Rules 127–128 are sound with respect to the theory of 2-LC-lc with sums and (co)algebras. Consequently, the whole system $\mathcal{R}_i$ is sound with respect to the theory of 2-LC-lc with sums and (co)algebras.*

In this version of the system there are no rules corresponding to substitution into `TL_fold` and `TL_unfold`. If we wanted to perform the substitutions we would have to prove that the following equations are consequences of Φ_i. This is easy but involves the axioms of Φ_i that are conditional equations.

$$\begin{aligned}
\texttt{\{}\delta\ \texttt{=}\ \delta\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ t\texttt{\}}\ \texttt{.}\ \texttt{TL\_fold}(g,u) &= \\
\texttt{TL\_fold}(g,\texttt{\{}\delta\ \texttt{=}\ \delta\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ t\texttt{\}}\ \texttt{.}\ u) & \\
\texttt{\{}\delta\ \texttt{=}\ \delta\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ t\texttt{\}}\ \texttt{.}\ \texttt{TL\_unfold}(g,u) &= \\
\texttt{TL\_unfold}(g,\texttt{\{}\delta\ \texttt{=}\ \delta\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ t\texttt{\}}\ \texttt{.}\ u) &
\end{aligned}$$

Then we could add rewrite rules based on these equations.

$$\begin{aligned}
\texttt{\{}\delta\ \texttt{=}\ t_1\texttt{;}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ \texttt{TL\_fold}(g,u) &\rightarrow \\
\texttt{\{}\delta\ \texttt{=}\ t_1\texttt{;}\ \epsilon\ \texttt{=}\ \texttt{T\_id}(f)\texttt{\}}\ \texttt{.}\ \texttt{TL\_fold}(g,\texttt{\{}\delta\ \texttt{=}\ \delta\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ t\texttt{\}}\ \texttt{.}\ u) & \qquad (143) \\
\texttt{\{}\delta\ \texttt{=}\ t_1\texttt{;}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ \texttt{TL\_unfold}(g,u) &\rightarrow \\
\texttt{\{}\delta\ \texttt{=}\ t_1\texttt{;}\ \epsilon\ \texttt{=}\ \texttt{T\_id}(h)\texttt{\}}\ \texttt{.}\ \texttt{TL\_unfold}(g,\texttt{\{}\delta\ \texttt{=}\ \delta\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ t\texttt{\}}\ \texttt{.}\ u) & \qquad (144)
\end{aligned}$$

Note that these reductions loop, whenever term t is an identity.

6.3.4 Conclusion

We have demonstrated several important properties of the mapping operation (Lemma 6.3.3, Corollary 6.3.4, Observation 6.3.5) and guided by them we have constructed a reduction system for the (co)inductive operations proving some partial results about the system (Theorem 6.3.8, Lemma 6.3.9). The languages described in this chapter do not resort to a meta-theory for mechanisms such as substitution or instantiation. Everything is modeled internally and explicitly. By building the equational theory of (co)inductive types on the basis of the multiplication and mapping operations we adhere to this principle. We capture polytypism (generic structured recursion) purely at the semantic level, without any meta-programming mechanisms.

While a programming language with such a degree of explicitness is not usable by humans without some syntactic sugar, it is perfectly amenable to modularization (see Chapter 5). The notions of dependency needed by a module language are here neither vague theoretical concepts nor mechanisms idiosyncratic to a favored execution model, but are concrete parts of the core language itself, general but necessarily compatible with whatever may be expressed in the language.

The availability of both inductive and coinductive types over sums and products makes i-Core a very expressive language. Like in Charity [24], alternating between inductive and coinductive types allows a programmer to express much more than just primitive recursive functions, at the same time assuring termination and allowing comprehensive optimization. On the other hand, the lack of exponential objects, which will be introduced in Section 8.1, makes this language rather weak as a core language for a module system. Without exponential objects, no folding or unfolding by morphisms from other modules is possible (unless the modules are themselves inductive in a strong sense, as proposed at the beginning of Section 9.1).

Part III

Variants and extensions

Chapter 7

Variants of the base categorical model

Variants of the Dule base categorical model are those mechanisms that have not been included in the implemented version of the core language. Therefore a reader interested in Dule from the programming perspective may skip this chapter on the first reading. Of all the other chapters of the thesis, only Chapter 8 refers extensively to some of the sections describing the variants. Nevertheless, the choice of axioms and rewriting rules throughout the thesis may look unclear and arbitrary without the framework developed below.

7.1 Adjunctions

The variants in this section are not proper programming languages because rewriting of their terms is problematic. However, the presented constructions are very general. They capture, using a single notion, most of the features of the core language categories described so far (see Section 7.1.3). Adjunctions provide insight into the fundamental mechanisms and allow for generic proofs of their common properties. Moreover they suggest, justify and systematize the construction of rewriting rules, especially the non-standard ones.

7.1.1 *2-LC-A* — *2-LC* with adjunctions

A *2-LC-A* is a *2-LC* with partial adjunction operations. More precisely, there is a partial operation of a right adjoint, of a unit of the adjunction and of a co-unit, and an analogous set of operations for the left adjoint.

By adjunctions we mean here the obvious categorical generalization, from **Cat** to arbitrary 2-category, of the notion of adjunction. Instead of describing the adjunctions by referring to the morphisms of 0-cells seen as categories, we describe them by equations on 2-cells representing the "morphisms" inside 0-cells. Moreover, the definedness of particular adjoints is not determined by properties of particular 0-cells, but given as a part of the algebraic structure. In particular, the reader can safely think about described operations and equations as pertaining to adjunctions in **Cat**, but he shouldn't assume that the set of defined adjoints in the model at hand coincides with the set of adjoints that exist in **Cat**.

Syntax

To the syntax of *2-LC* we append term constructors for various operations constituting adjunctions.

```
type cat =
  | C_PP of cat IList.t
  | C_BB
type funct =
  | F_ID of cat
  | F_COMP of funct * funct
  | F_PR of cat IList.t * IdIndex.t
  | F_RECORD of cat * funct IList.t
  | F_ri of funct
  | F_le of funct
type trans =
  | T_ID of cat
  | T_COMP of trans * trans
  | T_PR of cat IList.t * IdIndex.t
  | T_RECORD of cat * trans IList.t
  | T_FT of funct * trans
  | T_TF of trans * funct
  | T_id of funct
  | T_comp of trans * trans
  | T_repsilon of funct
  | T_reta of funct
  | T_lepsilon of funct
  | T_leta of funct
```

To improve the readability of equations and rules we will usually abbreviate `T_repsilon`(g) to $\varepsilon_{\mathtt{r}}(g)$, `T_reta`$(g)$ to $\eta_{\mathtt{r}}(g)$, `T_lepsilon`(g) to $\varepsilon_{\mathtt{l}}(g)$ and `T_leta`(g) to $\eta_{\mathtt{l}}(g)$. There is no analogy between this notation and the traditional names of reduction rules and equations, such as η-rule of λ-calculus or weak η-equation of Section 6.1.2. Any other adjunction constructors will be written in the abstract syntax defined above.

Semantics

We define the semantics of new term constructors. The operations they denote are partial. In particular all the new operations may be totally undefined, even though for some functors, like the identity functor, there are adjoints in every *2-LC-A*.

If an operation is defined for a particular functor, the choice of an adjunction is arbitrary, but should be consistent with the choice for the other two adjunction constituents, which then must be defined, and vice versa. As before, type-checking of `trans`-terms depends only on the semantics of `funct`-terms and `cat`-terms.

- `F_ri`$(g) : e \to c$
  is a generalized right adjoint to $g : c \to e$,
- `F_le`$(g) : e \to c$
  is a generalized left adjoint to $g : c \to e$,
- `T_repsilon`(g) : `F_ri`(g) `.` $g \to$ `F_ID`(e)
  is the co-unit of a generalized adjunction in which $g : c \to e$ is the left adjoint and `F_ri`(g) is the right adjoint,
- `T_reta`(g) : `F_ID`$(c) \to g$ `.` `F_ri`(g)
  is the unit of a generalized adjunction between $g : c \to e$ and `F_ri`(g),
- `T_lepsilon`$(g) : g$ `.` `F_le`$(g) \to$ `F_ID`(c)
  is the co-unit of a generalized adjunction in which `F_le`(g) is the left adjoint and $g : c \to e$ is the right adjoint,
- `T_leta`(g) : `F_ID`$(e) \to$ `F_le`(g) `.` g
  is the unit of a generalized adjunction between `F_le`(g) and $g : c \to e$.

Note that we do not assume that, e.g., `F_le`(`F_ri`(g)) is equal to g. The choice of adjoints is truly arbitrary. Neither do we require that the adjunction operations interchange or distribute over the product operations of *2-LC* as was required of the explicit compound products in Section 6.1.5. However such additional properties may be realized by a careful choice of adjunction operations for the particular categories.

An example of a *2-LC-A* is **Cat** with a two-object discrete category as the base category `*`. In this *2-LC-A* there can be no interesting expressible adjunctions (because the only expressible objects of this *2-LC-A* are labeled products of `*`, which are obviously discrete, and so every expressible functor has a trivial morphism part). But to make the definition of *2-LC-A* complete, we have to decide, which of the boring adjunctions are to be defined. Let us decide that no left adjoints will be defined and only one right adjoint will be defined.

We declare the morphism `F_ID(*)` to be the value of the right adjoint operation on `F_ID(*)`. In such primitive adjunctions the units and co-units have no more meaning than being witnesses of isomorphisms. And in this contrived category they are just identities, so we have no other choice than to declare the identities as the unit and co-unit for the adjunction.

The initial *2-LC-lc* is another example of a *2-LC-A*, where the operations of labeled products given by the *2-LC-lc* structure can serve to define a family of adjunctions. See Section 7.1.3 for more details.

A *2-LC-A* in which all the type-correct terms are defined would necessarily be trivial (at most one transformation with a given domain and codomain). Even the reasonably looking requirement that a category has at once products, coproducts, exponents and coexponents is enough to ensure degeneration, see Section 7.1.3.

Here are the standard diagrams depicting the triangular identities for units and co-units of an adjunction.

```
        F_ri(g) * T_reta(g)
F_ri(g) - - - - - - - - - -> F_ri(g) . g . F_ri(g)
     - _                             |
          - _                        |
               - _                   | T_repsilon(g) * F_ri(g)
    T_id(F_ri(g))   - _              |
                         - _         V
                              - > F_ri(g)
```

```
                g - _
                |       - _
                |             - _    T_id(g)
T_reta(g)  *  g |                   - _
                |                         - _
                V                               - _\
    g . F_ri(g) . g  - - - - - - - - - - - - - - - >  g
                        g * T_repsilon(g)
```

We draw the diagrams only for the `T_repsilon`(g) and `T_reta`(g) operations, since the diagrams for the other set of operations are nearly identical; see the equations below.

Equations

Recall that type-incorrect terms have been banned from equations. Note that this time a term may be type-correct but still undefined. Moreover, we cannot ban undefined terms from equations, because definedness of terms now depends on models. Without a way to know a model in advance we would have to ban all the adjunction terms, making illegal any equations describing adjunctions.

We allow undefined terms in equations and provide the definedness predicate to control definedness of particular terms. Our equations are conditional and if the premise holds, we interpret the equation using the existential equality of terms: an equation is satisfied in a model if and only if both sides are defined and equal. We declare that all the equations we write should be read as conditional equations with the single premise stating that both sides are defined.

The setup is similar to the strongest equality of terms that makes an equation satisfied if both sides are defined and equal or if any side is undefined. The main difference is that with the strongest equality the transitivity rule is conditional, depending on the definedness of terms (otherwise it would be unsound), while in our calculus the transitivity rule propagates definedness predicates of argument equations into the premise of the resulting equation. In this way the generated theories can contain many more (conditional) consequences and so a single theory describes many algebras differing in their partiality.

The overall intended meaning of this setting is that an algebra is not required to have an adjunction for a particular functor, but if it claims to have one, it should be correct according to the equations below. The definedness predicates do not endanger the existence of the initial *2-LC-A* algebras, because here and to the end of our paper the predicates are used only as premises of the conditional axioms.

These are the two axioms corresponding to the triangular identities of adjunctions [133].

$$(\mathtt{F\_ri}(g) * \eta_{\mathtt{r}}(g))\ \mathtt{.}\ (\varepsilon_{\mathtt{r}}(g) * \mathtt{F\_ri}(g)) = \mathtt{T\_id}(\mathtt{F\_ri}(g)) \tag{145}$$

$$(\eta_{\mathtt{r}}(g) * g)\ \mathtt{.}\ (g * \varepsilon_{\mathtt{r}}(g)) = \mathtt{T\_id}(g) \tag{146}$$

Below are the mirror images of the previous axioms, needed for the left adjoint operations.

$$(g * \eta_{\mathtt{l}}(g))\ \mathtt{.}\ (\varepsilon_{\mathtt{l}}(g) * g) = \mathtt{T\_id}(g) \tag{147}$$

$$(\eta_{\mathtt{l}}(g) * \mathtt{F\_le}(g))\ \mathtt{.}\ (\mathtt{F\_le}(g) * \varepsilon_{\mathtt{l}}(g)) = \mathtt{T\_id}(\mathtt{F\_le}(g)) \tag{148}$$

Let Φ_D be a set of formulae stating that whenever one constructor of a given adjunction is defined then all three should be defined and that all the operations from the language of *2-LC* are always defined. The theory generated by the four axioms above, all the axioms of theory Φ_2 of *2-LC* (all seen as conditional equations with premises stating that both sides are defined) and by Φ_D will be called Φ_A. This theory is very weak in the sense that, for example, an algebra with all adjunction operations undefined satisfies Φ_A. We will often strengthen the theory using the definedness predicate.

Let T be a set of *2-LC-A* terms (usually with variables). The theory generated by Φ_A and a set of formulae stating that all the terms in T are defined (for all defined valuations of variables) will be called $\Phi_A \cup T$. These theories are large enough and diverse enough for the study of many abstract properties of adjunctions (but not the properties of any single concrete adjunction). They also contain the boundary cases, for example the theory $\Phi_A \cup T$ where T contains only the type-correct terms (with variables) of the language of *2-LC* is the theory of the class of all *2-LC-A*. Another example is the class of *2-LC-A* freely generated from arbitrary sets of constants (e.g., a quotient of a term algebra). This class is described by $\Phi_A \cup T$ where T contains all the type-correct terms (with variables) of the language of *2-LC-A*. And then $\Phi_A \cup T$ is a trivial theory (equating all `trans`-terms with the same domain and codomain). Other, more natural examples are given in Section 7.1.3.

The following observation can be seen as a definition of *2-LC-A*. Alternatively, if we define *2-LC-A* as the generalization of **Cat** and its adjunctions, the observations becomes a theorem that Φ_A correctly equationally describes **Cat**'s adjunctions (which is a well-known fact [133]) and that Φ_A captures the requirement that adjunction operations for any given adjunction are either all defined or undefined, which is true by the inclusion of Φ_D.

Observation 7.1.1. *Φ_A is the theory of 2-LC-A.*

Unfortunately no good reduction system is known for the adjunctions in full generality and full strength. Some limited hints will be given in Section 7.1.3 and 7.1.4, where the adjunctions are presented using factorizers. However, if only a part of the equational theory is used for the rewriting, or only a small class of terms is considered, computation by rewriting is possible. Our equational presentation of adjunctions, especially the one we develop in the next section, helps to design the set of reduction rules in a uniform and exhaustive manner.

7.1.2 *2-LC-F* — *2-LC-A* by factorizers

2-LC-F is a *2-LC-A* where two additional operations — the factorizers defined in terms of the other operations — are added for all the existing adjunctions. Factorizers provide the unique morphisms of the definition of a (co)free object. Another name for the factorizer of t is the transpose of t, yet another is the adjunct of t.

When the transformations are built only with units and co-units, the resulting terms are unnecessarily large and contain excessive amounts of multiplication by a functor from the right. In a programming language such a multiplication seems exotic and potentially dangerous. It can change the target category of a value (a transformation with target $*$), thus moving it beyond the realm of programming language values. The availability of factorizers permits succinct notation and eliminates the multiplication from the right.

Syntax

We enrich the syntax of *2-LC-A* by the following terms, called right and left factorizers.

```
type trans =
  ...
  | T_rfact of funct * funct * trans
  | T_lfact of funct * funct * trans
```

See the example instances in Section 7.1.3 for an idea of the intended meaning of these two terms. In short, they play the role of the operation accumulating components (record, case expression, currying), as opposed to the constants triggering reductions (projection, injection, application morphisms) modeled by (co)units.

Semantics

The operations known from *2-LC-A* are defined as before and the semantics of the factorizers is given using these operations. When the operations are defined the corresponding factorizer is defined, when they are undefined the corresponding factorizer is undefined as well.

- $\texttt{T\_rfact}(g, f, t) : f \to h \ . \ \texttt{F\_ri}(g)$
  is the same transformation as
  $(f * \texttt{T\_reta}(g)) \ . \ (t * \texttt{F\_ri}(g))$,
  where $g : c \to e$, $f : a \to c$, $h : a \to e$ and $t : f \ . \ g \to h$,

- $\texttt{T\_lfact}(g, h, t) : f \ . \ \texttt{F\_le}(g) \to h$
  is the same transformation as
  $(t * \texttt{F\_le}(g)) \ . \ (h * \texttt{T\_lepsion}(g))$,
  where $g : c \to e$, $h : a \to c$, $f : a \to e$ and $t : f \to h \ . \ g$.

Equations

While the definition of semantics of *2-LC-F* has been an extension of the definition for *2-LC-A*, the equational theory of *2-LC-F* will not be based on the equational theory for *2-LC-A*. Instead we will start with axioms of the theory Φ_2 of *2-LC* and add six new axioms. Remember that axioms should be read as conditional equations with a premise that both sides are defined.

The first two axioms state that g has a right adjoint, in terms of factorizers. The next two axioms express that g has a left adjoint, again using factorizers.

$$\begin{aligned}
\texttt{T\_rfact}(g, f, (u * g)\ \texttt{.}\ (h * \varepsilon_{\texttt{r}}(g))) &= u && (149)\\
(\texttt{T\_rfact}(g, f, t) * g)\ \texttt{.}\ (h * \varepsilon_{\texttt{r}}(g)) &= t && (150)\\
(f * \eta_{\texttt{l}}(g))\ \texttt{.}\ (\texttt{T\_lfact}(g, h, t) * g) &= t && (151)\\
\texttt{T\_lfact}(g, h, (f * \eta_{\texttt{l}}(g))\ \texttt{.}\ (u * g)) &= u && (152)
\end{aligned}$$

Equations (150) and (151) will be called existence requirements, while equations (149) and (152) will be called uniqueness requirements. They generalize the product equations of the same names, see Section 7.1.3. Notice that we do not use `T_reta` nor `T_lepsilon` term constructors here. In fact, they will be expressed using the factorizers, as follows.

$$\begin{aligned}
\texttt{T\_reta}(g) &= \texttt{T\_rfact}(g, \texttt{F\_ID}(c), \texttt{T\_id}(g)) && (153)\\
\texttt{T\_lepsilon}(g) &= \texttt{T\_lfact}(g, \texttt{F\_ID}(c), \texttt{T\_id}(g)) && (154)
\end{aligned}$$

Let Φ_D be a set of formulae stating that whenever one constructor of a given adjunction is defined then all four should be defined. The theory generated from the axioms of the theory Φ_2 of *2-LC*, the six equations above and Φ_D will be called Φ_F. If T is a set of *2-LC-F* terms, the theory generated by Φ_F and a set of formulae stating that all the terms in T are defined will be called $\Phi_F \cup T$.

Theorem 7.1.2. *Φ_F is contained in the theory of 2-LC-F.*

Proof. Semantically a *2-LC-F* is just a *2-LC-A* with two additional operations equationally defined (in the semantic definitions above). Let us consider a theory generated by the theory Φ_A of *2-LC-A* translated to the language of *2-LC-F* and the two equations from the semantics, defining factorizers. By Observation 7.1.1 we know that this theory is sound for a *2-LC-F*. Therefore when we show that Φ_F follows from this theory, the thesis will be proven. The proof is carried out in Appendix B. □

Theorem 7.1.3. *Φ_F entails the theory of 2-LC-F.*

Proof. Reasoning analogously as in the proof of Theorem 7.1.2 we reduce the problem to deriving from the axioms of Φ_F equations 145–148 and the two equations implied by the semantic definition of factorizers. The full proof is given in Appendix B. □

7.1.3 Example adjunctions

In this section we will see what the adjunction axioms look like when specialized and written in a sugared notation. We will also try to derive rewriting rules, resembling the β-rule of λ-calculus, from these axioms. More observations about rewriting and about η-rules for adjunctions will be made in Section 7.1.4.

Product

Let $\Delta = \texttt{<}i_1 \texttt{ : F\_ID(*); } \ldots \texttt{; } i_n \texttt{ : F\_ID(*)>}$. The functor $\texttt{F\_ri}(\Delta)$ (if defined in the category at hand) behaves similarly as the labeled product functor `F_pp` of Section 4.3.

Let a be a category, le be the indexed list $i_1 \texttt{ - *; } \ldots \texttt{; } i_n \texttt{ - *}$ and $f : a \to \texttt{*}$ be a functor. Let lh be a list $i_1 \texttt{ : } h_1\texttt{; } \ldots \texttt{; } i_n \texttt{ : } h_n$ where $h_i : a \to \texttt{*}$. Let lt be a list $i_1 \texttt{ = } t_1\texttt{; } \ldots \texttt{; } i_n \texttt{ = } t_n$ where $t_i : f \to h_i$. We may then reconstruct all the record operations as follows.

$$\begin{aligned}
\texttt{F\_pp}(a, lh) &= \texttt{F\_RECORD}(a, lh) \texttt{ . F\_ri}(\Delta) \\
\texttt{T\_pp}(a, lt) &= \texttt{T\_RECORD}(a, lt) * \texttt{F\_ri}(\Delta) \\
\texttt{T\_pr}(lh, j) &= \texttt{F\_RECORD}(a, lh) * \texttt{T\_repsilon}(\Delta) * \texttt{T\_PR}(le, j) \\
\texttt{T\_record}(f, lt) &= \texttt{T\_rfact}(\Delta, f, \texttt{T\_RECORD}(a, lt))
\end{aligned}$$

Let additionally $lf' = i_1 \texttt{ : } f_1\texttt{; } \ldots \texttt{; } i_n \texttt{ : } f_n$ where $f_i : a \to \texttt{*}$. Let $f' = \texttt{F\_RECORD}(a, lf')$. Let $u : f \to f' \texttt{ . F\_ri}(\Delta)$ and let $t = \texttt{T\_RECORD}(a, lt)$. We will now derive the two main product equations from the theory Φ_F. We specialize axiom (149) and (150) to functor Δ (and to f, f', u, t), obtaining

$$\begin{aligned}
\texttt{T\_rfact}(\Delta, f, (u * \Delta) \texttt{ . } (f' * \varepsilon_{\texttt{r}}(\Delta))) &= u \\
(\texttt{T\_rfact}(\Delta, f, t) * \Delta) \texttt{ . } (h * \varepsilon_{\texttt{r}}(\Delta)) &= t
\end{aligned}$$

Now we use the uniqueness requirement for product in the category of objects and 2-morphisms (equation (27), Section 6.1.4)

$$\begin{aligned}
\texttt{T\_rfact}(\Delta, f, (u * \Delta) \texttt{ . <}i \texttt{ = } f' * \varepsilon_{\texttt{r}}(\Delta) * \texttt{T\_PR}(le, i)\texttt{; } \ldots\texttt{>}) &= u \\
(\texttt{T\_rfact}(\Delta, f, t) * \Delta) \texttt{ . <}i \texttt{ = } h * \varepsilon_{\texttt{r}}(\Delta) * \texttt{T\_PR}(le, i)\texttt{; } \ldots\texttt{>} &= t
\end{aligned}$$

and then we identify and mark the `T_pr` operation

$$\begin{aligned}
\texttt{T\_rfact}(\Delta, f, (u * \Delta) \texttt{ . <}i \texttt{ = T\_pr}(lf', i)\texttt{; } \ldots\texttt{>}) &= u \\
(\texttt{T\_rfact}(\Delta, f, t) * \Delta) \texttt{ . <}i \texttt{ = T\_pr}(lh, i)\texttt{; } \ldots\texttt{>} &= t
\end{aligned}$$

Then we multiply by Δ, using its definition

$$\begin{aligned}
\texttt{T\_rfact}(\Delta, f, \texttt{<}i \texttt{ = } u\texttt{; } \ldots\texttt{> . <}i \texttt{ = T\_pr}(lf', i)\texttt{; } \ldots\texttt{>}) &= u \\
\texttt{<}i \texttt{ = T\_rfact}(\Delta, f, t)\texttt{; } \ldots\texttt{> . <}i \texttt{ = T\_pr}(lh, i)\texttt{; } \ldots\texttt{>} &= t
\end{aligned}$$

and we vertically compose records component-wise

$$\begin{aligned}
\texttt{T\_rfact}(\Delta, f, \texttt{<}i \texttt{ = } u \texttt{ . T\_pr}(lf', i)\texttt{; } \ldots\texttt{>}) &= u \\
\texttt{<}i \texttt{ = T\_rfact}(\Delta, f, t) \texttt{ . T\_pr}(lh, i)\texttt{; } \ldots\texttt{>} &= t
\end{aligned}$$

At last, we identify and mark T_record operations

$$\begin{aligned} \texttt{T\_record}(f,\, i = u\ .\ \texttt{T\_pr}(lf', i);\ \ldots) &= u \\ \texttt{<}i = \texttt{T\_record}(f, lt)\ .\ \texttt{T\_pr}(lh, i);\ \ldots\texttt{>} &= t \end{aligned}$$

and we obtained, essentially, the uniqueness and the existence requirements for labeled product (equations (49) and (48) of Section 6.1.5).

Observation 7.1.4. *Let T be the set consisting of right adjoints to all the Δ functors for all $i_1, \ldots, i_n$ labels. Then $\Phi_F \cup T$ translated to the special notation for products given above coincides with the theory Φ_p of 2-LC-lc.*

To get a β-rule for labeled products we have to multiply the just obtained existence requirement by F_PR(le, i). We then get

$$\texttt{T\_record}(f, lt)\ .\ \texttt{T\_pr}(lh, i) \;\rightarrow\; t_i$$

which is the same as the β-rule (13) of the reduction system $\mathcal{R}_p$.

Coproduct

Again let $\Delta =$ <i_1 : F_ID(*); ... ; i_n : F_ID(*)>. Then the functor F_le(Δ) has a similar meaning to the labeled coproduct functor F_ss of Section 6.2.1. If the category we work with has exponential object (see next section) the coproduct will be distributive. If not, it needn't be. Here we will not assume distributivity. Consequently, the adjunction is dual to the product adjunction of the previous section.

The (non-distributive) coproduct operations may be defined as follows.

$$\begin{aligned} \texttt{F\_ss}(a, lf) &= \texttt{F\_RECORD}(a, lf)\ .\ \texttt{F\_le}(\Delta) \\ \texttt{T\_ss}(a, lt) &= \texttt{T\_RECORD}(a, lt) * \texttt{F\_le}(\Delta) \\ \texttt{T\_in}(lf, j) &= \texttt{F\_RECORD}(a, lf) * \texttt{T\_leta}(\Delta) * \texttt{T\_PR}(le, j) \\ \texttt{T\_case}(lt, h) &= \texttt{T\_lfact}(\Delta, h, \texttt{T\_RECORD}(a, lt)) \end{aligned}$$

Note that the injection is (essentially) a unit of the adjunction and not a co-unit, as was the case for projections.

The existence and the uniqueness requirements for labeled coproduct are given below. They can be derived analogously as those for the labeled product.

$$\begin{aligned} \texttt{<}i = \texttt{T\_in}(lh', i)\ .\ \texttt{T\_case}(lt, h);\ \ldots\texttt{>} &= t \\ \texttt{T\_case}(i = \texttt{T\_in}(lf, i)\ .\ u;\ \ldots, h) &= u \end{aligned}$$

Observation 7.1.5. *Let T be the set consisting of all right and left adjoints to the Δ functors. Then $\Phi_F \cup T$, when translated to the special-notation for products and coproducts, coincides with a subset of theory Φ_c of Section 6.2.3 (the theory Φ_c additionally states the distributiveness of the coproduct over the product).*

To get a β-rule for coproducts first we have to multiply the existence requirement above by $\texttt{F\_PR}(le, i)$.

$$\texttt{T\_in}(lh', i)\ \texttt{.}\ \texttt{T\_case}(lt, h) \ = \ t_i$$

But as the vertical composition in $\mathcal{R}_2$ is left associative, we also have to add a more complicated version of the rule, to maintain confluence in the presence of rule (43).

$$(u\ \texttt{.}\ \texttt{T\_in}(lh', i))\ \texttt{.}\ \texttt{T\_case}(lt, h) \ \rightarrow \ u\ \texttt{.}\ t_i$$

If the coproduct is distributive there is much more to do, as seen in Section 6.2.3.

Exponent

Let $lg = i_1 : g_1 \texttt{;}\ \ldots \texttt{;}\ i_n : g_n$, where the variables g_i are typed as follows: $g_i : \texttt{<>} \rightarrow \texttt{*}$. Let υ be a distinguished label not equal to any of $i_1, \ldots, i_n$. Let

$$\mathbf{X}(lg) \ = \ \texttt{F\_pp}(\texttt{*},\ \upsilon : \texttt{F\_ID}(\texttt{*})\texttt{;}\ i_1 : \texttt{<>}\ \texttt{.}\ g_1\texttt{;}\ \ldots\texttt{;}\ i_n : \texttt{<>}\ \texttt{.}\ g_n)$$

The functor $\texttt{F\_ri}(\mathbf{X}(lg))$ will be called the labeled exponent functor. This adjunction differs from the standard exponent in that the parameters are multiple and are labeled by $i_1,\ \ldots,\ i_n$, and that, during application, the function placed at the component labeled υ. The functor corresponds to the labeled function types of the Dule programming language.

Let a be a category and $f, h : a \rightarrow \texttt{*}$. Let $t : f\ \texttt{.}\ \mathbf{X}(lg) \rightarrow h$ and $u : f \rightarrow h$. Now let us define some operations.

$$\begin{aligned}
\texttt{F\_ee}(lg, h) &= h\ \texttt{.}\ \texttt{F\_ri}(\mathbf{X}(lg)) \\
\texttt{T\_ee}(lg, u) &= u * \texttt{F\_ri}(\mathbf{X}(lg)) \\
\texttt{T\_appl}(lg, h) &= h * \texttt{T\_repsilon}(\mathbf{X}(lg)) \\
\texttt{T\_curry}(t) &= \texttt{T\_rfact}(\mathbf{X}(lg), f, t)
\end{aligned}$$

Let additionally $f' : a \rightarrow \texttt{*}$. Now let $u : f \rightarrow f'\ \texttt{.}\ \texttt{F\_ri}(\mathbf{X}(lg))$. We specialize equations (149) and (150) to the $\mathbf{X}(lg)$ functor.

$$\begin{aligned}
\texttt{T\_rfact}(\mathbf{X}(lg), f, (u * \mathbf{X}(lg))\ \texttt{.}\ (f' * \varepsilon_{\texttt{r}}(\mathbf{X}(lg))))) &= u \\
(\texttt{T\_rfact}(\mathbf{X}(lg), f, t) * \mathbf{X}(lg))\ \texttt{.}\ (h * \varepsilon_{\texttt{r}}(\mathbf{X}(lg)))) &= t
\end{aligned}$$

Now we change notation by introducing `T_curry` and `T_appl`

$$\begin{aligned}
\texttt{T\_curry}((u * \mathbf{X}(lg))\ \texttt{.}\ \texttt{T\_appl}(lg, f')) &= u \\
(\texttt{T\_curry}(t) * \mathbf{X}(lg))\ \texttt{.}\ \texttt{T\_appl}(lg, h) &= t
\end{aligned}$$

then we use the definition of the $\mathbf{X}(lg)$ functor

$$\begin{array}{rcl}\texttt{T\_curry(T\_pp(}a\texttt{,}\ \upsilon\ \texttt{=}\ u\texttt{;}\ i_1\ \texttt{= <>}*g_1\texttt{;}\ \ldots\texttt{) . T\_appl(}lg, f'\texttt{))} & = & u\\ \texttt{T\_pp(}a\texttt{,}\ \upsilon\ \texttt{= T\_curry(}t\texttt{);}\ i_1\ \texttt{= <>}*g_1\texttt{;}\ \ldots\texttt{) . T\_appl(}lg, h\texttt{)} & = & t\end{array}$$

At last, we notice that some of the transformations are identities

$$\begin{array}{rcll}\texttt{T\_curry(T\_pp(}a\texttt{,}\ \upsilon\ \texttt{=}\ u\texttt{;}\ i_1\ \texttt{= T\_id(<> .}\ g_1\texttt{);}\ \ldots\texttt{) . T\_appl(}lg, f'\texttt{))} & = & u & (155)\\ \texttt{T\_pp(}a\texttt{,}\ \upsilon\ \texttt{= T\_curry(}t\texttt{);}\ i_1\ \texttt{= T\_id(<> .}\ g_1\texttt{);}\ \ldots\texttt{) . T\_appl(}lg, h\texttt{)} & = & t & (156)\end{array}$$

and we have got the uniqueness (155) and the existence (156) requirements for labeled exponents.

Transforming the existence requirement (156) into a β-rule similar to the one usually employed for function types is not straightforward. First, we have to eliminate `T_pp` from the equation

$$\texttt{T\_pp(}a\texttt{,}\ \upsilon\ \texttt{= T\_curry(}t\texttt{);}\ i_1\ \texttt{= T\_id(<> .}\ g_1\texttt{);}\ \ldots\texttt{) . T\_appl(}lg, h\texttt{)} \quad = \quad t$$

using (53) and getting

$$\texttt{\{}\upsilon\ \texttt{=}\ \upsilon\ \texttt{. T\_curry(}t\texttt{);}\ i_1\ \texttt{=}\ i_1\texttt{;}\ \ldots\texttt{\} . T\_appl(}lg, h\texttt{)} \quad = \quad t$$

Let $t_i : f \to \texttt{<>} \ \texttt{.}\ g_i$. Now we compose vertically the term

$$\texttt{\{}\upsilon\ \texttt{= T\_id(}f\texttt{);}\ i_1\ \texttt{=}\ t_i\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{\}}$$

with both sides of the equation. We then simplify the left hand side in several steps using the equations for the labeled product and obtain the following equation.

$$\begin{array}{ll}\texttt{\{}\upsilon\ \texttt{= T\_curry(}t\texttt{);}\ i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{\} . T\_appl(}lg, h\texttt{)} & \\ =\ \texttt{\{}\upsilon\ \texttt{= T\_id(}f\texttt{);}\ i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{\} .}\ t & (157)\end{array}$$

Finally, the equation may be directed from left to right to become the β-rule for the labeled exponent.

$$\begin{array}{l}\texttt{\{}\upsilon\ \texttt{= T\_curry(}t\texttt{);}\ i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{\} . T\_appl(}lg, h\texttt{)}\\ \to\ \texttt{\{}\upsilon\ \texttt{= T\_id(}f\texttt{);}\ i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{\} .}\ t\end{array}$$

The β-equation (157) is quite different from the existence requirement (156). In fact, after many attempts, we are strongly convinced that the existence requirement can not be recovered from the uniqueness requirement (155) and the β-equation. Let T_Δ be a set containing right adjoints to all the Δ functors. Let $T_\mathbf{X}$ contain T_Δ and right adjoints to all the $\mathbf{X}(lg)$ functors.

Conjecture 7.1.6. *The theory generated from Φ_p, the uniqueness requirement (155) and β-equation (157) does not contain the (unconditional part of) theory $\Phi_F \cup T_{\mathbf{X}}$.*

However, when we supplement the uniqueness requirement (155) with an equation that will be used as a weak η-rule for the labeled exponent (see Section 7.1.4)

$$\texttt{T\_curry(T\_pp(}a\texttt{, }\upsilon\texttt{ = }u\texttt{; }i_1\texttt{ = T\_id(}f_1\texttt{); }\ldots\texttt{) . }t\texttt{)} \;=\; u\texttt{ . T\_curry(}t\texttt{)}$$

then the existence requirement (156) can be recovered from β-equation.

Lemma 7.1.7. *Theory $\Phi_F \cup T_{\mathbf{X}}$ coincides with the theory generated from Φ_p, the weak η-equation above, β-equation (157) and the uniqueness requirement (155) for labeled exponent.*

Proof. We have seen that the β-equation derives from the existence requirement (156). That the weak η-equation follows from the existence and uniqueness requirements is proved in the general case in Section 7.1.4. Here we will show how the existence requirement follows from β-equation and the weak η-equation.

Let us take equation (157) substituting $\texttt{F\_pp(}a, lg'\texttt{)}$ for f (where $lg' = \upsilon\ \texttt{:}\ f\texttt{;}\ lg$) substituting the term $\texttt{T\_pr(}lg', i\texttt{)}$ for t_i and the term

$$\texttt{T\_pp(}a\texttt{, }\upsilon\texttt{ = }\upsilon\texttt{; }i_1\texttt{ = T\_id(<> . }g_1\texttt{)); }\ldots\texttt{) . }t$$

for t and obtaining

$$\begin{aligned}
&\texttt{\{}\upsilon\texttt{ = T\_curry(T\_pp(}a\texttt{, }\upsilon\texttt{ = }\upsilon\texttt{; }i_1\texttt{ = T\_id(<> . }g_1\texttt{); }\ldots\texttt{) . }t\texttt{);}\\
&\ i_1\texttt{ = }i_1\texttt{; }\ldots\texttt{\} . T\_appl(}lg, h\texttt{)}\\
&= \ \texttt{\{}\upsilon\texttt{ = T\_id(F\_pp(}a, lg'\texttt{)); }i_1\texttt{ = }i_1\texttt{; }\ldots\texttt{\}}\\
&\quad\ \texttt{. T\_pp(}a\texttt{, }\upsilon\texttt{ = }\upsilon\texttt{; }i_1\texttt{ = T\_id(<> . }g_1\texttt{); }\ldots\texttt{) . }t
\end{aligned}$$

Now using the weak η-equation for exponents at the left hand side, and the uniqueness requirement for products (49) at the right hand side, we get

$$\begin{aligned}
&\texttt{\{}\upsilon\texttt{ = }\upsilon\texttt{ . T\_curry(}t\texttt{); }i_1\texttt{ = }i_1\texttt{; }\ldots\texttt{\} . T\_appl(}lg, h\texttt{)}\\
&= \ \texttt{\{}\upsilon\texttt{ = }\upsilon\texttt{; }i_1\texttt{ = }i_1\texttt{; }\ldots\texttt{\} . }t
\end{aligned}$$

and using again the uniqueness requirement for products at the right hand side and complicating the left hand side we obtain

$$\texttt{\{}\upsilon\texttt{ = }\upsilon\texttt{ . T\_curry(}t\texttt{); }i_1\texttt{ = }i_1\texttt{ . T\_id(<> . }g_1\texttt{); }\ldots\texttt{\} . T\_appl(}lg, h\texttt{)} \;=\; t$$

then using the equation that defines the action of the product functor, on morphisms at the left hand side of the equation, we finally get the existence requirement for labeled exponents (156)

$$\texttt{T\_pp(}a\texttt{, }\upsilon\texttt{ = T\_curry(}t\texttt{); }i_1\texttt{ = T\_id(<> . }g_1\texttt{); }\ldots\texttt{) . T\_appl(}lg, h\texttt{)} \;=\; t$$

□

As we have seen, the β-equation (157) follows from the existence requirement for labeled exponents (156) without the use of the uniqueness requirement (155) or any additional equation relating to exponents. On the other hand, to recover the existence requirement, the weak η-equation is, as we believe, essential. This would show that, in contrast to the product and coproduct cases, the β-equation for exponent is weaker than the original existence requirement. This discrepancy between the product and exponent suggests that if a general rewriting procedure for arbitrary adjoints exists, it cannot have the same β-rule for all (right) adjoints.

Remark. Note that semantics of terms containing labeled exponent operations is not stable under naive α-conversion (renaming labels inside `T_curry` operation). This could be expected, as the labeled exponent can be simulated by a combination of the standard exponent operations and records, which have field names that are never α-convertible. Similarly as in records, well chosen fixed labels of exponent arguments can be used as a valuable programming aid. On the other hand they limit the effectiveness of implementations based on hash-consing and memoization that are rather essential for a system with typed target language, such as the one we are developing. This conflict and its solutions will not be discussed further in this thesis.

The programming properties of the labeled exponent are similar to the labeled function types of OCaml. In fact, we use the OCaml syntax as a concrete syntax for the labeled exponent operations. The analogy is not a coincidence, because the adjunction we have chosen to model the labeled exponent seems to be the only reasonable choice. If we want to generalize the exponent construction in a similar way as the product construction has been generalized to labeled product, then the only adjunction with acceptable properties is the one we presented. One could even state this claim formally and with a proof, although the formalized version wouldn't appear so general nor so strong anymore. An informal argument is carried out in [95].

Coexponent

We see that all the labeled data type kinds we have introduced can be recovered using just the labeled `F_RECORD` and the mechanism of adjunctions. The question arises: are there any more interesting data types to be constructed this way?

The left adjoint to the $\mathbf{X}(lf)$ functor in not interesting, since in the presence of labeled products its only effect is to degenerate the `*` category (at most one transformation with a given domain and codomain with targets in `*`). But the list of interesting adjoints is not yet closed.

Let $lg = i_1 : g_1\texttt{;}\ \ldots\texttt{;}\ i_n : g_n$, where $g_i : \texttt{<>} \to \texttt{*}$. Let υ be a distinguished label not equal to any of $i_1, \ldots, i_n$. Let

$$\mathbf{U}(lg) \;=\; \texttt{F\_ss(*,}\ \upsilon : \texttt{F\_ID(*);}\ i_1 : \texttt{<>}\ .\ g_1\texttt{;}\ \ldots\texttt{;}\ i_n : \texttt{<>}\ .\ g_n)$$

Then the functor $\texttt{F\_le}(\mathbf{U}(lg))$ is a labeled coexponent functor.

A detailed discussion of this adjoint is out of the scope of our thesis, so we will mention only that it can be used to model continuations, exception passing and side-effects in general, as argued in [46]. Unfortunately the full equational theory of this adjunction, when coupled with the equations for the exponent makes the category `*` trivial.

We conjecture that removing the η-equations prevents trivialization, even if the substitution equations are added to compensate. An interesting question is whether every adjoint expressible in the language of *2-LC-F* has the property that it either degenerates the category `*` or is expressible in terms of the four fundamental adjoints described above and the language of *2-LC*.

7.1.4 Reduction in *2-LC-F*

The case of distributive coproduct and the case of exponent show that generating the β-rules from adjunction equations is far from automatic. Here we will show that, in contrast to β-rules, the weak η-rules are constructed from the axioms uniformly for all adjunctions. The weak η-rules are important especially for right adjunctions, as they model substitution into factorizers. After that we will capture the action of the right adjoint and left adjoint functors on transformations. Even when multiplication by a functor is not allowed in a programming language, the developed schema are still useful for the insight they provide into the structure of rules concerning the mapping operation of Section 6.3.1.

Two halves of the uniqueness requirement

Consider the uniqueness requirements for adjunctions, given in axioms (149) and (152).

$$\begin{aligned}
\texttt{T\_rfact}(g, f, (u * g)\ \texttt{.}\ (h * \varepsilon_{\texttt{r}}(g))) &= u \\
\texttt{T\_lfact}(g, h, (f * \eta_{\texttt{l}}(g))\ \texttt{.}\ (u * g)) &= u
\end{aligned}$$

We will split each of the equations in two “halves”, prove their properties and use one of them as a rewriting rule.

First, we construct the “half” of each of the uniqueness requirements that is easy to derive but computationally meaningless. We substitute identities for u in both the equations and simplify.

$$\texttt{T\_rfact}(g, h\ \texttt{.}\ \texttt{F\_ri}(g), h * \varepsilon_{\texttt{r}}(g)) = \texttt{T\_id}(h\ \texttt{.}\ \texttt{F\_ri}(g)) \qquad (158)$$

$$\texttt{T\_lfact}(g, f\ \texttt{.}\ \texttt{F\_le}(g), f * \eta_{\texttt{l}}(g)) = \texttt{T\_id}(f\ \texttt{.}\ \texttt{F\_le}(g)) \qquad (159)$$

The theory obtained from axioms of Φ_F, but with equations (149) and (152) replaced by equations (158) and (159), will be called Φ_{id}.

Here are some instances of these equations.

$$\begin{aligned}
\texttt{T\_record}(\texttt{F\_pp}(a, lh),\ i = \texttt{T\_pr}(lh, i);\ \ldots) &= \texttt{T\_id}(\texttt{F\_pp}(a, lh)) \\
\texttt{T\_case}(i = \texttt{T\_in}(lf, i);\ \ldots,\ \texttt{F\_ss}(a, lf)) &= \texttt{T\_id}(\texttt{F\_ss}(a, lf)) \\
\texttt{T\_curry}(\texttt{T\_appl}(lg, h)) &= \texttt{T\_id}(\texttt{F\_ee}(lg, h))
\end{aligned}$$

There is no obvious computational meaning to these equations so no variant of η-rule will be derived from them. Yet, they are the complementary "halves" to the important equations defined below.

Theorem 7.1.8. *The following equations, called weak η-equations, follow from theory Φ_F.*

$$\texttt{T\_rfact}(g, f, (u * g)\ .\ t) = u\ .\ \texttt{T\_rfact}(g, h', t) \tag{160}$$
$$\texttt{T\_lfact}(g, h, t\ .\ (u * g)) = \texttt{T\_lfact}(g, f', t)\ .\ u \tag{161}$$

Proof. We start by substituting the right hand sides of equations (160) and (161) for u into equations (149) and (152).

$$\begin{aligned}
&\texttt{T\_rfact}(g, f, ((u\ .\ \texttt{T\_rfact}(g, h', t)) * g)\ .\ (h * \varepsilon_{\texttt{r}}(g))) \\
&\qquad = u\ .\ \texttt{T\_rfact}(g, h', t) \\
&\texttt{T\_lfact}(g, h, (f * \eta_{\texttt{l}}(g))\ .\ ((\texttt{T\_lfact}(g, f', t)\ .\ u) * g)) \\
&\qquad = \texttt{T\_lfact}(g, f', t)\ .\ u
\end{aligned}$$

Then we distribute multiplication.

$$\begin{aligned}
&\texttt{T\_rfact}(g, f, (u * g)\ .\ (\texttt{T\_rfact}(g, h', t) * g)\ .\ (h * \varepsilon_{\texttt{r}}(g))) \\
&\qquad = u\ .\ \texttt{T\_rfact}(g, h', t) \\
&\texttt{T\_lfact}(g, h, (f * \eta_{\texttt{l}}(g))\ .\ (\texttt{T\_lfact}(g, f', t) * g)\ .\ (u * g)) \\
&\qquad = \texttt{T\_lfact}(g, f', t)\ .\ u
\end{aligned}$$

We conclude using the existence requirements, that is axioms (150) and (151).

$$\begin{aligned}
\texttt{T\_rfact}(g, f, (u * g)\ .\ t) &= u\ .\ \texttt{T\_rfact}(g, h', t) \\
\texttt{T\_lfact}(g, h, t\ .\ (u * g)) &= \texttt{T\_lfact}(g, f', t)\ .\ u
\end{aligned}$$

□

The theory obtained from axioms of Φ_F but with axioms (149) and (152) replaced by equations (160) and (161) will be called Φ_{comp}.

Here are some instances of equations (160) and (161).

$$\begin{aligned}
\texttt{T\_record}(f,\ i_1 = u\ .\ t_1;\ \ldots) &= u\ .\ \texttt{T\_record}(h', lt) \\
\texttt{T\_case}(i_1 = t_1\ .\ u;\ \ldots,\ h) &= \texttt{T\_case}(lt, f')\ .\ u \\
\texttt{T\_curry}(\texttt{T\_pp}(a,\ v = u;\ i_1 = \texttt{T\_id}(f_1);\ \ldots)\ .\ t) &= u\ .\ \texttt{T\_curry}(t)
\end{aligned}$$

When directed from right to left they are called weak η-rules of their respective adjunctions. The first rule is exactly rule (57) of Section 6.1.5, the second corresponds to rule (84) of Section 6.2.3 and the third is written without the T_pp notation (and in context of a more complicated adjunction) as rule (239) in Section 8.1.3.

Theorem 7.1.9. *The sum of Φ_{id} and Φ_{comp} generates the whole theory Φ_F.*

Proof. Axiom (149) can be recovered in the following way. First, put $h * \varepsilon_{\mathtt{r}}(g)$ for t in equation (160)

$$\mathtt{T\_rfact}(g, f, (u * g)\ \mathtt{.}\ (h * \varepsilon_{\mathtt{r}}(g))) \;=\; u\ \mathtt{.}\ \mathtt{T\_rfact}(g, h', h * \varepsilon_{\mathtt{r}}(g))$$

then simplify the right hand side using equation (158)

$$\mathtt{T\_rfact}(g, f, (u * g)\ \mathtt{.}\ (h * \varepsilon_{\mathtt{r}}(g))) \;=\; u\ \mathtt{.}\ \mathtt{T\_id}(h')$$

and eliminate the identity, obtaining equation (149).

Axiom (152) can be recovered analogously. Since only the consequences of these two equations have been lacking in Φ_{id} and Φ_{comp}, upon their recovery we may induce all Φ_F. □

Theorem 7.1.10. *The theories Φ_{id} and Φ_{comp} are incomparable and are proper sub-theories of Φ_F.*

Proof. It is enough to show that $\Phi_{id} \cup T$ is not contained in $\Phi_{comp} \cup T$, for some set of functors T, and that $\Phi_{comp} \cup U$ is not contained in $\Phi_{id} \cup U$, for some set of functors U.

Let T be a singleton set containing the term F_ri(<>), so that $\Phi_F \cup T$ corresponds to the theory of 2-categories with terminal object (labeled product with the empty set of labels). The instance of the existence requirement (150) for the terminal object is trivial, the instance of equation (160) for terminal object is

$$\mathtt{\{\}} \;=\; u\ \mathtt{.}\ \mathtt{\{\}}$$

and the instance of equation (158) for terminal object is

$$\mathtt{\{\}} \;=\; \mathtt{T\_id(\{\})}$$

To disprove the inclusion of $\Phi_{id} \cup T$ in $\Phi_{comp} \cup T$, let us consider a *2-LC* with the * category interpreted as **Set**. Let the funct-term {} : $c \to$ * be interpreted as a constant functor, always giving the set of integers and let the family of trans-terms {} : $f \to$ {}, represent the natural transformation, which for each set f is a constant function giving zero. It is easy to check that such category satisfies $\Phi_{comp} \cup T$. However, {} : {} $\to$ {} is here, by definition, different than T_id({}).

Unfortunately, for operations typed as those of terminal object, $\Phi_{comp} \cup T$ is contained in $\Phi_{id} \cup T$, so U has to be different than T. Let U contain all the adjunction terms that correspond to labeled products of functors (T_{Δ} from above). We will sketch the construction of a category $*$, which induces *2-LC* that satisfies $\Phi_{id} \cup U$ but not $\Phi_{comp} \cup U$.

Let the counter-example $*$ contain the complete lattice $\mathcal{P}(\mathbb{Z})$, where the only morphisms are the set inclusions. Moreover let there be a copy of each morphism of the lattice, with the copies of identities identified with the identities, and with the operation of composition always producing the original morphisms of the lattice, except when composing with an identity. Such definition of composition satisfies the axioms of *2-LC*.

The lattice has all labeled products, but for our category let the `T_record` operation result in the copies of the result morphisms of the original `T_record` of the lattice. Then by the identification of copies of identities with the identities themselves we have

$$\texttt{T\_record}(\texttt{F\_pp}(a, lh),\, i\ \texttt{=}\ \texttt{T\_pr}(lh, i)\texttt{;}\ \ldots) \quad = \quad \texttt{T\_id}(\texttt{F\_pp}(a, lh))$$

while by composition returning original morphisms we falsify

$$\texttt{T\_record}(f,\, i_1\ \texttt{=}\ u\ \texttt{.}\ t_1\texttt{;}\ \ldots) \quad = \quad u\ \texttt{.}\ \texttt{T\_record}(h', lt)$$

by any u with a domain different than its codomain. □

An interesting question is how the three theories compare if T is very rich, for example if T contains all the type-correct *2-LC-F* `trans`-terms. In other words, whether any half of the uniqueness requirement is enough to induce a collapse, such as the one in the coexponent case.

The following proposition shows that the weak η-rules (directed instances of equations (160) and (161)) interact badly in reduction systems with instances of both left and right adjunctions.

Lemma 7.1.11. *Coupling one or more weak η-rules for left adjunction with one or more weak η-rules for right adjunction together with a system $\mathcal{R}_2$ results in a reduction relation that is not confluent.*

Proof. This is a generalization of the remark about rule (84) in Section 6.2.3.

Let g_l be a term to which we have a left adjoint and g_r be a term to which we have a right adjoint. Then consider the following term.

$$\texttt{T\_lfact}(g_l, f, t_l)\ \texttt{.}\ \texttt{T\_rfact}(g_r, f, t_r)$$

Note that this term is always well-typed, because the functor f is common to both factorizers. We may use the weak η-rule for the adjunction based on g_r to perform the first reduction.

$$\texttt{T\_rfact}(g_r, f', (\texttt{T\_lfact}(g_l, f, t_l) * g_r)\ \texttt{.}\ t_r)$$

This term may be further rewritten, but there is no rule that could allow us to change the outermost term constructor to anything else than T_rfact. Alternatively we may use the weak η-rule for the adjunction based on g_l to perform the first reduction.

$$\texttt{T\_lfact}(g_l, h', t_l \texttt{ . } (\texttt{T\_rfact}(g_r, f, t_r) * g_l))$$

Again there is no way to change the outermost term constructor, hence the system is not confluent. □

Moreover, neither the β-rules derived for various adjunctions so far, nor any other rules in the thesis, are able to resolve the critical pair in the proof above. Consequently, in practical rewriting systems the weak η-rules for left adjunctions should be restricted or ignored (see Section 6.2.3 for a discussion about the coproduct case). Fortunately the weak η-rules for right adjunctions, which are important because they model substitution into factorizers, are by themselves confluent.

Lemma 7.1.12. *The reduction system consisting of the rules of $\mathcal{R}_2$ and the weak η-rules for right adjunctions (160) is confluent.*

Proof. The applicability of concrete adjunction instances of weak η-rules for right adjunctions depends on the type of the codomain. Therefore, for any term, only one of the instances may be used at the root node. Because of that, they do not form critical pairs among themselves. Any critical pair involving only one of the weak η-rules is solved analogously as the critical pairs for the weak η-rule for products (14), which is already present in $\mathcal{R}_L \subseteq \mathcal{R}_2$. □

By symmetry, also a variant of $\mathcal{R}_2$ with right associative vertical composition together with weak η-rules for left adjunctions forms a confluent reduction system. It is also easy to recover confluence in original $\mathcal{R}_2$ together with weak η-rules for left adjunctions, as long as there are no weak η-rules for right adjunctions.

Action on morphisms

The only other interesting reduction rules we find useful in adjunctions are the ones concerning action of the adjoint functor on morphisms. The rules are obtained by directing the following equations from left to right.

$$t * \texttt{F\_ri}(g) \;=\; \texttt{T\_rfact}(g, f \texttt{ . } \texttt{F\_ri}(g), (f * \varepsilon_{\mathrm{r}}(g)) \texttt{ . } t) \qquad (162)$$
$$t * \texttt{F\_le}(g) \;=\; \texttt{T\_lfact}(g, h \texttt{ . } \texttt{F\_le}(g), t \texttt{ . } (h * \eta_{\mathrm{l}}(g))) \qquad (163)$$

Theorem 7.1.13. *Equations (162) and (163) are consequences of Φ_F.*

Proof. Let us begin with the left hand side of equation (162)

$$t * \texttt{F\_ri}(g)$$

and transform the term using equation (149)

$$\texttt{T\_rfact}(g, f \;\texttt{.}\; \texttt{F\_ri}(g), ((t * \texttt{F\_ri}(g)) * g) \;\texttt{.}\; h * \varepsilon_{\texttt{r}}(g))$$

Then using associativity

$$\texttt{T\_rfact}(g, f \;\texttt{.}\; \texttt{F\_ri}(g), (t * (\texttt{F\_ri}(g) \;\texttt{.}\; g)) \;\texttt{.}\; h * \varepsilon_{\texttt{r}}(g))$$

and interchange law (20)

$$\texttt{T\_rfact}(g, f \;\texttt{.}\; \texttt{F\_ri}(g), (f * \varepsilon_{\texttt{r}}(g)) \;\texttt{.}\; t)$$

we arrive at the right hand side.

Equation (163) is derived in an analogous way. □

Left hand sides of the equations are quite complicated when expressed verbatim using syntactic sugar for the specific adjunctions. For example, the left hand side of equation (162) instantiated to the labeled product adjunction translates to the following form.

$$\texttt{<}i \;\texttt{:}\; t\texttt{;}\; \ldots\texttt{>} * \texttt{F\_pp}(\texttt{<}le\texttt{>}, i \;\texttt{:}\; \texttt{F\_PR}(le, i)\texttt{;}\; \ldots)$$

Such expressions are best written using special term constructors — `T_pp` in case of the labeled product adjunction, `T_ss` in case of sum or `T_ee` in case of labeled exponent. Here are the instances of the equations with the left hand side written using the shorter syntax.

$$\begin{array}{rcl}
\texttt{T\_pp}(c, i \;\texttt{=}\; t\texttt{;}\; \ldots) & = & \texttt{T\_record}(\texttt{\{}lf\texttt{\}}, i \;\texttt{=}\; \texttt{T\_pr}(lf, i) \;\texttt{.}\; t\texttt{;}\; \ldots) \\
\texttt{T\_ss}(c, i \;\texttt{=}\; t\texttt{;}\; \ldots) & = & \texttt{T\_case}(i \;\texttt{=}\; t \;\texttt{.}\; \texttt{T\_in}(lh, i)\texttt{;}\; \ldots, \texttt{[}lh\texttt{]}) \\
\texttt{T\_ee}(lg, u) & = & \texttt{T\_curry}(\texttt{T\_appl}(lg, f) \;\texttt{.}\; u)
\end{array}$$

The first equation, directed from left to right, appears as rule (55) in Section 6.1.5, the second equation as rule (77) in Section 6.2.3 and the third is a special case of rule (241) derived in Section 8.1.3 for a richer adjunction. Notice that the `T_ee` constructor has only one transformation operand. This is because the *lg* functors representing the types of parameters are fixed in this version of the labeled exponent.

7.1.5 Conclusion

We have proved the equivalence of the two equational presentations of adjunctions we use in our thesis (Observation 7.1.1, Theorem 7.1.2, Theorem 7.1.3). The standard presentation, expressed in the language of *2-LC-A*, is based on units and co-units. The alternative presentation uses the language of *2-LC-F*, concentrating on factorizers. Both equational theories are constructed in an equational framework designed to make them as strong as possible, despite partiality of adjunction operations that hinders the use of transitivity rule of equational reasoning. The strength of the theories makes it easy to instantiate them to particular categories with fixed sets of defined adjunctions (Observation 7.1.4, Observation 7.1.5).

We have demonstrated on several examples how the factorizers together with other operations of *2-LC-F* generalize the basic mechanisms of a core programming language. We have shown how the theory of *2-LC-F* provides correctness criteria for reduction systems, as well as helps in assessing their completeness and comparing their subsystems. In particular, in the context of the theory of adjunctions we conjecture that the β-rule for exponents is weaker than the β-rules for products and coproducts (Conjecture 7.1.6, Lemma 7.1.7). The axioms of *2-LC-F* can also suggest novel reduction rules for core language constructs, as well as additional language mechanisms such as exceptions treated as dual to variables, etc.

We have offered analysis and partial solutions to the problem of rewriting general adjunctions. We propose a general reduction rule for rewriting multiplication by the adjoint functor (Theorem 7.1.13), which is related to our rules concerning the mapping operation of Section 6.3.1. We have discussed rules corresponding to η-expansion and prove that using weak η-rules exclusively for right adjoints as well as using weak η-rules exclusively for left adjoints results in a confluent reduction system (Lemma 7.1.12), while using both leads to unsolvable critical pairs (Lemma 7.1.11).

7.2 Opposite categories

The labeled exponent of Section 7.1.3 lacks full type parameterization. While the type of the result can be instantiated by composing the exponent operations with a functor from the left, the types of exponent parameters always remain unchanged. This phenomenon is visible in its simplest form when the exponent functor is composed with a functor f. Let us rewrite the composition

$$f \;.\; \mathtt{F\_ee}(lg, h)$$

first using the definition of the `F_ee` operation

$$f\ .\ (h\ .\ \texttt{F\_ri}(\mathbf{X}(lg)))$$

Then let's use the associativity of composition

$$(f\ .\ h)\ .\ \texttt{F\_ri}(\mathbf{X}(lg))$$

and finally the definition of `F_ee` again

$$\texttt{F\_ee}(lg, (f\ .\ h))$$

getting exponent with only the result type instantiated.

We will try to improve the labeled exponent datatype so that it is suitable for inclusion, as part of the core language, into a module system with parameterized modules. On the way to this goal we will try to discover why the exponent functor is causing such problems. The simple answer that exponent is contravariant will not satisfy us. In *2-LC* there is no contravariance, where is this aspect coming from? Designing the first draft of parameterized adjunctions will allow us to see that the $g \mapsto \texttt{F\_ri}(g)$ partial functor is contravariant itself and that the rest of the parameterized adjunction operations is even more convoluted. To be able to discuss contravariance, first we need to introduce opposite categories.

7.2.1 *2-LCO* — *2-LC* with opposite categories

We straightforwardly generalize the notion of duality from **Cat** to an arbitrary 2-category. *2-LCO* is a *2-LC* where for every object, 1-morphism and 2-morphism there is a generalized opposite counterpart. In **Cat**, which is an example of *2-LCO*, the opposite objects are just the opposite categories and the opposite functors are the same functions on objects and morphisms but considered as coming from opposite sources to opposite targets. Analogously, the opposite transformations are the same functions from objects to morphisms but considered with new domains and codomains. Functors and transformations of arbitrary *2-LCO* behave analogously, as will be formally stated in Observation 7.2.2.

Syntax

To the language of *2-LC* we add a term constructor for opposite categories, a term constructor for opposite functors and a term constructor for opposite transformations. The term constructor for opposite categories induces equations between `cat`-terms. This does not cause problems with type-checking of functors and transformations, since the equality of `cat`-terms remains easily decidable.

```
type cat =
  | C_PP of cat IList.t
  | C_BB
  | C_OP of cat
type funct =
  | F_ID of cat
  | F_COMP of funct * funct
  | F_PR of cat IList.t * IdIndex.t
  | F_RECORD of cat * funct IList.t
  | F_OP of funct
type trans =
  | T_ID of cat
  | T_COMP of trans * trans
  | T_PR of cat IList.t * IdIndex.t
  | T_RECORD of cat * trans IList.t
  | T_FT of funct * trans
  | T_TF of trans * funct
  | T_id of funct
  | T_comp of trans * trans
  | T_OP of trans
```

Semantics

We will sometimes write c^{op}, f^{op} and t^{op} for `C_OP`(c), `F_OP`(f) and `T_OP`(t), respectively. Below is the semantics of the new terms in *2-LCO*. All the added operations are total.

- `C_OP`(c)
  is the generalized (with respect to **Cat**) opposite category to c,
- `F_OP`$(f) : c^{op} \to e^{op}$
  is the generalized opposite functor to $f : c \to e$,
- `T_OP`$(t) : h^{op} \to f^{op}$
  is the generalized opposite transformation to $t : f \to h$ (notice the transposition of domain and codomain).

Equations

The only two axioms we introduce for `cat`-terms are the following.

$$\begin{aligned} \texttt{<}i_1 \texttt{ - } c_1 \texttt{; } \ldots \texttt{; } i_n \texttt{ - } c_n\texttt{>}^{op} &= \texttt{<}i_1 \texttt{ - } c_1^{op} \texttt{; } \ldots \texttt{; } i_n \texttt{ - } c_n^{op}\texttt{>} \\ \texttt{C\_OP}(c)^{op} &= c \end{aligned}$$

When rewriting the generalized duality operations for functors and transformation in a *2-LCO* extended with additional categorical constructs, one has to exchange their characteristic operations to their duals. For example when a category has adjunctions, one has to exchange left units for right co-units (with opposite source and target categories) and when a category has inductive and coinductive types one has to exchange initial constructors for terminal destructors, etc.

However, in case of a pure *2-LCO*, eliminating `F_OP` from `funct`-terms amounts to exchanging every `C_BB` and `C_BB`^{op} nested inside an odd number of `F_OP` constructors. This suggests that the semantics of the opposite functor operation can be captured by the following `funct`-equations, where the last one follows from the previous ones, within the class of reachable algebras.

$$\begin{aligned}
\texttt{F\_ID}(c)^{op} &= \texttt{F\_ID}(c^{op}) && (164)\\
\texttt{F\_COMP}(f,g)^{op} &= \texttt{F\_COMP}(f^{op},g^{op}) && (165)\\
\texttt{F\_PR}(i\ \texttt{-}\ c\texttt{;}\ \dots, i)^{op} &= \texttt{F\_PR}(i\ \texttt{-}\ c^{op}\texttt{;}\ \dots, i) && (166)\\
\texttt{F\_RECORD}(c, i\ \texttt{:}\ f\texttt{;}\ \dots)^{op} &= \texttt{F\_RECORD}(c^{op}, i\ \texttt{:}\ f^{op}\texttt{;}\ \dots) && (167)\\
\texttt{F\_OP}(f)^{op} &= f && (168)
\end{aligned}$$

It is easy to check that eliminating `T_OP` from ground `trans`-terms amounts to exchanging every `C_BB` and `C_BB`^{op} nested inside an odd number of `T_OP` and `F_OP`, additionally swapping the vertical composition operands occurring under an odd number of `T_OP` constructors. Thus opposite transformations in *2-LCO* are described with the following `trans`-equations.

$$\begin{aligned}
\texttt{T\_ID}(c)^{op} &= \texttt{T\_ID}(c^{op}) && (169)\\
\texttt{T\_COMP}(t_1,t_2)^{op} &= \texttt{T\_COMP}(t_1^{op},t_2^{op}) && (170)\\
\texttt{T\_PR}(i\ \texttt{-}\ c\texttt{;}\ \dots, i)^{op} &= \texttt{T\_PR}(i\ \texttt{-}\ c^{op}\texttt{;}\ \dots, i) && (171)\\
\texttt{T\_RECORD}(c, i\ \texttt{=}\ t\texttt{;}\ \dots)^{op} &= \texttt{T\_RECORD}(c^{op}, i\ \texttt{=}\ t^{op}\texttt{;}\ \dots) && (172)\\
\texttt{T\_FT}(f_1,t_2)^{op} &= \texttt{T\_FT}(f_1^{op},t_2^{op}) && (173)\\
\texttt{T\_TF}(t_1,f_2)^{op} &= \texttt{T\_TF}(t_1^{op},f_2^{op}) && (174)\\
\texttt{T\_id}(g)^{op} &= \texttt{T\_id}(g^{op}) && (175)\\
\texttt{T\_comp}(t,u)^{op} &= \texttt{T\_comp}(u^{op},t^{op}) && (176)\\
\texttt{T\_OP}(t)^{op} &= t && (177)
\end{aligned}$$

Within the class of reachable algebras, the last equation follows from the previous ones, too. Notice that in equation (176) the order of arguments of composition is reversed, while in equations (165) and (170) it is not.

Observation 7.2.1. *In every 2-LCO, each ground* `funct`*-term and ground* `trans`*-term has a counterpart term with the same semantics but written without the use of* `F_OP` *and* `T_OP` *term constructors.*

Let Φ^O be the theory generated by the axioms of the theory Φ_2 of *2-LC* and the `cat`, `funct` and `trans`-equations above. The following observation may be regarded as the definition of the generalized duality. It is easy to see that it holds for **Cat**.

Observation 7.2.2. *Φ^O is the theory of 2-LCO.*

7.2.2 Why exponent is contravariant

In this section we will informally use the notion of exponent category (the exponential object of **Cat**) and its related operations. In the next section we will express our conclusions using a language with only product categories. To simplify our mental experiment, we will gloss over the possible partiality of the discussed operations.

We mentioned earlier that the (partial) functor $g \mapsto$ `F_ri`(g) is contravariant. How to express this more strictly? One might conduct a mental experiment and incorporate this functor into the language by temporarily allowing an exponent category `EE`(c, e) as a new object of sort `cat`: the category of (partial) functors from c to e.

Theorem 7.2.3. *The function* `F_ri`(c, e)*, which takes a functor $f : c \to e$ to its right adjoint (written* `F_ri`(f)*), can be seen as a (partial) functor with type*

$$\texttt{F\_ri}(c, e) : \texttt{EE}(c, e) \to \texttt{EE}(e, c)^{op}$$

with its action on a transformation $t : f \to g$ equal to

$$(\texttt{F\_ri}(g) * \texttt{T\_reta}(f)) \;.\; (\texttt{F\_ri}(g) * t * \texttt{F\_ri}(f)) \;.\; (\texttt{T\_repsilon}(g) * \texttt{F\_ri}(f))$$

Proof. Diagram chasing shows that the application of functor `F_ri`(c, e) to a composition of transformations results in a composition of applications to the individual transformations (taking into account the reversed direction of morphisms in the target category). We also verify easily that `F_ri`(c, e) acts on identity giving an identity, as required. □

Let us suppose we have a functor

$$g : \texttt{<}\iota \texttt{ - } c\texttt{; } \kappa \texttt{ - } d\texttt{>} \to e$$

where the component at label ι is the main operand and κ is for auxiliary parameters. Let us observe how a procedure of obtaining parameterized right adjoint to g would look like in this setting. In other words, we want to adjoin with respect to the ι component and treat the component κ as a back-door for parameterization by types. To do so we would make use of an isomorphism (called curryfication) to transform this functor to a form with exponent target

$$g' = \texttt{F\_CURRY}(\kappa, g) : d \to \texttt{EE}(c, e)$$

and then compose it with the right adjoint functor

$$g' \ . \ \texttt{F\_ri}(c, e) : d \to \texttt{EE}(e, c)^{op}$$

Since the right adjoint functor is contravariant we have got a complicated codomain, so we switch to the opposite category

$$\texttt{F\_OP}(g' \ . \ \texttt{F\_ri}(c, e)) : d^{op} \to \texttt{EE}(e, c)$$

In this form it is both clear that d serves for parameterization, as desired, and that the parameterization would be contravariant. Finally, we may apply the inverse operation to curryfication, written here simply as `F_UNCURRY`

$$\texttt{F\_UNCURRY(F\_OP}(g' \ . \ \texttt{F\_ri}(c, e))) : \texttt{<}\iota \ \texttt{-} \ e\texttt{;} \ \kappa \ \texttt{-} \ d^{op}\texttt{>} \to c$$

and we get a parameterized adjoint functor without the exponent category in source or target.

Example. As an instance of a parameterized right adjoint let us consider the labeled exponent functor. Let $lb = i_1$ `- *; ... ;` i_n `- *` and let υ be a distinguished label not equal to any of $i_1, \ldots, i_n$. Let $le = \iota$ `- *;` κ `- <`lb`>`. Let

$$\mathbf{Z}(lb) \quad = \quad \texttt{\{}\upsilon \ \texttt{:} \ \texttt{F\_PR}(le, \iota)\texttt{;} \ i_1 \ \texttt{:} \ \texttt{F\_PR}(le, \kappa) \ . \ \texttt{F\_PR}(lb, i_1)\texttt{;} \ \ldots\texttt{\}}$$

We see that

$$\mathbf{Z}(lb) \quad : \quad \texttt{<}\iota \ \texttt{- *;} \ \kappa \ \texttt{- <}lb\texttt{>>} \to \texttt{*}$$

The result of applying the procedure for obtaining parameterized right adjoint functors, described above, would be a raw labeled exponent functor

$$\mathbf{E}(lb) \quad : \quad \texttt{<}\iota \ \texttt{- *;} \ \kappa \ \texttt{- <}lb\texttt{>}^{op}\texttt{>} \to \texttt{*}$$

that is contravariantly parameterized. We might now define a more user friendly syntax for the labeled exponent operation

$$\texttt{F\_ee}(lg, h) \quad = \quad \texttt{<}\iota \ \texttt{:} \ h\texttt{;} \ \kappa \ \texttt{: <}lg\texttt{>>} \ . \ \mathbf{E}(lb)$$

where $g_i : a \to \texttt{C\_BB}^{op}$, $h : a \to \texttt{C\_BB}$ and a is a category.

7.2.3 Attempt to accommodate parameterized adjoints

The procedure for obtaining parameterized right adjoints, outlined above, neither begins nor ends with exponent categories. This enables us to capture the type of the arguments and the type of the results of the procedure without extending our formalism. This leads to the following definition of a partial operation of parameterized right adjoints as an add-on to *2-LCO*.

- $\texttt{F\_ori}(g) : \texttt{<}\iota\ \texttt{-}\ e\texttt{;}\ \kappa\ \texttt{-}\ d^{op}\texttt{>} \to c$
  is the parameterized right adjoint to
  $g : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to e$.

Now we may capture formally the equality of the previous section:

$$\mathbf{E}(lb) \quad = \quad \texttt{F\_ori}(\mathbf{Z}(lb))$$

and state other properties of the operation, for instance:

$$\texttt{F\_PR}(\iota\ \texttt{-}\ \texttt{C\_BB;}\ \kappa\ \texttt{-}\ \texttt{C\_BB}^{op},\ \iota) \quad = \quad \texttt{F\_ori}(\texttt{F\_PR}(\iota\ \texttt{-}\ \texttt{C\_BB;}\ \kappa\ \texttt{-}\ \texttt{C\_BB},\ \iota))$$

We speak about `F_ori` combinator as being "parameterized on the component κ", because when we compose a record of functors with the combinator then the operand of the combinator is influenced by the component κ of the record. This allows for adjoining to "generic" functors and instantiating them later on, when their specific field of use is finally revealed. We will express this equationally. Let $lb = \iota\ \texttt{-}\ e\texttt{;}\ \kappa\ \texttt{-}\ b$, $lb' = \iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ b^{op}$ and let $f : b \to d^{op}$.

$$\begin{aligned}&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ \texttt{F\_ori}(g)\\ &= \texttt{F\_ori}(\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ \texttt{F\_OP}(f)\texttt{>}\ \texttt{.}\ g)\end{aligned} \tag{178}$$

As we see, now there is a way to "get inside" the adjoint. This equation follows from the properties of operations used in the construction of right adjoint functors outlined in the previous section.

Next we will try to advance a little further and define a partial operation of parameterized right co-unit, but we will fail badly. Originally the codomain of a right co-unit has been an identity, so now we use the projection ignoring additional parameterization component κ. The domain has been a composition and now we will compose, too, but additionally we must take care to pass the component κ to the functor g.

- $\texttt{T\_orepsilon}(g) : \texttt{<}\iota\ \texttt{:}\ \texttt{F\_ori}(g)\texttt{;}\ \kappa\ \texttt{:}\ \ldots\texttt{>}\ \texttt{.}\ g \to \texttt{F\_PR}(\iota\ \texttt{-}\ e\texttt{;}\ \kappa\ \texttt{-}\ d^{op},\ \iota)$
  if not for the unrealizable functor denoted by "...", this would be a parameterized co-unit of the adjunction between $g : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to e$ and $\texttt{F\_ori}(g)$.

Unfortunately we do not know what to put in place of the dots in the domain of co-unit. This functor would have to be from <ι - e; κ - d^{op}> to d. But if e is arbitrary and d is not trivial, then there may be no such morphism in *2-LCO*. This shows that there are worse problems to overcome than a contravariance in order to accommodate full parameterized adjunctions into our framework.

Example. In the presence of so strangely typed adjunction operations, one might wonder, how the labeled product adjunction that seems fully parameterized did fit into our framework, without even resorting to opposite categories. The answer is that, in the product adjunction, the functor to which we adjoin

$$\Delta \quad = \quad \texttt{<}i_1 \texttt{ : F\_PR(}\iota \texttt{ - *; } \kappa \texttt{ - <>, } \iota\texttt{); ...>}$$

is only parameterized trivially. Here d from the definition of right adjoint is equal to <> and so d is equal to its own opposite category. For this reason the contravariance of the right adjoint may be disregarded:

$$\texttt{F\_ori}(\Delta) \quad : \quad \texttt{<}\iota \texttt{ - <}i_1 \texttt{ - *; ...>; } \kappa \texttt{ - <>>} \rightarrow \texttt{*}$$

and the product functor is defined similarly as in Section 7.1.3:

$$\texttt{F\_pp}(a, lh) \quad = \quad \texttt{<}\iota \texttt{ : F\_RECORD}(a, lh)\texttt{; } \kappa \texttt{ : <>> . F\_ori}(\Delta)$$

Since the category of parameters d is here equal to its own opposite category, the functor to put in place of the dots in the definition of the co-unit is just the identity functor. Hence the definition of projection is easy:

$$\texttt{T\_pr}(lh, j) \quad = \quad \texttt{<}\iota \texttt{ : F\_RECORD}(a, lh)\texttt{; } \kappa \texttt{ : <>>} * \texttt{T\_orepsilon}(\Delta) * \texttt{T\_PR}(le, j)$$

But the instantiation of the projection is not achieved by changing the inside of the adjoint functor, and hence is not dependent on the parameterization of the right adjoint (void in this case). The instantiation is performed by multiplying the projection with a functor from the outside, where the results of the multiplication remain outside of the co-unit, similarly as in Section 7.1.3.

7.2.4 *2-LCX* — *2-LCO* with self-dualities

Due to the complicated domains and codomains, parameterized adjunctions cannot be added to ordinary *2-LCO*. We want *2-LCX* to be a category similar to *2-LCO* but with the ability to present contravariant functors as covariant ones and vice versa. This presentation will be achieved by means of composing with the additional "self-duality" morphism $\texttt{F\_CONTRA}(c) : c \rightarrow c^{op}$. The morphism itself may be thought of as a contravariant presentation of an identity, hence the name. A simple example of the behavior of the morphism is in the proof of Lemma 7.2.22.

The consequence of introducing `F_CONTRA` morphisms is that there exist no *2-LCX* 2-categories obtained by extending the **Cat** structure with additional operations. Moreover, even if we extend the carriers of **Cat** with additional morphisms, we cannot obtain a *2-LCX* without completely changing how the composition works. Therefore, we will define *2-LCX* not by extending *2-LCO*, but by translating the language of *2-LCX* to the language of *2-LCO*. In the next sections we will extend the translation to accommodate adjunctions.

Syntax

The syntax of *2-LCX* is an extension of the syntax of *2-LCO*. The new `funct`-term is `F_CONTRA` and the new `trans`-term is `T_CONTRA`. The `cat`-terms are unchanged.

```
type funct =
  ...
  | F_CONTRA of cat

type trans =
  ...
  | T_CONTRA of cat
```

Semantics

The translation from the language of *2-LCX* to the language of *2-LCO* will be denoted by double brackets, as in the first two cases of the translation of `funct`-terms:

$$\begin{aligned} [\![\texttt{F\_ID}(c)]\!] &= \texttt{F\_ID}([\![c]\!]) \\ [\![\texttt{F\_COMP}(f,g)]\!] &= \texttt{F\_COMP}([\![f]\!],[\![g]\!]) \end{aligned}$$

Strictly speaking the translation is determined by an interpretation of *2-LCX* operations in terms of *2-LCO* operations. The interpretation is written with the convention that variables from the left hand side are in double brackets on the right hand side, as is customary in arbitrary inductively defined translations. However, notice that the left hand side is here always a single *2-LCX* term constructor applied to variables, so the equalities determine an interpretation of *2-LCX* operations. This ensures that any reduct of *2-LCO* with respect to the translation has well defined operations corresponding to the *2-LCX* term constructors. To have well defined domain and codomain operations on *2-LCX* transformations we will additionally have to restrict the carriers of the reducts, as explained below.

We set a translation of a *2-LCX* variable to be the same variable but seen as *2-LCO* variable — this is sound since the carriers of a *2-LCX* are subsets of the carriers of the corresponding *2-LCO*, so every *2-LCX* valuation is a valid *2-LCO* valuation. The obvious clauses $[\![x]\!] = x$ for variables x of each syntactic domain are omitted below. Moreover we observe that the translation induces a function (inclusion) from the carrier of a *2-LCX* to the carrier of the *2-LCO* it was built from and we denote the function the same as the translation, that is, by $[\![\_]\!]$.

The idea behind the translation is that any ground terms c, f and t, not containing `F_CONTRA` nor `T_CONTRA`, translate as follows:

$$\begin{array}{rcl} [\![c]\!] & = & \texttt{<}\nu\ \texttt{-}\ c\texttt{;}\ \pi\ \texttt{-}\ c^{op}\texttt{>} \\ [\![f]\!] & = & \texttt{<}\nu\ \texttt{=}\ \texttt{F\_PR}(ld,\nu)\ \texttt{.}\ f\texttt{;}\ \pi\ \texttt{=}\ \texttt{F\_PR}(ld,\pi)\ \texttt{.}\ f^{op}\texttt{>} \\ [\![t]\!] & = & \texttt{<}\nu\ \texttt{=}\ \texttt{T\_PR}(ld,\nu) * t\texttt{;}\ \pi\ \texttt{=}\ \texttt{T\_PR}(ld,\pi) * t^{op}\texttt{>} \end{array}$$

where ν and π are distinguished labels and $ld = \nu$ `-` d`;` π `-` d^{op}. The first equation is for `cat`-terms, the second equation for `funct`-terms and the third for `trans`-terms. Because f and t do not contain `F_CONTRA` nor `T_CONTRA`, they can appear here as both *2-LCX* and *2-LCO* terms. Translations of terms with `F_CONTRA` or `T_CONTRA` differ in that the projections can be transposed in various ways.

If a category c is of the form:

$$\texttt{<}\nu\ \texttt{-}\ d\texttt{;}\ \pi\ \texttt{-}\ d^{op}\texttt{>}$$

which is the only form allowed for the carrier of *2-LCX*, we will use the following notation in the definition of the translation:

$$\begin{array}{rcl} [\![c]\!]^{\nu} & = & d \\ [\![c]\!]^{\pi} & = & d^{op} \end{array}$$

The translation of `cat`-terms is defined as follows. The language of *2-LCX* `cat`-terms is the same as the language of `cat`-terms in *2-LCO*, yet the translation is not the identity.

$$\begin{array}{rcl} [\![\texttt{C\_PP}(i\ \texttt{-}\ c\texttt{;}\ \ldots)]\!] & = & \texttt{<}\nu\ \texttt{-}\ \texttt{C\_PP}(i\ \texttt{-}\ [\![c]\!]^{\nu}\texttt{;}\ \ldots)\texttt{;}\ \pi\ \texttt{-}\ \texttt{C\_PP}(i\ \texttt{-}\ [\![c]\!]^{\pi}\texttt{;}\ \ldots)\texttt{>} \\ [\![\texttt{C\_BB}]\!] & = & \texttt{<}\nu\ \texttt{-}\ \texttt{C\_BB;}\ \pi\ \texttt{-}\ \texttt{C\_BB}^{op}\texttt{>} \\ [\![\texttt{C\_BB}^{op}]\!] & = & \texttt{<}\nu\ \texttt{-}\ \texttt{C\_BB}^{op}\texttt{;}\ \pi\ \texttt{-}\ \texttt{C\_BB>} \end{array}$$

In the translation of the language of `funct`-terms we use the notation:

$$\begin{array}{rcl} [\![f]\!]^{\nu} & = & [\![f]\!]\ \texttt{.}\ \texttt{F\_PR}(lc,\nu) \\ [\![f]\!]^{\pi} & = & [\![f]\!]\ \texttt{.}\ \texttt{F\_PR}(lc,\pi) \end{array}$$

The language of `funct`-terms translates as follows.

$$
\begin{aligned}
[\![\texttt{F\_ID}(c)]\!] &= \texttt{F\_ID}([\![c]\!]) \\
[\![\texttt{F\_COMP}(f,g)]\!] &= \texttt{F\_COMP}([\![f]\!],[\![g]\!]) \\
[\![\texttt{F\_PR}(i\ \texttt{-}\ c\texttt{;}\ \ldots, i)]\!] &= \texttt{<}\nu\ \texttt{:}\ \texttt{F\_PR}(ld,\nu)\ \texttt{.}\ \texttt{F\_PR}(i\ \texttt{-}\ [\![c]\!]^{\nu}\texttt{;}\ \ldots, i)\texttt{;} \\
&\quad\ \pi\ \texttt{:}\ \texttt{F\_PR}(ld,\pi)\ \texttt{.}\ \texttt{F\_PR}(i\ \texttt{-}\ [\![c]\!]^{\pi}\texttt{;}\ \ldots, i)\texttt{>} \\
[\![\texttt{F\_RECORD}(c, i\ \texttt{:}\ f\texttt{;}\ \ldots)]\!] &= \texttt{<}\nu\ \texttt{:}\ \texttt{F\_RECORD}([\![c]\!], i\ \texttt{:}\ [\![f]\!]^{\nu}\texttt{;}\ \ldots)\texttt{;} \\
&\quad\ \pi\ \texttt{:}\ \texttt{F\_RECORD}([\![c]\!], i\ \texttt{:}\ [\![f]\!]^{\pi}\texttt{;}\ \ldots)\texttt{>} \\
[\![\texttt{F\_OP}(f)]\!] &= \texttt{F\_OP}([\![f]\!]) \\
[\![\texttt{F\_CONTRA}(c)]\!] &= \texttt{<}\nu\ \texttt{:}\ \texttt{F\_PR}(lc,\pi)\texttt{;}\ \pi\ \texttt{:}\ \texttt{F\_PR}(lc,\nu)\texttt{>}
\end{aligned}
$$

In the third clause $ld = \nu\ \texttt{-}\ \texttt{<}\ i\ \texttt{-}\ [\![c]\!]^{\nu}\texttt{;}\ \ldots\texttt{>;}\ \pi\ \texttt{-}\ \texttt{<}\ i\ \texttt{-}\ [\![c]\!]^{\pi}\texttt{;}\ \ldots\texttt{>}$ and in the last clause $lc = \nu\ \texttt{-}\ [\![c]\!]^{\nu}\texttt{;}\ \pi\ \texttt{-}\ [\![c]\!]^{\pi}$, hence $\texttt{<}lc\texttt{>} = [\![c]\!]$. These abbreviations for indexed lists of categories retain the same meaning in the translation of `trans`-terms.

In the translation of the language of `trans`-terms we use the notation:

$$
\begin{aligned}
[\![t]\!]^{\nu} &= [\![t]\!] * \texttt{F\_PR}(lc,\nu) \\
[\![t]\!]^{\pi} &= [\![t]\!] * \texttt{F\_PR}(lc,\pi)
\end{aligned}
$$

The language of `trans`-terms translates as follows.

$$
\begin{aligned}
[\![\texttt{T\_ID}(c)]\!] &= \texttt{T\_ID}([\![c]\!]) \\
[\![\texttt{T\_COMP}(t_1,t_2)]\!] &= \texttt{T\_COMP}([\![t_1]\!],[\![t_2]\!]) \\
[\![\texttt{T\_PR}(i\ \texttt{-}\ c\texttt{;}\ \ldots, i)]\!] &= \texttt{<}\nu\ \texttt{=}\ \texttt{T\_PR}(ld,\nu) * \texttt{T\_PR}(i\ \texttt{-}\ [\![c]\!]^{\nu}\texttt{;}\ \ldots, i)\texttt{;} \\
&\quad\ \pi\ \texttt{=}\ \texttt{T\_PR}(ld,\pi) * \texttt{T\_PR}(i\ \texttt{-}\ [\![c]\!]^{\pi}\texttt{;}\ \ldots, i)\texttt{>} \\
[\![\texttt{T\_RECORD}(c, i\ \texttt{=}\ t\texttt{;}\ \ldots)]\!] &= \texttt{<}\nu\ \texttt{=}\ \texttt{T\_RECORD}([\![c]\!], i\ \texttt{=}\ [\![t]\!]^{\nu}\texttt{;}\ \ldots)\texttt{;} \\
&\quad\ \pi\ \texttt{=}\ \texttt{T\_RECORD}([\![c]\!], i\ \texttt{=}\ [\![t]\!]^{\pi}\texttt{;}\ \ldots)\texttt{>} \\
[\![\texttt{T\_FT}(f_1,t_2)]\!] &= \texttt{T\_FT}([\![f_1]\!],[\![t_2]\!]) \\
[\![\texttt{T\_TF}(t_1,f_2)]\!] &= \texttt{T\_TF}([\![t_1]\!],[\![f_2]\!]) \\
[\![\texttt{T\_id}(g)]\!] &= \texttt{T\_id}([\![g]\!]) \\
[\![\texttt{T\_comp}(t,u)]\!] &= \texttt{<}\nu\ \texttt{=}\ [\![t]\!]^{\nu}\ \texttt{.}\ [\![u]\!]^{\nu}\texttt{;}\ \pi\ \texttt{=}\ [\![u]\!]^{\pi}\ \texttt{.}\ [\![t]\!]^{\pi}\texttt{>} \\
[\![\texttt{T\_OP}(t)]\!] &= \texttt{T\_OP}([\![t]\!]) \\
[\![\texttt{T\_CONTRA}(c)]\!] &= \texttt{<}\nu\ \texttt{=}\ \texttt{T\_PR}(lc,\pi)\texttt{;}\ \pi\ \texttt{=}\ \texttt{T\_PR}(lc,\nu)\texttt{>}
\end{aligned}
$$

In the clause concerning `T_comp` notice the reversed order of vertical composition in the π component of the record.

The translation gives a precise semantics to the two new operations of *2-LCX*, at the cost of redefining all the others. The construction also assures us that there exist nontrivial *2-LCX* and provide a systematic way to construct the equational theory. A *2-LCX* algebra is obtained from a *2-LCO* algebra by the usual notion of an algebraic reduct with respect to the translation [134] and then by restricting the carrier of *2-LCX* as in Definition 7.2.4 below.

Definition 7.2.4. *The carrier of the 2-LCX constructed from a given 2-LCO consists of all and only elements of the 2-LCO satisfying the following conditions.*

1. *for a category of 2-LCO to belong to 2-LCX it must be of the form*

$$\texttt{<}\nu\ \texttt{-}\ c\texttt{;}\ \pi\ \texttt{-}\ c^{op}\texttt{>}$$

   where c is a category of the 2-LCO

2. *for a functor of 2-LCO to belong to 2-LCX it must be of the form*

$$\texttt{<}\nu\ \texttt{:}\ f\texttt{;}\ \pi\ \texttt{:}\ [\![\texttt{F\_CONTRA}(c)]\!]\ \texttt{.}\ \texttt{F\_OP}(f)\texttt{>}$$

3. *for a transformation of 2-LCO to belong to 2-LCX it must be of the form*

$$\texttt{<}\nu\ \texttt{=}\ u\texttt{;}\ \pi\ \texttt{=}\ [\![\texttt{T\_CONTRA}(c)]\!] * \texttt{T\_OP}(u)\texttt{>}$$

Observation 7.2.5. *The carrier of functors of 2-LCX is exactly the set of 2-LCO functors f that satisfy the following 2-LCX equality:*

$$f\ \texttt{.}\ \texttt{F\_CONTRA}(a) \quad = \quad \texttt{F\_CONTRA}(b)\ \texttt{.}\ \texttt{F\_OP}(f)$$

In other words, the `funct` *carrier of 2-LCX is exactly the set of symmetric functors, as defined in [48], between the categories of 2-LCO.*

Lemma 7.2.6. *The following equation holds in 2-LCX.*

$$t * \texttt{T\_CONTRA}(a) \quad = \quad \texttt{T\_CONTRA}(b) * \texttt{T\_OP}(t)$$

Proof. Consider any *2-LCO* and *2-LCX* obtained from it as above. Consider any value of t in the *2-LCX*; by Definition 7.2.4 we know that it is of the form (described in the language of *2-LCO*)

$$\texttt{<}\nu\ \texttt{=}\ u\texttt{;}\ \pi\ \texttt{=}\ [\![\texttt{F\_CONTRA}(b)]\!] * \texttt{T\_OP}(u)\texttt{>}$$

for some u. Then the value of the right hand side is obtained by translating to the *2-LCO*, which yields:

$$[\![\texttt{T\_CONTRA}(b)]\!] * \texttt{T\_OP}(\texttt{<}\nu\ \texttt{=}\ u\texttt{;}\ \pi\ \texttt{=}\ [\![\texttt{F\_CONTRA}(b)]\!] * \texttt{T\_OP}(u)\texttt{>})$$

By equations about `T_OP` this equals

$$[\![\texttt{T\_CONTRA}(b)]\!] * \texttt{<}\nu\ \texttt{=}\ \texttt{T\_OP}(u)\texttt{;}\ \pi\ \texttt{=}\ [\![\texttt{F\_CONTRA}(b^{op})]\!] * u\texttt{>}$$

and moving T_CONTRA into the record we obtain

$$<\nu = [\![\texttt{T\_CONTRA}(b)]\!] * \texttt{T\_OP}(u);\ \pi = [\![\texttt{T\_CONTRA}(b)]\!] * [\![\texttt{F\_CONTRA}(b^{op})]\!] * u>$$

This can be simplified to

$$<\nu = [\![\texttt{T\_CONTRA}(b)]\!] * \texttt{T\_OP}(u);\ \pi = u>$$

and, using the definition of $[\![\texttt{T\_CONTRA}(a)]\!]$ and the equations about records, transformed as follows

$$<\nu = u;\ \pi = [\![\texttt{T\_CONTRA}(b)]\!] * \texttt{T\_OP}(u)> * [\![\texttt{T\_CONTRA}(a)]\!]$$

which is just the value of the left hand side. □

Lemma 7.2.7. *Let $\mathcal{M}$ be a 2-LCO. Let $\mathcal{M}'$ be the 2-LCX constructed from $\mathcal{M}$ according to Definition 7.2.4. Let t be a 2-LCX term. Then the 2-LCO semantics of term $[\![t]\!]$ belongs to the carrier of $\mathcal{M}'$ (under any type-correct valuation of variables of t). This implies that the operations of 2-LCX are well defined.*

Proof. The prove is by structural induction on t. We will prove only the hardest case: that of horizontal composition.

Consider any type-correct valuation of variables of t. By definition of the translation we have

$$[\![\texttt{T\_COMP}(t_1, t_2)]\!] \quad = \quad \texttt{T\_COMP}([\![t_1]\!], [\![t_2]\!])$$

where t_1 and t_2 are *2-LCX* terms with the semantics of their translation belonging to the carrier of $\mathcal{M}'$. By Definition 7.2.4 we know that the value of $[\![t_2]\!]$ in $\mathcal{M}$ is equal to

$$<\nu = u_2;\ \pi = [\![\texttt{T\_CONTRA}(c)]\!] * \texttt{T\_OP}(u_2)>$$

for some transformation u_2.

Consequently, the value of the horizontal composition of $[\![t_1]\!]$ and $[\![t_2]\!]$ is

$$[\![t_1]\!] * <\nu = u_2;\ \pi = [\![\texttt{T\_CONTRA}(c)]\!] * \texttt{T\_OP}(u_2)>$$

which is equal to

$$\begin{aligned}<\nu &= [\![t_1]\!] * u_2;\\ \pi &= [\![t_1]\!] * [\![\texttt{T\_CONTRA}(c)]\!] * \texttt{T\_OP}(u_2)>\end{aligned}$$

which can be expressed as

$$\begin{aligned}<\nu &= [\![t_1]\!] * u_2;\\ \pi &= [\![t_1 * \texttt{T\_CONTRA}(c)]\!] * \texttt{T\_OP}(u_2)>\end{aligned}$$

By Lemma 7.2.6 we rewrite this to

$$\begin{aligned}&\texttt{<}\nu \;\texttt{=}\; [\![t_1]\!] * u_2\,\texttt{;}\\ &\;\pi \;\texttt{=}\; [\![\texttt{T\_CONTRA}(b) * \texttt{T\_OP}(t_1)]\!] * \texttt{T\_OP}(u_2)\texttt{>}\end{aligned}$$

we use the definition of the translation

$$\begin{aligned}&\texttt{<}\nu \;\texttt{=}\; [\![t_1]\!] * u_2\,\texttt{;}\\ &\;\pi \;\texttt{=}\; [\![\texttt{T\_CONTRA}(b)]\!] * \texttt{T\_OP}([\![t_1]\!]) * \texttt{T\_OP}(u_2)\texttt{>}\end{aligned}$$

and rewrite

$$\begin{aligned}&\texttt{<}\nu \;\texttt{=}\; [\![t_1]\!] * u_2\,\texttt{;}\\ &\;\pi \;\texttt{=}\; [\![\texttt{T\_CONTRA}(b)]\!] * \texttt{T\_OP}([\![t_1]\!] * u_2)\texttt{>}\end{aligned}$$

obtaining a form that clearly satisfies the condition of Definition 7.2.4. □

Remark. The translation alone allows us to define a *2-LCX* algebra, but not a *2-LCX* category. We restrict the carriers of the algebra so that elements of *2-LCX* have simple *2-LCO* domains and codomains, and so Definition 7.2.8 below is valid. Then we can define the domain and codomain operations in *2-LCX*, observe that carriers are closed under these operations and conclude that *2-LCX* is a category (since the other properties of a category are obvious for a *2-LCX*). The restriction is also crucial for *2-LCX* to be labeled cartesian. Note that the restricted carriers can still contain elements expressible in *2-LCO*, but not in *2-LCX*. For example:

$$\texttt{<}\nu \;\texttt{:}\; \texttt{F\_ID(<}\nu \;\texttt{-}\; \texttt{C\_BB;}\; \pi \;\texttt{-}\; \texttt{C\_BB}^{op}\texttt{>);}\; \pi \;\texttt{:}\; [\![\texttt{F\_CONTRA(C\_BB)}]\!]\texttt{>}$$

is not expressible in *2-LCX* built upon **Cat** with `C_BB` equal to **Set**. Consequently, reachable *2-LCO* may give rise to an unreachable *2-LCX*.

Definition 7.2.8. *Let us fix a 2-LCO and a 2-LCX such that the 2-LCX is constructed from the 2-LCO according to Definition 7.2.4. The 2-LCX source and target category of a 2-LCX functor or transformation is defined to be the source and target in the 2-LCO, respectively. We see that if t is a 2-LCX transformation then t is typed in the 2-LCO as follows (notice f and h in the domain)*

$$\begin{aligned}[\![t]\!] \;:\; &\texttt{<}\nu \;\texttt{:}\; f\texttt{;}\; \pi \;\texttt{:}\; [\![\texttt{F\_CONTRA}(c)]\!] \;\texttt{.}\; \texttt{F\_OP}(h)\texttt{>} \to\\ &\texttt{<}\nu \;\texttt{:}\; h\texttt{;}\; \pi \;\texttt{:}\; [\![\texttt{F\_CONTRA}(c)]\!] \;\texttt{.}\; \texttt{F\_OP}(f)\texttt{>}\end{aligned}$$

where f and h are 2-LCO functors uniquely determined by t. We define the 2-LCX domain and 2-LCX codomain of such t to be the following (notice the lack of h in the domain and of f in the codomain)

$$\begin{aligned}t \;:\; &\texttt{<}\nu \;\texttt{:}\; f\texttt{;}\; \pi \;\texttt{:}\; [\![\texttt{F\_CONTRA}(c)]\!] \;\texttt{.}\; \texttt{F\_OP}(f)\texttt{>} \to\\ &\texttt{<}\nu \;\texttt{:}\; h\texttt{;}\; \pi \;\texttt{:}\; [\![\texttt{F\_CONTRA}(c)]\!] \;\texttt{.}\; \texttt{F\_OP}(h)\texttt{>}\end{aligned}$$

where f and h are the functors from the 2-LCO typing.

By convention, if not stated otherwise, when we talk about elements of the carrier of *2-LCX*, domains and codomains are understood in the *2-LCX* sense; if we want to discuss a *2-LCO* domain f and codomain h of a *2-LCX* transformation t we will write $[\![t]\!] : f \to h$, where $[\![\_]\!]$ is the carrier inclusion.

The *2-LCX* domains and codomains belong to the *2-LCX* carrier of functors, unlike the *2-LCO* domains and codomains. The composition in *2-LCX* is given by `T_COMP` and the identity by `T_ID` — our translation defines the operations even for unreachable elements of the carriers.

Observation 7.2.9. *Every 2-LCX is a category. Moreover 2-LCX is a labeled cartesian subcategory of the 2-LCO it was constructed from. In particular, there are 2-LCX that are subcategories of* **Cat**.

More precisely, given a *2-LCO* and the *2-LCX* built upon it, both the underlying category and the category of objects and 2-morphisms of *2-LCX* are subcategories of the corresponding categories of *2-LCO*. However a *2-LCX* is not, in general, a sub-2-category of the *2-LCO*, because the typing of transformations and the semantics of vertical composition are not the same. In fact *2-LCX* needn't be a 2-category itself (see Corollary 7.2.23 below), although all of the $C(c, e)$ structures are categories.

Equations

The equational theory of *2-LCX* will be called Φ^X. Because the semantics of *2-LCX* is given by the translation to *2-LCO*, the theory Φ^X is a superset of the co-image of the theory Φ^O under the translation [134]. Here we will observe some facts about Φ^X. In particular we will explain why we think Φ^X doesn't have a finite equational axiomatization. Most of the proofs are based on similar ideas. Only some of the proofs are included here.

Observation 7.2.10. *Let u be a ground 2-LCO* `trans`*-term. Then there is a 2-LCX term t textually identical to u (but built with constructors of a different language). If f and h are ground 2-LCO* `funct`*-terms denoting 2-LCO domain and codomain of u then f and h (treated as 2-LCX* `funct`*-terms) denote the 2-LCX domain and codomain of t, respectively.*

Observation 7.2.11. *Let u be a ground 2-LCO* `trans`*-term. Let t be a 2-LCX term textually identical to u. Then the transformation denoted by the translation of t is equal to the transformation denoted by the following 2-LCO term.*

$$\texttt{<}\nu\ \texttt{=}\ \texttt{F\_PR}(lc, \nu) * u\texttt{;}\ \pi\ \texttt{=}\ \texttt{F\_PR}(lc, \pi) * u^{op}\texttt{>}$$

Corollary 7.2.12. *The theory Φ^X contains all ground equations of the theory Φ^O.*

Proof. If a ground equation

$$u_1 \;=\; u_2$$

belongs to Φ^O then, by congruence, equation

$$\begin{aligned}&\texttt{<}\nu \texttt{ = F\_PR}(lc,\nu) * u_1\texttt{;}\ \pi \texttt{ = F\_PR}(lc,\pi) * u_1^{op}\texttt{>}\\ =\ &\texttt{<}\nu \texttt{ = F\_PR}(lc,\nu) * u_2\texttt{;}\ \pi \texttt{ = F\_PR}(lc,\pi) * u_2^{op}\texttt{>}\end{aligned}$$

is in Φ^O, too. But if so then, by Observation 7.2.11, equation

$$t_1 \;=\; t_2$$

belongs to Φ^X, where t_1 and t_2 are *2-LCX* terms textually identical to u_1 and u_2. □

It is easy to verify, similarly as in the proof of Lemma 7.2.6, that theory Φ^X contains the following `funct`-equations:

$$f \texttt{ . F\_CONTRA}(a) \;=\; \texttt{F\_CONTRA}(b) \texttt{ . F\_OP}(f) \tag{179}$$
$$\texttt{F\_CONTRA}(a) \texttt{ . F\_CONTRA}(a^{op}) \;=\; \texttt{F\_ID}(a) \tag{180}$$
$$\texttt{F\_OP(F\_CONTRA}(a)\texttt{)} \;=\; \texttt{F\_CONTRA}(a^{op}) \tag{181}$$

and the following `trans`-equations:

$$t * \texttt{T\_CONTRA}(a) \;=\; \texttt{T\_CONTRA}(b) * \texttt{T\_OP}(t) \tag{182}$$
$$\texttt{T\_CONTRA}(a) \;=\; \texttt{T\_id(F\_CONTRA}(a)\texttt{)} \tag{183}$$

which entail also these:

$$\texttt{T\_CONTRA}(a) * \texttt{T\_CONTRA}(a^{op}) \;=\; \texttt{T\_ID}(a) \tag{184}$$
$$\texttt{T\_OP(T\_CONTRA}(a)\texttt{)} \;=\; \texttt{T\_CONTRA}(a^{op}) \tag{185}$$

Definition 7.2.13. *Let d be a category and let $ld = \nu$ - d; π - d^{op}. A 2-LCO functor $g : \texttt{<}\,ld\texttt{>} \to e$ is called X-covariant if there exist functors g' and g'' such that the following 2-LCO equality holds*

$$g \;=\; \texttt{<}\nu \texttt{ : F\_PR}(ld,\nu) \texttt{ . } g'\texttt{;}\ \pi \texttt{ : F\_PR}(ld,\pi) \texttt{ . } g''\texttt{>}$$

and analogously g is called X-contravariant if there exist g'' and g' such that the following 2-LCO equality holds

$$g \;=\; \texttt{<}\nu \texttt{ : F\_PR}(ld,\pi) \texttt{ . } g''\texttt{;}\ \pi \texttt{ : F\_PR}(ld,\nu) \texttt{ . } g'\texttt{>}$$

The 2-LCO functors such as $g' : d \to e$ in the two above equalities are called witnesses of, respectively, X-covariance of g and X-contravariance of g.

We will call a *2-LCX* functor f X-covariant, if $[\![f]\!]$ is X-covariant. The same convention applies for the other definitions in this section. Note that if g belongs to the *2-LCX* carrier then $g'' = \mathtt{F\_OP}(g')$, by point 2 of the definition of the carrier. We conjecture that in very special but nontrivial categories witnesses of X-covariance are not unique. Details are out of scope of this thesis.

Example. The functor denoted by $\mathtt{F\_CONTRA}(a)$ is X-contravariant, for every category a. On the other hand the functor

$$\mathtt{<}\delta\ \mathtt{:}\ \mathtt{F\_CONTRA(*);}\ \epsilon\ \mathtt{:}\ \mathtt{F\_ID(*)>}$$

is neither X-covariant nor X-contravariant. By generalizing the notion we may describe the following functor

$$\mathtt{<}\delta\ \mathtt{:}\ \mathtt{F\_PR}(la,\iota)\ \mathtt{.}\ \mathtt{F\_CONTRA}(a)\mathtt{;}\ \epsilon\ \mathtt{:}\ \mathtt{F\_PR}(la,\kappa)\mathtt{>}$$

as X-contravariant on the ι component of the source and X-covariant on the κ component.

Definition 7.2.14. *Let g : $\mathtt{<}ld\mathtt{>} \to e'$ be a functor in some 2-LCO, where $lc = \iota\ \mathtt{-}\ c\mathtt{;}\ \kappa\ \mathtt{-}\ d$, $lc^{op} = \iota\ \mathtt{-}\ c^{op}\mathtt{;}\ \kappa\ \mathtt{-}\ d^{op}$ and let $ld = \nu\ \mathtt{-}\ \mathtt{<}lc\mathtt{>;}\ \pi\ \mathtt{-}\ \mathtt{<}lc^{op}\mathtt{>}$ be indexed lists of objects in the 2-LCO. The functor g is called X-covariant on the ι component, (or shortly: ι-covariant) if there exist functors g' and g'' such that the following equality holds*

$$\begin{aligned} g \quad = \quad & \mathtt{<}\nu\ \mathtt{:}\ \mathtt{<}\iota\ \mathtt{:}\ \mathtt{F\_PR}(ld,\nu)\ \mathtt{.}\ \mathtt{F\_PR}(lc,\iota)\mathtt{;} \\ & \qquad \kappa\ \mathtt{:}\ \mathtt{<}\nu\ \mathtt{:}\ \mathtt{F\_PR}(ld,\nu)\ \mathtt{.}\ \mathtt{F\_PR}(lc,\kappa)\mathtt{;} \\ & \qquad\qquad \pi\ \mathtt{:}\ \mathtt{F\_PR}(ld,\pi)\ \mathtt{.}\ \mathtt{F\_PR}(lc^{op},\kappa)\mathtt{>>}\ \mathtt{.}\ g'\mathtt{;} \\ & \pi\ \mathtt{:}\ \mathtt{<}\iota\ \mathtt{:}\ \mathtt{F\_PR}(ld,\pi)\ \mathtt{.}\ \mathtt{F\_PR}(lc^{op},\iota)\mathtt{;} \\ & \qquad \kappa\ \mathtt{:}\ \mathtt{<}\nu\ \mathtt{:}\ \mathtt{F\_PR}(ld,\pi)\ \mathtt{.}\ \mathtt{F\_PR}(lc^{op},\kappa)\mathtt{;} \\ & \qquad\qquad \pi\ \mathtt{:}\ \mathtt{F\_PR}(ld,\nu)\ \mathtt{.}\ \mathtt{F\_PR}(lc,\kappa)\mathtt{>>}\ \mathtt{.}\ g''\mathtt{>} \end{aligned}$$

The 2-LCO functor such as

$$g' : \mathtt{<}\iota\ \mathtt{-}\ c\mathtt{;}\ \kappa\ \mathtt{-}\ \mathtt{<}\nu\ \mathtt{-}\ d\mathtt{;}\ \pi\ \mathtt{-}\ d^{op}\mathtt{>>} \to e$$

in the above equality is called a witness of ι-covariance of g.

Similarly one can define ι-contravariance, κ-covariance, etc. The X-covariance definition can be straightforwardly extended to transformations.

Definition 7.2.15. *Let d be a category and let $ld = \nu\ \mathtt{-}\ d\mathtt{;}\ \pi\ \mathtt{-}\ d^{op}$. A 2-LCO transformation t is called X-covariant if there exist transformations t' and t'' such that*

$$t \quad = \quad \mathtt{<}\nu\ \mathtt{=}\ \mathtt{F\_PR}(ld,\nu)\ *\ t'\mathtt{;}\ \pi\ \mathtt{=}\ \mathtt{F\_PR}(ld,\pi)\ *\ t''\mathtt{>}$$

We analogously define X-contravariant transformations, ι-covariant transformations, etc.

Observation 7.2.16. *Every X-covariant transformation has X-covariant 2-LCO domain, 2-LCO codomain, 2-LCX domain (which is always defined in this case) and 2-LCX codomain.*

Observation 7.2.17. *Every X-contravariant transformation has X-contravariant 2-LCO domain, 2-LCO codomain, 2-LCX domain and 2-LCX codomain.*

Recall the interchange law (20) of Section 6.1.4:

$$(t_1 \,.\, t_2) * (u_1 \,.\, u_2) \quad = \quad (t_1 * u_1) \,.\, (t_2 * u_2)$$

and rule (29) of the same section:

$$t * u \quad \rightarrow \quad (f * u) \,.\, (t * h)$$

Of all the axioms of Φ^O, only the interchange law does not hold in *2-LCX* and of all the rules of $\mathcal{R}_2$ only rule (29) is not valid for *2-LCX*. However, also the implicit equations defining domain and codomain functors of horizontal composition and multiplications are different for *2-LCX* than for *2-LCO*. The dependency of the form of *2-LCX* domains and codomains on the variance of operands is similar as for the interchange law, which is discussed below. The first of the observations below is easy, but tedious to prove. The other observations in the remainder of this section are proved similarly as Lemma 7.2.26 near the end of this section.

Observation 7.2.18. *Equations 4–8, 15–19, 21–28, and 164–177 (i.e., all the axioms of Φ^O with the exception of the interchange law) seen as 2-LCX equations belong to Φ^X.*

Observation 7.2.19. *Let t_1 and t_2 be identity transformations. Let u_1 and u_2 be arbitrary transformations. Then (if well-typed) these transformations satisfy the interchange law.*

Observation 7.2.20. *Let u_1 and u_2 be X-covariant 2-LCX transformations. Let t_1 and t_2 be arbitrary 2-LCX transformations. Then these transformations satisfy the interchange law.*

Corollary 7.2.21. *All rules of $\mathcal{R}_2$, except rule (29), are sound with respect to Φ^X.*

Proof. By Observation 7.2.18 we only need to consider rules that are derived from Φ_2 with the use of the interchange law (20). These are only rules (32) and (37) (and (29), which in general is not valid in *2-LCX*, but see Lemma 7.2.26). Rule (32) is derivable from Φ^X by Observation 7.2.19 and rule (37) by Observation 7.2.20. □

Lemma 7.2.22. *There are 2-LCX transformations u_1, u_2, t_1 and t_2 such that the interchange law is type-correct for them but does not hold in some 2-LCX (in particular in the 2-LCX built from* **Cat***, where* `C_BB` *represents* **Set***).*

Proof. Let us fix the *2-LCX* obtained as a reduct of **Cat** with **Set** as `C_BB`. Let $f' : \texttt{C\_BB}^{op} \to \texttt{C\_BB}$ be a *2-LCX* functor that given a set s produces the set of functions from s to $\mathbb{Z}$ (the set of integers). Let $f = \texttt{F\_CONTRA(C\_BB)} \;.\; f'$ so that $f : \texttt{C\_BB} \to \texttt{C\_BB}$ is an X-contravariant *2-LCX* functor. Let $u_1 = u_2 : f \to f$ be functionals that given a function $g : s \to \mathbb{Z}$ produce the function $x \mapsto g(x) + 1$. (Functional u_1 can be expressed as `T_CONTRA` horizontally composed with a natural transformation with source in $\texttt{C\_BB}^{op}$. The failure of the interchange law is caused just by the single addition of `T_CONTRA`.) Let $h : \texttt{<>} \to \texttt{C\_BB}$ represent $\mathbb{R}$. Let $t_1 : h \to h$ represent $r \mapsto r * 0.5$ and let $t_2 : h \to h$ represent $r \mapsto r + 2.2$. Due to equal domains and codomains, the interchange law (20) is type-correct for these transformations.

For these functors we compute the horizontal composition as follows (for the general case see Lemma 7.2.26)

$$t_1 * u_1 = (t_1 * f) \;.\; (h * u_1)$$

where $h * u_1$ is u_1 instantiated to functions taking real numbers and $t_1 * f$ represents the functional $g \mapsto (r \mapsto g(t_1(r)))$ (as will be captured in the general case by rule (241) of Section 8.1.3). Therefore

$$t_1 * u_1 = g \mapsto (r \mapsto g(t_1(r)) + 1)$$

and so the right hand side of the interchange law is

$$(t_1 * u_1) \;.\; (t_2 * u_2) = g \mapsto (r \mapsto g(t_1(t_2(r))) + 2)$$

and consequently

$$(t_1 * u_1) \;.\; (t_2 * u_2) = g \mapsto (r \mapsto g(r * 0.5 + 1.1) + 2)$$

On the other hand, $u_1 \;.\; u_2$ adds 2 to a function and $t_1 \;.\; t_2$ multiplies a real number by 0.5 and adds 2.2. Computing the horizontal composition in the same way as before we get

$$(t_1 \;.\; t_2) * (u_1 \;.\; u_2) = ((t_1 \;.\; t_2) * f) \;.\; (h * (u_1 \;.\; u_2))$$

So the left hand side of the interchange law is

$$(t_1 \;.\; t_2) * (u_1 \;.\; u_2) = g \mapsto (r \mapsto g(r * 0.5 + 2.2) + 2)$$

If we now take g to be the floor function from real numbers to integers and apply the right hand side of the interchange law to g and then to real number 0.17, we get 3, while with the left hand side we get 4. □

Corollary 7.2.23. *2-LCX, in general, is not a 2-category.*

We see that the interchange law holds for some *2-LCX* morphisms and does not hold for others. This shows that to design a finite axiomatization of Φ^X we would have to come up with some versions of the law that are not type-correct for the offending morphisms. But for this, the language of *2-LCX* would have to be richer; for instance, an operation that is type-correct only for X-covariant operands might be needed.

Nevertheless when we know the variance of the morphisms, we may obtain quite strong and useful results. These results patch some of the holes caused by the interchange law being not type-correct in many cases and not true in general.

Observation 7.2.24. *Let u_1 and u_2 be 2-LCX transformations with X-covariant domains and codomains. Let t_1 and t_2 be arbitrary 2-LCX transformations. Then these transformations satisfy the interchange law.*

Observation 7.2.25. *Let u_1 and u_2 be 2-LCX transformations with X-contravariant domain and codomain. Let t_1 and t_2 be arbitrary transformations. Then these transformations satisfy the following 2-LCX equation called the X-contravariant interchange law.*

$$(t_1 \;.\; t_2) * (u_1 \;.\; u_2) \;=\; (t_2 * u_1) \;.\; (t_1 * u_2) \tag{186}$$

If the domains and codomains of transformations are neither X-covariant nor X-contravariant, usually no such simple equation describes the interchange of compositions.

The consequence of the interchange law used as the rewriting rule (29) is not correct in general, either. But it has some variants that are valid in limited cases. The proof of the following lemma is given in Appendix B.

Lemma 7.2.26. *Let t be an arbitrary 2-LCX transformation. Let u be a transformation with either X-covariant 2-LCX domain and codomain or X-contravariant 2-LCX domain and codomain. Let f be the 2-LCX domain of u. If f is X-covariant then let h be the codomain of t else if f is X-contravariant then let h be the domain of t. In both cases the transformations and functors satisfy the following 2-LCX equality.*

$$t * u \;=\; (t * f) \;.\; (h * u) \tag{187}$$

Observation 7.2.27. *Let t be an arbitrary 2-LCX transformation. Let u be a transformation with κ-covariant domain and codomain or κ-contravariant domain and codomain. If f is X-covariant then let h be the codomain of t else if f is X-contravariant then let h be the domain of t. In both cases the transformations and functors satisfy the following 2-LCX equality.*

$$\begin{aligned} \texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(g)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * u \;=\; & (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(g)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * f)\;. \\ & (\texttt{<}\iota\ \texttt{:}\ \texttt{F\_id}(g)\texttt{;}\ \kappa\ \texttt{:}\ h\texttt{>} * u) \end{aligned} \tag{188}$$

The usefulness of the two above equations lies in the fact that there are type-correct instances of them for each u and t with appropriate source and target. This is possible because f and h being domains and codomains, are guaranteed to exist and are always *2-LCX* functors. In contrast, the *2-LCO* domains and codomains of u and t cease being *2-LCX* functors as soon as the *2-LCX* is not very poor.

7.2.5 Conclusion

Categorical frameworks capable of accommodating fully parameterized exponents tend to be complicated because of the contravariance of the exponent functor on its first argument. Nevertheless, we have proposed such a framework and proved several of its properties, showing that it can be used in most cases in the same manner as our simple categorical models (Corollary 7.2.12, Corollary 7.2.21, Observation 7.2.24). For other cases we have provided operations and equations allowing to express and use contravariance (Lemma 7.2.6, Observation 7.2.25, Observation 7.2.27). In our framework it is possible to express source and target categories of parameterized adjoints, in particular of the fully parameterized exponent, as well as domain and codomain functors of the units and co-units in the respective adjunctions.

First, we have extended the categorical model *2-LC* by the notions of opposite categories, functors and transformations, obtaining *2-LCO*. We have observed that in a reachable *2-LCO* the operations constructing opposite functors and transformations can be expressed using solely the language of *2-LC* and opposite categories (Observation 7.2.1). This observation gives rise to equational axiomatization of *2-LCO* (Observation 7.2.2). A speculative experiment (Theorem 7.2.3) with the general functor that takes functors to their right adjoints allowed us to outline a parameterized adjoint operation together with its source and target categories. The experiment also indicates that expressing domain and codomain of the parameterized adjunction operation is not possible in *2-LCO*.

We have defined the language of *2-LCX* to be the same as of *2-LCO* but with an additional operation for presenting contravariant functors as covariant. We have constructed a *2-LCX* based on an arbitrary *2-LCO* and defined *2-LCX* operations by a translation to *2-LCO*. The resulting *2-LCX* is a subcategory of *2-LCO* (Observation 7.2.9), but in general it is not a 2-category (Corollary 7.2.23). We have shown that *2-LCX* is very close to being a 2-category; of all the *2-LCO* axioms it lacks only the interchange law (Observation 7.2.18). We have argued that *2-LCX* has no finite equational axiomatization and we have discussed parts of its rich equational theory. In particular, we proved some equalities that, depending on circumstances, are capable of playing the role of the interchange law.

7.3 Parameterized adjunctions

With *2-LCX* we have enough terms to fill in the gaps of the proposed definition of partial parameterized adjunction operations of Section 7.2.3. Let $le' = \iota$ - e; κ - d^{op} and $lc = \iota$ - c; κ - d. A sketch of the semantics of such operation might look as follows.

- `F_ori`(g) : <le'> $\to c$
  is a parameterized right adjoint to
  g : <lc> $\to e$,

- `T_orepsilon`(g) :
  <ι : `F_ori`(g); κ : `F_PR`(le', κ) . `F_CONTRA`(d^{op})> . $g \to$ `F_PR`(le', ι)
  is the parameterized co-unit of the adjunction between g : <lc> $\to e$ and `F_ori`(g).

In this very technical section we will try to formally add such operations to *2-LCX*, defining category *2-LCX-A*, its equational theory, and discussing term rewriting.

In Section 7.2.4 we have given a translation from *2-LCX* to *2-LCO*. Since **Cat** is a *2-LCO*, its reduct with respect to the translation is a *2-LCX*, which happens to be a subcategory of **Cat**. However, the adjunctions in **Cat** are not the adjunctions by which we wish to extend *2-LCX*, even if the parameterization is void. The desired adjunctions would have to be constructed from products, adjoints and other morphisms of **Cat**.

We divide the definition of *2-LCO-A* into two stages: first we extend *2-LCO* to *2-LCO-A* (*2-LCO* with adjunction-like structures defined in the next section) and then we extend the translation to become a translation from *2-LCX-A* (*2-LCX* with parameterized adjunctions) to *2-LCO-A*. The meaning of the *2-LCO* adjunction-like structures is not apparent until the translation is defined.

7.3.1 *2-LCO-A* — *2-LCO* with opposite adjunctions

We extend *2-LCO* with operations analogous to those for adjunctions, but with different typing, called opposite adjunction operations. They resemble the parameterized adjunction operations proposed above, but the opposite units and co-units require that the parameter d consists of two components representing, respectively, covariant and contravariant behavior of arguments. The fully general adjunction operations embedded in a framework that adequately models contravariance will be defined in the semantics of *2-LCX-A* that will treat the opposite adjunction operations as components of the adjunction operations of *2-LCX-A*.

Though the language of *2-LCO-A* contains the language of *2-LC-A*, the semantics of the adjunction operations does not agree with their semantics in *2-LC-A*. In particular, **Cat** (with standard horizontal and vertical compositions) can be extended with the opposite adjunction, but they are not the true adjunctions in **Cat**. The notion of *2-LCO-A* is the last notion that has an instance obtained by extending the signature of **Cat**. Any subsequent categories are based on reducts — reducts of *2-LCO-A* or reducts of its instances and their extensions.

Syntax

The additional terms of *2-LCO-A* with respect to *2-LCO* are exactly the same as those added to *2-LC* to form the syntax of *2-LC-A*. Yet the impact on the language will be different as the new terms are typed differently.

```
type cat =
  | C_PP of cat IList.t
  | C_BB
  | C_OP of cat
type funct =
  | F_ID of cat
  | F_COMP of funct * funct
  | F_PR of cat IList.t * IdIndex.t
  | F_RECORD of cat * funct IList.t
  | F_OP of funct
  | F_ri of funct
  | F_le of funct
type trans =
  | T_ID of cat
  | T_COMP of trans * trans
  | T_PR of cat IList.t * IdIndex.t
  | T_RECORD of cat * trans IList.t
  | T_FT of funct * trans
  | T_TF of trans * funct
  | T_id of funct
  | T_comp of trans * trans
  | T_OP of trans
  | T_repsilon of funct
  | T_reta of funct
  | T_lepsilon of funct
  | T_leta of funct
```

Semantics

The semantics of the old constructs of *2-LCO-A* is as in *2-LCO*. The typing of the new constructs of *2-LCO-A* is defined below, while their semantics follows from their role in the definition of parameterized adjunction operations of *2-LCX-A* in the next section. The reason, we (informally) describe *2-LCO-A* before we describe *2-LCX-A*, is to be able to (formally) extend the translation for the cases of the adjunction operations, when defining *2-LCX-A*. The translation is then used to construct the equational theory of *2-LCX-A*, which automatically provides a strict account of *2-LCO-A*. Just as in *2-LC-A* all the new operations are partial, and if an operation is defined on a particular functor, then the other two adjunction constituents have to be defined, too.

Let c, d and e be categories and let $le' = \iota$ - e; κ - d^{op} and $lc = \iota$ - c; κ - d. The notation $[\![$`F_CONTRA`$(a)]\!]$ will be used here with the meaning

$$<\nu : \texttt{F\_PR}(\nu \text{ - } a;\ \pi \text{ - } a^{op}, \pi);\ \pi : \texttt{F\_PR}(\nu \text{ - } a;\ \pi \text{ - } a^{op}, \nu)>$$

as in the previous sections. Note that while d in adjoint definitions might be arbitrary, the use of `F_CONTRA` implies that d in unit and co-unit definitions must be of the form $<\nu$ - a; π - $a^{op}>$.

- `F_ri`$(g) : <le'> \to c$
  is an opposite right adjoint to $g : <lc> \to e$,

- `F_le`$(g) : <le'> \to c$
  is an opposite left adjoint to $g : <lc> \to e$,

- `T_repsilon`(g) :
  $<\iota : \texttt{F\_ri}(g);\ \kappa : \texttt{F\_PR}(le', \kappa)\ .\ [\![\texttt{F\_CONTRA}(a^{op})]\!]>\ .\ g \to \texttt{F\_PR}(le', \iota)$
  is the co-unit in the opposite adjunction between $g : <lc> \to e$ and `F_ri`(g),

- `T_reta`(g) :
  $\texttt{F\_PR}(lc, \iota) \to <\iota : g;\ \kappa : \texttt{F\_PR}(lc, \kappa)\ .\ [\![\texttt{F\_CONTRA}(a)]\!]>\ .\ \texttt{F\_ri}(g)$
  is the unit in the opposite adjunction between $g : <lc> \to e$ and `F_ri`(g),

- `T_lepsilon`(g) :
  $<\iota : g;\ \kappa : \texttt{F\_PR}(lc, \kappa)\ .\ [\![\texttt{F\_CONTRA}(a)]\!]>\ .\ \texttt{F\_le}(g) \to \texttt{F\_PR}(lc, \iota)$
  is the co-unit in the opposite adjunction between functors `F_le`(g) and $g : <lc> \to e$,

- `T_leta`(g) :
  $\texttt{F\_PR}(le', \iota) \to <\iota : \texttt{F\_le}(g);\ \kappa : \texttt{F\_PR}(le', \kappa)\ .\ [\![\texttt{F\_CONTRA}(a^{op})]\!]>\ .\ g$
  is the unit in the opposite adjunction between functors `F_le`(g) and $g : <lc> \to e$.

Equations

The axiomatization of the equational theory of *2-LCO-A* when presented in the language of *2-LCO-A* is quite unreadable. For this reason we will cite only two axioms here.

$$\begin{aligned}&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g)\\ =\ &\texttt{F\_ri(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ \texttt{F\_OP}(f)\texttt{>}\ \texttt{.}\ g\texttt{)}\end{aligned} \tag{189}$$

$$\begin{aligned}&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ \texttt{F\_le}(g)\\ =\ &\texttt{F\_le(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ \texttt{F\_OP}(f)\texttt{>}\ \texttt{.}\ g\texttt{)}\end{aligned} \tag{190}$$

These equations closely resemble equation (178). Slightly weaker equations will be postulated for all the adjunction operations in *2-LCX-A*. Yet the strong versions of these equations here are indispensable to take advantage of the fact that all adjunctions we are interested in are adjunctions to functors X-covariant on both ι and κ components of the source. The equations make it possible to pull the projections of the definition of X-covariance outside of the adjoint constructor, thus making the adjoints to X-covariant functors κ-contravariant. Such adjoints have some nice properties, such as the simple equational presentation of their action on morphisms, developed in Section 7.3.4.

The full equational theory of *2-LCO-A* is generated by the axioms of Φ^O, a set of formulae stating that whenever one constructor of a given opposite adjunction is defined then all three should be defined, equations 189–190 and the image of the *2-LCX-A* equations 191–206 under the translation to be developed in the next section. We refrain from quoting the translated equations here, only because their meaning and role in the theory of *2-LCO-A* is essentially the same as in *2-LCX-A*, while their size is doubled, making them unreadable. There is no hidden technical difficulty here.

7.3.2 *2-LCX-A* — *2-LCX* with adjunctions

A category *2-LCX-A* is a *2-LCX* extended with left and right partial parameterized adjunction operations, where for each given argument functor the adjoint, unit and co-unit operations are all defined or all undefined. This structure is analogous to the one of Section 7.1.1, but the adjunction operands may be instantiated even after the adjunction operations are applied.

There are no natural non-trivial examples of *2-LCX-A*, but *2-LCX-A* can be seen as a formalization and generalization of the mechanism of parameterized adjunctions in **Cat**, as informally used in categorical discourse. This claim is substantiated by the fact that voidly parameterized *2-LCX-A* adjunctions satisfy (up to isomorphisms, due to the changed typing) all the equations that hold for *2-LC-A* adjunctions.

Syntax

The syntax of *2-LCX-A* is an extension of the syntax of *2-LCX*, analogously as the syntax of *2-LCO-A* has been an extension of the syntax of *2-LCO*. The `cat`-terms are unchanged, while the sets of `funct`-terms and `trans`-terms are extended, similarly as in *2-LCO-A*. Thus the syntax is a merger of the syntaxes of *2-LC-A* and *2-LCX*.

```
type cat =
  | C_PP of cat IList.t
  | C_BB
  | C_OP of cat
type funct =
  | F_ID of cat
  | F_COMP of funct * funct
  | F_PR of cat IList.t * IdIndex.t
  | F_RECORD of cat * funct IList.t
  | F_OP of funct
  | F_CONTRA of cat
  | F_ri of funct
  | F_le of funct
type trans =
  | T_ID of cat
  | T_COMP of trans * trans
  | T_PR of cat IList.t * IdIndex.t
  | T_RECORD of cat * trans IList.t
  | T_FT of funct * trans
  | T_TF of trans * funct
  | T_id of funct
  | T_comp of trans * trans
  | T_OP of trans
  | T_CONTRA of cat
  | T_repsilon of funct
  | T_reta of funct
  | T_lepsilon of funct
  | T_leta of funct
```

Semantics

Extending the translation from *2-LCX* to *2-LCO* of Section 7.2.4 to a translation from *2-LCX-A* to *2-LCO-A* provides an essential intuition about the semantics of *2-LCX-A*. The strictly formal semantics will be obtained by defining the equational theory of *2-LCX-A* in the next subsection. The theory will be constructed with the help of the translation.

Let the following be indexed lists of *2-LCO-A* categories: $le = \iota$ - e; κ - d, $lc = \iota$ - c; κ - d, $le^{op} = \iota$ - e^{op}; κ - d^{op}, $lc^{op} = \iota$ - c^{op}; κ - d^{op}, $ld' = \nu$ - <le>; π - <le^{op}> and $ld = \nu$ - <lc>; π - <lc^{op}>. Then by $P_\nu(ld')$ we mean the *2-LCO* functor that is the value of the following term

$$\begin{array}{l}\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(ld',\nu)\ \texttt{.}\ \texttt{F\_PR}(le,\iota)\texttt{;}\\ \kappa\ \texttt{:}\ \texttt{<}\nu\ \texttt{:}\ \texttt{F\_PR}(ld',\pi)\ \texttt{.}\ \texttt{F\_PR}(le^{op},\kappa)\texttt{;}\ \pi\ \texttt{:}\ \texttt{F\_PR}(ld',\nu)\ \texttt{.}\ \texttt{F\_PR}(le,\kappa)\texttt{>>}\end{array}$$

and by $P_\pi(ld')$ we mean the value of

$$[\![\texttt{F\_CONTRA}(\texttt{<}le\texttt{>})]\!]\ \texttt{.}\ P_\nu(ld')^{op}$$

which is Φ^O-equal to

$$\begin{array}{l}\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(ld',\pi)\ \texttt{.}\ \texttt{F\_PR}(le^{op},\iota)\texttt{;}\\ \kappa\ \texttt{:}\ \texttt{<}\nu\ \texttt{:}\ \texttt{F\_PR}(ld',\nu)\ \texttt{.}\ \texttt{F\_PR}(le,\kappa)\texttt{;}\ \pi\ \texttt{:}\ \texttt{F\_PR}(ld',\pi)\ \texttt{.}\ \texttt{F\_PR}(le^{op},\kappa)\texttt{>>}\end{array}$$

Unlike the translation of Section 7.2.4, the "translation" we define below, though it provides an interpretation of (partial) *2-LCX-A* operations in terms of operations of *2-LCO-A*, does not induce a total function on terms. Firstly, we define the new *2-LCX-A* operations only for ι-covariant operands, so the interpretation becomes partial itself. This is, however, enough to obtain a *2-LCX-A* category by reduct of any *2-LCO-A* and the same restriction of carriers as in Section 7.2.4. Secondly, the interpretation refers to witnesses of ι-covariance, which are not always expressible in *2-LCO-A*. Therefore the actual translation is defined only for those *2-LCX-A* terms, for which all necessary ι-covariance witnesses exist and, moreover, are expressible, while the interpretation we use is not restricted by expressibility.

The interpretation of adjunction term constructors will be defined only for those *2-LCX-A* operand functors (shown here with typing expressed in the language of *2-LCX-A*)

$$g : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to e$$

that are ι-covariant. For each such functor g, let a witness (a *2-LCO-A* functor, as described in Definition 7.2.14) of the fact that

$$[\![g]\!] : \texttt{<}ld\texttt{>} \to \texttt{<}\nu\ \texttt{-}\ e\texttt{;}\ \pi\ \texttt{-}\ e^{op}\texttt{>}$$

is ι-covariant be called and typed as follows

$$g' : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ \texttt{<}\nu\ \texttt{-}\ d\texttt{;}\ \pi\ \texttt{-}\ d^{op}\texttt{>>} \to e$$

The witness is unique in all categories we are likely to use as a model of a programming language (details are beyond the scope of this thesis). However, to make the translation fully general, let us fix a choice of witnesses. In particular, for any X-covariant functor let us fix a choice of witness g_1 of its X-covariance and let the witness of its implied ι-covariance be

$$g' \;=\; \texttt{<}\iota\texttt{ : F\_PR}(lb,\iota)\texttt{; }\kappa\texttt{ : F\_PR}(lb,\kappa)\texttt{ . F\_PR}(la,\nu)\texttt{> . }g_1$$

Such a choice makes proofs about the translation easier; see for example the proof of Theorem 7.3.2 below.

The interpretation of *2-LCX* operations in *2-LCO* is extended by the following clauses, where g, g', ld', ld, P_ν and P_π are as discussed above.

$$\begin{aligned}
[\![\texttt{F\_ri}(g)]\!] &= \texttt{<}\nu\texttt{ : }P_\nu(ld')\texttt{ . F\_ri}(g')\texttt{;}\\
&\qquad \pi\texttt{ : }P_\pi(ld')\texttt{ . F\_le(F\_OP}(g')\texttt{))>}\\
[\![\texttt{F\_le}(g)]\!] &= \texttt{<}\nu\texttt{ : }P_\nu(ld')\texttt{ . F\_le}(g')\texttt{;}\\
&\qquad \pi\texttt{ : }P_\pi(ld')\texttt{ . F\_ri(F\_OP}(g')\texttt{))>}\\
[\![\texttt{T\_repsilon}(g)]\!] &= \texttt{<}\nu\texttt{ = }P_\nu(ld') * \texttt{T\_repsilon}(g')\texttt{;}\\
&\qquad \pi\texttt{ = }P_\pi(ld') * \texttt{T\_leta(F\_OP}(g')\texttt{))>}\\
[\![\texttt{T\_reta}(g)]\!] &= \texttt{<}\nu\texttt{ = }P_\nu(ld) * \texttt{T\_reta}(g')\texttt{;}\\
&\qquad \pi\texttt{ = }P_\pi(ld) * \texttt{T\_lepsilon(F\_OP}(g')\texttt{))>}\\
[\![\texttt{T\_lepsilon}(g)]\!] &= \texttt{<}\nu\texttt{ = }P_\nu(ld) * \texttt{T\_lepsilon}(g')\texttt{;}\\
&\qquad \pi\texttt{ = }P_\pi(ld) * \texttt{T\_reta(F\_OP}(g')\texttt{))>}\\
[\![\texttt{T\_leta}(g)]\!] &= \texttt{<}\nu\texttt{ = }P_\nu(ld') * \texttt{T\_leta}(g')\texttt{;}\\
&\qquad \pi\texttt{ = }P_\pi(ld') * \texttt{T\_repsilon(F\_OP}(g')\texttt{))>}
\end{aligned}$$

The partiality of the interpretation may be captured semantically by a requirement that only adjoint constructors applied to ι-covariant functors are defined in our algebras, and on the equational ground by ensuring that the definedness predicate is never used for terms containing adjunction constructors applied to functors that are not guaranteed to be ι-covariant. We do not know whether adjunctions to functors that are not ι-covariant may model any interesting features of programming languages and we even do not know which of their theories are consistent.

The carriers of *2-LCX-A* and domains and codomains are defined analogously as for *2-LCX*. The semantics of new operations is easily seen to be well defined, as stated in the following observation.

Observation 7.3.1. *The 2-LCO-A morphisms resulting from the translation belong to the 2-LCX-A carrier, satisfying the conditions of Definition 7.2.4.*

The *2-LCX-A* algebras obtained as reducts with respect to the partial interpretations of operations are partial as well: for a given *2-LCO-A* m and a *2-LCX-A* signature operation p the application of p to an argument a is undefined in the reduct *2-LCX-A* (the reduct of m with respect to the interpretation), if either the interpretation is undefined for p or the operations resulting from the interpretation of p are undefined in m for the argument a.

As can be easily proved using the definition of the translation, the results of application of adjoint constructors are always ι-covariant, just as their operands. In case of the standard adjoint functors, such as the product or parameterized exponent, the operands are additionally fully X-covariant (both ι-covariant and κ-covariant), which is stated in a general way below.

Theorem 7.3.2. *Adjoints to X-covariant functors are κ-contravariant.*

Proof. Let g be a *2-LCX-A* functor. Let g' be the chosen witness of ι-covariance of $[\![g]\!]$, just as in the definition of the translation, so that we have

$$\begin{aligned}[\![g]\!] \quad = \quad & \texttt{<}\nu : \texttt{<}\iota : \texttt{F\_PR}(ld, \nu) \ \texttt{.}\ \texttt{F\_PR}(lc, \iota)\texttt{;} \\ & \qquad\qquad \kappa : \texttt{<}\nu : \texttt{F\_PR}(ld, \nu) \ \texttt{.}\ \texttt{F\_PR}(lc, \kappa)\texttt{;} \\ & \qquad\qquad\qquad \pi : \texttt{F\_PR}(ld, \pi) \ \texttt{.}\ \texttt{F\_PR}(lc^{op}, \kappa)\texttt{>>} \ \texttt{.}\ g'\texttt{;} \\ & \quad \pi : \texttt{<}\iota : \texttt{F\_PR}(ld, \pi) \ \texttt{.}\ \texttt{F\_PR}(lc^{op}, \iota)\texttt{;} \\ & \qquad\qquad \kappa : \texttt{<}\nu : \texttt{F\_PR}(ld, \pi) \ \texttt{.}\ \texttt{F\_PR}(lc^{op}, \kappa)\texttt{;} \\ & \qquad\qquad\qquad \pi : \texttt{F\_PR}(ld, \nu) \ \texttt{.}\ \texttt{F\_PR}(lc, \kappa)\texttt{>>} \ \texttt{.}\ \texttt{F\_OP}(g')\texttt{>}\end{aligned}$$

Moreover let $[\![g]\!]$ be X-covariant with the witness g_1, so that

$$[\![g]\!] \quad = \quad \texttt{<}\nu : \texttt{F\_PR}(ld, \nu) \ \texttt{.}\ g_1\texttt{;}\ \pi : \texttt{F\_PR}(ld, \pi) \ \texttt{.}\ \texttt{F\_OP}(g_1)\texttt{>}$$

By the choice of witnesses we have

$$g' \quad = \quad \texttt{<}\iota : \texttt{F\_PR}(lb, \iota)\texttt{;}\ \kappa : \texttt{F\_PR}(lb, \kappa) \ \texttt{.}\ \texttt{F\_PR}(la, \nu)\texttt{>} \ \texttt{.}\ g_1$$

Then by equation (189) we derive

$$\texttt{F\_ri}(g') \quad = \quad \texttt{<}\iota : \texttt{F\_PR}(lb_1, \iota)\texttt{;}\ \kappa : \texttt{F\_PR}(lb_1, \kappa) \ \texttt{.}\ \texttt{F\_PR}(la^{op}, \nu)\texttt{>} \ \texttt{.}\ \texttt{F\_ri}(g_1)$$

On the other hand, from the definition of the translation we have

$$[\![\texttt{F\_ri}(g)]\!] \ \texttt{.}\ \texttt{F\_PR}(lc, \nu) \quad = \quad P_\nu(ld') \ \texttt{.}\ \texttt{F\_ri}(g')$$

which can be simplified using the equation above to

$$\begin{aligned}[\![\texttt{F\_ri}(g)]\!] \ \texttt{.}\ \texttt{F\_PR}(lc, \nu) \quad = \quad & \texttt{<}\iota : \texttt{F\_PR}(ld', \nu) \ \texttt{.}\ \texttt{F\_PR}(le, \iota)\texttt{;} \\ & \quad \kappa : \texttt{F\_PR}(ld', \pi) \ \texttt{.}\ \texttt{F\_PR}(le^{op}, \kappa)\texttt{>} \ \texttt{.}\ \texttt{F\_ri}(g_1)\end{aligned}$$

Let g_2 be the following functor

$$\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb_2,\iota)\ \texttt{.}\ \texttt{F\_PR}(la_2,\nu)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb_2,\kappa)\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g_1)$$

We see that

$$\begin{aligned}[\![\texttt{F\_ri}(g)]\!]\ \texttt{.}\ \texttt{F\_PR}(le,\nu) \;=\; &\texttt{<}\iota\ \texttt{:}\ \texttt{<}\nu\ \texttt{:}\ \texttt{F\_PR}(ld',\nu)\ \texttt{.}\ \texttt{F\_PR}(le,\iota)\texttt{;}\\ &\qquad\pi\ \texttt{:}\ \texttt{F\_PR}(ld',\pi)\ \texttt{.}\ \texttt{F\_PR}(le^{op},\iota)\texttt{>;}\\ &\kappa\ \texttt{:}\ \texttt{F\_PR}(ld',\pi)\ \texttt{.}\ \texttt{F\_PR}(le^{op},\kappa)\texttt{>}\ \texttt{.}\ g_2\end{aligned}$$

Reasoning analogously we can also find a functor g_2' such that

$$\begin{aligned}[\![\texttt{F\_ri}(g)]\!]\ \texttt{.}\ \texttt{F\_PR}(le,\pi) \;=\; &\texttt{<}\iota\ \texttt{:}\ \texttt{<}\nu\ \texttt{:}\ \texttt{F\_PR}(ld',\pi)\ \texttt{.}\ \texttt{F\_PR}(le^{op},\iota)\texttt{;}\\ &\qquad\pi\ \texttt{:}\ \texttt{F\_PR}(ld',\nu)\ \texttt{.}\ \texttt{F\_PR}(le,\iota)\texttt{>;}\\ &\kappa\ \texttt{:}\ \texttt{F\_PR}(ld',\nu)\ \texttt{.}\ \texttt{F\_PR}(le,\kappa)\texttt{>}\ \texttt{.}\ g_2'\end{aligned}$$

Together we obtain

$$\begin{aligned}[\![\texttt{F\_ri}(g)]\!] \;=\; &\texttt{<}\nu\ \texttt{:}\ \texttt{<}\iota\ \texttt{:}\ \texttt{<}\nu\ \texttt{:}\ \texttt{F\_PR}(ld',\nu)\ \texttt{.}\ \texttt{F\_PR}(le,\iota)\texttt{;}\\ &\qquad\qquad\pi\ \texttt{:}\ \texttt{F\_PR}(ld',\pi)\ \texttt{.}\ \texttt{F\_PR}(le^{op},\iota)\texttt{>;}\\ &\qquad\kappa\ \texttt{:}\ \texttt{F\_PR}(ld',\pi)\ \texttt{.}\ \texttt{F\_PR}(le^{op},\kappa)\texttt{>}\ \texttt{.}\ g_2\texttt{;}\\ &\pi\ \texttt{:}\ \texttt{<}\iota\ \texttt{:}\ \texttt{<}\nu\ \texttt{:}\ \texttt{F\_PR}(ld',\pi)\ \texttt{.}\ \texttt{F\_PR}(le^{op},\iota)\texttt{;}\\ &\qquad\qquad\pi\ \texttt{:}\ \texttt{F\_PR}(ld',\nu)\ \texttt{.}\ \texttt{F\_PR}(le,\iota)\texttt{>;}\\ &\qquad\kappa\ \texttt{:}\ \texttt{F\_PR}(ld',\nu)\ \texttt{.}\ \texttt{F\_PR}(le,\kappa)\texttt{>}\ \texttt{.}\ g_2'\texttt{>}\end{aligned}$$

which is exactly the definition of κ-contravariance of $[\![\texttt{F\_ri}(g)]\!]$ with g_2' as a witness. □

We do not require X-covariance of adjoint operands in the definition of the translation, because the set of adjoints to X-covariant functors is not closed under instantiation with arbitrary functors, unlike the set of adjoints to ι-covariant functors. The X-covariance of the operands is assumed only in Section 7.3.4, where we give a reduction rule for multiplication by adjoints. However, in this case the restriction to X-covariant operands can be overcome with the use of constructors such as `T_pp`, `T_ee`, etc., see Section 8.1.3.

Example. Let the parameterized exponent functor, with one argument labeled `a`, be $\texttt{F\_ri}(\mathbf{Z})$, where $lb = \texttt{a - *}$ and

$$\begin{aligned}\mathbf{Z} &= \texttt{\{}\upsilon\ \texttt{:}\ \texttt{F\_PR}(le,\iota)\texttt{;}\ \texttt{a}\ \texttt{:}\ \texttt{F\_PR}(le,\kappa)\ \texttt{.}\ \texttt{F\_PR}(lb,\texttt{a})\texttt{\}}\\ \mathbf{Z} &: \texttt{<}\iota\ \texttt{- *;}\ \kappa\ \texttt{- <}lb\texttt{>>} \to \texttt{*}\\ \texttt{F\_ri}(\mathbf{Z}) &: \texttt{<}\iota\ \texttt{- *;}\ \kappa\ \texttt{- <}lb\texttt{>>} \to \texttt{*}\end{aligned}$$

We see that the operand functor $\mathbf{Z}$ is X-covariant and so `F_ri`$(\mathbf{Z})$ is ι-covariant and κ-contravariant. Let us compose the exponent functor so as to obtain a nested exponent (the nested occurrence of the exponent will appear at the argument position of the top-level exponent):

$$\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ \texttt{<a : F\_ri}(\mathbf{Z})\texttt{>>}\ \texttt{.}\ \texttt{F\_ri}(\mathbf{Z})$$

The composition is neither κ-contravariant nor κ-covariant. By equation (195) given below it is equal to

$$\texttt{F\_ri(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ \texttt{<a : F\_ri}(\mathbf{Z})\texttt{>>}\ \texttt{.}\ \mathbf{Z})$$

and by definition of $\mathbf{Z}$ it is equal to

$$\texttt{F\_ri(\{}\upsilon\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \texttt{a}\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ \texttt{<a : F\_ri}(\mathbf{Z})\texttt{>}\ \texttt{.}\ \texttt{F\_PR}(lb,\texttt{a})\texttt{\})}$$

This can be simplified to the following form

$$\texttt{F\_ri(\{}\upsilon\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \texttt{a}\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ \texttt{F\_ri}(\mathbf{Z})\texttt{\})}$$

where the operand of the adjoint is evidently no longer X-covariant, but only ι-covariant.

We will now present the *2-LCX-A* typing of the new operations (determined as in Definition 7.2.8) assuming the operand is

$$g : \texttt{<}\iota\ \texttt{-}\ c\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to e$$

Note that the contravariance of adjoints is no longer visible in the typing below, instead the X-contravariance is clear from the semantics. The lack of opposite categories and functors as well as `F_CONTRA` morphisms in the typing makes it possible to treat instances of *2-LCX-A* as extensions of a (somewhat restricted, due to the changed domain and codomain functors for horizontal composition) *2-LC*, see Section 8.1.4 for explanation.

- `F_ri`$(g) : \texttt{<}\iota\ \texttt{-}\ e\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to c$
- `F_le`$(g) : \texttt{<}\iota\ \texttt{-}\ e\texttt{;}\ \kappa\ \texttt{-}\ d\texttt{>} \to c$
- `T_repsilon`$(g) : \texttt{<}\iota\ \texttt{:}\ \texttt{F\_ri}(g)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(le,\kappa)\texttt{>}\ \texttt{.}\ g \to \texttt{F\_PR}(le,\iota)$
- `T_reta`$(g) : \texttt{F\_PR}(lc,\iota) \to \texttt{<}\iota\ \texttt{:}\ g\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lc,\kappa)\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g)$
- `T_lepsilon`$(g) : \texttt{<}\iota\ \texttt{:}\ g\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lc,\kappa)\texttt{>}\ \texttt{.}\ \texttt{F\_le}(g) \to \texttt{F\_PR}(lc,\iota)$
- `T_leta`$(g) : \texttt{F\_PR}(le,\iota) \to \texttt{<}\iota\ \texttt{:}\ \texttt{F\_le}(g)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(le,\kappa)\texttt{>}\ \texttt{.}\ g$

In *2-LCX-A*, the *2-LCO-A* domains and codomains of transformations often fall outside of the *2-LCX-A* carrier (as described in Definition 7.2.4), with the important exception of domains and codomains of 2-identities. Although the units and co-units of expressible functors have expressible *2-LCX-A* domains and codomains, as seen in the above typing, their *2-LCO-A* domains and codomains usually do not even belong to the *2-LCX-A* carrier. Consequently, when for *2-LCO-A* terms we consider their *2-LCO-A* typing (as opposed to *2-LCX-A* typing from Definition 7.2.8), the translation does not preserve typing.

As mentioned before, the typing of `T_COMP` in the presence of `F_CONTRA` is very complex, as seen in the example below. This is especially apparent when horizontally composing terms involving the adjunction operations. The operation `T_COMP` is the categorical horizontal composition in *2-LC*, but not so in *2-LCX-A*, which needn't be a 2-category (Corollary 7.2.23). Restricting the use of `T_COMP` and `T_TF` helps in keeping instances of *2-LCX-A* close to the 2-categorical intuition, as argued in Section 8.1.4, but is not needed for *2-LCX-A* to be a category nor for its $C(c, e)$ structures to be categories — `T_COMP` is well defined, even if reasoning about it is very complicated in *2-LCX-A*.

Example. Let $\texttt{T\_pr}(lf, i)$ be an abbreviation for the usual projection operation related to the labeled product adjunction. Let $\texttt{F\_ri}(\mathbf{Z})$ be the exponent functor, as defined in the previous example. We will derive the *2-LCX-A* domain and codomain functors for the following horizontal composition (strictly speaking, multiplication by a functor from the right).

$$\texttt{T\_pr}(lf, i) * (\texttt{<}\iota \texttt{ : F\_ID(*); } \kappa \texttt{ : <a : F\_ID(*)>> . F\_ri}(\mathbf{Z}))$$

The term, when interpreted in *2-LCX-A* built upon **Cat** with **Set** as `*`, represents a functional that given a function applies it to a projected argument and then projects the result: $g \mapsto (p \mapsto \texttt{T\_pr}(g(\texttt{T\_pr}(p))))$.

We know that

$$\texttt{T\_pr}(lf, i) : \texttt{\{}lf\texttt{\}} \to f_i$$

With the standard *2-LC* typing we would get the following domain of the horizontal composition

$$\begin{aligned}\texttt{\{}lf\texttt{\} . (<}\iota \texttt{ : F\_ID(*); } \kappa \texttt{ : <a : F\_ID(*)>> . F\_ri}(\mathbf{Z})) &= \\ (\texttt{<}\iota \texttt{ : \{}lf\texttt{\}; } \kappa \texttt{ : <a : \{}lf\texttt{\}>> . F\_ri}(\mathbf{Z}))&\end{aligned}$$

and the following codomain

$$(\texttt{<}\iota \texttt{ : } f_i\texttt{; } \kappa \texttt{ : <a : } f_i\texttt{>> . F\_ri}(\mathbf{Z}))$$

which are wrong considering the interpretation in **Set**. In particular, the implied type of arguments p of the resulting function (the "`a`" field in the term's codomain) is not a product, so the arguments cannot be projected.

To compute the correct typing, let us translate the term to the *2-LCO-A* language

$$[\![\texttt{T\_pr}(lf,i) * (\texttt{<}\iota : \texttt{F\_ID(*)};\ \kappa : \texttt{<a : F\_ID(*)>> . F\_ri}(\mathbf{Z}))]\!]$$

obtaining

$$\begin{aligned}
&[\![\texttt{T\_pr}(lf,i)]\!]\\
&* \texttt{<}\nu : \texttt{<}\iota : \texttt{F\_PR}(lb',\nu);\ \kappa : \texttt{<a : F\_PR}(lb',\nu)\texttt{>>};\\
&\quad \pi : \texttt{<}\iota : \texttt{F\_PR}(lb',\pi);\ \kappa : \texttt{<a : F\_PR}(lb',\pi)\texttt{>>>}\\
&* \texttt{<}\nu : P_\nu(ld')\ \texttt{. F\_ri}(g');\\
&\quad \pi : P_\pi(ld')\ \texttt{. F\_le(F\_OP}(g'))\texttt{>}
\end{aligned}$$

where g' is the witness of ι-covariance of $[\![g]\!]$. This can be simplified to

$$\begin{aligned}
&\texttt{<}\nu = \texttt{<}\iota = [\![\texttt{T\_pr}(lf,i)]\!]^\nu;\ \kappa = \texttt{<a} = [\![\texttt{T\_pr}(lf,i)]\!]^\nu\texttt{>>};\\
&\ \pi = \texttt{<}\iota = [\![\texttt{T\_pr}(lf,i)]\!]^\pi;\ \kappa = \texttt{<a} = [\![\texttt{T\_pr}(lf,i)]\!]^\pi\texttt{>>>}\\
&* \texttt{<}\nu : P_\nu(ld')\ \texttt{. F\_ri}(g');\\
&\quad \pi : P_\pi(ld')\ \texttt{. F\_le(F\_OP}(g'))\texttt{>}
\end{aligned}$$

Let g_1 be the witness of X-covariance of $[\![g]\!]$, chosen as in the proof of Theorem 7.3.2. Then, similarly as in the proof, we can derive

$$\begin{aligned}
&\texttt{<}\nu = \texttt{<}\iota = [\![\texttt{T\_pr}(lf,i)]\!]^\nu;\ \kappa = \texttt{<a} = [\![\texttt{T\_pr}(lf,i)]\!]^\nu\texttt{>>};\\
&\ \pi = \texttt{<}\iota = [\![\texttt{T\_pr}(lf,i)]\!]^\pi;\ \kappa = \texttt{<a} = [\![\texttt{T\_pr}(lf,i)]\!]^\pi\texttt{>>>}\\
&* \texttt{<}\nu : \texttt{<}\iota : \texttt{F\_PR}(ld',\nu)\ \texttt{. F\_PR}(le,\iota);\\
&\qquad \kappa : \texttt{F\_PR}(ld',\pi)\ \texttt{. F\_PR}(le^{op},\kappa)\texttt{> . F\_ri}(g_1);\\
&\quad \pi : \texttt{<}\iota : \texttt{F\_PR}(ld',\pi)\ \texttt{. F\_PR}(le,\iota);\\
&\qquad \kappa : \texttt{F\_PR}(ld',\nu)\ \texttt{. F\_PR}(le^{op},\kappa)\texttt{> . F\_le(F\_OP}(g_1))\texttt{>}
\end{aligned}$$

which can be simplified to

$$\begin{aligned}
&\texttt{<}\nu = \texttt{<}\iota = [\![\texttt{T\_pr}(lf,i)]\!]^\nu;\\
&\qquad \kappa = \texttt{<a} = [\![\texttt{T\_pr}(lf,i)]\!]^\pi\texttt{>>} * \texttt{F\_ri}(g_1);\\
&\quad \pi = \texttt{<}\iota = [\![\texttt{T\_pr}(lf,i)]\!]^\pi;\\
&\qquad \kappa = \texttt{<a} = [\![\texttt{T\_pr}(lf,i)]\!]^\nu\texttt{>>} * \texttt{F\_le(F\_OP}(g_1))\texttt{>}
\end{aligned}$$

We have the following *2-LCO-A* (not *2-LCX-A*!) typing of the projection, by Definition 7.2.8

$$\begin{aligned}
[\![\texttt{T\_pr}(lf,i)]\!]^\nu &: [\![\texttt{\{}lf\texttt{\}}]\!]^\nu \to [\![f_i]\!]^\nu\\
[\![\texttt{T\_pr}(lf,i)]\!]^\pi &: [\![f_i]\!]^\pi \to [\![\texttt{\{}lf\texttt{\}}]\!]^\pi
\end{aligned}$$

Therefore the *2-LCO-A* domain of the horizontal composition is

$$\begin{aligned}
&\texttt{<}\nu : \texttt{<}\iota : [\![\texttt{\{}lf\texttt{\}}]\!]^\nu;\\
&\qquad \kappa : \texttt{<a} : [\![f_i]\!]^\pi\texttt{>> . F\_ri}(g_1);\\
&\quad \pi : \texttt{<}\iota : [\![f_i]\!]^\pi;\\
&\qquad \kappa : \texttt{<a} : [\![\texttt{\{}lf\texttt{\}}]\!]^\nu\texttt{>> . F\_le(F\_OP}(g_1))\texttt{>}
\end{aligned}$$

and the *2-LCO-A* codomain is

$$\begin{array}{l} \texttt{<}\nu \texttt{ : <}\iota \texttt{ : } [\![f_i]\!]^\nu\texttt{;} \\ \qquad \kappa \texttt{ : <a : } [\![\texttt{\{}lf\texttt{\}}]\!]^\pi\texttt{>> . F\_ri(}g_1\texttt{);} \\ \ \pi \texttt{ : <}\iota \texttt{ : } [\![\texttt{\{}lf\texttt{\}}]\!]^\pi\texttt{;} \\ \qquad \kappa \texttt{ : <a : } [\![f_i]\!]^\nu\texttt{>> . F\_le(F\_OP(}g_1\texttt{))>} \end{array}$$

Now we would like to revert to the g' functor, so we will prove that the following subterm of the codomain

$$\begin{array}{l} \texttt{<}\iota \texttt{ : } [\![f_i]\!]^\nu\texttt{;} \\ \ \kappa \texttt{ : <a : } [\![\texttt{\{}lf\texttt{\}}]\!]^\pi\texttt{>> . F\_ri(}g_1\texttt{)} \end{array}$$

is equal to

$$\begin{array}{l} [\![\texttt{<}\iota \texttt{ : } f_i\texttt{;} \\ \ \kappa \texttt{ : <a : \{}lf\texttt{\}>>}]\!] \texttt{ . } P_\nu(ld') \texttt{ . F\_ri(}g'\texttt{)} \end{array}$$

Let us again use the facts from the proof of Theorem 7.3.2 to transform the latter term to

$$\begin{array}{l} [\![\texttt{<}\iota \texttt{ : } f_i\texttt{;} \\ \ \kappa \texttt{ : <a : \{}lf\texttt{\}>>}]\!] \\ \texttt{ . <}\iota \texttt{ : F\_PR(}ld', \nu\texttt{) . F\_PR(}le, \iota\texttt{);} \\ \quad \kappa \texttt{ : F\_PR(}ld', \pi\texttt{) . F\_PR(}le^{op}, \kappa\texttt{)> . F\_ri(}g_1\texttt{)} \end{array}$$

then we will translate the record

$$\begin{array}{l} \texttt{<}\nu \texttt{ : <}\iota \texttt{ : } [\![f_i]\!]^\nu\texttt{; } \kappa \texttt{ : <a : } [\![\texttt{\{}lf\texttt{\}}]\!]^\nu\texttt{>>;} \\ \ \pi \texttt{ : <}\iota \texttt{ : } [\![f_i]\!]^\pi\texttt{; } \kappa \texttt{ : <a : } [\![\texttt{\{}lf\texttt{\}}]\!]^\pi\texttt{>>} \\ \texttt{ . <}\iota \texttt{ : F\_PR(}ld', \nu\texttt{) . F\_PR(}le, \iota\texttt{);} \\ \quad \kappa \texttt{ : F\_PR(}ld', \pi\texttt{) . F\_PR(}le^{op}, \kappa\texttt{)> . F\_ri(}g_1\texttt{)} \end{array}$$

and simplify to the expected form

$$\begin{array}{l} \texttt{<}\iota \texttt{ : } [\![f_i]\!]^\nu\texttt{;} \\ \ \kappa \texttt{ : <a : } [\![\texttt{\{}lf\texttt{\}}]\!]^\pi\texttt{>> . F\_ri(}g_1\texttt{)} \end{array}$$

The other subterms of the codomain and domain can be transformed analogously.

We may now revert to the g' functor and present the *2-LCO-A* domain as follows

$$\begin{array}{l} \texttt{<}\nu \texttt{ : } [\![\texttt{<}\iota \texttt{ : \{}lf\texttt{\};} \\ \qquad \kappa \texttt{ : <a : } f_i\texttt{>>}]\!] \texttt{ . } P_\nu(ld') \texttt{ . F\_ri(}g'\texttt{);} \\ \ \pi \texttt{ : } [\![\texttt{<}\iota \texttt{ : } f_i\texttt{;} \\ \qquad \kappa \texttt{ : <a : \{}lf\texttt{\}>>}]\!] \texttt{ . } P_\pi(ld') \texttt{ . F\_le(F\_OP(}g'\texttt{))>} \end{array}$$

and the *2-LCO-A* codomain as follows

$$
\begin{array}{l}
\texttt{<}\nu \texttt{ : } [\![\texttt{<}\iota \texttt{ : } f_i\texttt{;} \\
\qquad \kappa \texttt{ : <a : \{}lf\texttt{\}>>}]\!] \texttt{ . } P_\nu(ld') \texttt{ . F\_ri(}g'\texttt{);} \\
\pi \texttt{ : } [\![\texttt{<}\iota \texttt{ : \{}lf\texttt{\};} \\
\qquad \kappa \texttt{ : <a : } f_i\texttt{>>}]\!] \texttt{ . } P_\pi(ld') \texttt{ . F\_le(F\_OP(}g'\texttt{))>}
\end{array}
$$

Then by Definition 7.2.8 the *2-LCX-A* domain is

$$
\begin{array}{l}
\texttt{<}\nu \texttt{ : } [\![\texttt{<}\iota \texttt{ : \{}lf\texttt{\};} \\
\qquad \kappa \texttt{ : <a : } f_i\texttt{>>}]\!] \texttt{ . } P_\nu(ld') \texttt{ . F\_ri(}g'\texttt{);} \\
\pi \texttt{ : } [\![\texttt{<}\iota \texttt{ : \{}lf\texttt{\};} \\
\qquad \kappa \texttt{ : <a : } f_i\texttt{>>}]\!] \texttt{ . } P_\pi(ld') \texttt{ . F\_le(F\_OP(}g'\texttt{))>}
\end{array}
$$

which can be transformed to

$$
\begin{array}{l}
[\![\texttt{<}\iota \texttt{ : \{}lf\texttt{\};} \\
\ \kappa \texttt{ : <a : } f_i\texttt{>>}]\!] \texttt{ .} \\
\quad \texttt{<}\nu \texttt{ : } P_\nu(ld') \texttt{ . F\_ri(}g'\texttt{);} \\
\quad\ \pi \texttt{ : } \texttt{ . } P_\pi(ld') \texttt{ . F\_le(F\_OP(}g'\texttt{))>}
\end{array}
$$

which can be expressed as

$$[\![\texttt{<}\iota \texttt{ : \{}lf\texttt{\}; } \kappa \texttt{ : <a : } f_i\texttt{>> . F\_ri(}\mathbf{Z}\texttt{)}]\!]$$

Analogously we derive the *2-LCX-A* codomain

$$[\![\texttt{<}\iota \texttt{ : } f_i\texttt{; } \kappa \texttt{ : <a : \{}lf\texttt{\}>> . F\_ri(}\mathbf{Z}\texttt{)}]\!]$$

The above example hints at the following general fact.

Observation 7.3.3. *Let t_1 and t_2 be 2-LCX-A transformations with 2-LCX-A-expressible domains and codomains involving only adjunctions to X-covariant functors. Then the 2-LCX-A domain and codomain of $t_1 * t_2$ are expressible in the language of 2-LCX-A.*

As a corollary we get expressibility of all domain and codomain functors (and so of source and target categories, as well) in the internal core language of Dule, even when no restrictions are imposed on the use of horizontal composition involving parameterized exponents. Notice, however, that terms involving only adjoints to X-covariant functors can be presented as (often shorter) terms with adjoints to functors that are not X-covariant, as seen in the example on page 279. The adjunction factorizers, defined in the next section, often sport such not X-covariant arguments, even in the case of standard Dule adjunctions, such as parameterized exponents. However, as long as the operations can be expressed as other adjunction operations with only X-covariant arguments, as is the case for factorizers, domains and codomains of their horizontal compositions are expressible.

Equations

In the equations to follow `T_repsilon`(g) will be abbreviated to $\varepsilon_{\mathtt{r}}(g)$, `T_reta`$(g)$ to $\eta_{\mathtt{r}}(g)$, `T_lepsilon`(g) to $\varepsilon_{\mathtt{l}}(g)$ and `T_leta`(g) to $\eta_{\mathtt{l}}(g)$, as before. Here are the two main axioms of the theory of parameterized adjunctions Φ_A^X that we construct in this section.

$$\begin{aligned}
&(\texttt{<}\iota : \texttt{F\_ri}(g);\ \kappa : \texttt{F\_PR}(le,\kappa)\texttt{>} * \eta_{\mathtt{r}}(g)) \\
&\quad .\ (\texttt{<}\iota = \varepsilon_{\mathtt{r}}(g);\ \kappa = \texttt{T\_PR}(le,\kappa)\texttt{>} * \texttt{F\_ri}(g)) \;=\; \texttt{T\_id}(\texttt{F\_ri}(g)) \qquad (191)\\
&(\texttt{<}\iota = \eta_{\mathtt{r}}(g);\ \kappa = \texttt{T\_PR}(lc,\kappa)\texttt{>} * g) \\
&\quad .\ (\texttt{<}\iota : g;\ \kappa : \texttt{F\_PR}(lc,\kappa)\texttt{>} * \varepsilon_{\mathtt{r}}(g)) \;=\; \texttt{T\_id}(g) \qquad (192)
\end{aligned}$$

Below are the analogues of the previous equations, needed for the operations centered upon obtaining the left adjoint.

$$\begin{aligned}
&(\texttt{<}\iota : g;\ \kappa : \texttt{F\_PR}(lc,\kappa)\texttt{>} * \eta_{\mathtt{l}}(g)) \\
&\quad .\ (\texttt{<}\iota = \varepsilon_{\mathtt{l}}(g);\ \kappa = \texttt{T\_PR}(lc,\kappa)\texttt{>} * g) \;=\; \texttt{T\_id}(g) \qquad (193)\\
&(\texttt{<}\iota = \eta_{\mathtt{l}}(g);\ \kappa = \texttt{T\_PR}(lc,\kappa)\texttt{>} * \texttt{F\_le}(g)) \\
&\quad .\ (\texttt{<}\iota : \texttt{F\_le}(g);\ \kappa : \texttt{F\_PR}(lc,\kappa)\texttt{>} * \varepsilon_{\mathtt{l}}(g)) \;=\; \texttt{T\_id}(\texttt{F\_le}(g)) \qquad (194)
\end{aligned}$$

Now we would like to add equations expressing the parameterization of the adjunctions. Let $C(le)$ denote a family of `funct`-terms of the following form

$$\texttt{<}\iota : \texttt{F\_PR}(le,\iota);\ \kappa : \texttt{F\_PR}(le,\kappa)\ .\ \texttt{F\_CONTRA}(d)\texttt{>}$$

where the terms' targets $\texttt{<}\iota$ - e; κ - $d^{op}\texttt{>}$ are of the same form as the sources of the prototype operations `F_ori`(g) and `T_orepsilon`(g) from the beginning of Section 7.3. Notice that the `F_ri` and `T_repsilon` operations actually chosen for our semantics relate to the prototypes `F_ori` and `T_orepsilon` in the following way (as always, we assume the terms in equations are well typed and their domains and codomains are equal):

$$\begin{aligned}
\texttt{F\_ri}(g) &= C(le)\ .\ \texttt{F\_ori}(g)\\
\texttt{T\_repsilon}(g) &= C(le) * \texttt{T\_orepsilon}(g)
\end{aligned}$$

Let $lb = \iota$ - e; κ - b, $lb' = \iota$ - c; κ - b and let $f : b \to d$. Let us consider the term

$$\texttt{<}\iota : \texttt{F\_PR}(lb,\iota);\ \kappa : \texttt{F\_PR}(lb,\kappa)\ .\ f\texttt{>}\ .\ \texttt{F\_ri}(g)$$

and let us express the adjoint in the old syntax

$$\texttt{<}\iota : \texttt{F\_PR}(lb,\iota);\ \kappa : \texttt{F\_PR}(lb,\kappa)\ .\ f\texttt{>}\ .\ C(le)\ .\ \texttt{F\_ori}(g)$$

Then we can resolve the $C(le)$ abbreviation and simplify

$$\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f\ \texttt{.}\ \texttt{F\_CONTRA}(d)\texttt{>}\ \texttt{.}\ \texttt{F\_ori}(g)$$

and then use equation (179)

$$\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ \texttt{F\_CONTRA}(b)\ \texttt{.}\ \texttt{F\_OP}(f)\texttt{>}\ \texttt{.}\ \texttt{F\_ori}(g)$$

Finally, let us split the record, use equation (178) of Section 7.2.3 and return to the new syntax

$$\texttt{F\_ri(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)$$

We thus have the following equation.

$$\begin{aligned}&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g)\\ =\ &\texttt{F\_ri(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)\end{aligned}\tag{195}$$

This equation is textually very similar to equation (189) but in this context it is weaker, because the functor f must be a *2-LCX-A* morphism. Analogous equations for the other adjunction operations are needed, as well.

$$\begin{aligned}&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ \texttt{F\_le}(g)\\ =\ &\texttt{F\_le(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)\end{aligned}\tag{196}$$

$$\begin{aligned}&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f\texttt{>}\ *\ \texttt{T\_repsilon}(g)\\ =\ &\texttt{T\_repsilon(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)\end{aligned}\tag{197}$$

$$\begin{aligned}&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ *\ \texttt{T\_reta}(g)\\ =\ &\texttt{T\_reta(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)\end{aligned}\tag{198}$$

$$\begin{aligned}&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ *\ \texttt{T\_lepsilon}(g)\\ =\ &\texttt{T\_lepsilon(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)\end{aligned}\tag{199}$$

$$\begin{aligned}&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f\texttt{>}\ *\ \texttt{T\_leta}(g)\\ =\ &\texttt{T\_leta(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)\end{aligned}\tag{200}$$

We also need equations showing how parameterized adjoints act on morphisms. They are best expressed in the factorizer syntax as equations (219) and (220) in Section 7.3.3 below. Hereby we incorporate into our theory the two equations written in the *2-LCX-A* syntax (we do not quote them here, since they are much larger and less readable than their version using the factorizer notation below).

Finally we require that the choice of adjunction operations in opposite categories is symmetric, though we allow one (or both) of the counterparts to be undefined.

$$\begin{aligned}
\texttt{F\_ri}(g)^{op} &= \texttt{F\_le}(g^{op}) && (201)\\
\texttt{F\_le}(g)^{op} &= \texttt{F\_ri}(g^{op}) && (202)\\
\texttt{T\_repsilon}(g)^{op} &= \texttt{T\_leta}(g^{op}) && (203)\\
\texttt{T\_reta}(g)^{op} &= \texttt{T\_lepsilon}(g^{op}) && (204)\\
\texttt{T\_lepsilon}(g)^{op} &= \texttt{T\_reta}(g^{op}) && (205)\\
\texttt{T\_leta}(g)^{op} &= \texttt{T\_repsilon}(g^{op}) && (206)
\end{aligned}$$

Let Φ_A^X be the sum of the set of equations given above, theory Φ^X, a set of formulae stating that whenever one constructor of a given parameterized adjunction is defined then all three should be defined and the co-image under the translation of equations 189–190.

Definition 7.3.4. *An algebra satisfying Φ_A^X is called a 2-LCX-A.*

The above definition strictly defines both *2-LCX-A* and *2-LCO-A* classes of algebras. It may also be read as a proposition claiming that Φ_A^X captures well the informal description of *2-LCX-A* as the generalization of **Cat** to parameterized adjunctions (subsequently refined by hinting at the **Cat** models of *2-LCO-A*, defining the translation and discussing, with examples, the new *2-LCX-A* operations).

7.3.3 *2-LCX-F* — *2-LCX-A* by factorizers

2-LCX-F is a *2-LCX-A* with factorizers defined in terms of units and co-units in a similar way as in Section 7.1.2. Similarly as for non-parameterized adjunctions, the factorizers enable succinct notation and help to avoid the problematic operation of multiplication by a functor from the right.

Syntax

We enrich the syntax of *2-LCX-A* by the following terms, called the right and left factorizers.

```
type trans =
  ...
  | T_rfact of funct * funct * trans
  | T_lfact of funct * funct * trans
```

Semantics

Let everywhere below $la = \iota$ - a; κ - d, $lc = \iota$ - c; κ - d and $ld = \nu$ - d; π - d^{op}. The first attempt to define a right factorizer might have been the following, by analogy with the terms appearing in the axioms of Φ^X_A. The typing below is not in the language of *2-LCO-A*, but in the language of *2-LCX-F* (or equivalently, of *2-LCX-A*).

- `T_rfact`(g, f, t)
  could be the same transformation as
  (<ι : `F_PR`(la, ι) . f; κ : `F_PR`(la, κ)> $*$ `T_reta`(g))
  . (<ι = t; κ = `T_PR`(la, κ)> $*$ `F_ri`(g)),
  where g : <lc> $\rightarrow e$, $f : a \rightarrow c$, $h : a \rightarrow e$,
  t : <ι : `F_PR`(la, ι) . f; κ : `F_PR`(la, κ)> . $g \rightarrow$ `F_PR`(la, ι) . h
  and `T_rfact`(g, f, t) :
  `F_PR`(la, ι) . $f \rightarrow$ <ι : `F_PR`(la, ι) . h; κ : `F_PR`(la, κ)> . `F_ri`(g)

However, such definition is not very usable, since t is bound to have a specific source <la>, while the operation should accept transformations with arbitrary sources. We will (formally) define the partial operations of factorizers as follows.

- `T_rfact`(g, f, t)
  is the same transformation as
  (<ι : f; κ : `F_ID`(d)> $*$ `T_reta`(g))
  . (<ι = t; κ = `T_ID`(d)> $*$ `F_ri`(g)),
  where g : <lc> $\rightarrow e$, $f : d \rightarrow c$, $h : d \rightarrow e$,
  t : <ι : f; κ : `F_ID`(d)> . $g \rightarrow h$
  and `T_rfact`$(g, f, t) : f \rightarrow$ <ι : h; κ : `F_ID`(d)> . `F_ri`(g)

- `T_lfact`(g, h, t)
  is the same transformation as
  (<ι = t; κ = `T_ID`(d)> $*$ `F_le`(g))
  . (<ι : h; κ : `F_ID`(d)> $*$ `T_lepsilon`(g)),
  where g : <lc> $\rightarrow e$, $h : d \rightarrow c$, $f : d \rightarrow e$,
  $t : f \rightarrow$ <ι : h; κ : `F_ID`(d)> . g
  and `T_lfact`(g, h, t) : <ι : f; κ : `F_ID`(d)> . `F_le`$(g) \rightarrow h$

Equations

Just as for *2-LC-F* (see Section 7.1.2), the equational theory of *2-LCX-F* will not be based on the equational theory for *2-LCX-A*. Instead we will start with Φ^X together with the co-image of equations 189–190 and then add the new equations listed below.

The following four equations define adjunctions in terms of factorizers. The first two are for the right adjunctions and the next two are for the left adjunctions. Equations (208) and (209) will be called existence requirements, while equations (207) and (210) will be called uniqueness requirements.

$$\begin{aligned}
&\texttt{T\_rfact}(g, f, (\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g) \\
&\qquad \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\texttt{r}}(g))) \;=\; u \qquad (207)
\end{aligned}$$

$$\begin{aligned}
&(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_rfact}(g, f, t)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g) \\
&\qquad \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\texttt{r}}(g)) \;=\; t \qquad (208)
\end{aligned}$$

$$\begin{aligned}
&(\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \eta_{\texttt{l}}(g)) \\
&\texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_lfact}(g, h, t)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g) \;=\; t \qquad (209)
\end{aligned}$$

$$\begin{aligned}
&\texttt{T\_lfact}(g, h, (\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \eta_{\texttt{l}}(g)) \\
&\qquad \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)) \;=\; u \qquad (210)
\end{aligned}$$

Remark. As we required in Section 7.3.2, the functors used as operands of adjunction operations must be ι-covariant. Consequently, whenever the right hand side of any of the above equations has a type analogous to the types of the right hand sides of equations 149–152 of Section 7.1.2, there exists a type-correct instantiation of left hand side.

If arbitrary functors were allowed as operands of adjunction operations, there would usually be no type-correct instantiations of these equations because, intuitively, the vertical composition in the definition of factorizers would not be type-correct. For example, if functor g was F_CONTRA(<lc>) in equation (207) then the equation would not have any type-correct instantiation. This may be proved by showing that the *2-LCO-A* functor h, we would need, does not satisfy the conditions of Definition 7.2.4.

The T_reta and T_lepsilon term constructors are expressible in terms of factorizers as in Section 7.1.2.

$$\begin{aligned}
\texttt{T\_reta}(g) &= \texttt{T\_rfact}(\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(la, \iota)\texttt{;} \\
&\qquad\qquad \kappa\ \texttt{:}\ \texttt{F\_PR}(la, \kappa)\ \texttt{.}\ \texttt{F\_PR}(lc, \kappa)\texttt{>}\ \texttt{.}\ g, \\
&\qquad\qquad \texttt{F\_PR}(lc, \iota), \texttt{T\_id}(g)) \qquad (211) \\
\texttt{T\_lepsilon}(g) &= \texttt{T\_lfact}(\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(la, \iota)\texttt{;} \\
&\qquad\qquad \kappa\ \texttt{:}\ \texttt{F\_PR}(la, \kappa)\ \texttt{.}\ \texttt{F\_PR}(lc, \kappa)\texttt{>}\ \texttt{.}\ g, \\
&\qquad\qquad \texttt{F\_PR}(lc, \iota), \texttt{T\_id}(g)) \qquad (212)
\end{aligned}$$

We also need equations expressing parameterization of the adjunction operations.

$$\begin{aligned}
&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb, \iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb, \kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g) \\
&= \texttt{F\_ri}(\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb', \iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb', \kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g) \qquad (213)
\end{aligned}$$

$$\begin{aligned}
&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ \texttt{F\_le}(g)\\
&= \texttt{F\_le(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g) && (214)\\
&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f\texttt{>} * \texttt{T\_repsilon}(g)\\
&= \texttt{T\_repsilon(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g) && (215)\\
&\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f\texttt{>} * \texttt{T\_leta}(g)\\
&= \texttt{T\_leta(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g) && (216)\\
&h * \texttt{T\_rfact}(g,f,t)\\
&= \texttt{T\_rfact(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ h\texttt{>}\ \texttt{.}\ g,\\
&\qquad\qquad h\ \texttt{.}\ f, h * t) && (217)\\
&f * \texttt{T\_lfact}(g,h,t)\\
&= \texttt{T\_lfact(<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g,\\
&\qquad\qquad f\ \texttt{.}\ h, f * t) && (218)
\end{aligned}$$

In the following equations, showing how parameterization of adjoints acts on morphisms, let $H = \texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb',\kappa)\ \texttt{.}\ h'\texttt{>}\ \texttt{.}\ g$ and $F' = \texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb'',\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb'',\kappa)\ \texttt{.}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f'\texttt{>}\ \texttt{.}\ g$.

$$\begin{aligned}
&\texttt{<}\iota\ \texttt{=}\ \texttt{T\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_PR}(lb,\kappa) * t\texttt{>} * \texttt{F\_ri}(g)\ =\\
&\texttt{T\_rfact}(F',\texttt{F\_ri}(H),\\
&(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id(F\_ri}(H))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{F\_PR}(lb,\kappa) * t\texttt{>} * g)\ \texttt{.}\ \varepsilon_{\mathtt{r}}(H)) && (219)\\
&\texttt{<}\iota\ \texttt{=}\ \texttt{T\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_PR}(lb,\kappa) * t\texttt{>} * \texttt{F\_le}(g)\ =\\
&\texttt{T\_lfact}(F',\texttt{F\_le}(H),\\
&\eta_{\mathtt{l}}(H)\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id(F\_le}(H))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{F\_PR}(lb,\kappa) * t\texttt{>} * g)) && (220)
\end{aligned}$$

Finally, we require that the choice of adjunction operations in opposite categories is symmetric (however we allow one to be defined while the other is undefined).

$$\begin{aligned}
\texttt{F\_ri}(g)^{op} &= \texttt{F\_le}(g^{op}) && (221)\\
\texttt{F\_le}(g)^{op} &= \texttt{F\_ri}(g^{op}) && (222)\\
\texttt{T\_repsilon}(g)^{op} &= \texttt{T\_leta}(g^{op}) && (223)\\
\texttt{T\_leta}(g)^{op} &= \texttt{T\_repsilon}(g^{op}) && (224)\\
\texttt{T\_rfact}(g,f,t)^{op} &= \texttt{T\_lfact}(g^{op},f^{op},t^{op}) && (225)\\
\texttt{T\_lfact}(g,h,t)^{op} &= \texttt{T\_rfact}(g^{op},h^{op},t^{op}) && (226)
\end{aligned}$$

Note how right factorizer turns into a left factorizer and vice versa.

Let Φ_F^X be a sum of the set of equations given above, theory Φ^X, a set of formulae stating that whenever one constructor of a given parameterized adjunction is defined then all four should be defined and the co-image under the translation of equations 189–190. Then the following two theorems can be proved, as sketched in Appendix B, yielding the equivalence of Φ_F^X and the theory of *2-LCX-F*. To allow for syntactic proofs we use Definition 7.3.4 that equationally characterizes *2-LCX-A*, leading to an equation characterization of *2-LCX-F*.

Theorem 7.3.5. Φ_F^X *is contained in the theory of 2-LCX-F.*

Theorem 7.3.6. Φ_F^X *entails the theory of 2-LCX-F.*

7.3.4 Reduction in *2-LCX-F*

For the same reasons as discussed in Section 7.1.4, it is hard to construct a general reduction system for parameterized adjunctions. Yet for some particular sets of parameterized adjunctions satisfactory rewriting techniques exist. Here we will prove some consequences of Φ_F^X that we deem useful in reduction systems for many such settings.

Two halves of the uniqueness requirement

Similarly as in Section 7.1.4 the uniqueness requirements (207) and (210) can be split into two “halves”. Here are the “halves” with no apparent computational meaning.

$$\begin{aligned}&\texttt{T\_rfact}(g, f, (\texttt{<}\iota\ \texttt{:}\ h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\texttt{r}}(g))) \\ &\qquad = \texttt{T\_id}(\texttt{<}\iota\ \texttt{:}\ h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g))\end{aligned} \tag{227}$$

$$\begin{aligned}&\texttt{T\_lfact}(g, h, (\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \eta_{\texttt{l}}(g))) \\ &\qquad = \texttt{T\_id}(\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>}\ \texttt{.}\ \texttt{F\_le}(g))\end{aligned} \tag{228}$$

The other “halves”, to be used in rewriting (directed from right to left), are derived in the proof of the following theorem. As before, the two different weakenings of the uniqueness requirements (weak η-equations) induce incomparable theories, which together induce the whole Φ_F^X.

Theorem 7.3.7. *The following equations follow from theory* Φ_F^X.

$$\begin{aligned}&\texttt{T\_rfact}(g, f, (\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\ \texttt{.}\ t) \\ &\qquad\qquad = u\ \texttt{.}\ \texttt{T\_rfact}(g, h', t)\end{aligned} \tag{229}$$

$$\begin{aligned}&\texttt{T\_lfact}(g, h, t\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>}) * g) \\ &\qquad\qquad = \texttt{T\_lfact}(g, f', t)\ \texttt{.}\ u\end{aligned} \tag{230}$$

Proof. The proof is only slightly more complicated than the proof of analogous Theorem 7.1.8 for non-parameterized adjunctions.

We start by substituting the right hand sides of equations (229) and (230) for u into equations (207) and (210).

$$\begin{aligned}\texttt{T\_rfact}(g, f, (\texttt{<}\iota\ \texttt{=}\ u\ \texttt{.}\ \texttt{T\_rfact}(g, h', t)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\\ \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\texttt{r}}(g)))\\ = u\ \texttt{.}\ \texttt{T\_rfact}(g, h', t)\end{aligned}$$

$$\begin{aligned}\texttt{T\_lfact}(g, h, (\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \eta_{\texttt{l}}(g))\\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_lfact}(g, f', t)\ \texttt{.}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g))\\ = \texttt{T\_lfact}(g, f', t)\ \texttt{.}\ u\end{aligned}$$

Then we change record of compositions into composition of records.

$$\begin{aligned}\texttt{T\_rfact}(g, f, ((\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>}\\ \texttt{.}\ \texttt{<}\iota\ \texttt{=}\ \texttt{T\_rfact}(g, h', t)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>}) * g)\\ \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\texttt{r}}(g)))\\ = u\ \texttt{.}\ \texttt{T\_rfact}(g, h', t)\end{aligned}$$

$$\begin{aligned}\texttt{T\_lfact}(g, h, (\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \eta_{\texttt{l}}(g))\\ \texttt{.}\ ((\texttt{<}\iota\ \texttt{=}\ \texttt{T\_lfact}(g, f', t)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>}\\ \texttt{.}\ \texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>}) * g))\\ = \texttt{T\_lfact}(g, f', t)\ \texttt{.}\ u\end{aligned}$$

Then we distribute multiplication.

$$\begin{aligned}\texttt{T\_rfact}(g, f, (\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_rfact}g, h', t)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\\ \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\texttt{r}}(g))\\ = u\ \texttt{.}\ \texttt{T\_rfact}(g, h', t)\end{aligned}$$

$$\begin{aligned}\texttt{T\_lfact}(g, h, (\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \eta_{\texttt{l}}(g))\\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_lfact}(g, f', t)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>}) * g)\\ = \texttt{T\_lfact}(g, f', t)\ \texttt{.}\ u\end{aligned}$$

We conclude using the existence requirements, that is equations (208) and (209).

$$\begin{aligned}\texttt{T\_rfact}(g, f, (\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\ \texttt{.}\ t) &= u\ \texttt{.}\ \texttt{T\_rfact}(g, h', t)\\ \texttt{T\_lfact}(g, h, t\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>}) * g) &= \texttt{T\_lfact}(g, f', t)\ \texttt{.}\ u\end{aligned}$$

□

Action on morphisms

This is the only place where we have to insist that the operand functor of a partial adjunction operation be X-covariant not only on the ι component of the source but on the κ component as well. If the functor g in the equations below does not have this property then usually there will be no type-correct instantiations of the right hand side of these equations for a given left hand side and so the equations would be useless for rewriting. Fortunately all adjunctions used for the semantics of our core language, including the parameterized exponent of Section 8.1.1, are fully X-covariant in their basic (not instantiated) form.

This is also the place where the possibility of having functors of arbitrary variance at the κ component as operands for adjunction operations is essential for the usability of the result. We could restrict ourselves to adjoints to fully X-covariant functors but then for some sensible left hand sides of equation (231) there may be no corresponding instantiation of the right hand side. For example, if the adjoint at the left hand side was a parameterized exponent functor and the domain of t was again a parameterized exponent functor, then the functor F at the right hand side would not be fully X-covariant (as in the example on page 279) so the factorizer would be illegal.

The two equations presented in the following theorem can be used as reduction rules, when directed from left to right. The rules, coupled with a rewriting strategy that postpones instantiation of adjoint operands to maintain their X-covariance (using T_pp, T_ee, etc.), would allow to rewrite multiplications by all standard parameterized adjoint functors (though computing domains and codomains of such multiplication would be much more complicated than in a *2-LC-lc*). The form of the rules will also be inspiring in the design of rewrite rules for the T_map operation over not covariant functors, such as those involving exponents.

Theorem 7.3.8. *If the functor g is X-covariant then the following equations are consequences of Φ_F^X.*

$$\begin{aligned}
\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * \texttt{F\_ri}(g) \ &= \\
\texttt{T\_rfact}(F, \texttt{<}\iota : f\texttt{;}\ \kappa : h'\texttt{>}\ .\ \texttt{F\_ri}(g), & \\
(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(\texttt{<}\iota : f\texttt{;}\ \kappa : h'\texttt{>}\ .\ \texttt{F\_ri}(g))\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * g) & \\
.\ (\texttt{<}\iota : f\texttt{;}\ \kappa : h'\texttt{>} * \varepsilon_{\mathtt{r}}(g))\ .\ u) & \qquad (231)
\end{aligned}$$

$$\begin{aligned}
\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * \texttt{F\_le}(g) \ &= \\
\texttt{T\_lfact}(F, \texttt{<}\iota : f\texttt{;}\ \kappa : h'\texttt{>}\ .\ \texttt{F\_le}(g), & \\
u\ .\ (\texttt{<}\iota : f\texttt{;}\ \kappa : h'\texttt{>} * \eta_{\mathtt{l}}(g)) & \\
.\ (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(\texttt{<}\iota : f\texttt{;}\ \kappa : h'\texttt{>}\ .\ \texttt{F\_le}(g))\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * g)) & \qquad (232)
\end{aligned}$$

Proof. The structure of the proof corresponds to the proof of Theorem 7.1.13, but the derivation is much longer. Let $u : f \to h$ and $t : f' \to h'$. Let us begin with the left hand side of equation (231)

$$\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * \texttt{F\_ri}(g)$$

Let us complicate the expression

$$\texttt{<}\iota\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * \texttt{<}\iota\ \texttt{=}\ \texttt{T\_PR}(la,\iota) * u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_PR}(la,\kappa)\texttt{>} * \texttt{F\_ri}(g)$$

and transform it some more

$$\begin{aligned}&\texttt{<}\iota\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * \texttt{<}\iota\ \texttt{=}\ \texttt{T\_PR}(la,\iota) * u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(\texttt{<}la\texttt{>})\texttt{>} *\\ &\quad \texttt{<}\iota\ \texttt{=}\ \texttt{T\_PR}(le,\iota)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_PR}(le,\kappa) * \texttt{T\_PR}(la,\kappa)\texttt{>} * \texttt{F\_ri}(g)\end{aligned}$$

Then let's insert the functor inside the adjoint constructor

$$\begin{aligned}&\texttt{<}\iota\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * \texttt{<}\iota\ \texttt{=}\ \texttt{T\_PR}(la,\iota) * u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(\texttt{<}la\texttt{>})\texttt{>} *\\ &\quad \texttt{F\_ri}(\texttt{<}\iota : \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa : \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ \texttt{F\_PR}(la,\kappa)\texttt{>}\ \texttt{.}\ g)\end{aligned}$$

Now we declare a shorthand

$$G \quad = \quad \texttt{<}\iota : \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa : \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ \texttt{F\_PR}(la,\kappa)\texttt{>}\ \texttt{.}\ g$$

to be used in the term resulting from a rewrite by the uniqueness requirement (207)

$$\begin{aligned}&\texttt{<}\iota\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} *\\ &\texttt{T\_rfact}(G, \texttt{<}\iota : \texttt{F\_PR}(la,\iota)\ \texttt{.}\ f\texttt{;}\ \kappa : \texttt{F\_ID}(\texttt{<}la\texttt{>})\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(G),\\ &(\texttt{<}\iota\ \texttt{=}\ \texttt{<}\iota\ \texttt{=}\ \texttt{T\_PR}(la,\iota) * u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(\texttt{<}la\texttt{>})\texttt{>} * \texttt{F\_ri}(G)\texttt{;}\\ &\ \ \kappa\ \texttt{=}\ \texttt{T\_ID}(\texttt{<}la\texttt{>})\texttt{>} * G)\\ &\ \texttt{.}\ (\texttt{<}\iota : \texttt{F\_PR}(la,\iota)\ \texttt{.}\ h\texttt{;}\ \kappa : \texttt{F\_ID}(\texttt{<}la\texttt{>})\texttt{>} * \varepsilon_{\texttt{r}}(G)))\end{aligned}$$

Then we transform the result by, among others, associativity of horizontal composition

$$\begin{aligned}&\texttt{<}\iota\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} *\\ &\texttt{T\_rfact}(G, \texttt{<}\iota : \texttt{F\_PR}(la,\iota)\ \texttt{.}\ f\texttt{;}\ \kappa : \texttt{F\_ID}(\texttt{<}la\texttt{>})\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(G),\\ &(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_PR}(la,\iota) * u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(\texttt{<}la\texttt{>})\texttt{>}\\ &\ * (\texttt{<}\iota : \texttt{F\_ri}(G)\texttt{;}\ \kappa : \texttt{F\_PR}(la,\kappa)\texttt{>}\ \texttt{.}\ G)\\ &\ \texttt{.}\ (\texttt{<}\iota : \texttt{F\_PR}(la,\iota)\ \texttt{.}\ h\texttt{;}\ \kappa : \texttt{F\_ID}(\texttt{<}la\texttt{>})\texttt{>} * \varepsilon_{\texttt{r}}(G))))\end{aligned}$$

Now we simplify, using a variant of equation (188) applied at the vertical composition inside the factorizer (this is legal because the κ components are identities)

$$\begin{aligned}&\texttt{<}\iota\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} *\\ \texttt{T\_rfact}(G,\texttt{<}\iota\ \texttt{:}\ &\texttt{F\_PR}(la,\iota)\ \texttt{.}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID(<}la\texttt{>)>}\ \texttt{.}\ \texttt{F\_ri}(G),\\ &\texttt{<}\iota\ \texttt{=}\ \texttt{T\_PR}(la,\iota) * u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID(<}la\texttt{>)>} * \varepsilon_{\texttt{r}}(G))\end{aligned}$$

and using another variant of equation (188) (this time at the horizontal composition inside the factorizer)

$$\begin{aligned}&\texttt{<}\iota\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} *\\ \texttt{T\_rfact}(G,\texttt{<}\iota\ \texttt{:}\ &\texttt{F\_PR}(la,\iota)\ \texttt{.}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID(<}la\texttt{>)>}\ \texttt{.}\ \texttt{F\_ri}(G),\\ &(\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(la,\iota)\ \texttt{.}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID(<}la\texttt{>)>} * \varepsilon_{\texttt{r}}(G))\\ &\texttt{.}\ (\texttt{F\_PR}(la,\iota) * u))\end{aligned}$$

Then we transform by literally equation (188) used at the first horizontal composition, which second operand is not κ-contravariant, but its domain and codomain is, because the functor g is X-covariant

$$\begin{aligned}&(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} *\\ &(\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(la,\iota)\ \texttt{.}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID(<}la\texttt{>)>}\ \texttt{.}\ \texttt{F\_ri}(G)))\\ &\texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{;}\ \kappa\ \texttt{:}\ f'\texttt{>} *\\ \texttt{T\_rfact}(G,\texttt{<}\iota\ \texttt{:}\ &\texttt{F\_PR}(la,\iota)\ \texttt{.}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID(<}la\texttt{>)>}\ \texttt{.}\ \texttt{F\_ri}(G),\\ &(\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(la,\iota)\ \texttt{.}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID(<}la\texttt{>)>} * \varepsilon_{\texttt{r}}(G))\\ &\texttt{.}\ (\texttt{F\_PR}(la,\iota) * u)))\end{aligned}$$

We declare another abbreviation (for readability)

$$F \quad = \quad \texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ f'\texttt{>}\ \texttt{.}\ g$$

that will appear inside the adjunction constructors in the simplified term below (notice that F may be not fully X-covariant, but we have deliberately allowed this)

$$\begin{aligned}&(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(f)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{<}\iota\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>>}\\ &* \texttt{<}\iota\ \texttt{=}\ \texttt{T\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_PR}(lb,\kappa) * \texttt{T\_PR}(la,\kappa)\texttt{>} * \texttt{F\_ri}(g))\\ &\texttt{.}\ \texttt{T\_rfact}(F,\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{<}\iota\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{;}\ \kappa\ \texttt{:}\ f'\texttt{>>}\ \texttt{.}\ \texttt{F\_ri}(G),\\ &(\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{<}\iota\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{;}\ \kappa\ \texttt{:}\ f'\texttt{>>} * \varepsilon_{\texttt{r}}(G))\ \texttt{.}\ u)\end{aligned}$$

Then we simplify more

$$\begin{aligned}&(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(f)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * \texttt{F\_ri}(g))\\ &\texttt{.}\ \texttt{T\_rfact}(F,\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(F),\\ &(\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(F))\ \texttt{.}\ u)\end{aligned}$$

and use weak η-equation (229) for parameterized adjunctions

$$\begin{array}{l}
\texttt{T\_rfact}(F,\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ h'\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g),\\
(\texttt{<}\iota\ \texttt{=}\ \texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(f)\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * \texttt{F\_ri}(g)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{>} * F)\\
\ \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(F))\ \texttt{.}\ u)
\end{array}$$

Then we complicate

$$\begin{array}{l}
\texttt{T\_rfact}(F,\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ h'\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g),\\
(\texttt{<}\iota\ \texttt{=}\ \texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{>} *\\
\quad \texttt{<}\iota\ \texttt{=}\ \texttt{T\_PR}(lb,\iota)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_PR}(lb,\kappa) * t\texttt{>} * \texttt{F\_ri}(g)\texttt{;}\\
\ \ \kappa\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{>} * F)\\
\ \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(F))\ \texttt{.}\ u)
\end{array}$$

and use equation (219)

$$\begin{array}{l}
\texttt{T\_rfact}(F,\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ h'\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g),\\
(\texttt{<}\iota\ \texttt{=}\ \texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{>} *\\
\quad \texttt{T\_rfact}(F',\texttt{F\_ri}(H),\\
\quad (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(\texttt{F\_ri}(H))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{F\_PR}(lb,\kappa) * t\texttt{>} * g)\ \texttt{.}\ \varepsilon_{\texttt{r}}(H))\texttt{;}\\
\ \ \kappa\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{>} * F)\\
\ \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(F))\ \texttt{.}\ u)
\end{array}$$

Then we simplify

$$\begin{array}{l}
\texttt{T\_rfact}(F,\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ h'\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g),\\
(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_rfact}(F,\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ h'\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g),\\
(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ h'\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g))\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * g)\\
\ \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ h'\texttt{>} * \varepsilon_{\texttt{r}}(g)))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{>} * F)\\
\ \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(F))\ \texttt{.}\ u)
\end{array}$$

and using the existence requirement (208), at last we get the right hand side

$$\begin{array}{c}
\texttt{T\_rfact}(F,\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ h'\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g),\\
(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ h'\texttt{>}\ \texttt{.}\ \texttt{F\_ri}(g))\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * g)\\
\texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ h'\texttt{>} * \varepsilon_{\texttt{r}}(g))\ \texttt{.}\ u)
\end{array}$$

Equation (232) is derived in an analogous way. □

7.3.5 Conclusion

In this very technical section we have developed the language, theory and reduction system analogous to that of Section 7.1 but for adjoints that may be instantiated after they are created. The adjoints will not be a part of the user language, but they are used in the next section as a basis for extending the core language with function types and will be useful for future extensions such as mapping with not covariant types, coexponent types (exceptions/continuations), etc. We use the languages of *2-LCO* and *2-LCX* to describe domains and codomain of the introduced adjunction operations. An extension of the translation from *2-LCX* to *2-LCO* serves to describe semantics and equational theory of the adjunctions parameterized by types.

Just as in Section 7.1, after introducing units and co-units we extend the language with factorizers and provide an equivalent alternative equational theory based on them (Theorem 7.3.5, Theorem 7.3.6), proposing reduction rules for multiplication by adjoint functors (Theorem 7.3.8) and η-equivalence (Theorem 7.3.7). The weak η-rules turn out to be moderate generalizations of their counterparts for non-parameterized adjunctions. The rules for multiplication are considerably more complicated, just as the parts of equational theory that are used in their proof. The complexity is similar to that of equations superseding interchange law for morphisms of complicated variance in *2-LCX*.

The rule for multiplication by adjoint functors justifies the generality of parameterized adjunctions. We notice that even if the multiplication at the left hand side is by an adjoint to a fully X-covariant functor, the factorizer at the right hand side of the rule may refer to an adjoint to a not X-covariant functor. Therefore, if the definition of adjunction was less general and functor to which one adjoins was always required to be fully X-covariant, the rule would be invalid.

Chapter 8

Extensions to the core language

Extensions to the core language described here, while semantically important, useful in practice and included in the full user language of Dule, are deemed not to be the fundamental mechanisms of Dule nor its distinctive features. For this reason, although many of these topics are interesting on their own, they will have to wait for a separate deeper analysis, while in the thesis we treat them merely as tools to facilitate experiments with Dule.

8.1 e-Core — core language with parameterized exponent

8.1.1 Definition based on i-Core and *2-LCX-F*

Having incorporated adjunction parameterized by types into our framework we may return to the task of defining the labeled exponent functor parameterized on both the covariant and contravariant positions. The functor can be used to model function types of a programming language; in particular, the function types in module specifications, which can be arbitrarily parameterized. We base our definition on i-Core (described in Section 6.3, although p-Core of Section 6.1 would suffice) to maintain continuity of the presentation of the internal language.

On the other hand, we refrain from offering the possibilities of full interaction between function types and inductive types; in particular, we do not complete the rewriting of `T_map` constructor for the case of exponent functor. This means that for all practical purposes the user should avoid inductive types with "recursion" going through the exponent construction. The interaction between exponent and inductive constructions will be an interesting topic for further research, see Section 10.1. Our framework makes it possible to locate the exact place where the problems with (co)inductive operations acting over function types emerge. For future work, different sets of rules for `T_map` will allow one to tackle the problem in different ways. At present, for simplicity, we consider undefined any application of `T_map` (and any other (co)algebra-related term constructor) to the offending functors. Fortunately inductive types with the induction parameter nested inside a function type are not very common in programming practice.

In short, a parameterized labeled exponent is an extension of the concept of labeled exponent introduced in Section 7.1.3. Instead of using the normal adjunctions, we employ the machinery of parameterized adjunctions and we consider the adjoint to the functor $\mathbf{Z}(lb)$; the functor is already defined in Section 7.2.2, but we will recall the definition below. By *2-LCX-F* with sums, (co)algebras and exponents we understand a *2-LCX-F* with parameterized exponent operations, as defined below, and with products, sums and (co)algebras — but with (co)algebra operations defined only for functors where the induction parameter is not nested inside the exponent functor — satisfying properties analogous to those for *2-LC-lc* with sums and (co)algebras. The adjunction operations visible in the signature (as defined in the **Syntax** subsection below) are only those corresponding to the product, coproduct and exponent adjunctions, so only those and no other adjunctions need to be defined in the *2-LCX-F*.

8.1.2 Language

Syntax

We would like the parameterized labeled exponent operations to have the following syntax. First, we extend the syntax of i-Core by the following `funct`-term constructor.

```
type funct =
  ...
  | F_ee of funct IList.t * funct
```

Then we append three `trans`-term constructors to the list of `trans`-term constructors of i-Core, obtaining the syntax of the core programming language e-Core.

```
type trans =
  ...
  | T_ee of trans IList.t * trans
  | T_appl of funct IList.t * funct
  | T_curry of trans
```

Overview of the semantics

Let us fix a *2-LCX-F* with sums, (co)algebras and exponents. We give the semantics of the language e-Core in the fixed category. The parameterized labeled exponent functor will be used to model labeled function types of the Dule programming language. The functor differs from the standard (non-labeled) exponent in that the parameters are multiple and are labeled by i_1, ..., i_n, and that, during application, the function is placed at the component labeled υ. In the overview of the exponent operations we will not focus on the meaning of parameterization. It will be explained after we deploy the adjunction machinery to strictly define the exponent operations and derive an equational description of their properties — parameterizability in particular.

The semantics of the term constructors inherited from i-Core is analogous as in *2-LC-lc* with sums and (co)algebras. The overall semantics of the new term constructors of e-Core in the fixed *2-LCX-F* with sums, (co)algebras and exponents should look as described below. As usual, the typings are a bit too general. We will later restrict ourselves to term-constructors where targets e are the base category `*`.

- $\texttt{F\_ee}(lg, h) : c \to e$
  is a parameterized labeled exponent object (that is functor) in $C(c, e)$, where $g_i : c \to e$ and $h : c \to e$,
- $\texttt{T\_ee}(lt, u) : \texttt{F\_ee}(lh, f) \to \texttt{F\_ee}(lf, h)$
  is the action of the parameterized labeled exponent functor on morphisms of $C(c, e)$ where lt is an indexed list of transformations, such that $t_i : f_i \to h_i$, $f_i, h_i : c \to e$, and $u : f \to h$ is a transformation such that $f, h : c \to e$,
- $\texttt{T\_appl}(lg, h) : \texttt{\{}\upsilon\ \texttt{:}\ \texttt{F\_ee}(lg, h)\texttt{;}\ i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots\texttt{\}} \to h$
  is the parameterized evaluation (application) morphism in $C(c, e)$, where $g_k : c \to e$ and $h : c \to e$,
- $\texttt{T\_curry}(t) : f \to \texttt{F\_ee}(lg, h)$
  is the parameterized curryfication of the morphism (that is transformation) t in $C(c, e)$, where $t : \texttt{\{}\upsilon\ \texttt{:}\ f\texttt{;}\ i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots\texttt{\}} \to h$ and $f, g_k, h : c \to e$.

8.1.3 Construction using *2-LCX-F*

Now we will construct the parameterized labeled exponent operations, theory and rewriting, using the formalism of parameterized adjunctions of Section 7.3. This will assign a precise meaning to the requirement that our exponent be parameterized and demonstrate its computational meaning.

Semantics

Let $lb = i_1\ \texttt{-}\ \texttt{*}\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{-}\ \texttt{*}$ and let υ be a distinguished label different from $i_1, \ldots, i_n$. Let $le = \iota\ \texttt{-}\ \texttt{*}\texttt{;}\ \kappa\ \texttt{-}\ \texttt{<}lb\texttt{>}$. Let

$$\mathbf{Z}(lb) \;=\; \texttt{\{}\upsilon\ \texttt{:}\ \texttt{F\_PR}(le, \iota)\texttt{;}\ i_1\ \texttt{:}\ \texttt{F\_PR}(le, \kappa)\ \texttt{.}\ \texttt{F\_PR}(lb, i_1)\texttt{;}\ \ldots\texttt{\}}$$

be the functor, from which we will make the exponent by taking the parameterized right adjoint, similarly as in Section 7.2.2. By definition, all adjoints of the form $\texttt{F\_ri}(\mathbf{Z}(lb))$ are defined in the fixed *2-LCX-F* with sums, (co)algebras and exponents. We see that

$$\mathbf{Z}(lb) \;:\; \texttt{<}\iota\ \texttt{-}\ \texttt{*;}\ \kappa\ \texttt{-}\ \texttt{<}lb\texttt{>>} \to \texttt{*}$$

and moreover

$$\texttt{F\_ri}(\mathbf{Z}(lb)) \;:\; \texttt{<}\iota\ \texttt{-}\ \texttt{*;}\ \kappa\ \texttt{-}\ \texttt{<}lb\texttt{>>} \to \texttt{*}$$

which shows no signs of contravariance, yet. The fact that all component categories in the source and target are C_BB and none is dual to C_BB may be surprising, because an exponent should be contravariant, as discussed in Section 7.2.2. This will be explained later in this section.

We can now define the labeled exponent operations. Let a be a category and $f, h, g_i, f_i, h_i : a \to \texttt{*}$. Let $t_i : f_i \to h_i$ and $u : f \to h$. Let

$$t \;:\; \texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{<}lg\texttt{>>}\ \texttt{.}\ \mathbf{Z}(lb) \to h$$

in other words

$$t \;:\; \texttt{\{}\upsilon\ \texttt{:}\ f\texttt{;}\ i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots\texttt{\}} \to h$$

We define the exponent notation as follows.

$$\begin{aligned}
\texttt{F\_ee}(lg, h) &= \texttt{<}\iota\ \texttt{:}\ h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{<}lg\texttt{>>}\ \texttt{.}\ \texttt{F\_ri}(\mathbf{Z}(lb))\\
\texttt{T\_ee}(lt, u) &= \texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{<}lt\texttt{>>} * \texttt{F\_ri}(\mathbf{Z}(lb))\\
\texttt{T\_appl}(lg, h) &= \texttt{<}\iota\ \texttt{:}\ h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{<}lg\texttt{>>} * \texttt{T\_repsilon}(\mathbf{Z}(lb))\\
\texttt{T\_curry}(t) &= \texttt{T\_rfact(\{}\upsilon\ \texttt{=}\ \texttt{T\_PR}(lb,\iota)\texttt{;}\ i_1\ \texttt{=}\ \texttt{T\_PR}(lb,\kappa) * g_1\texttt{;}\ \ldots\texttt{\}}, f, t)
\end{aligned}$$

Note that the *2-LCX-F* typing of the $\texttt{T\_ee}(lt, u)$ transformation may be informally written as follows, which appears different from the typing given in the overview of the semantics.

$$\begin{aligned}
\texttt{T\_ee}(lt, u) \;:\; &\texttt{<}\iota\ \texttt{:}\ f_h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{<}lf_h\texttt{>>}\ \texttt{.}\ \texttt{F\_ri}(\mathbf{Z}(lb)) \to\\
&\texttt{<}\iota\ \texttt{:}\ h_f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{<}lh_f\texttt{>>}\ \texttt{.}\ \texttt{F\_ri}(\mathbf{Z}(lb))
\end{aligned}$$

where the functors f_h, h_f, $\texttt{<}lf_h\texttt{>}$ and $\texttt{<}lh_f\texttt{>}$ are *2-LCO-A* domains and codomains of u and $\texttt{<}lt\texttt{>}$, respectively, and are usually not *2-LCX-F* functors. For example, writing in the syntax of *2-LCO-A*:

$$[\![f_h]\!] \;=\; \texttt{<}\nu\ \texttt{:}\ f\ \texttt{.}\ \texttt{F\_PR}(lc,\nu)\texttt{;}\ \pi\ \texttt{:}\ h\ \texttt{.}\ \texttt{F\_PR}(lc,\pi)\texttt{>}$$

which is not a *2-LCX-F* functor (note the functor h at label π) unless f and h are equal (similarly for h_f as well as for lf_h and lh_f).

However, because adjoints are ι-covariant, and in this particular adjunction the adjoint is also κ-contravariant, the offending components of the records may be changed to the matching counterparts, without changing the overall semantics of the composition with the adjoint (as explained in more detail for a similar case in the example on page 281). For example, the π component of the $[\![f_h]\!]$ functor may be replaced by f . `F_PR`(lc, π) thus making the resulting *2-LCO-A* functor equal to $[\![f]\!]$. This leads us to the following *2-LCX-F* typing

$$\begin{aligned}\texttt{T\_ee}(lt, u) \;:\; & \texttt{<}\iota : f\texttt{;}\ \kappa : \texttt{<}lh\texttt{>>}\ .\ \texttt{F\_ri}(\mathbf{Z}(lb)) \to \\ & \texttt{<}\iota : h\texttt{;}\ \kappa : \texttt{<}lf\texttt{>>}\ .\ \texttt{F\_ri}(\mathbf{Z}(lb))\end{aligned}$$

that written in the exponent syntax looks precisely as given in the overview of the semantics above:

$$\texttt{T\_ee}(lt, u) \;:\; \texttt{F\_ee}(lh, f) \to \texttt{F\_ee}(lf, h)$$

Here the contravariance (precisely, X-contravariance on the κ component) of exponentiation finally shows up; it becomes visible in the exchange of lf and lh as the first operands of the `F_ee` operations (with respect to the typing of the transformations on the lt list). The formalism of *2-LCX-F* thus facilitates capturing the essence of exponent contravariance without introducing opposite categories or explicit `F_CONTRA` morphisms.

Equations

As the above construction of exponent operations implies, an accurate theory of parameterized labeled exponents is contained in an instance of Φ_F^X for parameterized adjunctions with factorizers (the instance is obtained by adding definedness predicates for all exponent, product and sum adjunctions). Here we will try to recast the main axioms of the theory in the language of e-Core. Being derived from Φ_F^X, the equations will be guaranteed to be sound with respect to the semantics of the exponent operations.

The following `funct`-equation expresses the full parameterizability of the exponent functor (compare with the partially parameterized exponent of Section 7.2)

$$f\ .\ \texttt{F\_ee}(i_1 : g_1\texttt{;}\ \ldots, h) \;=\; \texttt{F\_ee}(i_1 : f\ .\ g_1\texttt{;}\ \ldots, f\ .\ h) \qquad (233)$$

A similar `trans`-equation holds for the evaluation morphism.

$$f * \texttt{T\_appl}(i_1 : g_1\texttt{;}\ \ldots, h) \;=\; \texttt{T\_appl}(i_1 : f\ .\ g_1\texttt{;}\ \ldots, f\ .\ h) \qquad (234)$$

Next we consider the trans-equation defining the T_ee operation as the action of the parameterized exponent on morphisms.

$$\texttt{T\_ee}(lt, u) \;=\; \texttt{<}\iota\texttt{ = }u\texttt{; }\kappa\texttt{ = <}lt\texttt{>>} * \texttt{F\_ee}(i_1 \texttt{ : } \kappa \texttt{ . } i_1\texttt{; } \ldots, \iota) \tag{235}$$

Because we want to ignore variance of functors in our programming language, we derive an equation describing the action of the κ-contravariant labeled exponent functor on transformations in terms of other operations. The operation T_ee represents multiplication by the exponent functor from the right, at the same time preventing the instantiation of the exponent functor, so that the functor stays ι-covariant. This is important for the applicability of the rewrite rule based on equation (231) of Section 7.3.4. We have defined

$$\texttt{T\_ee}(lt, u) \;=\; \texttt{<}\iota\texttt{ = }u\texttt{; }\kappa\texttt{ = <}lt\texttt{>>} * \texttt{F\_ri}(\mathbf{Z}(lb))$$

Specializing the action equation (231) we get

$$\begin{aligned}\texttt{T\_ee}(lt, u) \;=&\\ \texttt{T\_rfact}(F, \texttt{<}\iota\texttt{ : }f\texttt{; }\kappa\texttt{ : <}lh'\texttt{>> . F\_ri}(g),&\\ (\texttt{<}\iota\texttt{ = T\_id(<}\iota\texttt{ : }f\texttt{; }\kappa\texttt{ : <}lh'\texttt{>> . F\_ri}(g))\texttt{; }\kappa\texttt{ = <}lt\texttt{>>} * g)&\\ \texttt{. (<}\iota\texttt{ : }f\texttt{; }\kappa\texttt{ : <}lh'\texttt{>>} * \varepsilon_{\mathrm{r}}(g)) \texttt{ . } u)&\end{aligned}$$

and using the exponent notation we obtain

$$\begin{aligned}\texttt{T\_ee}(lt, u) \;=&\\ \texttt{T\_curry(\{}\upsilon\texttt{ = T\_pr}(lf, \upsilon)\texttt{; }i\texttt{ = T\_pr}(lf, i)\texttt{ . }t_i\texttt{; }\ldots\texttt{\}}&\\ \texttt{. T\_appl}(lh', f)\texttt{ . }u)&\end{aligned}$$

which we write using syntactic sugar as follows.

$$\begin{aligned}\texttt{T\_ee}(lt, u) \;=&\\ \texttt{T\_curry(\{}\upsilon\texttt{ = }\upsilon\texttt{; }i\texttt{ = }i\texttt{ . }t_i\texttt{; }\ldots\texttt{\} . T\_appl}(lh', f)\texttt{ . }u)&\end{aligned} \tag{236}$$

The equation we have derived expresses the action of the exponent functor over transformations without multiplying by the exponent functor (or using horizontal composition with transformations having exponent functor in their typing, which would introduce variance-related complications just as well).

We will now specialize the uniqueness requirement (207) and existence requirement (208) of Section 7.3.3. Let additionally $f' : a \to \texttt{*}$, $la = \iota \texttt{ - *; } \kappa \texttt{ - } a$ and $u : f \to \texttt{F\_ee}(lg, f')$. Let functor g be

$$\texttt{<}\iota\texttt{ : F\_PR}(lb, \iota)\texttt{; }\kappa\texttt{ : F\_PR}(lb, \kappa)\texttt{ . <}lg\texttt{>> . }\mathbf{Z}(lb)$$

which is equal to

$$\{\upsilon\ \texttt{:}\ \texttt{F\_PR}(lb,\iota)\texttt{;}\ i_1\ \texttt{:}\ \texttt{F\_PR}(lb,\kappa)\ \texttt{.}\ g_1\texttt{;}\ \ldots\}$$

We use this notation in the equations below. Here are the uniqueness requirement (207) and the existence requirement (208).

$$\begin{aligned}
&\texttt{T\_rfact}(g, f, (\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{>} * g) \\
&\qquad \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ f'\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(g))) \quad = \quad u \\
&(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_rfact}(g, f, t)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{>} * g) \\
&\qquad \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(g)) \quad = \quad t
\end{aligned}$$

We use equation (215) to simplify the equations

$$\begin{aligned}
&\texttt{T\_rfact}(g, f, (\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{>} * g) \\
&\qquad \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ f'\texttt{;}\ \kappa\ \texttt{:}\ \texttt{<}lg\texttt{>>} * \varepsilon_{\texttt{r}}(\mathbf{Z}(lb)))) \quad = \quad u \\
&(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_rfact}(g, f, t)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{>} * g) \\
&\qquad \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ h\texttt{;}\ \kappa\ \texttt{:}\ \texttt{<}lg\texttt{>>} * \varepsilon_{\texttt{r}}(\mathbf{Z}(lb))) \quad = \quad t
\end{aligned}$$

then we change notation introducing `T_curry` and `T_appl`

$$\begin{aligned}
&\texttt{T\_curry}((\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{>} * g) \\
&\qquad \texttt{.}\ \texttt{T\_appl}(lg, f')) \quad = \quad u \\
&(\texttt{<}\iota\ \texttt{=}\ \texttt{T\_curry}(t)\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(a)\texttt{>} * g) \\
&\qquad \texttt{.}\ \texttt{T\_appl}(lg, h) \quad = \quad t
\end{aligned}$$

and we use the definition of functor g

$$\texttt{T\_curry}(\texttt{T\_pp}(a,\ \upsilon\ \texttt{=}\ u\texttt{;}\ i_1\ \texttt{=}\ \texttt{T\_id}(g_1)\texttt{;}\ \ldots)\ \texttt{.}\ \texttt{T\_appl}(lg, f')) \quad = \quad u \tag{237}$$

$$\texttt{T\_pp}(a,\ \upsilon\ \texttt{=}\ \texttt{T\_curry}(t)\texttt{;}\ i_1\ \texttt{=}\ \texttt{T\_id}(g_1)\texttt{;}\ \ldots)\ \texttt{.}\ \texttt{T\_appl}(lg, h) \quad = \quad t \tag{238}$$

The uniqueness (237) and the existence (238) requirements for labeled exponents thus obtained are almost identical to those of Section 7.1.3.

Reduction

We need a reduction system for the parameterized labeled exponent that is free from the adjunction syntax. Nevertheless, we will start with a reference to rewriting in *2-LCX-F*, by specializing the weak η-equation (229) of Section 7.3.4.

$$\texttt{T\_rfact}(g, f, (\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\ \texttt{.}\ t) \quad = \quad u\ \texttt{.}\ \texttt{T\_rfact}(g, h', t)$$

Instantiating and changing notation we transform it to

$$\texttt{T\_curry}((\texttt{<}\iota\ \texttt{=}\ u\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * \mathbf{Z}(lb))\ \texttt{.}\ t) \quad = \quad u\ \texttt{.}\ \texttt{T\_curry}(t)$$

and using the definition of $\mathbf{Z}(lb)$ we obtain

$$\begin{aligned}\texttt{T\_curry}(\texttt{\{}\upsilon\ \texttt{=}\ \texttt{T\_pr}(lf, \upsilon)\ \texttt{.}\ u\texttt{;}\ i_1\ \texttt{=}\ \texttt{T\_pr}(lf, i_1)\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ t)\\ = \ u\ \texttt{.}\ \texttt{T\_curry}(t)\end{aligned}$$

Directed from right to left this equation becomes the weak η-rule, which turns out to be exactly the same as the corresponding rule of Section 7.1.3.

$$u\ \texttt{.}\ \texttt{T\_curry}(t) \quad \to \quad \texttt{T\_curry}(\texttt{\{}\upsilon\ \texttt{=}\ \upsilon\ \texttt{.}\ u\texttt{;}\ i_1\ \texttt{=}\ i_1\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ t) \qquad (239)$$

The rule for instantiating exponent functors is almost the same as for non-parameterized exponents, but this time the argument types can be parameterized.

$$f\ \texttt{.}\ \texttt{F\_ee}(i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots, h) \quad \to \quad \texttt{F\_ee}(i_1\ \texttt{:}\ f\ \texttt{.}\ g_1\texttt{;}\ \ldots, f\ \texttt{.}\ h) \qquad (240)$$

The rule for `T_ee` is quite different than the corresponding rule of Section 7.1.3 unless the transformations lt are identities, in which case the rules are essentially the same.

$$\begin{aligned}\texttt{T\_ee}(lt, u) \ \to \\ \texttt{T\_curry}(\texttt{\{}\upsilon\ \texttt{=}\ \upsilon\texttt{;}\ i\ \texttt{=}\ i\ \texttt{.}\ t_i\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ \texttt{T\_appl}(lh', f)\ \texttt{.}\ u)\end{aligned} \qquad (241)$$

The β-rule, obtained by weakening the existence requirement (238), looks as follows

$$\begin{aligned}&\texttt{\{}\upsilon\ \texttt{=}\ \texttt{T\_curry}(t)\texttt{;}\ i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ \texttt{T\_appl}(lg, h)\\ &\to\ \texttt{\{}\upsilon\ \texttt{=}\ \texttt{T\_id}(f)\texttt{;}\ i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ t\end{aligned} \qquad (242)$$

and turns out to be the same as in Section 7.1.3.

All the other rules are changed only to reflect that both the argument and result of exponent can be instantiated.

$$\begin{aligned} f * \texttt{T\_appl}(i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots, h) &\to \texttt{T\_appl}(i_1\ \texttt{:}\ f\ \texttt{.}\ g_1\texttt{;}\ \ldots, f\ \texttt{.}\ h) && (243)\\ f * \texttt{T\_curry}(u) &\to \texttt{T\_curry}(f * u) && (244)\\ t * \texttt{F\_ee}(i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots, h) &\to \texttt{T\_ee}(i_1\ \texttt{=}\ t * g_1\texttt{;}\ \ldots, t * h) && (245)\end{aligned}$$

Observation 8.1.1. *Rules 239–245 are sound with respect to the theory generated by equations 233–238 and the theory Φ_p of 2-LC-lc.*

Let the reduction system $\mathcal{R}_e$ contain rules 239–245 together with all the basic rules of $\mathcal{R}_i$, except rule (29) for horizontal composition. All the basic rules of $\mathcal{R}_e$ (plus several ones pertaining to fixpoints, to be described in the next section) are listed in Appendix A.1.4. Due to the lack of a rule for mapping with exponent functor, the rewrite of (co)algebra-related constructors is not complete, but not harmful for the properties of $\mathcal{R}_e$, either.

Observation 8.1.2. *Reduction system $\mathcal{R}_e$ is locally confluent.*

Just as for $\mathcal{R}_i$ we believe $\mathcal{R}_e$ to be strongly normalizing, but we leave the proof for future work. Strong normalization and local confluence imply confluence by the Newman lemma.

Conjecture 8.1.3. *Reduction system $\mathcal{R}_e$ is strongly normalizing and confluent.*

Notice that, as always, we have been cautious with rules descending from the uniqueness requirement. We not only avoided a full η-expansion and η-contraction, but also didn't include other interesting rules that emerge here. For example, one could add a rule that reduces an application of function arguments to a case expression by moving the application into each branch of the case expression [58].

8.1.4 Moving back to *2-LC-lc*

The language of *2-LCX-F* could be, in theory, used for programming, assuming the existence of certain adjunctions. If we build transformations only with *2-LCX-F* units and co-units, the resulting terms are unnecessarily large and they contain numerous multiplications by a functor from the right. From the perspective of a programming language user, the multiplications from the right are exotic, rarely useful and potentially dangerous. They can change the target category of a value (a transformation with target `*`) thus moving it beyond the realm of programming language values. However, the language of adjunctions also contains the operation of a factorizer, which enables succinct notation and eliminates most multiplications by a functor from the right.

We have succeeded in constructing the parameterized exponent using the formalism of *2-LCX-F*. Our derived syntax for function type operations corresponds to the factorizer presentation of the exponent adjunction operations. This and the availability of operation `T_ee` allows the programmer to use function types without employing multiplications by a functor from the right, as well as any horizontal compositions with non-identity left hand sides.

The axioms and rules of *2-LCX-F* not concerning adjunctions are the same as in *2-LC*, with the exception of the interchange law (20) and some of its consequences, in particular rule (29) describing rewriting of horizontal composition.

Moreover, the typing of transformations and functors is the same as in *2-LC* with the exception of terms with horizontal composition or multiplication by a functor from the right, which can have much more complicated typing, as exemplified on page 281. Since we can easily program without horizontal composition and multiplication by a functor from the right in our programming language, these discrepancies are not problematic.

The rules and equations for the concrete voidly parameterized adjunctions described in Section 7.1.3, for example for the product adjunction, when expressed in the specialized syntax, are the same as in the extensions of *2-LC-lc*. (In fact, even the rules for exponent are not drastically changed, in comparison to the ones sketched in Section 7.1.3, except for the rule describing the action of the parameterized exponent functor on morphisms.) This means that if we accept two simple restrictions to the language, we can pretend that we work in an extension of *2-LC-lc*. The restrictions are: first, that the horizontal composition is not reducible to simpler terms in our system (and so shouldn't be used) and second, that the multiplication from the right by a functor containing exponents is banned. The restrictions ensure that not only the rules for *2-LC-lc* operations remain the same, but also the typing of functors and transformations is unchanged. Despite the required modifications of the equational theories, the equality test for functors is essentially unchanged. See Section A.1.2 for more details.

Moreover, the restrictions do not seem to be very grave even for a programmer willing to experiment with unusual language constructs, such as horizontal composition and multiplications. Propositions 7.2.24–7.2.27 show that in many common cases the horizontal composition may be written down using multiplications alone. If the user cares to make the typing and variance of terms explicit by using the multiplications, the terms from the *2-LC-lc* language are rewritten just as in the original reduction system for *2-LC-lc*. This is comforting, as the module language L-Dule described in Chapter 5 is embedded into *2-LC-lc* using the multiplications but with no use of horizontal composition.

Lifting the restriction on the use of the exponent functor would force us to complicate the derivation of domains and codomains for transformations, because the invariant that the domain of a multiplication by a functor is the composition of a domain and the functor no longer holds. We would have to change only the typing of horizontal composition and the multiplication from the right, but the new definition would have to be presented by cases over all the available term constructors.

However, the multiplication by an exponent functor does not seem to be very usable. The only use we are aware of is in the (co)inductive modules described in Section 9.1. In that particular context the multiplication may be reduced to simpler terms by an ad hoc rewriting procedure using the operation `T_ee`, as seen in Appendix A.2.4. A special form of multiplication would also be needed for (co)inductive types, if we allowed arbitrary exponents inside them. But again, the complete definition of the mapping operation would capture well all the complexity involved, without resorting to general multiplication. On the other hand, if rewriting of (co)inductive type operations were to be based on the non-distributive constructors (for example `T_fold` instead of `TL_fold`) then the multiplication would be needed (as seen, for example, in equation (101)).

Let Φ_e be a theory equationally generated from equations 233–238 and the axioms of theory Φ_i excluding the interchange law (20), but with the addition of rules of $\mathcal{R}_i$ except rule (29), treated as equations (see Section 6.3.3). Then the following fact, given without proof, holds. A long proof can be given by constructing a *2-LCO-A* with sums and (co)algebras using **Cat** and its exponent adjunction, then taking the reduct to obtain a *2-LCX-F* and then renaming the operations according to the language e-Core.

Observation 8.1.4. *Theory Φ_e is consistent.*

The theory Φ_e is not complete for *2-LCX-F* with sums, (co)algebras and exponents, because it does not contain the many specialized substitutes of the interchange law valid in *2-LCX-F*. The following observation follows from the respective theorems about components of Φ_e and from the definition of *2-LCX-F* with sums, (co)algebras and exponents.

Observation 8.1.5. *Φ_e is contained in the theory of 2-LCX-F with sums, (co)algebras and exponents.*

We believe that the complete theory of *2-LCX-F* with sums, (co)algebras and exponents has no finite equational axiomatization, just as we argued in the case of the theory of *2-LCX-F*.

8.1.5 Conclusion

We have defined exponents parameterizable on both the covariant and contravariant positions, as an instance of our mechanism of parameterized adjunctions. We identify the operation in which the exponent shows its contravariant nature, to be the multiplication by exponent functor from the right, where the domain and codomain at argument position are swapped. Using the properties of parameterized adjunctions we have managed to express the typing of the operation in the simple language e-Core. The construction of the rewriting rules for parameterized exponents benefits from our results about contravariance and parameterized adjunctions, but at the same time stays solely within the e-Core language. We observe the system is locally confluent (Observation 8.1.2) and conjecture it is strongly normalizing and so also confluent (Conjecture 8.1.3).

In the presence of inductive types, problems arise with rewriting terms that involve exponent constructors. In our reduction system the problems can be solved by providing reduction rules for the operation of mapping (an inductive analogue to multiplication). Our system facilitates easy experimentation with various sets of such reduction rules, while precluding adverse effects on the rewriting of other combinators.

We base our reduction system for e-Core on the theory of *2-LC-lc* with sums and (co)algebras extended with several most computationally important *2-LCX-F* equations expressed in the syntax of e-Core. In particular, we translate the weak η-equation for parameterized right adjunctions and we manage to sidestep the preconditions of the general reduction rule for multiplication by a right adjoint (Observation 8.1.5). The core language e-Core is adequate for use as a basis for our module system since all operations including exponent can be fully parameterized and at the same time the category closely resembles *2-LC-lc*.

8.2 f-Core — core language with general recursion

The addition of general recursion to a language based on such a simple categorical model as ours is problematic. The most direct approach is to add fixpoint or loop transformations to the model itself. There is also a possibility to treat fixpoint terms as infinite terms [29] and refuse to give meaning (set as undefined in the partial algebra) to those of them that do not rewrite to finite normal forms. The other solution with no semantics for looping programs is by regarding general recursion terms as shorthands for terms built using structured inductive and coinductive recursion. The combinations can be sometimes automatically synthesized, and the terms for which the automatic translation fails can be considered undefined, or not implementable.

We take the direct approach and postulate a fixpoint construction in the category, without determining the concrete method by which the fixpoint can be realized, such as complete ordering on morphisms [81]. We also weaken the coproduct construction in our categories, not imposing the uniqueness condition, but the weak eta-equation instead. We conjecture this prevents trivialization of our categories. In the rewriting we finally have to give up normalization, but we preserve confluence.

8.2.1 Language

Syntax

There are two additional `trans`-term constructors and no modifications to other syntactic domains.

```
type trans =
  ...
```

```
    | T_fix of trans
    | TL_fix of trans
```

Semantics

The first term constructor denotes the standard fixpoint operation we declare to be present in our categories. The second constructor acts on records with two fields labeled δ and ϵ, respectively, and treats the ϵ field as holding a constant parameter, looping only over the δ field. The equational theory presented in the next section shows that the operation corresponding to the second constructor is available in a *2-LCX-F* with sums, (co)algebras and exponents if only the standard fixpoints are available.

- $\texttt{T\_fix}(t) : \texttt{\{\}} \to h$
  is the fixpoint of transformation $t : h \to h$ where $h : c \to e$,
- $\texttt{TL\_fix}(t) : f \to h$
  is the fixpoint over the δ component of transformation $t : \texttt{\{}\delta : h\texttt{;}\ \epsilon : f\texttt{\}} \to h$ where $f, h : c \to e$.

8.2.2 Rewriting

Equations

The following equation describes `T_fix` as a fixpoint with respect to vertical composition.

$$\texttt{T\_fix}(t) \quad = \quad \texttt{T\_fix}(t)\ \texttt{.}\ t \tag{246}$$

The fixpoint operation is uniform with respect to multiplication from the left or, in other words, fixpoint is straightforwardly instantiated by types.

$$f * \texttt{T\_fix}(u) \quad = \quad \texttt{T\_fix}(f * u) \tag{247}$$

Just as for the case analysis and structured recursion distributive combinators, the main component of the domain of `TL_fix` is called δ and the auxiliary component for parameters is called ϵ. Unlike for the other combinators, there is no way to obtain a distributive fixpoint operation by composing normal fixpoint operation with some distribution morphism. As always, one can use the exponent to factor out the parameters and obtain the distributive version of the combinator.

$$\begin{aligned}
&\texttt{TL\_fix(\{}\delta\ \texttt{=}\ \delta\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ t\texttt{\}}\ \texttt{.}\ u\texttt{)} \ = \\
&\texttt{\{}\upsilon\ \texttt{= \{\} . T\_fix(T\_curry(\{}\delta\ \texttt{= T\_appl(}\epsilon\ \texttt{:}\ f\texttt{,}\ h\texttt{);}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\} .}\ u\texttt{));} \\
&\quad \epsilon\ \texttt{=}\ t\texttt{\} . T\_appl(}\epsilon\ \texttt{:}\ f\texttt{,}\ h\texttt{)}
\end{aligned} \tag{248}$$

Since for our fixpoint operation we have no equations concerning its uniqueness, we provide a second equation relating the two fixpoint operations.

$$\texttt{T\_fix}(t) \quad = \quad \texttt{TL\_fix(T\_pr}(\delta : h;\ \epsilon : \texttt{\{\}},\ \delta)\ .\ t) \tag{249}$$

We need a theory on which to base our reduction system for f-Core. Unfortunately, the theory induced by Φ_e and the three equations above is inconsistent [101]. The culprits are equation (246) and the uniqueness equation for coproducts (equation (68), Section 6.2.3). Since the uniqueness equation for coproducts is not directly used for rewriting, it does not follow from the theory $\overline{\mathcal{R}_e}$. We will define Φ_f to be the theory induced by the equations above, $\overline{\mathcal{R}_e}$ and the axioms of Φ_e with the exception of the uniqueness requirement for coproducts.

Conjecture 8.2.1. *Theory Φ_f is consistent.*

Reduction

There is much choice as for how to rewrite fixpoints. Here we define only the basic rules, leaving the details of their restriction or extension to be decided by the designer of a particular compiler based on the rewrite system. In particular, to prevent excessive fixpoint unwinding, one can adopt a kind of weak head normal form restriction, e.g., avoiding reduction under `T_curry`, `TL_case`, `TL_fold` and `TL_unfold` combinators. Alternatively, both the fixpoint and the vertical composition can be regarded with full laziness, with topological sorting of redexes or memoization helping to ensure that the fixpoints are not recomputed many times. In our prototype compiler we currently find a mixture of both approaches to give the best results. Regardless of the restrictions the reduction system is inherently divergent.

Note that the traditional approach of restricting fixpoint expressions only to function types is against the spirit of our development. We consider the composition, not function application, to be the fundamental operation. Nevertheless, the traditional behavior could be easily mimicked in our formalism. However, we need truly general fixpoints to define our (co)inductive modules. Fixpoint operations of most functional languages are too restricted to be sufficient.

We extend the reduction system $\mathcal{R}_e$ with the following rules.

$$\texttt{TL\_fix}(t) \quad \to \quad \texttt{\{}\delta \texttt{ = TL\_fix}(t)\texttt{;}\ \epsilon \texttt{ = T\_id}(f)\texttt{\}}\ .\ t \tag{250}$$

$$f * \texttt{TL\_fix}(u) \quad \to \quad \texttt{TL\_fix}(f * u) \tag{251}$$

$$\texttt{T\_fix}(t) \quad \to \quad \texttt{TL\_fix(T\_pr}(\delta : h;\ \epsilon : \texttt{\{\}},\ \delta)\ .\ t) \tag{252}$$

The first rule should surely be heavily restricted. We also add the rule describing substitution into the fixpoint constructor (where the second δ and the second ϵ are projections):

$$t\ .\ \texttt{TL\_fix}(u) \quad \to \quad \texttt{TL\_fix(\{}\delta \texttt{ = } \delta\texttt{;}\ \epsilon \texttt{ = } \epsilon\ .\ t\texttt{\}}\ .\ u) \tag{253}$$

Having a system that already needs restrictions to be practically useful, we may as well add the three remaining substitution rules of $\mathcal{R}_s$ and $\mathcal{R}_i$, that is rules (86), (143) and (144). Some remarks about evaluation strategy in the compiler of Dule, in particular, the restrictions on application of rules pertaining to fixpoints, are given in Section 10.3. The reduction system $\mathcal{R}_f$ is $\mathcal{R}_e$ enriched by the seven additional rules (250–253, (86), (143) and (144)), with no reduction restrictions.

Theorem 8.2.2. *Rules 250–253 are sound with respect to theory Φ_f.*

Proof. The soundness of rule (251) is easily proved using equation (248) and equations corresponding to the term constructors on its right hand side, in particular equation (247). Rule (253) follows trivially from equation (248) of Φ_f. Rules (86), (143) and (144) are derived similarly as other rules in their respective sections. We will now derive rule (250) from Φ_f.

We start with the left hand side

```
TL_fix(t)
```

and by equation (248) we get

```
{υ = {} . T_fix(T_curry({δ = T_appl(ϵ : f, h); ϵ = ϵ} . t));
 ϵ = T_id(f)} . T_appl(ϵ : f, h)
```

Then we use equation (246)

```
{υ = {} . T_fix(T_curry({δ = T_appl(ϵ : f, h); ϵ = ϵ} . t))
      . T_curry({δ = T_appl(ϵ : f, h); ϵ = ϵ} . t);
 ϵ = T_id(f)} . T_appl(ϵ : f, h)
```

and rule (239)

```
{υ = T_curry({υ = υ . {}
      . T_fix(T_curry({δ = T_appl(ϵ : f, h); ϵ = ϵ} . t)); ϵ = ϵ}
      . {δ = T_appl(ϵ : f, h); ϵ = ϵ} . t);
 ϵ = T_id(f)} . T_appl(ϵ : f, h)
```

We simplify using rule (242)

```
{υ = T_id(h); ϵ = T_id(f)} . {υ = υ . {}
 . T_fix(T_curry({δ = T_appl(ϵ : f, h); ϵ = ϵ} . t)); ϵ = ϵ}
 . {δ = T_appl(ϵ : f, h); ϵ = ϵ} . t
```

and using some other equations

```
{$\upsilon$ = {}
 . T_fix(T_curry({$\delta$ = T_appl($\epsilon$ : $f$, $h$); $\epsilon$ = $\epsilon$} . $t$)); $\epsilon$ = T_id($f$)}
 . {$\delta$ = T_appl($\epsilon$ : $f$, $h$); $\epsilon$ = $\epsilon$} . $t$
```

and then we simplify some more

```
{$\delta$ = {$\upsilon$ = {}
 . T_fix(T_curry({$\delta$ = T_appl($\epsilon$ : $f$, $h$); $\epsilon$ = $\epsilon$} . $t$)); $\epsilon$ = T_id($f$)}
 . T_appl($\epsilon$ : $f$, $h$);
 $\epsilon$ = T_id($f$)} . $t$
```

At last, we use equation (248) again

```
{$\delta$ = TL_fix($t$);
 $\epsilon$ = T_id($f$)} . $t$
```

obtaining the right hand side of rule (250). □

Because of the possible divergence caused by rule (250) the system $\mathcal{R}_f$ is not even weakly normalizing, hence solving of all critical pairs does not imply confluence. Nevertheless, the critical pairs are easily seen to be solvable.

Observation 8.2.3. *Reduction system $\mathcal{R}_f$ is locally confluent.*

All rules of $\mathcal{R}_f$ are left-linear with respect to `trans`-variables, as opposed to `funct`-variables, which however fixpoint terms cannot rewrite to. Therefore, assuming that $\mathcal{R}_e$ is strongly normalizing and confluent, we can use the technique of bounded recursion, as seen in [39] for a similar case, to conclude that the locally confluent system $\mathcal{R}_f$ is confluent. Details are out of the scope of our thesis.

Conjecture 8.2.4. *Reduction system $\mathcal{R}_f$ is confluent.*

8.2.3 Conclusion

The theory of strong coproducts and general fixpoints is inconsistent. However, while we need arbitrary fixpoints for our inductive module constructions, our analysis of reduction of sum type operations shows that the full strength of the coproduct uniqueness requirement is not necessary. Therefore we have constructed a theory Φ_f capturing the fixpoint properties but not containing the coproduct uniqueness equation. The theory is not complete for our categories, similarly as Φ_e has not been, but it has finite axiomatization. We conjecture Φ_f is consistent (Conjecture 8.2.1) and we show it is strong enough to entail each of the new rules of our reduction system for fixpoints $\mathcal{R}_f$ (Theorem 8.2.2). Assuming that $\mathcal{R}_f$ is confluent (Conjecture 8.2.4), we see that theory $\overline{\mathcal{R}_f}$ (which, however, does not contain numerous axioms of Φ_e) is consistent, since for instance terms `{}` and `{l = {}}` are different $\mathcal{R}_f$-normal forms.

The internal core language f-Core, with its reduction system $\mathcal{R}_f$, is the final step in the series of extensions of *2-LC-lc* developed in our thesis. System $\mathcal{R}_f$ is not normalizing, but locally confluent (Observation 8.2.3) and we conjecture it is confluent as well, guaranteeing unique normal forms for normalizing terms. The language f-Core is the richest core language instance for our base module system and the starting point for the module system extended with (co)inductive modules (Section 9.1), which requires inductive types and a fixpoint operation on values. The language is summarized in Appendix A.1 and a notation facilitating the use of this language by a programmer is presented in the next section.

Semantic combinators

In Section 4.4 we listed semantic combinators (as discussed in Section 3.2) of p-Core. After extending p-Core to f-Core we present the profiles of semantic combinators corresponding to the added operations. The semantic function for internal core language terms induced by the complete set of combinators (easily computed by substituting each term constructor with an application of the corresponding combinator) is written $[\![\_]\!]_i$.

Since the set of term constructors of the syntactic domain `cat` is not extended, we present no new combinators for `cat`. Here are semantic combinators corresponding to all new `funct`-term constructors.

```
val f_ss : Cat.t -> Funct.t IList.t -> Funct.t
val f_ii : Funct.t -> Funct.t
val f_tt : Funct.t -> Funct.t
val f_ee : Funct.t IList.t -> Funct.t -> Funct.t
```

There is also a large number of semantic combinators corresponding to new `trans`-term constructors.

```
val t_ss : Cat.t -> Trans.t IList.t -> Trans.t
val t_in : Funct.t IList.t -> IdIndex.t -> Trans.t
val t_case : Trans.t IList.t -> Funct.t -> Trans.t
val tl_case : Funct.t -> Trans.t IList.t -> Funct.t -> Trans.t
val t_map : Funct.t -> Trans.t -> Trans.t
val t_ii : Trans.t -> Trans.t
val t_con : Funct.t -> Trans.t
val t_fold : Funct.t -> Trans.t -> Trans.t
```

```
    val tl_fold : Funct.t -> Trans.t -> Trans.t
    val t_de : Funct.t -> Trans.t
    val t_tt : Trans.t -> Trans.t
    val t_uncon : Funct.t -> Trans.t
    val t_unfold : Funct.t -> Trans.t -> Trans.t
    val tl_unfold : Funct.t -> Trans.t -> Trans.t
    val t_unde : Funct.t -> Trans.t
    val t_ee : Trans.t IList.t -> Trans.t -> Trans.t
    val t_appl : Funct.t IList.t -> Funct.t -> Trans.t
    val t_curry : Trans.t -> Trans.t
    val t_fix : Trans.t -> Trans.t
    val tl_fix : Trans.t -> Trans.t
```

The added semantic combinators are used in implementation of the extensions presented in the subsequent sections, as well as in the definition of inductive modules in Section 9.1.2.

8.3 Extended notation for the core language

While f-Core has all the mechanisms we need in our core language, it still lacks the most common idioms and is obviously too verbose. The notation we develop for the bare core language in Section 8.3.1 improves the usability of the language and standardizes some patterns of operation, like the manner of propagating the environment, or the form of combinator operands. In Section 8.3.2 we complete the design of the Dule core language by defining the user core language, which extends the bare language by some more syntactic sugar. As before, we consider undefined any application of (co)algebra-related term constructors to such functors that cause problems due to contravariance, see Section 8.1.1. In particular, all propositions in this section have implicit side-conditions stating that the terms and derivations they refer to do not contain the offending functors.

8.3.1 Bare core language

The bare core language is based on the internal core language of Dule as developed throughout the thesis and summarized in Appendix A.1. In the design of the bare core language we've agreed on some simplifications, adopted some default conventions and added an array of quite complex abbreviations. In particular the source category of transformations is used to represent type variables and domain functor of transformations is used to represent value variables, see Section 4.2. Yet, we didn't use any meta-semantical mechanisms, like α-conversion or textual substitution. The variables are translated to projections, the binding constructions to categorical operations containing respective labels, etc. Then the compositional categorical semantics of the internal language applies.

Syntax

Here we will present only the concrete syntax. The abstract syntax of the bare core language can be found in module `BCore` in file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/core_front.ml`.

We will use the conventions for the description of grammars borrowed (with changes) from the reference manual of Casl [27]. The grammar uses uppercase words for nonterminal symbols, allowing also hyphens. All other characters stand for themselves, with the following exceptions:

- "`::=`" and "`|`" are generally used as meta-notation, as in BNF;
- "`%`" begins a comment, which is terminated by the end of line;
- a string of characters enclosed in double quotation marks `"..."` always stands for the enclosed characters themselves;
- "`N t...t N`" indicates zero or more repetitions of the nonterminal symbol `N` separated by the terminal symbol `t` (which is usually a semicolon or a bar);
- "`N...N`" is simply a sequence of zero or more repetitions of `N`,
- many nonterminals that are used in the repetitions have productions of the form "`LABEL SEPARATOR ITEM`" or "`PREFIX LABEL SEPARATOR ITEM`", for some syntactic domain of labels `LABEL`, and in this case the labels must not be duplicated throughout the sequence.

The labels that mark the entities of the bare core language are represented by four nonterminals.

- `DULE-LABEL` — capitalized labels,
- `TYPE-LABEL` — lowercase labels,
- `VALUE-LABEL` — lowercase labels,
- `CASE-LABEL` — lowercase and capitalized labels.

The productions are as follows.

```
FIELD-TYPE ::= VALUE-LABEL : TYPE
CASE-TYPE  ::= ‘ CASE-LABEL TYPE
PARAM-TYPE ::= ~ VALUE-LABEL : TYPE
TYPE       ::= TYPE . TYPE                      %composition
             | TYPE-LABEL                       %variable/projection
             | DULE-LABEL                       %variable/projection
             | { FIELD-TYPE ;...; FIELD-TYPE }  %product type
             | [ CASE-TYPE "|"..."|" CASE-TYPE ] %sum type
             | ind TYPE-LABEL : TYPE            %inductive type
             | coind TYPE-LABEL : TYPE          %coinductive type
             | PARAM-TYPE...PARAM-TYPE -> TYPE  %function type

FIELD      ::= VALUE-LABEL = VALUE
CASE       ::= ‘ CASE-LABEL VALUE
ARGUMENT   ::= ~ VALUE-LABEL : VALUE
PARAM      ::= ~ VALUE-LABEL : ( VALUE-LABEL : TYPE )
VALUE      ::= : TYPE                           %typing/identity
             | VALUE . VALUE                    %composition
             | VALUE-LABEL                      %variable/projection
             | DULE-LABEL                       %variable/projection
             | { FIELD ;...; FIELD }            %record
             | ‘ CASE-LABEL                     %variant/injection
             | [ CASE "|"..."|" CASE ]          %case analysis
             | map VALUE                        %mapping
             | con                              %inductive constructor
             | fold VALUE                       %inductive folding
             | de                               %inductive destructor
             | uncon                            %coinductive constructor
             | unfold VALUE                     %coinductive unfolding
             | unde                             %coinductive destructor
             | VALUE ARGUMENT...ARGUMENT        %application
             | fun PARAM...PARAM -> VALUE       %abstraction
             | VALUE ( ARGUMENT...ARGUMENT )    %partial application
             | assert VALUE in VALUE            %assertion
             | fix VALUE-LABEL : VALUE          %fixpoint
```

Warning: the case analysis, denoted by square brackets, takes operands of a slightly different form than the distributive case expression of f-Core that uses the same notation. Otherwise, if an expression of the bare language has the same syntax as an operation of the internal language then the typing and the semantics is also analogous.

Assigning kinds

Translation between the bare language and the internal language involves source, target, domain and codomain reconstruction. The type reconstruction algorithm (see Section 10.4) is quite intricate but the typing rules are simple. Before we present the rules, ordered by productions of the `TYPE` and `VALUE` syntactic domains, let us clarify the relationship between the semantics and typing of the bare core language. Let us fix an arbitrary categorical model of the internal core language. In Section 8.3.1 we define the semantics (translation) of the bare language in the internal language and, by composing with the semantics of the internal core language, the semantics of the bare language in the categorical model. We split the definition of the semantics in this way to enable the use of syntactic sugar for the internal language terms, which in this context is more readable than the OCaml notation with the semantic combinators resulting in the direct semantics in the categorical model. The same categorical model is used to decide semantic side conditions in the typing rules. The side conditions occur in a single rule for typing `TYPE`-terms and in a single rule for typing `VALUE`-terms.

Similarly as in the set of typing rules for the internal language, contained in Appendix A.1.3, the last rule for assigning kinds to `TYPE`-terms ties the typing with the semantics of kinds (the semantics $[\![\_]\!]_i$ of `cat`-terms in the categorical model). This allows one to read the rest of the rules as purely syntactic, assigning pairs of `cat`-terms (rather than pairs of categories) to `TYPE`-terms. The last rule in the next section serves an analogous purpose, enabling syntactic, though with semantic side conditions, derivation of types for `VALUE`-terms. This time the side conditions involve not only the fixed categorical model, but also the semantics of `TYPE`-terms (but not `VALUE`-terms) in the model. In the result, the typing depends on the semantics, which in turn depends on the typing, but the dependencies are stratified; they do not form a cycle.

We will denote the typings by " $\triangleright$ " not by " : " to avoid confusion with the colons of the language constructions. Remember that, as stated in Section 3.3, the typical identifiers for labels are: i, j, k; for categories (taken from the internal language and not expressible in the bare language; called kinds below): c, d, e, a, b; for types (elements of the syntactic domain `TYPE`): f, g, h and for values (elements of the syntactic domain `VALUE`): t and u.

First, we present the rules for deriving the source and target kinds of types of the bare core language. The kinds are just `cat`-terms of the internal core language and the derivation closely corresponds to the assignment of source and target categories to functors of the internal language. See next sections for a precise statement of this correspondence. The double typing rules for the (co)inductive types prevent problems, when label i is to occur free in the whole inductive type expression (to be a label of a component of the product source of the whole expression). Rules (258) and (260) apply only if label i does not occur among $j_1, \ldots, j_n$. With the other rules for (co)inductive types, the outer label i is not visible inside the expression and the just introduced label i is visible instead. The difference in typing will be meaningful in the semantics. The first two rules follow.

$$\frac{f \rhd c \longrightarrow d \qquad g \rhd d \longrightarrow e}{f \;\texttt{.}\; g \rhd c \longrightarrow e} \tag{254}$$

$$\frac{}{i \rhd \texttt{<}i \;\texttt{-}\; c\texttt{;}\; \ldots\texttt{>} \longrightarrow c} \tag{255}$$

In the presentation of the rules we use the same syntactic sugar as is given in Section 6.1.1 for describing indexed lists in equations and rewriting rules. According to the conventions the form of the rule above is equivalent to the following, more verbose form.

$$\frac{}{i_k \rhd \texttt{<}i_1 \;\texttt{-}\; c_1\texttt{;}\; \ldots\texttt{;}\; i_k \;\texttt{-}\; c_k\texttt{;}\; \ldots\texttt{;}\; i_n \;\texttt{-}\; c_n\texttt{>} \longrightarrow c_k}$$

We will mix the two styles of notation, tending towards the more succinct one, especially in case of longer rules.

$$\frac{f_1 \rhd c \longrightarrow \texttt{*} \quad \cdots \quad f_n \rhd c \longrightarrow \texttt{*}}{\texttt{\{}i_1 \;\texttt{:}\; f_1\texttt{;}\; \ldots\texttt{;}\; i_n \;\texttt{:}\; f_n\texttt{\}} \rhd c \longrightarrow \texttt{*}} \tag{256}$$

$$\frac{f_1 \rhd c \longrightarrow \texttt{*} \quad \cdots \quad f_n \rhd c \longrightarrow \texttt{*}}{\texttt{[`}i_1 \;\; f_1\texttt{|}\; \ldots\texttt{|`}i_n \;\; f_n\texttt{]} \rhd c \longrightarrow \texttt{*}} \tag{257}$$

$$\frac{g \rhd \texttt{<}i \;\texttt{-}\; \texttt{*}\texttt{;}\; j_1 \;\texttt{-}\; c_1\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{-}\; c_n\texttt{>} \longrightarrow \texttt{*}}{\texttt{ind}\; i\texttt{:}\; g \rhd \texttt{<}j_1 \;\texttt{-}\; c_1\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{-}\; c_n\texttt{>} \longrightarrow \texttt{*}} \tag{258}$$

$$\frac{g \rhd \texttt{<}i \;\texttt{-}\; \texttt{*}\texttt{;}\; j_1 \;\texttt{-}\; c_1\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{-}\; c_n\texttt{>} \longrightarrow \texttt{*}}{\texttt{ind}\; i\texttt{:}\; g \rhd \texttt{<}i \;\texttt{-}\; c\texttt{;}\; j_1 \;\texttt{-}\; c_1\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{-}\; c_n\texttt{>} \longrightarrow \texttt{*}} \tag{259}$$

$$\frac{g \rhd \texttt{<}i \;\texttt{-}\; \texttt{*}\texttt{;}\; j_1 \;\texttt{-}\; c_1\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{-}\; c_n\texttt{>} \longrightarrow \texttt{*}}{\texttt{coind}\; i\texttt{:}\; g \rhd \texttt{<}j_1 \;\texttt{-}\; c_1\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{-}\; c_n\texttt{>} \longrightarrow \texttt{*}} \tag{260}$$

$$\frac{g \rhd \texttt{<}i \;\texttt{-}\; \texttt{*}\texttt{;}\; j_1 \;\texttt{-}\; c_1\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{-}\; c_n\texttt{>} \longrightarrow \texttt{*}}{\texttt{coind}\; i\texttt{:}\; g \rhd \texttt{<}i \;\texttt{-}\; c\texttt{;}\; j_1 \;\texttt{-}\; c_1\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{-}\; c_n\texttt{>} \longrightarrow \texttt{*}} \tag{261}$$

$$\frac{g_1 \rhd c \longrightarrow \texttt{*} \quad \cdots \quad g_n \rhd c \longrightarrow \texttt{*} \qquad h \rhd c \longrightarrow \texttt{*}}{\texttt{\~{}}i_1\texttt{:}g_1 \;\ldots\; \texttt{\~{}}i_n\texttt{:}g_n \;\texttt{->}\; h \rhd c \longrightarrow \texttt{*}} \tag{262}$$

$$\frac{f \rhd c \to d \qquad [\![c]\!]_i = [\![c']\!]_i \qquad [\![d]\!]_i = [\![d']\!]_i}{f \rhd c' \to d'} \tag{263}$$

The reconstruction of source and target kinds of a given type is an easy task, since the bound identifiers in the only operations introducing them (the (co)inductive type constructors) are explicitly listed and are always assigned the base kind `*`. However, the type projection introduces enough nondeterminism in rule (255) to warrant nontrivial definition of the least detailed kinds of a type (a variation on the usual definition of the most general type). The definition below assigns to a projection the following least detailed source and target:

$$i : \texttt{<}\ i \texttt{: <>>} \to \texttt{<>}$$

Definition 8.3.1. *An indexed list of kinds (categories) lc is said to be less or equally detailed than ld, if for each element c of lc at index k, the element k of ld exists and is less or equally detailed than c. A* `cat`*-term c is less or equally detailed than d if* $[\![c]\!]_i$ *and* $[\![d]\!]_i$ *are equal or c is the empty product category* `<>` *or c and d have the same root constructor and the corresponding category or category list subterms of c are less or equally detailed than the corresponding subterms of d.*

Observation 8.3.2. *For any type f, if the set of pairs of source and target derivable for f is nonempty, it contains the pair consisting of the least detailed source of all sources and the least detailed target of all targets of f.*

Assigning types

Now we give the rules for derivation of domain and codomain types of values of the bare core language. The derived domain and codomain type of a value agrees with the internal language domain and codomain functor of the represented transformation. This property will be strictly formulated after we give the semantics of the bare core language.

To improve readability of typing rules, we temporarily add an additional constructor of `TYPE`-terms, called semantic type substitution. The operation resembles the standard textual type substitution, but we do not make it a meta-language operation, since substitutions are easily internalizable in the formalism of the internal core language.

```
TYPE ::= ...
       | TYPE [ TYPE / TYPE-LABEL ]
```

The bare core language with the additional constructor will be called the extended core language. Each ground `TYPE`-term of the extended core language can also be expressed in the bare core language, as will be proved later. However, the additional operation is not expressible in the pure bare language, so it is not just a shorthand. The constructor has the following rule for assigning kinds and its semantics is defined alongside the other operations in the next section.

$$\frac{h \triangleright \texttt{<}j_1\ \texttt{-}\ c_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{-}\ c_n\texttt{>} \rightarrow \texttt{*} \qquad g \triangleright \texttt{<}i\ \texttt{-}\ \texttt{*}\texttt{;}\ j_1\ \texttt{-}\ c_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{-}\ c_n\texttt{>} \rightarrow \texttt{*}}{g[h/i] \triangleright \texttt{<}j_1\ \texttt{-}\ c_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{-}\ c_n\texttt{>} \rightarrow \texttt{*}} \tag{264}$$

The typing rules for values follow. The rules 271–277 have additional, unwritten premises concerning the kinds of type g, needed for the semantic type substitutions in the rules to be type-correct. As always, and in particular in rule (280), the order of indexed list components is unimportant and the labels must not be repeated. In rule (279) neither the internal nor the external names of the parameters may be duplicated. The label `it` is a member of the `VALUE-LABEL` syntactic domain and the user is permitted to use it in ordinary program code.

Arguments to combinators in all of the rules (270), (271), (273) and (276) are typed in the same way: as functions with one of the parameters called `it`. The results of the combinators are also typed in this way so that using the results of nested combinators as arguments to others is easy. In rule (280) we introduce the distinctive syntax for partial application, described in Section 2.1.3. There are two rules for the fixpoint construction to cope with the case of label i free in the whole expression, similarly as in the rules for the inductive types. Rule (282) applies only if i does not occur in $j_1, \ldots, j_n$.

$$\frac{}{\texttt{:}\ g \triangleright g \rightarrow g} \tag{265}$$

$$\frac{t \triangleright f \rightarrow g \qquad u \triangleright g \rightarrow h}{t\ \texttt{.}\ u \triangleright f \rightarrow h} \tag{266}$$

$$\frac{}{i \triangleright \texttt{\{}i\ \texttt{:}\ f\texttt{;}\ \ldots\texttt{\}} \rightarrow f} \tag{267}$$

$$\frac{t_1 \triangleright f \rightarrow h_1 \quad \cdots \quad t_n \triangleright f \rightarrow h_n}{\texttt{\{}i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{\}} \triangleright f \rightarrow \texttt{\{}i_1\ \texttt{:}\ h_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ h_n\texttt{\}}} \tag{268}$$

$$\frac{}{\texttt{`}i \triangleright f \rightarrow \texttt{[`}i\ \ f\texttt{|}\ \ldots\texttt{]}} \tag{269}$$

$$\frac{\begin{array}{c} t_1 \triangleright f \to \texttt{~it:}f_1\ \texttt{~}j_1\texttt{:}g_1\ \ldots\ \texttt{~}j_n\texttt{:}g_n\ \texttt{->}\ h \\ t_2 \triangleright f \to \texttt{~it:}f_2\ \texttt{~}j_1\texttt{:}g_1\ \ldots\ \texttt{~}j_n\texttt{:}g_n\ \texttt{->}\ h \\ \vdots \end{array}}{\begin{array}{c} \texttt{[`}i_1\ \ t_1\texttt{|`}i_2\ \ t_2\texttt{|}\ \ldots\texttt{]} \triangleright f \to \\ \texttt{~it:[`}i_1\ \ f_1\texttt{|`}i_2\ \ f_2\texttt{|}\ \ldots\texttt{]}\ \texttt{~}j_1\texttt{:}g_1\ \ldots\ \texttt{~}j_n\texttt{:}g_n\ \texttt{->}\ h \end{array}} \tag{270}$$

$$\frac{t \triangleright f \to \texttt{~it:}g'\ \texttt{~}j_1\texttt{:}g_1\ \ldots\ \texttt{~}j_n\texttt{:}g_n\ \texttt{->}\ h}{\texttt{map}\ t \triangleright f \to \texttt{~it:}g[g'/i]\ \texttt{~}j_1\texttt{:}g_1\ \ldots\ \texttt{~}j_n\texttt{:}g_n\ \texttt{->}\ g[h/i]} \tag{271}$$

$$\frac{}{\texttt{con} \triangleright g[\texttt{ind}\ i\texttt{:}\ g/i] \to \texttt{ind}\ i\texttt{:}\ g} \tag{272}$$

$$\frac{t \triangleright f \to \texttt{~it:}g[h/i]\ \texttt{~}j_1\texttt{:}g_1\ \ldots\ \texttt{~}j_n\texttt{:}g_n\ \texttt{->}\ h}{\texttt{fold}\ t \triangleright f \to \texttt{~it:}(\texttt{ind}\ i\texttt{:}\ g)\ \texttt{~}j_1\texttt{:}g_1\ \ldots\ \texttt{~}j_n\texttt{:}g_n\ \texttt{->}\ h} \tag{273}$$

$$\frac{}{\texttt{de} \triangleright \texttt{ind}\ i\texttt{:}\ g \to g[\texttt{ind}\ i\texttt{:}\ g/i]} \tag{274}$$

$$\frac{}{\texttt{uncon} \triangleright g[\texttt{coind}\ i\texttt{:}\ g/i] \to \texttt{coind}\ i\texttt{:}\ g} \tag{275}$$

$$\frac{t \triangleright f \to \texttt{~it:}h\ \texttt{~}j_1\texttt{:}g_1\ \ldots\ \texttt{~}j_n\texttt{:}g_n\ \texttt{->}\ g[h/i]}{\texttt{unfold}\ t \triangleright f \to \texttt{~it:}h\ \texttt{~}j_1\texttt{:}g_1\ \ldots\ \texttt{~}j_n\texttt{:}g_n\ \texttt{->}\ \texttt{coind}\ i\texttt{:}\ g} \tag{276}$$

$$\frac{}{\texttt{unde} \triangleright \texttt{coind}\ i\texttt{:}\ g \to g[\texttt{coind}\ i\texttt{:}\ g/i]} \tag{277}$$

$$\frac{t \triangleright f \to \texttt{~}i_1\texttt{:}g_1\ \ldots\ \texttt{~}i_n\texttt{:}g_n\ \texttt{->}\ h \qquad t_1 \triangleright f \to g_1 \quad \cdots \quad t_n \triangleright f \to g_n}{t\ \texttt{~}i_1\texttt{:}t_1\ \ldots\ \texttt{~}i_n\texttt{:}t_n \triangleright f \to h} \tag{278}$$

$$\frac{t \triangleright \texttt{\{}i_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{;}\ k_1\ \texttt{:}\ g_1\texttt{;}\ \ldots\texttt{;}\ k_n\ \texttt{:}\ g_n\texttt{\}} \to h}{\begin{array}{c} \texttt{fun}\ \texttt{~}j_1\texttt{:(}k_i\texttt{:}g_1\texttt{)}\ \ldots\ \texttt{~}j_n\texttt{:(}k_n\texttt{:}g_n\texttt{)}\ \texttt{->}\ t \triangleright \\ \texttt{\{}i_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{;}\ k_{n_1}\ \texttt{:}\ h_{n_1}\texttt{;}\ \ldots\texttt{\}} \to (\texttt{~}j_1\texttt{:}g_1\ \ldots\ \texttt{~}j_n\texttt{:}g_n\ \texttt{->}\ h) \end{array}} \tag{279}$$

$$\frac{t \rhd f \to \texttt{~}i_1\texttt{:}g_1 \ \ldots\ \texttt{~}i_n\texttt{:}g_n\ \texttt{->}\ h \qquad t_1 \rhd f \to g_1 \quad \cdots \quad t_k \rhd f \to g_k}{t\ \texttt{(~}i_1\texttt{:}t_1\ \ldots\ \texttt{~}i_k\texttt{:}t_k\texttt{)} \rhd f \to \texttt{~}i_{k+1}\texttt{:}g_{k+1}\ \ldots\ \texttt{~}i_n\texttt{:}g_n\ \texttt{->}\ h} \tag{280}$$

$$\frac{u \rhd f \to \texttt{[`True \{\}|`False \{\}]} \qquad t \rhd f \to h}{\texttt{assert}\ u\ \texttt{in}\ t \rhd f \to h} \tag{281}$$

$$\frac{t \rhd \texttt{\{}i\ \texttt{:}\ h\texttt{;}\ j_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ f_n\texttt{\}} \to h}{\texttt{fix}\ i\texttt{:}\ t \rhd \texttt{\{}j_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ f_n\texttt{\}} \to h} \tag{282}$$

$$\frac{t \rhd \texttt{\{}i\ \texttt{:}\ h\texttt{;}\ j_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ f_n\texttt{\}} \to h}{\texttt{fix}\ i\texttt{:}\ t \rhd \texttt{\{}i\ \texttt{:}\ g\texttt{;}\ j_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ f_n\texttt{\}} \to h} \tag{283}$$

$$\frac{t \rhd f \to g \qquad [\![f]\!]_b = [\![f']\!]_b \qquad [\![g]\!]_b = [\![g']\!]_b}{t \rhd f' \to g'} \tag{284}$$

The reconstruction of domain and codomain types of a given value is very difficult. A portion of the difficulty comes from the complicated two-part unification procedure, reflecting the additional way a type may be more general than the other — by having less component in some of its indexed lists. Another difficulty comes from the absence of a fixed "environment". Still another complication comes from the polytypic combinators and their interaction with unification of indexed lists. The most problematic combinator seems to be the mapping combinator, as captured by rule (271), where type g is hard to guess on the basis of its two occurrences with substituted subterms (in particular if the subterms are not proper!). The definition of generality of typing that seems closest to the similar definitions for conventional type systems is the following.

Definition 8.3.3. *An indexed list of types lf is said to be less or equally detailed than lg, if for each element f of lf at index k, the element k of lg exists and is less or equally detailed than f. A type term f is less or equally detailed than g if $[\![f]\!]_b$ and $[\![g]\!]_b$ are equal or f is the empty product type* `{}` *or f and g have the same root constructor and the corresponding type or type list subterms of f are less or equally detailed than the corresponding subterms of g.*

Observation 8.3.4. *There are typable values that have no least detailed elements in their sets of derivable domains nor in their sets of derivable codomains. For example the term*

```
(map fun ~it:(it:[`A {}]) -> {} .`B) ~it:{} .`A
```

can have both `[‘A {}]` *and* `[‘B {}]` *as its codomain, but cannot have* `{}` *(their only lower bound).*

See Section 2.1.2 of the Dule tutorial for example uses of the mapping combinator and Section 10.4 for a discussion of our type-reconstruction algorithm that finds a minimally detailed typing for a given value. Choosing which of the minimally detailed typings are to be generated by the compiler will be an interesting research topic, most probably to be based on case studies and pragmatics of their use. We believe that with a good choice of typing defaults in the compiler, the programmer will very rarely have to direct type-reconstruction by explicit typing, especially that our language encourages fine-grained modularization, already providing numerous typing hints.

Semantics

We define the semantics $[\![\_]\!]_b$ of types and values of the extended core language into the functors and transformations of the fixed categorical model of the internal language, respectively. See also its implementation in module `SemConTTrans` in the file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/core_front.ml`. The semantics is only assigned to type-correct terms and depends on the derivation of their typing. Usually, first we derive the least detailed kinds for a given type and only then compute its semantics. Similarly, first we derive domain and codomain types for a given value (the least detailed, if possible), such that the simultaneously derived kinds for the domain and codomain are the least detailed, and only then compute the semantics of the value. To define the semantics we provide the translation $[\![\_]\!]$ from the bare core language to the internal core language and we set

$$[\![x]\!]_b = [\![[\![x]\!]]\!]_i$$

where x is a type or a value and $[\![\_]\!]_i$ is the semantics of the internal language terms in the fixed categorical model, defined in Section 8.2.3.

The types have the following semantics, defined by induction over the structure of derivation of the type's kinds. The definition is mostly straightforward; in fact, unlike the semantics of values, the definition doesn't need the whole derivation of kinds, only the result of the derivation. Despite four typing rules for the (co)inductive types we define the semantics using only two equations. The difference between the alternative typing rules for the (co)inductive types affects the semantic equations only in the domain of functor projections ι and κ. The projections have the product component i inside the component κ, if and only if rule (259) or (261) has been used in the last step of derivation. Labels υ, ι, κ, δ and ϵ are not accessible to the user.

$$[\![f\ .\ g \triangleright c \rightarrow e]\!] \quad = \quad \texttt{F\_COMP}([\![f]\!], [\![g]\!])$$

$$\begin{aligned}
[\![i \triangleright \texttt{<}i\ \texttt{-}\ c\texttt{;}\ \ldots\texttt{>} \to c]\!] &= \texttt{F\_PR}(i\ \texttt{-}\ c\texttt{;}\ \ldots, i) \\
[\![\texttt{\{}i_1 : f_1\texttt{;}\ \ldots\texttt{;}\ i_n : f_n\texttt{\}} \triangleright c \to *]\!] &= \texttt{F\_pp}(c, i_1 : [\![f_1]\!]\texttt{;}\ \ldots\texttt{;}\ i_n : [\![f_n]\!]) \\
[\![\texttt{[`}i_1\ f_1 \texttt{|}\ \ldots \texttt{|`}i_n\ f_n\texttt{]} \triangleright c \to *]\!] &= \texttt{F\_ss}(c, i_1 : [\![f_1]\!]\texttt{;}\ \ldots\texttt{;}\ i_n : [\![f_n]\!]) \\
[\![\texttt{ind}\ i\texttt{:}\ g \triangleright \texttt{<}j_1\ \texttt{-}\ c_1\texttt{;}\ \ldots\texttt{>} \to *]\!] &= \texttt{F\_ii}(\texttt{<}i : \iota\texttt{;}\ j_1 : \kappa\ \texttt{.}\ j_1\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ [\![g]\!]) \\
[\![\texttt{coind}\ i\texttt{:}\ g \triangleright \texttt{<}j_1\ \texttt{-}\ c_1\texttt{;}\ \ldots\texttt{>} \to *]\!] &= \texttt{F\_tt}(\texttt{<}i : \iota\texttt{;}\ j_1 : \kappa\ \texttt{.}\ j_1\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ [\![g]\!]) \\
[\![\texttt{\~{}}i_1\texttt{:}g_1\ \ldots\ \texttt{\~{}}i_n\texttt{:}g_n\ \texttt{->}\ h \triangleright c \to *]\!] &= \texttt{F\_ee}(i_1 : [\![g_1]\!]\texttt{;}\ \ldots\texttt{;}\ i_n : [\![g_n]\!], [\![h]\!])
\end{aligned}$$

$$\begin{aligned}
&[\![g[h/i] \triangleright \texttt{<}j_1\ \texttt{-}\ c_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{-}\ c_n\texttt{>} \to *]\!] = \\
&\texttt{F\_COMP}(\texttt{F\_RECORD}(\texttt{<}j_1\ \texttt{-}\ c_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{-}\ c_n\texttt{>}, \\
&\qquad\qquad i : [\![h]\!]\texttt{;}\ j_1 : j_1\texttt{;}\ \ldots\texttt{;}\ j_n : j_n), [\![g]\!])
\end{aligned}$$

The same semantics may be presented more succinctly by using more syntactic sugar and ignoring the kinds.

$$\begin{aligned}
[\![f\ \texttt{.}\ g]\!] &= [\![f]\!]\ \texttt{.}\ [\![g]\!] \\
[\![i]\!] &= i \\
[\![\texttt{\{}i_1 : f_1\texttt{;}\ \ldots\texttt{;}\ i_n : f_n\texttt{\}}]\!] &= \texttt{\{}i_1 : [\![f_1]\!]\texttt{;}\ \ldots\texttt{;}\ i_n : [\![f_n]\!]\texttt{\}} \\
[\![\texttt{[`}i_1\ f_1 \texttt{|}\ \ldots \texttt{|`}i_n\ f_n\texttt{]}]\!] &= \texttt{[`}i_1\ [\![f_1]\!] \texttt{|}\ \ldots \texttt{|`}i_n\ [\![f_n]\!]\texttt{]} \\
[\![\texttt{ind}\ i\texttt{:}\ g]\!] &= \texttt{F\_ii}(\texttt{<}i : \iota\texttt{;}\ j_1 : \kappa\ \texttt{.}\ j_1\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ [\![g]\!]) \\
[\![\texttt{coind}\ i\texttt{:}\ g]\!] &= \texttt{F\_tt}(\texttt{<}i : \iota\texttt{;}\ j_1 : \kappa\ \texttt{.}\ j_1\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ [\![g]\!]) \\
[\![\texttt{\~{}}i_1\texttt{:}g_1\ \ldots\ \texttt{\~{}}i_n\texttt{:}g_n\ \texttt{->}\ h]\!] &= \texttt{F\_ee}(i_1 : [\![g_1]\!]\texttt{;}\ \ldots\texttt{;}\ i_n : [\![g_n]\!], [\![h]\!])
\end{aligned}$$

$$[\![g[h/i]]\!] = \texttt{<}i : [\![h]\!]\texttt{;}\ j_1 : j_1\texttt{;}\ \ldots\texttt{;}\ j_n : j_n\texttt{>}\ \texttt{.}\ [\![g]\!]$$

Values have the following semantics, which is defined by induction over the structure of derivation of the value's domain and codomain type. The result of the derivation does not suffice, because the interaction of composition, projection, injection and record makes it impossible to reconstruct the derivation from its result, as in `{sum = `in; dud = {}} . dud`. Consequently, semantic clauses have to be read together with derivation rules. The typing of subterms makes it possible to determine the types on the right hand sides of semantic equalities, e.g., the types called lg' in the clauses below and others hidden in the syntactic sugar.

Dependencies of all the types of the right hand sides on the types of the left hand sides are explicitly encoded in the parts of the Dule compiler corresponding to the semantic clauses below (see `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/core_front.ml`). Among the semantic equations we have only one clause for the fixpoint operation, but its right hand side depends on the typing of the left hand side, as obtained using one of the two rules for fixpoints.

$$[\![\texttt{:}\ g]\!] = \texttt{:}\ [\![g]\!]$$

$$
\begin{aligned}
\llbracket t\ \texttt{.}\ u \rrbracket &= \llbracket t\rrbracket\ \texttt{.}\ \llbracket u\rrbracket\\
\llbracket i\rrbracket &= i\\
\llbracket \texttt{\{}i_1\ \texttt{=}\ t_1\texttt{;}\ \dots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{\}}\rrbracket &= \texttt{\{}i_1\ \texttt{=}\ \llbracket t_1\rrbracket\texttt{;}\ \dots\texttt{;}\ i_n\ \texttt{=}\ \llbracket t_n\rrbracket\texttt{\}}\\
\llbracket \texttt{`}i\rrbracket &= \texttt{`}i
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \texttt{[`}i_1\ \ t_1\texttt{|`}i_2\ \ t_2\texttt{|}\ \dots\texttt{]}\rrbracket = \texttt{T\_curry(}\\
&\texttt{\{}\delta\ \texttt{=}\ \texttt{it;}\ \epsilon\ \texttt{=}\ \texttt{\{}\upsilon\ \texttt{=}\ \upsilon\texttt{;}\ j_1\ \texttt{=}\ j_1\texttt{;}\ \dots\texttt{\}\}}\\
&\ \texttt{.}\ \texttt{[`}i_1\ \ \texttt{\{it}\ \texttt{=}\ \delta\texttt{;}\ \upsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ \upsilon\ \texttt{.}\ \llbracket t_1\rrbracket\texttt{;}\ j_1\ \texttt{=}\ \epsilon\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{\}}\ \texttt{.}\ \texttt{T\_appl}(lg', h)\\
&\quad \texttt{|`}i_2\ \ \texttt{\{it}\ \texttt{=}\ \delta\texttt{;}\ \upsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ \upsilon\ \texttt{.}\ \llbracket t_2\rrbracket\texttt{;}\ j_1\ \texttt{=}\ \epsilon\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{\}}\ \texttt{.}\ \texttt{T\_appl}(lg', h)\\
&\quad \texttt{|}\dots\texttt{])}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \texttt{map}\ t\rrbracket &= \texttt{T\_curry(}\\
&\quad \texttt{\{}\delta\ \texttt{=}\ \texttt{it;}\ \epsilon\ \texttt{=}\ \texttt{\{}\upsilon\ \texttt{=}\ \upsilon\texttt{;}\ j_1\ \texttt{=}\ j_1\texttt{;}\ \dots\texttt{\}\}}\\
&\quad\ \texttt{.}\ \texttt{T\_map}(g, \texttt{\{it}\ \texttt{=}\ \delta\texttt{;}\ \upsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ \upsilon\ \texttt{.}\ \llbracket t\rrbracket\texttt{;}\\
&\qquad\qquad\qquad j_1\ \texttt{=}\ \epsilon\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{\}}\ \texttt{.}\ \texttt{T\_appl}(lg', h)))\\
\llbracket \texttt{con}\rrbracket &= \texttt{T\_con(<}i\ \texttt{:}\ \iota\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{>}\ \texttt{.}\ g)\\
\llbracket \texttt{fold}\ t\rrbracket &= \texttt{T\_curry(}\\
&\quad \texttt{\{}\delta\ \texttt{=}\ \texttt{it;}\ \epsilon\ \texttt{=}\ \texttt{\{}\upsilon\ \texttt{=}\ \upsilon\texttt{;}\ j_1\ \texttt{=}\ j_1\texttt{;}\ \dots\texttt{\}\}}\\
&\quad\ \texttt{.}\ \texttt{TL\_fold}(g, \texttt{\{it}\ \texttt{=}\ \delta\texttt{;}\ \upsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ \upsilon\ \texttt{.}\ \llbracket t\rrbracket\texttt{;}\\
&\qquad\qquad\qquad j_1\ \texttt{=}\ \epsilon\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{\}}\ \texttt{.}\ \texttt{T\_appl}(lg', h)))\\
\llbracket \texttt{de}\rrbracket &= \texttt{T\_de(<}i\ \texttt{:}\ \iota\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{>}\ \texttt{.}\ g)\\
\llbracket \texttt{uncon}\rrbracket &= \texttt{T\_uncon(<}i\ \texttt{:}\ \iota\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{>}\ \texttt{.}\ g)\\
\llbracket \texttt{unfold}\ t\rrbracket &= \texttt{T\_curry(}\\
&\quad \texttt{\{}\delta\ \texttt{=}\ \texttt{it;}\ \epsilon\ \texttt{=}\ \texttt{\{}\upsilon\ \texttt{=}\ \upsilon\texttt{;}\ j_1\ \texttt{=}\ j_1\texttt{;}\ \dots\texttt{\}\}}\\
&\quad\ \texttt{.}\ \texttt{TL\_unfold}(g, \texttt{\{it}\ \texttt{=}\ \delta\texttt{;}\ \upsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ \upsilon\ \texttt{.}\ \llbracket t\rrbracket\texttt{;}\\
&\qquad\qquad\qquad j_1\ \texttt{=}\ \epsilon\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{\}}\ \texttt{.}\ \texttt{T\_appl}(lg', h')))\\
\llbracket \texttt{unde}\rrbracket &= \texttt{T\_unde(<}i\ \texttt{:}\ \iota\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{>}\ \texttt{.}\ g)
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket t\ \texttt{\char`~}i_1\texttt{:}t_1\ \dots\ \texttt{\char`~}i_n\texttt{:}t_n\rrbracket =\\
&\texttt{\{}\upsilon\ \texttt{=}\ \llbracket t\rrbracket\texttt{;}\ i_1\ \texttt{=}\ \llbracket t_1\rrbracket\texttt{;}\ \dots\texttt{;}\ i_n\ \texttt{=}\ \llbracket t_n\rrbracket\texttt{\}}\ \texttt{.}\ \texttt{T\_appl}(lg, h)
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \texttt{fun}\ \texttt{\char`~}j_1\texttt{:(}k_1\texttt{:}g_1\texttt{)}\ \dots\ \texttt{\char`~}j_n\texttt{:(}k_n\texttt{:}g_n\texttt{)}\ \texttt{->}\ t\rrbracket =\\
&\texttt{T\_curry(\{}i_1\ \texttt{=}\ \upsilon\ \texttt{.}\ i_1\texttt{;}\ \dots\texttt{;}\ k_1\ \texttt{=}\ j_1\texttt{;}\ \dots\texttt{;}\ k_n\ \texttt{=}\ j_n\texttt{\}}\ \texttt{.}\ \llbracket t\rrbracket)
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket t\ \texttt{(\char`~}i_1\texttt{:}t_1\ \dots\ \texttt{\char`~}i_k\texttt{:}t_k\texttt{)}\rrbracket = \texttt{T\_curry(}\\
&\texttt{\{}\upsilon\ \texttt{=}\ \upsilon\ \texttt{.}\ \llbracket t\rrbracket\texttt{;}\ i_1\ \texttt{=}\ \upsilon\ \texttt{.}\ \llbracket t_1\rrbracket\texttt{;}\ \dots\texttt{;}\ i_k\ \texttt{=}\ \upsilon\ \texttt{.}\ \llbracket t_k\rrbracket\texttt{;}\\
&\ i_{k+1}\ \texttt{=}\ i_{k+1}\texttt{;}\ \dots\texttt{;}\ i_n\ \texttt{=}\ i_n\texttt{\}}\ \texttt{.}\ \texttt{T\_appl}(lg, h))
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \texttt{assert}\ u\ \texttt{in}\ t\rrbracket &= \llbracket t\rrbracket\\
\llbracket \texttt{fix}\ i\texttt{:}\ t\rrbracket &= \texttt{TL\_fix(\{}i\ \texttt{=}\ \delta\texttt{;}\ j_1\ \texttt{=}\ \epsilon\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{\}}\ \texttt{.}\ \llbracket t\rrbracket)
\end{aligned}
$$

Remark. The semantics we defined corresponds to the behavior of the Dule compiler started with the option `--no-assert`. The equation for `assert` shows that the assertions themselves have no effect on the semantics of a value. However, without `--no-assert` the value of an expression containing assertion might be undefined, if the assertion fails. Whether the assertions are verified and at which moment, is left for the compiler designer to decide. For example, in the current implementation the assertions are simplified and verified already at compile time, if possible.

Properties

The typing system for the bare core language enjoys the following soundness properties.

Observation 8.3.5. *Let $f \rhd c \longrightarrow e$ be a result of a derivation of source and target kinds (`cat`-terms) for an extended core language type. Then the source and target categories of the functor $[\![f \rhd c \longrightarrow e]\!]_b$ in the fixed categorical model of the internal language are $[\![c]\!]_i$ and $[\![e]\!]_i$, respectively. (Remember that kinds are terms of the internal language, as opposed to types, which are terms of the bare language.)*

Observation 8.3.6. *Let $v \rhd f \longrightarrow h$ be a result of a derivation of a domain and codomain types for an extended core language value. Then the domain and codomain functors of the transformation $[\![v \rhd f \longrightarrow h]\!]_b$ in the fixed model of the internal language are $[\![f]\!]_b$ and $[\![g]\!]_b$, respectively.*

The following lemma states that the same bare language type term can be considered as having any number of void dependencies on other types.

Lemma 8.3.7. *Let h be a ground bare language type term, such that*

$$h \rhd \texttt{<}j_1 \texttt{ - } c_1 \texttt{;} \ \ldots \texttt{;} \ j_n \texttt{ - } c_n\texttt{>} \longrightarrow *$$

Then there is a derivation of the following source and target kinds for h

$$h \rhd \texttt{<}i \texttt{ - } c\texttt{;} \ j_1 \texttt{ - } c_1 \texttt{;} \ \ldots \texttt{;} \ j_n \texttt{ - } c_n\texttt{>} \longrightarrow *$$

where c is arbitrary and additionally

$$\begin{aligned} &[\![\texttt{<}j_1 \texttt{ : } j_1\texttt{;} \ \ldots \texttt{;} \ j_n \texttt{ : } j_n\texttt{>} \texttt{ . } [\![h \rhd \texttt{<}j_1 \texttt{ - } c_1\texttt{;} \ \ldots \texttt{;} \ j_n \texttt{ - } c_n\texttt{>} \longrightarrow *]\!]]\!]_i \\ = \ &[\![h \rhd \texttt{<}i \texttt{ - } c\texttt{;} \ j_1 \texttt{ - } c_1\texttt{;} \ \ldots \texttt{;} \ j_n \texttt{ - } c_n\texttt{>} \longrightarrow *]\!]_b \end{aligned}$$

Proof. The proof of the first part is by an easy induction on the derivation of typing for h. The double typing rules for the (co)inductive types prevent problems with the additional label i not being free in subterms.

The proof of the equation is also by induction on the derivation of typing for h. The hardest cases are those for the (co)inductive types. We present the most complicated of the cases for the inductive type. The equation to prove is

$$\begin{aligned}&[\![\texttt{<}j_1\ \texttt{:}\ j_1\texttt{;}\ \dots\texttt{;}\ j_n\ \texttt{:}\ j_n\texttt{>}\ \texttt{.}\ [\![\texttt{(ind}\ i\texttt{:}\ g\texttt{)} \rhd \texttt{<}j_1\ \texttt{-}\ c_1\texttt{;}\ \dots\texttt{;}\ j_n\ \texttt{-}\ c_n\texttt{>} \longrightarrow \texttt{*}]\!]]\!]_i\\ &= [\![\texttt{(ind}\ i\texttt{:}\ g\texttt{)} \rhd \texttt{<}i\ \texttt{-}\ c\texttt{;}\ j_1\ \texttt{-}\ c_1\texttt{;}\ \dots\texttt{;}\ j_n\ \texttt{-}\ c_n\texttt{>} \longrightarrow \texttt{*}]\!]_b\end{aligned}$$

The typings of the inductive type at the left and the right hand sides must have come from the use of the two different rules, because label i bound in the inductive construction appears in the source kind of the inductive type in the second typing, but not in the first.

We start with the left hand side of the equation (and move from the level of the categorical model to the level of the internal core language, at which our equation theory operates).

$$\texttt{<}j_1\ \texttt{:}\ j_1\texttt{;}\ \dots\texttt{;}\ j_n\ \texttt{:}\ j_n\texttt{>}\ \texttt{.}\ [\![\texttt{(ind}\ i\texttt{:}\ g\texttt{)} \rhd \texttt{<}j_1\ \texttt{-}\ c_1\texttt{;}\ \dots\texttt{;}\ j_n\ \texttt{-}\ c_n\texttt{>} \longrightarrow \texttt{*}]\!]$$

We use the semantics of the inductive type, obtaining

$$\begin{aligned}&\texttt{<}j_1\ \texttt{:}\ j_1\texttt{;}\ \dots\texttt{;}\ j_n\ \texttt{:}\ j_n\texttt{>}\\ &\ \texttt{. F\_ii(<}i\ \texttt{:}\ \iota\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{>}\ \texttt{.}\ [\![g]\!]\texttt{)}\end{aligned}$$

We use equation (87) of Section 6.3.3

$$\begin{aligned}&\texttt{F\_ii(<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ \texttt{<}j_1\ \texttt{:}\ j_1\texttt{;}\ \dots\texttt{;}\ j_n\ \texttt{:}\ j_n\texttt{>>}\\ &\qquad \texttt{.}\ \texttt{<}i\ \texttt{:}\ \iota\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{>}\ \texttt{.}\ [\![g]\!]\texttt{)}\end{aligned}$$

Simplifying, we get

$$\texttt{F\_ii(<}i\ \texttt{:}\ \iota\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \dots\texttt{>}\ \texttt{.}\ [\![g]\!]\texttt{)}$$

which, by the semantics of the inductive type, is equal to

$$[\![\texttt{(ind}\ i\texttt{:}\ g\texttt{)} \rhd \texttt{<}i\ \texttt{-}\ c\texttt{;}\ j_1\ \texttt{-}\ c_1\texttt{;}\ \dots\texttt{;}\ j_n\ \texttt{-}\ c_n\texttt{>} \longrightarrow \texttt{*}]\!]$$

The correct typing of the result follows from Observation 8.3.5 and the fact that the source categories of the internal language projections ι and κ have changed upon the application of equation (87). □

We know by the definition of semantics in the previous subsection that type substitution is internalizable in the internal core language (using functor composition). The following theorem states that the results of type substitution are also expressible in the bare core language. In the proof of the theorem we see that the semantic type substitution strictly coincides with the standard textual type substitution of terms for projections of the bare core language (not for meta-variables as seen in rules and equations!). Such property is called referential transparency.

Theorem 8.3.8. *The language of types is referentially transparent with respect to type projections, that is:*

Let h and g be ground type terms of the bare core language. Let

$$h \rhd \texttt{<}j_1 \texttt{ - } c_1\texttt{;} \ \dots\texttt{;}\ j_n \texttt{ - } c_n\texttt{>} \to \texttt{*}$$

and

$$g \rhd \texttt{<}i \texttt{ - *;}\ j_1 \texttt{ - } c_1\texttt{;} \ \dots\texttt{;}\ j_n \texttt{ - } c_n\texttt{>} \to \texttt{*}$$

Then the ground type term f obtained by textually substituting h for i in g has the following source and target kinds

$$f \rhd \texttt{<}j_1 \texttt{ - } c_1\texttt{;} \ \dots\texttt{;}\ j_n \texttt{ - } c_n\texttt{>} \to \texttt{*}$$

and the ground equation $[\![f]\!] = [\![g[h/i]]\!]$ belongs to the equational theory of the internal core language (and so holds in the fixed categorical model, as well).

Proof. The proof is by induction on the structure of the derivation of typing for g. The following set of equations covers all the cases defining textual substitution. There are two cases for the inductive types and two for coinductive. The first case of each pair is trivial because the variable for which we substitute is equal to the bound variable of the operation and so no substitution inside is need.

$$\begin{array}{rcl}
[\![(f \texttt{ . } g)[h/i]]\!] & = & [\![f[h/i] \texttt{ . } g]\!] \\
[\![i[h/i]]\!] & = & [\![h]\!] \\
[\![j[h/i]]\!] & = & [\![j]\!] \\
[\![\texttt{\{}i_1 \texttt{ : } f_1\texttt{;} \ \dots\texttt{;}\ i_n \texttt{ : } f_n\texttt{\}}[h/i]]\!] & = & [\![\texttt{\{}i_1 \texttt{ : } f_1[h/i]\texttt{;} \ \dots\texttt{;}\ i_n \texttt{ : } f_n[h/i]\texttt{\}}]\!] \\
[\![\texttt{[`}i_1\ \ f_1\texttt{|} \ \dots\texttt{|`}i_n\ \ f_n\texttt{]}[h/i]]\!] & = & \texttt{[`}i_1\ \ f_1[h/i]\texttt{|} \ \dots\texttt{|`}i_n\ \ f_n[h/i]\texttt{]}]\!] \\
[\![(\texttt{ind } i\texttt{: } g)[h/i]]\!] & = & [\![\texttt{ind } i\texttt{: } g]\!] \\
[\![(\texttt{ind } j\texttt{: } g)[h/i]]\!] & = & [\![\texttt{ind } j\texttt{: } (g[h/i])]\!] \\
[\![(\texttt{coind } i\texttt{: } g)[h/i]]\!] & = & [\![\texttt{coind } i\texttt{: } g]\!] \\
[\![(\texttt{coind } j\texttt{: } g)[h/i]]\!] & = & [\![\texttt{coind } j\texttt{: } (g[h/i])]\!] \\
[\![(\texttt{\~{}}i_1\texttt{:}g_1 \ \dots\ \texttt{\~{}}i_n\texttt{:}g_n \texttt{ -> } f)[h/i]]\!] & = & [\![\texttt{\~{}}i_1\texttt{:}g_1[h/i] \ \dots\ \texttt{\~{}}i_n\texttt{:}g_n[h/i] \texttt{ -> } f[h/i]]\!]
\end{array}$$

The most difficult cases are for the (co)inductive types. Their proof is similar but harder than for the analogous cases in the proof of Lemma 8.3.7. The lemma itself is used in our argument. This time the hardest case is that with no duplication of the inductive label, as in the second equation for the inductive type. We start with the left hand side of the equation

$$[\![(\texttt{ind } j\texttt{: } g)[h/i]]\!]$$

use the definition of semantic substitution

$$\texttt{<}i \texttt{ : } [\![h]\!]\texttt{;}\ j_1 \texttt{ : } j_1\texttt{;} \ \dots\texttt{;}\ j_n \texttt{ : } j_n\texttt{>} \texttt{ . } [\![\texttt{ind } j\texttt{: } g]\!]$$

and the semantics of the inductive type

$$\texttt{<}i\ \texttt{:}\ [\![h]\!]\texttt{;}\ j_1\ \texttt{:}\ j_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ j_n\texttt{>}$$
$$\texttt{.}\ \texttt{F\_ii(<}j\ \texttt{:}\ \iota\texttt{;}\ i\ \texttt{:}\ \kappa\ \texttt{.}\ i\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ [\![g]\!]\texttt{)}$$

Then by equation (87) we obtain

$$\texttt{F\_ii(<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ \texttt{<}i\ \texttt{:}\ [\![h]\!]\texttt{;}\ j_1\ \texttt{:}\ j_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ j_n\texttt{>>}$$
$$\texttt{.}\ \texttt{<}j\ \texttt{:}\ \iota\texttt{;}\ i\ \texttt{:}\ \kappa\ \texttt{.}\ i\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ [\![g]\!]\texttt{)}$$

and simplifying we get

$$\texttt{F\_ii(<}j\ \texttt{:}\ \iota\texttt{;}\ i\ \texttt{:}\ \kappa\ \texttt{.}\ [\![h]\!]\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ [\![g]\!]\texttt{)}$$

Complicating, we obtain

$$\texttt{F\_ii(<}j\ \texttt{:}\ \iota\texttt{;}\ i\ \texttt{:}\ \kappa\ \texttt{.}\ i\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \ldots\texttt{>}$$
$$\texttt{.}\ \texttt{<}j\ \texttt{:}\ j\texttt{;}\ i\ \texttt{:}\ \texttt{<}i\ \texttt{:}\ i\texttt{;}\ j_1\ \texttt{:}\ j_1\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ [\![h]\!]\texttt{;}\ j_1\ \texttt{:}\ j_1\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ [\![g]\!]\texttt{)}$$

and we use Lemma 8.3.7

$$\texttt{F\_ii(<}j\ \texttt{:}\ \iota\texttt{;}\ i\ \texttt{:}\ \kappa\ \texttt{.}\ i\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \ldots\texttt{>}$$
$$\texttt{.}\ \texttt{<}j\ \texttt{:}\ j\texttt{;}\ i\ \texttt{:}\ [\![h]\!]\texttt{;}\ j_1\ \texttt{:}\ j_1\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ [\![g]\!]\texttt{)}$$

Then using the definition of $g[h/i]$ again

$$\texttt{F\_ii(<}j\ \texttt{:}\ \iota\texttt{;}\ i\ \texttt{:}\ \kappa\ \texttt{.}\ i\texttt{;}\ j_1\ \texttt{:}\ \kappa\ \texttt{.}\ j_1\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ [\![g[h/i]]\!]\texttt{)}$$

and using the semantics of the inductive type

$$[\![\texttt{ind}\ j\texttt{:}\ (g[h/i])]\!]$$

we arrive at the right hand side. □

We see that the operation $g[h/i]$ could be defined syntactically at the bare language level. However, this is against our principle of the fundamental role of the categorical internal language model, where the bare and user languages are treated just as syntactic extensions. In our compiler the operations corresponding to $g[h/i]$ are not performed at the bare language level, but at the internal language level using categorical composition, together with all the computation.

Corollary 8.3.9. *Every domain or codomain type inferred for a bare core language value can be expressed in the bare core language.*

Proof. Induction on the structure of an extended core language type term. The only nontrivial induction step is easily performed using Theorem 8.3.8. □

In the next section we will see that the whole syntactic domain of types of the bare language is accessible to the user. On the other hand the type reconstruction algorithm implemented in our prototype compiler is faithful to the typing rules presented above. Consequently, Corollary 8.3.9 implies that any type reconstructed for a Dule value is expressible in Dule as a closed type expression. This is a rare property among programming languages with sufficiently rich type structure (Standard ML, for example) allowed for mainly by the categorical generality of our type constructors.

Analogously as for the type language, one can prove referential transparency of value language expressions with respect to value projections. Again the most difficult cases in the proof would be the ones with double typing rules, but this time there is only one such construct — the fixed point expression.

Observation 8.3.10. *The language of values is referentially transparent with respect to value projections, that is*

$$[\![\{i = u;\ \dots\}\ .\ t]\!] \;=\; [\![v]\!]$$

where value term v is the result of textual substitution of u for i in t.

Syntactic value substitution is directly expressible in the value language so the above property has an elegant formulation. However, type substitution (type instantiation) is not expressible in the value language, just as it has not been in the type language, so a strict formulation of the following fact would involve internal core language, similarly as in the case of the type language (though the proof would be easier, since there are no type variable binding operations in the value language).

Observation 8.3.11. *The language of values is referentially transparent with respect to type projections.*

8.3.2 User core language

In this section we present the core language, as seen by the programmer. With respect to the bare core language there is some additional syntactic sugar that a parser alone can resolve.

Syntax

We present only the concrete syntax, since there is no abstract syntax for the user language. The parsing produces directly phrases of the abstract syntax of the bare core language. The details of the grammar, syntactic sugar and resolving ambiguities can be found in files describing the grammar for the universal parsing tools. The description of the LR parser for the OCaml Yacc [105] is in `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/parser.mly` an the description of the LL parser for CamlP4 [37] is in `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/ll_parser.ml`. The same conventions as for the bare language apply in the description of the grammar below.

```
FIELD-TYPE  ::= VALUE-LABEL : TYPE
CASE-TYPE   ::= ‘ CASE-LABEL TYPE
              | ‘ CASE-LABEL
PARAM-TYPE  ::= ~ VALUE-LABEL : TYPE
TYPE        ::= TYPE . TYPE
              | TYPE-LABEL
              | DULE-LABEL
              | { FIELD-TYPE ;...; FIELD-TYPE }
              | [ CASE-TYPE "|"..."|" CASE-TYPE ]
              | ind TYPE-LABEL : TYPE
              | coind TYPE-LABEL : TYPE
              | PARAM-TYPE...PARAM-TYPE -> TYPE
              | ( TYPE )

FIELD       ::= VALUE-LABEL = VALUE
              | VALUE-LABEL
CASE        ::= ‘ CASE-LABEL EMBEDDING
EMBEDDING   ::= VALUE
              | PATTERN -> VALUE
              | -> VALUE
ARGUMENT    ::= ~ VALUE-LABEL : VALUE
              | ~ VALUE-LABEL
DECLARATION ::= PATTERN = VALUE
              | rec PATTERN = VALUE
PARAM       ::= ~ VALUE-LABEL
              | ~ VALUE-LABEL : PATTERN
              | ~ ( VALUE-LABEL : TYPE )
PATTERN     ::= VALUE-LABEL : TYPE
              | ( VALUE-LABEL : TYPE )
              | VALUE-LABEL
              | _ % underscore
VALUE       ::= : TYPE
              | VALUE . VALUE
              | VALUE-LABEL
              | DULE-LABEL
              | { FIELD ;...; FIELD }
              | VALUE . ‘ CASE-LABEL
              | ‘ CASE-LABEL
              | [ CASE "|"..."|" CASE ]
              | map EMBEDDING
              | con
              | fold EMBEDDING
              | de
              | uncon
              | unfold EMBEDDING
              | unde
              | VALUE ARGUMENT ARGUMENT...ARGUMENT
              | VALUE ~
              | match VALUE with VALUE
              | let DECLARATION...DECLARATION in VALUE
              | fun PARAM...PARAM -> VALUE
              | VALUE ( ARGUMENT ARGUMENT...ARGUMENT )
              | if VALUE then VALUE else VALUE
              | assert VALUE in VALUE
              | fail
              | ( VALUE )
```

Notice that the fixpoint operation (called `fix`) of the bare core language is no longer accessible. Neither are accessible the kinds (categories) or the additional constructor corresponding to type substitution. On the other hand, three additional syntactic domains are provided, called `EMBEDDING`, `DECLARATION` and `PATTERN`.

Translation

Here we show how the syntactic sugar of the user core language translates to the constructions of the bare core language. The terms of all the domains of the user language translate to terms of the domain with the same name of the bare language. The exceptions are the domains `EMBEDDING` that translates to the `VALUE` domain of the bare language, `DECLARATION` that translates to `FIELD` domain and `PATTERN` that translates to `FIELD-TYPE`.

While describing the translation of a particular construction we assume that the subterms are already translated. A grammar may be thought of as an instance of an inductive type. Thus, speaking in the language of inductive types, we provide a map that could be used as an argument to the `fold` operation acting over a syntactic phrase, resulting in our translation.

The notational conventions are similar as before. This time, additionally "`:::`" and "`-->`" are used as meta-notation and the "empty-list" means a zero-length list of elements of the syntactic domain `PARAM`.

Question marks denote unknown types that are determined by the type reconstruction algorithm. The determined types may be distinct, for distinct occurrences of the question mark. Consequently, the translation of a phrase is not uniquely determined. A user language phrase is considered correct, if among its translations there is a term such that there exists a derivation of a bare language typing for the term. In our implementation, instead of generating many translations, we put a fresh type meta-variable in place of each occurrence of the question mark and let the type reconstruction algorithm reconstruct the full translation.

```
FIELD-TYPE  ::: i : f            --> i : f
CASE-TYPE   ::: ‘i f             --> ‘i f
              | ‘i               --> ‘i {}
PARAM-TYPE  ::: ~i:f             --> ~i:f
TYPE        ::: f . f            --> f . f
              | i                --> i
              | j                --> j
              | {fi1; ...; fin}  --> {fi1; ...; fin}
              | [c1"|" ..."|"cn]  --> [c1"|" ..."|"cn]
              | ind i: f         --> ind i: f
              | coind i: f       --> coind i: f
              | a1 ... an -> f   --> a1 ... an -> f
              | (f)              --> f

FIELD       ::: i = t            --> i = t
              | i                --> i = i
CASE        ::: ‘i t             --> ‘i t
EMBEDDING   ::: t                --> t
              | j : f -> t       --> fun ~it:(j:f) -> t
              | -> t             --> fun ~it:(_:?) -> t
ARGUMENT    ::: ~i:t             --> ~i:t
              | ~i               --> ~i:i
DECLARATION ::: j : f = t        --> j : f = t
              | rec j : f = t    --> j : f = fix j: t
PARAM       ::: ~i               --> ~i:(i:?)
              | ~i: j : f        --> ~i:(j:f)
              | ~(i:f)           --> ~i:(i:f)
PATTERN     ::: j : f            --> j : f
              | (j : f)          --> j : f
              | j                --> j : ?
              | _                --> _ : ?
VALUE       ::: : f              --> : f
              | t1 . t2          --> t1 . t2
              | i_t              --> i_t
              | i_d              --> i_d
              | {fi1; ...; fin}  --> {fi1;...; fin}
              | t . ‘i           --> t . ‘i
              | ‘i               --> {} . ‘i
              | [c1"|" ..."|"cn]  --> [c1"|" ..."|"cn]
              | map t            --> map t
              | con              --> con
              | fold t           --> fold t
              | de               --> de
              | uncon            --> uncon
              | unfold t         --> unfold t
              | unde             --> unde
              | t a0 a1 ... an   --> t a0 ... an
              | t ~              --> t empty-list
              | match t1 with t2 --> t2 ~it:t1
              | let i1 : f1 = t1 ... in : fn = tn in t
```

```
   --> (fun ~i1:(i1:f1) ... ~in:(in:fn) -> t)
         ~i1:t1 ... ~in:tn
| fun p1 ... pn -> t --> fun p1 ... pn -> t
| t (a0 a1 ... an)   --> t (a0 ... an)
| if t1 then t2 else t3
   --> [‘True -> fun ~it:(_:{}) -> t2
       |‘False -> fun ~it:(_:{}) -> t3] ~it:t1
| assert t1 in t2    --> assert t1 in t2
| fail
   --> assert ‘False in let rec it = it in it
| (t)                --> t
```

8.3.3 Conclusion

We have developed the bare and the user languages that make programming in our internal core language more comfortable. We have provided typing of the bare language terms and shown it is compatible with the typing of their semantics in the internal language (Observation 8.3.5, Observation 8.3.6). We have proved that the inferred type of every value is expressible, moreover expressible as a closed bare language type expression (Corollary 8.3.9). We have shown our language has no principal typing property (Observation 8.3.4) and sketched the finer points of our type reconstruction algorithm that infers minimal types for values.

Similarly as with the internal language, the typing of the bare language assigns to a value not only the codomain type but also the domain type. The domain type models the types of values in the environment. In this setting we model type variables and value variables as projections of the internal language. We also design a uniform way of supplying operands to coproduct, (co)inductive and mapping combinators. There is a lot of substitution notation in the typing of these combinators but there is no syntactic substitution in our languages. We have proved the composition in our categories faithfully and semantically implements the type substitution (Theorem 8.3.8) or, in other words, the language of types is referentially transparent. We have also shown referential transparency of the language of values (Observation 8.3.10).

Chapter 9

Extensions to the module language

Extensions to the module language are necessary to make the categorical system of modules we have developed in Chapter 5 usable in practice. Our categorical model of modules is simple, strict, with typing wired into the category and not assigned ad hoc to terms. This simplicity and strictness provides perfect ground for making fundamental design choices. The simplicity and generality also ensure that the model is extendible, as the addition of inductive modules, described below, witnesses.

For practical programming one needs an extensive array of operations, default conventions, syntactic sugar, programming tools, and these are best designed in an outer denotational layer of the semantics, not in the categorical kernel. While the internal module language, including inductive modules, is defined using direct semantics into SCM (the Simple Category of Modules of Section 5.1), the more user-friendly bare and environment-dependent languages are defined using a more relaxed denotational approach, based on the internal language.

9.1 I-Dule — module system with inductive modules

Under the modularization style specific to Dule, individual modules are numerous but small. Consequently many of the dependencies among values and types of a program carry across module boundaries. The dependencies are often mutual, hence for expressing them in the module language one needs a mechanism for defining mutually dependent modules. The inductive and coinductive module constructions enable relaying mutual dependencies in the core language to the level of modules and even imposing patterns of dependency that have no parallels in the core language (of which the last module cited in the Dule tutorial, Chapter 2, is the simplest example).

The type part of (co)inductive modules is constructed using (co)inductive types and the value part using fixpoints, but there are many possible variants of the implementation. They differ mainly in the ways the (co)inductive nature of the resulting types can be exploited at the level of the core language. One of the more interesting possibilities is using core language structured recursion over mutually dependent type parts of modules. In simple implementations, like the one we describe here (in Section 9.1.2), such mutual structured recursion is limited and requires manual preparation when defining the particular (co)inductive modules.

A welcome extension would be that at least `fold`, `unfold` and similar combinators automatically admit types resulting from the (co)inductive modules. Then the distinction between inductive and coinductive modules would be meaningful (and (co)inductive modular closure would no longer be observationally indistinguishable from a module fixpoint built using recursive types) while no restrictions on the pattern of mutual dependency among modules would be incurred. It is here that the explicit compound products of Section 4.3.1 would be necessary. We would also need to adapt some of the other combinators to non-basic kinds, in particular the `fold` and `unfold` combinators themselves.

In the implemented variant, accessing the (co)inductive module's types from the outside of modules is possible only by explicitly adding some suitable conversion operations to the signatures. An example of such an operation is `t2ind` discussed on page 54. These conversions are either less general or less efficient than a uniform solution would be. This is a promising area for future work.

9.1.1 Language

The following outline of the syntax and semantics of the (co)inductive modules focuses at the behavior at the module level and, therefore, is common for all variants that differ in the core language support for (co)inductive types resulting from modules.

Syntax

We extend the abstract syntax of L-Dule (Section 5.3) by the term constructors of the inductive and the coinductive module.

```
type dule =
  ...
  | M_Ind of sign IList.t * dule IList.t
  | M_CoInd of sign IList.t * dule IList.t
```

Semantics

These two new term constructors look very similar to the three term constructors that L-Dule introduces into the language. An important difference is that the inductive module constructions, unlike the three already described generalizations of the record of modules, are not definable in terms of the W-Dule language alone (Section 5.2). To give their formal semantics we have to use extensively the full internal core language. For example, in the version of the simple inductive modules described in detail in the next section we use core language (co)inductive types with constructors and destructors and a core language fixpoint operation on values.

Here we present an overview of the semantics in the same style as for W-Dule and L-Dule. In particular we provide domain and codomain `sign`-terms of the language W-Dule for each of the introduced constructors.

- `M_Ind`(lr, lm) : `S_Pp`$(lr) \to$ `S_Pp`(ls)
  written "`ind {`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}`"
  is an inductive module that groups mutually dependent modules lm, where the elements of lm are m_i : `S_Pp`$(lr_i) \to s_i$, each lr_i is a sublist of lr `@@` ls, lr and ls are disjoint, lr may be arbitrarily large and ls has the same labels as lm,

- `M_CoInd`(lr, lm) : `S_Pp`$(lr) \to$ `S_Pp`(ls)
  written "`coind {`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}`"
  is a coinductive module that groups mutually dependent modules lm, where the elements of lm are m_i : `S_Pp`$(lr_i) \to s_i$, each lr_i is a sublist of lr `@@` ls, lr and ls are disjoint, lr may be arbitrarily large and ls has the same labels as lm.

The relations among domain and codomain signatures of the inductive and coinductive module constructors and their operands are the same as for the `M_Link` constructor of Section 5.2. The only difference is that here cyclic dependencies among modules are permitted. While the linking constructor unifies the domains of the modules by composing, the inductive constructor takes care of the parameters outside of lr by using inductive types capturing the mutual type dependencies and recursively defined values. Examples of inductive modules (written with some syntactic sugar) are given in Section 2.3.3.

The (co)inductive modules are separately compilable in the following sense. Each of the module operands of the (co)inductive module constructor can be compiled separately from the other module operands, because its compilation depends only on the codomain signatures of all the operands and not on their bodies. The whole (co)inductive module construction with the product codomain obviously depends on all the compiled operand modules. Every composition of the construction with an individual projection (representing an individual module with all the recursive dependencies satisfied) uses the result of the compilation of the (co)inductive module construction and therefore depends on the results of compilation of all the operand modules. Consequently recompilation of any of the operand modules does not require recompilation of other operands, but triggers recompilation of the (co)inductive module construction and so of any of the compositions with a projection that are present in the program.

These are the profiles of the combinators associated to the two new terms constructors. The definitions of the simplest variant of these constructors is given in the next section.

```
  val m_Ind : Sign.t IList.t -> Dule.t IList.t ->
    [‘OK of Dule.t|‘Error of string]
  val m_CoInd : Sign.t IList.t -> Dule.t IList.t ->
    [‘OK of Dule.t|‘Error of string]
```

9.1.2 Simple inductive modules

The precise definition of the simple inductive module operations is given in module `SemIDule` included in file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/mod_back.ml` and is presented below with comments. There is also a slightly more modularized version written in the Dule language itself, see Appendix C.2. The definitions are comparatively long and, perhaps, not yet structured properly. The work on other variants of inductive modules should provide o broader perspective and help with the presentation.

General structure

The definition of the inductive module operations has been guided from the very beginning by the module invariant of Section 5.2.2. Nevertheless, the proof of the module invariant in this case consists of long, tedious and technical tracing of the source code and is here omitted. While discussing key points of the source code we will see how the module invariant influences their construction.

Observation 9.1.1. *The module operations* `m_Ind` *and* `m_CoInd` *satisfy the properties stated as the module invariant in Lemma 5.2.3.*

Here is the OCaml definition of the two semantic combinators. The combinators are dual, so the analogous parts of their definitions will always be presented in pairs.

```
let m_Ind = m_XInd m_Ind_ordinary


let m_CoInd = m_XInd m_CoInd_ordinary
```

The auxiliary combinators m_Ind_ordinary and m_CoInd_ordinary perform (co)inductive closure of a single self-referencing module. The auxiliary operation m_XInd, to be defined later on, extends the combinators to operate on a family of modules, as required by the arity of (co)inductive module constructors.

Closure of a single module

Arguments to the main (co)inductive module combinators are families of modules with mutual references. Here we describe the main point of inductive modules: performing the module recursive closure after the family of modules is collapsed to a single module with a product domain and codomain. Note that the recursive closure is close to, but not exactly, a module fixpoint. Application of a module to its closure is not equal, but only observationally indistinguishable (where types defined inside the closed module are not observable), to the closure itself, due to the use of (co)inductive types instead of recursive types. However, the difference only applies to module's types and is further blurred when the types are seen as abstract from within other modules.

In the domain of the module to be closed there is a distinguished component called AtIndex.atr consisting of all codomain specifications of the modules and used to represent the mutual references among the modules. Whenever an entity inside the single module refers to the component AtIndex.atr, this represents a dependency on the module's result. We would like to replace the references to AtIndex.atr by the actual recursively closed code. This is done in single-module (co)inductive combinators m_Ind_ordinary and m_CoInd_ordinary defined below.

```
let m_Ind_ordinary =
  m_XInd_ordinary
    ToolIDule.repair_ii
    ToolIDule.close_type_ii
    ToolIDule.fix_value

let m_CoInd_ordinary =
  m_XInd_ordinary
    ToolIDule.repair_tt
    ToolIDule.close_type_tt
    ToolIDule.fix_value
```

Both of the combinators are implemented with the use of an auxiliary combinator m_XInd_ordinary defined as follows. The notions of local and context types have been introduced in Definition 5.2.4. The auxiliary function unpp obtains the indexed list of components of a product — the same list as in the case of function unpp_ok discussed in Section 5.1.1.

```
let m_XInd_ordinary repair close_type fix_value  m =
  (* the result is : S_Pp lr -> s' *)
  (* m : S_Pp lrr -> s,
     lrr = cons (AtIndex.atr, s) lr,
     [AtIndex.atr] not in [lr] nor [lb] (local types of [s]),
     labels of [lr] and [lb] are disjoint,
     labels of context types of [r_i] are in [lr]
     labels of context types of [s] are the labels of [lrr],
     labels of context types of [s'] are the labels of [lr],
     [s'] depends on its own local types
     instead of on [AtIndex.atr]
   *)
  let s = Dule.codomain m in
  let h = Sign.s2f s in
  let f = Dule.type_part m in (* : c -> e *)
  let t = Dule.value_part m in
    (* : r -> f_COMP f h, r = S_Pp lrr *)
(* analizing f:*)
  let c = SrcFCore.src f in (* = src (S_Pp lrr) *)
  let lc = unPP c in (* = cons (AtIndex.atr, b) la *)
  let e = SrcFCore.trg f in (* = src s *)
  let le = unPP e in (* = lb @@ lc *)
  let b = find AtIndex.atr lc in
  let lb = unPP b in (* normally, these have labels of [ls] *)
  let la = remove AtIndex.atr lc in
    (* [la] labels = [lr] labels *)
  let a = c_PP la in
(* cutting f: *)
  let pib = imap (f_PR le) lb in
  let fc = f_COMP f (f_RECORD e pib) in (* : c -> b *)
(* close_type f: *)
  let fi = close_type fc b la in (* : a -> b *)
(* instantiating t: *)
  let lapr = imap (f_PR la) la in
  let lrfa = cons (AtIndex.atr, fi) lapr in
  let rf = f_RECORD a lrfa in (* : a -> c *)
  let rft = t_FT rf t in
    (* : f_COMP rf r -> f_COMP rf (f_COMP f h) *)
(* repair t: *)
  let lrta = vmap (fun f -> t_id f) lrfa in
  let lbpr = imap (f_PR lb) lb in
  let adc = repair lrta lbpr fi h a in
  let tad = t_comp rft adc in (* : g -> tb = f_COMP fif h *)
(* analizing t: *)
  let g = DomFCore.dom tad in (* = f_COMP rf r *)
  let lg = unpp g in (* = cons (AtIndex.atr, tb) lta *)
  let tb = find AtIndex.atr lg in
    (* normally, the [ls] labels *)
  let lta = remove AtIndex.atr lg in
    (* [lta] labels = [lr] labels *)
(* fix_value t:*)
  let t' = fix_value tad tb lta a in (* : ta -> tb *)
(* changing s to s': *)
  let ld = remove AtIndex.atr le in
  let d = c_PP ld in
  let pid = imap (f_PR ld) ld in
  let pib = imap (fun i -> find i pid) lb in
  let reb = f_RECORD d pib in
  let pir = cons (AtIndex.atr, reb) pid in
  let red = f_RECORD d pir in (* d -> e *)
  let h' = f_COMP red h in
  let s' = Sign.f2s h' in
(* extending f to s': *)
  let pifb = vmap (fun pr -> f_COMP fi pr) lbpr in
  let f' = f_RECORD a (pifb @@ lapr) in (* : a -> d *)
  Dule.pack (f', t', s')
```

The combinator takes three additional arguments before it accepts the module `m` to be closed. The additional argument `fix_type` is a function used for closing the type part of the module `m`. The function cannot produce strictly a fixpoint of a type, because our core language does not feature recursive types. Instead, either an inductive or a coinductive closure of the type part is generated.

After the use of `close_type` we get rid of the reference of values of `m` to the types of `m` by instantiating the value part with the just closed type part. Next we would like to close the value part of the module using the function `fix_value` that produces real fixpoints of core language values. However, the codomain of the instantiated value part is too complex and thus in general incompatible with the domain, which prevents the use of value fixpoint.

Let us try to understand this problem on a simplified example. Consider a module $\mathtt{m} : \mathtt{s} \to \mathtt{s}$, where the domain `s` represents a reference of the module to itself. We would like to give a semantics to this module using the inductive module construction. Let the module consists of a type part `f` and a value part `t`.

Recall that this implies

$$\mathtt{t} : \mathtt{s} \to \mathtt{f\_COMP\ f\ s}$$

Let the inductively closed type part of `m` be `fi` = `f_ii f`, where `f_ii` is the inductive type combinator of the core language (see Section 6.3). The instantiation of `t` with `fi`, needed to eliminate the reference of `t` to the types of `s`, is just a multiplication of transformation `t` by the functor `fi` from the left; `rft` = `t_FT fi t`. We have:

$$\mathtt{rft} : \mathtt{f\_COMP\ fi\ s} \to \mathtt{f\_COMP\ fi\ (f\_COMP\ f\ s)}$$

Now we would like to take a fixpoint of `rft`, but to apply the fixpoint operator we need (simplifying even more) a transformation with the same domain and codomain. The transformation `rft` fails to meet such a criterion, which identifies the problem.

Moreover, since the closed type part `fi` is not a true type fixpoint of the type construction performed by the module `m`, the problem cannot be resolved by using the equality `f_COMP fi f` = `fi` or a similar simple coercion. However, `fi` is an inductive type, and for inductive types there are morphisms — in fact: isomorphisms — with domain `f_COMP fi f` and codomain `fi`. The isomorphisms are inductive constructors.

Continuing with the simplified example, what we need is to use the inductive type constructor `t_con`, which has the following domain and codomain

$$\mathtt{t\_con\ f} : \mathtt{f\_COMP\ fi\ f} \to \mathtt{fi}$$

and compose it with `rft`. For this we need to multiply the constructor by `s` obtaining

$$\mathtt{adc} = \mathtt{t\_TF\ (t\_con\ f)\ s}$$

with the type

$$\mathtt{adc} : \mathtt{f\_COMP\ (f\_COMP\ (fi\ f))\ s} \to \mathtt{f\_COMP\ fi\ s}$$

In the end we vertically compose the transformations getting

$$\mathtt{tad} = \mathtt{t\_comp\ rft\ adc}$$

which has the required domain and codomain

$$\mathtt{tad} : \mathtt{f\_COMP\ fi\ s} \to \mathtt{f\_COMP\ fi\ s}$$

Now the fixpoint operation can be applied successfully to the value `tad`.

The transformation `adc` is usually much harder to construct than in our simplified example. It is generated by the first additional argument to `m_XInd_ordinary`, called `repair`. The operation `repair` is the only place in the semantic definition of our module system where we use the horizontal composition of the core language (more precisely a multiplication of a transformation by a functor from the right). Consequently, this is the place where we stumble upon the variance problems incurred by exponent types, see Section 8.1.4.

Repairing the value part

The two variants of the `repair` operation are used for the inductive and coinductive modules, respectively.

```
let repair_ii = XToolIDule.repair unii t_con t_de

let repair_tt = XToolIDule.repair untt t_uncon t_unde
```

The generic code of the `repair` operation first depends on three basic operations of the inductive or coinductive type and then on five arguments consisting of core language entities. As explained in Section 8.1.4, for some transformations the horizontal composition is not valid. Here we simulate horizontal composition using an ad hoc auxiliary combinator `t_TF_coco`. Strictly speaking, while for a functor not containing exponent `t_TF_coco` produces the true multiplication by the functor from the right, in general it produces a different transformation, but with a simpler typing than the true multiplication. See Appendix A.2.4 for details.

To make inductive modules possible, a core language has to feature either a horizontal composition (f-Core has one, though restricted) or enough mapping combinators to enable simulation (such as in the definition of `t_TF_coco` in the source code of the Dule compiler, which takes advantage of the fact that we work in a reachable model and we know how to rewrite mapping for reachable arguments). We believe the languages with advanced structured recursion mechanisms, such as Charity [54] and Functorial ML [88] should be rich enough to fall into the latter category. As a side note, both the languages just mentioned have some restrictions on the use of the exponent types, just like our language.

Simplifying, the function `repair` multiplies the constructor of the (co)inductive type resulting from closure of the type part of module `m` by the codomain signature of `m`. For various technical reasons the definition also involves projections, destructors, etc.

```
let repair unii t_con t_de  lrta lbpr fi h a =
  let unf = unii fi in (* : c_PP (coi b a) -> b *)
  let fcon = t_con unf in
  let pifc = vmap (fun pr -> t_TF fcon pr) lbpr in
  let tc = t_RECORD a (lrta @@ pifc) in
  (* we do not use [t_TF tc h], because it is
     not valid for [F_ee] and because we want
     simple [dom] and [cod] of the result *)
  let fde = t_de unf in
  let pifd = vmap (fun pr -> t_TF fde pr) lbpr in
  let td = t_RECORD a (lrta @@ pifd) in
  t_TF_coco (tc, td) h
```

Closure of types and values

Similarly as in the case of the `repair` operation, there are two variants of the `close_type` operation.

```
let close_type_ii = XToolIDule.close_type f_ii Cat.coi

let close_type_tt = XToolIDule.close_type f_tt Cat.coi
```

The (co)inductive functor combinators used here are slightly generalized versions of the combinators described in Section 6.3. They are used to perform a closure of a tuple of mutually dependent types and therefore have to be able to take operands with target more complex than `*`. Similarly the (co)inductive constructors and destructors used in `repair` are generalized. No other (co)inductive operations are needed in this variant and so the rewriting of the generalized terms remains essentially unchanged. In the variant with folding over modules, we would need to generalize the folding and mapping operations, too. Such a change would necessitate additional rewriting rules and would complicate most proofs of facts about the rewriting system and the equational theories.

The operation `close_type` is parameterized by the very combinator of the inductive or the coinductive type. The remaining arguments of `close_type` are a functor and some categories. The main purpose of the operation is to apply the (co)inductive combinator to the functor. Before that, several auxiliary operations have to be performed to place the important domain components at appropriate positions.

```
let close_type f_ii coi  f b la =
  (* f : cons (AtIndex.atr, b) la -> b *)
  let lapr = imap (f_PR la) la in
  let a = c_PP la in
  let lba = coi b a in
  let prba = f_PR lba AtIndex.atk in
  let lpr = vmap (fun pr -> f_COMP prba pr) lapr in
  let lcopr = cons (AtIndex.atr, f_PR lba AtIndex.atj) lpr in
  let rba = f_RECORD (c_PP lba) lcopr in
  let fca = f_COMP rba f in
  f_ii fca (* : a -> b *)
```

The operation `fix_value` is not parameterized, because closing the values is performed using a real fixpoint, which cooperates equally well with inductive and coinductive constructions. Otherwise the structure of the code is completely analogous to that of `close_type`.

```
let fix_value t tb lta a =
  (* t : cons (AtIndex.atr, tb) lta -> tb *)
  let ltapr = imap (t_pr lta) lta in
  let ta = f_pp a lta in
  let lba = Funct.cof tb ta in
  let prba = t_pr lba AtIndex.ate in
  let lpr = vmap (fun pr -> t_comp prba pr) ltapr in
  let lcopr = cons (AtIndex.atr, t_pr lba AtIndex.atd) lpr in
  let rba = t_record (f_pp a lba) lcopr in
  let fca = t_comp rba t in
  tl_fix fca (* : ta -> tb *)
```

Closure of multiple modules

Both the main combinators of the (co)inductive modules are implemented with the use of an auxiliary combinator `m_XInd`. The combinator prepares the special domain component `AtIndex.atr` consisting of all codomain specifications of the operand modules and appends it to the list of auxiliary parameters `lr`. Then the operand modules are joined together, the combinator `m_Ind_ordinary` is used to perform the (co)inductive closure and the result is trimmed to the desired signature.

```
let m_XInd m_Ind_ordinary lr lm = (* : S_Pp lr -> S_Pp ls *)
  (* m_i : S_Pp lr_i -> s_i,
     [lr_i} contained in [lr @@ ls],
     [s_i] has labels of context types inside [lr @@ ls],
     [AtIndex.atr] not in [lr] nor [ls],
     labels of context types of [r_i] are in [lr]
   *)
  let ls = vmap Dule.codomain lm in
  (match SemWSign.s_Pp ls with
  |‘OK s ->
      let lrr = cons (AtIndex.atr, s) lr in
      (match m_Ripcord lrr lm with (* : S_ssr -> S_Pp ls’ *)
      |‘OK m ->
          let mind =
            m_Ind_ordinary m in (* : S_Pp lr -> S_Pp ls’’ *)
          SemWDule.m_Trim mind s
      |‘Error er -> ‘Error er)
  |‘Error er -> ‘Error er)
```

The auxiliary combinator `m_Ripcord` represents the indexed list of modules `lm`, with mutual dependencies represented as references to various domain components, as a single module with references to the distinguished domain component `AtIndex.atr`.

```
let m_Ripcord lrr lm = (* : S_Pp lrr -> S_Pp ls' *)
  (* m_i : S_Pp lr_i -> s_i,
     [lr_i} contained in [lr @@ ls],
     lrr = cons (AtIndex.atr, S_Pp ls) lr
     [s_i] has labels of context types inside [lr @@ ls],
     [AtIndex.atr] not in [ls], too,
     labels of context types of [r_i] are in [lr]
     s'_i = S_Ww (re_i, s_i),
     where domain of each [re_i] is [S_Pp lr],
     so the labels of context types of [S_Pp ls']
     are exactly the labels of [lrr]
   *)
  (match SemWSign.s_Pp lrr with
  |`OK rr ->
      (match imap1ok (SemWDule.m_Pr lrr) lrr with
      |`OK lprr ->
          let ls = vmap Dule.codomain lm in
          (match imap1ok (SemWDule.m_Pr ls) ls with
          |`OK lps ->
              let prr = find AtIndex.atr lprr in
              let lpc = vmap (fun pr ->
                SemWDule.m_Comp prr pr) lps in
              let lpr = remove AtIndex.atr lprr in
              let lprs = lpc @@ lpr in
              let prm m =
                let lf = Sign.value_part (Dule.domain m) in
                let lmf = imap (fun i -> find i lprs) lf in
                (match SemWDule.m_Record rr lmf with
                |`OK re ->
                    SemWDule.m_Inst re m
                |`Error er -> `Error er)
              in
              (match vmap1ok prm lm with
              |`OK lm ->
                  SemWDule.m_Record rr lm
              |`Error er -> `Error er)
          |`Error er -> `Error er)
      |`Error er -> `Error er)
  |`Error er -> `Error er)
```

The definition of the combinator `m_Ripcord` is somewhat close to the definitions of the three combinators of L-Dule. In particular, essentially only the combinators of W-Dule are used in the definition. No mechanisms of the core language are assumed nor used, in contrast to the auxiliary functions described in previous sections.

9.1.3 Conclusion

We have extended our module language with the ability to define mutually dependent modules, which is crucial for our programming style with numerous small modules. The mutual dependencies among the core language entities scattered in separate modules are captured by the operations of the module language. The main module operation `m_XInd_ordinary` performs (up to observational indistinguishability) a fixpoint closure of a single module. Several mutually dependent modules are reduced to the module fixpoint by a transformation described solely in the module language W-Dule. The (co)inductive module operations have the same operands and their typing relations as the operation of linking modules and, similarly, preserve full separate compilation of the operand modules.

Every step of the recursive closure of a single module is expressed in the language of f-Core and so is fully typed. By careful construction we have ensured that the resulting module satisfies the module language invariants (Observation 9.1.1). The approach with full typing enabled us to pinpoint the problems caused by the interaction between (co)inductively closing types and multiplicating values by arbitrary functor from the right (for example by the exponent functor, which is not needed for our construction but is present in the language f-Core and adds to the usability of modules). We have overcome the problems without resorting to recursive types and any other forms of identifying similar types or bypassing type verification.

Despite the use of inductive types, the "inductive" modules in the current implementation are, in fact, more like "recursive" modules. Only the constructor and destructor of the (co)inductive structure are used, and the recursion is performed using general recursion, as opposed to structured recursion. There are also no tools to access the inductive structure on types, generated by the inductive modules, from the outside of the modules. Consequently, there is even no observable difference between the inductive and the coinductive module construction in this version.

An advantage of the use of general recursion is that every pattern of mutual dependency between module's values is accepted. Therefore, the computation of the resulting values is not guaranteed to terminate, but there are no restrictions on the form of recursion. Very informally, we expect that any such restrictions, allowing us to perform the recursive closure using structured recursion, would turn out to be either impractical or undecidable. However, inductive combinators traversing types resulting from inductive modules seem a plausible future extension, made possible by our inductive setting.

9.2 Extended notation for modules

The internal module language, while semantically powerful, is awkward to use, even in its concrete syntax. In this section we will design, in steps, a much more comfortable and succinct module language based on the semantics of the internal module language.

9.2.1 Bare module language

The bare module language is based on the internal module language of Dule as developed throughout the thesis and summarized in Appendix A.2. While the internal module language has the syntactic domain of signatures that are used as domains or codomains of modules, here we see specifications that are complete profiles of modules. Specifications describe both the parameters and the result of the module, and the description of the result may depend on the parameters.

Syntacticly, the bare module language is, more or less, a small subset of the Dule module language accessible to the user. The user language is laid out in Section 9.2.4 and is translated to the bare language through an intermediate module language, called environment dependent module language, described in Section 9.2.3.

Syntax

The same conventions as for the core languages (see Section 8.3.1) apply in the description of the bare module language grammar below and in all the other grammars for the module languages. Just as for the bare core language, we give only the concrete syntax. The abstract syntax, used in our compiler code, can be found in module `BDule` contained in file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/mod_front.ml`. We use the syntactic domains `TYPE` and `VALUE` and three of the four domains of labels from the bare core language.

```
FIELD-SP   ::= DULE-LABEL : SP
PARAM-SP   ::= ~ DULE-LABEL : SP
BB-ITEM    ::= type TYPE-LABEL
             | value VALUE-LABEL : TYPE
SP         ::= { FIELD-SP ;...; FIELD-SP }         % product
             | {{ FIELD-SP ;...; FIELD-SP }}       % double product
             | PARAM-SP...PARAM-SP -> SP           % parameterized
             | sig BB-ITEM...BB-ITEM end           % base specification
             | sig {{ FIELD-SP ;...; FIELD-SP }}     % --- with explicit
                  BB-ITEM...BB-ITEM end              %     parameters
             | DULE "|" SP                           % specialized
             | SP with DULE                          % --- with trimming
             | rec DULE-LABEL
                 { FIELD-SP ;...; FIELD-SP}          % recursive
             | [ SP ]                                % isolated

FIELD-DULE ::= DULE-LABEL = DULE
BASE-ITEM  ::= type TYPE-LABEL = TYPE
             | value VALUE-LABEL = VALUE
DULE       ::= DULE-LABEL                            % projection
             | { FIELD-DULE ;...; FIELD-DULE }       % record
             | {{ FIELD-DULE ;...; FIELD-DULE }}     % double record
             | :: SP DULE                            % explicit
             | struct BASE-ITEM...BASE-ITEM end      % base module
             | DULE "|" DULE                         % instantiation
             | DULE with DULE                        % --- with trimming
             | DULE :> SP                            % trimmed module
             | link { FIELD-SP ;...; FIELD-SP ,
                    FIELD-DULE ;...; FIELD-DULE ,
                    FIELD-DULE ;...; FIELD-DULE }    % to be linked
             | ind { FIELD-DULE ;...; FIELD-DULE }   % inductive
             | coind { FIELD-DULE ;...; FIELD-DULE } % coinductive
             | [ DULE ]                              % isolated
```

Saturation of specifications

The relation of saturation of specifications (“$\Rightarrow$”) and the relation of typing of modules (“$\triangleright$”) will be mutually recursive. Let us fix an arbitrary categorical model of the internal core language. The semantics $[\![\_]\!]_B$ of the bare module language in the categorical model, defined in section **Semantics** below, is meant to be applied to saturated and typed terms. Both the saturation and the typing depend on the semantics of the bare module language. The saturation, typing and semantics are partial relations, but the partiality does not come from the mutual dependency of semantics and typing. The semantics of a module depends only on the semantics of its proper subterms and the terms found in its typing. The typing, in turn, applies the semantic function only to already typed terms. The following property of the set of derivation rules (to be given below) for relation “$\triangleright$” implies that typing side conditions can be checked after the whole derivation tree is built. This enables simplification and improvement in efficiency for the specification reconstruction algorithm.

Observation 9.2.1. *Let us consider a node in a tree representing a derivation of a specification for a module (notice that, due to the mutual dependency of saturation and typing, the node may correspond to a rule of saturation of specifications). The rule represented by the node can have side conditions referring to* *the semantics of terms. If so, then the derivations needed for the computation of the semantics of the term are already represented by subtrees of the node.*

The dependency of saturation upon the semantics of the bare module language is seen in rule (294), which employs the $[\![\_]\!]_B$ function. In short, the semantic function $[\![\_]\!]_B$ is the composition of the semantics of the bare module language in the internal module language and the semantics of the internal module language in the fixed categorical model, developed throughout the thesis. The semantic function is used in rule (294) to allow the result of saturation to be changed to any equivalent counterpart. The saturation of specifications and the typing of modules use also the semantics $[\![\_]\!]_b$ of bare core language in the categorical model, defined in Section 8.3.1. For simplicity, we assume the bare core language typing of all bare core language values is given and fixed. The semantics of the bare core language is used together with the semantics of the bare module language in rules (289) and (300) to check the typing of core language components of base specifications and base modules.

Notice that the saturation and typing rules are not deterministic. There are two important, though perhaps not immediately apparent, goals that are achieved by means of the nondeterminism. The first is that every base specification is changed to the special syntactic form of the base specification with explicit parameters. The second is that base modules are assigned appropriate "guessed" specifications, written using the notation "`:: p m`" of explicitly specified module. If a base specification or a base module are isolated in the program then the "guessed" components are determined to be trivial (through rules (293) and (307)).

In this subsection we present the rules for saturation of specifications (which inhabit the syntactic domain `SP`). The rules, although fewer than the rules for typing of modules, are somewhat less standard. Apart from introducing and managing nondeterminism, the saturation consists of flattening parameterized specifications, uncovering the parameter parts of the other specifications and expressing some compound specifications using simpler ones. The saturation relation is denoted by $\Rightarrow$. For studying the rules defining saturation relation, a few formal statements concerning the relation may prove useful.

Definition 9.2.2. *If $p \Rightarrow p'$ then p' is called a saturation of p.*

Definition 9.2.3. *A specification p is called saturated if $p \Rightarrow p$.*

Note that for a saturated specification p there may exist q such that $[\![q]\!]_B \neq [\![p]\!]_B$ and $p \Rightarrow q$, because the saturation relation is not deterministic. A saturated specification is always a parameterized specification, where all arbitrarily nested specification subterms, except those nested in module subterms, are product or base or specialized specifications.

We leave the following general property without proof, which might proceed by easy induction over the derivation using the rules below. If the saturation relation was deterministic, the property would imply that the saturation operation is idempotent.

Observation 9.2.4. *If $p \Rightarrow p'$ then p' is saturated.*

We define a partial order, pronounced "less or equally detailed", on specifications and on indexed lists of specifications. In the absence of variables, it serves a similar purpose as the usual notion of more or equally general typing.

Definition 9.2.5. *A list of specifications lp is said to be less or equally detailed than lq, if for each element p of lp at index k, the element k of lq exists and is less or equally detailed than p. A specification term p is less or equally detailed than q if $[\![p]\!]_B$ and $[\![q]\!]_B$ are equal or p is the empty product specification* `{}` *or p and q have the same root constructor and the corresponding core language subterms have equal semantics and the specification or specification list subterms of p are less or equally detailed than the corresponding subterms of q.*

The conjecture below implies that rule (293) is well defined. Its proof is beyond the scope of our thesis, as it involves a similar proposition for the module typing and requires a considerable extension to the formalism including most of the notions related to the specification reconstruction algorithm.

Conjecture 9.2.6. *For any specification p, if a saturation of p exists then the least detailed saturation of p exists.*

Example. Specification

```
sig type u value v : u.t end
```

(notice that label `u` appears twice) has no saturation. On the other hand, specification

```
sig type u value v : T.t end
```

has many saturations, in particular the least detailed one:

```
~T : sig {{}} type t end ->
  sig {{T : sig {{}} type t end}}
    type u value v : T.t
  end
```

but also others:

```
~Arg : sig {{}} end
~T : sig {{Arg : sig {{}} end}} type u type t end ->
  sig {{Arg : sig {{}} end;
        T : sig {{Arg : sig {{}} end}} type u type t end}}
    type u value v : T.t
  end
```

where the last one is the least detailed saturation of

```
~T : ~Arg : sig end ->
       sig type u type t end
-> sig
     type u value v : T.t
   end
```

The rules presented in this section differ from the rules implied by the implementation of our saturation algorithm contained in the file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/mod_front.ml`. Auxiliary functions, like those used heavily in the implementation, are cumbersome when embedded inside the typing formalism and their precise definitions are long and obscured by details. Instead of most of the functions, here we add a few additional syntactic constructors and some corresponding rules. However, we believe the rules implied by the implementation have the same semantics as the rules presented here.

In this and the next section we use the identifier a (for "additional parameter") together with the usual p and q as variables denoting specifications. The nondeterminism of saturation relation allows one to derive saturations with arbitrary (non-clashing) additional parameters (k : a'; ...) in each rule, including rule (293) for isolated specifications. The isolated specifications as well as isolated modules are used in Section 9.2.3 to mark "exit points" of a type-checker — subterms of a whole program that should be assigned types separately of the others.

Rule (285) introduces nondeterminism with specifications $a', \dots$ that are result parts of arbitrary saturated specifications. The rule is sound because extending a module's typing with superfluous parameters preserves type-correctness (however, in case of a base specification its "context signature" has to be extended accordingly). The convention that the indexed lists have no duplicated labels enforces textual identity (and not equality up to the semantic equivalence) of any parameter specifications appearing with the same label at several parameter lists of the same rule. Rule (294) relaxes these requirements allowing semantic equivalence to be taken into account.

Rule (287) shows how to add parameters using the construction of explicitly parameterized specification. Note that if the specification q' in rule (287) is isolated (is of the form `[`q`]`) the additional parameters do not have any effect on the result part of the specification, unlike in the case when q' is a base specifications. Rule (290) is the only one that directly involves the module typing relation. Since specifications derived for modules are guaranteed to be saturated, the result part of the specification for m can be directly compared with the parameter part of saturation of q. Rule (292) is discussed after all the rules are presented.

$$\frac{\begin{array}{c} a \Rightarrow\ \ldots\ \texttt{->}\ a' \quad \cdots \\ p \Rightarrow \texttt{\~{}}i_1\texttt{:}p'_1\ \ldots\ \texttt{\~{}}i_n\texttt{:}p'_n\ \texttt{->}\ p' \quad \cdots \end{array}}{\begin{array}{c} \texttt{\{}i\ \texttt{:}\ p\texttt{;}\ \ldots\texttt{\}} \Rightarrow \\ \texttt{\~{}}k\texttt{:}a'\ \ldots\ \texttt{\~{}}i_1\texttt{:}p'_1\ \ldots\ \texttt{\~{}}i_n\texttt{:}p'_n\ \ldots\ \texttt{->}\ \texttt{\{}i\ \texttt{:}\ p'\texttt{;}\ \ldots\texttt{\}} \end{array}} \tag{285}$$

$$\frac{\texttt{\{}i\ \texttt{:}\ p\texttt{;}\ \ldots\texttt{\}} \Rightarrow \texttt{\~{}}k\texttt{:}a'\ \ldots\ \texttt{->}\ \texttt{\{}i\ \texttt{:}\ p'\texttt{;}\ \ldots\texttt{\}}}{\begin{array}{c} \texttt{\{\{}i\ \texttt{:}\ p\texttt{;}\ \ldots\texttt{\}\}} \Rightarrow \\ \texttt{\~{}}k\texttt{:}a'\ \ldots\ \texttt{->}\ \texttt{\{}k\ \texttt{:}\ a'\texttt{;}\ \ldots\texttt{;}\ i\ \texttt{:}\ p'\texttt{;}\ \ldots\texttt{\}} \end{array}} \tag{286}$$

$$\frac{\begin{array}{c} \texttt{\{\{}i\ \texttt{:}\ p\texttt{;}\ \ldots\texttt{\}\}} \Rightarrow\ \ldots\ \texttt{->}\ \texttt{\{}k\ \texttt{:}\ a'\texttt{;}\ \ldots\texttt{\}} \\ q \Rightarrow \texttt{\~{}}k\texttt{:}a'\ \ldots\ \texttt{->}\ q' \end{array}}{\texttt{\~{}}i\texttt{:}p\ \ldots\ \texttt{->}\ q \Rightarrow \texttt{\~{}}k\texttt{:}a'\ \ldots\ \texttt{->}\ q'} \tag{287}$$

$$\frac{\texttt{sig \{\{\}\} type}\ i\ \ldots\ \texttt{value}\ j\ \texttt{:}\ h\ \ldots\ \texttt{end} \Rightarrow q}{\texttt{sig type}\ i\ \ldots\ \texttt{value}\ j\ \texttt{:}\ h\ \ldots\ \texttt{end} \Rightarrow q} \tag{288}$$

$$\frac{\begin{array}{c} \texttt{\{\{}i_1\ \texttt{:}\ p_1\texttt{;}\ \ldots\texttt{\}\}} \Rightarrow\ \ldots\ \texttt{->}\ \texttt{\{}k\ \texttt{:}\ a'\texttt{;}\ \ldots\texttt{\}} \\ [\![\texttt{\{}k\ \texttt{:}\ a'\texttt{;}\ \ldots\texttt{\}}]\!]_B : \texttt{<}j_1\ \texttt{:}\ c_1\texttt{;}\ \ldots\texttt{>} \to \texttt{*} \\ [\![h]\!]_b : \texttt{<}i\ \texttt{:}\ \texttt{*;}\ \ldots\texttt{;}\ j_1\ \texttt{:}\ c_1\texttt{;}\ \ldots\texttt{>} \to \texttt{*} \ \cdots \end{array}}{\begin{array}{c} \texttt{sig \{\{}i_1\ \texttt{:}\ p_1\texttt{;}\ \ldots\texttt{\}\} type}\ i\ \ldots\ \texttt{value}\ j\ \texttt{:}\ h\ \ldots\ \texttt{end} \Rightarrow \\ \texttt{\~{}}k\texttt{:}a'\ \ldots\ \texttt{->} \\ \texttt{sig \{\{}k\ \texttt{:}\ a'\texttt{;}\ \ldots\texttt{\}\} type}\ i\ \ldots\ \texttt{value}\ j\ \texttt{:}\ h\ \ldots\ \texttt{end} \end{array}} \tag{289}$$

$$\frac{\begin{array}{c} m \rhd \texttt{\~{}}i_1\texttt{:}p'_1\ \ldots\ \texttt{->}\ \texttt{\{}j_1\ \texttt{:}\ q'_1\texttt{;}\ \ldots\texttt{\}} \\ q \Rightarrow \texttt{\~{}}j_1\texttt{:}q'_1\ \ldots\ \texttt{->}\ q' \end{array}}{m\ \texttt{|}\ q \Rightarrow \texttt{\~{}}i_1\texttt{:}p'_1\ \ldots\ \texttt{->}\ m\ \texttt{|}\ q'} \tag{290}$$

$$\frac{\begin{array}{c} q \Rightarrow \texttt{\~{}}j_1\texttt{:}q'_1\ \ldots\ \texttt{->}\ q' \\ (m\ \texttt{:>}\ \texttt{\{}j_1\ \texttt{:}\ q'_1\texttt{;}\ \ldots\texttt{\}})\ \texttt{|}\ q \Rightarrow p \end{array}}{q\ \texttt{with}\ m \Rightarrow p} \tag{291}$$

$$\frac{\begin{array}{c}(j_t\ :\ q_t\texttt{;}\ \ldots) = k\ :\ a'\texttt{;}\ \ldots\texttt{;}\ i\ :\ \texttt{strip}\,p'\texttt{;}\ j\ :\ \texttt{strip}\,q'\texttt{;}\ \ldots\\ p \Rightarrow \texttt{\~{}}j_t\texttt{:}q_t\ \ldots\ \texttt{->}\ p'\\ q \Rightarrow \texttt{\~{}}j_t\texttt{:}q_t\ \ldots\ \texttt{->}\ q'\\ \vdots\\ (j_F\ :\ q_F\texttt{;}\ \ldots) = (j_t\ :\ q_t\texttt{;}\ \ldots)\ \texttt{;}\ (i\ :\ p'\texttt{;}\ j\ :\ q'\texttt{;}\ \ldots)\end{array}}{\texttt{rec}\ i\ \texttt{\{}i\ :\ p\texttt{;}\ j\ :\ q\texttt{;}\ \ldots\texttt{\}} \Rightarrow \texttt{\~{}}j_F\texttt{:}q_F\ \ldots\ \texttt{->}\ p'} \qquad (292)$$

$$\frac{\begin{array}{c}a \Rightarrow\ \ldots\ \texttt{->}\ a'\quad\cdots\\ p \Rightarrow \texttt{\~{}}i_1\texttt{:}p'_1\ \ldots\ \texttt{\~{}}i_n\texttt{:}p'_n\ \texttt{->}\ p'\\ \text{and } \texttt{\~{}}i_1\texttt{:}p'_1\ \ldots\ \texttt{\~{}}i_n\texttt{:}p'_n\ \texttt{->}\ p' \text{ is the least}\\ \text{detailed saturation of } p\end{array}}{\texttt{[}p\texttt{]} \Rightarrow \texttt{\~{}}k\texttt{:}a'\ \ldots\ \texttt{\~{}}i_1\texttt{:}p'_1\ \ldots\ \texttt{\~{}}i_n\texttt{:}p'_n\ \texttt{->}\ p'} \qquad (293)$$

$$\frac{\begin{array}{c}p \Rightarrow \texttt{\~{}}i_1\texttt{:}p'_1\ \ldots\ \texttt{\~{}}i_n\texttt{:}p'_n\ \texttt{->}\ p'\\ [\![p'_1]\!]_B = [\![q'_1]\!]_B\quad\cdots\quad [\![p'_n]\!]_B = [\![q'_n]\!]_B\qquad [\![p']\!]_B = [\![q']\!]_B\\ q_1 \Rightarrow\ \ldots\ \texttt{->}\ q'_1\quad\cdots\quad q_n \Rightarrow\ \ldots\ \texttt{->}\ q'_n\qquad q \Rightarrow\ \ldots\ \texttt{->}\ q'\end{array}}{p \Rightarrow \texttt{\~{}}i_1\texttt{:}q'_1\ \ldots\ \texttt{\~{}}i_n\texttt{:}q'_n\ \texttt{->}\ q'} \qquad (294)$$

The meaning of rule (292) is best illustrated by the following idealized and simplified variant of the rule.

$$\frac{\begin{array}{c}\texttt{\~{}}i\texttt{:}p_r\ \texttt{\~{}}j\texttt{:}q_r\ \ldots\ \texttt{->}\ p \Rightarrow p_r\\ \texttt{\~{}}i\texttt{:}p_r\ \texttt{\~{}}j\texttt{:}q_r\ \ldots\ \texttt{->}\ q \Rightarrow q_r\\ \vdots\end{array}}{\texttt{rec}\ i\ \texttt{\{}i\ :\ p\texttt{;}\ j\ :\ q\texttt{;}\ \ldots\texttt{\}} \Rightarrow p_r} \qquad (295)$$

The problem is that, informally, the only derivations based on this variant are "infinite derivations" producing "infinite terms". The terms would still have finite denotations but only when assigned by "transfinite semantic computations" [29]. Rule (292) approximates the process in a single step of saturation. The results are always the same as in the idealized rule, and the classes of accepted terms, though incomparable, are both satisfactory from the programming perspective.

The operation `strip`, defined precisely in Appendix A.2.5, recursively traverses a component of a saturated specification. The base specifications are stripped of their explicit parameters and the types of their values are removed, so that only names of types are retained, together with an empty list of parameters. The symbol ";" when used as an operation on indexed lists denotes the catenation of lists, where the second one takes precedence. So "lp ; lq" is equal to "lq `@@` (`subtract` lp lq)", where "`@@`" is a catenation of indexed lists with no common indices and `subtract` is an indexed list subtraction (i.e., `subtract` $l\ l'$ contains those and only those elements of l that are labeled by indices not occurring in l').

Typing of modules

Here we present the typing of module terms (which inhabit the syntactic domain `DULE`). For a type-correct module or specification, the semantics in categorical model is almost always defined, see Section 9.2.2 for details. The dependency of typing on saturation is limited by the following fact.

Observation 9.2.7. *Every specification derived for a module is saturated.*

For typing isolated modules in rule (307) we have to state the following property.

Conjecture 9.2.8. *For any module m, if the set of specifications derivable for m is nonempty, it contains the least detailed specification.*

Specification reconstruction based on the typing rules for modules is implemented in the Dule compiler, but the algorithm awaits a theoretical analysis, see Section 10.4. The property stated as the above conjecture is fundamental for our specification reconstruction algorithm, which finds (if successful) the least detailed specification of a module. However, the partiality of the algorithm seems unavoidable, since the rules to implement involve semantics and the semantic function is not injective (e.g., in the case of `S_Pp`, see Section 5.2.4) so it is sometimes impossible to reconstruct the intended specifications based on the abstraction class of their semantics. However, the problem occurs only in very specific cases, so we believe a future proof of correctness of our specification reconstruction algorithm will lead to a constructive proof of Conjecture 9.2.8.

In rule (303) we use the semantic function $[\![\_]\!]_B$ both for specifications and for modules. The coercibility relation "$\geq$" can be described in terms of definability of trimming (the `m_Trim` combinator of Section 5.2.5): $m \geq r$ if and only if $m :> r$ is defined in the semantics of the internal module language. The relation "p' is a sub-specification of q'", where p' and q' are components of saturated specifications, defined by `M_Id`($[\![p']\!]_B$) $\geq [\![q']\!]_B$, is a superset of the order on specifications from Definition 9.2.5 restricted to components of saturated specifications. The main difference is that a base specification is a sub-specification of a base specification with fewer components, whereas only the empty base specification, having the same semantics as the empty product specification, is less or equally detailed than any other base specifications.

The linking combinator of the internal module language takes a list of defined specifications ($q', \ldots$) and libraries ($m', \ldots$), as in the user language, and the list of modules to be linked together. Note that the outcome of the typing of the specifications and the libraries is not used anywhere in rule (304). The type-checking of these environments is performed only to ensure that errors are captured already after defining and not as late as at the first use, which may never happen.

The operation `diff` is an indexed list subtraction, which additionally verifies identity of components at the same indices in the argument lists. The subtraction used in rule (304) ensures that only module parameters that are not implemented inside the linking expression are regarded as parameters of the whole linking expression. The linking combinator expects its operands to have no circular dependencies among themselves. Checking this would amount to performing topological sorting. A simpler and textually obvious way, employed in rule (304), is to ensure the parameters of a specification of a module mention only names of global parameters and previous modules. This is what the auxiliary function `depend_on_previous` checks and here, unlike everywhere else, the order of elements on the indexed list matters.

The specification p in rule (296) is arbitrary; the only item that occurs on the indexed list (i : p; ...) at the left had side of "$\Rightarrow$" and is determined by the typed module (a module projection/variable in this case) is the index i. Such nondeterminism is indispensable, because the module projection can appear in an arbitrary context, referring to an arbitrary module. In rule (300) both the base module parameters and the types of values are indeterministically "guessed". In the last list of premises in the rule, we extend the semantic substitution operation of the bare core language to multiple arguments in the obvious way.

$$\frac{\texttt{\{}i\ \texttt{:}\ p\texttt{;}\ \ldots\texttt{\}} \Rightarrow \texttt{\~{}}k\texttt{:}a'\ \ldots\ \texttt{->}\ \texttt{\{}i\ \texttt{:}\ p'\texttt{;}\ \ldots\texttt{\}}}{i \rhd \texttt{\~{}}k\texttt{:}a'\ \ldots\ \texttt{->}\ p'} \tag{296}$$

$$\frac{\begin{array}{c} m_1 \rhd p_1 \quad \cdots \quad m_n \rhd p_n \\ \texttt{\{}i_1\ \texttt{:}\ p_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ p_n\texttt{\}} \Rightarrow q \end{array}}{\texttt{\{}i_1\ \texttt{=}\ m_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ m_n\texttt{\}} \rhd q} \tag{297}$$

$$\frac{\begin{array}{c} m_1 \rhd p_1 \quad \cdots \quad m_n \rhd p_n \\ \texttt{\{\{}i_1\ \texttt{:}\ p_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ p_n\texttt{\}\}} \Rightarrow q \end{array}}{\texttt{\{\{}i_1\ \texttt{=}\ m_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ m_n\texttt{\}\}} \rhd q} \tag{298}$$

$$\frac{\begin{array}{c} p \Rightarrow \texttt{~}i_1\!:\!p'_1 \ \ldots\ \texttt{~}i_n\!:\!p'_n \ \texttt{->}\ p' \\ m \rhd \texttt{~}k\!:\!a' \ \ldots\ \texttt{~}i_1\!:\!p'_1 \ \ldots\ \texttt{~}i_n\!:\!p'_n \ \texttt{->}\ p' \end{array}}{\texttt{::}\ p\ m \rhd \texttt{~}k\!:\!a' \ \ldots\ \texttt{~}i_1\!:\!p'_1 \ \ldots\ \texttt{~}i_n\!:\!p'_n \ \texttt{->}\ p'} \tag{299}$$

$$\frac{\begin{array}{c} \texttt{\{\{}i_1 \ \texttt{:}\ p_1\texttt{;}\ \ldots\texttt{\}\}} \Rightarrow\ \ldots\ \texttt{->}\ \texttt{\{}k\ \texttt{:}\ a'\texttt{;}\ \ldots\texttt{\}} \\ f = [\![\texttt{\{}k\ \texttt{:}\ a'\texttt{;}\ \ldots\texttt{\}}]\!]_B \qquad f : c \to * \\ [\![g]\!]_b : c \to * \ \cdots \\ [\![t]\!]_b : f \to [\![h[g/i,\ldots]]\!]_b \ \cdots \end{array}}{\begin{array}{c} \texttt{struct type}\ i\ \texttt{=}\ g\ \ldots\ \texttt{value}\ j\ \texttt{=}\ t\ \ldots\ \texttt{end} \rhd \\ \texttt{~}k\!:\!a' \ \ldots\ \texttt{->} \\ \texttt{sig \{\{}k\ \texttt{:}\ a'\texttt{;}\ \ldots\texttt{\}\} type}\ i\ \ldots\ \texttt{value}\ j\ \texttt{:}\ h\ \ldots\ \texttt{end} \end{array}} \tag{300}$$

$$\frac{\begin{array}{c} m_2 \rhd \texttt{~}j_1\!:\!q'_1 \ \ldots\ \texttt{->}\ q' \\ m_1 \rhd \texttt{~}i_1\!:\!p'_1 \ \ldots\ \texttt{->}\ \texttt{\{}j_1\ \texttt{:}\ q'_1\texttt{;}\ \ldots\texttt{\}} \end{array}}{m_1\ \texttt{|}\ m_2 \rhd \texttt{~}i_1\!:\!p'_1 \ \ldots\ \texttt{->}\ m_1\ \texttt{|}\ q'} \tag{301}$$

$$\frac{\begin{array}{c} m_2 \rhd \texttt{~}j_1\!:\!q'_1 \ \ldots\ \texttt{->}\ q' \\ (m_1\ \texttt{:>}\ \texttt{\{}j_1\ \texttt{:}\ q'_1\texttt{;}\ \ldots\texttt{\}})\ \texttt{|}\ m_2 \rhd p \end{array}}{m_2\ \texttt{with}\ m_1 \rhd p} \tag{302}$$

$$\frac{\begin{array}{c} p \Rightarrow \texttt{~}i_1\!:\!p'_1 \ \ldots\ \texttt{~}i_n\!:\!p'_n \ \texttt{->}\ p' \\ m \rhd \texttt{~}k\!:\!a' \ \ldots\ \texttt{~}i_1\!:\!p'_1 \ \ldots\ \texttt{~}i_n\!:\!p'_n \ \texttt{->}\ q' \\ [\![m]\!]_B \geq [\![p']\!]_B \end{array}}{m\ \texttt{:>}\ p \rhd \texttt{~}k\!:\!a' \ \ldots\ \texttt{~}i_1\!:\!p'_1 \ \ldots\ \texttt{~}i_n\!:\!p'_n \ \texttt{->}\ p'} \tag{303}$$

$$\frac{\begin{array}{c} m \rhd p \quad \cdots \\ q' \Rightarrow q'' \quad \cdots \\ m' \rhd p'' \quad \cdots \\ \texttt{\{}i\ \texttt{:}\ p\texttt{;}\ \ldots\texttt{\}} \Rightarrow \texttt{~}k\!:\!a' \ \ldots\ \texttt{->}\ \texttt{\{}i\ \texttt{:}\ p'\texttt{;}\ \ldots\texttt{\}} \\ (j_F\ \texttt{:}\ q_F\texttt{;}\ \ldots) = \texttt{diff}\ (k\ \texttt{:}\ a'\texttt{;}\ \ldots)\ (i\ \texttt{:}\ p'\texttt{;}\ \ldots) \\ \texttt{depend\_on\_previous}\ (j_F\ \texttt{:}\ q_F\texttt{;}\ \ldots)\ (i\ \texttt{:}\ p\texttt{;}\ \ldots) \end{array}}{\begin{array}{c} \texttt{link \{}(j'\ \texttt{:}\ q'\texttt{;}\ \ldots),\ (i'\ \texttt{=}\ m'\texttt{;}\ \ldots),\ (i\ \texttt{=}\ m\texttt{;}\ \ldots)\texttt{\}} \rhd \\ \texttt{~}j_F\!:\!q_F \ \ldots\ \texttt{->}\ \texttt{\{}i\ \texttt{:}\ p'\texttt{;}\ \ldots\texttt{\}} \end{array}} \tag{304}$$

$$\frac{\begin{array}{c} m \rhd p \quad \cdots \\ \texttt{\{}i \texttt{ : } p\texttt{; ...\}} \Rightarrow \texttt{\~{}}k\texttt{:}a' \texttt{ ... -> \{}i \texttt{ : } p'\texttt{; ...\}} \\ (j_F \texttt{ : } q_F\texttt{; ...}) = \texttt{diff } (k \texttt{ : } a'\texttt{; ...}) \; (i \texttt{ : } p'\texttt{; ...}) \end{array}}{\texttt{ind \{}i \texttt{ = } m\texttt{; ...\}} \rhd \texttt{\~{}}j_F\texttt{:}q_F \texttt{ ... -> \{}i \texttt{ : } p'\texttt{; ...\}}} \tag{305}$$

$$\frac{\begin{array}{c} m \rhd p \quad \cdots \\ \texttt{\{}i \texttt{ : } p\texttt{; ...\}} \Rightarrow \texttt{\~{}}k\texttt{:}a' \texttt{ ... -> \{}i \texttt{ : } p'\texttt{; ...\}} \\ (j_F \texttt{ : } q_F\texttt{; ...}) = \texttt{diff } (k \texttt{ : } a'\texttt{; ...}) \; (i \texttt{ : } p'\texttt{; ...}) \end{array}}{\texttt{coind \{}i \texttt{ = } m\texttt{; ...\}} \rhd \texttt{\~{}}j_F\texttt{:}q_F \texttt{ ... -> \{}i \texttt{ : } p'\texttt{; ...\}}} \tag{306}$$

$$\frac{\begin{array}{c} m \rhd \texttt{\~{}}i_1\texttt{:}p'_1 \texttt{ ... } \texttt{\~{}}i_n\texttt{:}p'_n \texttt{ -> } p' \\ \text{and } \texttt{\~{}}i_1\texttt{:}p'_1 \texttt{ ... } \texttt{\~{}}i_n\texttt{:}p'_n \texttt{ -> } p' \text{ is the least} \\ \text{detailed specification derivable for } m \end{array}}{\texttt{[}m\texttt{]} \rhd \texttt{\~{}}i_1\texttt{:}p'_1 \texttt{ ... } \texttt{\~{}}i_n\texttt{:}p'_n \texttt{ -> } p'} \tag{307}$$

$$\frac{\begin{array}{c} m \rhd \texttt{\~{}}i_1\texttt{:}p'_1 \texttt{ ... } \texttt{\~{}}i_n\texttt{:}p'_n \texttt{ -> } p' \\ [\![p'_1]\!]_B = [\![q'_1]\!]_B \quad \cdots \quad [\![p'_n]\!]_B = [\![q'_n]\!]_B \qquad [\![p']\!]_B = [\![q']\!]_B \\ q_1 \Rightarrow \texttt{ ... -> } q'_1 \quad \cdots \quad q_n \Rightarrow \texttt{ ... -> } q'_n \qquad q \Rightarrow \texttt{ ... -> } q' \end{array}}{m \rhd \texttt{\~{}}i_1\texttt{:}q'_1 \texttt{ ... } \texttt{\~{}}i_n\texttt{:}q'_n \texttt{ -> } q'} \tag{308}$$

Semantics

Here is the definition of the semantic function for specifications and modules $[\![\_]\!]_B$ used in the typing rules. For module arguments the function is defined using the semantics of bare module language in internal module language $[\![\_]\!]$ (defined below) and the function $[\![\_]\!]_I$, assigning the semantics in the fixed categorical model of the internal core language to the internal module language terms. The latter function is given throughout the thesis by means of the semantic combinators and is summarized in Appendix A.2.2.

$$[\![m]\!]_B = [\![[\![m]\!]]\!]_I$$

For arguments that are components of saturated specifications, the definition is in terms of the semantic function $[\![\_]\!]^r$ defined below in this section and the function $[\![\_]\!]_I$.

$$[\![p]\!]_B = [\![[\![p]\!]^r]\!]_I$$

The semantic function can be applied only after a specification is saturated. Every saturated specification is a parameterized specification with a result that is not a parameterized specification. We will give three semantic functions for specifications translating them to the internal module language; $[\![\_]\!]^l$ and $[\![\_]\!]^s$ for parameterized specifications and $[\![\_]\!]^r$ for their individual components. The first of the semantic functions for the parameterized specifications provides all the flattened parameters and the second one — the result. Let $p = \texttt{\~{}}i_1 \texttt{:} p'_1 \; \ldots \; \texttt{\~{}}i_n \texttt{:} p'_n \; \texttt{->} \; p'$ be a saturated specification.

$$[\![p]\!]^l = i_1 \;\texttt{:}\; [\![p'_1]\!]^r \texttt{;} \; \ldots \texttt{;} \; i_n \;\texttt{:}\; [\![p'_n]\!]^r$$

$$[\![p]\!]^s = [\![p']\!]^r$$

Here is the semantic function for the specifications appearing as components of saturated specifications. The definition is by cases on the possible forms of the components. The semantic function $[\![\_]\!]^f$ that we use in the second semantic equation is the semantic function of the bare core language types and values in the categorical model, defined in Section 8.3.1, applied to the elements b_i of the syntactic domain `BASE-ITEM`.

$$\begin{array}{rcl}
[\![\texttt{\{}i \;\texttt{:}\; p'\texttt{;} \; \ldots\texttt{\}}]\!]^r & = & \texttt{\{}i \;\texttt{:}\; [\![p']\!]^r\texttt{;} \; \ldots\texttt{\}} \\
[\![\texttt{sig \{\{}i_1 \;\texttt{:}\; p_1\texttt{;} \; \ldots\texttt{\}\}} \; b_1 \; \ldots \; \texttt{end}]\!]^r & = & \texttt{sig} \; [\![\texttt{\{}i_1 \;\texttt{:}\; p_1\texttt{;} \; \ldots\texttt{\}}]\!]^r \\
 & & \quad [\![b_1]\!]^f \; \ldots \; \texttt{end} \\
[\![m \;\texttt{|}\; q]\!]^r & = & [\![m]\!] \;\texttt{|}\; [\![q]\!]^r
\end{array}$$

Here is the semantics of modules, applicable only to typed module expressions. The curly brackets at the right hand side of the fourth line denote `M_Accord`, not `M_Record`. The "`::`" symbol followed by `struct` at the right hand side of the fifth equation is not the explicit specification operation of the bare language but a starting syntactic symbol of the base module operator of the internal module language.

$$\begin{array}{rcl}
[\![i \triangleright p]\!] & = & \texttt{M\_Pr}([\![p]\!]^l, i) \\
[\![\texttt{\{}i_1 \texttt{ = } m_1\texttt{;} \; \ldots\texttt{;} \; i_n \texttt{ = } m_n\texttt{\}}]\!] & = & \texttt{\{}i_1 \texttt{ = } [\![m_1]\!]\texttt{;} \; \ldots\texttt{;} \; i_n \texttt{ = } [\![m_n]\!]\texttt{\}} \\
[\![\texttt{\{\{}i_1 \texttt{ = } m_1\texttt{;} \; \ldots\texttt{;} \; i_n \texttt{ = } m_n\texttt{\}\}}]\!] & = & \texttt{\{\{}i_1 \texttt{ = } [\![m_1]\!]\texttt{;} \; \ldots\texttt{;} \; i_n \texttt{ = } [\![m_n]\!]\texttt{\}\}} \\
[\![\texttt{::} \; p \; m]\!] & = & [\![m]\!] \\
[\![\texttt{struct} \; b_1 \; \ldots \; \texttt{end} \triangleright p]\!] & = & \texttt{::} \; \texttt{\{}[\![p]\!]^l\texttt{\}} \; \texttt{->} \; [\![p]\!]^s \\
 & & \texttt{struct} \; [\![b_1]\!]^f \; \ldots \; \texttt{end} \\
[\![m_1 \;\texttt{|}\; m_2]\!] & = & [\![m_1]\!] \;\texttt{|}\; [\![m_2]\!] \\
[\![(m_2 \triangleright p) \; \texttt{with} \; m_1]\!] & = & ([\![m_1]\!] \;\texttt{:>}\; \texttt{\{}[\![p]\!]^l\texttt{\}}) \;\texttt{|}\; [\![m_2]\!] \\
[\![(m \;\texttt{:>}\; p) \triangleright q]\!] & = & [\![m]\!] \;\texttt{:>}\; [\![q]\!]^s
\end{array}$$

$$\begin{aligned}
[\![\texttt{link \{}(j' : q';\ \ldots),\qquad & \\
(i' = m';\ \ldots), & \\
(i = m;\ \ldots)\texttt{\}}]\!] &= \texttt{link \{}i = [\![m]\!];\ \ldots\texttt{\}} \\
[\![\texttt{ind \{}i = m;\ \ldots\texttt{\}}]\!] &= \texttt{ind \{}i = [\![m]\!];\ \ldots\texttt{\}} \\
[\![\texttt{coind \{}i = m;\ \ldots\texttt{\}}]\!] &= \texttt{coind \{}i = [\![m]\!];\ \ldots\texttt{\}} \\
[\![\texttt{[}m\texttt{]}]\!] &= [\![m]\!]
\end{aligned}$$

Remark. The direct dependency of the module semantics on the module typing (visible only in some of the semantic equations, because concrete syntax of the internal module language allows us to omit many signatures in writing) could be eliminated by using additional operands of the bare language module constructors and "guessing" their contents. However, the semantics would still be sensible only for type-correct modules.

The converse dependency, the dependency of the typing on the semantics, is more deeply rooted into our module system. The typing of modules depends on their semantics, because it depends on semantics of specifications and they may contain modules hidden inside the specialization construct. Also for this reason, the equality of signatures cannot be syntactic and involves the semantics of modules. On the other hand, only the type definitions found inside modules are needed for type-checking of modules, so a clever implementation may strip modules of their value definitions and only then perform the semantic computations. Thus, termination of type-checking of modules would be ensured (unlike in the current straightforward implementation).

9.2.2 Completeness of module typing

Most of the operations of the internal module language are expressible using the bare module language. The major lacking operations are module identity and module "categorical" composition. In contrast to the internal language, well-typed expressions of the bare module language are very rarely undefined. This is possible, because the lack of module composition precludes most problematic cases. At the end of this section we will state and prove important properties of the bare module language. In particular, we will characterize the definedness conditions in the absence of module composition.

For the sake of completeness, below we present the syntax, typing and semantics of the module identity and composition, as if they were a part of the bare module language. We will illustrate, using the introduced syntax and semantics, where the danger of module composition lies. The definitions have also been used to provide an optional extension to the Dule compiler, making it possible to test the internal parts of modules' compilation.

Module identity and composition

We will temporarily extend the syntactic domain of modules with term constructors of module identity and module composition.

```
DULE  ::= ...
        | : {{ FIELD-SP ;...; FIELD-SP }}  % identity module
        | DULE . DULE                      % composition
```

The typing rules are not more complex than those for the bare language operations. Notice, however, that the outcome specification in rule (310) needn't be saturated, for instance if q' is a base specification with nontrivial context.

$$\frac{\texttt{\{\{}i_1\ \texttt{:}\ p_1\texttt{;}\ \ldots\texttt{\}\}} \Rightarrow \ldots\ \texttt{->}\ \texttt{\{}k\ \texttt{:}\ a'\texttt{;}\ \ldots\texttt{;}\ i_1\ \texttt{:}\ p'_1\texttt{;}\ \ldots\texttt{\}}}{\begin{array}{c}\texttt{:}\ \texttt{\{\{}i_1\ \texttt{:}\ p_1\texttt{;}\ \ldots\texttt{\}\}} \triangleright \\ \texttt{\~{}}k\texttt{:}a'\ \ldots\ \texttt{\~{}}i_1\texttt{:}p'_1\ \ldots\ \texttt{->}\ \texttt{\{}k\ \texttt{:}\ a'\texttt{;}\ \ldots\texttt{;}\ i_1\ \texttt{:}\ p'_1\texttt{;}\ \ldots\texttt{\}}\end{array}} \tag{309}$$

$$\frac{\begin{array}{c}m_1 \triangleright \texttt{\~{}}i_1\texttt{:}p'_1\ \ldots\ \texttt{->}\ \texttt{\{}j_1\ \texttt{:}\ q'_1\texttt{;}\ \ldots\texttt{\}} \\ m_2 \triangleright \texttt{\~{}}j_1\texttt{:}q'_1\ \ldots\ \texttt{->}\ q'\end{array}}{m_1\ \texttt{.}\ m_2 \triangleright \texttt{\~{}}i_1\texttt{:}p'_1\ \ldots\ \texttt{->}\ q'} \tag{310}$$

The semantics of composition is straightforward. The form of the syntax and semantics of module identity will be discussed below.

$$\begin{aligned} [\![\texttt{:}\ \texttt{\{\{}i_1\ \texttt{:}\ p_1\texttt{;}\ \ldots\texttt{\}\}} \triangleright q]\!] &= \texttt{:}\ [\![q]\!]^s \\ [\![m_1\ \texttt{.}\ m_2]\!] &= [\![m_1]\!]\ \texttt{.}\ [\![m_2]\!] \end{aligned}$$

The module identity is written with a colon and a row of specifications enclosed in double curly braces:

```
: {{Bool1 : sig type bool end;
    Bool3 : ~Bool2:sig type bool end -> sig type bool end}}
```

The identity morphism is taken on the product of all the results of the specifications in the row together with all their parameters. The syntax of the module identity is reminiscent of the syntax of the double product specification. This is not a coincidence; rule (309) states that the result part of the specification of the identity module above is the same as of specification:

```
{{Bool1 : sig type bool end;
  Bool3 : ~Bool2:sig type bool end -> sig type bool end}}
```

For the purpose of explicitly assigning whole specifications to modules, the module identity operation would be awkward. In the standard bare module language, the explicitly specified module construction is used in this role. Yet for asserting only the specifications of arguments or result of modules, the identity would be unrivaled. The following example begins with an identity, the only role of which is to assert the specification of the parameters of the linking expression.

```
: {{Elem : sig type t end}} .
link {nil, nil,
  List1 = :: ~Elem : sig type t end -> sig end struct end;
  List2 = :: ~Elem : sig type t end -> sig end struct end
          with {Elem = Elem};
  List3 = List2;
  Elem2 = Elem
}
```

Modules are composed with an infix dot, as in:

```
: {{Bool3 : ~Bool2:sig type bool end -> sig type bool end}}
  . : {{Bool3 : ~Bool2:sig type bool end -> sig type bool end}}
    . Bool2
```

The second composition is type-correct, because double product implicitly includes all its parameters among its components — parameter `Bool2` in this case. As expected, identity is the neutral element of composition and the semantics ensures that composition is associative.

The composition is a somewhat dangerous operation. If the specification of the second operand is parameterized, then the codomain of the composition may depend on the specifications of the old parameters, instead of the new ones. For example:

```
{Elem = :: sig type t end
        struct type t = {} end}
  . :: ~Elem : sig type t end ->
         sig value v : ~x:Elem.t -> {} end
    struct value v = fun ~x -> {} end
```

The result part of the least detailed specification of the above composition is the same as the result specification of the second operand

```
sig {{Elem : sig {{}} type t end}}
  value v : ~x:Elem.t -> {}
end
```

and thus depends on the type inside the parameter Elem. On the other hand, the whole specification of the composition

```
->
sig {{Elem : sig {{}} type t end}}
  value v : ~x:Elem.t -> {}
end
```

has no parameters and the module Elem is not accessible from the outside of the composition.

The least worrying phenomenon is that the specification of the composition above is not expressible in the user module language (not even via saturation). Much graver is the fact that this specifications is not saturated and, moreover, it has no saturated but equivalent counterpart. Even worse things can happen when such a module is put together with others. In particular, as seen in Section 5.2.4, the dependency on hidden parameters may cause undefinedness of module product operations. In the presence of composition most of the record error examples of Section 5.2.4 can be rewritten as type-correct Dule programs. Understanding and locating the resulting errors is then difficult, because the user is forced to work with the subtle definedness conditions of the internal semantics, instead of the simplified type-correctness rules for the bare module language.

If fact, of all the module operations, only composition exhibits the dangerous behavior, so it is banned, to simplify reasoning about user-written modules. On the other hand, instantiation, another variant of module composition where the output specification is explicitly specialized by any “hidden” module, is safe. We include it in the bare language as a general way to capture module superposition, so that the lack of categorical module composition does not affect practical usability of the language.

Properties

First, we state the properties analogous to the ones of the bare core language.

Observation 9.2.9. *Let $m \triangleright p$ be a result of a derivation of a specification for a module of the bare language (optionally, extended with the identity and composition constructors). Let $[\![m \triangleright p]\!] : r \to s$ be a result of a derivation of the internal module language domain and codomain* sign*-terms. Then, if $[\![r]\!]_I$ and $[\![s]\!]_I$ are defined, the following equations hold:*

$$[\![r]\!]_I = [\![\{[\![p]\!]^l\}]\!]_I \qquad [\![s]\!]_I = [\![[\![p]\!]^s]\!]_I$$

The semantic function for the internal module language terms $[\![\_]\!]_I$ is as used in Section 9.2.1. The proof of the observation, by tedious but easy induction over the derivation of $m \triangleright p$, is omitted.

Observation 9.2.10. *Each specification inferred for a bare language module can be expressed in the bare language.*

Note, however, that the recursive specifications are, in general, not expressible without the `rec` construction. Consequently, in the user module language (to be described in Section 9.2.4) all specifications derived for modules are expressible, but the recursive specifications are not, in general, expressible anonymously (that is as closed terms). One has to use the recursive form of specification declaration.

The ban on using module composition ensures that any type-correct module and specification has a semantics, unless it contains contradicting or gravely violated sharing requirements. To demonstrate this, first we prove the following property.

Lemma 9.2.11. *Let p be a saturated specification. Then the set of context types (as in Definition 5.2.4) of $[\![p]\!]^s$ is contained in the set of (all) types of $\{[\![p]\!]^l\}$.*

Proof. The proof is by induction on the derivation of $q \Rightarrow p$ (by p being saturated, such derivation, for some q, exists). The derivation may include the derivations of types for modules contained in specialized specifications, which complicates the proof.

Context types are introduced into saturated specifications in rule (289). The thesis holds there regardless of the induction hypothesis on the first premise, because all the context signatures of the resulting base signature are contained in its parameters. The other rules pass around the parameters intact, making the inductive steps possible. The most difficult of the inductive steps concerns rule (290). There, types from the domain of module m are added to the set of context types of the resulting specialized specification. However, by Observation 9.2.9 the types of the domain of m are contained in $\{[\![p]\!]^l\}$, and $[\![p]\!]^l$ is added to the parameters of the resulting signature. $\square$

The following fact is a positive statement of the property that with no module composition allowed, the specification of a module will never exhibit the worrying behavior of depending on a parameter that has disappeared.

Corollary 9.2.12. *Let $m \triangleright p$ be derivable according to the rules for typing bare language modules (without the categorical composition!). Then the set of context types (as in Definition 5.2.4) of $[\![p]\!]^s$ is contained in the set of (all) types of $\{[\![p]\!]^l\}$.*

Proof. By Observation 9.2.7, p is saturated, so by Lemma 9.2.11 it satisfies the property. $\square$

The next lemma is nontrivial, e.g., because it implies that if module composition is banned, signature specialization is always defined.

Lemma 9.2.13. *Let p be a saturated specification. Then the internal module language semantics of $[\![p]\!]^s$ is defined unless $[\![p]\!]^s$ contains as a subterm a module with undefined semantics or a product signature with undefined semantics.*

Proof. A product signature may be undefined, e.g., because of a contradicting sharing requirement. An undefined module may become a part of a signature through the signature specialization construct. The main positive point of the lemma is that if a specifications is saturated and the contained modules are defined then all the definedness condition for the specialization constructs are met.

The proof proceeds by structural induction over $[\![p]\!]^s$ with the lemma's thesis as the inductive hypothesis. The semantics of the bare language specifications yields four inductive cases to check. The case for product specifications is easily discharged, because either the internal language semantics of the specification is defined, or it trivially contains an undefined product signature. The hypothesis holds in the case of the base signature, because it is always defined.

The only case that remains is for signature specialization. Here we use the induction hypothesis on the signature being specialized and we have that it is either defined, or contains a module or a product signature with undefined semantics. In the latter case our signature specialization also contains such an offending subterm and the proposition hold. Therefore we only need to consider the case of both operands of the specialization having defined semantics.

As mentioned in Section 5.2.5 the two possible causes of undefinedness of specialization cannot occur in Dule programs. The cases with inconsistencies between the types on which the signature depends and the types provided by the module are rejected during type checking, because the parameters of a specialized specification are compared with the codomain of the module in rule (290). The parameters are guaranteed to be the only source of the signature context types, by Corollary 9.2.12.

The second cause of undefinedness of specialization, the failed merging of the module and signature types (Section 5.2.5), is avoided thanks to the naming convention encoded in the grammar, which differentiates modules and core language types. Bare language typing rules assure that all internal language modules that are the semantics of bare language modules have product domain. Therefore their domain type parts will be named by capitalized labels, while the signature local types are always named by lowercase labels. If either of the indexed lists is empty, the merging succeeds trivially. □

The lack of module composition removes all but one of the reasons for undefined module semantics. In particular, when no module composition is involved, the module instantiation is guaranteed to be always defined, the record cannot be invalid due to a clash between parameters of different fields, and the module operations built using module record do not add any more possibility for undefinedness, despite their various sophisticated soundness requirements.

Lemma 9.2.14. *Let m be a module such that $m \rhd p$, for some specification p. Then the internal module language semantics of $[\![m]\!]$ is defined unless inside the I-Dule language expression $[\![m]\!]$ there is a signature with undefined semantics or a record-like module (with* `M_Record`*,* `M_Accord`*,* `M_Concord`*,* `M_Link`*,* `M_Ind` *or* `M_CoInd` *in the root of the module expression) with a field name being the same as one of the parameters, but with a different content.*

Proof. (Sketch.) Structural induction over $[\![m]\!]$. If the module is well typed, then there can be no undefinedness caused by instantiation, because this would result in wrong specialized signatures, which would not pass saturation by Lemma 9.2.13. The case of record module with not agreeing fields (and similar cases) are discharged using Corollary 9.2.12. The proposition we are proving permits that a record module m is undefined in certain cases. Corollary 9.2.12 implies that the only not permitted cause of undefinedness, the sharing conflict between the fields of a module record, would result in conflicting parameters of their specifications, and thus the whole specification of the record would not pass saturation. $\square$

We conclude by merging Lemma 9.2.13 and 9.2.14 into a formalization of our claim that only contradicting sharing requirement expressed as a product signature and a very special kind of sharing violation pass the bare language type checking and produce error results in the internal language semantics.

Corollary 9.2.15. *A saturated specification or a typable module has defined semantics, unless it contains a product specification with undefined semantics or a record-like module with a field name being the same as one of the parameters, but with a different content.*

9.2.3 Environment-dependent module language

The bare module language, although quite usable, forces the programmer to frequently retype whole specifications as they are used in different places of the program. On the rare occasions when in different places the same module has to be applied to different parameters, the whole module has to be retyped too (as opposed the situation of a single module being the argument to several modules that is easily handled already in SCM using records or linking). By employing a text editor one can move from retyping to copying, which eliminates retyping errors, but still leads to problems when one of the copies is changed and not all of the others are updated. A good macro preprocessor can alleviate the inconveniences, yet the separate copies of modules or specifications would still be separately parsed, type-checked and compiled. The most comfortable solution is to introduce specialized and restricted macro-like capabilities to the module system itself. This is what the environment-dependent module language, as introduced in this section, is for.

Our somewhat radical delegation of the mechanisms for code reuse to the domain of programming tools, or at least auxiliary outer layers of the language semantics is connected to our views on code reuse, as sketched in the discussion on previous work in Section 1.2. The shallow and mostly syntactic nature of the environment-dependent module language seems to substantiate our views.

Syntax

In the description of the syntax we use the same conventions as for the bare module language. Just as in the bare module language, the syntactic domains `TYPE` and `VALUE` are taken from the bare core language. We use the three kinds of labels that were used in the bare module language and a new syntactic domain of labels, consisting of capitalized identifiers, called `SP-LABEL`. The abstract syntax, used in our compiler code, can be found in module `EDule` contained in file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/mod_front.ml`.

```
FIELD-SP   ::= DULE-LABEL : SP
PARAM-SP   ::= ~ DULE-LABEL : SP
BB-ITEM    ::= type TYPE-LABEL
             | value VALUE-LABEL : TYPE
SP         ::= { FIELD-SP ;...; FIELD-SP }   % product specification
             | {{ FIELD-SP ;...; FIELD-SP }} % double product
             | PARAM-SP...PARAM-SP -> SP     % parameterized
             | sig BB-ITEM...BB-ITEM end     % base specification
             | SP with DULE                  % specialized
             | SP-LABEL                      % recalled

FIELD-DULE ::= DULE-LABEL = DULE
BASE-ITEM  ::= type TYPE-LABEL = TYPE
             | value VALUE-LABEL = VALUE
DEF-DULE   ::= DULE-LABEL = DULE
DEF-SP     ::= SP-LABEL = SP
IND-DULE   ::= DEF-DULE and...and DEF-DULE
COIND-DULE ::= DEF-DULE and...and DEF-DULE
REC-SP     ::= DEF-SP and...and DEF-SP
LINK-ITEM  ::= module DEF-DULE
             | module ind IND-DULE
             | module coind COIND-DULE
             | spec DEF-SP
             | spec rec REC-SP
             | library DEF-DULE
             | library ind IND-DULE
             | library coind COIND-DULE
```

```
DULE       ::= DULE-LABEL                         % projection
             | { FIELD-DULE ;...; FIELD-DULE }    % record
             | {{ FIELD-DULE ;...; FIELD-DULE }} % double record
             | :: SP DULE                         % explicily specified
             | struct BASE-ITEM...BASE-ITEM end   % base module
             | DULE "|" DULE                      % instantiation
             | DULE with DULE                     % --- with trimming
             | DULE :> SP                         % trimmed module
             | link LINK-ITEM...LINK-ITEM end     % to be linked
             | load DULE-LABEL                    % load library

START      ::= LINK-ITEM...LINK-ITEM
```

Translation

We give a translation from the environment-dependent module language to the bare module language. This translation is type-less, which means that only names are dereferenced and no specifications for modules are derived or used. The translation is compositional and deterministic and is generated by the rules for syntactic domains `SP`, `DULE`, `LINK-ITEM`, `DEF-DULE`, `IND-DULE`, `COIND-DULE`, `REC-SP` and `START`. A faithful implementations of this translation is in module `ElabEDule` in file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/mod_front.ml`.

We begin with the rule for `START`, which implies that the result of a program is a linking construction applied to all the top-level definitions, evaluated in empty environments.

$$\frac{\emptyset, \emptyset \vdash \texttt{link}\ d_1\ \ldots\ d_n\ \texttt{end} \Rightarrow m'}{\vdash d_1\ \ldots\ d_n \Rightarrow m'} \tag{311}$$

Translating specifications

The rules for `SP` are all straightforward. Rule (317) is responsible for dereferencing named specifications. These specifications are stored in the environment (indexed list) *PE*. The environment *ME* stores library modules and is only passed around here. The notation *PE*(i) means "i-th element of the indexed list *PE*".

$$\frac{PE, ME \vdash p \Rightarrow p' \quad \cdots}{PE, ME \vdash \texttt{\{}i\ \texttt{:}\ p\texttt{;}\ \ldots\texttt{\}} \Rightarrow \texttt{\{}i\ \texttt{:}\ p'\texttt{;}\ \ldots\texttt{\}}} \tag{312}$$

$$\frac{PE, ME \vdash p \Rightarrow p' \quad \cdots}{PE, ME \vdash \texttt{\{\{}i\ \texttt{:}\ p\texttt{;}\ \ldots\texttt{\}\}} \Rightarrow \texttt{\{\{}i\ \texttt{:}\ p'\texttt{;}\ \ldots\texttt{\}\}}} \tag{313}$$

$$\frac{\begin{array}{c} PE, ME \vdash p_1 \Rightarrow p'_1 \\ \vdots \\ PE, ME \vdash p_n \Rightarrow p'_n \\ PE, ME \vdash q \Rightarrow q' \end{array}}{PE, ME \vdash \texttt{\~{}}i_1 : p_1 \ \ldots \ \texttt{\~{}}i_n : p_n \ \texttt{->} \ q \Rightarrow \texttt{\~{}}i_1 : p'_1 \ \ldots \ \texttt{\~{}}i_n : p'_n \ \texttt{->} \ q'} \tag{314}$$

$$\frac{}{PE, ME \vdash \texttt{sig}\ b_1 \ \ldots \ b_n \ \texttt{end} \Rightarrow \texttt{sig}\ b_1 \ \ldots \ b_n \ \texttt{end}} \tag{315}$$

$$\frac{PE, ME \vdash p \Rightarrow p' \qquad PE, ME \vdash m \Rightarrow m'}{PE, ME \vdash p \ \texttt{with} \ m \Rightarrow p' \ \texttt{with} \ m'} \tag{316}$$

$$\frac{PE(i) = p'}{PE, ME \vdash i \Rightarrow p'} \tag{317}$$

Translating modules

Now we give the rules for the syntactic domain `DULE`. Rule (328) is responsible for dereferencing libraries. Notice that rules (318) and (319), describing the translation of module projections, do not use the library environment *ME*. Which module is projected depends on the record with which the projection is composed. As a shorthand, if there is a named specification with a matching name, the projection is associated with this specification.

In rule (327) subsequent definitions are being translated in environments enriched with the results of previous definitions. The enrichment is expressed by semicolon that, as before, denotes a catenation of indexed lists, where the contents of the second list takes precedence. Then the results are put together using the operator "`@@`", which fails if its operand indexed lists have any common indexes. The results are put inside the linking constructor and the outcome module is isolated. The rest of the rules are straightforward. The auxiliary predicate `not_in`, introduced in Section 3.2, is true if an index is not on the argument indexed list.

$$\frac{PE(i) = p'}{PE, ME \vdash i \Rightarrow \texttt{:}\texttt{:}\ p'\ i} \tag{318}$$

$$\frac{\texttt{not\_in}\ i\ PE}{PE, ME \vdash i \Rightarrow i} \tag{319}$$

$$\frac{PE, ME \vdash m \Rightarrow m' \quad \cdots}{PE, ME \vdash \texttt{\{}i\ \texttt{=}\ m\texttt{;}\ \ldots\texttt{\}} \Rightarrow \texttt{\{}i\ \texttt{=}\ m'\texttt{;}\ \ldots\texttt{\}}} \tag{320}$$

$$\frac{PE, ME \vdash m \Rightarrow m' \quad \cdots}{PE, ME \vdash \texttt{\{\{}i\ \texttt{=}\ m\texttt{;}\ \ldots\texttt{\}\}} \Rightarrow \texttt{\{\{}i\ \texttt{=}\ m'\texttt{;}\ \ldots\texttt{\}\}}} \tag{321}$$

$$\frac{PE, ME \vdash p \Rightarrow p' \qquad PE, ME \vdash m \Rightarrow m'}{PE, ME \vdash \texttt{::}\ p\ m \Rightarrow \texttt{::}\ p'\ m'} \tag{322}$$

$$\frac{}{PE, ME \vdash \texttt{struct}\ b_1\ \ldots\ b_n\ \texttt{end} \Rightarrow \texttt{struct}\ b_1\ \ldots\ b_n\ \texttt{end}} \tag{323}$$

$$\frac{PE, ME \vdash m_1 \Rightarrow m_1' \qquad PE, ME \vdash m_2 \Rightarrow m_2'}{PE, ME \vdash m_1\ \texttt{|}\ m_2 \Rightarrow m_1'\ \texttt{|}\ m_2'} \tag{324}$$

$$\frac{PE, ME \vdash m_1 \Rightarrow m_1' \qquad PE, ME \vdash m_2 \Rightarrow m_2'}{PE, ME \vdash m_2\ \texttt{with}\ m_1 \Rightarrow m_2'\ \texttt{with}\ m_1'} \tag{325}$$

$$\frac{PE, ME \vdash p \Rightarrow p' \qquad PE, ME \vdash m \Rightarrow m'}{PE, ME \vdash m\ \texttt{:>}\ p \Rightarrow m'\ \texttt{:>}\ p'} \tag{326}$$

$$\frac{\begin{array}{c} PE_0, ME_0 \vdash d_1 \Rightarrow PE_1, ME_1, ME_1^R \\ (PE_0\ ;\ PE_1), (ME_0\ ;\ ME_1) \vdash d_2 \Rightarrow PE_2, ME_2, ME_2^R \\ \vdots \\ (PE_0\ ;\ \ldots\ ;\ PE_{n-1}), (ME_0\ ;\ \ldots\ ;\ ME_{n-1}) \vdash d_n \Rightarrow PE_n, ME_n, ME_n^R \\ PE = PE_1\ \texttt{@@}\ PE_2\ \texttt{@@}\ \ldots\ \texttt{@@}\ PE_n \\ ME = ME_1\ \texttt{@@}\ ME_2\ \texttt{@@}\ \ldots\ \texttt{@@}\ ME_n \\ ME^R = ME_1^R\ \texttt{@@}\ ME_2^R\ \texttt{@@}\ \ldots\ \texttt{@@}\ ME_n^R \end{array}}{PE_0, ME_0 \vdash \texttt{link}\ d_1\ d_2\ \ldots\ d_n\ \texttt{end} \Rightarrow \texttt{[link \{}PE\texttt{,}\ ME\texttt{,}\ ME^R\texttt{\}]}} \tag{327}$$

$$\frac{ME(i) = m'}{PE, ME \vdash \texttt{load}\ i \Rightarrow m'} \tag{328}$$

Translating definitions

Now we give the rules for `LINK-ITEM`. We isolate specifications and modules at each declaration, so that type reconstruction has to be finished successfully without any knowledge of the context of the declaration. This is needed for separate compilation, makes tracking down errors easier, speeds up the compiler, etc.

Remark. On the other hand, the isolated specification construction is the only obstacle to strict referential transparency of the environment-dependent module language with respect to specification identifiers. If not for the isolated specifications, exchanging every specification name for its textual definition would not change the internal semantics of a module or a specification. Similarly, library loads are exchangeable with their corresponding defined library modules, but only after the modules are additionally isolated. For more fundamental reasons, there is no referential transparency with respect to module variables, expressed as module projections. For example trimming a module variable may fail, while trimming the module it refers to would succeed. This referential transparency only holds where module abstraction is enforced, for example at the top level of a linking construct.

The isolated specification and isolated module constructions could be removed if the internal language semantics of base signatures was changed so that the context signature is ignored if not mentioned in the types of values, see the remarks on page 156. However, this change would complicate the semantics a lot and make it closely dependent on the properties of the core language. For example, a core language allowing implicit opening of parameter modules would require a special extension to the module semantics. Another complication is that type reconstruction and specification reconstruction would become mutually recursive.

Isolating of an entire environment, denoted $\texttt{[}PE_y\texttt{]}$ and $\texttt{[}ME_z\texttt{]}$ below, is an abbreviation for isolating each element of the indexed list. So "$\texttt{[}i\ \texttt{:}\ p\texttt{;}\ \ldots\texttt{]}$" means "$i\ \texttt{:}\ \texttt{[}p\texttt{]}\texttt{;}\ \ldots$", and similarly for the environments consisting of modules. The operation `subtract` is used in rules (333), (335) and (336) for arguments y and z that are not, technically, indexed lists but elements of the syntactic domains `REC-SP`, `IND-DULE` and `COIND-DULE`. However, they can be trivially translated to proper indexed lists.

$$\frac{PE, ME \vdash i \texttt{ = } m \Rightarrow m'}{PE, ME \vdash \texttt{module } i \texttt{ = } m \Rightarrow \emptyset, \emptyset, i \texttt{ = } m'} \tag{329}$$

$$\frac{PE, ME \vdash z \Rightarrow ME_z}{PE, ME \vdash \texttt{module ind } z \Rightarrow \emptyset, \emptyset, ME_z} \tag{330}$$

$$\frac{PE, ME \vdash z \Rightarrow ME_z}{PE, ME \vdash \texttt{module coind } z \Rightarrow \emptyset, \emptyset, ME_z} \tag{331}$$

$$\frac{PE, ME \vdash p \Rightarrow p'}{PE, ME \vdash \texttt{spec}\ i \texttt{ = } p \Rightarrow i \texttt{ : [}p'\texttt{]}, \emptyset, \emptyset} \tag{332}$$

$$\frac{PE_d = \texttt{subtract}\ PE\ y \qquad PE_d, ME \vdash y \Rightarrow PE_y}{PE, ME \vdash \texttt{spec rec}\ y \Rightarrow \texttt{[}PE_y\texttt{]}, \emptyset, \emptyset} \tag{333}$$

$$\frac{PE, ME \vdash i \texttt{ = } m \Rightarrow m'}{PE, ME \vdash \texttt{library}\ i \texttt{ = } m \Rightarrow \emptyset, i \texttt{ = [}m'\texttt{]}, \emptyset} \tag{334}$$

$$\frac{ME_d = \texttt{subtract}\ ME\ z \qquad PE, ME_d \vdash z \Rightarrow ME_z}{PE, ME \vdash \texttt{library ind}\ z \Rightarrow \emptyset, \texttt{[}ME_z\texttt{]}, \emptyset} \tag{335}$$

$$\frac{ME_d = \texttt{subtract}\ ME\ z \qquad PE, ME_d \vdash z \Rightarrow ME_z}{PE, ME \vdash \texttt{library coind}\ z \Rightarrow \emptyset, \texttt{[}ME_z\texttt{]}, \emptyset} \tag{336}$$

Here are the rules for DEF-DULE. The first one introduces an abbreviation: if there is a specification with the same name as the module we are defining (a library module or an ordinary module) then we assign this specification to the module.

$$\frac{PE(i) = p' \qquad PE, ME \vdash m \Rightarrow m'}{PE, ME \vdash i \texttt{ = } m \Rightarrow \texttt{ :: } p'\ m'} \tag{337}$$

$$\frac{\texttt{not\_in}\ i\ PE \qquad PE, ME \vdash m \Rightarrow m'}{PE, ME \vdash i \texttt{ = } m \Rightarrow m'} \tag{338}$$

We supply the rules for IND-DULE and COIND-DULE. The first one uses the inductive module constructor and the second one the coinductive module constructor. Otherwise, they are the same.

$$\frac{\begin{array}{c} PE, ME \vdash i_1 \texttt{ = } m_1 \Rightarrow m_1' \\ \vdots \\ PE, ME \vdash i_n \texttt{ = } m_n \Rightarrow m_n' \\ m' = \texttt{ind \{}i_1 \texttt{ = } m_1'\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ = } m_n'\texttt{\}} \end{array}}{\begin{array}{c} PE, ME \vdash i_1 \texttt{ = } m_1 \texttt{ and } \ldots \texttt{ and } i_n \texttt{ = } m_n \Rightarrow \\ i_1 \texttt{ = } m' \texttt{ . } i_1\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ = } m' \texttt{ . } i_n \end{array}} \tag{339}$$

$$\frac{\begin{array}{c} PE, ME \vdash i_1 \texttt{ = } m_1 \Rightarrow m'_1 \\ \vdots \\ PE, ME \vdash i_n \texttt{ = } m_n \Rightarrow m'_n \\ m' = \texttt{coind \{}i_1 \texttt{ = } m'_1\texttt{;} \ \ldots \texttt{;} \ i_n \texttt{ = } m'_n\texttt{\}} \end{array}}{\begin{array}{c} PE, ME \vdash i_1 \texttt{ = } m_1 \texttt{ and } \ldots \texttt{ and } i_n \texttt{ = } m_n \Rightarrow \\ i_1 \texttt{ = } m' \texttt{ . } i_1\texttt{;} \ \ldots \texttt{;} \ i_n \texttt{ = } m' \texttt{ . } i_n \end{array}} \tag{340}$$

Then we supply the rule for `REC-SP`. In the same way as in the case of `IND-DULE`, the environments are not used in any recursive manner. Even more — the specification names $i_1, \ldots, i_n$ are not visible inside the specifications $p_1, \ldots, p_n$ and similarly for the (co)inductive modules. The `subtract` operations in rules (333), (335) and (336) enforce this from the outside.

$$\frac{\begin{array}{c} PE, ME \vdash p_1 \Rightarrow p'_1 \\ \vdots \\ PE, ME \vdash p_n \Rightarrow p'_n \\ PE' = i_1 \texttt{ = } p'_1\texttt{;} \ \ldots \texttt{;} \ i_n \texttt{ = } p'_n \end{array}}{\begin{array}{c} PE, ME \vdash i_1 \texttt{ = } p_1 \texttt{ and } \ldots \texttt{ and } i_n \texttt{ = } p_n \Rightarrow \\ i_1 \texttt{ = rec } i_1 \ \texttt{\{}PE'\texttt{\};} \ \ldots \texttt{;} \ i_n \texttt{ = rec } i_n \ \texttt{\{}PE'\texttt{\}} \end{array}} \tag{341}$$

9.2.4 User module language

Here we define the language accessible to the user. The details of the grammar, and the implementation of the translation can be found in files describing the grammar for the universal parsing tools. The description of the LR parser for the OCaml Yacc [105] is in `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/parser.mly` an the description of the LL parser for CamlP4 [37] is in `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/ll_parser.ml`.

Syntax

As with the user core language, there is no abstract syntax and the presented concrete syntax is parsed directly to the syntax of the environment dependent language. The syntactic domains `TYPE` and `VALUE` are taken from the user core language.

```
FIELD-SP   ::= DULE-LABEL : SP
             | DULE-LABEL
PARAM-SP   ::= ~ DULE-LABEL : SP
             | ~ DULE-LABEL
BB-ITEM    ::= type TYPE-LABEL
             | value VALUE-LABEL : TYPE
SP         ::= { FIELD-SP ;...; FIELD-SP }
             | {{ FIELD-SP ;...; FIELD-SP }}
             | PARAM-SP...PARAM-SP -> SP
             | sig BB-ITEM...BB-ITEM end
             | SP with DULE
             | SP-LABEL
             | ( SP )

FIELD-DULE ::= DULE-LABEL = DULE
             | DULE-LABEL
BASE-ITEM  ::= type TYPE-LABEL = TYPE
             | value VALUE-LABEL = VALUE
             | value rec VALUE-LABEL = VALUE
DEF-DULE   ::= DULE-LABEL = DULE
DEF-SP     ::= SP-LABEL = SP
AND-DULE   ::= and DEF-DULE
ONE-DULE   ::= DEF-DULE AND-DULE...AND-DULE
AND-SP     ::= and DEF-SP
ONE-SP     ::= DEF-SP AND-SP...AND-SP
LINK-ITEM  ::= DEF-DULE
             | module DEF-DULE
             | module ind ONE-DULE
             | module coind ONE-DULE
             | spec DEF-SP
             | spec rec ONE-SP
             | library DEF-DULE
             | library ind ONE-DULE
             | library coind ONE-DULE
DULE       ::= DULE-LABEL
             | { FIELD-DULE ;...; FIELD-DULE }
             | {{ FIELD-DULE ;...; FIELD-DULE }}
             | :: SP DULE
             | struct BASE-ITEM...BASE-ITEM end
             | DULE "|" DULE
             | DULE with DULE
             | DULE :> SP
             | link LINK-ITEM...LINK-ITEM end
             | load DULE-LABEL
             | ( DULE )

START      ::= LINK-ITEM...LINK-ITEM
```

Translation

Here we show how the syntactic sugar of the user module language translates to the constructions of the environment-dependent module language. The terms of all the domains of the user language translate to terms of the domain with the same name of the environment-dependent language. The terms of the user core language embedded into the module phrases translate to the terms of the bare core language according to the translation of Section 8.3.2.

While describing the translation, we use the same conventions as those for the case of the user core language. The translation is very straightforward, since most of the syntactic domain definitions in the grammar of the user language and in the grammar of the environment-dependent language are identical.

```
FIELD-SP   ::: i : p                --> i : p
             | i                    --> i : i
PARAM-SP   ::: ~i:p                 --> ~i:p
             | ~i                   --> ~i:i
BB-ITEM    ::: type i               --> type i
             | value i : f          --> value i : f
SP         ::: {fi1; ...; fin}      --> {fi1; ...; fin}
             | {{fi1; ...; fin}     --> {{fi1; ...; fin}}
             | a1 ... an -> p       --> a1 ... an -> p
             | sig b1 ... bn end    --> sig b1 ... bn end
             | p with m             --> p with m
             | i                    --> i
             | (p)                  --> p

FIELD-DULE ::: i = m                --> i = m
             | i                    --> i = i
BASE-ITEM  ::: type i = f           --> type i = f
             | value i = t          --> value i = t
             | value rec i = t      --> value i = fix i: t
DEF-DULE   ::: i = m                --> i = m
DEF-SP     ::: i = p                --> i = p
AND-DULE   ::: and d                --> and d
ONE-DULE   ::: d o1 ... on          --> d o1 ... on
AND-SP     ::: and d                --> and d
ONE-SP     ::: d o1 ... on          --> d o1 ... on
LINK-ITEM  ::: d                    --> module d
             | module d             --> module d
             | module ind o         --> module ind o
             | module coind o       --> module coind o
             | spec d               --> spec d
             | spec rec o           --> spec rec o
             | library d            --> library d
             | library ind o        --> library ind o
             | library coind o      --> library coind o
DULE       ::: i                    --> i
```

```
                | {fi1; ...; fin}      --> {fi1; ...; fin}
                | {{fi1; ...; fin}}    --> {{fi1; ...; fin}}
                | :: p m               --> :: p m
                | struct b1 ... bn end --> struct b1 ... bn end
                | m1 "|" m2            --> m1 "|" m2
                | m2 with m1           --> m2 with m1
                | m :> p               --> m :> p
                | link d1 ... dn end   --> link d1 ... dn end
                | load i               --> load i
                | (m)                  --> m

START      ::: d1 ... dn               --> d1 ... dn
```

9.2.5 Conclusion

We have defined three extended module languages and their semantics in the internal module language. With the first of the extended languages, the bare module language, we introduce the notion of a module specification. A specification is a joint notation for the domain and the codomain of a module. It expresses in clearer and more suggestive way the usual dependency of values of the codomain on the types of the domain. Perhaps surprisingly, specifications can be nested, despite the lack of exponential object in our module category. We have provided a translation, called saturation, which flattens the pseudo-higher-order specifications, while preserving their type dependencies. Using the saturation we have also managed to resolve mutual dependencies of recursive specifications by enriching each of them with types of all others.

Nondeterminism in the rules of saturation allows us to model nontrivial specification inference and consequently remove most of signature operands from module operations. Our specification reconstruction algorithm usually successfully resolves such ambiguities and then it produces the least detailed specification of a module; otherwise it fails. We limit the nondeterministic behavior in a larger scale by introducing the operations of isolating specifications and modules. The operations are used in the environment-dependent module language to fix the places at which specification reconstruction should occur during compilation.

We have shown that every specification derivable for a module is saturated (Observation 9.2.7) and expressible in the bare module language (Observation 9.2.10). We have demonstrated that any specification inferred for a bare language module agrees with the domain and codomain of the internal language semantics of the module (Observation 9.2.9). We have proved that due to excluding module composition from the bare language, almost everything that type-checks in the bare language has an internal module language semantics (Corollary 9.2.15).

The environment-dependent module language allows the user to name specifications and libraries. The user module language provides some more syntactic sugar. Both these languages are easily translatable to the bare module language without performing typing of modules. The introduced shorthands and default assignment of specifications, together with the linking operation and implicit sharing, almost totally eliminate module headers, which in our preferred fine-grained modularization style might easily constitute more than a half of the program code.

Part IV

Conclusion

Chapter 10

Final remarks

In our thesis we design a programming language Dule with a module system and their constructive semantics in an abstract categorical model. Our programming language features mechanisms inspired by category theory, enabling fine-grained modularization without prohibitive programming overhead. Our categorical model is simple and general, admitting many variants and extensions, such as data types expressible as adjunctions, (co)inductive constructions and mutually dependent modules.

The abstract nature of our mathematical model allows us to prove fundamental properties of programming notions, such as module grouping with sharing requirements or the mapping combinator. On the other hand, our model is linked to program execution via combinator reduction system we develop in our thesis. Our compiler of Dule, after type and specification reconstruction, computes the semantics of modular programs in the categorical model and verifies its type-correctness, almost literally following formal semantic definitions. All example programs given in our thesis compile successfully to the language of the base part of our categorical model and yield the expected results through the implemented combinator reduction system.

We find fascinating the immediate practical influence our theoretical results provide in this framework. The formalism allows us to state many interesting questions, many of which are still unanswered. Numerous programming language improvements and extensions will surely follow. In the subsequent sections we sketch a few questions, problems and tasks we would like to see undertaken in the immediate future.

10.1 Theory

We show how to extend the categorical model of our module system with data types expressible as adjunctions, with (co)inductive constructions and with functors of mixed variance; in particular, with the fully parameterized exponents. It would be interesting to make a detailed comparison of our construction enabling functors of mixed variance, developed in Section 7.2.4, with the construction of dinatural transformations [10] and with the formalism of involutory categories [48]. In Section 8.1.4 we present our construction as an almost 2-category, by restricting horizontal composition. Perhaps this kind of categories coincides with some known concept of category theory, such as a variant of weak n-categories. Otherwise, one could try to investigate the properties of this new notion, regardless of its use for hosting functors of mixed variance and parameterized adjunctions.

In our thesis, on the basis of the extended model and without introducing recursive types we define the mechanism of mutual dependency among modules, in the form of (co)inductive module construction. Our module system contains the essential mechanisms of modular programming, such as type sharing, transparent as well as non-transparent functor application and grouping of related modules. However, our module system features no higher-order modules. We conjecture that SCM, our category of modules, has no exponent, even if its core language *2-LC-lc* has exponents at all three levels, and even if the definition of signatures and modules is substantially refined. The reason is that system F is not set-theoretic [127], while SCM is, as seen in Section 5.1, and the module exponent would enable a simulation of system F (because SCM treats all modules as, roughly, base modules and such treatment of higher-order modules would allow self-application). But perhaps it is possible to add higher-order functors after all, not into the SCM, but at the outer layer of semantics? Or maybe some advanced external tools for managing libraries for code reuse — a library database and semi-automated code-adjusting engine [9] — would be a better investment of effort?

To increase the usefulness of the mapping combinator, introduced in Section 6.3, we have to experiment with extending its rewriting rules for the case of function types. The general categorical model is already in place, just as all the necessary syntax, other rewriting rules and an analysis of rewriting of general multiplication by a functor in Section 7.3.4. We would suggest that the rules are proposed first for strictly positive types, then for types with only positive occurrences of type variables and then for simple cases of mixed variance such as X-contravariance of Section 7.2.4.

We expect there is a universal categorical characterization of the mapping combinator, similar, though much more general (meta-polytypic), as for the other combinators of our categorical model. Currently, the definition is by cases on all other combinators, and so it is complete only for reachable models. We do not expect the characterization to be used as a rewrite rule, but it would be very unfortunate if all such characterizations proved to be completely external to our formalism. If a characterization can be expressed in the language of our categories, it would aid in verification and systemization of the mapping combinator equations and rules, just as our general framework of adjunctions does for other combinators.

We would like to verify the nine named conjectures stated in our thesis. First of all, Conjectures 6.3.10, 8.2.1, 9.2.6 and 9.2.8, on which the correctness and coherence of introduced formalisms depend. We are convinced they hold by, in addition to other arguments, the analysis and testing of numerous examples using the Dule compiler. Moreover, if Conjectures 9.2.6 and 9.2.8 fail and the least detailed specifications are not guaranteed to exist, no mandatory modifications to the specification reconstruction algorithm are induced, since it is already inherently partial (see Section 10.4). We would need, however, to complicate the semantics by acknowledging the partiality in the typing rules for modules and specifications that refer to the least detailed specifications.

Then we would like to verify somewhat less crucial Conjectures 6.3.7, 8.1.3 and 8.2.4 that describe properties of our reduction systems and which, given the size and complexity of the systems, would be best verified using specialized proof assistants or, if the semi-automatic methods fail, by adapting known proofs of related systems. Then we could attempt Conjectures 6.3.2 and 7.1.6 that would be nice to have, but their proofs seem troublesome. At last we would like to tackle some auxiliary conjectures that are stated informally and their proofs would require the introduction of additional formalisms.

One of most interesting future extensions to our module language, the structured recursion over types produced by the (co)inductive module construction discussed in Section 9.1, necessitates additional theoretical work. Both the programming methodology employing such recursion and the implementation issues seem to be very promising topics of research, but first we have to overcome some problems with extending the core language rewriting to basic operations typed with complex kinds.

10.2 Pragmatics

The module system and programming language Dule resulting from our research turns out to be quite rich and very different from other programming languages. Due to its novelty, our language still has many rough edges. However, we are convinced we have overcome what we perceive the main obstacles to practical modular programming. The operations available in our module system provide for modular programming style with no mandatory module headers, no explicit module applications and no hard to localize module errors. The model of our module system facilitates viewing the same module with different levels of abstraction. A new methodology resulting from our study of inductive types grants precise control over the concreteness of module interfaces. The grouping of modules can be done is several powerful and specialized ways, without verbose notation and without sacrificing abstraction.

The Dule language needs a lot of additional case studies, especially concerning its module system. For serious large-scale experiments, a serious set of libraries should be designed, which themselves will be substantial case studies of the Dule modular programming style. We should explore and identify all modularization methodologies favored by our module system and analyze the impact of Dule modularization style on the core language programming. After a form of formal specifications (e.g., specifications of Section 9.2 enriched with axioms) is chosen for the module system, the categorical model should be extended accordingly and methodologies studied once again. The specification specialization operation of our module system, as introduced in Section 5.2.5, suggests that the programming language code will itself be a part of the logic used for specifying modules (so that some specialized specifications could be expressed as base specifications with axioms containing code, similar to the axioms of Extended ML [92]).

We would like to explore possible programming idioms for multiple specifications of a single module and find one that is general enough and avoids multiple projections, as well as instantiations and libraries, see Section 2.4.2. Perhaps this can be achieved with a new programming style, or perhaps some syntactic sugar is needed, or some new scoping rules should be proposed. The scoping rules for library and specification names evolve already: those recently implemented in the compiler ban specification overwriting, while those defined in Section 9.2.3 remain less restrictive (the compiler works according to the definition given here with the `--overwriting` option). The rules have to be rethought based on case studies. Such shallow modifications and variations are easy in our framework, because they are naturally confined to the semantics of the environment-dependent module language.

An interesting concept and methodology is the merger of operations of algebras and co-algebras sharing the same carrier [44]. The concept is already expressible in Dule by defining a co-algebra inside a module and providing both co-algebraic and algebraic conversion operations for the carrier (such as `t2ind` on page 54). However, experiments are needed to find out in which cases such programming discipline is practical and if we should extend the language and categorical model in order to increase the usability and efficiency of compilation for this construction. Another methodology applicable to Dule programming is the strictly categorical programming style of Charity [23]. We would like to see the elegant and intricate example Charity programs [22] translated to Dule, possibly with an automatic translator. Perhaps Dule could even be used as a module system for the Charity programming language (in the version of Charity extended with exponents). We hope the ideas underlying our module system can be applied to improve other systems, in particular the OCaml module system. However, using the original Dule module system to structure OCaml programs would require an extension to the OCaml core language itself, even after the recent addition of (almost) arbitrary fixpoints to OCaml.

The notion dual to the exponent, the coexponent [46], could be used to model co-variables or exceptions shared among all values in a module. Both rewriting of coexponent and the pragmatics of co-variables would be an original topic of research. One may also check if any other adjunction expressible in our formalism is interesting from the programming point of view, or if parameterization of any other adjunction than the exponent makes sense. See the remarks at the end of Section 7.1.3.

We would like to improve pattern matching in Dule. In particular, the preliminary version of record patterns should be improved and integrated with a recently added default variant notation that defines the outcome of a case analysis expression for all unlisted variants of a sum type. The latter extension required changes at all levels of the compiler, even down to the abstract machine (combinator reduction) and is not yet sufficiently tested. We would also like to implement a "three dots" notation for records and this extension should be doable at the type reconstruction level, or in other words, in the bare core language of Section 8.3.1. Perhaps some other abbreviations we introduce for presenting reduction rules could be implemented in the Dule language as well. We wonder how Haskell monads would fit into our categorical framework, in particular when used according to the methodology of merging monads and folds [116]. With monads, the Haskell list comprehensions generalized to other datatypes could possibly also be added.

10.3 Implementation

The semantics of Dule is simple and constructive enough that the design of a Dule compiler straightforwardly implementing the semantics was possible; see Appendix C.2 and the Dule homepage at `http://www.mimuw.edu.pl/~mikon/dule.html`. On the other hand, the language proved powerful enough to facilitate writing, in an extremely modular and yet concise manner, various example Dule programs, including the main parts of Dule compiler itself. However, the Dule version of the compiler lacks most boring, basic tools and technical components, and is not planned to bootstrap in the near future. The example programs can be type-checked, compiled and executed in the OCaml version of the Dule compiler. To help in identifying programmer's (or compiler's) errors, some rudimentary error reporting facilities are implemented, but many more improvements, programming tools and manuals are needed to make programming in Dule comfortable.

To enable efficient recompilation, Dule would require a version of the "`make`" utility that is not tied to files but recompiles only those of the many small modules contained in a file that are changed. The implementation should also take advantage of other aspects of separate compilation in Dule, such as the possibility to salvage compiled code of all unchanged modules of a set of mutually dependent modules, as discussed in Section 9.1.1. We would also like to find a way to provide low-level libraries, or even arbitrary C libraries (or OCaml libraries, for a start) as a fake compiled code with a Dule-expressible specification. To enable bootstrapping of the Dule compiler written in Dule we will also need to interface a parser generator (e.g., `ocamlyacc`) to Dule.

We should improve readability of compiler reports about typing errors and efficiency of type-checking, compilation and execution. In particular, the execution of code containing arbitrary fixpoints (defined in Section 8.2) should be improved [74]. Generally, we would like to experiment some more with the evaluation mechanism in our programming language, taking ideas from the Categorical Abstract Machine [31], Charity abstract machine [146], type-free reduction of Covariant Types [86], Typed Intermediate Language [139], Typed Assembly Language [124] and others. Staying with our conventional combinator reduction, we can modify reduction rules and, for a fixed set of rules, we can try to guess which evaluation strategy is the best. Our set of rewriting rules admits many more strategies than the standard eager/lazy options of λ-calculus. As a source for additional reduction rules, we can consider the portions of the developed equational theories that are not yet used, especially the equalities of exponents given in Section 8.1.3, or investigate the additional equalities of the theories of initial models. Currently we experiment with a version of our proposed reduction system with reduction restrictions and, despite many tries and even hash-consing of generated code, we find a generalization of the weak head normal form restriction (avoiding reduction under `T_curry`, `TL_case`, `TL_fold` and `TL_unfold` combinators) to be necessary for efficient evaluation in the presence of fixpoints.

The code, in Section 5.2.4, defining the product of module signatures should be simplified more, without affecting the overall semantics. We may also consider implementing the linking combinator in a more conventional manner, in which the compiled modules are not composed and reduced (with possible copying of some code), but instead the compiled bodies stay separated and their functions are called whenever necessary. Perhaps a version of this idea is implementable just by changing or restricting the reduction rules for core language records.

10.4 Type reconstruction algorithms

Despite the mandatory types of values listed in specifications, Dule programs require type reconstruction. First of all, there can be locally defined values with undeclared types, and secondly, there is significant ambiguity of typing introduced by the anonymous sum and record types, the explicit (co)inductive types, the built in (co)inductive combinators and mapping. The typing rules and an introduction to type reconstruction, including some related fundamental definitions, are given in Section 8.3.1.

Similarly, although the preferred coding style in the module language assumes explicit specifications of almost all basic modules, the module projections, which can be separately compiled, and the various grouping operations (as defined in Sections 5.3 and 9.1) introduce a lot of ambiguity that has to be resolved by specification reconstruction. Module typing rules and introductory remarks about specification reconstruction are located in Section 9.2.1.

Our type reconstruction algorithm for bare core language values, unlike the classical Hindley-Milner algorithm for the ML type system, features deferred unification and uses an additional kind of type variables representing indexed lists of types. The current, implemented version of the algorithm seems to be complete, but the proof has yet to be attempted. The efficiency of the algorithm, as well as of the nonstandard substitution and unification algorithms have to be improved. Then we would like to verify that the typing reconstructed by the algorithm is minimal in the sense of Definition 8.3.3 and study the possible choices of minimal typings from the perspective of programming practice. We would also like to prove that the complexity of the reconstruction algorithm is polynomial, assuming a constant bound on the nesting of the mapping combinator. We also conjecture the unrestricted problem is exponential, due to the thorough state space search when unifying types generated by nested mapping.

Our specification reconstruction is general and light-weight, by not using the semantics of the core language. Only the completely reconstructed least detailed specifications (if reconstruction does not fail) are checked for soundness using core language type-checking. We suspect that any specification reconstruction based on our typing rules for modules must be partial, because it has to approximate unification modulo the semantic equality (very nontrivial on the module level) by a unification with two kinds of signature variables, performed modulo a simple syntactic equality. We wonder if the algorithm can be made complete without sacrificing compositionality that is crucial for separate compilation. The main problems with the unification are that the module product is not injective (see Section 5.2.4) and that the conditions for interchange of signature specialization with signature product (as stated in Lemma 5.2.8 in Section 5.2.5) are not always met. However, for natural examples, as those in Appendix C.2, the algorithm succeeds and only in very special cases the reconstruction has to be aided with spurious specification annotations.

Part V

Appendices

Appendix A

Language summaries

The core and module internal languages are defined piece by piece. The pieces are located in different chapters of our thesis. On the other hand, the derived, more user friendly languages are each presented in one piece and their descriptions refer to the internal language as a whole. To make studying the definitions based on the internal languages easier, we summarize all the syntax and semantics of the mainstream variant of the core and module internal languages. There is also a reformulation of their typing definitions using the more readable form of typing rules. The reformulation enables direct comparison with derived languages, in which the typing is always presented using typing rules.

A.1 Internal core language

A Dule program (including modules) is compiled to the internal core language. The model of the language is, approximately, a *2-LC-lc* extended with various operations. We have been building this framework incrementally throughout many chapters ending in the section devoted to the core language f-Core. Here is the summary.

A.1.1 Syntax

We will base our presentation on the abstract syntax. A concrete, more readable syntax of some operations will be given when we describe their semantics. While the concrete syntax is usually more readable, part of the readability comes from the fact that some of the typing information is omitted. Usually the typing and consequently the semantics may be recovered from the context, but we rely on the abstract syntax for cases where this is not possible. Moreover, we appreciate having a link to the exemplary implementation of the Dule compiler, where all the details of the semantics are spelled out using the abstract syntax. The Dule compiler defines the abstract syntax in modules `FCat`, `FFunct` and `FTrans`, which are all located in the file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/core_back.ml`.

The syntax presented below is a raw abstract syntax. For a term to belong to the language it has to be type-correct as described in the next sections.

```
type cat =
  | C_PP of cat IList.t
  | C_BB
type funct =
  | F_ID of cat
  | F_COMP of funct * funct
  | F_PR of cat IList.t * IdIndex.t
  | F_RECORD of cat * funct IList.t
  | F_pp of cat * funct IList.t
  | F_ss of cat * funct IList.t
  | F_ii of funct
  | F_tt of funct
  | F_ee of funct IList.t * funct
type trans =
  | T_ID of cat
  | T_COMP of trans * trans
  | T_PR of cat IList.t * IdIndex.t
  | T_RECORD of cat * trans IList.t
  | T_FT of funct * trans
  | T_TF of trans * funct
  | T_id of funct
  | T_comp of trans * trans
  | T_pp of cat * trans IList.t
  | T_pr of funct IList.t * IdIndex.t
  | T_record of funct * trans IList.t
  | T_ss of cat * trans IList.t
  | T_in of funct IList.t * IdIndex.t
  | T_case of trans IList.t * funct
  | TL_case of funct * trans IList.t * funct
  | T_map of funct * trans
  | T_ii of trans
  | T_con of funct
  | T_fold of funct * trans
  | TL_fold of funct * trans
  | T_de of funct
  | T_tt of trans
```

```
    | T_uncon of funct
    | T_unfold of funct * trans
    | TL_unfold of funct * trans
    | T_unde of funct
    | T_ee of trans IList.t * trans
    | T_appl of funct IList.t * funct
    | T_curry of trans
    | T_fix of trans
    | TL_fix of trans
```

While the sub-languages of `cat`-terms and `funct`-terms are very concise, the language of `trans`-terms is by far not the smallest among those with the same expressive power, whether taking into account the equational theory or the reduction system (as given by the complete set of reduction rules for our language listed below, in Section A.1.4).

Rules (39), (40), (55), (77), (123), (124) and (241) show that the term constructors `T_ID`, `T_PR`, `T_pp`, `T_ss`, `T_ii`, `T_tt` and `T_ee`, respectively, are not essential for the expressiveness of the language. Rules 116–122 show that `T_map` is expressible using the other term constructors. Rules (78), (125), (126) and (252) show that `T_case`, `T_fold`, `T_unfold` and `T_fix` may be omitted, too. Additionally, at the cost of some decline in performance, `T_de` and `T_uncon` may be expressed using the other operations. The multiplications by a functor could easily be eliminated using horizontal composition with an identity transformation. The minimal grammar of `trans`-terms looks as follows.

```
  type trans =
    | T_COMP of trans * trans
    | T_RECORD of cat * trans IList.t
    | T_id of funct
    | T_comp of trans * trans
    | T_pr of funct IList.t * IdIndex.t
    | T_record of funct * trans IList.t
    | T_in of funct IList.t * IdIndex.t
    | TL_case of funct * trans IList.t * funct
    | T_con of funct
    | TL_fold of funct * trans
    | TL_unfold of funct * trans
    | T_unde of funct
    | T_appl of funct IList.t * funct
    | T_curry of trans
    | TL_fix of trans
```

There are also other ways of expressing some operations. For example, in the presence of the exponent functor, the term constructors `TL_case`, `TL_fold`, `TL_unfold` and `TL_fix` are expressible using `T_case`, `T_fold`, `T_unfold` and `T_fix`, respectively. Another possibility is to eliminate `TL_fold` and `TL_unfold` as well as `T_fold` and `T_unfold` and express the structured (co)recursion using `T_con`, `T_de`, `T_uncon`, `T_unde` and general recursion, for example `TL_fix`.

Every set of basic operations has its assets both in terms of ease of use and complexity of rewriting. For these reasons we include all the operations and, while providing a set of rewrite rules biased towards the minimal grammar presented above, we leave the way open for other implementations of the reduction.

A.1.2 Semantics

The model in which we give the semantics of the operations, is an (approximately) 2-category with some additional structure. More precisely the model consists of the underlying category as well as the category of objects and 2-morphisms with horizontal composition. These two categories relate to each other as in a 2-category. There is also a category $C(c, e)$ for each pair of objects (c, e).

As discussed in Section 7.2.4, with respect to a generic 2-category we lack the interchange law (20) — the compositions in categories $C(c, e)$ do not interchange with each other with respect to the horizontal composition. Since there is no explicit mention of the interchange law in any of the definitions of operations of *2-LC-lc*, as well as its extensions and the module languages, each of them can be assigned meaning in this model without problems. The interchange law is cited only in the descriptions of equational theories and, consequently, some details of the semantics differ, whenever the theories that capture them are affected by the restriction of the applicability of the interchange law. Here we present the overview of the semantics, without delving into the details precisely captured by the numerous theories and translations constructed throughout the thesis.

As discussed in Section 8.1.4, since the model permits parameterized adjunctions, the exponent operations are legal. By banning both multiplications and horizontal compositions that involve an exponent functor in the second operand we allow the typing of the operations to be as simple as in *2-LC-lc*. Let us assume we have a fixed categorical model of the core internal language. The semantic function for the whole internal language in the categorical model is written $[\![\_]\!]_i$.

The semantics of `cat`-terms is as follows.

- `C_PP`(lc)
  written "`<`i_1 `-` c_1 `;` i_2 `-` c_2 `;` $\ldots$ `;` i_n `-` c_n`>`"
  is a labeled product of an indexed list of categories lc,

- `C_BB`
  written "`*`"
  is a distinguished constant, called the base category.

The semantics of `funct`-terms, presented below, is sometimes too general, allowing morphisms that are not needed in the language, like products with target other that `*`. The more strict typing of the operations, in this respect, is given in the next section.

- `F_ID`$(c) : c \to c$
  is the identity functor on category c,

- `F_COMP`$(f, g) : c \to e$
  written "f `.` g"
  is the composition of $f : c \to d$ and $g : d \to e$,

- `F_PR`(lc, i) : `<`lc`>` $\to c_i$
  written "i"
  is the projection at label i, where the i-th element of lc called c_i is required to be present,

- `F_RECORD`$(c, lf) : c \to$ `<`le`>`
  written "`<`i_1 `:` f_1`;` i_2 `:` f_2`;` $\ldots$`;` i_n `:` f_n`>`"
  is the record in the underlying category (labeled tuple, universal morphism of the product) of an indexed list of functors lf, where the elements of lf are $f_i : c \to e_i$,

- `F_pp`$(c, lf) : c \to e$
  written "`{`i_1 `:` f_1`;` i_2 `:` f_2`;` $\ldots$`;` i_n `:` f_n`}`"
  is a labeled product of an indexed list of functors lf (in the category $C(c, e)$, where functors are objects), where $f_i : c \to e$,

- `F_ss`$(c, lf) : c \to e$
  written "`[`‘i_1 f_1`|`‘i_2 f_2`|` $\ldots$`|`‘i_n f_n`]`", where "‘" is back-quote,
  is a labeled coproduct in $C(c, e)$ of an indexed list of objects (that is functors) lf, where $f_i : c \to e$,

- `F_ii`$(g) : d \to c$
  is an initial algebra of a functor g : `<`ι `-` c`;` κ `-` d`>` $\to c$, where ι and κ are distinguished labels and the induction is over the ι component while the κ component is kept for parameters,

- `F_tt`$(g) : d \to c$
  is a terminal co-algebra of a functor g : `<`ι `-` c`;` κ `-` d`>` $\to c$, where the co-induction is over the ι component and the κ component is kept for parameters,

- F_ee$(lg, h) : c \to e$
  is a parameterized labeled exponent object (that is functor) in $C(c, e)$ where $g_i : c \to e$ and $h : c \to e$.

The semantics of trans-terms is a bit too general just as the semantics of funct-terms. The typing will be written more strictly in the next section. We use here the notation $g[h]$ introduced in Section 6.3.2.

- T_ID(c) : F_ID$(c) \to$ F_ID(c)
  is the identity in the category of objects and 2-morphisms,

- T_COMP(t_1, t_2) : F_COMP$(f_1, f_2) \to$ F_COMP(h_1, h_2)
  written "t_1 * t_2"
  is the composition in the category of objects and 2-morphisms (horizontal composition) of $t_1 : f_1 \to h_1$ and $t_2 : f_2 \to h_2$, where $f_1, h_1 : c \to d$ and $f_2, h_2 : d \to e$,

- T_PR(lc, i) : F_PR$(lc, i) \to$ F_PR(lc, i)
  is the projection in the category of objects and 2-morphisms,

- T_RECORD(c, lt) : F_RECORD$(c, lf) \to$ F_RECORD(c, lh)
  written "<i_1 = t_1; i_2 = t_2; ...; t_n = f_n>"
  is the record (labeled tuple) in the category of objects and 2-morphisms of an indexed list of transformations lt, where $t_i : f_i \to h_i$ and $f_i, h_i : c \to e_i$,

- T_FT(f_1, t_2) : F_COMP$(f_1, f_2) \to$ F_COMP(f_1, h_2)
  written "f_1 * t_2"
  is the multiplication of the transformation $t_2 : f_2 \to h_2$ by the functor f_1 from the left, where $f_1 : c \to d$ and $f_2, h_2 : d \to e$,

- T_TF(t_1, f_2) : F_COMP$(f_1, f_2) \to$ F_COMP(h_1, f_2)
  written "t_1 * f_2"
  is the multiplication of the transformation $t_1 : f_1 \to h_1$ by the functor f_2 from the right, where $f_1, h_1 : c \to d$ and $f_2 : d \to e$,

- T_id$(g) : g \to g$
  written ": g"
  is the 2-identity on $g : c \to e$, that is the identity on object g in the category $C(c, e)$,

- T_comp$(t, u) : f \to h$
  written "t . u"
  is the composition in $C(c, e)$ (vertical composition) of $t : f \to g$ and $u : g \to h$, where $f, g, h : c \to e$,

- `T_pp`(c, lt) : `F_pp`$(c, lf) \to$ `F_pp`(c, lh)
  is the action of the labeled product functor on morphisms lt of $C(c, e)$ where lt is an indexed list of transformations such that $t_i : f_i \to h_i$ and $f_i, h_i : c \to e$,

- `T_pr`(lf, i) : `F_pp`$(c, lf) \to f_i$
  written "i"
  is the projection in $C(c, e)$, where $f_k : c \to e$,

- `T_record`$(f, lt) : f \to$ `F_pp`(c, lh)
  written "`{`i_1 `=` t_1`;` i_2 `=` t_2`;` ... `;` i_n `=` t_n`}`"
  is the record (labeled tuple) in $C(c, e)$ of an indexed list of transformations lt, where $t_i : f \to h_i$ and $f, h_i : c \to e$,

- `T_ss`(c, lt) : `F_ss`$(c, lf) \to$ `F_ss`(c, lh)
  is the action of the coproduct functor on morphisms of $C(c, e)$ where lt is an indexed list of transformations, such that $t_i : f_i \to h_i$ and $f_i, h_i : c \to e$,

- `T_in`$(lf, i) : f_i \to$ `F_ss`(c, lf)
  written "`‘`i", where "`‘`" is back-quote,
  is the injection morphism in $C(c, e)$, where $f_k : c \to e$,

- `T_case`(lt, h) : `F_ss`$(c, lf) \to h$
  is the case expression (variant analysis) of an indexed list of transformations lt, where $t_i : f_i \to h$ and $f_i, h : c \to e$.

- `TL_case`(f, lt, h) : `{`δ `:` `F_ss`(c, lg)`;` ϵ `:` f`}` $\to h$
  written "`[‘`i_1 t_1`|‘`i_2 t_2`|` ... `|‘`i_n t_n`]`"
  is the distributive case expression of an indexed list of transformations lt, where δ and ϵ are distinguished labels, case analysis is performed over the δ components, t_i : `{`δ `:` g_i`;` ϵ `:` f`}` $\to h$ and $f, g_i, h : c \to e$,

- `T_map`(g, t) : `{`δ `:` $g[f_\delta]$`;` ϵ `:` f_ϵ`}` $\to g[h]$
  is the action of the ι component of the functor g : `<`ι `-` c`;` κ `-` d`>` $\to c$, on the δ component of the transformation t : `{`δ `:` f_δ`;` ϵ `:` f_ϵ`}` $\to h$ where $f_\delta, f_\epsilon, h : d \to c$,

- `T_ii`(t) : `F_ii`$(f) \to$ `F_ii`(h)
  is the action of the initial algebra functor on transformation $t : f \to h$, where f, h : `<`ι `-` c`;` κ `-` d`>` $\to c$,

- `T_con`(g) : $g[$`F_ii`$(g)] \to$ `F_ii`(g)
  is the constructor morphism of the initial algebra for a functor g : `<`ι `-` c`;` κ `-` d`>` $\to c$,

- `T_fold`(g, t) : `F_ii`$(g) \to h$
  is the iteration (folding) over transformation $t : g[h] \to h$ in the initial algebra for a functor g : <ι - c; κ - d> $\to c$, where $h : d \to c$,

- `TL_fold`(g, t) : {δ : `F_ii`(g); ϵ : f} $\to h$
  is the iteration (folding) over the δ component of transformation t : {δ : $g[h]$; ϵ : f} $\to h$, in the initial algebra for a functor g : <ι - c; κ - d> $\to c$, where $f, h : d \to c$,

- `T_de`(g) : `F_ii`$(g) \to g[$`F_ii`$(g)]$
  is the destructor morphism of the initial algebra for a functor g : <ι - c; κ - d> $\to c$,

- `T_tt`(t) : `F_tt`$(f) \to$ `F_tt`(h)
  is the action of the terminal co-algebra functor on transformation $t : f \to h$, where f, h : <ι - c; κ - d> $\to c$,

- `T_uncon`(g) : $g[$`F_tt`$(g)] \to$ `F_tt`(g)
  is the constructor morphism of the terminal co-algebra for a functor g : <ι - c; κ - d> $\to c$,

- `T_unfold`(g, t) : $h \to$ `F_tt`(g)
  is the co-iteration (unfolding) over transformation $t : h \to g[h]$ in the terminal co-algebra for a functor g : <ι - c; κ - d> $\to c$, where $h : d \to c$,

- `TL_unfold`(g, t) : {δ : h; ϵ : f} $\to$ `F_tt`(g)
  is the co-iteration (unfolding) over the δ component of transformation t : {δ : h; ϵ : f} $\to g[h]$, in the terminal co-algebra for a functor g : <ι - c; κ - d> $\to c$, where $f, h : d \to c$,

- `T_unde`(g) : `F_tt`$(g) \to g[$`F_tt`$(g)]$
  is the destructor morphism of the terminal co-algebra for a functor g : <ι - c; κ - d> $\to c$,

- `T_ee`(lt, u) : `F_ee`$(lh, f) \to$ `F_ee`(lf, h)
  is the action of the parameterized labeled exponent functor on morphisms of $C(c, e)$ where lt is an indexed list of transformations, such that $t_i : f_i \to h_i$, $f_i, h_i : c \to e$, and $u : f \to h$ is a transformation such that $f, h : c \to e$,

- `T_appl`(lg, h) : {υ : `F_ee`(lg, h); i_1 : g_1; $\ldots$} $\to h$
  is the parameterized evaluation (application) morphism in $C(c, e)$, where $g_k : c \to e$ and $h : c \to e$,

- `T_curry`(t) : $f \to$ `F_ee`(lg, h)
  is the parameterized curryfication of the morphism (that is transformation) t in $C(c, e)$, where t : {υ : f; i_1 : g_1; $\ldots$} $\to h$ and $f, g_i, h : c \to e$,

- $\texttt{T\_fix}(t) : \texttt{\{\}} \to h$
  is the fixpoint of transformation $t : h \to h$ where $h : c \to e$,

- $\texttt{TL\_fix}(t) : f \to h$
  is the fixpoint over the δ component of transformation $t : \texttt{\{}\delta \texttt{ : } h\texttt{;} \ \epsilon \texttt{ : } f\texttt{\}} \to h$ where $f, h : c \to e$.

A.1.3 Typing rules

The sources and targets of functors, and domains and codomains of transformations are strictly defined in the 2-categorical model. They are also explicitly written for each operation while defining its semantics in the model. Here we repeat the definition of typing of term constructors using typing rules and relying on the concrete syntax, whenever feasible. This time we do not over-generalize typing. Hence, for example, the labeled coproduct functor always has target `*`.

There is an implementation of a type-checker based on the typing system that allows the user to verify type-correctness of the compiled code (as opposed to reconstructing types of the user-written source code). The implementation resides in the module `ElabFCore` in file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/core_back.ml`. The Dule compiler verifies type-correctness of all generated code, unless the user invokes it with the `--no-verification` command line option.

The presentation below uses the conventions for typical identifiers described in Section 3.3. First, we assign the source and target categories to functors.

$$\frac{}{\texttt{F\_ID}(c) : c \to c} \tag{342}$$

$$\frac{f : c \to d \qquad g : d \to e}{f \texttt{ . } g : c \to e} \tag{343}$$

$$\frac{}{\texttt{F\_PR}(lc, i) : \texttt{<}i \texttt{ - } c\texttt{; } \dots\texttt{>} \to c} \tag{344}$$

$$\frac{f_1 : c \to e_1 \quad \cdots \quad f_n : c \to e_n}{\texttt{<}i_1 \texttt{ : } f_1\texttt{; } \dots\texttt{; } i_n \texttt{ : } f_n\texttt{>} : c \to \texttt{<}i_1 \texttt{ - } e_1\texttt{; } \dots\texttt{; } i_n \texttt{ - } e_n\texttt{>}} \tag{345}$$

$$\frac{f_1 : c \to \texttt{*} \quad \cdots \quad f_n : c \to \texttt{*}}{\texttt{\{}i_1 \texttt{ : } f_1\texttt{; } \dots\texttt{; } i_n \texttt{ : } f_n\texttt{\}} : c \to \texttt{*}} \tag{346}$$

$$\frac{f_1 : c \rightarrow \texttt{*} \quad \cdots \quad f_n : c \rightarrow \texttt{*}}{\texttt{[`}i_1\ \ f_1\texttt{|}\ \ldots\texttt{|`}i_n\ \ f_n\texttt{]} : c \rightarrow \texttt{*}} \tag{347}$$

$$\frac{g : \texttt{<}\iota\ \texttt{-}\ \texttt{*;}\ \kappa\ \texttt{-}\ d\texttt{>} \rightarrow \texttt{*}}{\texttt{F\_ii}(g) : d \rightarrow \texttt{*}} \tag{348}$$

$$\frac{g : \texttt{<}\iota\ \texttt{-}\ \texttt{*;}\ \kappa\ \texttt{-}\ d\texttt{>} \rightarrow \texttt{*}}{\texttt{F\_tt}(g) : d \rightarrow \texttt{*}} \tag{349}$$

$$\frac{g_1 : c \rightarrow \texttt{*} \quad \cdots \quad g_n : c \rightarrow \texttt{*} \qquad h : c \rightarrow \texttt{*}}{\texttt{F\_ee}(i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ g_n\texttt{,}\ h) : c \rightarrow \texttt{*}} \tag{350}$$

$$\frac{f : c \rightarrow d \qquad [\![c]\!]_i = [\![c']\!]_i \qquad [\![d]\!]_i = [\![d']\!]_i}{f : c' \rightarrow d'} \tag{351}$$

The last rule explicitly captures the fact that typing of `funct`-terms depends on the semantics of `cat`-terms in a model of the internal language at hand. The meta-variables in the informal descriptions of the semantics are interpreted up to semantic equivalence. However, the same meta-variables in typing rules are conventionally interpreted as denoting syntactically equal terms, so here and in few other places we add the rules explicitly relating typing to semantics. In particular, among the rules for assigning domain and codomain `funct`-terms to `trans`-terms below, the last one handles the semantic equivalence of `funct`-term.

Many of the rules below have additional, unwritten premises ensuring that the terms that appear in them have compatible source and target categories. For example, in rule (353) we require that the target of functor f_1 is equal to the source of functor f_2. Similarly in rule (367) and many others we require that the typing of functor g is such that the horizontal composition shorthand $g[\_]$ is well-typed. We could dispense of all the additional implicit premises by adding to the definition of type-correctness of `trans`-terms the requirement that the resulting domain and codomain functors should have derivable source and target categories.

$$\frac{}{\texttt{T\_ID}(c) : \texttt{F\_ID}(c) \rightarrow \texttt{F\_ID}(c)} \tag{352}$$

$$\frac{t_1 : f_1 \rightarrow h_1 \qquad t_2 : f_2 \rightarrow h_2}{t_1 \ast t_2 : f_1\ \texttt{.}\ f_2 \rightarrow h_1\ \texttt{.}\ h_2} \tag{353}$$

$$\frac{}{\texttt{T\_PR}(lc, i) : \texttt{F\_PR}(lc, i) \rightarrow \texttt{F\_PR}(lc, i)} \tag{354}$$

$$\frac{t_1 : f_1 \rightarrow h_1 \quad \cdots \quad t_n : f_n \rightarrow h_n}{\texttt{<}i_1 \texttt{ = } t_1\texttt{;} \ \ldots\texttt{;} \ i_n \texttt{ = } t_n\texttt{>} : \texttt{<}i_1 \texttt{ : } f_1\texttt{;} \ \ldots\texttt{>} \rightarrow \texttt{<}i_1 \texttt{ : } h_1\texttt{;} \ \ldots\texttt{>}} \tag{355}$$

$$\frac{t_2 : f_2 \rightarrow h_2}{f_1 * t_2 : f_1 \texttt{ . } f_2 \rightarrow f_1 \texttt{ . } h_2} \tag{356}$$

$$\frac{t_1 : f_1 \rightarrow h_1}{t_1 * f_2 : f_1 \texttt{ . } f_2 \rightarrow h_1 \texttt{ . } f_2} \tag{357}$$

$$\frac{}{\texttt{(: } g\texttt{)} : g \rightarrow g} \tag{358}$$

$$\frac{t : f \rightarrow g \qquad u : g \rightarrow h}{t \texttt{ . } u : f \rightarrow h} \tag{359}$$

$$\frac{t_1 : f_1 \rightarrow h_1 \quad \cdots \quad t_n : f_n \rightarrow h_n}{\texttt{T\_pp}(c, lt) : \texttt{F\_pp}(c, lf) \rightarrow \texttt{F\_pp}(c, lh)} \tag{360}$$

$$\frac{}{i : \texttt{\{}i \texttt{ : } f\texttt{;} \ \ldots\texttt{\}} \rightarrow f} \tag{361}$$

$$\frac{t_1 : f \rightarrow h_1 \quad \cdots \quad t_n : f \rightarrow h_n}{\texttt{\{}i_1 \texttt{ = } t_1\texttt{;} \ \ldots\texttt{;} \ i_n \texttt{ = } t_n\texttt{\}} : f \rightarrow \texttt{\{}i_1 \texttt{ : } h_1\texttt{;} \ \ldots\texttt{;} \ i_n \texttt{ : } h_n\texttt{\}}} \tag{362}$$

$$\frac{t_1 : f_1 \rightarrow h_1 \quad \cdots \quad t_n : f_n \rightarrow h_n}{\texttt{T\_ss}(c, lt) : \texttt{F\_ss}(c, lf) \rightarrow \texttt{F\_ss}(c, lh)} \tag{363}$$

$$\frac{}{\texttt{`}i : f \rightarrow \texttt{[`}i \ \ f\texttt{|} \ \ldots\texttt{]}} \tag{364}$$

$$\frac{t_1 : f_1 \rightarrow h \quad \cdots \quad t_n : f_n \rightarrow h}{\texttt{T\_case}(lt, h) : \texttt{[`}i_1 \ \ f_1\texttt{|} \ \ldots\texttt{|`}i_n \ \ f_n\texttt{]} \rightarrow h} \tag{365}$$

$$\frac{t_1 : \texttt{\{}\delta \;:\; g_1\texttt{;}\; \epsilon \;:\; f\texttt{\}} \to h \quad \cdots \quad t_n : \texttt{\{}\delta \;:\; g_n\texttt{;}\; \epsilon \;:\; f\texttt{\}} \to h}{\texttt{[`}i_1 \;\; t_1\texttt{|} \;\ldots\texttt{|`}i_n \;\; t_n\texttt{]} : \texttt{\{}\delta \;:\; \texttt{[`}i_1 \;\; g_1\texttt{|} \;\ldots\texttt{|`}i_n \;\; g_n\texttt{]}\texttt{;}\; \epsilon \;:\; f\texttt{\}} \to h} \tag{366}$$

$$\frac{t : \texttt{\{}\delta \;:\; f_\delta\texttt{;}\; \epsilon \;:\; f_\epsilon\texttt{\}} \to h}{\texttt{T\_map}(g, t) : \texttt{\{}\delta \;:\; g[f_\delta]\texttt{;}\; \epsilon \;:\; f_\epsilon\texttt{\}} \to g[h]} \tag{367}$$

$$\frac{t : f \to h}{\texttt{T\_ii}(t) : \texttt{F\_ii}(f) \to \texttt{F\_ii}(h)} \tag{368}$$

$$\frac{}{\texttt{T\_con}(g) : g[\texttt{F\_ii}(g)] \to \texttt{F\_ii}(g)} \tag{369}$$

$$\frac{t : g[h] \to h}{\texttt{T\_fold}(g, t) : \texttt{F\_ii}(g) \to h} \tag{370}$$

$$\frac{t : \texttt{\{}\delta \;:\; g[h]\texttt{;}\; \epsilon \;:\; f\texttt{\}} \to h}{\texttt{TL\_fold}(g, t) : \texttt{\{}\delta \;:\; \texttt{F\_ii}(g)\texttt{;}\; \epsilon \;:\; f\texttt{\}} \to h} \tag{371}$$

$$\frac{}{\texttt{T\_de}(g) : \texttt{F\_ii}(g) \to g[\texttt{F\_ii}(g)]} \tag{372}$$

$$\frac{t : f \to h}{\texttt{T\_tt}(t) : \texttt{F\_tt}(f) \to \texttt{F\_tt}(h)} \tag{373}$$

$$\frac{}{\texttt{T\_uncon}(g) : g[\texttt{F\_tt}(g)] \to \texttt{F\_tt}(g)} \tag{374}$$

$$\frac{t : h \to g[h]}{\texttt{T\_unfold}(g, t) : h \to \texttt{F\_tt}(g)} \tag{375}$$

$$\frac{t : \texttt{\{}\delta \;:\; h\texttt{;}\; \epsilon \;:\; f\texttt{\}} \to g[h]}{\texttt{TL\_unfold}(g, t) : \texttt{\{}\delta \;:\; h\texttt{;}\; \epsilon \;:\; f\texttt{\}} \to \texttt{F\_tt}(g)} \tag{376}$$

$$\frac{}{\texttt{T\_unde}(g) : \texttt{F\_tt}(g) \to g[\texttt{F\_tt}(g)]} \tag{377}$$

$$\frac{u : f \to h \qquad t_1 : f_1 \to h_1 \quad \cdots \quad t_n : f_n \to h_n}{\texttt{T\_ee}(lt, u) : \texttt{F\_ee}(lh, f) \to \texttt{F\_ee}(lf, h)} \tag{378}$$

$$\frac{}{\texttt{T\_appl}(lg, h) : \texttt{\{}v \texttt{ : } \texttt{F\_ee}(i_1 \texttt{ : } g_1 \texttt{; } \ldots, h)\texttt{; } i_1 \texttt{ : } g_1 \texttt{; } \ldots\texttt{\}} \to h} \tag{379}$$

$$\frac{t : \texttt{\{}v \texttt{ : } f\texttt{; } i_1 \texttt{ : } g_1\texttt{; } \ldots\texttt{; } i_n \texttt{ : } g_n\texttt{\}} \to h}{\texttt{T\_curry}(t) : f \to \texttt{F\_ee}(i_1 \texttt{ : } g_1\texttt{; } \ldots\texttt{; } i_n \texttt{ : } g_n,\, h)} \tag{380}$$

$$\frac{t : h \to h}{\texttt{T\_fix}(t) : \texttt{\{\}} \to h} \tag{381}$$

$$\frac{t : \texttt{\{}\delta \texttt{ : } h\texttt{; } \epsilon \texttt{ : } f\texttt{\}} \to h}{\texttt{TL\_fix}(t) : f \to h} \tag{382}$$

$$\frac{t : f \to g \qquad [\![f]\!]_i = [\![f']\!]_i \qquad [\![g]\!]_i = [\![g']\!]_i}{t : f' \to g'} \tag{383}$$

A.1.4 Reduction

The reduction system presented below is the system $\mathcal{R}_f$ of Section 8.2.2. Here we put together all the rules of the system introduced throughout our thesis. The rules that are potential obstacles to termination are placed at the end. There is a faithful implementation of this rewrite system in file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/core_back.ml`.

These are the reduction rules for `funct`-terms.

$$\begin{aligned}
f \texttt{ . } \texttt{F\_ID}(d) &\to f && (10)\\
\texttt{F\_ID}(c) \texttt{ . } g &\to g && (11)\\
f \texttt{ . } (g_1 \texttt{ . } g_2) &\to (f \texttt{ . } g_1) \texttt{ . } g_2 && (12)\\
\texttt{<}i \texttt{ : } f_i\texttt{; } \ldots\texttt{>} \texttt{ . } \texttt{F\_PR}(ld, i) &\to f_i && (13)\\
f \texttt{ . } \texttt{<}i \texttt{ : } g\texttt{; } \ldots\texttt{>} &\to \texttt{<}i \texttt{ : } f \texttt{ . } g\texttt{; } \ldots\texttt{>} && (14)
\end{aligned}$$

$$
\begin{array}{rcll}
f\ \texttt{.}\ \texttt{\{}i\ \texttt{:}\ g\texttt{;}\ \ldots\texttt{\}} & \to & \texttt{\{}i\ \texttt{:}\ f\ \texttt{.}\ g\texttt{;}\ \ldots\texttt{\}} & (54)\\
f\ \texttt{.}\ \texttt{[`}i\ \ g\texttt{|}\ \ldots\texttt{]} & \to & \texttt{[`}i\ \ f\ \texttt{.}\ g\texttt{|}\ \ldots\texttt{]} & (76)\\
f\ \texttt{.}\ \texttt{F\_ii}(g) & \to & \texttt{F\_ii(<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g) & (114)\\
f\ \texttt{.}\ \texttt{F\_tt}(g) & \to & \texttt{F\_tt(<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g) & (115)\\
f\ \texttt{.}\ \texttt{F\_ee}(i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots, h) & \to & \texttt{F\_ee}(i_1\ \texttt{:}\ f\ \texttt{.}\ g_1\texttt{;}\ \ldots, f\ \texttt{.}\ h) & (240)
\end{array}
$$

Having listed all the `funct`-rules in one place we can make some remarks about their similarities. Rules (14), (54) and (76) all look the same, because the product functor of rule (54) and the sum functor of rule (76) are in fact compositions of a record functor with an adjoint (see Section 7.1.3) so they rewrite similarly to the record functor of rule (14). Rules (114) and (115) look the same, because the inductive and coinductive functors are dual. In the rules for `trans`-terms the symmetry is often broken, however, to disambiguate the reduction order.

Below are the reduction rules for `trans`-terms. Rule (29)

$$t * u \ \to\ (f * u)\ \texttt{.}\ (t * h)$$

is omitted, being a direct consequence of the interchange law. In the presence of parameterized exponent, the rule would not be valid in some particular cases, as described in Section 8.1.4. Therefore, the use of horizontal composition and its rewriting is banned.

First, we list the rules that do not require restrictions on their use.

$$
\begin{array}{rcll}
f * \texttt{<}i\ \texttt{=}\ u\texttt{;}\ \ldots\texttt{>} & \to & \texttt{<}i\ \texttt{=}\ f * u\texttt{;}\ \ldots\texttt{>} & (30)\\
f * \texttt{T\_id}(g) & \to & \texttt{T\_id}(f\ \texttt{.}\ g) & (31)\\
f * (u_1\ \texttt{.}\ u_2) & \to & (f * u_1)\ \texttt{.}\ (f * u_2) & (32)\\
t * \texttt{F\_ID}(d) & \to & t & (33)\\
t * (f\ \texttt{.}\ g) & \to & (t * f) * g & (34)\\
\texttt{<}i\ \texttt{=}\ t_i\texttt{;}\ \ldots\texttt{>} * \texttt{F\_PR}(ld, i) & \to & t_i & (35)\\
\texttt{T\_id}(f) * \texttt{F\_PR}(ld, i) & \to & \texttt{T\_id}(f\ \texttt{.}\ \texttt{F\_PR}(ld, i)) & (36)\\
(t_1\ \texttt{.}\ t_2) * \texttt{F\_PR}(ld, i) & \to & (t_1 * \texttt{F\_PR}(ld, i))\ \texttt{.}\ (t_2 * \texttt{F\_PR}(ld, i)) & (37)\\
t * \texttt{<}i\ \texttt{:}\ g\texttt{;}\ \ldots\texttt{>} & \to & \texttt{<}i\ \texttt{=}\ t * g\texttt{;}\ \ldots\texttt{>} & (38)\\
\texttt{T\_ID}(c) & \to & \texttt{T\_id}(\texttt{F\_ID}(c)) & (39)\\
\texttt{T\_PR}(lc, i) & \to & \texttt{T\_id}(\texttt{F\_PR}(lc, i)) & (40)\\
t\ \texttt{.}\ \texttt{T\_id}(g) & \to & t & (41)\\
\texttt{T\_id}(f)\ \texttt{.}\ t & \to & t & (42)\\
t\ \texttt{.}\ (u_1\ \texttt{.}\ u_2) & \to & (t\ \texttt{.}\ u_1)\ \texttt{.}\ u_2 & (43)
\end{array}
$$

$$\mathtt{T\_pp}(c,\ i\ \mathtt{=}\ t\mathtt{;}\ \ldots) \ \rightarrow\ \mathtt{\{}i\ \mathtt{=}\ \mathtt{T\_pr}(lf, i)\ \mathtt{.}\ t\mathtt{;}\ \ldots\mathtt{\}} \tag{55}$$

$$\mathtt{\{}i\ \mathtt{=}\ t_i\mathtt{;}\ \ldots\mathtt{\}}\ \mathtt{.}\ \mathtt{T\_pr}(lg, i) \ \rightarrow\ t_i \tag{56}$$

$$t\ \mathtt{.}\ \mathtt{\{}i\ \mathtt{=}\ u\mathtt{;}\ \ldots\mathtt{\}} \ \rightarrow\ \mathtt{\{}i\ \mathtt{=}\ t\ \mathtt{.}\ u\mathtt{;}\ \ldots\mathtt{\}} \tag{57}$$

$$f * \mathtt{T\_pr}(i\ \mathtt{:}\ g\mathtt{;}\ \ldots, j) \ \rightarrow\ \mathtt{T\_pr}(i\ \mathtt{:}\ f\ \mathtt{.}\ g\mathtt{;}\ \ldots, j) \tag{58}$$

$$f * \mathtt{\{}i\ \mathtt{=}\ u\mathtt{;}\ \ldots\mathtt{\}} \ \rightarrow\ \mathtt{\{}i\ \mathtt{=}\ f * u\mathtt{;}\ \ldots\mathtt{\}} \tag{59}$$

$$t * \mathtt{\{}i\ \mathtt{:}\ g\mathtt{;}\ \ldots\mathtt{\}} \ \rightarrow\ \mathtt{T\_pp}(c,\ i\ \mathtt{=}\ t * g\mathtt{;}\ \ldots) \tag{60}$$

$$\mathtt{T\_ss}(c,\ i\ \mathtt{=}\ t\mathtt{;}\ \ldots) \ \rightarrow\ \mathtt{T\_case}(i\ \mathtt{=}\ t\ \mathtt{.}\ \mathtt{T\_in}(lg, i)\mathtt{;}\ \ldots, h) \tag{77}$$

$$\mathtt{T\_case}(i\ \mathtt{=}\ t\mathtt{;}\ \ldots, h) \ \rightarrow\ \mathtt{\{}\delta\ \mathtt{=}\ \mathtt{T\_id}(g)\mathtt{;}\ \epsilon\ \mathtt{=}\ \mathtt{\{\}\}}\ \mathtt{.}\ \mathtt{[`}i\ \ \delta\ \mathtt{.}\ t\mathtt{|}\ \ldots\mathtt{]} \tag{78}$$

$$\mathtt{\{}\delta\ \mathtt{=}\ \mathtt{`}i\mathtt{;}\ \epsilon\ \mathtt{=}\ t\mathtt{\}}\ \mathtt{.}\ \mathtt{[`}i\ \ u_i\mathtt{|}\ \ldots\mathtt{]} \ \rightarrow\ \mathtt{\{}\delta\ \mathtt{=}\ \mathtt{T\_id}(g_i)\mathtt{;}\ \epsilon\ \mathtt{=}\ t\mathtt{\}}\ \mathtt{.}\ u_i \tag{79}$$

$$\mathtt{\{}\delta\ \mathtt{=}\ t_1\ \mathtt{.}\ \mathtt{`}i\mathtt{;}\ \epsilon\ \mathtt{=}\ t\mathtt{\}}\ \mathtt{.}\ \mathtt{[`}i\ \ u_i\mathtt{|}\ \ldots\mathtt{]} \ \rightarrow\ \mathtt{\{}\delta\ \mathtt{=}\ t_1\mathtt{;}\ \epsilon\ \mathtt{=}\ t\mathtt{\}}\ \mathtt{.}\ u_i \tag{80}$$

$$f * \mathtt{T\_in}(i\ \mathtt{:}\ g\mathtt{;}\ \ldots, j) \ \rightarrow\ \mathtt{T\_in}(i\ \mathtt{:}\ f\ \mathtt{.}\ g\mathtt{;}\ \ldots, j\ ) \tag{81}$$

$$f * \mathtt{[`}i\ \ u\mathtt{|}\ \ldots\mathtt{]} \ \rightarrow\ \mathtt{[`}i\ \ f * u\mathtt{|}\ \ldots\mathtt{]} \tag{82}$$

$$t * \mathtt{[`}i\ \ g\mathtt{|}\ \ldots\mathtt{]} \ \rightarrow\ \mathtt{T\_ss}(c,\ i\ \mathtt{=}\ t * g\mathtt{;}\ \ldots) \tag{83}$$

$$\mathtt{T\_map}(\mathtt{F\_PR}(lc, \kappa), t) \ \rightarrow\ \mathtt{T\_pr}(\delta\ \mathtt{:}\ \mathtt{F\_ID(*)}\mathtt{;}\ \epsilon\ \mathtt{:}\ f_\epsilon,\ \delta) \tag{116}$$

$$\begin{gathered}\mathtt{T\_map}(\mathtt{F\_PR}(lc, \kappa)\ \mathtt{.}\ i_n\ \mathtt{.}\ \ldots\ \mathtt{.}\ i_2\ \mathtt{.}\ i_1, t) \ \rightarrow \\ \mathtt{T\_pr}(\delta\ \mathtt{:}\ (i_n\ \mathtt{.}\ \ldots\ \mathtt{.}\ i_2\ \mathtt{.}\ i_1)[f_\delta]\mathtt{;}\ \epsilon\ \mathtt{:}\ f_\epsilon,\ \delta)\end{gathered} \tag{117}$$

$$\mathtt{T\_map}(\mathtt{F\_PR}(lc, \iota), t) \ \rightarrow\ t \tag{118}$$

$$\mathtt{T\_map}(\mathtt{\{}lg\mathtt{\}}, t) \ \rightarrow\ \mathtt{\{}i\ \mathtt{=}\ \mathtt{\{}\delta\ \mathtt{=}\ \delta\ \mathtt{.}\ i\mathtt{;}\ \epsilon\ \mathtt{=}\ \epsilon\mathtt{\}}\ \mathtt{.}\ \mathtt{T\_map}(g_i, t)\mathtt{;}\ \ldots\mathtt{\}} \tag{119}$$

$$\mathtt{T\_map}(\mathtt{[}lg\mathtt{]}, t) \ \rightarrow\ \mathtt{[`}i\ \ \mathtt{T\_map}(g_i, t)\ \mathtt{.}\ \mathtt{T\_in}(lh, i)\mathtt{|}\ \ldots\mathtt{]} \tag{120}$$

$$\begin{aligned}\mathtt{T\_map}(\mathtt{F\_ii}(g), t) \ \rightarrow\ & \mathtt{TL\_fold}(g_f, (g_a * \mathtt{T\_map}(g_p, \kappa * t)) \\ & \mathtt{.}\ \mathtt{T\_con}(g_h))\end{aligned} \tag{121}$$

$$\begin{aligned}\mathtt{T\_map}(\mathtt{F\_tt}(g), t) \ \rightarrow\ & \mathtt{TL\_unfold}(g_h, \mathtt{\{}\delta\ \mathtt{=}\ \delta\ \mathtt{.}\ \mathtt{T\_unde}(g_f)\mathtt{;}\ \epsilon\ \mathtt{=}\ \epsilon\mathtt{\}} \\ & \mathtt{.}\ (g_c * \mathtt{T\_map}(g_p, \kappa * t)))\end{aligned} \tag{122}$$

$$\mathtt{T\_ii}(t) \ \rightarrow\ \mathtt{T\_fold}(f, t[\mathtt{F\_ii}(h)]\ \mathtt{.}\ \mathtt{T\_con}(h)) \tag{123}$$

$$\mathtt{T\_tt}(t) \ \rightarrow\ \mathtt{T\_unfold}(h, \mathtt{T\_unde}(f)\ \mathtt{.}\ t[\mathtt{F\_tt}(f)]) \tag{124}$$

$$\mathtt{T\_fold}(g, t) \ \rightarrow\ \mathtt{\{}\delta\ \mathtt{=}\ \mathtt{T\_id}(\mathtt{F\_ii}(g))\mathtt{;}\ \epsilon\ \mathtt{=}\ \mathtt{\{\}\}}\ \mathtt{.}\ \mathtt{TL\_fold}(g, \delta\ \mathtt{.}\ t) \tag{125}$$

$$\mathtt{T\_unfold}(g, t) \ \rightarrow\ \mathtt{\{}\delta\ \mathtt{=}\ \mathtt{T\_id}(h)\mathtt{;}\ \epsilon\ \mathtt{=}\ \mathtt{\{\}\}}\ \mathtt{.}\ \mathtt{TL\_unfold}(g, \delta\ \mathtt{.}\ t) \tag{126}$$

$$\begin{gathered}\mathtt{\{}\delta\ \mathtt{=}\ \mathtt{T\_con}(g)\mathtt{;}\ \epsilon\ \mathtt{=}\ t\mathtt{\}}\ \mathtt{.}\ \mathtt{TL\_fold}(g, u) \ \rightarrow \\ \mathtt{\{}\delta\ \mathtt{=}\ \mathtt{\{}\delta\ \mathtt{=}\ \mathtt{T\_id}(f)\mathtt{;}\ \epsilon\ \mathtt{=}\ t\mathtt{\}}\ \mathtt{.}\ \mathtt{T\_map}(g, \mathtt{TL\_fold}(g, u))\mathtt{;}\ \epsilon\ \mathtt{=}\ t\mathtt{\}}\ \mathtt{.}\ u\end{gathered} \tag{127}$$

$$\begin{gathered}\mathtt{\{}\delta\ \mathtt{=}\ t_1\ \mathtt{.}\ \mathtt{T\_con}(g)\mathtt{;}\ \epsilon\ \mathtt{=}\ t\mathtt{\}}\ \mathtt{.}\ \mathtt{TL\_fold}(g, u) \ \rightarrow \\ \mathtt{\{}\delta\ \mathtt{=}\ \mathtt{\{}\delta\ \mathtt{=}\ t_1\mathtt{;}\ \epsilon\ \mathtt{=}\ t\mathtt{\}}\ \mathtt{.}\ \mathtt{T\_map}(g, \mathtt{TL\_fold}(g, u))\mathtt{;}\ \epsilon\ \mathtt{=}\ t\mathtt{\}}\ \mathtt{.}\ u\end{gathered} \tag{128}$$

$$\begin{aligned}&\texttt{TL\_unfold}(g,t)\ \texttt{.}\ \texttt{T\_unde}(g)\ \rightarrow\\ &\texttt{\{}\delta\ \texttt{=}\ t\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}}\ \texttt{.}\ \texttt{T\_map}(g,\texttt{TL\_unfold}(g,t))\end{aligned}\tag{129}$$

$$\begin{aligned}&(t_1\ \texttt{.}\ \texttt{TL\_unfold}(g,t))\ \texttt{.}\ \texttt{T\_unde}(g)\ \rightarrow\\ &t_1\ \texttt{.}\ \texttt{\{}\delta\ \texttt{=}\ t\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}}\ \texttt{.}\ \texttt{T\_map}(g,\texttt{TL\_unfold}(g,t))\end{aligned}\tag{130}$$

$$\texttt{T\_con}(g)\ \texttt{.}\ \texttt{T\_de}(g)\ \rightarrow\ \texttt{T\_id}(g[\texttt{F\_ii}(g)])\tag{131}$$

$$(t_1\ \texttt{.}\ \texttt{T\_con}(g))\ \texttt{.}\ \texttt{T\_de}(g)\ \rightarrow\ t_1\tag{132}$$

$$\texttt{T\_uncon}(g)\ \texttt{.}\ \texttt{T\_unde}(g)\ \rightarrow\ \texttt{T\_id}(g[\texttt{F\_tt}(g)])\tag{133}$$

$$(t_1\ \texttt{.}\ \texttt{T\_uncon}(g))\ \texttt{.}\ \texttt{T\_unde}(g)\ \rightarrow\ t_1\tag{134}$$

$$f * \texttt{T\_con}(g)\ \rightarrow\ \texttt{T\_con(<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)\tag{135}$$

$$f * \texttt{TL\_fold}(g,t)\ \rightarrow\ \texttt{TL\_fold(<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g, f * t)\tag{136}$$

$$f * \texttt{T\_de}(g)\ \rightarrow\ \texttt{T\_de(<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)\tag{137}$$

$$f * \texttt{T\_uncon}(g)\ \rightarrow\ \texttt{T\_uncon(<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)\tag{138}$$

$$f * \texttt{TL\_unfold}(g,t)\ \rightarrow\ \texttt{TL\_unfold(<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g, f * t)\tag{139}$$

$$f * \texttt{T\_unde}(g)\ \rightarrow\ \texttt{T\_unde(<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)\tag{140}$$

$$t * \texttt{F\_ii}(g)\ \rightarrow\ \texttt{T\_ii(<}\iota\ \texttt{=}\ \iota\texttt{;}\ \kappa\ \texttt{=}\ \kappa * t\texttt{>} * g)\tag{141}$$

$$t * \texttt{F\_tt}(g)\ \rightarrow\ \texttt{T\_tt(<}\iota\ \texttt{=}\ \iota\texttt{;}\ \kappa\ \texttt{=}\ \kappa * t\texttt{>} * g)\tag{142}$$

$$u\ \texttt{.}\ \texttt{T\_curry}(t)\ \rightarrow\ \texttt{T\_curry(\{}\upsilon\ \texttt{=}\ \upsilon\ \texttt{.}\ u\texttt{;}\ i_1\ \texttt{=}\ i_1\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ t)\tag{239}$$

$$\texttt{T\_ee}(lt,u)\ \rightarrow\ \texttt{T\_curry(\{}\upsilon\ \texttt{=}\ \upsilon\texttt{;}\ i\ \texttt{=}\ i\ \texttt{.}\ t_i\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ \texttt{T\_appl}(lh',f)\ \texttt{.}\ u)\tag{241}$$

$$\begin{aligned}&\texttt{\{}\upsilon\ \texttt{=}\ \texttt{T\_curry}(t)\texttt{;}\ i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ \texttt{T\_appl}(lg,h)\\ &\rightarrow\ \texttt{\{}\upsilon\ \texttt{=}\ \texttt{T\_id}(f)\texttt{;}\ i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ t\end{aligned}\tag{242}$$

$$f * \texttt{T\_appl}(i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots, h)\ \rightarrow\ \texttt{T\_appl}(i_1\ \texttt{:}\ f\ \texttt{.}\ g_1\texttt{;}\ \ldots, f\ \texttt{.}\ h)\tag{243}$$

$$f * \texttt{T\_curry}(u)\ \rightarrow\ \texttt{T\_curry}(f * u)\tag{244}$$

We omit rule (245), too,

$$t * \texttt{F\_ee}(i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots, h)\ \rightarrow\ \texttt{T\_ee}(i_1\ \texttt{=}\ t * g_1\texttt{;}\ \ldots, t * h)$$

because the exponent functor as the right operand of multiplication is banned, to simplify typing.

The system up to here is confluent and strongly normalizing. Now we add the rules for the fixpoints. The first one should be heavily restricted to avoid spurious divergence. Anyway, regardless of the restrictions, now we lose even weak normalization, while still maintaining confluence.

$$\texttt{TL\_fix}(t)\ \rightarrow\ \texttt{\{}\delta\ \texttt{=}\ \texttt{TL\_fix}(t)\texttt{;}\ \epsilon\ \texttt{=}\ \texttt{T\_id}(f)\texttt{\}}\ \texttt{.}\ t\tag{250}$$

$$f * \texttt{TL\_fix}(u)\ \rightarrow\ \texttt{TL\_fix}(f * u)\tag{251}$$

$$\texttt{T\_fix}(t)\ \rightarrow\ \texttt{TL\_fix(T\_pr(}\delta\ \texttt{:}\ h\texttt{;}\ \epsilon\ \texttt{:}\ \texttt{\{\}}, \delta)\ \texttt{.}\ t)\tag{252}$$

$$t\ \texttt{.}\ \texttt{TL\_fix}(u)\ \rightarrow\ \texttt{TL\_fix(\{}\delta\ \texttt{=}\ \delta\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\ \texttt{.}\ t\texttt{\}}\ \texttt{.}\ u)\tag{253}$$

The application of the substitution rules below is restricted to the instances where the terms t are not of the form $\texttt{T\_id}(f)$. Without this restriction the reduction could diverge even in the absence of fixpoints.

$$\begin{array}{r}\{\delta = t_1;\ \epsilon = t\}\ .\ [\text{`}i\ \ u|\ \ldots]\ \rightarrow\\ \{\delta = t_1;\ \epsilon = \texttt{T\_id}(f)\}\ .\ [\text{`}i\ \ \{\delta = \delta;\ \epsilon = \epsilon\ .\ t\}\ .\ u|\ \ldots]\end{array} \tag{86}$$

$$\begin{array}{r}\{\delta = t_1;\ \epsilon = t\}\ .\ \texttt{TL\_fold}(g, u)\ \rightarrow\\ \{\delta = t_1;\ \epsilon = \texttt{T\_id}(f)\}\ .\ \texttt{TL\_fold}(g, \{\delta = \delta;\ \epsilon = \epsilon\ .\ t\}\ .\ u)\end{array} \tag{143}$$

$$\begin{array}{r}\{\delta = t_1;\ \epsilon = t\}\ .\ \texttt{TL\_unfold}(g, u)\ \rightarrow\\ \{\delta = t_1;\ \epsilon = \texttt{T\_id}(h)\}\ .\ \texttt{TL\_unfold}(g, \{\delta = \delta;\ \epsilon = \epsilon\ .\ t\}\ .\ u)\end{array} \tag{144}$$

The substitution rules are needed when we have eagerly linked modules. Without them, whole collection of compiled modules would be appended to a fold or case expressions, waiting for further reduction, for example for a variant to be chosen. With the substitution rule for sums, the modules are substituted at once into all the variants of the case expression, thus duplicating some code, but eliminating all the portions of the collection that are "dead code" with respect to the variants. Whether this is an efficient compilation strategy, or if it should be further restricted, should be extensively tested in practice using an implementation of this rule in the Dule compiler. Another option is hash-consing of transformations of the internal core language. Yet another is a restrictive generalized weak head normal form reduction strategy.

A.1.5 Equality testing

The equality testing procedure for the whole internal core language is a straightforward extension of the basic procedure of Section 6.1.3. Here is the source code in OCaml.

```
(* Library functions are explained in Section 3.2.
   Arguments [f] and [g] are in normal form
   and must have the same sources
   but in this version we check the targets, wherever needed. *)
let rec eq_funct f g =
  match f, g with
  | F_ID _, F_ID _ -> true
  | F_COMP (f1, f2), F_COMP (g1, g2) ->
      eq_funct f1 g1 && eq_funct f2 g2
  | F_PR (_, i), F_PR (_, j) when IdIndex.eq i j -> true
  | F_RECORD (_, lf), F_RECORD (_, lg) -> eqset eq_funct lf lg
  | _, F_RECORD (d, lg) ->
```

```
      let ef = trg f in
      let eg = trg g in
      eq_cat ef eg && bforall (eq_pi d [] f) lg
  | F_RECORD (c, lf), _ ->
      let ef = trg f in
      let eg = trg g in
      eq_cat ef eg && bforall (eq_pi c [] g) lf
(* to here we are in LC, next 5 cases are extensions *)
  | F_pp (_, lf), F_pp (_, lg) -> eqset eq_funct lf lg
  | F_ss (_, lf), F_ss (_, lg) -> eqset eq_funct lf lg
  | F_ii f, F_ii g ->
      let c1 = find AtIndex.atj (unPP (src f)) in
      let d1 = find AtIndex.atj (unPP (src g)) in
      eq_cat c1 d1 && eq_funct f g
  | F_tt f, F_tt g ->
      let c1 = find AtIndex.atj (unPP (src f)) in
      let d1 = find AtIndex.atj (unPP (src g)) in
      eq_cat c1 d1 && eq_funct f g
  | F_ee (lf, f), F_ee (lg, g) ->
      eq_funct f g && eqset eq_funct lf lg
  | _ ->
      let ef = trg f in
      let eg = trg g in
      eq_cat ef eg && is_trivial ef
(* [eq_pi] gives [true] if and only if
   [f] composed with projections at labels of [li]
   (in reverse order) and the projection at [i]
   (as the last one) is equal to [g] *)
and eq_pi d li f (i, g) =
  assert (EqFCat.eq d (src f) && EqFCat.eq d (src g));
  (* and [f] is not a record
     and [i] and [li] result in type-correct projections *)
  match g with
  | F_PR (_, j) when IdIndex.eq i j ->
      li = [] && eq_funct f (F_ID d)
  | F_COMP (f', F_PR (_, j)) when IdIndex.eq i j ->
      (match li with
      | [] -> eq_funct f f'
      | i1 :: r -> eq_pi d r f (i1, f'))
  | F_RECORD (_, lf) ->
      let ili = i :: li in
      bforall (eq_pi d ili f) lf
  | _ -> false
and is_trivial c =
  match c with
  | C_BB -> false
  | C_PP lc -> vforall is_trivial lc
```

The Dule compiler, when using the equality function defined as above, compiles programs correctly (see the module `EqFFunct` in file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/core_back.ml`). However, for efficiency, the equality is currently implemented taking advantage of a carefully guided η-contraction. Such implementation is especially efficient in the context of hash-consing with memoization, as seen in the source code in the same file. Then the equality test is just structural identity, where the indexed lists are treated just as dictionaries. In fact, with hash-consing, the equality test may even be performed using physical equality — by comparing addresses.

A.2 Internal module language

The internal module language has been developed in three stages, beginning in Section 5.2, continuing in Section 5.3, and concluding in Section 9.1 with the module system I-Dule. Here we provide the summary of I-Dule.

A.2.1 Syntax

In presenting the syntax and semantics we will adhere to the conventions proposed for the internal core language. In particular we will base our presentation on the abstract syntax, while providing the concrete syntax along the definition of the semantics. Our presentation of the abstract syntax is based on the definition contained in module `IDule` located in file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/mod_back.ml`.

The `cat`, `funct` and `trans`-terms referred to below come from the internal core language. Below we present the raw syntax of the two other syntactic domains of the internal module language, `sign` and `dule`. The raw `sign`-terms always belong to the language, if only their embedded raw `dule`-terms do. The conditions on the signatures of modules for the raw `dule`-terms to belong to the language are described alongside the overview of the semantics in the next section.

```
    type sign =
      | S_Pp of sign IList.t
      | S_Bb of sign * cat IList.t * funct IList.t
      | S_Ww of dule * sign
```

```
and dule =
  | M_Id of sign
  | M_Comp of dule * dule
  | M_Pr of sign IList.t * IdIndex.t
  | M_Record of sign * dule IList.t
  | M_Base of sign * sign * funct IList.t * trans IList.t
  | M_Inst of dule * dule
  | M_Trim of dule * sign
  | M_Accord of sign IList.t * dule IList.t
  | M_Concord of sign IList.t * dule IList.t
  | M_Link of sign IList.t * dule IList.t
  | M_Ind of sign IList.t * dule IList.t
  | M_CoInd of sign IList.t * dule IList.t
```

A.2.2 Semantics

Now we give an informal overview of the semantics of the signature and module operations. For a precise account, refer to the appropriate individual sections or to the OCaml definitions in the file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/mod_back.ml`. The whole semantics of the complete internal module language to (the SCM based on an extension of) *2-LC-lc* is denoted by $[\![\_]\!]_I$.

- `S_Pp`(ls)
  written "`{`i_1 `:` s_1`;` i_2 `:` s_2`;` $\ldots$ `;` i_n `:` s_n`}`"
  is a signature labeled product with name-driven sharing,

- `S_Bb`(r, la, lf)
  written "`sig` r `type` i $\ldots$ `value` j `:` f $\ldots$ `end`"
  or just "`sig type` i $\ldots$ `value` j `:` f $\ldots$ `end`"
  when the context signature r *can be recovered from the context*
  is the base signature specifying core language types and values of a base module with a domain signature r or richer,

- `S_Ww`(m_1, s_2)
  written "m_1 `|` s_2"
  is the signature s_2 specialized by a module m_1, as explained in detail elsewhere.

Each `dule`-term constructor is given with its domain and codomain `sign`-term as well as the concrete syntax. For a presentation of the signature assignment using typing rules, refer to the next section.

- `M_Id`$(s) : s \to s$
  written "`:` s"
  is the identity module — the identity morphism of the SCM,

- `M_Comp`$(m_1, m_2) : r_1 \to s_2$
  written "m_1 `.` m_2"
  is the composition (in the SCM) of morphisms $m_1 : r_1 \to s$ and $m_2 : s \to s_2$,

- `M_Pr`(lr, i) : `S_Pp`$(lr) \to r_i$
  written "i"
  is the module projection (with name-driven sharing) at label i, where the i-th element of lr (called r_i) is required to be present,

- `M_Record`$(r, lm) : r \to$ `S_Pp`(ls)
  written "`{`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}`"
  is the record (with name-driven sharing) of modules, where the elements of lm are $m_i : r \to s_i$,

- `M_Base`$(r, s, lg, lt) : r \to s$
  written "`::` r `->` s `struct type` i `=` g $\ldots$ `value` j `=` t $\ldots$ `end`"
  is the base module that uses elements available from an argument satisfying r to define core language types and values as specified in s,

- `M_Inst`$(m_1, m_2) : r_1 \to$ `S_Ww`(m_1, s_2)
  written "m_1 `|` m_2"
  is the instantiation of a module $m_2 : s \to s_2$ by a module $m_1 : r_1 \to s$,

- `M_Trim`$(m_1, r_2) : r_1 \to r_2$
  written "m_1 `:>` r_2"
  is the module $m_1 : r_1 \to s_1$ with the codomain changed to r_2 and some subcomponents removed, if necessary,

- `M_Accord`(lr, lm) : `S_Pp`$(lr) \to$ `S_Pp`(ls)
  written "`{`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}`"
  is the record of modules that are augmented by projections so that they have the common domain `S_Pp`(lr), where the elements of lm are m_i : `S_Pp`$(lr_i) \to s_i$, each lr_i is a sublist of lr (which may be arbitrarily large, not restricted to the sum of lr_i) and ls has the same labels as lm,

- `M_Concord`(lr, lm) : `S_Pp`$(lr) \to$ `S_Pp`$(lr$ `@@ diff` ls $lr)$
  written "`{{`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}}`"
  is the record of the projections with domain `S_Pp`(lr) and modules m_i : `S_Pp`$(lr_i) \to s_i$ augmented by projections so that they have the common domain `S_Pp`(lr), where if a label j appears both on lm and lr, then the s_j

has to be equal to r_j and the projection at j is not included in the record of modules, lr_i is a sublist of lr, lr may be arbitrarily large and ls has the same labels as lm,

- `M_Link`(lr, lm) : `S_Pp`$(lr) \to$ `S_Pp`(ls)
  written "`link {`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}`"
  is the record of modules obtained from lm and all the projections with domain `S_Pp`(lr) by composing each module with records of the others until the resulting domain becomes `S_Pp`(lr), where the elements of lm are m_i : `S_Pp`$(lr_i) \to s_i$, each lr_i is a sublist of lr `@@` ls, lm and lr are disjoint, lr may be arbitrarily large, ls has the same labels as lm and there no cyclic dependency among lm (which ensures termination),

- `M_Ind`(lr, lm) : `S_Pp`$(lr) \to$ `S_Pp`(ls)
  written "`ind {`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}`"
  is an inductive module that groups mutually dependent modules lm, where the elements of lm are m_i : `S_Pp`$(lr_i) \to s_i$, each lr_i is a sublist of lr `@@` ls, lr and ls are disjoint, lr may be arbitrarily large and ls has the same labels as lm,

- `M_CoInd`(lr, lm) : `S_Pp`$(lr) \to$ `S_Pp`(ls)
  written "`coind {`i_1 `=` m_1`;` i_2 `=` m_2`;` $\ldots$ `;` i_n `=` m_n`}`"
  is a coinductive module that groups mutually dependent modules lm, where the elements of lm are m_i : `S_Pp`$(lr_i) \to s_i$, each lr_i is a sublist of lr `@@` ls, lr and ls are disjoint, lr may be arbitrarily large and ls has the same labels as lm.

A.2.3 Typing rules

Here we present the typing of the internal module language operations in the form of typing rules. We do not formulate additional definedness side conditions, hence our rules do not completely capture definedness conditions of the module operations. They only indicate whether a raw term belongs to the language and what its domain and codomain are. The complete definedness conditions are given and discussed in detail in sections providing formal semantics of respective module operations. As always, we assume there are no repetitions on indexed lists of signatures.

Rule (387) describes the typing of the operation `M_Record`, while rule (391) describes the typing of `M_Accord`, which has the same concrete syntax. The presented typing rules for `M_Link`, `M_Ind` and `M_CoInd` are identical, although the additional definedness side conditions of the operations differ.

$$\frac{}{(:\ s) : s \to s} \tag{384}$$

$$\frac{m_1 : r_1 \rightarrow s \qquad m_2 : s \rightarrow s_2}{m_1 \;\texttt{.}\; m_2 : r_1 \rightarrow s_2} \tag{385}$$

$$\frac{}{i : \texttt{\{}i \;\texttt{:}\; r\texttt{;} \ldots\texttt{\}} \rightarrow r} \tag{386}$$

$$\frac{m_1 : r \rightarrow s_1 \quad \cdots \quad m_n : r \rightarrow s_n}{\texttt{\{}i_1 \texttt{ = } m_1\texttt{;} \ \ldots\texttt{;}\ i_n \texttt{ = } m_n\texttt{\}} : r \rightarrow \texttt{\{}i_1 \;\texttt{:}\; s_1\texttt{;} \ \ldots\texttt{;}\ i_n \;\texttt{:}\; s_n\texttt{\}}} \tag{387}$$

$$\frac{}{(\texttt{::}\ r \texttt{ -> } s \quad \texttt{struct type } i \texttt{ = } g \ \ldots \ \texttt{value } j \texttt{ = } t \ \ldots \ \texttt{end}) : r \rightarrow s} \tag{388}$$

$$\frac{m_1 : r_1 \rightarrow s \qquad m_2 : s \rightarrow s_2}{m_1 \;\texttt{|}\; m_2 : r_1 \rightarrow m_1 \;\texttt{|}\; s_2} \tag{389}$$

$$\frac{m_1 : r_1 \rightarrow s_1}{m_1 \;\texttt{:>}\; r_2 : r_1 \rightarrow r_2} \tag{390}$$

$$\frac{m_1 : \texttt{\{}j_1 \;\texttt{:}\; r_1\texttt{;} \ldots\texttt{\}} \rightarrow s_1 \quad \cdots \quad m_n : \texttt{\{}j_n \;\texttt{:}\; r_n\texttt{;} \ldots\texttt{\}} \rightarrow s_n}{\begin{array}{c}\texttt{\{}i_1 \texttt{ = } m_1\texttt{;} \ \ldots\texttt{;}\ i_n \texttt{ = } m_n\texttt{\}} : \\ \texttt{\{}j \;\texttt{:}\; r\texttt{;} \ \ldots\texttt{;}\ j_1 \;\texttt{:}\; r_1\texttt{;} \ \ldots\texttt{;}\ j_n \;\texttt{:}\; r_n\texttt{;} \ \ldots\texttt{\}} \rightarrow \\ \texttt{\{}i_1 \;\texttt{:}\; s_1\texttt{;} \ \ldots\texttt{;}\ i_n \;\texttt{:}\; s_n\texttt{\}}\end{array}} \tag{391}$$

$$\frac{m_1 : \texttt{\{}j_1 \;\texttt{:}\; r_1\texttt{;} \ldots\texttt{\}} \rightarrow s_1 \quad \cdots \quad m_n : \texttt{\{}j_n \;\texttt{:}\; r_n\texttt{;} \ldots\texttt{\}} \rightarrow s_n}{\begin{array}{c}\texttt{\{\{}i_1 \texttt{ = } m_1\texttt{;} \ \ldots\texttt{;}\ i_n \texttt{ = } m_n\texttt{\}\}} : \\ \texttt{\{}j \;\texttt{:}\; r\texttt{;} \ \ldots\texttt{;}\ j_1 \;\texttt{:}\; r_1\texttt{;} \ \ldots\texttt{;}\ j_n \;\texttt{:}\; r_n\texttt{;} \ \ldots\texttt{\}} \rightarrow \\ \texttt{\{}j \;\texttt{:}\; r\texttt{;} \ \ldots\texttt{;}\ j_1 \;\texttt{:}\; r_1\texttt{;} \ \ldots\texttt{;}\ j_n \;\texttt{:}\; r_n\texttt{;} \ \ldots\texttt{;}\ i_1 \;\texttt{:}\; s_1\texttt{;} \ \ldots\texttt{;}\ i_n \;\texttt{:}\; s_n\texttt{\}}\end{array}} \tag{392}$$

$$\frac{\begin{array}{c}m_1 : \texttt{\{}i_{k_1} \;\texttt{:}\; s_{k_1}\texttt{;} \ldots\texttt{;}\ j_1 \;\texttt{:}\; r_1\texttt{;} \ldots\texttt{\}} \rightarrow s_1 \\ \vdots \\ m_n : \texttt{\{}i_{k_n} \;\texttt{:}\; s_{k_n}\texttt{;} \ldots\texttt{;}\ j_n \;\texttt{:}\; r_n\texttt{;} \ldots\texttt{\}} \rightarrow s_n\end{array}}{\begin{array}{c}\texttt{link \{}i_1 \texttt{ = } m_1\texttt{;} \ \ldots\texttt{;}\ i_n \texttt{ = } m_n\texttt{\}} : \\ \texttt{\{}j_1 \;\texttt{:}\; r_1\texttt{;} \ \ldots\texttt{;}\ j_n \;\texttt{:}\; r_n\texttt{;} \ \ldots\texttt{\}} \rightarrow \texttt{\{}i_1 \;\texttt{:}\; s_1\texttt{;} \ \ldots\texttt{;}\ i_n \;\texttt{:}\; s_n\texttt{\}}\end{array}} \tag{393}$$

$$\frac{\begin{array}{c} m_1 : \{i_{k_1} : s_{k_1}; \ldots; j_1 : r_1; \ldots\} \to s_1 \\ \vdots \\ m_n : \{i_{k_n} : s_{k_n}; \ldots; j_n : r_n; \ldots\} \to s_n \end{array}}{\begin{array}{c} \texttt{ind}\ \{i_1 = m_1;\ \ldots;\ i_n = m_n\} : \\ \{j_1 : r_1;\ \ldots;\ j_n : r_n;\ \ldots\} \to \{i_1 : s_1;\ \ldots;\ i_n : s_n\} \end{array}} \tag{394}$$

$$\frac{\begin{array}{c} m_1 : \{i_{k_1} : s_{k_1}; \ldots; j_1 : r_1; \ldots\} \to s_1 \\ \vdots \\ m_n : \{i_{k_n} : s_{k_n}; \ldots; j_n : r_n; \ldots\} \to s_n \end{array}}{\begin{array}{c} \texttt{coind}\ \{i_1 = m_1;\ \ldots;\ i_n = m_n\} : \\ \{j_1 : r_1;\ \ldots;\ j_n : r_n;\ \ldots\} \to \{i_1 : s_1;\ \ldots;\ i_n : s_n\} \end{array}} \tag{395}$$

A.2.4 Simulation of multiplication

The combinator

```
val t_TF_coco : Trans.t * Trans.t ->  Funct.t -> Trans.t
```

is used in the definition of `m_XInd_ordinary` inductive module combinator in Section 9.1.2 on page 344.

The semantics of `t_TF_coco (tc, td) f` in a *2-LCX-F* with sums, (co)algebras and exponents is similar to the semantics of `t_TF tc f`, provided that `tc` is the inverse of `td` with respect to vertical composition. However, in the definition of `t_TF_coco` in terms of other e-Core operations, no multiplication by a functor from the right (`t_TF`) nor horizontal composition (`t_COMP`) is used, which greatly simplifies analysis and implementation of the inductive module constructors.

The discrepancy between `t_TF_coco (tc, td) f` and `t_TF tc f` appears only if `f` is constructed using the exponent functor (which is common, since modules contain not only constant values but also functions). As long as we multiply by covariant portions of `f` we can use `tc`, but in case of contravariance we swap to `td`. Due to the special treatment of exponent functor, the domain and codomain functors of the outcome transformation are just compositions of the domain and codomain of `tc` with `f`, respectively. In *2-LC-lc* the typing of multiplication is always that simple, but in the presence of fully parameterizable exponents the typing could be arbitrarily more complicated, hindering the construction of inductive module value parts.

The definition of the combinator is taken from file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/core_back.ml` from module `SemFTrans`. Notice the exchange of `tc` and `td` in the `F_ee` case and the failure, caused by functor `f` not in normal form with respect to $\mathcal{R}_f$, in the `F_COMP` case. No other cases are needed, since the algebra modeled by the compiler is reachable.

```
let rec t_TF_coco (tc, td) f =
(* for [m_XInd_ordinary]
   to approximately simulate [t_TF tc f]
   [f] in normal form *)
  let result = t_TF_coco' (tc, td) f in
  assert
    (let dtc = dom tc in
    let dtd = dom td in
    (* [tc] is the inverse of [td], so in particular: *)
    EqFFunct.eq dtc (cod td) &&
    EqFFunct.eq dtd (cod tc) &&
    (* we simulate [t_TF tc f], but with simpler typing: *)
    EqFFunct.eq (dom result) (f_COMP dtc f) &&
    EqFFunct.eq (cod result) (f_COMP dtd f)
    );
  result
and t_TF_coco' ((tc, td) as tctd) f =
  let d = src (dom tc) in
  match Funct.t2de f with
  |‘F_ID c -> tc
  |‘F_COMP (f1, f2) ->
      (match Funct.t2de f2 with
      |‘F_PR (lc, i) ->
          let ad = t_TF_coco tctd f1 in
          t_TF ad f2
      | _ -> failwith "SemFTrans.t_TF_coco:‘F_COMP ")
  |‘F_PR (lc, i) ->
      t_TF tc f
  |‘F_RECORD (c, lf) ->
      let lad = vmap (t_TF_coco tctd) lf in
      t_RECORD d lad
  |‘F_pp (c, lf) ->
      let lad = vmap (t_TF_coco tctd) lf in
      t_pp d lad
  |‘F_ss (c, lf) ->
      let lad = vmap (t_TF_coco tctd) lf in
      t_ss d lad
  |`F_ii f1 ->
      let b = find AtIndex.atj (unPP (src f1)) in
      let ci = coi b d in
      let pj = f_PR ci AtIndex.atk in
      let ptc = t_FT pj tc in
      let ptd = t_FT pj td in
      let idpi = t_id (f_PR ci AtIndex.atj) in
      let cp = c_PP ci in
      let ntc = t_RECORD cp (coi idpi ptc) in
      let ntd = t_RECORD cp (coi idpi ptd) in
      let ad = t_TF_coco (ntc, ntd) f1 in
      t_ii ad
  |`F_tt f1 ->
      let b = find AtIndex.atj (unPP (src f1)) in
      let ci = coi b d in
      let pj = f_PR ci AtIndex.atk in
      let ptc = t_FT pj tc in
      let ptd = t_FT pj td in
      let idpi = t_id (f_PR ci AtIndex.atj) in
      let cp = c_PP ci in
      let ntc = t_RECORD cp (coi idpi ptc) in
      let ntd = t_RECORD cp (coi idpi ptd) in
      let ad = t_TF_coco (ntc, ntd) f1 in
      t_tt ad
  |`F_ee (lf, f) ->
      let lad = vmap (t_TF_coco (td(*!!!*), tc(*!!!*))) lf in
      let ad = t_TF_coco tctd f in
      t_ee lad ad
```

A.2.5 Specification stripping

The operation `strip` is used in rule (292) of Section 9.2.1 to describe the simplified process of saturation of recursive specifications. The operation recursively traverses its argument — a component of a saturated specification — removing all types on the way.

The definition of the operation is taken, with simplifications, from file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/mod_front.ml`. It is given in terms of, approximately, the abstract syntax of the internal module language signatures, because components of saturated specifications have as simple form as internal language signatures.

```
let rec strip s =
  (match term_sign s with
  | S_Pp ls -> S_Pp (IList.vmap strip ls)
  | S_Bb (r, lc, lf) ->
      S_Bb (S_Pp IList.nil, lc, IList.nil)
  | S_Ww (m1, s2) -> strip s2
  | S_Mm (n, s) -> S_Mm (n ^ "@Stripped", strip s)
```

The S_Pp case captures the recursive traversal through product specification components. The S_Bb case shows how base specifications are stripped of their explicit parameters and types of their values, so that only names of types remain. The S_Ww case describes how every specialized specification "m | q" is replaced by a stripped down result of q. The S_Mm case represents isolated specifications and shows that stripping goes inside isolated specifications.

A.3 Syntax of the full user language

Here we summarize the syntax of the language accessible to the Dule programmer. This is the user core language and the user module language put together. The main nonterminal is START, which means that the top-level consists of specifications, libraries and modules that are then linked together at compile time.

The full user language is translated by the parser to the full bare language as described in Sections 8.3.2 and 9.2.4. Both the core bare language and the module bare language are typed as seen in the typing rules of Appendix A.1.3 and A.2.3, and have simple semantics into the internal core language. The internal core language, in turn, is just a language of a specific simple category. The study of the properties of this category has been one of the main topics of the thesis.

The grammar is presented using the notation introduced in Section 8.3.1. Only the concrete syntax is presented, as there is no abstract syntax for the user language. The details of the grammar, syntactic sugar and resolving ambiguities can be found in files describing the grammar for the universal parsing tools. The description of the LR parser for the OCaml Yacc [105] is in http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/parser.mly an the description of the LL parser for CamlP4 [37] is in http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/ll_parser.ml.

```
FIELD-TYPE  ::= VALUE-LABEL : TYPE
CASE-TYPE   ::= ‘ CASE-LABEL TYPE
              | ‘ CASE-LABEL
PARAM-TYPE  ::= ~ VALUE-LABEL : TYPE
TYPE        ::= TYPE . TYPE
              | TYPE-LABEL
              | DULE-LABEL
```

```
               | { FIELD-TYPE ;...; FIELD-TYPE }
               | [ CASE-TYPE "|"..."|" CASE-TYPE ]
               | ind TYPE-LABEL : TYPE
               | coind TYPE-LABEL : TYPE
               | PARAM-TYPE...PARAM-TYPE -> TYPE
               | ( TYPE )

FIELD       ::= VALUE-LABEL = VALUE
              | VALUE-LABEL
CASE        ::= ‘ CASE-LABEL EMBEDDING
EMBEDDING   ::= VALUE
              | PATTERN -> VALUE
              | -> VALUE
ARGUMENT    ::= ~ VALUE-LABEL : VALUE
              | ~ VALUE-LABEL
DECLARATION ::= PATTERN = VALUE
              | rec PATTERN = VALUE
PARAM       ::= ~ VALUE-LABEL
              | ~ VALUE-LABEL : PATTERN
              | ~ ( VALUE-LABEL : TYPE )
PATTERN     ::= VALUE-LABEL : TYPE
              | ( VALUE-LABEL : TYPE )
              | VALUE-LABEL
              | _ % underscore
VALUE       ::= : TYPE
              | VALUE . VALUE
              | VALUE-LABEL
              | DULE-LABEL
              | { FIELD ;...; FIELD }
              | VALUE . ‘ CASE-LABEL
              | ‘ CASE-LABEL
              | [ CASE "|"..."|" CASE ]
              | map EMBEDDING
              | con
              | fold EMBEDDING
              | de
              | uncon
              | unfold EMBEDDING
              | unde
              | VALUE ARGUMENT ARGUMENT...ARGUMENT
              | VALUE ~
              | match VALUE with VALUE
              | let DECLARATION...DECLARATION in VALUE
              | fun PARAM...PARAM -> VALUE
              | VALUE ( ARGUMENT ARGUMENT...ARGUMENT )
              | if VALUE then VALUE else VALUE
              | assert VALUE in VALUE
              | fail
              | ( VALUE )
```

```
FIELD-SP   ::= DULE-LABEL : SP
             | DULE-LABEL
PARAM-SP   ::= ~ DULE-LABEL : SP
             | ~ DULE-LABEL
BB-ITEM    ::= type TYPE-LABEL
             | value VALUE-LABEL : TYPE
SP         ::= { FIELD-SP ;...; FIELD-SP }
             | {{ FIELD-SP ;...; FIELD-SP }}
             | PARAM-SP...PARAM-SP -> SP
             | sig BB-ITEM...BB-ITEM end
             | SP with DULE
             | SP-LABEL
             | ( SP )

FIELD-DULE ::= DULE-LABEL = DULE
             | DULE-LABEL
BASE-ITEM  ::= type TYPE-LABEL = TYPE
             | value VALUE-LABEL = VALUE
             | value rec VALUE-LABEL = VALUE
DEF-DULE   ::= DULE-LABEL = DULE
DEF-SP     ::= SP-LABEL = SP
AND-DULE   ::= and DEF-DULE
ONE-DULE   ::= DEF-DULE AND-DULE...AND-DULE
AND-SP     ::= and DEF-SP
ONE-SP     ::= DEF-SP AND-SP...AND-SP
LINK-ITEM  ::= DEF-DULE
             | module DEF-DULE
             | module ind ONE-DULE
             | module coind ONE-DULE
             | spec DEF-SP
             | spec rec ONE-SP
             | library DEF-DULE
             | library ind ONE-DULE
             | library coind ONE-DULE
DULE       ::= DULE-LABEL
             | { FIELD-DULE ;...; FIELD-DULE }
             | {{ FIELD-DULE ;...; FIELD-DULE }}
             | :: SP DULE
             | struct BASE-ITEM...BASE-ITEM end
             | DULE "|" DULE
             | DULE with DULE
             | DULE :> SP
             | link LINK-ITEM...LINK-ITEM end
             | load DULE-LABEL
             | ( DULE )

START      ::= LINK-ITEM...LINK-ITEM
```

Appendix B

Longer proofs from Chapter 6 and Chapter 7

B.1 Proofs for Chapter 6

Lemma 6.1.20. *The following equation is a consequence of Φ_p.*

$$\texttt{T\_pp}(c,\ i\ \texttt{=}\ t\texttt{;}\ \ldots) \;=\; \texttt{\{}i\ \texttt{=}\ \texttt{T\_pr}(\mathit{lf}, i)\ \texttt{.}\ t\texttt{;}\ \ldots\texttt{\}} \tag{53}$$

Proof. This is a special case of Theorem 7.1.13 proved in Section 7.1.4. Here we present a much longer but direct proof.

First, we will derive the following equation from Φ_p.

$$\begin{aligned}\texttt{\{}i\ \texttt{=}\ t\texttt{;}\ \ldots\texttt{\}} \;=\; &\texttt{\{}i\ \texttt{=}\ \texttt{T\_id}(a)\texttt{;}\ \ldots\texttt{\}}\\ &\quad\texttt{.}\ (\texttt{T\_RECORD}(a,\ i\ \texttt{=}\ t\texttt{;}\ \ldots)\\ &\qquad *\ \texttt{F\_pp}(\texttt{<}lc\texttt{>},\ i\ \texttt{:}\ \texttt{F\_PR}(lc, i)\texttt{;}\ \ldots))\end{aligned} \tag{396}$$

We begin with the right hand side

$$\begin{aligned}&\texttt{\{}i\ \texttt{=}\ \texttt{T\_id}(a)\texttt{;}\ \ldots\texttt{\}}\\ &\quad\texttt{.}\ (\texttt{T\_RECORD}(a,\ i\ \texttt{=}\ t\texttt{;}\ \ldots)\\ &\qquad *\ \texttt{F\_pp}(\texttt{<}lc\texttt{>},\ i\ \texttt{:}\ \texttt{F\_PR}(lc, i)\texttt{;}\ \ldots))\end{aligned}$$

By the uniqueness requirement (49) we may transform it into

$$\begin{aligned}\texttt{\{}i\ \ \texttt{=}\ \ &\texttt{\{}i\ \texttt{=}\ \texttt{T\_id}(a)\texttt{;}\ \ldots\texttt{\}}\\ &\quad\texttt{.}\ (\texttt{T\_RECORD}(a,\ i\ \texttt{=}\ t\texttt{;}\ \ldots)\\ &\qquad *\ \texttt{F\_pp}(\texttt{<}lc\texttt{>},\ i\ \texttt{:}\ \texttt{F\_PR}(lc, i)\texttt{;}\ \ldots))\\ &\quad\texttt{.}\ \texttt{T\_pr}(\mathit{lf}, i)\texttt{;}\ \ldots\texttt{\}}\end{aligned}$$

and by interchange law (20) into

$$\begin{aligned}\texttt{\{}i \;\texttt{=}\; & \texttt{\{}i\,\texttt{=}\,\texttt{T\_id}(a)\texttt{;}\ \ldots\texttt{\}}\\ & \texttt{. T\_pr}(lf', i)\\ & \texttt{. (T\_RECORD}(a,\ i\,\texttt{=}\,t\texttt{;}\ \ldots)\\ & \quad *\ \texttt{F\_PR}(lc, i))\texttt{;}\ \ldots\texttt{\}}\end{aligned}$$

Then simplifying both records

$$\begin{aligned}\texttt{\{}i \;\texttt{=}\; & \texttt{T\_id}(a)\\ & \texttt{.}\ t\texttt{;}\ \ldots\texttt{\}}\end{aligned}$$

and concluding simplification

$$\texttt{\{}i\,\texttt{=}\,t\texttt{;}\ \ldots\texttt{\}}$$

we obtain the left hand side.

Now we begin the main argument. Let us start with the left hand side of equation (53).

$$\texttt{T\_pp}(c,\ i\,\texttt{=}\,t\texttt{;}\ \ldots)$$

Using equation (47) we may transform it into

$$\texttt{<}lt\texttt{>}\ *\ \texttt{F\_pp(<}lc\texttt{>},\ i\ \texttt{:}\ \texttt{F\_PR}(lc, i)\texttt{;}\ \ldots)$$

and by the uniqueness requirement (49) into

$$\texttt{\{}i\,\texttt{=}\,(\texttt{<}lt\texttt{>}\ *\ \texttt{F\_pp(<}lc\texttt{>},\ i\ \texttt{:}\ \texttt{F\_PR}(lc, i)\texttt{;}\ \ldots))\ \texttt{. T\_pr}(lf', i)\texttt{;}\ \ldots\texttt{\}}$$

Then we use the already proved equation (396)

$$\begin{aligned}&\texttt{\{}i\,\texttt{=}\,\texttt{T\_id}(a)\texttt{;}\ \ldots\texttt{\}}\\ &\ \texttt{. (<}i\,\texttt{=}\,(\texttt{<}lt\texttt{>}\ *\ \texttt{F\_pp(<}lc\texttt{>},\ i\ \texttt{:}\ \texttt{F\_PR}(lc, i)\texttt{;}\ \ldots))\ \texttt{. T\_pr}(lf', i)\texttt{;}\ \ldots\texttt{>}\\ &\qquad *\ \texttt{F\_pp(<}lc\texttt{>},\ i\ \texttt{:}\ \texttt{F\_PR}(lc, i)\texttt{;}\ \ldots))\end{aligned}$$

and split the big record over vertical composition (44)

$$\begin{aligned}&\texttt{\{}i\,\texttt{=}\,\texttt{T\_id}(a)\texttt{;}\ \ldots\texttt{\}}\\ &\ \texttt{. ((<}i\,\texttt{=}\,\texttt{<}lt\texttt{>}\ *\ \texttt{F\_pp(<}lc\texttt{>},\ i\ \texttt{:}\ \texttt{F\_PR}(lc, i)\texttt{;}\ \ldots)\texttt{;}\ \ldots\texttt{>}\\ &\qquad \texttt{. <}i\,\texttt{=}\,\texttt{T\_pr}(lf', i)\texttt{;}\ \ldots\texttt{>)}\\ &\quad *\ \texttt{F\_pp(<}lc\texttt{>},\ i\ \texttt{:}\ \texttt{F\_PR}(lc, i)\texttt{;}\ \ldots))\end{aligned}$$

Then we eliminate repeated record, where Δ = <i : F_ID(*); ...>

$$\begin{aligned}&\{i = \texttt{T\_id}(a);\ \dots\}\\ &\ .\ (((<lt> * \texttt{F\_pp}(<lc>,\ i : \texttt{F\_PR}(lc, i);\ \dots) * \Delta)\\ &\qquad .\ <i = \texttt{T\_pr}(lf', i);\ \dots>)\\ &\quad * \texttt{F\_pp}(<lc>,\ i : \texttt{F\_PR}(lc, i);\ \dots))\end{aligned}$$

and by associativity and complicating projections we get

$$\begin{aligned}&\{i = \texttt{T\_id}(a);\ \dots\}\\ &\ .\ (((<lt> * (\texttt{F\_pp}(<lc>,\ i : \texttt{F\_PR}(lc, i);\ \dots)\ .\ \Delta))\\ &\qquad .\ (<lf'> * <i = \texttt{T\_pr}(lh, i);\ \dots>))\\ &\quad * \texttt{F\_pp}(<lc>,\ i : \texttt{F\_PR}(lc, i);\ \dots))\end{aligned}$$

Then we use the interchange law (20)

$$\begin{aligned}&\{i = \texttt{T\_id}(a);\ \dots\}\\ &\ .\ (((<lf> * <i = \texttt{T\_pr}(lh, i);\ \dots>)\\ &\qquad .\ <lt>)\\ &\quad * \texttt{F\_pp}(<lc>,\ i : \texttt{F\_PR}(lc, i);\ \dots))\end{aligned}$$

and simplify projection

$$\begin{aligned}&\{i = \texttt{T\_id}(a);\ \dots\}\\ &\ .\ ((<i = \texttt{T\_pr}(lf, i);\ \dots>\\ &\qquad .\ <lt>)\\ &\quad * \texttt{F\_pp}(<lc>,\ i : \texttt{F\_PR}(lc, i);\ \dots))\end{aligned}$$

At last, we join records (44)

$$\begin{aligned}&\{i = \texttt{T\_id}(a);\ \dots\}\\ &\ .\ (\texttt{T\_RECORD}(a,\ i = \texttt{T\_pr}(lf, i)\ .\ t;\ \dots)\\ &\quad * \texttt{F\_pp}(<lc>,\ i : \texttt{F\_PR}(lc, i);\ \dots))\end{aligned}$$

again we use equation (396)

$$\{i = \texttt{T\_pr}(lf, i)\ .\ t;\ \dots\}$$

and thus we reach the right hand side. □

Lemma 6.2.2. *The theory Φ_c based on* T_case *and the theory Φ_s based on* TL_case *are equal.*

Proof. Proving that Φ_c is contained in Φ_s is comparatively easy, so we will only show the opposite inclusion.

Before we start the proof let us observe that the following equation holds.

$$p_s(lg, f) \quad = \quad \texttt{\{}\delta\ \texttt{=}\ \texttt{T\_case}(i\ \texttt{=}\ \delta\ \texttt{.}\ \text{`}i\texttt{;}\ \ldots, h)\texttt{;}\ \epsilon\ \texttt{=}\ \texttt{T\_case}(i\ \texttt{=}\ \epsilon\texttt{;}\ \ldots, h)\texttt{\}}$$

We begin the main argument by deriving equation (72) from theory Φ_c.

$$\begin{aligned}&\texttt{T\_case}(i\ \texttt{=}\ u_i\texttt{;}\ \ldots, h)\\ &=\ \texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id([}lg\texttt{]);}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ \texttt{[}\text{`}i\ \ \texttt{T\_pr}(lf^i, \delta)\ \texttt{.}\ u_i\texttt{|}\ \texttt{...]}\end{aligned}$$

We transform the right hand side using equation (71)

$$\texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id([}lg\texttt{]);}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ d_s(lg, f)\ \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ \delta\ \texttt{.}\ u_i\texttt{;}\ \ldots, h)$$

then using equation (67) we obtain

$$\begin{aligned}&\texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id([}lg\texttt{]);}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ d_s(lg, f)\\ &\ \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ \delta\ \texttt{.}\ \text{`}i\\ &\qquad\qquad\quad \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ u_i\texttt{;}\ \ldots, h)\texttt{;}\ \ldots, h)\end{aligned}$$

and again using equation (67) we get

$$\begin{aligned}&\texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id([}lg\texttt{]);}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ d_s(lg, f)\\ &\ \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ \text{`}i\ \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ \delta\ \texttt{.}\ \text{`}i\texttt{;}\ \ldots, h)\\ &\qquad\qquad\quad \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ u_i\texttt{;}\ \ldots, h)\texttt{;}\ \ldots, h)\end{aligned}$$

Then we rewrite by equation (68)

$$\begin{aligned}&\texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id([}lg\texttt{]);}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ d_s(lg, f)\\ &\ \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ \delta\ \texttt{.}\ \text{`}i\texttt{;}\ \ldots, h)\\ &\ \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ u_i\texttt{;}\ \ldots, h)\end{aligned}$$

and use our observation about $p_s(lg, f)$

$$\begin{aligned}&\texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id([}lg\texttt{]);}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ d_s(lg, f)\\ &\ \texttt{.}\ p_s(lg, f)\ \texttt{.}\ \delta\\ &\ \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ u_i\texttt{;}\ \ldots, h)\end{aligned}$$

Then by equation (70)

$$\begin{aligned}&\texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id([}lg\texttt{]);}\ \epsilon\ \texttt{=}\ t\texttt{\}}\ \texttt{.}\ \delta\\ &\ \texttt{.}\ \texttt{T\_case}(i\ \texttt{=}\ u_i\texttt{;}\ \ldots, h)\end{aligned}$$

and simplifying, we obtain the left hand side

$$\texttt{T\_case}(i \texttt{ = } u_i\texttt{;}\ \ldots, h)$$

Now we derive equation (73) from Φ_c.

$$\texttt{v}^i(lg, f)\ \texttt{.}\ \texttt{[`}i\ \ u_i\texttt{|}\ \ldots\texttt{]}\ =\ u_i$$

We transform the left had side using equation (67) and the original definition of $p_s(lg, f)$

$$\texttt{`}i\ \texttt{.}\ p_s(lg, f)\ \texttt{.}\ \texttt{[`}i\ \ u_i\texttt{|}\ \ldots\texttt{]}$$

and we use equation (71)

$$\texttt{`}i\ \texttt{.}\ p_s(lg, f)\ \texttt{.}\ d_s(lg, f)\ \texttt{.}\ \texttt{T\_case}(i \texttt{ = } u_i\texttt{;}\ \ldots, h)$$

Then we simplify by equation (69)

$$\texttt{`}i\ \texttt{.}\ \texttt{T\_case}(i \texttt{ = } u_i\texttt{;}\ \ldots, h)$$

and by equation (67) we have obtained the right hand side. Equation (74) is derived similarly. □

Lemma 6.3.3. *Let $t : f \to h$, $f, h : d \to \texttt{*}$ and let g be a ground* `funct`*-term such that $g : \texttt{<}\iota\ \texttt{-}\ \texttt{*;}\ \kappa\ \texttt{-}\ d\texttt{>} \to \texttt{*}$. Then the following equation follows from Φ_i.*

$$\texttt{T\_map}(g, \delta\ \texttt{.}\ t)\ =\ \delta\ \texttt{.}\ g[t] \tag{109}$$

Proof. The proof is an induction over the $\mathcal{R}_i$-normal form of g using the order on terms from the proof of Lemma 6.3.6. The cases where this differs from a standard structural induction are those for the (co)inductive types. There we need to assume the induction hypothesis for the functor g_p which is not strictly a subterm of g, but contains one less (co)inductive type constructor.

The typing of g implies that it cannot be an identity or a record. We analyze all the possible forms of g:

- $\texttt{F\_PR}(lc, \iota)$
  By equation (95) left hand side is $\delta\ \texttt{.}\ t$, which is equal to the right hand side.

- $\texttt{F\_PR}(lc, \kappa)$
  By equation (96) left hand side is δ, which is equal to the right hand side.

- $\texttt{F\_COMP}(f_1, f_2)$
  This case is analogous to the previous one, because the composition, being a normal form term, is a sequence of projections. The sequence is longer than one-element, so the target of the first projection has to be different than `*`, hence the first projection is at κ.

- $\texttt{F\_pp}(*, lg)$
  In this case the left hand side is

  $$\texttt{T\_map}(\texttt{\{}lg\texttt{\}}, \delta \texttt{ . } t)$$

  We can use equation (97), obtaining

  $$\texttt{\{}i \texttt{ = \{}\delta \texttt{ = } \delta \texttt{ . } i\texttt{; } \epsilon \texttt{ = } \epsilon\texttt{\} . T\_map}(g_i, \delta \texttt{ . } t)\texttt{; } \ldots\texttt{\}}$$

  and induction hypothesis, obtaining

  $$\texttt{\{}i \texttt{ = \{}\delta \texttt{ = } \delta \texttt{ . } i\texttt{; } \epsilon \texttt{ = } \epsilon\texttt{\} . } \delta \texttt{ . } g_i[t]\texttt{; } \ldots\texttt{\}}$$

  Then we simplify

  $$\texttt{\{}i \texttt{ = } \delta \texttt{ . } i \texttt{ . } g_i[t]\texttt{; } \ldots\texttt{\}}$$

  and transform

  $$\delta \texttt{ . \{}i \texttt{ = } i \texttt{ . } g_i[t]\texttt{; } \ldots\texttt{\}}$$

  By equation (53)

  $$\delta \texttt{ . T\_pp}(c, i \texttt{ = } g_i[t]\texttt{; } \ldots)$$

  using the definition of the horizontal composition shorthand

  $$\delta \texttt{ . T\_pp}(c, i \texttt{ = <}\iota \texttt{ = } t\texttt{; } \kappa \texttt{ = T\_ID}(c)\texttt{>} * g_i\texttt{; } \ldots)$$

  and by equation (60), we get

  $$\delta \texttt{ . (<}\iota \texttt{ = } t\texttt{; } \kappa \texttt{ = T\_ID}(c)\texttt{>} * \texttt{F\_pp}(*, lg))$$

  which is equal to the right hand side.

- $\texttt{F\_ss}(*, lg)$
  The right hand side is

  $$\delta \texttt{ . F\_ss}(*, lg)[t]$$

  We use the definition of the horizontal composition shorthand

  $$\delta \texttt{ . (<}\iota \texttt{ = } t\texttt{; } \kappa \texttt{ = T\_ID}(c)\texttt{>} * \texttt{F\_ss}(*, lg))$$

  and equation (83)

  $$\delta \texttt{ . T\_ss}(c, i \texttt{ = <}\iota \texttt{ = } t\texttt{; } \kappa \texttt{ = T\_ID}(c)\texttt{>} * g_i\texttt{; } \ldots)$$

Then we use the definition of the shorthand again

$$\delta \;\texttt{.}\; \texttt{T\_ss}(c, i \texttt{ = } g_i[t]\texttt{; } \ldots)$$

and equation (75)

$$\delta \;\texttt{.}\; \texttt{T\_case}(i \texttt{ = } g_i[t] \;\texttt{.}\; \texttt{`}i\texttt{; } \ldots, h')$$

and then equation (72)

$$\delta \;\texttt{.}\; \texttt{\{}\delta \texttt{ = T\_id(}f'\texttt{); } \epsilon \texttt{ = \{\}\}} \;\texttt{.}\; \texttt{[`}i \;\; \delta \;\texttt{.}\; g_i[t] \;\texttt{.}\; \texttt{`}i\texttt{| } \ldots\texttt{]}$$

Then we simplify

$$\texttt{\{}\delta \texttt{ = } \delta\texttt{; } \epsilon \texttt{ = \{\}\}} \;\texttt{.}\; \texttt{[`}i \;\; \delta \;\texttt{.}\; g_i[t] \;\texttt{.}\; \texttt{`}i\texttt{| } \ldots\texttt{]}$$

and use the property of terminal object

$$\texttt{\{}\delta \texttt{ = } \delta\texttt{; } \epsilon \texttt{ = } \epsilon\texttt{\}} \;\texttt{.}\; \texttt{[`}i \;\; \delta \;\texttt{.}\; g_i[t] \;\texttt{.}\; \texttt{`}i\texttt{| } \ldots\texttt{]}$$

Then we simplify again

$$\texttt{[`}i \;\; \delta \;\texttt{.}\; g_i[t] \;\texttt{.}\; \texttt{`}i\texttt{| } \ldots\texttt{]}$$

and after using the induction hypothesis

$$\texttt{[`}i \;\; \texttt{T\_map}(g_i, \delta \;\texttt{.}\; t) \;\texttt{.}\; \texttt{`}i\texttt{| } \ldots\texttt{]}$$

and equation (98)

$$\texttt{T\_map}(\texttt{[`}i \;\; g_i\texttt{| } \ldots\texttt{]}, \delta \;\texttt{.}\; t)$$

we arrive at the right hand side.

- $\texttt{F\_ii}(g')$
  The left hand side of the equation is

$$\texttt{T\_map}(\texttt{F\_ii}(g'), \delta \;\texttt{.}\; t)$$

By equation (99) this is equal to

$$\texttt{TL\_fold}(g_f, (g_a * \texttt{T\_map}(g_p, \kappa * (\delta \;\texttt{.}\; t))) \;\texttt{.}\; \texttt{T\_con}(g_h))$$

where

$$g_p = \texttt{<}\iota \texttt{ : } \kappa \;\texttt{.}\; \iota\texttt{; } \kappa \texttt{ : <}\iota \texttt{ : } \iota\texttt{; } \kappa \texttt{ : } \kappa \;\texttt{.}\; \kappa\texttt{>>} \;\texttt{.}\; g'$$

Changing the order of vertical and horizontal compositions

$$\texttt{TL\_fold}(g_f, (g_a * \texttt{T\_map}(g_p, \delta \;\texttt{.}\; (\kappa * t))) \;\texttt{.}\; \texttt{T\_con}(g_h))$$

by induction hypothesis we get

$$\texttt{TL\_fold}(g_f, (g_a * (\delta\ \texttt{.}\ g_p[\kappa * t]))\ \texttt{.}\ \texttt{T\_con}(g_h))$$

We write the term without the shorthand

$$\begin{array}{l}\texttt{TL\_fold}(g_f,\\ (g_a * (\delta\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ \kappa * t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(c)\texttt{>} * g_p)))\ \texttt{.}\ \texttt{T\_con}(g_h))\end{array}$$

and perform multiplication by g_a

$$\begin{array}{l}\texttt{TL\_fold}(g_f,\\ (\delta\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{<}\iota\ \texttt{:}\ \texttt{F\_ii}(g_h)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(c)\texttt{>>} * g_p))\ \texttt{.}\ \texttt{T\_con}(g_h))\end{array}$$

After another multiplication, this time by g_p, we obtain

$$\begin{array}{l}\texttt{TL\_fold}(g_f,\\ (\delta\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(\texttt{F\_ii}(g_h))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(c)\texttt{>>} * g'))\\ \ \texttt{.}\ \texttt{T\_con}(g_h))\end{array}$$

Now we transform the right hand side of the equation, which is

$$\delta\ \texttt{.}\ \texttt{F\_ii}(g')[t]$$

We write the term without the horizontal composition shorthand

$$\delta\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(c)\texttt{>} * \texttt{F\_ii}(g'))$$

We use equation (141)

$$\delta\ \texttt{.}\ \texttt{T\_ii}(\texttt{<}\iota\ \texttt{=}\ \iota\texttt{;}\ \kappa\ \texttt{=}\ \kappa * \texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(c)\texttt{>>} * g')$$

and equation (91)

$$\begin{array}{l}\delta\ \texttt{.}\ \texttt{T\_fold}(g_f,\\ (\texttt{<}\iota\ \texttt{=}\ \iota\texttt{;}\ \kappa\ \texttt{=}\ \kappa * \texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(c)\texttt{>>} * g')[\texttt{F\_ii}(g_h)]\ \texttt{.}\ \texttt{T\_con}(g_h))\end{array}$$

After writing without shorthands and simplifying

$$\begin{array}{l}\delta\ \texttt{.}\ \texttt{T\_fold}(g_f,\\ (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(\texttt{F\_ii}(g_h))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(c)\texttt{>>} * g')\ \texttt{.}\ \texttt{T\_con}(g_h))\end{array}$$

we transform the term by equation (125)

$$\begin{array}{l}\delta\ \texttt{.}\ \texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id}(\texttt{F\_ii}(g_f))\texttt{;}\ \epsilon\ \texttt{=}\ \texttt{\{\}\}}\\ \ \texttt{.}\ \texttt{TL\_fold}(g_f,\\ \delta\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(\texttt{F\_ii}(g_h))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(c)\texttt{>>} * g')\\ \ \texttt{.}\ \texttt{T\_con}(g_h))\end{array}$$

Using the property of terminal object

$$\begin{aligned}&\texttt{\{}\delta\ \texttt{=}\ \delta\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}}\\ &\ \texttt{. TL\_fold}(g_f,\\ &\delta\ \texttt{. (<}\iota\ \texttt{= T\_id(F\_ii(}g_h\texttt{));}\ \kappa\ \texttt{= <}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{= T\_ID(}c\texttt{)>>}\ *\ g')\\ &\ \texttt{. T\_con(}g_h\texttt{))}\end{aligned}$$

and simplifying

$$\begin{aligned}&\texttt{TL\_fold}(g_f,\\ &\delta\ \texttt{. (<}\iota\ \texttt{= T\_id(F\_ii(}g_h\texttt{));}\ \kappa\ \texttt{= <}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{= T\_ID(}c\texttt{)>>}\ *\ g')\\ &\ \texttt{. T\_con(}g_h\texttt{))}\end{aligned}$$

we have obtained the same term as was derived for the left hand side.

- $\texttt{F\_tt}(g')$
  The proof of this case is very similar to the proof of the previous one.

□

Corollary 6.3.4. *Let $t : f \to h$, $f, h : d \to *$ and let g be a ground* funct-*term such that $g : * \to *$. Then the following equation follows from Φ_i.*

$$t * g \quad = \quad \texttt{\{}\delta\ \texttt{= T\_id(}f\ \texttt{.}\ g\texttt{);}\ \epsilon\ \texttt{= \{\}\} . T\_map(}\iota\ \texttt{.}\ g, \delta\ \texttt{.}\ t\texttt{)} \qquad (110)$$

Proof. Transforming the right hand side, using equation (109) proved in Lemma 6.3.3, we obtain

$$\texttt{\{}\delta\ \texttt{= T\_id(}f\ \texttt{.}\ g\texttt{);}\ \epsilon\ \texttt{= \{\}\} .}\ \delta\ \texttt{. (}\iota\ \texttt{.}\ g\texttt{)}[t]$$

Writing without the horizontal composition shorthand

$$\texttt{\{}\delta\ \texttt{= T\_id(}f\ \texttt{.}\ g\texttt{);}\ \epsilon\ \texttt{= \{\}\} .}\ \delta\ \texttt{. (<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{= T\_ID(}c\texttt{)>}\ *\ \iota * g\texttt{)}$$

and simplifying the vertical composition, we get

$$\texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{= T\_ID(}c\texttt{)>}\ *\ \iota * g$$

Finally, simplifying the horizontal composition

$$t * g$$

we arrive at the left hand side. □

Lemma 6.3.9. *Ground instances of all rules of $\mathcal{R}_i$ except rules 127–128 are sound with respect to Φ_i.*

Proof. The cases of rules 114–124, (135) and (140) are trivial. We will start the proof by deriving rule (125)

$$\texttt{T\_fold}(g,t) \;\rightarrow\; \texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id(F\_ii(}g\texttt{));}\ \epsilon\ \texttt{=}\ \texttt{\{\}\}}\ \texttt{.}\ \texttt{TL\_fold}(g,\delta\ \texttt{.}\ t)$$

from Φ_i. We transform the right hand side using equation (107).

$$\texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id(F\_ii(}g\texttt{));}\ \epsilon\ \texttt{=}\ \texttt{\{\}\}}\ \texttt{.}\ d_i(g,\texttt{\{\}})\ \texttt{.}\ \texttt{T\_fold(\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \texttt{\{\}\}},\delta\ \texttt{.}\ t)$$

It is easy to verify that if we show the following equation

$$p_i(g,\texttt{\{\}})\ \texttt{.}\ \delta\ \texttt{.}\ \texttt{T\_fold}(g,t) \;=\; \texttt{T\_fold(\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \texttt{\{\}\}},\delta\ \texttt{.}\ t)$$

the rule will be derived. We prove the equation using the conditional equation (103). The condition in this case is

$$\begin{aligned}&\texttt{T\_con(\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \texttt{\{\}\})}\ \texttt{.}\ p_i(g,\texttt{\{\}})\ \texttt{.}\ \delta\ \texttt{.}\ \texttt{T\_fold}(g,t)\\ &=\ \texttt{\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \texttt{\{\}\}}[p_i(g,\texttt{\{\}})\ \texttt{.}\ \delta\ \texttt{.}\ \texttt{T\_fold}(g,t)]\ \texttt{.}\ \delta\ \texttt{.}\ t\end{aligned} \tag{397}$$

We transform the left hand side of the condition

$$\texttt{T\_con(\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \texttt{\{\}\})}\ \texttt{.}\ p_i(g,\texttt{\{\}})\ \texttt{.}\ \delta\ \texttt{.}\ \texttt{T\_fold}(g,t)$$

using the definition of $p_i(g,f)$

$$\begin{aligned}&\texttt{T\_con(\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \texttt{\{\}\})}\ \texttt{.}\ \texttt{T\_fold(\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \texttt{\{\}\}},\texttt{v}_i(g,\texttt{\{\}}))\\ &\ \texttt{.}\ \delta\ \texttt{.}\ \texttt{T\_fold}(g,t)\end{aligned}$$

and then using equation (101)

$$\begin{aligned}&\texttt{\{}\delta\ \texttt{:}\ g\texttt{;}\ \epsilon\ \texttt{:}\ \texttt{\{\}\}}[p_i(g,\texttt{\{\}})]\ \texttt{.}\ \texttt{v}_i(g,\texttt{\{\}})\\ &\ \texttt{.}\ \delta\ \texttt{.}\ \texttt{T\_fold}(g,t)\end{aligned}$$

Then we simplify

$$\texttt{\{}\delta\ \texttt{=}\ \delta\ \texttt{.}\ g[p_i(g,\texttt{\{\}})]\texttt{;}\ \epsilon\ \texttt{=}\ \texttt{\{\}\}}\ \texttt{.}\ \texttt{v}_i(g,\texttt{\{\}})\ \texttt{.}\ \delta\ \texttt{.}\ \texttt{T\_fold}(g,t)$$

use the definition of $\texttt{v}_i(g,\texttt{\{\}})$

$$\texttt{\{}\delta\ \texttt{=}\ \delta\ \texttt{.}\ g[p_i(g,\texttt{\{\}})]\ \texttt{.}\ g[\delta]\ \texttt{.}\ \texttt{T\_con}(g)\texttt{;}\ \epsilon\ \texttt{=}\ \texttt{\{\}\}}\ \texttt{.}\ \delta\ \texttt{.}\ \texttt{T\_fold}(g,t)$$

and simplify some more

$$\delta\ \texttt{.}\ g[p_i(g,\texttt{\{\}})]\ \texttt{.}\ g[\delta]\ \texttt{.}\ \texttt{T\_con}(g)\ \texttt{.}\ \texttt{T\_fold}(g,t)$$

Then we use equation (101) again

$$\delta \;.\; g[p_i(g,\texttt{\{\}})] \;.\; g[\delta] \;.\; g[\texttt{T\_fold}(g,t)] \;.\; t$$

and consolidate the resulting term

$$\delta \;.\; g[p_i(g,\texttt{\{\}}) \;.\; \delta \;.\; \texttt{T\_fold}(g,t)] \;.\; t$$

Then, complicating

$$\texttt{\{}\delta = \delta \;.\; g[p_i(g,\texttt{\{\}}) \;.\; \delta \;.\; \texttt{T\_fold}(g,t)]\texttt{;}\ \epsilon = \texttt{\{\}\}} \;.\; \delta \;.\; t$$

and writing as the multiplication by a product functor

$$\texttt{\{}\delta \,:\, g\texttt{;}\ \epsilon \,:\, \texttt{\{\}\}}[p_i(g,\texttt{\{\}}) \;.\; \delta \;.\; \texttt{T\_fold}(g,t)] \;.\; \delta \;.\; t$$

we arrive at the right hand side. Thus we have proved the condition, so we have shown equation (397), which ends the derivation of rule (125).

Next we derive rule (126)

$$\texttt{T\_unfold}(g,t) \;\rightarrow\; \texttt{\{}\delta = \texttt{T\_id}(h)\texttt{;}\ \epsilon = \texttt{\{\}\}} \;.\; \texttt{TL\_unfold}(g, \delta \;.\; t)$$

using the conditional equation (104) with the following condition.

$$\begin{aligned}&\texttt{\{}\delta = \texttt{T\_id}(h)\texttt{;}\ \epsilon = \texttt{\{\}\}} \;.\; \texttt{TL\_unfold}(g,\delta \;.\; t) \;.\; \texttt{T\_unde}(g)\\ =\;& t \;.\; g[\texttt{\{}\delta = \texttt{T\_id}(h)\texttt{;}\ \epsilon = \texttt{\{\}\}} \;.\; \texttt{TL\_unfold}(g,\delta \;.\; t)]\end{aligned}$$

We transform the left hand side using equation (108)

$$\begin{aligned}&\texttt{\{}\delta = \texttt{T\_id}(h)\texttt{;}\ \epsilon = \texttt{\{\}\}}\\ &\;.\; \texttt{T\_unfold}(g, \texttt{\{}\delta = \delta \;.\; t\texttt{;}\ \epsilon = \epsilon\texttt{\}} \;.\; d(g,h,\texttt{\{\}})) \;.\; \texttt{T\_unde}(g)\end{aligned}$$

and then simplify using equation (102)

$$\begin{aligned}&\texttt{\{}\delta = \texttt{T\_id}(h)\texttt{;}\ \epsilon = \texttt{\{\}\}}\\ &\;.\; \texttt{\{}\delta = \delta \;.\; t\texttt{;}\ \epsilon = \epsilon\texttt{\}} \;.\; d(g,h,\texttt{\{\}})\\ &\;.\; g[\texttt{T\_unfold}(g, \texttt{\{}\delta = \delta \;.\; t\texttt{;}\ \epsilon = \epsilon\texttt{\}} \;.\; d(g,h,\texttt{\{\}}))]\end{aligned}$$

We simplify further

$$\begin{aligned}&\texttt{\{}\delta = t\texttt{;}\ \epsilon = \texttt{\{\}\}} \;.\; d(g,h,\texttt{\{\}})\\ &\;.\; g[\texttt{TL\_unfold}(g,\delta \;.\; t)]\end{aligned}$$

and use the definition of $d(g,h,f)$

$$\begin{aligned}&\texttt{\{}\delta = t\texttt{;}\ \epsilon = \texttt{\{\}\}} \;.\; \texttt{T\_map}(g, \texttt{T\_id}(\texttt{\{}\delta \,:\, h\texttt{;}\ \epsilon \,:\, \texttt{\{\}\}}))\\ &\;.\; g[\texttt{TL\_unfold}(g,\delta \;.\; t)]\end{aligned}$$

We complicate the expression under T_map

$$\begin{aligned}&\texttt{\{}\delta\ \texttt{=}\ t\texttt{;}\ \epsilon\ \texttt{= \{\}\}}\ \texttt{.}\ \texttt{T\_map}(g, \delta\ \texttt{.}\ \texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id}(h)\texttt{;}\ \epsilon\ \texttt{= \{\}\}})\\ &\ \texttt{.}\ g[\texttt{TL\_unfold}(g, \delta\ \texttt{.}\ t)]\end{aligned}$$

and use equation (109) of Lemma 6.3.3

$$\begin{aligned}&\texttt{\{}\delta\ \texttt{=}\ t\texttt{;}\ \epsilon\ \texttt{= \{\}\}}\ \texttt{.}\ \delta\ \texttt{.}\ g[\texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id}(h)\texttt{;}\ \epsilon\ \texttt{= \{\}\}}]\\ &\ \texttt{.}\ g[\texttt{TL\_unfold}(g, \delta\ \texttt{.}\ t)]\end{aligned}$$

Again we simplify

$$\begin{aligned}&t\ \texttt{.}\ g[\texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id}(h)\texttt{;}\ \epsilon\ \texttt{= \{\}\}}]\\ &\ \texttt{.}\ g[\texttt{TL\_unfold}(g, \delta\ \texttt{.}\ t)]\end{aligned}$$

then consolidate the multiplications

$$t\ \texttt{.}\ g[\texttt{\{}\delta\ \texttt{=}\ \texttt{T\_id}(h)\texttt{;}\ \epsilon\ \texttt{= \{\}\}}\ \texttt{.}\ \texttt{TL\_unfold}(g, \delta\ \texttt{.}\ t)]$$

and we arrive at the right hand side of the condition thus completing the derivation of the rule.

Now we show how to derive rule (129)

$$\begin{aligned}\texttt{TL\_unfold}(g,t)\ \texttt{.}\ \texttt{T\_unde}(g)\ &\rightarrow\\ \texttt{\{}\delta\ \texttt{=}\ t\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}}\ \texttt{.}\ \texttt{T\_map}(g, \texttt{TL\_unfold}(g,t))&\end{aligned}$$

from Φ_i (the soundness of rule (130) follows immediately). We transform the left hand side of rule (129) using equation (108)

$$\texttt{T\_unfold}(g, \texttt{\{}\delta\ \texttt{=}\ t\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}}\ \texttt{.}\ d(g,h,f))\ \texttt{.}\ \texttt{T\_unde}(g)$$

and rewrite the resulting term according to equation (102)

$$\texttt{\{}\delta\ \texttt{=}\ t\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}}\ \texttt{.}\ d(g,h,f)\ \texttt{.}\ g[\texttt{TL\_unfold}(g,t)]$$

Then we use equation (113) from Observation 6.3.5

$$\texttt{\{}\delta\ \texttt{=}\ t\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}}\ \texttt{.}\ \texttt{T\_map}(g, \texttt{TL\_unfold}(g,t))$$

to obtain the right hand side.

Next we derive rule (131) (rules 132–134 are derived analogously).

$$\texttt{T\_con}(g)\ \texttt{.}\ \texttt{T\_de}(g)\ \rightarrow\ \texttt{T\_id}(g[\texttt{F\_ii}(g)])$$

First, we transform the left hand side using equation (93)

$$\texttt{T\_con}(g)\ \texttt{.}\ \texttt{T\_fold}(g, g[\texttt{T\_con}(g)])$$

then using equation (101)

$$g[\mathtt{T\_fold}(g, g[\mathtt{T\_con}(g)])] \mathbin{\mathtt{.}} g[\mathtt{T\_con}(g)]$$

and we consolidate multiplications obtaining

$$g[\mathtt{T\_fold}(g, g[\mathtt{T\_con}(g)]) \mathbin{\mathtt{.}} \mathtt{T\_con}(g)]$$

Then we turn to the right hand side of the rule

$$\mathtt{T\_id}(g[\mathtt{F\_ii}(g)])$$

and transform the identity into horizontal composition using, among others, equation (31)

$$g[\mathtt{T\_id}(\mathtt{F\_ii}(g))]$$

Using a trivial instance of the conditional equation (103) (where $u = \mathtt{T\_id}(\mathtt{F\_ii}(g))$) we finally transform the right hand side to

$$g[\mathtt{T\_fold}(g, \mathtt{T\_con}(g))]$$

The transformed left and right hand sides lead to the following equality.

$$g[\mathtt{T\_fold}(g, g[\mathtt{T\_con}(g)]) \mathbin{\mathtt{.}} \mathtt{T\_con}(g)] \quad = \quad g[\mathtt{T\_fold}(g, \mathtt{T\_con}(g))]$$

To derive rule (131) it now suffices to show that

$$\mathtt{T\_fold}(g, g[\mathtt{T\_con}(g)]) \mathbin{\mathtt{.}} \mathtt{T\_con}(g) \quad = \quad \mathtt{T\_fold}(g, \mathtt{T\_con}(g))$$

using conditional equation (103). The condition will be

$$\begin{aligned} &\mathtt{T\_con}(g) \mathbin{\mathtt{.}} \mathtt{T\_fold}(g, g[\mathtt{T\_con}(g)]) \mathbin{\mathtt{.}} \mathtt{T\_con}(g) \\ &= \; g[\mathtt{T\_fold}(g, g[\mathtt{T\_con}(g)]) \mathbin{\mathtt{.}} \mathtt{T\_con}(g)] \mathbin{\mathtt{.}} \mathtt{T\_con}(g) \end{aligned}$$

which is easily verified by transforming the left hand side using equation (101).

Now we will show the soundness of rule (136). Rule (137) is proved similarly. Before we start the main derivation let us prove the following equation where the non-distributive folding occupies the place of the distributive one in rule (136).

$$f * \mathtt{T\_fold}(g, t) \quad = \quad \mathtt{T\_fold}(\mathtt{<}\iota : \iota;\ \kappa : \kappa \mathbin{\mathtt{.}} f\mathtt{>} \mathbin{\mathtt{.}} g, f * t) \tag{398}$$

To derive equation (398) we use the conditional equation (103), with the condition

$$\begin{aligned} &\mathtt{T\_con}(\mathtt{<}\iota : \iota;\ \kappa : \kappa \mathbin{\mathtt{.}} f\mathtt{>} \mathbin{\mathtt{.}} g) \mathbin{\mathtt{.}} (f * \mathtt{T\_fold}(g, t)) \\ &= (\mathtt{<}\iota : \iota;\ \kappa : \kappa \mathbin{\mathtt{.}} f\mathtt{>} \mathbin{\mathtt{.}} g)[f * \mathtt{T\_fold}(g, t)] \mathbin{\mathtt{.}} (f * t) \end{aligned}$$

Using equation (89) we can transform the condition to

$$\begin{aligned}&(f * \texttt{T\_con}(g)) \texttt{ . } (f * \texttt{T\_fold}(g,t))\\ &= (f * (g[\texttt{T\_fold}(g,t)])) \texttt{ . } (f * t)\end{aligned}$$

Then by consolidating the multiplications we obtain

$$\begin{aligned}&f * (\texttt{T\_con}(g) \texttt{ . } \texttt{T\_fold}(g,t))\\ &= f * (g[\texttt{T\_fold}(g,t)] \texttt{ . } t)\end{aligned}$$

which follows from equation (101).

Rule (136) we are to derive is the following.

$$f * \texttt{TL\_fold}(g,t) \quad\rightarrow\quad \texttt{TL\_fold}(\texttt{<}\iota : \iota\texttt{;}\ \kappa : \kappa \texttt{ . } f\texttt{>} \texttt{ . } g, f * t)$$

By equation (107) the left hand side is equal to

$$f * (d_i(g,f') \texttt{ . } \texttt{T\_fold}(\texttt{\{}\delta : g\texttt{;}\ \epsilon : \kappa \texttt{ . } f'\texttt{\}}, t))$$

which can be rewritten to

$$(f * d_i(g,f')) \texttt{ . } (f * \texttt{T\_fold}(\texttt{\{}\delta : g\texttt{;}\ \epsilon : \kappa \texttt{ . } f'\texttt{\}}, t))$$

Since $d_i(g,f')$ is an isomorphism with inverse $p_i(g,f')$, the transformation $f * d_i(g,f')$ is and isomorphism with inverse $f * p_i(g,f')$. From equation (398) we easily see that

$$\begin{aligned}&f * p_i(g,f')\\ &= \ p_i(\texttt{<}\iota : \iota\texttt{;}\ \kappa : \kappa \texttt{ . } f\texttt{>} \texttt{ . } g, f \texttt{ . } f')\end{aligned} \tag{399}$$

Therefore $f * d_i(g,f')$ is the inverse of $p_i(\texttt{<}\iota : \iota\texttt{;}\ \kappa : \kappa \texttt{ . } f\texttt{>} \texttt{ . } g, f \texttt{ . } f')$. Isomorphisms have unique inverses, so we have

$$\begin{aligned}&f * d_i(g,f')\\ &= \ d_i(\texttt{<}\iota : \iota\texttt{;}\ \kappa : \kappa \texttt{ . } f\texttt{>} \texttt{ . } g, f \texttt{ . } f')\end{aligned} \tag{400}$$

Using equation (400) we can now rewrite the left hand side of rule (136) to the following form

$$d_i(\texttt{<}\iota : \iota\texttt{;}\ \kappa : \kappa \texttt{ . } f\texttt{>} \texttt{ . } g, f \texttt{ . } f') \texttt{ . } (f * \texttt{T\_fold}(\texttt{\{}\delta : g\texttt{;}\ \epsilon : \kappa \texttt{ . } f'\texttt{\}}, t))$$

By equation (398) the above term can be rewritten to

$$\begin{aligned}&d_i(\texttt{<}\iota : \iota\texttt{;}\ \kappa : \kappa \texttt{ . } f\texttt{>} \texttt{ . } g, f \texttt{ . } f')\\ &\texttt{ . } \texttt{T\_fold}(\texttt{\{}\delta : \texttt{<}\iota : \iota\texttt{;}\ \kappa : \kappa \texttt{ . } f\texttt{>} \texttt{ . } g\texttt{;}\ \epsilon : \kappa \texttt{ . } f \texttt{ . } f'\texttt{\}}, f * t)\end{aligned}$$

Finally, by (107) we arrive at

$$\texttt{TL\_fold}(\texttt{<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g, f * t)$$

which is the right hand side of the rule to be derived.

Next we derive rule (139) from Φ_i. Rule (138) can be derived similarly. The following equation

$$f * \texttt{T\_unfold}(g, t) \quad = \quad \texttt{T\_unfold}(\texttt{<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g, f * t) \qquad (401)$$

is derived analogously as equation (398). The left hand side of rule (139)

$$f * \texttt{TL\_unfold}(g, t) \quad \longrightarrow \quad \texttt{TL\_unfold}(\texttt{<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g, f * t)$$

can be rewritten by equation (108) to

$$f * (\texttt{T\_unfold}(g, \texttt{\{}\delta\ \texttt{=}\ t\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}})\ \texttt{.}\ d(g, h, f'))$$

which is equal to

$$(f * \texttt{T\_unfold}(g, \texttt{\{}\delta\ \texttt{=}\ t\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}}))\ \texttt{.}\ (f * d(g, h, f'))$$

Now we use equation (401) and equation (111) from Observation 6.3.5, obtaining

$$\begin{array}{l}\texttt{T\_unfold}(\texttt{<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g, \texttt{\{}\delta\ \texttt{=}\ f * t\texttt{;}\ \epsilon\ \texttt{=}\ \epsilon\texttt{\}})\\ \quad \texttt{.}\ d(\texttt{<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g, f\ \texttt{.}\ h, f\ \texttt{.}\ f')\end{array}$$

and using equation (108) again we reach the right hand side.

Finally, we will derive rule (141)

$$t * \texttt{F\_ii}(g) \quad \longrightarrow \quad \texttt{T\_ii}(\texttt{<}\iota\ \texttt{=}\ \iota\texttt{;}\ \kappa\ \texttt{=}\ \kappa * t\texttt{>} * g)$$

from the axioms of Φ_i. Rule (142) is derived analogously. We will call the left hand side of rule (141) u and the right hand side t'. By equation (91), t' is equal to

$$\texttt{T\_fold}(f', (\texttt{<}\iota\ \texttt{=}\ \iota\texttt{;}\ \kappa\ \texttt{=}\ \kappa * t\texttt{>} * g)[\texttt{F\_ii}(h')]\ \texttt{.}\ \texttt{T\_con}(h'))$$

where $t : f \longrightarrow h$, $f' = \texttt{<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g$ and $h' = \texttt{<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ h\texttt{>}\ \texttt{.}\ g$. The term can be written without the shorthand for horizontal composition as

$$\texttt{T\_fold}(f', (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id}(\texttt{F\_ii}(h'))\texttt{;}\ \kappa\ \texttt{=}\ t\texttt{>} * g)\ \texttt{.}\ \texttt{T\_con}(h'))$$

We will show that u is equal to t' using the following instance of the conditional equation (103):

$$\texttt{T\_con}(f')\ \texttt{.}\ u = f'[u]\ \texttt{.}\ t'' \quad \Rightarrow \quad u = \texttt{T\_fold}(f', t'')$$

where

$$t'' = (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id(F\_ii(}h'\texttt{));}\ \kappa\ \texttt{=}\ t\texttt{>} * g)\ \texttt{.}\ \texttt{T\_con(}h'\texttt{)}$$

To prove the condition we need to show that

$$\texttt{T\_con(}f'\texttt{)}\ \texttt{.}\ (t * \texttt{F\_ii(}g\texttt{)}) \quad = \quad f'[t * \texttt{F\_ii(}g\texttt{)}]\ \texttt{.}\ t''$$

The right hand side of this equation can be written as

$$\begin{aligned}&(\texttt{<}\iota\ \texttt{=}\ t * \texttt{F\_ii(}g\texttt{);}\ \kappa\ \texttt{=}\ \texttt{T\_ID(}c\texttt{)>}\\ &\ * \texttt{<}\iota\ \texttt{:}\ \iota\texttt{;}\ \kappa\ \texttt{:}\ \kappa\ \texttt{.}\ f\texttt{>}\ \texttt{.}\ g)\\ &\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id(F\_ii(}h'\texttt{));}\ \kappa\ \texttt{=}\ t\texttt{>} * g)\ \texttt{.}\ \texttt{T\_con(}h'\texttt{)}\end{aligned}$$

We simplify the expression

$$\begin{aligned}&(\texttt{<}\iota\ \texttt{=}\ t * \texttt{F\_ii(}g\texttt{);}\ \kappa\ \texttt{=}\ \texttt{T\_id(}f\texttt{)>} * g)\\ &\ \texttt{.}\ (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id(F\_ii(}h'\texttt{));}\ \kappa\ \texttt{=}\ t\texttt{>} * g)\ \texttt{.}\ \texttt{T\_con(}h'\texttt{)}\end{aligned}$$

and compose pairwise

$$(\texttt{<}\iota\ \texttt{=}\ t * \texttt{F\_ii(}g\texttt{);}\ \kappa\ \texttt{=}\ t\texttt{>} * g)\ \texttt{.}\ \texttt{T\_con(}h'\texttt{)}$$

Then we transform

$$(t * (\texttt{<}\iota\ \texttt{=}\ \texttt{T\_id(F\_ii(}g\texttt{));}\ \kappa\ \texttt{=}\ \texttt{T\_ID(d)>}\ \texttt{.}\ g))\ \texttt{.}\ (h * \texttt{T\_con(}g\texttt{)})$$

use the interchange law (20) and simplify

$$t * \texttt{T\_con(}g\texttt{)}$$

Then we complicate and use another instance of the interchange law

$$(f * \texttt{T\_con(}g\texttt{)})\ \texttt{.}\ (t * \texttt{F\_ii(}g\texttt{)})$$

Rewriting the term using equation (89)

$$\texttt{T\_con(}f'\texttt{)}\ \texttt{.}\ (t * \texttt{F\_ii(}g\texttt{)})$$

we arrive at the left hand side of the condition. By deriving the condition we prove the conclusion of the conditional equation, showing that the left hand side of our rule is equal to the right hand side in Φ_i. $\square$

B.2 Proofs for Chapter 7

Theorem 7.1.2. Φ_F *is contained in the theory of 2-LC-F.*

Proof. Semantically a *2-LC-F* is just a *2-LC-A* with two additional operations equationally defined (in the semantic definitions). Let us consider a theory generated by the theory Φ_A of *2-LC-A* translated to the language of *2-LC-F* and the two equations from the semantics, defining factorizers. By Observation 7.1.1 we know that this theory is sound for a *2-LC-F*. Therefore when we show that Φ_F follows from this theory, the thesis will be proven.

Equations (153) and (154) easily follow from the two defining equations for factorizers implied by the semantics. We will now only concentrate on equations (149) and (150), since equations (151) and (152) are derivable analogously.

To obtain equation (149), let us first multiply both sides of equation (145) by $h : a \to e$.

$$(h * \texttt{F\_ri}(g) * \eta_{\texttt{r}}(g)) \;.\; (h * \varepsilon_{\texttt{r}}(g) * \texttt{F\_ri}(g)) \;=\; \texttt{T\_id}(h \;.\; \texttt{F\_ri}(g))$$

Let $f : a \to c$ and $u : f \to h \;.\; \texttt{F\_ri}(g)$. Composing vertically u with both sides we get an equation

$$u \;.\; (h * \texttt{F\_ri}(g) * \eta_{\texttt{r}}(g)) \;.\; (h * \varepsilon_{\texttt{r}}(g) * \texttt{F\_ri}(g)) \;=\; u \;.\; \texttt{T\_id}(h \;.\; \texttt{F\_ri}(g))$$

Then simplifying right hand side and complicating left hand side we get

$$(u * \texttt{T\_ID}(c)) \;.\; (h * \texttt{F\_ri}(g) * \eta_{\texttt{r}}(g)) \;.\; (h * \varepsilon_{\texttt{r}}(g) * \texttt{F\_ri}(g)) \;=\; u$$

Now using the interchange law (20) on the first vertical composition at the left hand side we obtain

$$(f * \eta_{\texttt{r}}(g)) \;.\; (u * (g \;.\; \texttt{F\_ri}(g))) \;.\; (h * \varepsilon_{\texttt{r}}(g) * \texttt{F\_ri}(g)) \;=\; u$$

Then we shift the multiplication to the right

$$(f * \eta_{\texttt{r}}(g)) \;.\; (((u * g) \;.\; (h * \varepsilon_{\texttt{r}}(g))) * \texttt{F\_ri}(g)) \;=\; u$$

and write the left hand side as a factorizer, getting equation (149)

$$\texttt{T\_rfact}(g, f, (u * g) \;.\; (h * \varepsilon_{\texttt{r}}(g))) \;=\; u$$

Now let us derive equation (150). First, we multiply both sides of equation (146) by $f : a \to c$.

$$(f * \eta_{\texttt{r}}(g) * g) \;.\; (f * g * \varepsilon_{\texttt{r}}(g)) \;=\; \texttt{T\_id}(f \;.\; g)$$

Let $h : a \to e$ and $t : f \mathbin{.} g \to h$. Composing vertically t with both sides we get

$$(f * \eta_{\mathtt{r}}(g) * g) \mathbin{.} (f * g * \varepsilon_{\mathtt{r}}(g)) \mathbin{.} t \quad = \quad \mathtt{T\_id}(f \mathbin{.} g) \mathbin{.} t$$

Simplifying right hand side and complicating left hand side we get

$$(f * \eta_{\mathtt{r}}(g) * g) \mathbin{.} (f * g * \varepsilon_{\mathtt{r}}(g)) \mathbin{.} (t * \mathtt{T\_ID}(e)) \quad = \quad t$$

We use the interchange law (20) on the last vertical composition at the left hand side, obtaining

$$(f * \eta_{\mathtt{r}}(g) * g) \mathbin{.} (t * (\mathtt{F\_ri}(g) \mathbin{.} g)) \mathbin{.} (h * \varepsilon_{\mathtt{r}}(g)) \quad = \quad t$$

and we perform two more shifts of multiplications

$$(((f * \eta_{\mathtt{r}}(g)) \mathbin{.} (t * \mathtt{F\_ri}(g))) * g) \mathbin{.} (h * \varepsilon_{\mathtt{r}}(g)) \quad = \quad t$$

Writing a nested subexpression of the left hand side as a factorizer, we get equation (150)

$$(\mathtt{T\_rfact}(g, f, t) * g) \mathbin{.} (h * \varepsilon_{\mathtt{r}}(g)) \quad = \quad t$$

To be sure that we derived the same equations as those found in theory Φ_F we have to check one more thing. The equations in Φ_F have premises stating the definedness of their left hand and right hand sides. We have to make sure that the derived equations have premises equivalent (with respect to Φ_D) to the original ones.

During the derivation we do not need any new adjunctions, concentrating all the time on the right adjoint to g. The new adjunction constructors we use all pertain to the same single adjunction. Since definedness of a single adjunction operation implies definedness of all operations of this adjunction and since both the resulting equations involve the right adjoint to g, the premises are equivalent, which concludes the proof. □

Theorem 7.1.3. *Φ_F entails the theory of 2-LC-F.*

Proof. Reasoning analogously as in the proof of Theorem 7.1.2 we reduce the problem to deriving from the axioms of Φ_F equations 145–148 and the two equations implied by the semantic definition of factorizers.

First, we prove the equation defining the right factorizer.

$$\mathtt{T\_rfact}(g, f, t) \quad = \quad (f * \mathtt{T\_reta}(g)) \mathbin{.} (t * \mathtt{F\_ri}(g))$$

We start with the right hand side

$$(f * \mathtt{T\_reta}(g)) \mathbin{.} (t * \mathtt{F\_ri}(g))$$

and transform the term to another term with the same semantics, using equation (149)

$$\mathtt{T\_rfact}(g, f, (((f * \mathtt{T\_reta}(g)) \;.\; (t * \mathtt{F\_ri}(g))) * g) \;.\; (h * \varepsilon_{\mathtt{r}}(g)))$$

Then we multiply and change associativity, getting

$$\mathtt{T\_rfact}(g, f, ((f * \mathtt{T\_reta}(g)) * g) \;.\; ((t * (\mathtt{F\_ri}(g) \;.\; g)) \;.\; (h * \varepsilon_{\mathtt{r}}(g))))$$

Now we use the interchange law (20)

$$\mathtt{T\_rfact}(g, f, ((f * \mathtt{T\_reta}(g)) * g) \;.\; (((f \;.\; g) * \varepsilon_{\mathtt{r}}(g)) \;.\; (t * \mathtt{T\_ID}(e))))$$

simplify and change associativity again

$$\mathtt{T\_rfact}(g, f, (((f * \mathtt{T\_reta}(g)) * g) \;.\; (f * (g * \varepsilon_{\mathtt{r}}(g)))) \;.\; t)$$

Then we pull functor f outside the vertical composition

$$\mathtt{T\_rfact}(g, f, (f * ((\mathtt{T\_reta}(g) * g) \;.\; (g * \varepsilon_{\mathtt{r}}(g)))) \;.\; t)$$

and by equation (153) we eliminate `T_reta`

$$\mathtt{T\_rfact}(g, f, (f * ((\mathtt{T\_rfact}(g, \mathtt{F\_ID}(c), \mathtt{T\_id}(g)) * g) \;.\; (g * \varepsilon_{\mathtt{r}}(g)))) \;.\; t)$$

Finally, we use equation (150)

$$\mathtt{T\_rfact}(g, f, (f * \mathtt{T\_id}(g)) \;.\; t)$$

and after simplifying we get the left hand side

$$\mathtt{T\_rfact}(g, f, t)$$

The second of the definitional equations for factorizers follows in an analogous way. We obtain equations 145–148 by putting 2-identity for u or t in each of equations 149–152 and then using one of the just proven equations defining factorizers.

It is easy to verify that we didn't switch between adjunctions during these manipulations, so the premises of the derived equations are as required, which concludes the proof. □

Lemma 7.2.26. *Let t be an arbitrary 2-LCX transformation. Let u be a transformation with either X-covariant 2-LCX domain and codomain or X-contravariant 2-LCX domain and codomain. Let f be the 2-LCX domain of u. If f is X-covariant then let h be the codomain of t else if f is X-contravariant then let h be the domain of t. In both cases the transformations and functors satisfy the following 2-LCX equality.*

$$t * u \;=\; (t * f) \;.\; (h * u) \tag{187}$$

Proof. Let f be the *2-LCX* domain of u. We will prove the equation in case when f is X-contravariant. The other case can be proved similarly. Let h be the *2-LCX* domain of t. Let g be the *2-LCX* codomain of u.

We start with the *2-LCX* horizontal composition found at the left hand side of the equation and consider the similarly looking horizontal composition in the *2-LCO*.

$$[\![t]\!] * [\![u]\!]$$

To shorten, notation let the superscripting by ν or π mean composition of *2-LCO* functors or multiplication of *2-LCO* transformations with projections at the ν or π components, respectively:

$$\begin{aligned} f^\nu &= [\![f]\!] \;.\; \texttt{F\_PR}(lc, \nu) \\ f^\pi &= [\![f]\!] \;.\; \texttt{F\_PR}(lc, \pi) \\ t^\nu &= [\![t]\!] * \texttt{F\_PR}(lc, \nu) \\ t^\pi &= [\![t]\!] * \texttt{F\_PR}(lc, \pi) \end{aligned}$$

Because t and u belong to the *2-LCX* we may present their horizontal composition in *2-LCO* using projections and records.

$$\texttt{<}\nu = \texttt{<}\nu = t^\nu;\ \pi = t^\pi\texttt{>} * u^\nu;\ \pi = \texttt{<}\nu = t^\nu;\ \pi = t^\pi\texttt{>} * u^\pi\texttt{>}$$

Now we compose the transformations with identities.

$$\begin{aligned} &\texttt{<}\nu = \texttt{<}\nu = \texttt{T\_id}(h^\nu) \;.\; t^\nu;\ \pi = t^\pi \;.\; \texttt{T\_id}(h^\pi)\texttt{>} * (\texttt{T\_id}(f^\nu) \;.\; u^\nu); \\ &\quad \pi = \texttt{<}\nu = \texttt{T\_id}(h^\nu) \;.\; t^\nu;\ \pi = t^\pi \;.\; \texttt{T\_id}(h^\pi)\texttt{>} * (u^\pi \;.\; \texttt{T\_id}(f^\pi))\texttt{>} \end{aligned}$$

We split the vertical composition over the records.

$$\begin{aligned} &\texttt{<}\nu = \texttt{<}\nu = \texttt{T\_id}(h^\nu);\ \pi = t^\pi\texttt{>} \;.\; \texttt{<}\nu = t^\nu;\ \pi = \texttt{T\_id}(h^\pi)\texttt{>} * (\texttt{T\_id}(f^\nu) \;.\; u^\nu); \\ &\quad \pi = \texttt{<}\nu = \texttt{T\_id}(h^\nu);\ \pi = t^\pi\texttt{>} \;.\; \texttt{<}\nu = t^\nu;\ \pi = \texttt{T\_id}(h^\pi)\texttt{>} * (u^\pi \;.\; \texttt{T\_id}(f^\pi))\texttt{>} \end{aligned}$$

Then we use the interchange law (20) in the *2-LCO* twice.

$$\begin{aligned} &\texttt{<}\nu = (\texttt{<}\nu = \texttt{T\_id}(h^\nu);\ \pi = t^\pi\texttt{>} * \texttt{T\_id}(f^\nu)) \;.\; (\texttt{<}\nu = t^\nu;\ \pi = \texttt{T\_id}(h^\pi)\texttt{>} * u^\nu); \\ &\quad \pi = (\texttt{<}\nu = \texttt{T\_id}(h^\nu);\ \pi = t^\pi\texttt{>} * u^\pi) \;.\; (\texttt{<}\nu = t^\nu;\ \pi = \texttt{T\_id}(h^\pi)\texttt{>} * \texttt{T\_id}(f^\pi))\texttt{>} \end{aligned}$$

Because f is X-contravariant, f^ν is X-contravariant and f^π is X-covariant. Consequently, we may change the identities to anything else.

$$\begin{aligned} &\texttt{<}\nu = (\texttt{<}\nu = t^\nu;\ \pi = t^\pi\texttt{>} * \texttt{T\_id}(f^\nu)) \;.\; (\texttt{<}\nu = t^\nu;\ \pi = \texttt{T\_id}(h^\pi)\texttt{>} * u^\nu); \\ &\quad \pi = (\texttt{<}\nu = \texttt{T\_id}(h^\nu);\ \pi = t^\pi\texttt{>} * u^\pi) \;.\; (\texttt{<}\nu = t^\nu;\ \pi = t^\pi\texttt{>} * \texttt{T\_id}(f^\pi))\texttt{>} \end{aligned}$$

We simplify the presentation.

$$\begin{aligned}&\texttt{<}\nu\ \texttt{=}\ (t * f^{\nu})\ \texttt{.}\ (\texttt{<}\nu\ \texttt{=}\ t^{\nu}\texttt{;}\ \pi\ \texttt{=}\ \texttt{T\_id}(h^{\pi})\texttt{>} * u^{\nu})\texttt{;}\\ &\ \pi\ \texttt{=}\ (\texttt{<}\nu\ \texttt{=}\ \texttt{T\_id}(h^{\nu})\texttt{;}\ \pi\ \texttt{=}\ t^{\pi}\texttt{>} * u^{\pi})\ \texttt{.}\ (t * f^{\pi})\texttt{>}\end{aligned}$$

We compose vertically with identities, again.

$$\begin{aligned}&\texttt{<}\nu\ \texttt{=}\ (t * f^{\nu})\ \texttt{.}\ (\texttt{T\_id}([\![h]\!])\ \texttt{.}\ \texttt{<}\nu\ \texttt{=}\ t^{\nu}\texttt{;}\ \pi\ \texttt{=}\ \texttt{T\_id}(h^{\pi})\texttt{>}) * (u^{\nu}\ \texttt{.}\ \texttt{T\_id}(g^{\nu}))\texttt{;}\\ &\ \pi\ \texttt{=}\ (\texttt{<}\nu\ \texttt{=}\ \texttt{T\_id}(h^{\nu})\texttt{;}\ \pi\ \texttt{=}\ t^{\pi}\texttt{>}\ \texttt{.}\ \texttt{T\_id}([\![h]\!])) * (\texttt{T\_id}(g^{\pi})\ \texttt{.}\ u^{\pi})\ \texttt{.}\ (t * f^{\pi})\texttt{>}\end{aligned}$$

Again we use the interchange law (20) in the *2-LCO* twice (at once simplifying notation).

$$\begin{aligned}&\texttt{<}\nu\ \texttt{=}\ ([\![t]\!] * f^{\nu})\ \texttt{.}\ ([\![h]\!] * u^{\nu})\ \texttt{.}\ (\texttt{<}\nu\ \texttt{=}\ t^{\nu}\texttt{;}\ \pi\ \texttt{=}\ \texttt{T\_id}(h^{\pi})\texttt{>} * g^{\nu})\texttt{;}\\ &\ \pi\ \texttt{=}\ (\texttt{<}\nu\ \texttt{=}\ \texttt{T\_id}(h^{\nu})\texttt{;}\ \pi\ \texttt{=}\ t^{\pi}\texttt{>} * g^{\pi})\ \texttt{.}\ ([\![h]\!] * u^{\pi})\ \texttt{.}\ ([\![t]\!] * f^{\pi})\texttt{>}\end{aligned}$$

Because g is X-contravariant, g^{ν} is X-contravariant and g^{π} is X-covariant. Consequently, we may change the projections of $[\![t]\!]$ to anything we choose.

$$\begin{aligned}&\texttt{<}\nu\ \texttt{=}\ ([\![t]\!] * f^{\nu})\ \texttt{.}\ ([\![h]\!] * u^{\nu})\ \texttt{.}\ (\texttt{<}\nu\ \texttt{=}\ \texttt{T\_id}(h^{\nu})\texttt{;}\ \pi\ \texttt{=}\ \texttt{T\_id}(h^{\pi})\texttt{>} * g^{\nu})\texttt{;}\\ &\ \pi\ \texttt{=}\ (\texttt{<}\nu\ \texttt{=}\ \texttt{T\_id}(h^{\nu})\texttt{;}\ \pi\ \texttt{=}\ \texttt{T\_id}(h^{\pi})\texttt{>} * g^{\pi})\ \texttt{.}\ ([\![h]\!] * u^{\pi})\ \texttt{.}\ ([\![t]\!] * f^{\pi})\texttt{>}\end{aligned}$$

This enables us to simplify terms.

$$\begin{aligned}&\texttt{<}\nu\ \texttt{=}\ ([\![t]\!] * f^{\nu})\ \texttt{.}\ ([\![h]\!] * u^{\nu})\ \texttt{.}\ \texttt{T\_id}(h^{\nu}\ \texttt{.}\ g^{\nu})\texttt{;}\\ &\ \pi\ \texttt{=}\ \texttt{T\_id}(h^{\nu}\ \texttt{.}\ g^{\pi})\ \texttt{.}\ ([\![h]\!] * u^{\pi})\ \texttt{.}\ ([\![t]\!] * f^{\pi})\texttt{>}\end{aligned}$$

By eliminating identities we obtain

$$\begin{aligned}&\texttt{<}\nu\ \texttt{=}\ ([\![t]\!] * f^{\nu})\ \texttt{.}\ ([\![h]\!] * u^{\nu})\texttt{;}\\ &\ \pi\ \texttt{=}\ ([\![h]\!] * u^{\pi})\ \texttt{.}\ ([\![t]\!] * f^{\pi})\texttt{>}\end{aligned}$$

which is, by definition of the translation, equal to the *2-LCX* vertical composition of the following *2-LCO* transformations

$$[\![t]\!] * [\![f]\!] \quad \text{and} \quad [\![h]\!] * [\![u]\!]$$

which shows that the equation holds with the choice of h and f as in thesis. □

Theorem 7.3.5. *Φ^X_F is contained in the theory of 2-LCX-F.*

Proof. Reasoning analogously as in the proof of Theorem 7.1.2 for *2-LC-F*, we reduce the problem to showing that the equations listed in Section 7.1.2 follow from the defining equations for factorizers and Φ^X_A.

The proofs that particular equations of the list 207–226 follow from the theory are either easy or similar to the proofs for the *2-LC-F* case. Here we will only derive equation (207).

Let $h : a \to e$. We multiply both sides of equation (191) by functor $\texttt{<}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{>}$.

$$\begin{aligned}&(\texttt{<}\iota : \texttt{<}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{> . F\_ri}(g)\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \eta_{\texttt{r}}(g))\\ &\ \texttt{.}\ (\texttt{<}\iota \texttt{ = <}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(g)\texttt{;}\ \kappa \texttt{ = T\_ID}(a)\texttt{>} * \texttt{F\_ri}(g))\\ &= \texttt{T\_id}(\texttt{<}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{> . F\_ri}(g))\end{aligned}$$

Let $f : a \to c$ and $u : f \to \texttt{<}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{> . F\_ri}(g)$. Composing vertically u with both sides we get:

$$\begin{aligned}&u\ \texttt{.}\ (\texttt{<}\iota : \texttt{<}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{> . F\_ri}(g)\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \eta_{\texttt{r}}(g))\\ &\ \texttt{.}\ (\texttt{<}\iota \texttt{ = <}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(g)\texttt{;}\ \kappa \texttt{ = T\_ID}(a)\texttt{>} * \texttt{F\_ri}(g))\\ &= u\ \texttt{.}\ \texttt{T\_id}(\texttt{<}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{> . F\_ri}(g))\end{aligned}$$

Then we simplify the right hand side and complicate the left hand side.

$$\begin{aligned}&(\texttt{<}\iota \texttt{ = } u\texttt{;}\ \kappa \texttt{ = T\_ID}(a)\texttt{>} * \texttt{F\_PR}(le, \iota))\\ &\ \texttt{.}\ (\texttt{<}\iota : \texttt{<}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{> . F\_ri}(g)\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \eta_{\texttt{r}}(g))\\ &\ \texttt{.}\ (\texttt{<}\iota \texttt{ = <}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(g)\texttt{;}\ \kappa \texttt{ = T\_ID}(a)\texttt{>} * \texttt{F\_ri}(g))\\ &= u\end{aligned}$$

Since $\eta_{\texttt{r}}(g)$ is ι-covariant and the κ component of the terms at the left hand sides of multiplications is always an identity, we may now use a variant of equation (188) on the first vertical composition at the left hand side of the equation, getting

$$\begin{aligned}&(\texttt{<}\iota : f\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \eta_{\texttt{r}}(g))\\ &\ \texttt{.}\ (\texttt{<}\iota \texttt{ = } u\texttt{;}\ \kappa \texttt{ = T\_ID}(a)\texttt{>} * \texttt{<}\iota : g\texttt{;}\ \kappa : \texttt{F\_PR}(lc, \kappa)\texttt{>} * \texttt{F\_ri}(g))\\ &\ \texttt{.}\ (\texttt{<}\iota \texttt{ = <}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(g)\texttt{;}\ \kappa \texttt{ = T\_ID}(a)\texttt{>} * \texttt{F\_ri}(g))\\ &= u\end{aligned}$$

We then simplify the left hand side

$$\begin{aligned}&(\texttt{<}\iota : f\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \eta_{\texttt{r}}(g))\\ &\ \texttt{.}\ (\texttt{<}\iota \texttt{ = <}\iota \texttt{ = } u\texttt{;}\ \kappa \texttt{ = T\_ID}(a)\texttt{>} * g\texttt{;}\ \kappa \texttt{ = T\_ID}(a)\texttt{>} * \texttt{F\_ri}(g))\\ &\ \texttt{.}\ (\texttt{<}\iota \texttt{ = <}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(g)\texttt{;}\ \kappa \texttt{ = T\_ID}(a)\texttt{>} * \texttt{F\_ri}(g))\\ &= u\end{aligned}$$

and perform one more shift of multiplications

$$\begin{aligned}&(\texttt{<}\iota : f;\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \eta_{\texttt{r}}(g))\\ &\ .\ (\texttt{<}\iota = \texttt{<}\iota = u;\ \kappa = \texttt{T\_ID}(a)\texttt{>} * g;\ \kappa = \texttt{T\_ID}(a)\texttt{>}\\ &\ .\ \texttt{<}\iota = \texttt{<}\iota : h;\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(g);\ \kappa = \texttt{T\_ID}(a)\texttt{>}) * \texttt{F\_ri}(g)\\ &= u\end{aligned}$$

Then we accumulate the record

$$\begin{aligned}&(\texttt{<}\iota : f;\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \eta_{\texttt{r}}(g))\\ &\ .\ (\texttt{<}\iota = (\texttt{<}\iota = u;\ \kappa = \texttt{T\_ID}(a)\texttt{>} * g)\ .\\ &(\texttt{<}\iota : h;\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(g));\ \kappa = \texttt{T\_ID}(a)\texttt{>} * \texttt{F\_ri}(g))\\ &= u\end{aligned}$$

and writing the left hand side as a factorizer

$$\begin{aligned}&\texttt{T\_rfact}(g, f, (\texttt{<}\iota = u;\ \kappa = \texttt{T\_ID}(a)\texttt{>} * g)\ .\\ &(\texttt{<}\iota : h;\ \kappa : \texttt{F\_ID}(a)\texttt{>} * \varepsilon_{\texttt{r}}(g)))\\ &= u\end{aligned}$$

we get the desired equation. □

Theorem 7.3.6. Φ_F^X *entails the theory of 2-LCX-F.*

Proof. Reasoning analogously as in the proof of Theorem 7.3.5 we reduce the problem to deriving from the axioms of Φ_F^X equations 191–206 and the two equations implied by the semantic definition of factorizers.

The derivations are similar as for the *2-LC-F* case. As an example, we will derive the equation defining the right factorizer.

$$\begin{aligned}\texttt{T\_rfact}(g, f, t) \;=\;& (\texttt{<}\iota : f;\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \eta_{\texttt{r}}(g))\\ &\ .\ (\texttt{<}\iota = t;\ \kappa = \texttt{T\_ID}(d)\texttt{>} * \texttt{F\_ri}(g))\end{aligned}$$

We start with the right hand side

$$\begin{aligned}&(\texttt{<}\iota : f;\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \eta_{\texttt{r}}(g))\\ &\ .\ (\texttt{<}\iota = t;\ \kappa = \texttt{T\_ID}(d)\texttt{>} * \texttt{F\_ri}(g))\end{aligned}$$

and transform the term to another term with the same semantics, using equation (207)

$$\begin{aligned}&\texttt{T\_rfact}(g, f,\\ &(\texttt{<}\iota = (\texttt{<}\iota : f;\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \eta_{\texttt{r}}(g))\\ &\qquad .\ (\texttt{<}\iota = t;\ \kappa = \texttt{T\_ID}(d)\texttt{>} * \texttt{F\_ri}(g));\ \kappa = \texttt{T\_ID}(d)\texttt{>} * g)\\ &\ .\ (\texttt{<}\iota : h;\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\texttt{r}}(g)))\end{aligned}$$

Then we move the vertical composition outside of the record

$$\begin{array}{l}
\texttt{T\_rfact}(g, f,\\
(\texttt{<}\iota\ \texttt{=}\ (\texttt{<}\iota : f\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \eta_{\mathtt{r}}(g))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\\
\ \ .\ (\texttt{<}\iota\ \texttt{=}\ (\texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * \texttt{F\_ri}(g))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\\
\ \ .\ (\texttt{<}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\mathtt{r}}(g)))
\end{array}$$

and we eliminate the nested record and change associativity, getting

$$\begin{array}{l}
\texttt{T\_rfact}(g, f,\\
(\texttt{<}\iota\ \texttt{=}\ (\texttt{<}\iota : f\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \eta_{\mathtt{r}}(g))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\\
\ \ .\ (\texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * (\texttt{<}\iota : \texttt{F\_ri}(g)\texttt{;}\ \kappa : \texttt{F\_PR}(ld, \kappa)\texttt{>}\ .\ g))\\
\ \ .\ (\texttt{<}\iota : h\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\mathtt{r}}(g)))
\end{array}$$

Since $\varepsilon_{\mathtt{r}}(g)$ is ι-covariant and the κ component of the terms at the left hand sides of multiplications is always an identity, we may now use a variant of equation (188) on the last vertical composition, getting

$$\begin{array}{l}
\texttt{T\_rfact}(g, f,\\
(\texttt{<}\iota\ \texttt{=}\ (\texttt{<}\iota : f\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \eta_{\mathtt{r}}(g))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\\
\ \ .\ (\texttt{<}\iota : \texttt{<}\iota : f\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>}\ .\ g\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\mathtt{r}}(g))\\
\ \ .\ (\texttt{<}\iota\ \texttt{=}\ t\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * \texttt{F\_PR}(ld, \iota)))
\end{array}$$

which is equal to

$$\begin{array}{l}
\texttt{T\_rfact}(g, f,\\
(\texttt{<}\iota\ \texttt{=}\ (\texttt{<}\iota : f\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \eta_{\mathtt{r}}(g))\texttt{;}\ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\\
\ \ .\ (\texttt{<}\iota : \texttt{<}\iota : f\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>}\ .\ g\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\mathtt{r}}(g))\\
\ \ .\ t)
\end{array}$$

Let us define the following shorthand

$$g_1 = \texttt{<}\iota : \texttt{F\_PR}(la, \iota)\texttt{;}\ \kappa : \texttt{F\_PR}(la, \kappa)\ .\ \texttt{F\_PR}(lc, \kappa)\texttt{>}\ .\ g$$

We eliminate $\eta_{\mathtt{r}}$ using equation (211) and present the result using the just introduced shorthand

$$\begin{array}{l}
\texttt{T\_rfact}(g, f,\\
(\texttt{<}\iota\ \texttt{=}\ (\texttt{<}\iota : f\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \texttt{T\_rfact}(g_1, \texttt{F\_PR}(lc, \iota), \texttt{T\_id}(g)))\texttt{;}\\
\ \ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g)\\
\ \ .\ (\texttt{<}\iota : \texttt{<}\iota : f\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>}\ .\ g\texttt{;}\ \kappa : \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\mathtt{r}}(g))\\
\ \ .\ t)
\end{array}$$

Since

$$\texttt{<}\iota\ \texttt{:}\ \texttt{F\_PR}(lb', \iota)\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_PR}(lb', \kappa)\ \texttt{.}\ \texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>>}\ \texttt{.}\ g_1 \quad = \quad g$$

we can use equation (217) to eliminate multiplication

$$\begin{array}{l}
\texttt{T\_rfact}(g, f, \\
(\texttt{<}\iota\ \texttt{=}\ (\texttt{T\_rfact}(g, f, \texttt{T\_id}(\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>}\ \texttt{.}\ g)))\texttt{;} \\
\ \ \kappa\ \texttt{=}\ \texttt{T\_ID}(d)\texttt{>} * g) \\
\ \texttt{.}\ (\texttt{<}\iota\ \texttt{:}\ \texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>}\ \texttt{.}\ g\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>} * \varepsilon_{\texttt{r}}(g)) \\
\ \texttt{.}\ t)
\end{array}$$

The we simplify the term using equation (208)

$$\begin{array}{l}
\texttt{T\_rfact}(g, f, \\
(\texttt{T\_id}(\texttt{<}\iota\ \texttt{:}\ f\texttt{;}\ \kappa\ \texttt{:}\ \texttt{F\_ID}(d)\texttt{>}\ \texttt{.}\ g)) \\
\ \texttt{.}\ t)
\end{array}$$

and we obtain the expected result

$$\texttt{T\_rfact}(g, f, t)$$

The second of the definitional equations for factorizers follows in an analogous way. We obtain equations 191–194 by putting 2-identity for u or t and g_1 for g in each of equations 207–210 and then using one of the just proven equations defining factorizers. □

Appendix C

Program examples in Dule

We provide several longer examples of Dule code taken from the rewrite of the Dule compiler in the Dule language itself. These examples interact with each other, as is visible for instance in the first thirty lines of the code in Section C.2.4, but we will not focus on their relations. Even without considering the global dependencies, the matrix of module interaction within individual examples should be dense enough to demonstrate much of the flavor of fine-grained modular programming.

Names of the modules and functions are either self-explanatory or known from the semantic descriptions in the thesis. Therefore, the reader should be able to grasp the overall meaning while concentrating on the construction of the module hierarchies.

C.1 Standard prelude

The standard prelude is a set of commonly used specifications and libraries automatically provided at the beginning of each compilation, unless the option `--no-prelude` is given to the compiler. The specifications can be recalled and the libraries loaded, as if they were compiled during the elaboration of a preceding file argument. The file containing standard prelude resides at `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/code/prelude.dul`.

Currently the compiler provides the standard prelude by compiling its source code anew on each run. In the future, the standard prelude should be loaded in an already compiled form, taking advantage of the Dule properties of separate compilation. The libraries should also be made more comprehensive, efficient and maintainable. Currently they are rather bizarre and irregular, so as to demonstrate interesting points of Dule and enable testing of rare language constructions.

C.1.1 Module of natural numbers

The first case study taken from the standard prelude presents a module of natural numbers. The module is constructed incrementally (in a rather ad hoc and not extremely fine-grained manner) and the subsequent steps of the construction are expressed modularly. Notice the differences and similarities (despite the differences) in the implementations of `add` and `pred` on one hand and `mult` and `is_zero` on the other hand. The differences are introduced only for the sake of this discussion, normally we would place `pred` and `add` in separate modules and thus eliminate their privileged access to the inductive type.

A more efficient implementation of the natural numbers datatype; for example one based on a binary representation can be obtained just by rewriting the implementation of module `NatInd`. Structural recursion over unary representation of natural numbers would then still be valid in the remaining modules. However, using it would sometimes sidestep the benefits of the type's more complex representation, so changing some of the other individual modules could be needed, as it happens with non-artificial code-reuse scenarios.

```
spec Nat =
  sig
    type t
    value t2ind : ~it:t -> ind nat: [‘Zero|‘Succ nat]
    value tde : ~it:t -> [‘Zero|‘Succ t]
    value zero : t
    value succ : ~n:t -> t
    value pred : ~n:t -> t
    value add : ~n:t ~it:t -> t
    value mult : ~n:t ~it:t -> t
    value sub : ~n:t ~it:t -> t
    value is_zero : ~it:t -> [‘True|‘False]
    value leq : ~n:t ~it:t -> [‘True|‘False]
    value eq : ~n:t ~it:t -> [‘True|‘False]
  end
Nat =
link
  spec NatInd =
    sig
      type t
      value t2ind : ~it:t -> ind nat: [‘Zero|‘Succ nat]
      value tde : ~it:t -> [‘Zero|‘Succ t]
      value zero : t
      value succ : ~n:t -> t
      value pred : ~n:t -> t
      value add : ~n:t ~it:t -> t
    end
  NatInd =
    struct
      type t = ind nat: [‘Zero|‘Succ nat]
      value t2ind = fun ~it -> it
      value tde = fun ~it -> it . de
      value zero = ‘Zero . con
      value succ = fun ~n -> n .‘Succ . con
      value pred = fun ~n ->
        match n . de with
        [‘Zero -> n
        |‘Succ nn -> nn]
      value add = fun ~n ~it ->
        match n with
        fold [‘Zero -> it
             |‘Succ nn -> nn .‘Succ . con]
    end
  spec NatMult =
  ~NatInd ->
    sig
      value mult : NatInd . ~n:t ~it:t -> t
    end
  NatMult =
    struct
      value mult = NatInd . fun ~n ~it ->
        match t2ind ~it with
        fold [‘Zero -> zero
             |‘Succ it -> add ~n ~it]
    end
  spec NatSub =
  ~NatInd ->
    sig
      value is_zero : ~it:NatInd.t -> [‘True|‘False]
      value sub : NatInd . ~n:t ~it:t -> t
    end
  NatSub =
    struct
      value is_zero = fun ~it ->
        match NatInd.tde ~it with
        [‘Zero -> ‘True
        |‘Succ -> ‘False]
      value sub = NatInd . fun ~n ~it ->
        match t2ind ~it with
        fold [‘Zero -> n
             |‘Succ n -> pred ~n]
    end
  spec NatLeq =
  ~NatSub ->
    sig
      value leq : ~n:NatInd.t ~it:NatInd.t -> [‘True|‘False]
    end
  NatLeq =
    struct
```

```
      value leq = fun ~n ~it ->
        NatSub.is_zero ~it:(NatSub.sub ~n ~it)
    end
  NatAll =
    :: ~NatMult ~NatLeq -> Nat
    struct
      type t = NatInd.t
      value t2ind = NatInd.t2ind
      value tde = NatInd.tde
      value zero = NatInd.zero
      value succ = NatInd.succ
      value pred = NatInd.pred
      value add = NatInd.add
      value mult = NatMult.mult
      value sub = NatSub.sub
      value is_zero = NatSub.is_zero
      value leq = NatLeq.leq
      value eq = NatLeq . fun ~n ~it ->
        match leq ~n ~it with
        [`True -> leq ~n:it ~it:n
        |`False -> `False]
    end
end | NatAll
```

C.1.2 Module of character strings

The module of character strings from the standard prelude makes use of the modules of booleans, characters and lists. We present the module of character literals in abbreviated form, not spelling out most of the boring details, which we will probably implement using compiler primitives, eventually. The strings are implemented as lists of characters and the implementation is not hidden — the module openly carries the type of character lists in itself. To make up for the temporary absence of special lexer support for string constants, we extend the module with hardwired names of special identifiers, such as `it` and `atu`. The equality predicate on strings is implemented in an interesting way, using solely structured recursion. First, we create a function by folding over the second argument of the predicate and then we apply the function to the first argument.

```
spec Bool =
  sig
    value tt : [`True|`False]
    value ff : [`True|`False]
    value neg : ~it:[`True|`False] -> [`True|`False]
    value conj : ~b:[`True|`False] ~it:[`True|`False] -> [`True|`False]
    value disj : ~b:[`True|`False] ~it:[`True|`False] -> [`True|`False]
  end
Bool =
  struct
    value tt = ‘True
    value ff = ‘False
    value neg = [‘True -> ‘False
                |‘False -> ‘True]
    value conj = [‘True fun ~b ~it -> b
                 |‘False fun ~b ~it -> ‘False]
    value disj = [‘True fun ~b ~it -> ‘True
                 |‘False fun ~b ~it -> b]
  end
spec Char =
~Nat ->
  sig
    type t
    value tde : ~it:t ->
      [‘HT9|‘LF10|‘space|‘bang|‘quote|‘hash|‘buck|‘mod|‘amp|‘tick
      |‘left|‘right|‘star|‘plus|‘comma|‘hyphen|‘dot|‘slash
      |‘d0|‘d1|‘d2|‘d3|‘d4|‘d5|‘d6|‘d7|‘d8|‘d9
      |‘colon|‘semi|‘from|‘equals|‘to|‘what
      |‘at|‘A|‘B|‘C|‘D|‘E|‘F|‘G|‘H|‘I|‘J|‘K|‘L|‘M|‘N|‘O|‘P|‘Q
      |‘R|‘S|‘T|‘U|‘V|‘W|‘X|‘Y|‘Z|‘square|‘back|‘unsquare|‘hat|‘floor
      |‘grave|‘a|‘b|‘c|‘d|‘e|‘f|‘g|‘h|‘i|‘j|‘k|‘l|‘m|‘n|‘o|‘p|‘q
      |‘r|‘s|‘t|‘u|‘v|‘w|‘x|‘y|‘z|‘brace|‘pipe|‘unbrace|‘tilde
      |‘UNPRINTABLE]
    value tcon : ~it:
      [‘HT9|‘LF10|‘space|‘bang|‘quote|‘hash|‘buck|‘mod|‘amp|‘tick
      |‘left|‘right|‘star|‘plus|‘comma|‘hyphen|‘dot|‘slash
      |‘d0|‘d1|‘d2|‘d3|‘d4|‘d5|‘d6|‘d7|‘d8|‘d9
      |‘colon|‘semi|‘from|‘equals|‘to|‘what
      |‘at|‘A|‘B|‘C|‘D|‘E|‘F|‘G|‘H|‘I|‘J|‘K|‘L|‘M|‘N|‘O|‘P|‘Q
      |‘R|‘S|‘T|‘U|‘V|‘W|‘X|‘Y|‘Z|‘square|‘back|‘unsquare|‘hat|‘floor
      |‘grave|‘a|‘b|‘c|‘d|‘e|‘f|‘g|‘h|‘i|‘j|‘k|‘l|‘m|‘n|‘o|‘p|‘q
      |‘r|‘s|‘t|‘u|‘v|‘w|‘x|‘y|‘z|‘brace|‘pipe|‘unbrace|‘tilde
      |‘UNPRINTABLE] -> t
    value t2nat : ~it:t -> Nat.t
    value nat2t : ~it:Nat.t -> t
    value eq : ~c:t ~it:t -> [‘True|‘False]
    value is_upper : ~it:t -> [‘True|‘False]
    value upper : ~it:t -> t
    value lower : ~it:t -> t
  end
Char =
  :: ~Bool -> Char
  struct
    type t = Nat.t (* ASCII codes *)
    value tde = fun ~it ->
      if Nat.eq ~n:65 ~it then ‘A
      else if Nat.eq ~n:66 ~it then ‘B
      else if Nat.eq ~n:67 ~it then ‘C
      else ‘UNPRINTABLE (* TODO *)
    value tcon = [‘A -> 65|‘B -> 66|‘C -> 67
                 |_ -> 33 (* TODO *)]
    value t2nat = fun ~it -> it
    value nat2t = fun ~it -> it
    value eq = fun ~c ~it -> Nat.eq ~n:c ~it
    value is_upper = fun ~it ->
      Bool.conj
        ~b:(Nat.leq ~n:65 ~it)
        ~it:(Nat.leq ~n:it ~it:90)
    value upper = fun ~it -> Nat.sub ~n:it ~it:32
    value lower = fun ~it -> Nat.add ~it ~n:32
  end
spec Elem = sig type t end
spec List =
~Elem ->
  sig
    type t
    value t2ind : ~it:t -> ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value ind2t : ~it:ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}] -> t
    value tde : ~it:t -> [‘Nil|‘Cons {head : Elem.t; tail : t}]
    value nil : t
    value cons : ~head:Elem.t ~tail:t -> t
  end
library List =
  struct
    type t = ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value t2ind = fun ~it -> it
    value ind2t = fun ~it -> it
    value tde = fun ~it -> it . de
    value nil = ‘Nil . con
    value cons = fun ~head ~tail -> {head; tail} .‘Cons . con
  end
spec ListOps =
~List ->
  sig
    value append : List . ~l1:t ~l2:t -> t
    value is_nil : ~l:List.t -> [‘True|‘False]
    value endo_map : ~f: ~it:Elem.t -> Elem.t ~l:List.t -> List.t
  end
library ListOps =
  struct
    value append = fun ~l1 ~l2 ->
      match List.t2ind ~it:l1 with
      fold [‘Nil -> l2
           |‘Cons ht -> List.cons ~head:ht.head ~tail:ht.tail]
    value is_nil = fun ~l ->
      match List.tde ~it:l with
      [‘Nil -> ‘True
      |‘Cons -> ‘False]
    value endo_map = fun ~f ~l ->
```

```
      List.ind2t
        ~it:(match List.t2ind ~it:l with
             map f)
  end
spec CharList = List with {Elem = Char}
CharList = load List with {Elem = Char}
spec CharListOps = ListOps with {Elem = Char; List = CharList}
CharListOps = load ListOps with {Elem = Char; List = CharList}
spec String =
~CharList ->
  sig
    value eq : ~l:CharList.t ~it:CharList.t -> [‘True|‘False]
    value append : CharList . ~l1:t ~l2:t -> t
    value prefix_by : ~head:Char.t ~tail:CharList.t -> CharList.t
    value shout : CharList . ~it:t -> t
    value atu : CharList.t
    value it : CharList.t
  end
String =
  :: ~Bool ~CharListOps -> String
  struct
    value eq =
      let non_nil_eq = fun ~it -> fun ~l ->
        match CharList.tde ~it:l with
        [‘Nil -> Bool.ff
        |‘Cons lht ->
           let head_matches = Char.eq ~c:it.head ~it:lht.head
               tail_equal = it.tail ~l:lht.tail in
           Bool.conj ~b:head_matches ~it:tail_equal]
      in
      fun ~l ~it ->
        (match CharList.t2ind ~it with
         fold [‘Nil -> CharListOps.is_nil
              |‘Cons non_nil_eq]) ~l
    value append = CharListOps.append
    value prefix_by = CharList.cons
    value shout = fun ~it ->
      CharListOps.endo_map ~l:it ~f:Char.upper
    value atu = CharList.cons ~head:(Char.tcon ~it:‘a)
                  ~tail:(CharList.cons ~head:(Char.tcon ~it:‘t)
                           ~tail:(CharList.cons ~head:(Char.tcon ~it:‘u)
                                    ~tail:CharList.nil))
    value it = CharList.cons ~head:(Char.tcon ~it:‘i)
                 ~tail:(CharList.cons ~head:(Char.tcon ~it:‘t)
                          ~tail:CharList.nil)
   end
```

C.2 Dule compiler written in Dule

Samples of a study version of the Dule compiler written in Dule are presented here. The code of the second, third and fourth examples contains the complete semantic definition of the Dule module language — an equivalent of all the OCaml code presented in Chapter 5 and Section 9.1. On the other hand, the Dule version of the compiler lacks most boring, basic tools and technical components, and is not planned to bootstrap in the near future. Moreover, the part of the compiler implementing the core language is represented here only by the first and shortest code snippet. For a more complete picture of the study compiler code, the reader is referred to the file `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/test/compiler.dul`.

C.2.1 Assigning categories to functors

The first code fragment is an implementation of the assignment of source and target categories to functors of the internal core language, as summarized in Appendix A.1.3. The partial correctness of this code follows from Definition 7.2.8 and Observation 7.3.3. The pattern of recursion (in the case of `src`, as simple as structured recursion) and Theorem 6.1.4 assure that the code is terminating.

The `trg_bad` variant of the target function is not working as desired, because the inductive nature of the type of functors is manually recovered from a set of mutually dependent modules in a too weak manner. Consequently, the type does not allow structured recursion across some of its component types and into its nested instances. The problem can be overcome using general recursion, as shown in the code, or more complex typing of the `t2ind` conversion function, which is not a practical solution, or intermodule structured recursion, which is a future work topic.

We prepend the definition of the main module `SrcCore` by most of the code of modules defining the used types. Many other modules mentioned in the code are omitted. Operation `MapCF.vmap_fc` is the mapping operation `vmap` described in Section 3.2, instantiated to functions from the set of functors to the set of categories.

```
spec rec CatIList = IdIList with {{Value = Cat}}
and Cat =
~IdIndex ->
  sig
    type t
    value tde : ~c:t ->
             [‘C_PP CatIList.t
             |‘C_BB]
    value c_PP : ~lc:CatIList.t -> t
    value c_BB : t
  end
module ind CatIList = load IdIList with {{Value = Cat}}
and Cat = (* here and below only normal form terms, no reduction, yet *)
  struct
    type t =
      ind t: [‘C_PP CatIList.t
             |‘C_BB]
    value tde = fun ~c -> c . de
    value c_PP = fun ~lc -> lc .‘C_PP . con
    value c_BB = ‘C_BB . con
  end
spec rec FunctIList = IdIList with {{Value = Funct}}
and Funct =
~Cat ->
  sig
    type t
    value t2ind : ~f:t ->
      ind t: [‘F_ID Cat.t
             |‘F_COMP {f1 : t; f2 : t}
             |‘F_PR {lc : CatIList.t; i : IdIndex.t}
             |‘F_RECORD {c : Cat.t; lf : FunctIList.t}
             |‘F_pp {c : Cat.t; lf : FunctIList.t}
             |‘F_ss {c : Cat.t; lf : FunctIList.t}
             |‘F_ii t
             |‘F_tt t
             |‘F_ee {lg : FunctIList.t; h : t}]
    value tde : ~f:t ->
             [‘F_ID Cat.t
             |‘F_COMP {f1 : t; f2 : t}
             |‘F_PR {lc : CatIList.t; i : IdIndex.t}
             |‘F_RECORD {c : Cat.t; lf : FunctIList.t}
             |‘F_pp {c : Cat.t; lf : FunctIList.t}
             |‘F_ss {c : Cat.t; lf : FunctIList.t}
             |‘F_ii t
             |‘F_tt t
             |‘F_ee {lg : FunctIList.t; h : t}]
    value f_ID : ~c:Cat.t -> t
    value f_COMP : ~f1:t ~f2:t -> t
    value f_PR : ~lc:CatIList.t ~i:IdIndex.t -> t
    value f_RECORD : ~c:Cat.t ~lf:FunctIList.t -> t
    value f_pp : ~c:Cat.t ~lf:FunctIList.t -> t
    value f_ss : ~c:Cat.t ~lf:FunctIList.t -> t
    value f_ii : ~g:t -> t
    value f_tt : ~g:t -> t
    value f_ee : ~lg:FunctIList.t ~h:t -> t
  end
module ind FunctIList = load IdIList with {{Value = Funct}}
and Funct =
  struct
    type t =
        ind t: [‘F_ID Cat.t
               |‘F_COMP {f1 : t; f2 : t}
               |‘F_PR {lc : CatIList.t; i : IdIndex.t}
               |‘F_RECORD {c : Cat.t; lf : FunctIList.t}
               |‘F_pp {c : Cat.t; lf : FunctIList.t}
               |‘F_ss {c : Cat.t; lf : FunctIList.t}
               |‘F_ii t
               |‘F_tt t
               |‘F_ee {lg : FunctIList.t; h : t}]
      value t2ind = fun ~f -> f
      value tde = fun ~f -> f . de
      value f_ID = fun ~c -> c .‘F_ID . con
(* below, in the definition of f_COMP,
   rewriting according to system R_L should be implemented *)
      value f_COMP = (* ... *)
(* currently just a stub: *)
        fun ~f1 ~f2 -> {f1; f2} .‘F_COMP . con
      value f_PR = fun ~lc ~i -> {lc; i} .‘F_PR . con
      value f_RECORD = fun ~c ~lf -> {c; lf} .‘F_RECORD . con
      value f_pp = fun ~c ~lf -> {c; lf} .‘F_pp . con
      value f_ss = fun ~c ~lf -> {c; lf} .‘F_ss . con
      value f_ii = fun ~g -> g .‘F_ii . con
      value f_tt = fun ~g -> g .‘F_tt . con
      value f_ee = fun ~lg ~h -> {lg; h} .‘F_ee . con
    end
(* ... *)
  spec SrcCore =
  ~Funct ->
    sig
      value src : ~f:Funct.t -> Cat.t
      value trg : ~f:Funct.t -> Cat.t
      value trg_bad : ~f:Funct.t -> Cat.t
    end
  module SrcCore =
    :: ~AtIndex ~CatIListOps ~MapCF ~EqIListCat ~SemCat -> SrcCore
    struct
      value src =
        let fing = fun ~it ->
          let l = SemCat.unPP ~c:it in
          CatIListOps.find ~i:AtIndex.atk ~l
        in
        fun ~f ->
          match Funct.t2ind ~f with
          fold [‘F_ID c -> c
               |‘F_COMP {f1; f2} -> f1
               |‘F_PR {lc; i} -> Cat.c_PP ~lc
               |‘F_RECORD {c; lf} -> c
               |‘F_pp {c; lf} -> c
               |‘F_ss {c; lf} -> c
               |‘F_ii fing
```

```
              |‘F_tt fing
              |‘F_ee {lg; h} -> h]
      value rec trg = fun ~f ->
        match Funct.tde ~f with
        [‘F_ID c -> c
        |‘F_COMP {f1; f2} -> trg ~f:f2
        |‘F_PR {lc; i} -> CatIListOps.find ~i ~l:lc
        |‘F_RECORD {c; lf} ->
            Cat.c_PP ~lc:(MapCF.vmap_fc ~f:(fun ~v -> trg ~f:v) ~l:lf)
        |‘F_pp -> Cat.c_BB
        |‘F_ss -> Cat.c_BB
        |‘F_ii g -> trg ~f:g
        |‘F_tt g -> trg ~f:g
        |‘F_ee -> Cat.c_BB]
      value trg_bad = fun ~f ->
        match Funct.t2ind ~f with
        fold [‘F_ID c -> c
             |‘F_COMP {f1; f2} -> f2
             |‘F_PR {lc; i} -> CatIListOps.find ~i ~l:lc
             |‘F_RECORD {c; lf} -> fail (* FIXME: Funct.t2ind weak *)
             |‘F_pp -> Cat.c_BB
             |‘F_ss -> Cat.c_BB
             |‘F_ii g -> g
             |‘F_tt g -> g
             |‘F_ee -> Cat.c_BB]
    end
```

C.2.2 Implementation of SCM

The following two modules implement the Simple Category of Modules, as defined in Section 5.1. Just as in the OCaml version of the compiler code, we state the main invariants of the operations using both informal comments and run-time tested assertions. The correctness of the invariants, and so of the code, trivially follows from the fact that the OCaml renditions of the same invariants are used as definitions of the implemented notions in Section 5.1.

```
  spec Sign =
  ~Funct ->
    sig
      type t
      value f2s : ~f:Funct.t -> t
(*
   the application [f2s f] results in a signature
   iff [f] is of the form [f_pp (c_PP lc) lf]
*)
      value s2f : ~s:t -> Funct.t
      value type_part : ~s:t -> CatIList.t
      value value_part : ~s:t -> FunctIList.t
    end
  Sign =
    :: ~SemCat ~SrcCore ~SemFunct -> Sign
    struct
      type t = Funct.t
      value f2s = fun ~f ->
        assert
          match SemFunct.unpp_ok ~f with
          [`OK _ ->
            match SemCat.unPPok ~c:(SrcCore.src ~f) with
            [`OK _ -> `True
            |`Error -> `False]
          |`Error -> `False]
        in f
      value s2f = fun ~s -> s
      value type_part = fun ~s:f -> SemCat.unPP ~c:(SrcCore.src ~f)
      value value_part = fun ~s:f -> SemFunct.unpp ~f
    end
  spec Dule =
  ~Trans ~Sign ->
    sig
      type t
      value pack : ~f:Funct.t ~t:Trans.t ~s:Sign.t -> t
(*
   the application [pack (f, t, s)] results in a module
   iff there is a signature [r] such that
   1. f : src r -> src s
   2. t : r -> f_COMP f s
*)
      value domain : ~m:t -> Sign.t
      value codomain : ~m:t -> Sign.t
      value type_part : ~m:t -> Funct.t
      value value_part : ~m:t -> Trans.t
    end
  Dule =
    :: ~Bool ~SrcCore ~DomCore ~EqCat ~EqFunct -> Dule
    struct
      type t = {f : Funct.t; t : Trans.t; s : Sign.t}
      value pack = fun ~f ~t ~s ->
        assert
          let g = DomCore.dom ~t in
          let r = Sign.f2s ~f:g in
          let h = Sign.s2f ~s in
          Bool.conj
            ~b:(Bool.conj
                  ~b:(EqCat.eq
                        ~it:(SrcCore.src ~f)
                        ~v:(SrcCore.src ~f:g))
                  ~it:(EqCat.eq
                         ~it:(SrcCore.trg ~f)
```

```
                      ~v:(SrcCore.src ~f:h)))
          ~it:(EqFunct.eq
                 ~it:(DomCore.cod ~t)
                 ~v:(Funct.f_COMP ~f1:f ~f2:h))
        in
        {f; t; s}
      value domain = fun ~m ->
        Sign.f2s ~f:(DomCore.dom ~t:m.t)
      value codomain = fun ~m -> m.s
      value type_part = fun ~m -> m.f
      value value_part = fun ~m -> m.t
    end
```

C.2.3 Implementation of W-Dule and L-Dule

The compiler fragment presented in this section, the first of the two longer fragments, implements module languages W-Dule and L-Dule with all their semantic details. The correctness (well definedness with respect to SCM and adherence to the categorical description of Chapter 5) of this part of the compiler code is argued in Section 5.2 and 5.3 — in the proofs of Theorem 5.2.1 and 5.2.7, in particular. The narrow interface between W-Dule and L-Dule is here presented using one of many possible conventions involving linking expressions and their name-space visibility rules; see the next section for further discussion.

```
(* WDule is the simplest module engine;
   a link expression is used to group and hide auxiliary modules
   while implementing SemWSign and SemWDule that encode its semantics *)
  WDule =
link
  spec PartitionWSign =
  ~Sign ->
    sig
      value partition : ~s:Sign.t ->
        {la : CatIList.t;
         lb : CatIList.t}
    end
  PartitionWSign = (* W-Dule --- module system with specialization *)
    :: ~CatIListOps ~FunctIListOps ~EqIListCat ~SemCat -> PartitionWSign
    struct
      value partition = fun ~s ->
        let le = Sign.type_part ~s in
        let lty = CatIListOps.vfilter ~p:SemCat.isBB ~l:le in
        let lep = EqIListCat.diff ~l1:le ~l2:lty in
        (* if [s] has been [S_Pp] (perhaps inside [S_Ww]) *)
        (* then [lty] is [nil] else [lePp] is [nil]: *)
        let lh = Sign.value_part ~s in
        let is_in_lh = fun ~it -> FunctIListOps.is_in ~i:it ~l:lh in
        let lePp = CatIListOps.ifilter ~p:is_in_lh ~l:lep in
      let la = CatIListOps.append ~l1:lty ~l2:lePp
           (* local types of [s] *)
          lb = EqIListCat.diff ~l1:lep ~l2:lePp
           (* context types of [s] *) in
      {la; lb}
    end
  spec TypesWSign =
  ~SignIList ->
    sig
      value typesPp : ~ls:SignIList.t -> [`OK CatIList.t|`Error]
      value typesBb : ~r:Sign.t ~la:CatIList.t -> [`OK CatIList.t|`Error]
      value typesWw : ~f1:Funct.t ~s2:Sign.t ->
        [`OK {lc1 : CatIList.t;
              le1 : CatIList.t;
              la : CatIList.t;
              lb : CatIList.t;
              lc : CatIList.t}
        |`Error]
    end
  TypesWSign =
    :: ~CatIListOps ~EqIListCat ~SemCat ~SrcCore
       ~PartitionWSign -> TypesWSign
    struct
      value typesPp = fun ~ls -> (* type part of [S_Pp(ls)] *)
        let name_local = fun ~it ->
          match it with
          [`Nil -> CatIList.nil .`OK
          |`Cons {i; v; l} ->
              match l with
              [`OK r ->
                  let {la; lb} = PartitionWSign.partition ~s:v in
                  let l1 = CatIList.cons ~i ~v:(Cat.c_PP ~lc:la)
                    ~l:CatIList.nil in
                  match EqIListCat.append_ok ~l1 ~l2:lb with
                  [`OK lc ->
                      EqIListCat.append_ok ~l1:lc ~l2:r
                  |`Error er -> er .`Error]
              |`Error er -> er .`Error]]
        in
        match SignIList.t2ind ~it:ls with
        fold name_local
      value typesBb = fun ~r ~la -> (* type part of [S_Bb(r, la, lf)] *)
        let ld = Sign.type_part ~s:r in (* some will be hidden *)
        if CatIListOps.vforall ~p:SemCat.isPP ~l:la then
          let lb = CatIListOps.subtract ~l1:ld ~l2:la in (* context types *)
          if CatIListOps.vforall ~p:SemCat.isBB ~l:la then
            (* we merge local and context types: *)
            (CatIListOps.append ~l1:la ~l2:lb) .`OK
          else `Error
        else `Error
      value typesWw = fun ~f1 ~s2 ->
        (* f1 = Dule.type_part m1,
         m1 : r1 -> s1
         *)
        let lc1 = SemCat.unPP ~c:(SrcCore.src ~f:f1)
              (* type part of [r1] *)
            le1 = SemCat.unPP ~c:(SrcCore.trg ~f:f1)
              (* type part of [s1] *) in
        let {la; lb} = PartitionWSign.partition ~s:s2 in
        if EqIListCat.subset ~l1:lb ~l2:le1 then
          match EqIListCat.append_ok ~l1:la ~l2:lc1 with
          [‘OK lc -> (* type part of [S_Ww (m1, s2)] *)
             {lc1; le1; la; lb; lc} .‘OK
          |‘Error er -> er .‘Error]
        else ‘Error
    end
  spec FootWSign =
  ~Sign ->
    sig
      value footPp : ~lc:CatIList.t ~i:IdIndex.t ~s:Sign.t -> Funct.t
    end
  FootWSign =
    :: ~CatIListOps ~FunctIListOps ~MapCF ~EqIListCat ~SemCat -> FootWSign
    struct
     value footPp = fun ~lc ~i ~s ->
      (* [lc] is the type part of a product signature
         of which [s] is the component at label [i]
       *)
        let le = Sign.type_part ~s
            li = SemCat.unPP ~c:(CatIListOps.find ~i ~l:lc) in
        let lb = CatIListOps.subtract ~l1:le ~l2:li in
        let la = EqIListCat.diff ~l1:le ~l2:lb in
        let f_PR_lc_i = Funct.f_PR ~lc ~i in
        let pia = MapCF.imap_cf ~f:(fun ~i ->
          Funct.f_COMP ~f1:f_PR_lc_i ~f2:(Funct.f_PR ~lc:la ~i)) ~l:la in
        let pib = MapCF.imap_cf ~f:Funct.f_PR(~lc) ~l:lb in
        let c = Cat.c_PP ~lc in
        Funct.f_RECORD ~c ~lf:(FunctIListOps.append ~l1:pia ~l2:pib)
    end
  spec SemWSign =
  ~SignIList ~Dule ->
    sig
      value s_Pp : ~ls:SignIList.t -> [‘OK Sign.t|‘Error]
      value s_Bb : ~r:Sign.t ~la:CatIList.t ~lf:FunctIList.t ->
        [‘OK Sign.t|‘Error]
      value s_Ww : ~m1:Dule.t ~s2:Sign.t -> [‘OK Sign.t|‘Error]
    end
  SemWSign =
    :: ~CatIListOps ~FunctIListOps ~EqCat ~MapCF ~SrcCore
       ~FootWSign ~TypesWSign -> SemWSign
```

```
struct
  value s_Pp = fun ~ls ->
    match TypesWSign.typesPp ~ls with
    [‘OK lc ->
        let legPp = fun ~i ~v:s ->
          let foot = FootWSign.footPp ~lc ~i ~s in
          Funct.f_COMP ~f1:foot ~f2:(Sign.s2f ~s)
        in
        let bmap_sf = fun ~f ~l ->
          match SignIList.t2ind ~it:l with
          fold [‘Nil -> FunctIList.nil
               |‘Cons {i; v; l} ->
                   let v = f ~i ~v in
                   FunctIList.cons ~i ~v ~l]
        in
        let legs = bmap_sf ~f:legPp ~l:ls in
        let c = Cat.c_PP ~lc in
        let body = Funct.f_pp ~c ~lf:legs in
        (Sign.f2s ~f:body) .‘OK
    |‘Error er -> er .‘Error]
  value s_Bb = fun ~r ~la ~lf ->
    (* [la] are local types of this signature,
       [lf] is the value part of this signature
     *)
    match TypesWSign.typesBb ~r ~la with
    [‘OK lc -> (* type part of the signature *)
        let c = Cat.c_PP ~lc in
        let ld = MapCF.vmap_fc ~f:(fun ~v ->
          SrcCore.src ~f:v) ~l:lf in
        match CatIListOps.vforall ~p:EqCat.eq(~v:c) ~l:ld with
        [‘True ->
            let le = MapCF.vmap_fc ~f:(fun ~v ->
              SrcCore.trg ~f:v) ~l:lf in
            match CatIListOps.vforall
              ~p:EqCat.eq(~v:Cat.c_BB) ~l:le with
            [‘True ->
                (Sign.f2s ~f:(Funct.f_pp ~c ~lf)) .‘OK
            |‘False -> ‘Error]
        |‘False -> ‘Error]
    |‘Error er -> er .‘Error]
  value s_Ww = fun ~m1 ~s2 ->
    let f1 = Dule.type_part ~m:m1 in
    match TypesWSign.typesWw ~f1 ~s2 with
    [‘OK {lc1; le1; la; lb; lc} ->
       let c = Cat.c_PP ~lc in
       let pic1 = MapCF.imap_cf ~f:Funct.f_PR(~lc) ~l:lc1 in
       let re1 = Funct.f_RECORD ~c ~lf:pic1 in
       let f = Funct.f_COMP ~f1:re1 ~f2:f1 in
       let f_COMP_f = fun ~i ->
         Funct.f_COMP ~f1:f ~f2:(Funct.f_PR ~lc:le1 ~i) in
```

```
          let pib = MapCF.imap_cf ~f:f_COMP_f ~l:lb
              pia = MapCF.imap_cf ~f:Funct.f_PR(~lc) ~l:la in
          let pipi = FunctIListOps.append ~l1:pib ~l2:pia in
          let re = Funct.f_RECORD ~c ~lf:pipi in
          let h = Funct.f_COMP ~f1:re ~f2:(Sign.s2f ~s:s2) in
          (Sign.f2s ~f:h) .‘OK
       |‘Error er -> er .‘Error]
    end
  spec OkWDule =
  ~Trans ~SignIList ->
    sig
      value typesRecord : ~lf:FunctIList.t ~ls:SignIList.t ->
        [‘OK FunctIList.t|‘Error]
      value typesBase : ~r:Sign.t ~s:Sign.t ~lg:FunctIList.t ->
        [‘OK Funct.t|‘Error]
      value typesInst : ~f1:Funct.t ~f2:Funct.t ~s2:Sign.t ->
        [‘OK Funct.t|‘Error]
      value typesTrim : ~e:Cat.t ~c:Cat.t -> [‘OK Funct.t|‘Error]
      value valuesTrim : ~h:Funct.t ~f:Funct.t -> [‘OK Trans.t|‘Error]
    end
  OkWDule =
    :: ~CatIListOps ~FunctIListOps ~SignIListOps ~MapCF ~MapFT
       ~EqIListCat ~EqIListFunct ~SemCat ~SrcCore ~SemFunct
       ~PartitionWSign ~TypesWSign -> OkWDule
    struct
      value typesRecord = fun ~lf ~ls ->
        let cut_at_i = fun ~i ~v ~l ->
          let s = SignIListOps.find ~i ~l:ls in
          let {la; lb} = PartitionWSign.partition ~s in
          let le = CatIListOps.append ~l1:la ~l2:lb in
          let e = Cat.c_PP ~lc:le in
          let lfib = MapCF.imap_cf ~f:(fun ~i ->
            Funct.f_COMP ~f1:v ~f2:(Funct.f_PR ~lc:le ~i)) ~l:lb in
          let pia = MapCF.imap_cf ~f:Funct.f_PR(~lc:le) ~l:la in
          let fia =
            Funct.f_COMP ~f1:v ~f2:(Funct.f_RECORD ~c:e ~lf:pia) in
          let l1 = FunctIList.cons ~i ~v:fia ~l:FunctIList.nil in
          match EqIListFunct.append_ok ~l1 ~l2:lfib with
          [‘OK lg ->
              EqIListFunct.append_ok ~l1:lg ~l2:l
          |‘Error er -> er .‘Error]
        in
        FunctIListOps.bfold1ok ~init:FunctIList.nil ~f:cut_at_i ~l:lf
      value typesBase = fun ~r ~s ~lg ->
        let lc = Sign.type_part ~s:r in
        let c = Cat.c_PP ~lc in
        let ld = MapCF.vmap_fc ~f:(fun ~v ->
          SrcCore.src ~f:v) ~l:lg in
        match CatIListOps.vforall ~p:EqCat.eq(~v:c) ~l:ld with
        [‘True ->
```

```
          let {la; lb} = PartitionWSign.partition ~s in
          let le = MapCF.vmap_fc ~f:(fun ~v ->
            SrcCore.trg ~f:v) ~l:lg in
          if EqIListCat.eqset ~l1:le ~l2:la then
            if EqIListCat.subset ~l1:lb ~l2:lc then
              let pilb = MapCF.imap_cf ~f:Funct.f_PR(~lc) ~l:lb in
              let lf = FunctIListOps.append ~l1:lg ~l2:pilb in
              let f = Funct.f_RECORD ~c ~lf in
              f .‘OK
            else ‘Error
          else ‘Error
      |‘False -> ‘Error]
    value typesInst = fun ~f1 ~f2 ~s2 ->
      match TypesWSign.typesWw ~f1 ~s2 with
      [‘OK {lc1; le1; la; lb; lc(*_*)} ->
          let le2 = CatIListOps.append ~l1:la ~l2:lb in
          let f12 = Funct.f_COMP ~f1 ~f2 in
          let pia = MapCF.imap_cf ~f:(fun ~i ->
            Funct.f_COMP ~f1:f12 ~f2:(Funct.f_PR ~lc:le2 ~i)) ~l:la in
          let pic1 = MapCF.imap_cf ~f:Funct.f_PR(~lc:lc1) ~l:lc1 in
          match EqIListFunct.append_ok ~l1:pia ~l2:pic1 with
          [‘OK lf ->
            let pib = MapCF.imap_cf ~f:(fun ~i ->
              Funct.f_COMP ~f1:f2 ~f2:(Funct.f_PR ~lc:le2 ~i)) ~l:lb in
            let pie1 = MapCF.imap_cf ~f:Funct.f_PR(~lc:le1) ~l:lb in
            if EqIListFunct.eqset ~l1:pib ~l2:pie1 then
              (Funct.f_RECORD ~c:(Cat.c_PP ~lc:lc1) ~lf) .‘OK
            else ‘Error
          |‘Error er -> er .‘Error]
      |‘Error er -> er .‘Error]
    value rec typesTrim = fun ~e ~c ->
      match EqCat.eq ~it:e ~v:c with
      [‘True ->
          Funct.f_ID ~c .‘OK
      |‘False ->
          match SemCat.unPPok ~c with
          [‘OK lc ->
             match SemCat.unPPok ~c:e with
             [‘OK le ->
                 let fsu = fun ~i ~v ->
                   match CatIListOps.find_ok ~i ~l:le with
                   [‘OK e ->
                       match typesTrim ~e ~c:v with
                       [‘OK sf ->
                           (Funct.f_COMP
                              ~f1:(Funct.f_PR ~lc:le ~i)
                              ~f2:sf) .‘OK
                       |‘Error er -> er .‘Error]
                   |‘Error er -> er .‘Error]
                 in
                 match MapCF.bmap1ok_cf ~f:fsu ~l:lc with
                 [‘OK lf ->
                     (Funct.f_RECORD ~c:e ~lf) .‘OK
                 |‘Error er -> er .‘Error]
             |‘Error er -> er .‘Error]
          |‘Error er -> er .‘Error]]
    value rec valuesTrim = fun ~h ~f ->
      match EqFunct.eq ~it:h ~v:f with
        [‘True ->
            Trans.t_id ~f .‘OK
        |‘False ->
            match SemFunct.unpp_ok ~f with
            [‘OK lf ->
                match SemFunct.unpp_ok ~f:h with
                [‘OK lh ->
                    let fsu = fun ~i ~v ->
                      match FunctIListOps.find_ok ~i ~l:lh with
                      [‘OK h ->
                          match valuesTrim ~h ~f:v with
                          [‘OK sf ->
                              (Trans.t_comp
                                 ~t1:(Trans.t_pr ~lf:lh ~i)
                                 ~t2:sf) .‘OK
                          |‘Error er -> er .‘Error]
                      |‘Error er -> er .‘Error]
                    in
                    match MapFT.bmap1ok_ft ~f:fsu ~l:lf with
                    [‘OK lt ->
                        (Trans.t_record ~f:h ~lt) .‘OK
                    |‘Error er -> er .‘Error]
                |‘Error er -> er .‘Error]
            |‘Error er -> er .‘Error]]
  end
spec SemWDule =
~Trans ~SignIList ~DuleIList ->
  sig
    value m_Id : ~s:Sign.t -> Dule.t
    value m_Comp : ~m1:Dule.t ~m2:Dule.t -> Dule.t
    value m_Pr : ~lr:SignIList.t ~i:IdIndex.t ->
      [‘OK Dule.t|‘Error]
    value m_Record : ~r:Sign.t ~lm:DuleIList.t ->
      [‘OK Dule.t|‘Error]
    value m_Base : ~r:Sign.t ~s:Sign.t
                   ~lg:FunctIList.t ~lt:TransIList.t ->
                     [‘OK Dule.t|‘Error]
    value m_Inst : ~m1:Dule.t ~m2:Dule.t -> [‘OK Dule.t|‘Error]
    value m_Trim : ~m1:Dule.t ~r2:Sign.t -> [‘OK Dule.t|‘Error]
  end
SemWDule =
  :: ~FunctIListOps ~TransIListOps ~MapFT ~EqFunct ~SrcCore ~DomCore
```

```
      ~FootWSign ~SemWSign ~OkWDule ~SignIListOps -> SemWDule
    struct
(*
   Invariant:
     If m : r -> s type-correct then [m] is a module with [r], [s].
*)
      value m_Id = fun ~s -> (* : s -> s *)
        let h = Sign.s2f ~s in
        let c = SrcCore.src ~f:h in
        Dule.pack ~f:(Funct.f_ID ~c) ~t:(Trans.t_id ~f:h) ~s
      value m_Comp = fun ~m1 ~m2 -> (* : r1 -> s2 *)
        let f1 = Dule.type_part ~m:m1 (* : r1 -> s1 *)
            t1 = Dule.value_part ~m:m1
            f2 = Dule.type_part ~m:m2 (* : r2 -> s2 *)
            t2 = Dule.value_part ~m:m2 in
        let f = Funct.f_COMP ~f1 ~f2 (* s1 = r2 *)
            it2 = Trans.t_FT ~f1 ~t2 in
        let t = Trans.t_comp ~t1 ~t2:it2
            s2 = Dule.codomain ~m:m2 in
        Dule.pack ~f ~t ~s:s2
(*
The following drawing shows the domains and codomains
of the transformations appearing in the definition of [m_Comp].

       t1    f1         t2    f2         t    f1
  r1 ------> --    r2 ------> --    r1 -----> --
             s1               s2              f2
                                              --
                                              s2

The drawing below illustrates the value part
of the result of module composition.
Horizontal composition of transformations is here represented
by placing the first transformation above the second.
Vertical composition is represented by sharing a common domain/codomain.

                t_id
       t1    f1 ------> f1
  r1 ------> --         --
                        f2
             s1 ------> --       s1 = r2
                 t2    s2
*)
      value m_Pr = fun ~lr ~i -> (* : S_Pp lr -> s *)
        match SignIListOps.find_ok ~i ~l:lr with
        [‘OK s ->
            match SemWSign.s_Pp ~ls:lr with
            [‘OK r ->
                let lc = Sign.type_part ~s:r in
                let foot_i = FootWSign.footPp ~lc ~i ~s
```

```
              legs = Sign.value_part ~s:r in
            let t = Trans.t_pr ~lf:legs ~i in
            Dule.pack ~f:foot_i ~t ~s .‘OK
        |‘Error er -> er .‘Error]
    |‘Error er -> er .‘Error]
value m_Record = fun ~r ~lm -> (* : r -> S_Pp ls *)
  let lm_ind = DuleIList.t2ind ~it:lm in
  let lf = FunctIListOps.ind2t ~it:
             match lm_ind with
             map m -> Dule.type_part ~m (* : r -> s_i *)
      lt = TransIListOps.ind2t ~it:
             match lm_ind with
             map m -> Dule.value_part ~m
      ls = SignIListOps.ind2t ~it:
             match lm_ind with
             map m -> Dule.codomain ~m
  in
  match OkWDule.typesRecord ~lf ~ls with
  [‘OK lf ->
     let g = Sign.s2f ~s:r in
     let c = SrcCore.src ~f:g in
     let f = Funct.f_RECORD ~c ~lf
         t = Trans.t_record ~f:g ~lt in
     match SemWSign.s_Pp ~ls with
     [‘OK s ->
         Dule.pack ~f ~t ~s .‘OK
     |‘Error er -> er .‘Error]
  |‘Error er -> er .‘Error]
value m_Base = fun ~r ~s ~lg ~lt -> (* : r -> s *)
  match OkWDule.typesBase ~r ~s ~lg with
  [‘OK f ->
      let g = Sign.s2f ~s:r in
      let lgt = MapFT.vmap_tf ~f:(fun ~v ->
        DomCore.dom ~t:v) ~l:lt in
      match FunctIListOps.vforall ~p:EqFunct.eq(~v:g) ~l:lgt with
      [‘True ->
          let t = Trans.t_record ~f:g ~lt in
          let ts = DomCore.cod ~t in
          let h = Sign.s2f ~s in
          let fs = Funct.f_COMP ~f1:f ~f2:h in
          match EqFunct.eq ~it:ts ~v:fs with
          [‘True ->
              Dule.pack ~f ~t ~s .‘OK
          |‘False -> ‘Error]
      |‘False -> ‘Error]
  |‘Error er -> er .‘Error]
value m_Inst = fun ~m1 ~m2 -> (* : r1 -> S_Ww (m1, s2) *)
  let f1 = Dule.type_part ~m:m1 (* : r1 -> s1=r2 *)
      t1 = Dule.value_part ~m:m1
      f2 = Dule.type_part ~m:m2 (* : s1=r2 -> s2 *)
```

```
            t2 = Dule.value_part ~m:m2
            s2 = Dule.codomain ~m:m2 in
        match OkWDule.typesInst ~f1 ~f2 ~s2 with
        [‘OK f ->
           let it2 = Trans.t_FT ~f1 ~t2 in
           let t = Trans.t_comp ~t1 ~t2:it2 in
           match SemWSign.s_Ww ~m1 ~s2 with
           [‘OK s ->
               Dule.pack ~f ~t ~s .‘OK
           |‘Error er -> er .‘Error]
        |‘Error er -> er .‘Error]
      value m_Trim = fun ~m1 ~r2 -> (* : r1 -> r2 *)
        let f1 = Dule.type_part ~m:m1 (* : r1 -> s1 *)
            t1 = Dule.value_part ~m:m1
            g2 = Sign.s2f ~s:r2 in
        let e1 = SrcCore.trg ~f:f1 (* [src s1] *)
            c2 = SrcCore.src ~f:g2 in
        match OkWDule.typesTrim ~e:e1 ~c:c2 with
        [‘OK scf ->
           let f = Funct.f_COMP ~f1 ~f2:scf in
           let fcr2 = Funct.f_COMP ~f1:f ~f2:g2
               f1s1 = DomCore.cod ~t:t1 in (* [f_COMP f1 s1] *)
           match OkWDule.valuesTrim ~h:f1s1 ~f:fcr2 with
           [‘OK sct ->
              let t = Trans.t_comp ~t1 ~t2:sct in
              (Dule.pack ~f ~t ~s:r2) .‘OK
           |‘Error er -> er .‘Error]
        |‘Error er -> er .‘Error]
    end
end

(* LDule is a module system for grouping of modules,
   implemented using solely SemWSign and SemWDule *)
  (* arguments from siblings (and with weaker specifications): *)
  spec SemWSign_for_LDule =
  ~SignIList ->
    sig
      value s_Pp : ~ls:SignIList.t -> [‘OK Sign.t|‘Error]
    end
  SemWSign_for_LDule = WDule | SemWSign :> SemWSign_for_LDule
  spec SemWDule_for_LDule =
  ~SignIList ~DuleIList ->
    sig
      value m_Comp : ~m1:Dule.t ~m2:Dule.t -> Dule.t
      value m_Pr : ~lr:SignIList.t ~i:IdIndex.t ->
        [‘OK Dule.t|‘Error]
      value m_Record : ~r:Sign.t ~lm:DuleIList.t ->
        [‘OK Dule.t|‘Error]
    end
  SemWDule_for_LDule = WDule | SemWDule :> SemWDule_for_LDule
  LDule =
link
  spec SemWSign = SemWSign_for_LDule
  SemWSign = SemWSign_for_LDule
  spec SemWDule = SemWDule_for_LDule
  SemWDule = SemWDule_for_LDule
  (* results: *)
  spec SemLDule =
  ~SignIList ~DuleIList ->
    sig
      value m_Accord : ~lr:SignIList.t ~lm:DuleIList.t ->
        [‘OK Dule.t|‘Error]
      value m_Concord : ~lr:SignIList.t ~lm:DuleIList.t ->
        [‘OK Dule.t|‘Error]
      value m_Link : ~lr:SignIList.t ~lm:DuleIList.t ->
        [‘OK Dule.t|‘Error]
      value m_Link_ordered : ~lr:SignIList.t ~lm:DuleIList.t ->
        [‘OK Dule.t|‘Error]
    end
  SemLDule =
    :: ~SemFunct ~SemWSign ~SemWDule ~DuleIListOps ~MapSM -> SemLDule
    struct
      value m_Accord = fun ~lr ~lm -> (* : S_Pp lr -> S_Pp ls *)
        match SemWSign.s_Pp ~ls:lr with
        [‘OK r ->
            match MapSM.imap1ok_sm ~f:SemWDule.m_Pr(~lr) ~l:lr with
            [‘OK lpr ->
                let prm = fun ~v:m ->
                  let lf = Sign.value_part ~s:(Dule.domain ~m) in
                  let lmf = MapSM.imap_fm
                        ~f:DuleIListOps.find(~l:lpr) ~l:lf in
                  match SemWDule.m_Record ~r ~lm:lmf with
                  [‘OK re ->
                      (SemWDule.m_Comp ~m1:re ~m2:m) .‘OK
                  |‘Error er -> er .‘Error]
                in
                match DuleIListOps.vmap1ok ~f:prm ~l:lm with
                [‘OK lm ->
                    SemWDule.m_Record ~r ~lm
                |‘Error er -> er .‘Error]
             |‘Error er -> er .‘Error]
        |‘Error er -> er .‘Error]
      value m_Concord = fun ~lr ~lm ->
               (*: S_Pp lr -> S_Pp (ls @@ diff lr ls)*)
        match SemWSign.s_Pp ~ls:lr with
        [‘OK r ->
            match MapSM.imap1ok_sm ~f:SemWDule.m_Pr(~lr) ~l:lr with
            [‘OK lpr ->
                let prm = fun ~v:m ->
```

```
                  let lf = Sign.value_part ~s:(Dule.domain ~m) in
                  let lmf = MapSM.imap_fm
                        ~f:DuleIListOps.find(~l:lpr) ~l:lf in
                  match SemWDule.m_Record ~r ~lm:lmf with
                  [`OK re ->
                      (SemWDule.m_Comp ~m1:re ~m2:m) .`OK
                  |`Error er -> er .`Error]
                in
                match DuleIListOps.vmap1ok ~f:prm ~l:lm with
                [`OK lm ->
                    let lm = DuleIListOps.append
                      ~l1:lm ~l2:(DuleIListOps.subtract ~l1:lpr ~l2:lm) in
                    SemWDule.m_Record ~r ~lm
                |`Error er -> er .`Error]
             |`Error er -> er .`Error]
        |`Error er -> er .`Error]
      (* here order of lm doesn't matter,
         but no circularity allowed: *)
      value m_Link = fun ~lr ~lm -> (* : S_Pp lr -> S_Pp ls *)
        match SemWSign.s_Pp ~ls:lr with
        [`OK r ->
            match MapSM.imap1ok_sm ~f:SemWDule.m_Pr(~lr) ~l:lr with
            [`OK lpr ->
                let rec rlink = fun ~i ->
                  match DuleIListOps.find_ok ~i ~l:lpr with
                  [`OK pr -> pr .`OK
                  |`Error er ->
                      let m = DuleIListOps.find ~i ~l:lm in
                      let lf = Sign.value_part ~s:(Dule.domain ~m) in
                      match MapSM.imap1ok_fm ~f:rlink ~l:lf with
                      [`OK lmf ->
                          match SemWDule.m_Record ~r ~lm:lmf with
                          [`OK re ->
                              (SemWDule.m_Comp ~m1:re ~m2:m) .`OK
                          |`Error er -> er .`Error]
                      |`Error er -> er .`Error]]
                in
                match DuleIListOps.imap1ok ~f:rlink ~l:lm with
                [`OK lm ->
                    SemWDule.m_Record ~r ~lm
                |`Error er -> er .`Error]
            |`Error er -> er .`Error]
        |`Error er -> er .`Error]
      (* here we assume a module depends
         only on the previous ones in lm: *)
      value m_Link_ordered = fun ~lr ~lm -> (* : S_Pp lr -> S_Pp ls *)
        match SemWSign.s_Pp ~ls:lr with
        [`OK r ->
            match MapSM.imap1ok_sm ~f:SemWDule.m_Pr(~lr) ~l:lr with
            [`OK lpr ->
```

```
                let pro = fun ~i ~v:m ~l ->
                  let lf = Sign.value_part ~s:(Dule.domain ~m) in
                  match MapSM.imap1ok_fm
                    ~f:DuleIListOps.find_ok(~l) ~l:lf
                  with
                  [`OK lmf ->
                      match SemWDule.m_Record ~r ~lm:lmf with
                      [`OK re ->
                          let m = SemWDule.m_Comp ~m1:re ~m2:m in
                          (DuleIList.cons ~i ~v:m ~l) .`OK
                      |`Error er -> er .`Error]
                  |`Error er -> er .`Error]
                in
                match DuleIListOps.bfold1ok ~init:lpr ~f:pro ~l:lm  with
                [`OK lprlm ->
                    let lm = DuleIListOps.subtract ~l1:lprlm ~l2:lpr in
                    SemWDule.m_Record ~r ~lm
                |`Error er -> er .`Error]
            |`Error er -> er .`Error]
        |`Error er -> er .`Error]
  end
end
```

C.2.4 Implementation of I-Dule

Second of the longer examples implements the I-Dule extension to the module system, almost literally reimplementing the definitions of the operations given in Section 9.1.2. Notice that no inductive module constructions are used in the implementation itself.

The first thirty-line piece of code defines the narrow interface between W-Dule and L-Dule on one hand and I-Dule on the other hand. First, there are some modified specifications of W-Dule and L-Dule that describe the major arguments to the module implementing the I-Dule constructions. Notice that in specification `SemWSign_for_IDule` only the operation `s_Pp` is required to be present — no other operations of `SemWSign` are needed for the construction of inductive modules. The actual arguments are explicitly trimmed to the new specifications and, in the first few lines of the linking expression, the original versions are overwritten. Within the scope of the linking expression the old names will refer to the new trimmed versions.

This modular idiom for interfacing large module collections has many alternatives, some of them written with libraries and implicit trimming (using module instantiation) others with nested linking expressions. The form used in the example emphasizes equal status and separate compilation of the few main components of the compiler and subordinate character of the other local and global auxiliary modules.

```
(* IDule is enriched with inductive modules: *)
  (* arguments from siblings *)
  spec SemWSign_for_IDule =
  ~SignIList ->
    sig
      value s_Pp : ~ls:SignIList.t -> [‘OK Sign.t|‘Error]
    end
  SemWSign_for_IDule = WDule | SemWSign :> SemWSign_for_IDule
  spec SemWDule_for_IDule =
  ~Trans ~SignIList ~DuleIList ->
    sig
      value m_Id : ~s:Sign.t -> Dule.t
      value m_Comp : ~m1:Dule.t ~m2:Dule.t -> Dule.t
      value m_Pr : ~lr:SignIList.t ~i:IdIndex.t ->
        [‘OK Dule.t|‘Error]
      value m_Record : ~r:Sign.t ~lm:DuleIList.t ->
        [‘OK Dule.t|‘Error]
      value m_Base : ~r:Sign.t ~s:Sign.t
                     ~lg:FunctIList.t ~lt:TransIList.t ->
                       [‘OK Dule.t|‘Error]
      value m_Inst : ~m1:Dule.t ~m2:Dule.t -> [‘OK Dule.t|‘Error]
      value m_Trim : ~m1:Dule.t ~r2:Sign.t -> [‘OK Dule.t|‘Error]
    end
  SemWDule_for_IDule = WDule | SemWDule :> SemWDule_for_IDule

  IDule =
link
  spec SemWSign = SemWSign_for_IDule
  SemWSign = SemWSign_for_IDule
  spec SemWDule = SemWDule_for_IDule
  SemWDule = SemWDule_for_IDule
  (* results: *)
  spec XToolIDule =
  ~AtIndex ~Trans ->
    sig
      value repair :
        ~unii:(~f:Funct.t -> Funct.t)
        ~t_con:(~g:Funct.t -> Trans.t)
        ~t_de:(~g:Funct.t -> Trans.t) ->
          ~lrta:TransIList.t ~lbpr:FunctIList.t
          ~fi:Funct.t ~h:Funct.t ~a:Cat.t ->
            Trans.t
      value close_type :
        ~f_ii:(~g:Funct.t -> Funct.t)
        ~coi:(~atj:Cat.t ~atk:Cat.t -> CatIList.t) ->
          ~f:Funct.t ~b:Cat.t ~la:CatIList.t ->
            Funct.t
    end
  module XToolIDule =
    :: ~FunctIListOps ~TransIListOps ~MapCF ~MapFT -> XToolIDule
    struct
      value repair =
        fun ~unii ~t_con ~t_de ->
          fun ~lrta ~lbpr ~fi ~h ~a ->
            let unf = unii ~f:fi in (* : c_PP (coi b a) -> b *)
            let fcon = t_con ~g:unf in
            let pifc = MapFT.vmap_ft ~f:(fun ~v:pr ->
              Trans.t_TF ~t1:fcon ~f2:pr) ~l:lbpr in
            let tc = Trans.t_RECORD ~c:a
              ~lt:(TransIListOps.append ~l1:lrta ~l2:pifc) in
            (* idealized: *)
            Trans.t_TF ~t1:tc ~f2:h
      value close_type =
        fun ~f_ii ~coi ->
          fun ~f ~b ~la ->
            (* f : cons (AtIndex.atr, b) la -> b *)
            let lapr = MapCF.imap_cf ~f:Funct.f_PR(~lc:la) ~l:la in
            let a = Cat.c_PP ~lc:la in
            let lba = coi ~atj:b ~atk:a in
            let prba = Funct.f_PR ~lc:lba ~i:AtIndex.atk in
            let lpr = FunctIListOps.vmap ~f:(fun ~v:pr ->
              Funct.f_COMP ~f1:prba ~f2:pr) ~l:lapr in
            let lcopr =
              FunctIList.cons ~i:AtIndex.atr
                              ~v:(Funct.f_PR ~lc:lba ~i:AtIndex.atj)
                              ~l:lpr in
            let rba = Funct.f_RECORD ~c:(Cat.c_PP ~lc:lba) ~lf:lcopr in
            let fca = Funct.f_COMP ~f1:rba ~f2:f in
            f_ii ~g:fca (* : a -> b *)
    end
  spec ToolIDule =
  ~Trans ->
    sig
      value repair_ii :
          ~lrta:TransIList.t ~lbpr:FunctIList.t
          ~fi:Funct.t ~h:Funct.t ~a:Cat.t ->
            Trans.t
      value repair_tt :
          ~lrta:TransIList.t ~lbpr:FunctIList.t
          ~fi:Funct.t ~h:Funct.t ~a:Cat.t ->
            Trans.t
      value close_type_ii :
          ~f:Funct.t ~b:Cat.t ~la:CatIList.t ->
            Funct.t
      value close_type_tt :
          ~f:Funct.t ~b:Cat.t ~la:CatIList.t ->
            Funct.t
      value fix_value :
          ~t:Trans.t ~tb:Funct.t ~lta:FunctIList.t ~a:Cat.t ->
            Trans.t
    end
  module ToolIDule =
    :: ~XToolIDule ~TransIListOps ~SemCat ~SemFunct ~MapFT -> ToolIDule
    struct
      value repair_ii = XToolIDule.repair ~unii:SemFunct.unii
                                          ~t_con:Trans.t_con
                                          ~t_de:Trans.t_de
      value repair_tt = XToolIDule.repair ~unii:SemFunct.untt
                                          ~t_con:Trans.t_uncon
                                          ~t_de:Trans.t_unde
      value close_type_ii =
        XToolIDule.close_type ~f_ii:Funct.f_ii ~coi:SemCat.coi
      value close_type_tt =
        XToolIDule.close_type ~f_ii:Funct.f_tt ~coi:SemCat.coi
      value fix_value = fun ~t ~tb ~lta ~a ->
        (* t : cons (AtIndex.atr, tb) lta -> tb *)
        let ltapr = MapFT.imap_ft ~f:Trans.t_pr(~lf:lta) ~l:lta in
        let ta = Funct.f_pp ~c:a ~lf:lta in
        let lba = SemFunct.cof ~atd:tb ~ate:ta in
        let prba = Trans.t_pr ~lf:lba ~i:AtIndex.ate in
        let lpr = TransIListOps.vmap ~f:(fun ~v:pr ->
          Trans.t_comp ~t1:prba ~t2:pr) ~l:ltapr in
        let lcopr =
          TransIList.cons ~i:AtIndex.atr
                          ~v:(Trans.t_pr ~lf:lba ~i:AtIndex.atd)
                          ~l:lpr in
        let rba = Trans.t_record ~f:(Funct.f_pp ~c:a ~lf:lba) ~lt:lcopr in
        let fca = Trans.t_comp ~t1:rba ~t2:t in
        Trans.tl_fix ~t:fca (* : ta -> tb *)
    end
  spec XOrdinaryIDule =
  ~Trans ~Dule ->
    sig
      value m_XInd_ordinary :
        ~repair:(~lrta:TransIList.t ~lbpr:FunctIList.t
                 ~fi:Funct.t ~h:Funct.t ~a:Cat.t ->
                   Trans.t)
        ~close_type:(~f:Funct.t ~b:Cat.t ~la:CatIList.t ->
                     Funct.t)
        ~fix_value:(~t:Trans.t ~tb:Funct.t ~lta:FunctIList.t ~a:Cat.t ->
                      Trans.t) ->
          ~m:Dule.t ->
            Dule.t
    end
  module XOrdinaryIDule =
    :: ~SrcCore ~DomCore ~SemCat ~SemFunct
       ~CatIListOps ~FunctIListOps ~TransIListOps
       ~MapCF ~MapFT -> XOrdinaryIDule
    struct
      value m_XInd_ordinary = (* : S_Pp lr -> s' *)
```

```
fun ~repair ~close_type ~fix_value ->
  fun ~m ->
(* m : S_Pp lrr -> s,
   lrr = cons (AtIndex.atr, s) lr,
   [AtIndex.atr] not in [lr] nor [lb] (local types of [s]),
   labels of [lr] and [lb] are disjoint,
   labels of context types of [r_i] are in [lr]
   labels of context types of [s] are the labels of [lrr],
   labels of context types of [s'] are the labels of [lr],
   [s'] depends on its own local types instead of on [AtIndex.atr]
 *)
let s = Dule.codomain ~m in
let h = Sign.s2f ~s in
let f = Dule.type_part ~m in (* : c -> e *)
let t = Dule.value_part ~m in (* : r -> f_COMP f h, r = S_Pp lrr *)
(* analizing f:*)
let c = SrcCore.src ~f in (* = src (S_Pp lrr) *)
let lc = SemCat.unPP ~c in (* = cons (AtIndex.atr, b) la *)
let e = SrcCore.trg ~f in (* = src s *)
let le = SemCat.unPP ~c:e in (* = lb @@ lc *)
let b = CatIListOps.find ~i:AtIndex.atr ~l:lc in
let lb = SemCat.unPP ~c:b in
  (* normally, these have labels of [ls] *)
let la = CatIListOps.remove ~i:AtIndex.atr ~l:lc in
  (* [la] labels = [lr] labels *)
let a = Cat.c_PP ~lc:la in
(* cutting f: *)
let pib = MapCF.imap_cf ~f:Funct.f_PR(~lc:le) ~l:lb in
let fc = Funct.f_COMP ~f1:f ~f2:(Funct.f_RECORD ~c:e ~lf:pib) in
  (* : c -> b *)
(* f_ii f: *)
let fi = close_type ~f:fc ~b ~la in (* : a -> b *)
(* instantiating t: *)
let lapr = MapCF.imap_cf ~f:Funct.f_PR(~lc:la) ~l:la in
let lrfa = FunctIList.cons ~i:AtIndex.atr ~v:fi ~l:lapr in
let rf = Funct.f_RECORD ~c:a ~lf:lrfa in (* : a -> c *)
let rft = Trans.t_FT ~f1:rf ~t2:t in
  (* : f_COMP rf r -> f_COMP rf (f_COMP f h) *)
(* repairing t: *)
let lrta = MapFT.vmap_ft ~f:(fun ~v:f -> Trans.t_id ~f) ~l:lrfa in
let lbpr = MapCF.imap_cf ~f:Funct.f_PR(~lc:lb) ~l:lb in
let adc = repair ~lrta ~lbpr ~fi ~h ~a in
let tad = Trans.t_comp ~t1:rft ~t2:adc in
  (* : g -> tb = f_COMP fif h *)
(* analizing t: *)
let g = DomCore.dom ~t:tad in (* = f_COMP rf r *)
let lg = SemFunct.unpp ~f:g in (* = cons (AtIndex.atr, tb) lta *)
let tb = FunctIListOps.find ~i:AtIndex.atr ~l:lg in
  (* normally, the [ls] labels *)
let lta = FunctIListOps.remove ~i:AtIndex.atr ~l:lg in
```

```
        (* [lta] labels = [lr] labels *)
      (* tl_fix t (analogous to f_ii f):*)
      let t' = fix_value ~t:tad ~tb ~lta ~a in (* : ta -> tb *)
      (* changing s to s': *)
      let ld = CatIListOps.remove ~i:AtIndex.atr ~l:le in
      let d = Cat.c_PP ~lc:ld in
      let pid = MapCF.imap_cf ~f:Funct.f_PR(~lc:ld) ~l:ld in
      let pib = MapCF.imap_cf ~f:FunctIListOps.find(~l:pid) ~l:lb in
      let reb = Funct.f_RECORD ~c:d ~lf:pib in
      let pir = FunctIList.cons ~i:AtIndex.atr ~v:reb ~l:pid in
      let red = Funct.f_RECORD ~c:d ~lf:pir in (* d -> e *)
      let h' = Funct.f_COMP ~f1:red ~f2:h in
      let s' = Sign.f2s ~f:h' in
      (* extending f to s': *)
      let pifb = FunctIListOps.vmap ~f:(fun ~v:pr ->
        Funct.f_COMP ~f1:fi ~f2:pr) ~l:lbpr in
      let f' = Funct.f_RECORD ~c:a
        ~lf:(FunctIListOps.append ~l1:pifb ~l2:lapr) in (* : a -> d *)
      Dule.pack ~f:f' ~t:t' ~s:s'
  end
spec OrdinaryIDule =
~Dule ->
  sig
    value m_Ind_ordinary : ~m:Dule.t -> Dule.t
    value m_CoInd_ordinary : ~m:Dule.t -> Dule.t
  end
module OrdinaryIDule =
  :: ~XOrdinaryIDule ~ToolIDule -> OrdinaryIDule
  struct
    value m_Ind_ordinary = XOrdinaryIDule.m_XInd_ordinary
      ~repair:ToolIDule.repair_ii
      ~close_type:ToolIDule.close_type_ii
      ~fix_value:ToolIDule.fix_value
    value m_CoInd_ordinary = XOrdinaryIDule.m_XInd_ordinary
      ~repair:ToolIDule.repair_tt
      ~close_type:ToolIDule.close_type_tt
      ~fix_value:ToolIDule.fix_value
  end
spec Ripcord =
~AtIndex ~SignIList ~DuleIList ->
  sig
    value m_Ripcord : ~lrr:SignIList.t ~lm:DuleIList.t ->
      [‘OK Dule.t|‘Error]
  end
Ripcord =
  :: ~SemWSign ~SemWDule ~SignIListOps ~DuleIListOps ~MapSM -> Ripcord
  struct
    value m_Ripcord = fun ~lrr ~lm -> (* : S_Pp lrr -> S_Pp ls' *)
      (* m_i : S_Pp lr_i -> s_i,
         [lr_i} contained in [lr @@ ls],
        lrr = cons (AtIndex.atr, S_Pp ls) lr
        [s_i] has labels of context types inside [lr @@ ls],
        [AtIndex.atr] not in [ls], too,
        labels of context types of [r_i] are in [lr]
        s'_i = S_Ww (re_i, s_i),
        where domain of each [re_i] is [S_Pp lr],
        so the labels of context types of [S_Pp ls']
        are exactly the labels of [lrr]
       *)
      match SemWSign.s_Pp ~ls:lrr with
      [`OK rr ->
          match MapSM.imap1ok_sm ~f:SemWDule.m_Pr(~lr:lrr) ~l:lrr with
          [`OK lprr ->
              let lm_ind = DuleIList.t2ind ~it:lm in
              let ls =
                SignIListOps.ind2t ~it:
                  match lm_ind with
                  map m -> Dule.codomain ~m
              in
              match MapSM.imap1ok_sm ~f:SemWDule.m_Pr(~lr:ls) ~l:ls with
              [`OK lps ->
                  let prr = DuleIListOps.find ~i:AtIndex.atr ~l:lprr in
                  let lpc = DuleIListOps.vmap ~f:(fun ~v:pr ->
                    SemWDule.m_Comp ~m1:prr ~m2:pr) ~l:lps in
                  let lpr = DuleIListOps.remove ~i:AtIndex.atr ~l:lprr in
                  let lprs = DuleIListOps.append ~l1:lpc ~l2:lpr in
                  let prm = fun ~v:m ->
                    let lf = Sign.value_part ~s:(Dule.domain ~m) in
                    let lmf = MapSM.imap_fm
                      ~f:DuleIListOps.find(~l:lprs) ~l:lf in
                    match SemWDule.m_Record ~r:rr ~lm:lmf with
                    [`OK re ->
                        SemWDule.m_Inst ~m1:re ~m2:m
                    |`Error er -> er .`Error]
                  in
                  match DuleIListOps.vmap1ok ~f:prm ~l:lm with
                  [`OK lm ->
                    SemWDule.m_Record ~r:rr ~lm
                  |`Error er -> er .`Error]
              |`Error er -> er .`Error]
          |`Error er -> er .`Error]
      |`Error er -> er .`Error]
  end
spec XSemIDule =
~SignIList ~DuleIList ->
  sig
    value m_XInd : ~m_Ind_ordinary:(~m:Dule.t -> Dule.t) ->
      ~lr:SignIList.t ~lm:DuleIList.t ->
        [`OK Dule.t|`Error]
  end
  XSemIDule =
    :: ~AtIndex ~SemWSign ~SemWDule ~SignIListOps ~DuleIListOps
       ~Ripcord -> XSemIDule
    struct
      value m_XInd = (* : S_Pp lr -> S_Pp ls *)
        fun ~m_Ind_ordinary ->
          fun ~lr ~lm ->
        (* m_i : S_Pp lr_i -> s_i,
          [lr_i} contained in [lr @@ ls],
          [s_i] has labels of context types inside [lr @@ ls],
          [AtIndex.atr] not in [lr] nor [ls],
          labels of context types of [r_i] are in [lr]
         *)
        let lm_ind = DuleIList.t2ind ~it:lm in
        let ls =
          SignIListOps.ind2t ~it:
            match lm_ind with
            map m -> Dule.codomain ~m
        in
        match SemWSign.s_Pp ~ls with
        [`OK s ->
            let lrr = SignIList.cons ~i:AtIndex.atr ~v:s ~l:lr in
            match Ripcord.m_Ripcord
              ~lrr ~lm with (* : S_ssr -> S_Pp ls' *)
            [`OK m ->
                let mind =
                  m_Ind_ordinary ~m in (* : S_Pp lr -> S_Pp ls'' *)
                SemWDule.m_Trim ~m1:mind ~r2:s
            |`Error er -> er .`Error]
        |`Error er -> er .`Error]
    end
  spec SemIDule =
  ~SignIList ~DuleIList ->
    sig
      value m_Ind : ~lr:SignIList.t ~lm:DuleIList.t ->
        [`OK Dule.t|`Error]
      value m_CoInd : ~lr:SignIList.t ~lm:DuleIList.t ->
        [`OK Dule.t|`Error]
    end
  SemIDule =
    :: ~XSemIDule ~OrdinaryIDule -> SemIDule
    struct
      value m_Ind =
        XSemIDule.m_XInd ~m_Ind_ordinary:OrdinaryIDule.m_Ind_ordinary
      value m_CoInd =
        XSemIDule.m_XInd ~m_Ind_ordinary:OrdinaryIDule.m_CoInd_ordinary
    end
end
```

Bibliography

[1] D. Ancona and E. Zucca. A Primitive Calculus for Module Systems. In G. Nadathur, editor, *PPDP'99 - Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 62–79. Springer, 1999.

[2] D. Ancona and E. Zucca. A Calculus of Module Systems. *Journ. of Functional Programming*, 12(2):91–132, March 2002.

[3] D. Ancona and E. Zucca. A Theory of Mixin Modules: Algebraic Laws and Reduction Semantics. *Mathematical Structures in Computer Science*, 12, 2002.

[4] Andrew W. Appel. A Critique of Standard ML. *Journal of Functional Programming*, 3(4):391–429, October 1993.

[5] Egidio Astesiano, Michel Bidoit, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[6] Lennart Augustsson. Cayenne—A language with dependent types. In *ACM SIGPLAN International Conference on Functional Programming (ICFP) , Baltimore, Maryland*, pages 239–250, 1998.

[7] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming –An Introduction–. In S. Doaitse Swierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Revised Lectures from 3rd Int. School on Advanced Functional Programming, AFP'98, Braga, Portugal, 12–19 Sept. 1998*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, Berlin, 1999.

[8] J. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Processes. *Communications of the ACM*, 21(8):613–641, August 1978.

[9] Anya Helene Bagge, Martin Bravenboer, Karl Trygve Kalleberg, Koen Muilwijk, and Eelco Visser. Adaptive Code Reuse by Aspects, Cloning and Renaming. Technical Report UU-CS-2005-031, Institute of Information and Computing Sciences, Utrecht University, 2005.

[10] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, January 1990.

[11] Henk P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland, Amsterdam, second edition, 1984.

[12] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice Hall International Series in Computer Science. Prentice Hall, second edition, 1995.

[13] Hubert Baumeister, Maura Cerioli, Anne Haxthausen, Till Mossakowski, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL semantics. In CASL Reference Manual [28], part III. Edited by D. Sannella and A. Tarlecki.

[14] Jeffrey M. Bell, Francoise Bellegarde, and James Hook. Type-driven Defunctionalization. Technical Report CSE-96-009, Oregon Graduate Institute of Science and Technology, November 20, 1996.

[15] Dave Berry. Lessons from the design of a Standard ML library. *Journal of Functional Programming*, 3(4):527–552, October 1993.

[16] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. Lecture Notes in Computer Science 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[17] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273, 2002.

[18] P. Burmeister. *A Model Theoretic Oriented Approach to Partial Algebras*. Mathematical Research 31. Akademie-Verlag, Berlin, 1986.

[19] Gian Luca Cattani. Presheaf Models for Concurrency (Unrevised). Dissertation Series DS-99-1, BRICS, Department of Computer Science, University of Aarhus, April 1999. PhD thesis.

[20] Robin Cockett. Introduction to Distributive Categories. *Math. Struct. in Comp. Science*, pages 1–20, 1991.

[21] Robin Cockett. Distributive Logic. `ftp://ftp.cpsc.ucalgary.ca/pub/projects/charity/literature/papers_and_reports/DistLogic.ps`, 1992.

[22] Robin Cockett. Charitable Thoughts, 1996. (draft lecture notes, `http://pll.cpsc.ucalgary.ca/charity1/www/home.html`).

[23] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[24] Robin Cockett and Dwight Spencer. Strong Categorical Datatypes I. In R. A. G. Seely, editor, *Proc. of Int. Summer Category Theory Meeting, Montréal, Québec, 23–30 June 1991*, volume 13 of *Canadian Mathematical Society Conf. Proceedings*, pages 141–169. American Mathematical Society, Providence, RI, 1992.

[25] Robin Cockett and Dwight Spencer. Strong Categorical Datatypes II. A term logic for categorical programming. *Theoretical Computer Science*, 139(1–2):69–113, March 1995.

[26] CoFI Language Design Group. CASL Summary. In CASL Reference Manual [28], part I. Edited by B. Krieg-Brückner and P. D. Mosses.

[27] CoFI Language Design Group. CASL Syntax. In CASL Reference Manual [28], part II. Edited by B. Krieg-Brückner and P. D.Mosses.

[28] CoFI (The Common Framework Initiative). CASL *Reference Manual*. Lecture Notes in Computer Science 2960 (IFIP Series). Springer-Verlag, 2004.

[29] Andrea Corradini and Fabio Gadducci. CPO Models for Infinite Term Rewriting. In V. S. Alagar and M. Nivat, editors, *Algebraic Methodology and Software Technology, 4th International Conference, AMAST'95, Montreal, Canada, July 3–7, 1995, Proceedings*, volume 936 of *Lecture Notes in Computer Science*, pages 368–384. Springer-Verlag, 1995.

[30] R. Di Cosmo and D. Kesner. A confluent reduction for the extensional typed λ–calculus with pairs, sums, recursion and terminal object. In Andrzej Lingas, Rolf Karlsson, and Svante Carlsson, editors, *Proceedings of International Conference on Automata, Languages and Programming (ICALP '91)*, volume 700 of *Lecture Notes in Computer Science*, pages 645–656, Berlin, Germany, July 1993. Springer.

[31] G. Cousineau, P. L. Curien, and M. Mauny. The Categorial Abstract Machine. *Science of Computer Programming*, 8:173–202, 1987.

[32] Karl Crary, Robert Harper, and Sidd Puri. What is a Recursive Module? In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, 1999.

[33] Tristan Crolard. Subtractive logic. *Theoretical Computer Science*, 254(1–2):151–185, 2001.

[34] P.-L. Curien. Categorical Combinators. *Information and Control*, 69(1-3):189–254, 1986.

[35] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Progress in Theoretical Computer Science. Birkhäuser, Boston, 2nd edition, 1993. (1st ed., Pitman Publishing, London, and J. Wiley and Sons, New York).

[36] P.-L. Curien and R. Di Cosmo. A confluent reduction for the λ-calculus with surjective pairing and terminal object. *Journal of Functional Programming*, 6(2):299–327, March 1996.

[37] Daniel de Rauglaudre. Camlp4 — Reference Manual, 2003. `http://caml.inria.fr/camlp4/manual`.

[38] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

[39] Daniel J. Dougherty. Some lambda calculi with categorical sums and products. In Claude Kirchner, editor, *Proceedings of the 5th International Conference on Rewriting Techniques and Applications (RTA-93)*, volume 690 of *Lecture Notes in Computer Science*, pages 137–151, Berlin, June 16–18 1993. Springer-Verlag.

[40] Derek Dreyer. Understanding and Evolving the ML Module System. PhD thesis, CMU Technical Report CMU-CS-05-131, May 2005.

[41] Derek Dreyer, Karl Crary, and Robert Harper. A Type System for Higher-Order Modules. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, pages 236–249, New Orleans, January 2003.

[42] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation.* PhD thesis, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, December 1998. Available as Tech. Report 99/3.

[43] Martin Elsman. Static Interpretation of Modules. In *International Conference on Functional Programming*, pages 208–219, 1999.

[44] Martin Erwig. Categorical programming with abstract data types. In Armando Martin Haeberer, editor, *AMAST*, volume 1548 of *Lecture Notes in Computer Science*, pages 406–421. Springer, 1998.

[45] Leonidas Fegaras and Tim Sheard. Revisiting Catamorphisms over Datatypes with Embedded Functions (or, Programs from Outer Space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

[46] Andrzej Filinski. Declarative Continuations and Categorical Duality. Technical Report 89/11, DIKU, University of Copenhagen, Denmark, 1989. Masters Thesis.

[47] Robert Bruce Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. *ACM SIGPLAN Notices*, 34(1):94–104, January 1999.

[48] Marcelo P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps.* PhD thesis, University of Edinburgh, 1994.

[49] M. M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics.* University of Utrecht, 1992.

[50] M. M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[51] M. M. Fokkinga and L. Meertens. Adjunctions. Memoranda Informatica, University of Twente, June 1994.

[52] P. Freyd. Structural Polymorphism. *Theoretical Computer Science*, 115(1):107–129, July 1993.

[53] Peter J. Freyd. Algebraically complete categories. In A. Carboni, M. C. Pedicchio, and G. Rosolini, editors, *Proc. of Int. Conf. Category Theory '90, CT'90, Como, Italy, 22–28 July 1990*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer-Verlag, Berlin, 1991.

[54] Tom Fukushima and Charles Tuckey. *Charity User Manual*, January 1996. (draft, `http://pll.cpsc.ucalgary.ca/charity1/www/home.html`).

[55] Jacques Garrigue. Labeled and optional arguments for Objective Caml. In JSSST Workshop on Programming and Programming Languages, Kameoka, Japan, March 2001.

[56] Jaques Garrigue and Hassan Aït-Kaci. The Typed Polymorphic Label-Selective λ-Calculus. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 35–47, 1994.

[57] Neil Ghani. *Adjoint Rewriting.* PhD thesis, University of Edinburgh, 1995.

[58] Neil Ghani. Beta-Eta-Equality for Coproducts. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications: Proc. of the 2nd International Conference on Typed Lambda Calculi and Applications (TLCA-95)*, pages 171–185. Springer, Berlin, Heidelberg, 1995.

[59] Neil Ghani, Valeria de Paiva, and Eike Ritter. Categorical Models of Explicit Substitutions. In *Foundations of Software Science and Computation Structure*, pages 197–211, 1999.

[60] Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. When is a Function a Fold or an Unfold? In *Proceedings of the 4th International Workshop on Coalgebraic Methods in Computer Science*, volume 44.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, April 2001.

[61] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE (system description). In V. van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA-04*, Lecture Notes in Computer Science 3091, pages 210–220, Valencia, Spain, June 3-5, 2004. Springer.

[62] Girard, J.-Y., Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1989.

[63] Neal Glew and Greg Morrisett. Type-Safe Linking and Modular Assembly Language. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 250–261, New York, NY, 1999.

[64] Joseph A. Goguen. A Categorical Manifesto. Technical Monograph PRG-72, Oxford University Computing Laboratory, Programming Research Group, March 1989.

[65] Joseph A. Goguen and Will Tracz. An Implementation-Oriented Semantics for Module Composition. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 231–263. Cambridge University Press, New York, NY, 2000.

[66] Warren D. Goldfarb. The Undecidability of the Second-Order Unification Problem. *Theoretical Computer Science*, 13:225–230, 1981.

[67] Tatsuya Hagino. *A Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh, Department of Computer Science, 1987. CST-47-87 (also published as ECS-LFCS-87-38).

[68] T. Hardin. Confluence results for the pure strong categorical logic CCL lambda-calculi as subsystems of CCL. *Theoretical Computer Science*, 65(3):291–342, July 1989.

[69] T. Hardin. From Categorical Combinators to Lambda Sigma-Calculi, a Quest for Confluence. Technical Report No. 1777, INRIA, Le Chesnay, France, 1992.

[70] T. Hardin and A. Laville. Proof of termination of the rewriting system SUBST on CCL. *Theoretical Computer Science*, 46(2-3):305–312, 1986.

[71] Robert Harper and Mark Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[72] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-Order Modules and the Phase Distinction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990.

[73] Masahito Hasegawa. Decomposing typed lambda calculus into a couple of categorical programming languages. In David Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Proceedings of the 6th International Conference on Category Theory and Computer Science (CTCS'95)*, volume 953 of *Lecture Notes in Computer Science*, pages 200–219, Berlin, GER, August 1995. Springer.

[74] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Call-by-Value Mixin Modules. In *13th European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 64–78. Springer-Verlag, 2004.

[75] Brian T. Howard. Inductive, Coinductive, and Pointed Types. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 102–109, 1996.

[76] G. P. Huet. Cartesian Closed Categories and Lambda-Calculus. In G. Cousineau, P.-L. Curien, and B. Robinet, editors, *Combinators and Functional Programming Languages*, pages 123–135. Springer-Verlag, Berlin, DE, 1986. Lecture Notes in Computer Science 242.

[77] Graham Hutton. Fold and Unfold for Program Semantics. In *Proc. of 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept. 1998*, volume 34(1) of *SIGPLAN Notices*, pages 280–288. ACM Press, New York, 1998.

[78] Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

[79] Bart Jacobs. *Categorical Type Theory.* PhD thesis, Computing Science Inst., Univ. of Nijmegen, 1991.

[80] Bart Jacobs and Jan Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of the EATCS*, 62:222–259, 1996.

[81] C. B. Jay. Fixpoint and loop constructions as colimits. In M.C. Pedicchio A. Carboni and G. Rosolini, editors, *Proceedings Summer Conference on Category Theory, Como 1990*, volume 1488 of *Lecture Notes in Mathematics*, pages 187–192. Springer Verlag, 1991.

[82] C. B. Jay. An introduction to categories in computing. Technical Report UTS-SOCS-93.9, University of Technology, Sydney, 1993.

[83] C. B. Jay. Tail recursion through universal invariants. *Theoretical Computer Science*, 115:151–189, 1993.

[84] C. B. Jay. Matrices, monads and the fast Fourier transform. In *Proceedings of the Massey Functional Programming Workshop 1994*, pages 71–80, 1994.

[85] C. B. Jay. Data categories. In M.E. Houle and P. Eades, editors, *Computing: The Australasian Theory Symposium Proceedings, Melbourne, Australia, 29–30 January, 1996*, volume 18, pages 21–28. Australian Computer Science Communications, 1996. ISSN 0157–3055.

[86] C. B. Jay. Type-free term reduction for covariant types. Technical Report 96.07, University of Technology, Sydney, 1996.

[87] C. B. Jay. Covariant types. *Theoretical Computer Science*, 185:237–258, 1997.

[88] C. B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[89] C. B. Jay and N. Ghani. The virtues of eta-expansion. *J. of Functional Programming*, 5(2):135–154, 1995.

[90] C. B. Jay, E. Moggi, and G. Bellè. Functors, types and shapes. In R. Backhouse and T. Sheard, editors, *Workshop on Generic Programming: Marstrand, Sweden, 18th June, 1998*, pages 21–40. Chalmers University of Technology, 1998.

[91] Johan Jeuring and Patrik Jansson. Polytypic Programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text from 2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, Berlin, 1996.

[92] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 28 February 1997.

[93] S. Klinger. The Haskell Programmer's Guide to the IO Monad - Don't Panic. Technical Report 05–54, Centre for Telematics and Information Technology (CTIT), December 2005.

[94] Jan Willem Klop. Term Rewriting Systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–117. Oxford University Press, Oxford, 1992.

[95] Mikołaj Konarski. Labeled functions in OCaml — categorically. `http://www.mimuw.edu.pl/~mikon/Dule/download/dule-papers/fooldule.ps`, 1 April 2003.

[96] Mikołaj Konarski. Anonymous (co)inductive types: A way for structured recursion to cohabit with modular abstraction. `http://www.mimuw.edu.pl/~mikon/Dule/download/dule-papers/anonymous.pdf`, 2006.

[97] Mikołaj Konarski et al. Source code of the Dule compiler. `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd`, 2005.

[98] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge Studies in Advanced Mathematics. CUP, 1986.

[99] Slawomir Lasota. Open maps as a bridge between algebraic observational equivalence and bisimilarity. In Francesco Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 285–299. Springer, 1997.

[100] John Launchbury and Tim Sheard. Warm Fusion: Deriving Build-Catas from Recursive Definitions. In *Conf. Record 7th ACM SIGPLAN-SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 314–323. ACM Press, New York, 1995.

[101] F. W. Lawvere. Diagonal arguments and cartesian closed categories. In *Category Theory, Homology Theory and their Applications II*, pages 143–145. Springer Lecture Notes in Mathematics 92, 1969.

[102] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st symp. Principles of Programming Languages*, pages 109–122. ACM press, 1994.

[103] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd symp. Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[104] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[105] Xavier Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

[106] Xavier Leroy et al. Source code of the Ocaml compiler, 2005. `http://caml.inria.fr/ocaml/release.en.html`.

[107] Jacek Leszczyłowski, Andrzej Borzyszkowski, Ryszard Kubiak, and Stefan Sokołowski. Towards a set-theoretic Type Theory. Research note, Inst. of Comp. Sc., Polish Academy of Sciences, Gdańsk, Poland, September 1988.

[108] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *POPL'95, 22nd ACM Symposium on Principles of Programming Languages*, pages 333–343. ACM, 1995.

[109] Kai Lin and Joseph Goguen. Morphisms and Semantics for Higher Order Parameterized Programming. `http://www-cse.ucsd.edu/users/goguen/pps/shom.ps`.

[110] S. MacLane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[111] Dave MacQueen et al. Source code of the Standard ML of New Jersey compiler, 2005. `http://www.smlnj.org//index.html`.

[112] M. Mauny and D. de Rauglaudre. A complete and realistic implementation of quotations for ML. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 94.

[113] Lambert Meertens. Functor Pulling. In Roland Backhouse and Tim Sheard, editors, *Informal Proc. of Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998*. Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ., June 1998.

[114] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In John Hughes, editor, *Proceedings of Functional Programming Languages an Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Berlin, Germany, August 1991. Springer.

[115] Erik Meijer and Graham Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Conf. Record of 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 324–333. ACM Press, New York, 1995.

[116] Erik Meijer and Johan Jeuring. Merging Monads and Folds for Functional Programming. In J. Jeuring and E. Meijer, editors, *Tutorial Text 1st Int. Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, 24–30 May 1995*, volume 925, pages 228–266. Springer-Verlag, Berlin, 1995.

[117] Paul-Andre Mellies and Jerome Vouillon. Recursive polymorphic types and parametricity in an operational framework. In Prakash Panangaden, editor, *Proceedings of the Twentieth Annual IEEE Symp. on Logic in Computer Science, LICS 2005*, pages 82–91. IEEE Computer Society Press, June 2005.

[118] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, Cambridge, MA, 1991.

[119] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1989.

[120] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[121] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, FL, January 1991.

[122] John C. Mitchell and Gordon D. Plotkin. Abstract Types Have Existential Type. *ACM TOPLAS*, 10(3):470–502, July 1988.

[123] Eugenio Moggi. A Category-Theoretic Account of Program Modules. *Mathematical Structures in Computer Science*, 1(1):103–139, March 1991.

[124] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[125] H. Reichel. Initially restricting algebraic theories. In Piotr Dembinski, editor, *Mathematical Foundations of Computer Science*, pages 504–514. Springer, 1980. Lecture Notes in Computer Science, Volume 88.

[126] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Oxford University Press, 1987.

[127] John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156, Berlin, 1984. Springer-Verlag.

[128] Claudio V. Russo. Types for Modules. PhD thesis, Univ. of Edinburgh, 1998.

[129] Claudio V. Russo. Non-dependent Types for Standard ML Modules. In *Principles and Practice of Declarative Programming*, pages 80–97, 1999.

[130] Claudio V. Russo. Recursive structures for Standard ML. *ACM SIGPLAN Notices*, 36(10):50–61, October 2001.

[131] D. E. Rydeheard and R. M. Burstall. *Computational Category Theory.* Prentice-Hall, New York, 1988.

[132] D. Sannella and R. M. Burstall. Structured Theories in LCF. In *Proceedings of the 8th Colloquium on Trees in Algebra and Programming*, pages 377–391. Springer-Verlag, 1983.

[133] D. Sannella and A. Tarlecki. Category theory. In *Foundations of Algebraic Specifications and Formal Program Development*, chapter 3. Cambridge University Press, to appear. See `http://wwwat.mimuw.edu.pl/~tarlecki/book/`.

[134] D. Sannella and A. Tarlecki. Essential Concepts of Algebraic Specification and Program Development. *Formal Aspects of Computing*, 9(3):229–269, 1997.

[135] Marc A. Schroeder. Higher-order Charity. Master's thesis, The University of Calgary, July 1997.

[136] Tim Sheard and Leonidas Fegaras. A Fold for All Seasons. In *Proc. of 6th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pages 233–242. ACM Press, New York, 1993.

[137] Tim Sheard and Emir Pasalic. Two-level types and parameterized modules. *J. Funct. Program*, 14(5):547–587, 2004.

[138] Perdita Stevens. Experiences with the ML Module System, or, Why I Hate ML. Transparencies for a talk given to the Edinburgh ML Club and Glasgow Functional Programming group. `http://www.dcs.ed.ac.uk/home/pxs/talksEtc.html`, 1998.

[139] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[140] Charles Tuckey. Pattern matching in Charity. Master's thesis, The University of Calgary, July 1997.

[141] Peter M. Vesely. Categorical combinators for Charity. Master's thesis, The University of Calgary, November 1996.

[142] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Steve Munchnik, editor, *Proceedings, 14th Symposium on Principles of Programming Languages*, pages 307–312. Association for Computing Machinery, 1987.

[143] Mats Weber. *Proposals for Enhancement for the Ada Programming Language: A Software Engineering Perspective.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 1994.

[144] M. Wirsing. Algebraic Specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 675–788. North-Holland, Amsterdam, 1990.

[145] Niklaus Wirth. *Programming in Modula-2 (3rd corrected edition)*. Springer-Verlag, New York, NY, 1985.

[146] Dale Barry Yee. Implementing the Charity Abstract Machine. Master's thesis, The University of Calgary, September 1995.