A Transformational Approach to Generic Software Development Based on Higher-Order, Typed Functional Signatures

Vom Promotionsausschuss der

Technischen Universität Hamburg-Harburg

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von

Daniel Lincke

aus

Saalfeld

2012

Betreuer:
Prof. Dr. Sibylle Schupp, Institut für Softwaresysteme, Technische Universität Hamburg-Harburg

Zweitgutachter:
Prof. Dr. Friedrich Mayer-Lindenberg, Institut für Rechnertechnologie, Technische Universität Hamburg-Harburg

Drittgutachter:
Prof. Dr. Carlo Jaeger, Global Climate Forum Berlin

Datum der mündlichen Prüfung: 27. Juni 2012

URN: urn:nbn:de:gbv:830-tubdok-11682

Contents

1 Introduction **1**
- 1.1 Related Work . . . 3
- 1.2 Motivating Example . . . 4
- 1.3 Preliminaries and Notation . . . 6
  - 1.3.1 Haskell . . . 6
  - 1.3.2 Concepts . . . 7
  - 1.3.3 C++ . . . 8
  - 1.3.4 Backus-Naur Form (BNF) . . . 10
- 1.4 Contributions and Outline of the Thesis . . . 10

2 Functional, Higher-Order Types **12**
- 2.1 An Overview of Type Systems . . . 13
- 2.2 Untyped Lambda-Calculus . . . 15
- 2.3 Simply Typed Lambda-Calculus . . . 16
- 2.4 Polymorphic Lambda-Calculus (System F) . . . 18
- 2.5 Lambda-Calculus with Type Constructors . . . 18
- 2.6 Kinds . . . 20
- 2.7 Higher-Order Polymorphic Lambda-Calculus (System F_ω) . . . 21
- 2.8 Higher-Order Functional Types . . . 23
- 2.9 Higher-Order Types and Object-Oriented Languages . . . 24
- 2.10 Types and Specification . . . 25
- 2.11 Related Topics and Conclusion . . . 26

3 Generic Software **28**
- 3.1 Generic Programming . . . 28
- 3.2 Genericity . . . 29
  - 3.2.1 Genericity by value . . . 30
  - 3.2.2 Genericity by type . . . 30
  - 3.2.3 Genericity by function . . . 30
  - 3.2.4 Genericity by concepts (Genericity by structure) . . . 31
  - 3.2.5 Genericity by property . . . 33
  - 3.2.6 Genericity by stage . . . 33
  - 3.2.7 Genericity by shape . . . 34
- 3.3 Polymorphism . . . 35
  - 3.3.1 Universal polymorphism . . . 35

3.3.2 Ad-hoc polymorphism . . . 36
3.3.3 Concept-controlled polymorphism . . . 37
3.3.4 Genericity and polymorphism . . . 37
3.4 Related Topics and Conclusion . . . 38

4 Program Transformation 39
4.1 Overview . . . 40
4.2 Defunctionalisation . . . 42
4.3 Our Transformational Approach . . . 43
4.4 Related Topics and Conclusion . . . 44

5 Kernel Languages 46
5.1 The Functional Type Kernel Language (FTKL) . . . 46
5.1.1 Syntax . . . 47
5.1.2 Kind rules . . . 49
5.1.3 Kind inference and kind checking . . . 50
5.1.4 Kind structure . . . 56
5.2 The Concept Type Kernel Language (CTKL) . . . 57
5.2.1 Syntax . . . 57
5.2.2 Kind rules . . . 58
5.2.3 Kind structure . . . 61

6 Transformation 63
6.1 Overview . . . 63
6.2 Uncurrying of Type Constructor Applications . . . 66
6.3 Defunctionalisation of the Codomain of a Function Signature . . . 68
6.4 Defunctionalisation of the Domain of a Function Signature . . . 71
6.5 Encoding Higher Kinded Type Variables . . . 74
6.6 Optimising Context . . . 75
6.7 The Complete Transformation and its Properties . . . 77

7 Backends 79
7.1 Language Features . . . 81
7.1.1 ConceptC++ . . . 81
7.1.2 Scala . . . 83
7.1.3 Haskell . . . 85
7.1.4 Other languages . . . 86
7.2 ConceptC++ Backend . . . 89
7.3 Scala Backend . . . 91
7.4 Generic Haskell Backend . . . 94
7.5 Conclusion . . . 96

8 Implementation 97
8.1 Architecture . . . 97
8.2 Compiler . . . 98
8.3 Performance . . . 99
8.4 Test systems . . . 104

9 Applications **107**
9.1 Higher-order Functions . . . 108
9.2 Recursive Data Types as Fixed Points . . . 110
9.3 A Domain Specific Language for Vulnerability Modelling . . . 111
9.3.1 Vulnerability formalisation and functional prototype . . . 112
9.3.2 Transformation and ConceptC++ implementation . . . 115

10 Discussion, Outlook, and Conclusion **118**

Bibliography **120**

A Algorithms **129**
A.1 Algorithm: find_fct . . . 129
A.2 Algorithm: replace_type . . . 130

B Test Output for ConceptC++ **131**

C Transformation Output for Origami++ **137**
C.1 Map . . . 137
C.2 Fold . . . 138
C.3 Unfold . . . 139
C.4 Hylo . . . 140

Acknowledgements **141**

List of Figures

1.1 A very simple intermediate model of vulnerability, in Haskell and in FTKL . . . 5
1.2 The CTKL function signature of the vulnerability function . . . 5
1.3 The C++ function object generated from the CTKL function signature . . . 6

2.1 The syntax of the untyped lambda-calculus . . . 15
2.2 The syntax of the simply typed lambda-calculus . . . 17
2.3 The typing rules of the simply typed lambda-calculus . . . 18
2.4 The syntax of the polymorphic lambda-calculus . . . 19
2.5 The typing rules of the polymorphic lambda-calculus . . . 19
2.6 The relation between values, types, and kinds (after [Pierce, 2002]) . . . 21
2.7 The syntax of the lambda-calculus with type constructors . . . 22
2.8 The kinding rules of the lambda-calculus with type constructors . . . 22
2.9 The kinding rules of the lambda-calculus with type constructors . . . 23
2.10 The lambda cube (from [Barendregt, 1991]) . . . 23

3.1 Kinds of polymorphism (from [Cardelli and Wegner, 1985]) . . . 35

4.1 Transformation $\mathscr{T}$ and language bindings . . . 44

5.1 The syntax of FTKL . . . 47
5.2 Derivation tree of the fold function signature from the Origami library in FTKL 48
5.3 Kind rules in FTKL . . . 50
5.4 Kind checking for the signature of the `fold` function . . . 51
5.5 Computation of the local kinds for the fold function signature . . . 55
5.6 The fold signature in FTKL and its kind structure . . . 57
5.7 The syntax of CTKL . . . 58
5.8 Derivation of the fold function signature from the Origami library in CTKL . . 59
5.9 Kind rules in CTKL . . . 60
5.10 The fold signature in CTKL and its kind structure . . . 61

6.1 Specification of transformation $\mathscr{T}$ . . . 77

7.1 A user-defined type constructor mapping . . . 80
7.2 Generic inversion in ConceptC++ . . . 82
7.3 Generic inversion in Scala—with subtyping . . . 84
7.4 Generic inversion in Scala—with implicits . . . 85
7.5 Generic inversion in Haskell . . . 86

7.6 Generic inversion in Java 87
7.7 Generic inversion in D 88
7.8 Function concepts in C++ 89
7.9 TypeConstructor concepts in C++ 89
7.10 Function concepts in Scala 92
7.11 Type constructor concepts in Scala 92
7.12 Function concepts in Haskell 95
7.13 Type constructor concepts in Haskell 96

8.1 The architecture of our transformation system 98
8.2 The compiler and its stages 99
8.3 Function concept and function datatypes in C++ 101
8.4 The test cases for the compiler 105

9.1 The Haskell function signatures we transform for the FCPPC library 108
9.2 The C++ implementation of function composition 109
9.3 Function signatures of the recursion operators of the Origami library 111
9.4 Function signatures of the list algebra implemented in Origami++ 111
9.5 The function signatures of the vulnerability model prototype 114
9.6 The vulnerability computation in ConceptC++ 116

C.1 The generated function signatures of the map function 137
C.2 The generated function signatures of the fold function 138
C.3 The generated function signatures of the unfold function 139
C.4 The generated function signatures of the hylo function 140

List of Tables

1.1 BNF syntax used 10

2.1 Statically typed, dynamically typed, and untyped languages 13
2.2 Curry-Howard isomorphism 25

7.1 Support of generic programming in different languages 81
7.2 Rewriting of a CTKL type in ConceptC++ 90
7.3 Rewriting of a CTKL function signature head in ConceptC++ 91
7.4 Rewriting of a CTKL function signature body in ConceptC++ 91
7.5 Rewriting of a CTKL concept in ConceptC++ 92
7.6 Rewriting of a CTKL type in Scala 93
7.7 Rewriting of a CTKL function signature head in Scala 94
7.8 Rewriting of a CTKL function signature body in Scala 94
7.9 Rewriting of a CTKL concept in Scala 95

8.1 Performance of function application, scaled to the fastest implementation (concepts) with -03 102
8.2 Performance of higher-order functions 103
8.3 Performance of the application of composed functions, scaled to the fastest implementation (concept-based approach) 103
8.4 Performance of the application of partially applied, curried functions, scaled to the fastest implementation 104

9.1 Functional C++-libraries: number of classes (and concepts), generated code, and total size (LOC). The ratio between these two is computed as the ratio between the lines of code generated and the lines of code of the whole library (including parts that are outside the transformation, like data-type definitions). . 107

List of Algorithms

5.1 kind_check: Kind checking for an FTKL function signature 51
5.2 infer_kinds: kind inference for an FTKL signature f 52
5.3 infer_local_kinds: local kind inference for an FTKL type t 54
6.1 uncurry_tc: type constructor uncurrying of an FTKL type t 67
6.2 codomain_defunctionalisation: defunctionalisation of the codomain type of an FTKL' function signature 69
6.3 domain_defunctionalisation: defunctionalisation of the domain type of an FTKL' function signature 71
6.4 concept_encode_HOF: encoding function types in the domain of a function signature with concept-constrained type variables 72
6.5 encode_higher_kinds: encode higher kinds of type variables in a CTKL function signature 74
6.6 encode_higher_kinds_in_type: encode higher kinds of type variables in a CTKL type 75
6.7 optimise_requirements: optimise the context (requirements and type parameters) of a CTKL function type signature 76
A.1 find_fct: find the first function type in an FTKL type 129
A.2 replace_type: replace in an FTKL type one type by another type 130

Chapter 1

Introduction

This dissertation presents a transformational approach to generic software development that is based on higher-order, typed functional signatures. This transformation supports the development of generic programs from functional prototypes and can be applied, among others, in scientific model development and in library development.

Models in scientific disciplines, such as climate change and economics, are generally complex and complicated, and their implementation in a (possibly low-level) programming language is an effort of several person-years. Therefore, prototypes are useful to test different designs, to improve the model on a conceptual level iteratively, and to thus avoid potential errors that might otherwise be apparent only after the costly stage of coding. Especially in the beginning, the problems that the model should help to analyse are often not well-understood, or not even precisely stated. Often, this missing clarity is caused by the fact that such models are located at the interface between computer science and, for instance, physics, economy, ecology, or social sciences. At this interface, computational modelling has to deal with different concerns (scientific, economic, political, ethical), with different methodologies (empirical, simulations, Gedankenexperimente, stakeholder dialogues, participatory games), and different specialised languages. In this mixture of methods and notions it is difficult to find a common formal framework to formulate the problems to be solved.

In such interdisciplinary setting, functional programming languages can prove helpful for efficient prototyping: their notation is close to mathematics, the language common across the disciplines, and they allow prototypical software development [Hudak and Jones, 1994]. Functional programs therefore are sometimes seen as executable specifications [Turner, 1985], and the programming style used to obtain such functional prototypes is referred to as exploratory programming [Sommerville, 2006]. We call functional prototypes in scientific modelling intermediate models.

Typically, those models do not aim at one stand-alone application but rather at a generic library, where one can plug together different subsystems and thus reuse the same functionality in multiple ways. For prototyping plug-in architectures, advanced type systems are essential. Here, type annotations do not just improve readability, they also allow for safe parameterisation. Because of static type checking, type errors and type parameter substitution failures can be detected in a mechanical way and at design time. Another advantage of functional languages is that, if they are typed, they can provide powerful type systems that include function types and type constructors. Function types allow higher-order functions, that is, functions parameterised with other functions—a feature that allows subcomputations in computations to be

replaced easily. Type constructors, in particular higher-order type constructors, allow abstraction over higher structures, for instance parameterised containers. Using a functional language for writing a prototype of a generic library for scientific models therefore has two major advantages. The first is its almost-mathematical syntax, which makes a functional prototype a vehicle for clear and precised communication that is also executable. The second advantage is the expressive type system that is provided by a statically typed functional language. Thus, such a functional prototype allows expressing dependencies on a higher level of abstraction and experimenting with parameterisation in advanced type systems.

By definition, a prototype is only a model of a software system, not the system itself. Once it has enabled the developers to gain an understanding of the structure and the dependencies of the system, the prototype has served its purpose and may thus be discarded. On the other hand, it could also be advantageous to base the implementation directly on the prototype, for example, by successive refinement. For the functional prototypes in scientific modelling just described, such refinement is neither done nor possible—mostly because the implementation languages are too different: common languages for implementation of scientific models are at their core object-oriented languages, and not functional ones. In particular those features that suggest the use of a functional language typically have no counterpart in the implementation languages. Without any link to the prototype, however, a large gap between the functional prototype and its implementation remains.

The work presented in this thesis bridges this gap as it allows reusing parts of functional prototypes even if the implementation language is different. We extract the function signatures from a functional prototype and present a transformation that puts them into function types in the implementation language. The function signatures capture tricky dependencies and abstractions at the level of types, and the transformation allows for the reuse of these dependencies. In functional languages, those dependencies are supported directly by the type system as function types or higher-order type constructors. In the target language we use features from generic programming to express these dependencies by type parameters constrained with concepts.

On the algorithmic side of the transformation defunctionalisation is a major step. The transformation removes higher-order function types from function signatures—a well-known problem in the processing of functional programming languages. The first program transformation that removed higher-order functions was suggested by Reynolds [Reynolds, 1972] under the name *defunctionalisation*. Variants of this method are known as *firstification* [Nelan, 1994], *higher-order removal* [Chin and Darlington, 1996], and *closure conversion* [Reynolds, 1972]. In contrast to other defunctionalisation methods, we transform only function types (signatures) but not the function implementations themselves. We defunctionalise polymorphic function signatures into polymorphic function signatures, and our target language is designed such that it provides special features to support generic defunctionalisation. Another remarkable difference is that source and target language commonly are the same in defunctionalisation, but different in our case; furthermore we do not need to know the complete source code but can already transform single function signatures. While defunctionalisation for a typed, polymorphic λ-calculus is known to be difficult, one can show that an extension of System F with guarded algebraic data types allows preserving types [Pottier and Gauthier, 2004].

Actually one would expect a transformation that should refine a specification to preserve the original semantics. However, formulating precisely what "to preserve semantics" should mean is not easy; in fact it is not obvious at all at what level one should compare target and source. Even if two types—the most important constituents of a signature— in source and

target language share the name, their semantics typically differ, so that one cannot expect the transformation to preserve semantics at the level of types. We solve the incompatibility by moving onto a higher level: we use *kinds* to describe the semantics of function type signatures. In brief, kinds introduce a notion to classify types and type constructors as functions at the level of types.

To capture the semantics of function type signatures in a way that is independent of any concrete programming language, we introduce two intermediate languages, the functional type kernel language (FTKL) and the concept type kernel language (CTKL). These two languages cover the essentials of function signatures in a functional and a non-functional, but concept-enabled language. Thus, the former contains a function type that can be used as any other type, while the latter provides descriptions for functional structures at the level of concepts, which are abstractions over types. The kernel languages serve two purposes: First of all, they allow defining a semantics for function signatures, namely a kind structure, which we define for expressions from both languages. This kind structure provides information about the kinds of types and type constructors in a function signature and the relation between these kinds. Secondly, these two intermediate languages make the transformation retargetable. Different languages can be used as frontend for FTKL and backend for CTKL; in Chap. 7 we present some backends.

1.1 Related Work

This thesis is the latest piece of work done in a research agenda on functional prototyping in scientific computing, starting with the work by Botta et al. [2006] and Botta and Ionescu [2007] who used a functional model as a prototype for a generic parallel computing library, and continued by Ionescu [2008] who used Haskell to implement an intermediate model of vulnerability in the domain of climate impact research. In Botta et al. [2011] the benefits of functional prototyping and intermediate models are discussed in detail by means of an agent-based model of exchange economies.

Earlier work of Zalewski et al. [2007] developed some initial ideas for a mapping from generic Haskell specifications to C++ with concepts, yet without providing any implementation. This thesis generalises the mapping by making it retargetable, formally proving its correctness, and providing an implementation.

Blending functional with object-oriented features has been a topic for a long time already; it is usually done feature by feature and typically requires language extensions. One important aspect of our work is the implementation of functional programming constructs as higher-order functions in nonfunctional languages such as C++. As this is a relevant topic of practical interest, there is some previous work, starting with Läufer [1995] and including the Boost Lambda library [Järvi et al., 2003a], the FC++ library [McNamara, 2004], and the Boost Fusion library [de Guzman and Marsden]. While all the previous approaches rely heavily on C++ template meta-programming [Abrahams and Gurtovoy, 2005], we encode function types at the level of concepts. Thus, our approach is more general: it is applicable to every language that provides concepts. Further, the other approaches just provide an ad-hoc implementation of higher-order functions but no transformation that is based on theoretical considerations.

Another aspect is the implementation of type constructor polymorphism, which, for example, has been introduced in Java by a language extension [Cremet and Altherr, 2008]; for Scala,

a research agenda [Moors et al., 2007, 2008, Moors, 2009] resulted in the introduction of type constructor polymorphism in the language. C++, on the other hand, allows simulating type constructor polymorphism with *template template* parameters [Vandervoorde and Josuttis, 2002], although those are for practical reasons not widely used. In any of those projects, however, researchers focused on one language and provided manually solutions that work exactly for that language. In contrast, our transformational approach works for more than one language and makes functional features available without language extensions. The downside, of course, is that at object-oriented level those functional features are not directly available.

There is previous work on relating generic programming in different languages. While Garcia et al. [2003, 2007] analyse the support of generic programming provided by different programming languages, Bernardy et al. [2008, 2010] directly compare type classes in Haskell with C++ concepts. Of particular importance for this thesis is the literature on the various aspects of the design, formalisation, and implementation of C++ concepts [Gregor et al., Zalewski, 2008, Zalewski and Schupp, 2009].

1.2 Motivating Example

In Fig. 1.1, we present an intermediate model of vulnerability. This model is implemented in the functional higher-order, typed language Haskell and presented in a related thesis [Ionescu, 2008]. It provides on the one hand an example of the kind of models we target and illustrates the higher-order, typed function signatures that characterise them. On the other hand, it serves as an example of the principles of our transformation.

The functional prototype is the outcome of a mathematical formalisation of the notion of vulnerability, as used in fields such as climate change, food security, or natural hazard studies, done by Ionescu [2008]. In principle, the mathematical formalisation captures the idea that vulnerability is a measure of the possible future harm for a given system. The evolution dynamics of the system can be of very different nature, formalised by different type constructors: deterministic behaviour is modelled by an identity type constructor, non-deterministic behaviour by a list type constructor, and stochastic behaviour by a probability distribution type constructor.

As the definition of vulnerability is highly generic, the implementation of vulnerability computations makes use of several higher-order, typed constructs. In particular, it uses higher-order functions to model the measurement (argument `measure`) and the harm function (argument `harm`), type constructors as the list type constructor, and type variables with higher kinds to abstract from the different system behaviours (`m`). Additionally, constraints on higher kinds (`Monad m`) are used.

Fig. 1.1 gives a flavour of the model itself, but also of the function signatures in Haskell and in our intermediate language FTKL. The biggest differences between the two languages is that FTKL supports only function signature declarations, no bodies. The function signatures themselves are quite similar, as, syntactically, FTKL coincides with a subset of Haskell. We explain the basics of Haskell syntax in Sect. 1.3; for now it suffices to know that FTKL covers function signatures and that these function signatures may contain higher-order function types (`(i -> x -> m x)`, `([x] -> h)`, and `(m h -> v)`) and higher-order type constructors (`m`). These higher-order constructs constitute the main difference between FTKL and its non-functional counterpart CTKL: FTKL provides such higher-order constructs at the level of types, while CTKL provides them as concepts. Thus, as the transformation processes FTKL function

```
-- Haskell
add_hist :: (Monad m) => (x -> m x) -> [x] -> m [x]
add_hist f (x : xs) = liftM (:(x : xs)) (f x)

micro_trj     :: (Monad m) =>  (i -> (x -> m x)) -> [i] -> x -> m [x]
micro_trj sys [] x      =  return [x]
micro_trj sys (i:is) x  =  (add_hist (sys i)) =<< (micro_trj sys is x)

vulnerability :: (Functor m, Monad m) =>
                 ((i -> x -> m x), ([x] -> h), (m h -> v), [i], x) -> v
vulnerability sys harm measure ts = measure . fmap harm . micro_trj sys ts


-- FTKL
add_hist      :: (Monad m) => (x -> m x) -> List x -> m (List x)
micro_trj     :: (Monad m) =>  (i -> (x -> m x)) -> (List i) -> x -> m (List x)
vulnerability :: (Functor m, Monad m) =>
                 ((i -> x -> m x), ((List x) -> h), (m h -> v), List i, x) -> v
```

Figure 1.1: A very simple intermediate model of vulnerability, in Haskell and in FTKL

```
vulnerability <m, FUN2, FUN3, FUN4> ::
    TypeConstructor m,
    Functor m,
    Monad m,
    Function FUN2.Codomain,
    SameType (FUN2.Codomain.Codomain, m<FUN2.Codomain.Domain1> ),
    Function FUN2,
    Function FUN3,
    SameType (FUN3.Domain1, List<FUN2.Codomain.Domain1> ),
    Function FUN4,
    SameType (FUN4.Domain1, m<FUN3.Codomain>)
    => (FUN2, FUN3, FUN4, List<FUN2.Domain1>, FUN2.Codomain.Domain1)
       -> FUN4.Codomain
```

Figure 1.2: The CTKL function signature of the vulnerability function

signatures and converts them into CTKL function signatures, higher-order functions have to be moved from the level of types to the level of concept-constrained type parameters. This process is known as defunctionalisation, and we introduce a defunctionalisation that encodes higher-order functions in the context of a function signature. The higher-order function types are replaced by type parameters constrained with a `Function` requirement. As identities between types are no longer expressible by identical type variables, they have to be made explicit by `SameType` requirements.

For the `vulnerability` function from Fig. 1.1, we present the output of the transformation in Fig. 1.2. It shows that the basic structure of the function is still the same (a five-tuple of input arguments), but it no longer contains higher-order functions. Further, the higher-order type constructor `m` is explicitly encoded by a `TypeConstructor` requirement.

CTKL is designed such that CTKL function signatures can be rewritten into actual program-

ming language code, Fig. 1.3 shows the final result of the transformation in concept-enabled C++. In short, the CTKL function signature is rewritten as a C++ function object. For details of

```
template<class m, class FUN2, class FUN3, class FUN4>
  requires
    Functor<m>,
    Monad<m>,
    Function<FUN2::Codomain>,
    SameType<FUN2::Codomain::Codomain, m::Apply<FUN2::Codomain::Domain1>>,
    Function<FUN2>,
    Function<FUN3>,
    SameType<FUN3::Domain1, List<FUN2::Codomain::Domain1>>,
    Function<FUN4>,
    SameType<FUN4::Domain1, m::Apply<FUN3::Codomain>>
struct vulnerability {

    typedef   FUN2  Domain1;
    typedef   FUN3  Domain2;
    typedef   FUN4  Domain3;
    typedef   List<FUN2::Domain1>  Domain4;
    typedef   FUN2::Codomain::Domain1  Domain5;
    typedef   FUN4::Codomain  Codomain;

    Codomain operator() (Domain1 x1, Domain2 x2, Domain3 x3,
                         Domain4 x4, Domain5 x5) {
    }

};
```

Figure 1.3: The C++ function object generated from the CTKL function signature

the syntax, we refer to Sect. 1.3.

Starting from a functional prototype and its mapping to FTKL (Fig. 1.1), the transformation generates its function types in a non-functional language. This function type provides a specification at the level of types, what remains for the developer is the implementation of the application operator—the typing of the function, thus its type-level specification, has already been generated automatically.

1.3 Preliminaries and Notation

As language for functional prototyping we mainly use Haskell; C++ as the main implementation language and so-called concepts for describing the requirements on type variables. In the following, we summarise the relevant features, notions, and syntax used in the remainder of the thesis.

1.3.1 Haskell

Even though any typed functional language with appropriate support of generic programming could serve for intermediate modelling and prototyping, we restrict ourselves to Haskell [Pey-

ton Jones, 2003], without any of the language extensions, as the sole prototyping language throughout the thesis.

We annotate every function with a signature, even though this is not necessary, as the type system of Haskell allows type inference of (most) functions. A function signature, for instance the one of the `vulnerability` function in Fig. 1.1, consists of a function identifier (`vulnerability`), followed by the type declaration operator `::`, an optional context (`Functor m, Monad m`), separated from the signature proper by `=>`, and the function type itself. Type variables in the function type are in lower case, type constructors start with an upper case letter. If the function type contains type variables, we speak of generic function signatures. Then, type classes can be used to constrain type variables with concepts. In the `vulnerability` function signature, the type classes `Functor` and `Monad` are used to restrict the type variable `m` to be replaced only by type constructors that are a functor and a monad.

Haskell provides a special syntax for function application using juxtaposition: `f a` denotes the application of `f` to `a`, parentheses are not needed. Functions with more than one argument are usually written in a curried form. The function `micro_trj` from Fig. 1.1, for instance, has the function signature `micro_trj :: (Monad m)=> (i -> (x -> m x))-> [i] -> x -> m [x]`, thus, it is a function that expects one argument and returns another function. The application of `micro_trj` to a value `sys` of type `(i -> (x -> m x))`, denoted `micro_trj sys`, returns another unary function, which, applied to a value `inputs` of type `[i]`, computes another unary function. The final result of `micro_trj` is computed by applying that last unary function to a value `initial_state` of type `x`. The repeated function application can be denoted by juxtaposition: `micro_trj sys inputs initial_state` denotes the application of `micro_trj` to `sys`, `inputs`, and `initial_state`. The syntax of function application is also used for type constructor application, which is, as we explain in Chap. 2, function application at the level of types. Thus, `m x` in the `vulnerability` function signature denotes the application of `m` to `x`.

Function composition is denoted by `f . g`, which is read "`f` following `g`", and has the type `(.) :: (b ->c) ->(a -> b) -> (a -> c)`. If the function identifier is put in parentheses, as `(.)` in the function composition example, that means that the function can be used in infix notation. The higher-order function `vulnerability`, for instance, is implemented as a composition of the functions `measure`, `fmap harm`, and `micro_trj`.

Haskell provides a built-in list type; the type of lists of elements of type `t` is denoted by `[t]`. This type provides two constructors, the empty list constructor, denoted by `[]`, and a non-empty list constructor, denoted by `(x:xs)` and representing the value of a list consisting of head `x` and tail `xs`.

Functions in Haskell are often defined by pattern matching, which means that they provide a separate definition for every constructor the data type of the function parameter provides. An example is the function that computes the length of an integer list:

```
1 list_length_int :: [Int] -> Int
2 list_length_int []     = 0
3 list_length_int (x:xs) = 1 + list_length_int xs
```

1.3.2 Concepts

Concepts are a programming language mechanism to constrain type parameters. Many generic functions are not applicable to arguments of any type, but rely on some properties of such type. Concepts are linguistic means to state these properties. They are similar to signatures in

algebraic specification as they are an abstract specification of a set of (tuples of) types, where the body contains associated entities: values, functions, or types themselves. Furthermore, a concept may state requirements on its types and include logical constraints, stated as equation-based axioms. Semantically, a concept is a predicate over types. When a type (or a tuple of types) satisfies this predicate—by providing appropriate definitions for each associated entity of the concept—it is called a *model* of the concept. If a concept is used to constrain a type parameter, the combination of the concept and the type parameter is called *requirement*. For instance, in Fig. 1.1, the concepts `Monad` and `Functor` are used in the requirements `Monad m` and `Functor m`, and similarly, but with different syntax, in Fig. 1.3 in the requirements `Monad<m>` and `Functor<m>`. In Haskell, the `Functor` concept is implemented as the following type class:

```
1 class Functor f where
2   fmap  :: (a -> b) -> f a -> f b
3   -- axiom  fmap id == id
4   -- axiom  fmap f.g == fmap f . fmap g
```

This concept declares a functor as a type constructor with an associated operation `fmap`. From its mathematical definition, a functor has to fulfil two axioms: It has to preserve identities and function composition. These axioms can not be included in a type class, however, they are stated as comments in the `Functor` type class above.

Historically, concepts are associated with the Standard Template Library (STL) in C++ [Stepanov and Lee, 1994]. In C++, concepts play an important role today in the ISO/ANSI standard libraries and the de-facto standard libraries from Boost [Boo], although they are not yet supported by standard C++. As the previous Haskell example already shows, concepts are not exclusive to C++. Comprehensive studies have mapped concept terminology to language features in a variety of languages and could show that, among others, Java interfaces and Scala traits can be seen as implementations of concepts in the object-oriented world, as well as Haskell type classes [Garcia et al., 2003, 2007, Bernardy et al., 2008, 2010]. We discuss the implementation of concepts in different languages in Chap. 7.

1.3.3 C++

Although our transformation is retargetable, all libraries we discuss in Chap. 9 are written in C++ [Becker, 2011, Stroustrup, 2000]. It is sensible, thus, to summarise upfront the major language features used. In C++, every function has to have a function signature in its definition:

```
1 float increment(float x) {
2     return x+1.0;
3 }
```

This declaration defines a function with name `increment`, with return type `float`, and with one argument of type `float`. Functions defined in this way are second-class values: they can be defined but not used as parameter for computations.

In C++, generic programming is supported by means of templates, which are pieces of code parameterised by types (and values). A template starts with the keyword `template` followed by the list of its type parameters in angle brackets. Every type parameter is preceded with the keyword `class` or `typename`, which are freely exchangeable. The function or class can then be defined as usual, and the type parameter can be used in the same way types are, for example in the declaration of argument or return types:

```
template<class T>
T f(T x, int y) {
    ...
}
```

If a template is used with a concrete type, it has to be instantiated. For instance, the notation `f<float> (10.0, 10)` instantiates the example from above with the type `float`.

C++ provides the `typedef` keyword to define type aliases. If such a type alias is defined within a class it is called a member type. The membership resolution operator is denoted by `::`.

Both functions, `increment` and the generic function `f`, are examples of free functions, as they are known from imperative programming. However, in C++, and actually any other object-oriented programming language, there is another way to define and to use functions, called function objects. Every function can be represented as a separate class, the example function `increment` from above becomes then:

```
struct Increment {
    typedef float  Domain;
    typedef float  Codomain;

    Codomain operator()(Domain x) {
        return x+1.0;
    }
};
```

In this case, `Increment` is a class, defined using the `struct` keyword, and therefore a type. The class `Increment` provides three members: two type aliases for the domain and the codomain type and an application operator that implements a mapping between these two types in the body of the function `increment`.

The application operator can be used in confix notation, and thus mimics usual function-invocation syntax. For example, for the variable `inc` of type `Increment` the expression `int x = inc(10);` reads as follows: the application operator is applied to `inc` as first and `10` as second argument, and its result is stored in `x`.

```
float x=1.1;
increment(x);
Increment inc;
inc(x);
```

Objects are first-class entities in object-oriented programming languages, they can be passed as arguments. Using function objects thus simulates the effect of higher-order functions. The disadvantage of representing functions as function objects is, however, that every function object defines a separate type. For instance, incrementation and decrementation define two classes `Increment` (see above) and `Decrement`, and if a function expects an argument of type `Increment` we can not provide a value of type `Decrement`. At the level of types it is not possible to state that a function expects a function object mapping integers to integers. Thus, function objects alone provide higher-order functions in a very limited sense. However, in this thesis we will show how to use generic programming, concepts, and function objects to provide full support of higher-order functions.

Concepts are not supported in standard C++, but available as an experimental language extension and subject to recent research (see Chap. 7). In standard C++ there is no possibility

to constrain type parameters; requirements on these type parameters are implicit and document a kind of contract between the designer and the user of a generic library.

1.3.4 Backus-Naur Form (BNF)

Throughout this thesis, we make frequent use of the Backus-Naur Form (BNF) to describe the syntax of the languages we introduce. Defined by Backus [1959], the BNF provides a notation for context-free grammars, thus is suitable to define and describe the syntax of most programming languages. Since many variants exist today, we summarize in Table 1.1 the BNF syntax we use throughout the rest of the thesis.

BNF-syntax	Meaning
<x>	nonterminal item
"x"	terminal item
x | y	x and y are alternative items
[x]	x is an optional item
x+	x is repeated 1 or more times
{ x }	x is repeated 0 or more times
::=	the BNF declaration operator

Table 1.1: BNF syntax used

1.4 Contributions and Outline of the Thesis

This thesis is divided into two parts. The first part, comprising Chap. 2 - Chap. 4, introduces the theoretical background and sets the stage for the rest of the thesis.

Chap. 2 introduces type systems and shows why they are important for producing correct software. This chapter also provides a classification based on Barendregt's lambda cube, and enables us to precisely define the type systems that provide the input function signatures to our transformation. Further, this chapter introduces kinds, which provide the possibility to classify types. Kinds are later used to define a semantics of type expressions and to prove correctness properties of the transformation. Chap. 3 summarises the idea of generic programming and relates generic programming to type systems. It provides an overview of different kinds of generic programming and analyses the usage of the terms genericity and polymorphism. In particular, it reviews the usage of concepts in generic programming and explains the notion of concept-controlled polymorphism. Chap. 4 discusses the idea of program transformation and provides a classification of different program transformation methods. In particular, it summarises prior work on defunctionalisation, the most important step in our transformation. At the end of this chapter, we have set the terminology to characterise our transformation so that the chapter concludes with an outline of our transformation system.

The second part of the thesis, Chap. 5 - Chap. 9 contains the main contributions of the thesis. Chap. 5 defines the two kernel languages we use, the functional kernel language FTKL and the concept kernel language CTKL. The chapter provides their syntax and their semantics. We define a semantics on a higher level of abstraction: we use kinds and introduce the *kind structure* of FTKL and CTKL expressions. This kind structure allows us to relate FTKL and CTKL

expressions. Chap. 6 describes the transformation itself. It consists of six algorithms that subsequently uncurry type constructor applications, defunctionalise function types, encode higher kinds with concepts, and optimise the generated requirements. For each algorithm we show that well-kindedness and kind structure are preserved and that therefore the whole transformation preserves the semantics of expressions. In Chap. 7, we describe the backends we provide for different languages, in Chap. 8, we show how the whole transformation system is actually implemented. Chap. 9 demonstrates how the transformation is used in the development process of different libraries and applications.

Finally, the thesis concludes with an outlook and discussions (Chap. 10).

Chapter 2

Functional, Higher-Order Types

The adjective "HOT", meaning *Higher-Order, Typed*, covers a broad class of programming languages characterised by advanced type and module systems on the one hand, and higher-order features like first-class functions or objects on the other hand [MacQueen, 2001]. In principle, functional and object-oriented programming can be seen as different instances of the more general phenomenon of HOT languages. Even if their induced way of thinking and their style of programming is quite different, both paradigms are grounded in expressive type systems. The presence of such expressive type systems shapes the way software is developed, Pierce and Turner [2000] call it *higher-order programming*.

Due to the broad usage of the term, a precise definition is lacking. This chapter aims at providing a working definition of functional HOT languages and positions HOT in the design space of type systems, using as vehicle the simplest functional programming language there is: the lambda-calculus. To set the stage, we provide a brief overview of the idea of types in computing, their history, the intention and the benefits of type systems, followed by an explanation of how type systems can evolve by incorporating different levels of abstraction. Starting from an untyped version, we show the problems that occur and how to avoid them by introducing a type system. However, a simple type system has limitations itself. The type of polymorphic functions, which take arguments of different types, is not expressible in a simple type system. Thus, functions that work independently of the type of their arguments have to be implemented for every type they are supposed to work on. We show how to overcome this limitation with a polymorphic lambda-calculus. Another extension of the simply typed lambda-calculus is the introduction of type constructors, that is, type functions that transform types into types. A type system that supports type constructors allows expressing, for instance, array or list types. Type constructors can be combined with polymorphism, yielding higher-order polymorphic type systems.

Finally, we present a working definition of functional, higher-order, typed languages. We consider functional, higher-order type systems those type systems that provide a function type supporting higher-order functions, type constructors, and polymorphism. A functional, higher-order, typed language is then a functional language with a functional, higher-order type system.

2.1 An Overview of Type Systems

While type systems are standard in many modern programming languages, they originate not in computer science but in mathematics. There, types were introduced to avoid logical paradoxes, reaching back to Whitehead and Russell [1925–1927] and Ramsey [1925]. The simply typed lambda-calculus [Church, 1940, 1941] made a first connection between types and programming.
In actual programming, type systems were introduced in the late 1950s and early 1960s in Fortran [Backus, 1981] and Algol [Naur, 1963]. As the reason for their introduction originally was to improve performance of numerical computation, they just offered the possibility for very simple distinctions between integer and floating-point representations of numbers. However, they were extended rather soon, leading to arrays, record types, and higher-order functions. From the 1970s, type systems developed into a form of formal methods, which support programming in various ways and build a strong theoretical base of computing.

As with other concepts that are used across, or differently within, communities, it is difficult to give a definition that is commonly accepted. One possible definition of a type system from the computer science point of view (given by Pierce [2002]) reads as follows:

Definition 1 *A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.*

According to this definition, type systems are tools that classify phrases (expressions) from the code of a program. Such a classification can be done without executing the program, and even before the program is compiled into an executable format. Thus, type systems, as defined above, are tools that support the compilation, and type checking is embedded in the compilation process. In this thesis, we only consider languages that ensure that every expression has a type attached at compilation time (which is not to say that the types have to be attached by the programmer, as a type inference system can derive them automatically). Such languages are often called statically typed languages, in contrast to dynamically typed languages, which attach at runtime type tags to values. Type violations in dynamically typed languages are only detected when an operation is executed. Finally, there are untyped languages, such as most assembly languages, which in principle allow any operation to be performed on any data. Such languages usually operate at the level of registers, and data are considered as sequences of bits. Operations are performed bitwise without any interpretation of the bit sequences. Type errors might become visible when the results of mistyped operations are interpreted, for instance by a hardware interface. There are also untyped high-level languages, for instance varieties of Forth.

Table 2.1 collects some examples of untyped, dynamically typed, and statically typed languages. The table also gives an impression of the different nature of the differently typed lan-

Statically typed	Dynamically typed	Untyped
C, C++, C#	Ruby, Python	Assembler
Java, Scala	JavaScript	Forth
Haskell	Lisp	

Table 2.1: Statically typed, dynamically typed, and untyped languages

guages. While untyped languages are rather close to machine programming, dynamically typed languages often are scripting languages. Such languages, also called extension languages, are used to control applications. For such languages, it is more important that they are easy to understand than to provide high performance. Therefore, typing in such languages is done automatically, which leads to decreased complexity of the language but also to decreased performance. Statically typed languages, in contrast, are often referred to as general purpose languages. Their expressive type systems are the base of a rich syntax and semantics, and compilers for these languages use type information to generate highly efficient code. In the following, when we speak of type systems, we always refer to statically typed languages; dynamically typed and untyped languages are beyond the scope of this thesis.
Statically typed languages can be either explicitly typed or implicitly typed. In the first case, the programmer has to attach a type to every declaration and to every function argument. In contrast, the compiler can infer the types of expressions and declarations automatically in the second case. Examples of explicitly typed languages are most mainstream statically typed languages, such as C++, C#, and Java. Implicitly typed languages include, for instance, Haskell, Scala, and ML. However, some explicitly typed languages, such as Java or C#, support partial type inference in certain cases.

Type systems offer advantages that justify the additional complexity they introduce in a programming language. The main benefits of type systems are:

- **Error detection:** Logical errors in the design of a computer program can show up as inconsistencies at the level of types. For example, the expression `s + x`, where `s` is of type `string` and `x` is of type `integer`, indicates a logical error in the design of the program. A type checker should detect such an error (unless there is a overloaded version of + that adds strings and integers) and reject the program. Thus, type systems can catch errors at the level of types at compile time and avoid runtime failure cause by this errors.

- **Efficiency:** In numerical computations, compilers can increase efficiency by distinguishing between different kinds of numerical values. For instance, memory allocations can be done at compile time if the type of a variable is known, leading to increased runtime efficiency of programs. Moreover, there are different versions of machine instructions of primitive operations for different types. In a typed language, the compiler can choose the most appropriate version and thus increase the speed of execution of a program.

- **Safety:** As type systems prevent type errors at runtime, they increase the reliability of software. Type errors at runtime can lead to core dumps or to unexpected program behaviour, which might have disastrous effects. Some languages (as, for instance, ML) prevent type errors at runtime completely as they ensure that every expression will have the according type. Such languages are sometimes called type-safe languages. Other languages allow the programmer to explicitly violate type safety by changing the type of a value or expression. Such operations, called type casts, are allowed, for instance, in C++.

- **Abstraction:** Complex types allow programmers to think about programs at a higher level of abstraction. A matrix type, for instance, enables a programmer to think of a matrix as a sequence of elements instead of as an array of bytes. Further, types specify the interfaces of objects, modules, or functions, thus facilitating interoperability and communication between such subsystems.

- **Documentation:** Type annotations document some aspects of the semantics of values. By their names, types can give the programmer information about their semantics. If a type in an economic model is named `goodstock`, the name documents that the values of this type are intended to stand for good stocks, whereas values of a type `moneyflow` are intended to stand for a time derivation of an amount of money. Thus, a typed program provides much more information to a programmer than an untyped program.

In the following sections, we study the evolution of type systems. As type systems are integrated in a programming language, we explain them using the simplest possible programming language: the lambda-calculus. The presentation of the different versions of the lambda-calculus is inspired by the standard book by Pierce [2002].

2.2 Untyped Lambda-Calculus

The lambda-calculus, introduced by Church [1936], is a formal system that captures the essence of computation: function definition, function application, and recursion. In the lambda-calculus, there are just terms and values, Fig. 2.1 shows the syntax of the untyped lambda-calculus in a pure form. In this calculus, all values have to be represented as λ-abstractions. Richer versions of the untyped lambda-calculus include other values: numeric and boolean constants, strings, and so on. As we will see, however, all these values can be represented in the pure calculus from Fig. 2.1 as λ-abstractions. There are four kinds of terms. Variables are the simplest

```
t ::=                   | terms:
        x               | variable
        λx.t            | λ-abstraction
        t t             | term application
        (t)             | parentheses

v ::=                   | values:
        λx.t            | abstraction value
```

Figure 2.1: The syntax of the untyped lambda-calculus

terms, complex terms are the abstraction of a variable `x` from a term `t`, denoted as `λx.t`, and the application of a term `t1` to another term `t2` is denoted `t1 t2`. Finally, a term in parentheses is also a term. The notion of a λ-abstraction is central to all versions of the lambda-calculus. The term `λx.t` denotes the function that, for each `x`, yields `t`. Thus, the identity function can be denoted as `λx.x`. A λ-abstraction denotes a function that does not have an identifier; therefore λ-abstractions are also called anonymous functions, and are available in the syntax of functional programming languages. For instance, in Haskell, the identity function can be written as λ-abstraction `\x -> x`, where the backslash `\` represents the Haskell notation for λ.
In term application, every term can be (syntactically) applied to every term. In particular, the argument term `t2` in a term application `t1 t2` can be a λ-abstraction, denoting a function itself. Thus, in the lambda-calculus the arguments accepted by functions can themselves be functions, and the result returned by a function can also be a function. In programming languages, such

functions that accept or return functions are referred to by a special name: higher-order functions.
The pure lambda-calculus does not provide a notion for tuples, thus functions can have only one input argument. A function that operates on more inputs can be expressed by a function on one argument that returns another function operating on one argument. The transformation of a function operating on a tuple of inputs into a function operating on one input that returns another function is called currying, the procedure of currying, applying the curried function and then uncurrying the resulting function is called partial application of a function.

It is possible to express booleans and natural numbers in the untyped lambda-calculus. Values are expressed as λ-abstractions according to the syntax from Fig. 2.1. For instance, the boolean values `true` and `false` can be defined as:

```
true  := λt. λf. t
false := λt. λf. f
```

Boolean operators can then be defined accordingly, resulting in an extension of the untyped lambda-calculus that provides boolean logic. Thus, even if its syntax is rather limited, the lambda-calculus is indeed a universal programming language providing the possibility to implement the usual values and operations.

Although the lambda-calculus and all its extensions are very simple and idealised programming languages, they can be shown to be Turing-complete. The semantics of the lambda-calculus is given in the form of evaluation rules for terms. However, as we are interested here in type systems only, the semantics of terms of the lambda-calculi is not in scope of this thesis, for further readings we refer to Pierce and Turner [2000].

The untyped lambda-calculus is easy to use. Yet, it allows some nonsensical expressions that could easily be ruled out if types would be taken into account. For instance, in the version extended with booleans, the term `true (λx. x)` cannot be evaluated as the application of the evaluation rules gets stuck when trying to evaluate the application term. In this example, a boolean value is applied to a function, which is not possible because the boolean value `true` has a type that is not applicable to anything. Such nonsensical expressions can be precluded by a type system. A type checker is applied before the program is executed. The program is only executed if it passes the type checker, otherwise it is rejected with a message stating that the program is not correctly typed. Expressions and programs that pass the type checker are called well-typed; if a program is well-typed all expressions it contains are well-typed. In the next section, we show how a simple type system can be introduced into the untyped lambda-calculus.

2.3 Simply Typed Lambda-Calculus

As shown above, the original version of the lambda-calculus, which is untyped, allows certain nonsensical expressions. One class of such nonsensical expressions is that of term applications `t1 t2` where either the first term `t1` is not an applicable term or the term `t1` is an applicable term but may not be applied to the term `t2`.

To rule out such terms, Church [1940, 1941] introduced a typed version of the lambda-calculus. Fig. 2.2 presents the syntax of the simply typed lambda-calculus. The part that defines the type system, and therefore marks the difference to the untyped lambda-calculus (Fig. 2.1),

```
t ::=                   | terms:
      x                 | variable
      λx:T.t            | abstraction
      t t               | application
      (t)               | parentheses

v ::=                   | values:
      λx:T.t            | abstraction value

T ::=                   | types:
      bool, int, ...    | basic types
      T -> T            | type of functions
      (T)               | parentheses

Γ ::=                   | contexts:
      ∅                 | empty context
      Γ, x:T            | term variable binding
```

Figure 2.2: The syntax of the simply typed lambda-calculus

is printed in red. In particular, every λ-abstraction is now annotated with the expected type of its arguments. In addition, the calculus now contains a very simple grammar for types and rules for contexts. As there are no λ-abstractions at the level of types, we introduce terminal strings for basic types such as `bool` for booleans and `int` for integer values. In addition, function types `T1 -> T2` can be constructed from any two types. In a function type `T1 -> T2`, the type `T1` is called domain type and the type `T2` is called codomain type. Parentheses can be used to group types, which is in particular useful in the case of complex function types. The function type construction associates to the right, so `T1 -> T2 -> T3` denotes the same type as `T1 -> (T2 -> T3)`; to denote the type `(T1 -> T2)-> T3`, parentheses are necessary. The typing context Γ in the last production contains a sequence of variables and their types; it can be an empty context or an existing context that is extended by adding a new binding of a variable to its type.

The introduction of types comes along with rules that compute the type of a term given the types of its subterms. These typing rules comprise three rules, which are presented in Fig. 2.3. The typing rules read as follows: the upper part, called premise, states type information that is given. The lower part, called conclusion, states type information that follows from the premise. The formula $\Gamma \vdash$ `t : T` states that the term `t` has type `T` under the assumptions given by the context Γ.

In particular, these three rules state the following: the first rule, (T-Var), states that if a context contains a certain binding, that binding can be deduced from the context. The second rule, (T-Abs), states that a λ-abstraction has type `T1 -> T2` if its abstraction variable has type `T1` and its body has type `T2`. The third rule, (T-App), states that if a function with type `T1 -> T2` is applied to an argument of type `T1`, the result will be of type `T2`. The typing rules can be applied to perform a type check: given an expression from the calculus, the rules can be used to check whether there is a type that can be assigned to that expression using these rules.

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x}:\mathtt{T}} \quad \text{(T-Var)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2:\mathtt{T}_2}{\Gamma \vdash (\lambda \mathtt{x}:\mathtt{T}_1.\mathtt{t}_2):\mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \to \mathtt{T}_2 \quad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_1}{\Gamma \vdash (\mathtt{t}_1\ \mathtt{t}_2):\mathtt{T}_2} \quad \text{(T-App)}$$

Figure 2.3: The typing rules of the simply typed lambda-calculus

2.4 Polymorphic Lambda-Calculus (System F)

The type system of the simply typed lambda-calculus does not allow quantification over types and thus does not support polymorphic functions. Therefore, it is inconvenient to define functions like the identity: in principle, there has to be an identity function λ`x:T.x` for every type `T`. To overcome these limitations, a mechanism of universal quantification over types was introduced by Reynolds [1974] (under the name polymorphic lambda-calculus) and Girard [1971] (who called it System F). Fig. 2.4 shows the syntax of the polymorphic lambda-calculus, the type system is printed in red. The polymorphic lambda-calculus adds another level of abstraction: λ-abstractions over types, denoted by Λ. The notion Λ`T.e` denotes the expression that, for each type `T`, yields `e`. Thus, the polymorphic identity function, for instance, can be written as `id = ΛT.λx:T.x`, which reads as the expression that, for every type `T`, yields `λx:T.x`, the identity function over `T`. The notion `t [T]` specifies that a polymorphic term `t` is instantiated with type `T`. Thus, `id [float]` denotes the identity function on floating-point numbers. In addition, universal type quantification is introduced to denote the type of polymorphic expressions. For instance, the type of the identity function can be denoted as `∀X. X -> X`. To type-check polymorphic expressions, the typing rules of the simply typed lambda-calculus (Fig. 2.3) are extended by two new typing rules (Fig. 2.5). The rule (T-TAbs) states that a Λ-abstraction has universal type, the rule (T-TApp) states that a type application on a universal type results in a type where the type variable of the abstraction is replaced with the type application argument; the notion `[X -> T] T`$_2$ denotes substitution of `X` with `T` in the term `T`$_2$. For example, the type of `id [float]` is `[X →  float] (λx:X.x)` and thus `λx:float.x`.

2.5 Lambda-Calculus with Type Constructors

In the simply typed lambda-calculus, it is possible to apply terms to terms, if their types match. In the polymorphic lambda-calculus, it is possible to apply terms to types, leading to the notion of term instantiation. Another level of abstraction can be introduced by allowing the application of types to types, yielding type functions. Such type functions are also called type constructors as they construct new types out of given argument types. In programming languages, type constructors are implemented as parameterised types. An example is the array type constructor, which is usually parameterised on the type of the elements of the array. Being not a type itself

```
t ::=                   | terms:
        x               | variable
        λx:T.t          | abstraction
        t t             | application
        ΛT.t            | type abstraction
        t [T]           | type application

v ::=                   | values:
        λx:T.t          | abstraction value
        λX.t            | abstraction type


T ::=                   | types:
        bool, int, ...  | basic types
        T -> T          | type of functions
        ∀X.T            | universal type
        (T)             | parentheses

Γ ::=                   | contexts:
        ∅               | empty context
        Γ, x:T          | term variable binding
        Γ, X            | type variable binding
```

Figure 2.4: The syntax of the polymorphic lambda-calculus

$$\frac{\Gamma, X \;\vdash\; \mathtt{t} : \mathtt{T}}{\Gamma \;\vdash\; \Lambda \mathtt{X}.\mathtt{t} : \forall \mathtt{X}.\mathtt{T}} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \forall \mathtt{X}.\mathtt{T}_1}{\Gamma \;\vdash\; \mathtt{t}\,[\mathtt{T}_2] : [\mathtt{X} \to \mathtt{T}_2]\mathtt{T}_1} \qquad \text{(T-TApp)}$$

Figure 2.5: The typing rules of the polymorphic lambda-calculus

(an array is not a type), it constructs a type out of the type parameter (an array of floats is a type). Most languages provide limited support for type constructors, as built-in parameterised types, for instance array-types, are provided, but the possibility for user-defined parameterised types is mostly lacking.

To extend the lambda-calculus with type constructors, a formal notion for type abstraction and application is needed. This notion was already introduced in the polymorphic lambda-calculus: `ΛX.T` denotes a function that, given a type `X`, computes the type `T`. For instance, `ΛX. Array X` encodes the array type constructor. The notion of application is accordingly expanded to types, thus `(ΛX. Array X) int` denotes the application of the array type constructor to the

type `int`.

However, in a sense, the expression ΛX.T is as untyped as the expression λx.t in the untyped lambda-calculus. The missing type information for types leads to the same problems as in the case of terms: not every type expression can be applied to every type expression. To formalise the lambda-calculus with type constructors, types of type constructors have to be provided. To avoid confusion, such types are called kinds and introduced below. Afterwards, we present the lambda-calculus with type constructors.

2.6 Kinds

Kinds characterise types in the same manner as types characterise values; in that sense, kinds are "types of types." Thus, kinding is a well-formedness relation that prevents nonsensical type expressions. As an example, let us consider the type expression `(Bool Nat)`, which can be written according to the syntactic rules of type application. However, `(Bool Nat)` is meaningless as `Bool` is not a type constructor that could be applied to a type.

To avoid such problematic expressions, kinds are introduced to classify type expressions according to their structure. Kinds are built from three basic constructors: an atomic kind $*$ (pronounced *type*), a unary constructor `=>`, and parentheses, which may be applied to every kind. Examples of kinds are:

- $*$ is the kind of all data types that can be seen as nullary type constructors. Those types are also called proper types. The type constructor `Bool` is an example of a nullary type.
- $* \Rightarrow *$ is the kind of a unary type constructor that constructs a new type out of a given type. An example is the list type constructor.
- $* \Rightarrow * \Rightarrow *$ is the kind of a (curried) binary type constructor that constructs a unary type constructor out of a given type. An example is the pair type constructor.
- $(* \Rightarrow *) \Rightarrow *$ is the kind of a higher-order type operator from unary type constructors to proper types. An example is an `ApplyToInt` operator, which expects as argument a type constructor and returns a type that is constructed by applying that type constructor to the type `Int`.

Fig. 2.6 shows how values are classified by types and types are classified by kinds. The typing relation rules out nonsensical expressions such as `true 5` as they are not well-typed. Other expressions get types according to their structure: λ-abstractions get function types, and applied λ-abstractions get the codomain type of the function type. At the level of types, nonsensical expressions such as `bool int` or `Array Array` are ruled out as not well-kinded.

The syntax for kinds reflects that the syntax for terms and types does not provide tuples. Thus, analogously to functions on terms, type constructors that operate on two or more inputs can be expressed by a unary type constructor that returns another type constructor. The pair type constructor is an example: it is a type constructor that can be applied to an argument, resulting in another type constructor applicable to an argument. Thus, `Pair int` is a type constructor, while `Pair int float` is a type.

The kinds defined above can be included in the lambda-calculus with type constructors, as shown in Fig. 2.7. While the type system of the calculus is printed in red as before, the

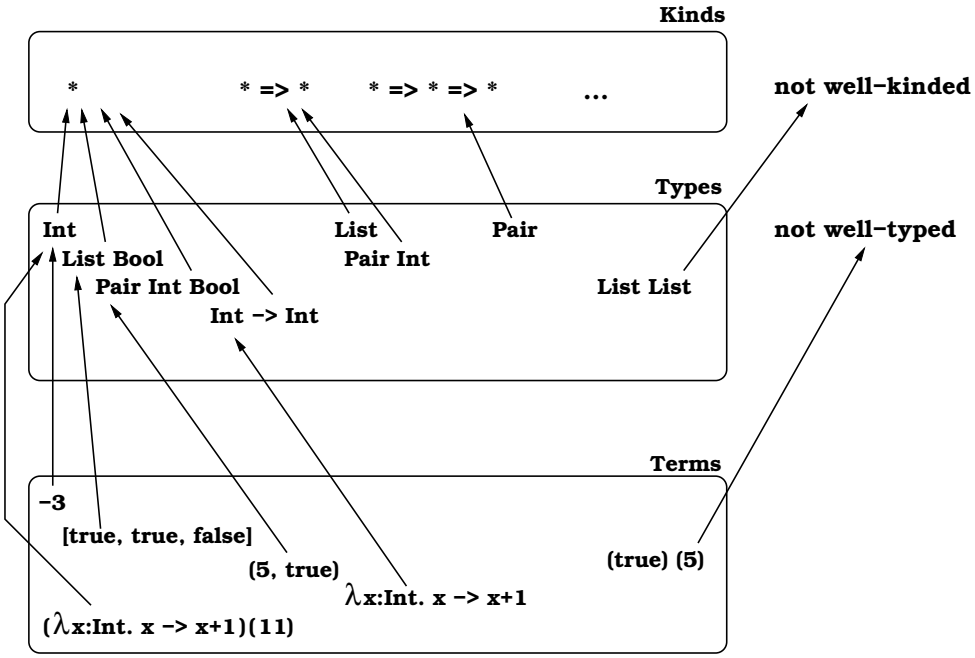

Figure 2.6: The relation between values, types, and kinds (after [Pierce, 2002])

kind system is printed in blue. In comparison with the syntax of the simply typed lambda-calculus, the lambda-calculus with type constructors is extended with a simple grammar for kinds and kind annotations for type constructor abstractions. Terms of this calculus do not only have to be type-checked, but also kind-checked. Kind checking is based on four kinding-rules, summarised in Fig. 2.8. The first of these rules, (K-Var), is the counterpart of the typing rule (T-Var) and states that if a context contains a certain kind binding, that binding can be induced from the context. The second rule, (K-Abs), states that a Λ-abstraction has kind `K1 => K2` if its abstraction type variable has kind `K1` and its body has kind `K2`. The third rule, (K-App), states when a type application is well-kinded. This is the case if a type operator with kind `K1 => K2` is applied to a type with kind `K1`; the result will then be of kind `K2`. Finally, the rule (K-Arrow) declares that a function type can only be built from two proper types and has then proper type itself.

2.7 Higher-Order Polymorphic Lambda-Calculus (System F_ω)

The lambda-calculus with type constructors and the polymorphic lambda-calculus both extend the simply typed lambda-calculus but neither extends the other: the polymorphic lambda-calculus misses type constructors, the lambda-calculus with type constructors misses the notions of type abstraction and type application (for terms). Thus, it is a natural next step to combine both, yielding the higher-order polymorphic lambda-calculus, also known as System F_ω [Girard, 1972].

System F_ω extends the lambda-calculus with type constructors and quantification over types to higher-order quantification over type operators. In comparison with the lambda-cal-

```
t ::=                   | terms:
      x                 | variable
      λx:T.t            | abstraction
      t t               | application

v ::=                   | values:
      λx:T.t            | abstraction value


T ::=                   | types:
      bool, int, ...    | basic types
      T -> T            | type of functions
      λX::K.t           | type constructor abstraction
      T T               | type application

Γ ::=                   | contexts:
      ∅                 | empty context
      Γ, x:T            | term variable binding
      Γ, X::K           | term variable binding

K ::=                   | kinds:
      *                 | kind of proper types
      K => K            | kind of operators
```

Figure 2.7: The syntax of the lambda-calculus with type constructors

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \quad \text{(K-Var)}$$

$$\frac{\Gamma, X :: K_1 \vdash T :: K_2}{\Gamma \vdash \lambda X :: K_1 . T :: K_1 => K_2} \quad \text{(K-Abs)}$$

$$\frac{\Gamma \vdash T_1 :: K_1 => K_2 \quad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1\ T_2 :: K_2} \quad \text{(K-App)}$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 -> T_2 :: *} \quad \text{(K-Arrow)}$$

Figure 2.8: The kinding rules of the lambda-calculus with type constructors

culus with type constructors, syntax for a universal type $\forall X::K.T$ is added. To distinguish type annotations from kind annotations, the latter are written using a double colon. Kind checking

of such universal types, is done by an additional kinding rule, shown in Fig. 2.9. The rule states

$$\frac{\Gamma, \mathtt{X} :: \mathtt{K}_1 \ \vdash \ \mathtt{T} :: *}{\Gamma \ \vdash \ \forall \mathtt{X} :: \mathtt{K}_1 . \mathtt{T} \ :: *} \quad \text{(K-Universal)}$$

Figure 2.9: The kinding rules of the lambda-calculus with type constructors

that universal quantification over a type expression has proper type if the type expression itself has proper type.

2.8 Higher-Order Functional Types

As shown above, the untyped lambda-calculus can be extended with types yielding the simply typed lambda-calculus. This simple type system, in turn, can be extended in several, orthogonal ways, defined by the different ways in which terms and types can depend on each other. In the polymorphic lambda-calculus, terms can be constructed that depend on types, in the lambda-calculus with type constructors, types can be constructed that depend on types. In the same spirit, it is possible to introduce type systems in which types depend on terms. Such types are called dependent types [Altenkirch et al., 2006] and are beyond the scope of this thesis. The different dimensions of type systems in the lambda-calculus are summarised in the lambda

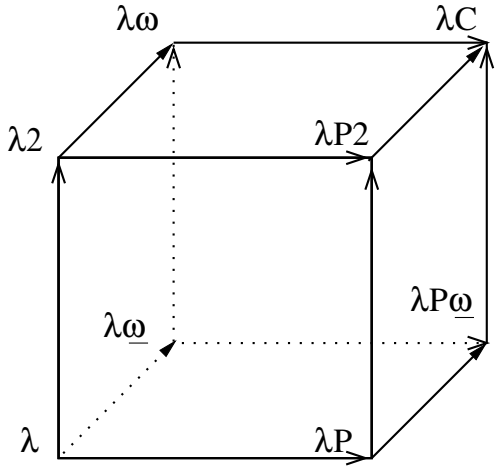

Figure 2.10: The lambda cube (from [Barendregt, 1991])

cube [Barendregt, 1991]. Fig. 2.10 depicts the lambda cube: it includes eight different typed lambda-calculi. Extensions are indicated by arrow directions. The simply typed lambda-calculus is located in the lower left corner. The top face of the cube represents type systems with polymorphism (the polymorphic lambda-calculus is denoted $\lambda 2$), the back face type systems with type constructors (the lambda-calculus with type constructors, denoted $\lambda \underline{\omega}$), and the right face type systems that support dependent types (the dependently typed lambda-calculus is de-

noted λP). In the upper right corner there is the most general lambda-calculus, called universal lambda-calculus and denoted λC, containing all three extensions.

With this classification of type systems, we can define functional and higher-order type systems in the following way:

Definition 2 *A functional type system is a type system that supports function types in such a way that domain and codomain types can be function types themselves (higher-order functions).*

Definition 3 *A type system is a higher-order type system if it supports higher-order types (type polymorphism) and higher-order type constructors (type constructor polymorphism).*

Regarding the type systems in the lambda cube, we consider them all as functional as they all support higher-order functions. Higher-order functional type system are those of the higher-order polymorphic lambda-calculus $\lambda\omega$ and of the universal lambda-calculus λC. However, as dependent types are not in the scope of the thesis, we concentrate on types from $\lambda\omega$. In fact, the examples from Fig. 1.1 can only be expressed in the type system of $\lambda\omega$ or an extension of it, as they make use of higher-order functions, polymorphic types, type constructors, and type constructor polymorphism. However, it is not clear how to express such types in languages with type systems that are not based on the lambda-calculus. In the next subsection, we will discuss the relation of higher-order types and object-oriented languages.

2.9 Higher-Order Types and Object-Oriented Languages

As mentioned above, functional programming languages in general originate from the lambda-calculus. Therefore, they provide first-class functions and often some kind of polymorphism. Haskell, for instance, allows types that can be expressed in the higher-order polymorphic lambda-calculus, which means that it also supports type constructor polymorphism.

Object-oriented languages, on the other side, promote other modelling strategies and therefore require other type systems. While functional languages emphasise the idea of a pure computation and thus provide every computation as a function operating on separate input and output data, object-oriented languages emphasise the idea of objects that combine functions and the data they compute on in one structure. The first-class structures in functional languages are functions, while in object-oriented languages they are objects. The different first-class structures are reflected in the type systems of these languages: the type system of a functional language offers support for higher-order functions, the type system of an object-oriented language does not, and offers instead support for higher-order objects.
In object-oriented languages, there are usually two sorts of types: implementation types (classes) and interface types. Classes usually define a new type, and can be subclasses of other classes and inherit implementations from them. If inheritance is embedded in the type system it implements subtyping, otherwise inheritance is independent of subtyping [MacQueen, 2001]. Subtyping can be integrated in the lambda-calculus yielding System $F^{\omega}_{\leq}$. Considered as a programming language, however, $F^{\omega}_{\leq}$ is still a functional one supporting higher-order functions and not providing a notion of objects.

Polymorphism is supported by many object-oriented languages by parametrisation techniques for classes (C++ templates [Vandervoorde and Josuttis, 2002], Java Generics [Gosling

et al., 2005], Ada Generics, and so on). While this parametrisation is a common feature of functional and object-oriented languages, it is limited to types in object-oriented languages, but allows type constructors in functional languages like Haskell. There are experimental solutions to overcome the missing higher-order type constructors in object-oriented languages. In C++, *template template* parameters are possible but due to technical limitations rarely used. In Java, type constructor polymorphism was introduced experimentally, supplemented by a prototype compiler [Cremet and Altherr, 2008].

As both functions and objects seem to be valuable first-class elements of a programming language, there are many attempts to combine both language paradigms. On one side, the absence of first-class functions in object-oriented languages is a well known inconvenience and there are attempts to include first-class functions in object-oriented languages. These attempts range from workarounds and libraries [Läufer, 1995, Zalewski, 2008, McNamara and Smaragdakis, 2004, Abrahams and Gurtovoy, 2005] to completely new hybrid languages (sometimes referred as object-functional languages) like Pizza [Odersky and Wadler, 1997, 1998], Scala [Odersky et al., 2006], and F# and C# from .NET [Petricek and Skeet, 2009]. On the other side, subtyping and inheritance seem to be useful techniques for functional languages and there are attempts to bring them into functional programming. OCaml, for instance, implements objective ML, an object-oriented extension of ML [Remy and Vouillon, 1997], and O'Haskell is a programming language derived from Haskell by the addition of concurrent reactive objects and subtyping [Nordlander, 2002].

2.10 Types and Specification

There is a remarkable correspondence between type theory and logic, known as Curry-Howard isomorphism or Curry-Howard correspondence [Curry and Feys, 1958, Howard, 1980]. Briefly, it states that every type induces a logic proposition, and every logic proposition can be expressed as a type. Further, elements of a type are proofs that the type is inhabited by values and therefore proves that the proposition connected with the type is true. There are indeed types that have no values, for instance, the type of all integers that are even and odd. Such a type encodes the contradiction $n \in Z \wedge n \; mod \; 2 = 0 \wedge n \; mod \; 2 = 1$. Logic combinators correspond

Types	Logic
type T	proposition P_T
element of type T	proof of proposition P_T
pair type $T \times U$	conjunction proposition $P_T \wedge P_U$
sum type $T \mid U$	disjunction proposition $P_T \vee P_U$
function type $T \rightarrow U$	implication proposition $P_T \rightarrow P_U$

Table 2.2: Curry-Howard isomorphism

to certain types, yielding the correspondence summarised in Table 2.2.

The Curry-Howard correspondence is not limited to a particular type system and logic. In fact, it can be applied to a variety of type systems and logics. For instance, System F corresponds to a second-order constructive logic and System F_ω corresponds to a higher-order logic.

The Curry-Howard correspondence also justifies the idea of viewing types as specifications. The type of a computation, which is the type of the function it computes, induces a logical proposition that specifies the computation. When such a computation exists in form of an algorithm, this algorithm is a proof of the proposition that specifies the computation. Translating function types from one language into another transfers the specification, and, if there already exists a computation in the source language, ensures that there exists one in the target language.

2.11 Related Topics and Conclusion

In this chapter, we described the idea of types in computing and explained the benefits of type systems. We studied type systems on the basis of the lambda-calculus, which is the simplest form of a programming language. We showed that an untyped version of the calculus runs into problems that can be avoided when types are introduced. As a simple type system has limitations, we showed how it can be extended in two directions, with polymorphism, or with type constructors. The extension that combines both, yielding a polymorphic type system with type constructors and type constructor polymorphism, is what we consider a higher-order type system. If this type system is in addition of functional nature we call it a functional, higher-order type system.

The field of type systems is a wide one and there are several topics that were not mentioned in this chapter, but are loosely related to the thesis and which we shortly mention here.

The lambda-calculus can also be extended with existential types. While the polymorphic lambda-calculus allows universal quantification over types, it is also possible to introduce existential quantification over types. Such types are called existential types [Mitchell and Plotkin, 1988] and have proved useful in data abstraction and information hiding.

Closely related to existential types are higher-rank types [Jones et al., 2007]. It is a well-known restriction of the higher-order polymorphic lambda-calculus that higher-rank polymorphism [Leivant, 1983] is not supported. A type has rank n if the maximal depth of a universal quantifier in a domain type in the syntax tree of the type is n. Thus, a function type with concrete types such as `int -> int` has rank zero, and a polymorphic type such as `∀X. X -> X` has rank one. The function type `int -> (∀X. X -> X)` also has rank one, but the type `(∀X. X -> X) -> int` has rank two.
Very few programming languages support higher-rank types. One example is Haskell, where higher-rank types are available as extension of the type system and used in generic programming approaches such as scrap your boilerplate [Lämmel and Peyton Jones, 2003].

Another related topic is that of type inference. In typed programming languages (like the typed lambda-calculi), not every expression has to be annotated with a type. Type inference algorithms can automatically deduce and reconstruct the type of an expression in a programming language. The most prominent example is the Hindley-Milner type inference algorithm [Hindley, 1969, Milner, 1978], which is the base of the type system of many programming languages (Haskell, ML). C++ templates and Java Generics offer limited type inference, which enables the compiler to deduce the right type argument for polymorphic functions calls (but nor for polymorphic objects).

As already mentioned, dependent types are beyond the scope of this thesis. However, they recently became of interest to the computer science community, as they deepen the relation be-

tween types and specification [Altenkirch et al., 2006]. With dependent types, it is for instance possible to specify in the function signature that an algorithm that zips two lists into a list of pair expects two lists of the same length and produces a list of that length as result.

Chapter 3

Generic Software

In the previous chapter, we introduced type systems with different levels of abstraction, leading to the definition of higher-order functional types. As higher-order types depend on polymorphic λ-calculi, they can only be implemented in programming languages that provide support for parameterisation on entities other than values. This feature is called genericity, and the style of programming it induces is known as generic programming.

In this chapter, we give a short introduction to generic programming and show its relation to higher-order types. We explain the different kinds of generic programming and how generic programming can be supported by programming languages. The role of concepts in generic programming and the important idea of concept-controlled polymorphism are also discussed in this chapter.

In this thesis, generic programming with concepts will be used to lift functional higher-order types from the level of types to a higher level: that of concepts. Concepts specify operations and entities that have to be provided by types, and we will use them to characterise what makes a type a function type or a type constructor.

3.1 Generic Programming

Programming in general is about abstraction. Instead of writing the same code (with minor differences) several times, variable parts are abstracted from it and provided as parameters. Abstraction already starts at the level of values:

> "Procedural (or functional) abstraction is a key feature of essentially all programming languages. Instead of writing the same calculation over and over, we write a procedure or function that performs the calculation generically, in terms of one or more named parameters, and then instantiate this function as needed, providing values for the parameters in each case." [Pierce and Turner, 2000]

In that sense, every programming language that provides the possibility to define functions, provides a form of generic programming. Some low-level languages for direct machine programming, for instance Assembler dialects, do not support functions and are therefore completely free of generic programming elements.

However, when referring to generic programming, one often means the possibility to parameterise code on entities other than values. What kind of parametrisation is emphasised

depends on the actual programming community. Therefore, a unique and precise definition is lacking; different research communities use different definitions. For instance, Siek et al. [2002] emphasise that generic programming has a separation aspect:

> "Generic programming is a methodology for program design and implementation that separates data structures and algorithms through the use of abstract requirement specifications." [Siek et al., 2002]

In contrast, Gibbons [2007] states that the common property connecting the different kinds of generic programming is parametrisation of programs by non-traditional parameters, where the meaning of non-traditional parameters depends on the particular programming language. Therefore, we can say more generally that generic programming is about any form of parametrisation. What all definitions of generic programming also have in common is the general goal of making programs adaptable without loosing efficiency, and to achieve that by abstraction and parameterisation.

The major advantage of generic programming from a practical point of view is that it usually reduces the amount of code to be written, since genericity enables programmers to write programs that solve a class of problems once and for all, instead of writing new code for each instance of the problem [Backhouse et al., 1999]. Another advantage is that generic programming allows type-safe parametrisation, and thus avoids errors induced by the copy-and paste style of programming. Furthermore, generic programs have a great potential for reuse, since generic programs are natural candidates for incorporation in libraries [Backhouse et al., 1999].

3.2 Genericity

Genericity in programming languages is also called parametricity; it describes what kind of parameters are supported. Thus, the genericity available in a programming language determines what levels of abstraction are supported by that language. The following classification according to Gibbons [2007] distinguishes seven main styles of genericity and clarify the scope of the transformation we present in the upcoming chapters. The styles of genericity that Gibbons [2007] distinguishes are:

- Genericity by value
- Genericity by type
- Genericity by function
- Genericity by structure
- Genericity by property
- Genericity by stage
- Genericity by shape

Below, we describe these seven styles in detail. The examples in this subsection are given in a functional pseudo-code that is close to Haskell.

3.2.1 Genericity by value

Genericity by value corresponds to the nowadays fundamental constructs of functions and procedures. Instead of writing a computation like `(2 * 2 * 2)` for different values over and over again, one introduces one piece of code that is parameterised by the values the computation should operate on. In the example above, this parameterised code could be the following function:

```
1 cubic :: Int -> Int
2 cubic x = x * x * x
```

This functions has one parameter, `x`, standing for a value of type `Int`. The computation above can then be expressed as function invocation, `cubic(2)`, and it is obvious that this form of abstraction makes writing of code much more convenient. Therefore, genericity by value is nowadays provided by all high-level languages, but is still not available in direct machine programming and some Assembler languages.

3.2.2 Genericity by type

Genericity by type means that programs can also be parameterised by types instead of only by values. There are many functions that work uniformly over arguments of different types. For instance, most list functions do not dependent on the type of the elements stored in the list. A function that computes the length of a list could be implemented separately for every type:

```
1 list_length_int :: [Int] -> Int
2 list_length_int []     = 0
3 list_length_int (x:xs) = 1 + list_length_int xs
4
5 list_length_float :: [Float] -> Int
6 list_length_float []     = 0
7 list_length_float (x:xs) = 1 + list_length_float xs
8
9 ...
```

Genericity by type enables one to write one generic function that is not only parameterised by the input value, but also by the type of that value:

```
1 list_length_generic :: [a] -> Int
2 list_length_generic []     = 0
3 list_length_generic (x:xs) = 1 + list_length_generic xs
```

Instead of a concrete type, the signature of this version of the length function has a type parameter `a`, which can be replaced by any type. When the function is invoked, this parameter has to be specified, which can be done explictly or, as for instance in Haskell, implicitly.
Genericity by type was first available in the 1980s in Ada; nowadays type parameterisation is a standard feature in almost every programming language that has a static type system.

3.2.3 Genericity by function

Genericity by function means that programs can be parameterised by functions. Often, the definition of a function depends on another function. A function that performs an operation on all elements of a list is an example. For such a function different versions for different input

functions could be written. For instance, incrementing or decrementing a list element-wise could be implemented by the following two functions:

```
1 list_inc :: [Int] -> [Int]
2 list_inc []     = []
3 list_inc (x:xs) = (inc x):(list_inc xs)
4
5 list_dec :: [Int] -> [Int]
6 list_dec []     = []
7 list_dec (x:xs) = (dec x):(list_dec xs)
```

With genericity by function, one can write one generic function that is not only parameterised by the input value, but also by the function that should be executed for every element:

```
1 list_map :: (Int -> Int) -> [Int] -> [Int]
2 list_map f []     = []
3 list_map f (x:xs) = (f x):(list_map f xs)
```

Parametrisation by functions is a fundamental feature of functional programming languages. As already discussed in Chap. 2, in object-oriented and imperative languages, parametrisation by functions is generally not supported and has to be provided by workarounds. Genericity by function can be seen as a special case of genericity by value, where value has a type that is a function type. As genericity by type is available in languages that do not provide genericity by function, it is a key idea of the transformational approach of this thesis to transform genericity by function to genericity by type and genericity by structure (see below).

3.2.4 Genericity by concepts (Genericity by structure)

Another kind of genericity introduced by Gibbons [2007] is genericity by structure. As genericity by structure means that algorithms are separated from data structures through the use of abstract requirement specifications [Siek et al., 2002], and these abstract requirement specifications are known as concepts, we call this kind of genericity also genericity by concepts. This kind of genericity has it origins in Ada, and the most prominent library that is implemented based on it is the C++ Standard Template Library (STL) [Stepanov and Lee, 1994, Josuttis, 1999]. It is an extension of genericity by type with the possibility to put requirements on type parameters.

Concepts are implemented differently in different programming languages. In C++, they are implemented as C++ concepts [Gregor et al., Gregor and Stroustrup, 2006, Gregor et al., 2008], in Haskell as type classes [Wadler and Blott, 1989], and in Java and Scala as interfaces [Gosling et al., 2005, Odersky et al., 2006]. Even if they all implement the same idea of a specification of requirements on types, they are technically quite different. We will discuss these technicalities in Chap. 7.

Genericity by type was about parameterising functions by types in such a way that a function works uniformly for every type that can replace the type parameter. There are many functions that can be implemented that way, but there are other functions that need some functionality that depends on the actual type. For instance, the following generic function that sums all elements in a list

```
1 list_sum :: [a] -> a
2 list_sum []     = zero
3 list_sum (x:xs) = x + list_sum xs
```

is rejected by a type checker (more precisely, by a concept checker). In this example it is unclear, what `zero` and `+` actually mean. While operations with these names are available for numeric types such as `Int` or `Float`, their meaning is not clear when the function `list_sum` is, for example, called for a list of strings. In this case, we might have concrete semantics for these symbols in mind: `zero` could represent the empty string and the addition operation could represent string concatenation.

However, the compiler generally cannot deduce the operations that `zero` and `+` refer to for an arbitrary type. Thus, information on the concrete implementations of `zero` and `+` has to be provided explicitly in the code by introducing requirements on the type parameters. The `list_sum` function is extended with an requirement `Addable a`:

```
1 list_sum :: (Addable a) => [a] -> a
2 list_sum []     = zero
3 list_sum (x:xs) = x + list_sum xs
```

Here, `Addable` is a concept; we also say that the type parameter `a` is concept-constrained. The requirement `Addable a` states that `a` can only be replaced by those types that have the property to be `Addable`. What it means to be `Addable`, is defined by the concept `Addable`; in our example it means to provide an addition operator and a zero element. The concept definition, in this case implemented as a Haskell type class, reads as follows:

```
1 class Addable a where
2     (+)  :: a -> a -> a
3     zero :: a
```

If a type has the property to be `Addable`, this can be made explicit by declaring this type as a model of the concept. Using Haskell pseudocode, this is done with an instance declaration:

```
1 instance Addable Int where
2     (+)  = ...
3     zero = ...
```

Such a modelling declares how a type provides the necessary operations for a concept. The compiler then knows which function to invoke when a concept-constrained type parameter is instantiated with a concrete type.

Genericity by concepts unites genericity by value, genericity by type, and genericity by function, as a concept is a collection of functions, types, and values. The function `list_sum` above could also be implemented as a function parameterised by four parameters: a type, a function (the `+` operation), and two values (the `zero` value and the input list). Genericity by concepts, however, allows bundling these parameters of different types to a concept and thus adds a higher level of abstraction to a language.

Genericity by function as introduced before can be expressed by genericity by concepts. A function that is parameterised by another function can be implemented by a function that is parameterised by a type. This type should be a function type and a concept requirement can be used to make this explicit. We will describe this function concept in more detail in Chap. 5; it is an important vehicle for bridging functional and object-oriented programming.

Genericity by concepts is an emerging feature of programming languages; as mentioned above, it is available, among others, in Haskell, C++ (where it is an experimental feature), Java, and Scala. However, genericity by concepts is also used in languages that do not directly support concepts. The C++ Standard Template Library (STL) is organised using genericity by

concepts, even if earlier versions of C++ did not provide concepts. The concepts in the STL were stated implicitly, in the documentation; programmers who used the STL had to ensure that generic functions are only instantiated with types that are models of the required concepts, otherwise compilation could fail because the compiler would not be able to find the appropriate functions.

3.2.5 Genericity by property

Genericity by property extents genericity by concept by augmenting concept requirements by logical constraints. Concepts like `Monoid`, `Functor`, or `Monad` come with logical properties, but also the type class `Addable` from above can be augmented, for instance, with the following constraints (stated with the keyword `axiom` in our pseudo-code, which is not available in Haskell):

```
1 class Addable a where
2     (+)  :: a -> a -> a
3     zero :: a
4     axiom :: for all x: x + zero == x
5     axiom :: for all x: zero + x == x
6     axiom :: for all x y z : (x + y) + z == x + (y + z)
```

These properties state that the addition operation is associative and the zero element acts as a neutral element (in mathematical terms, the type `a` is a monoid). Such properties are important for the mathematical structure behind the concept, and sometimes algorithms depend on them. There is almost no programming language where these axioms can be expressed at the level of language syntax. One exception are C++ concepts, where the `axiom` keyword is available to state logical properties in concepts. Other exeptions are languages that are based on theorem proof systems, such as Agda and Idris. Even where such properties can be stated in the code, it is generally impossible for a compiler to check whether a type that is a model of a concept fulfils the axioms stated in the concept. The axioms can, for instance, be undecidable.

However, if such properties can be stated in the code they can be used to increase the usability of concepts. First of all, they serve as an additional documentation of the concept. Secondly, the compiler can check such properties at least for syntactic correctness. And thirdly, they can serve as a base for probabilistic testing during compilation [Bagge et al., 2008, Bagge and Haveraaen, 2009]. Such probabilistic testing is also available in Haskell [Claessen and Hughes, 2000], but there, the possibility to express axioms is not integrated in the type system.

3.2.6 Genericity by stage

Genericity by stage means that programs can be parameterised by programs themselves. The term covers different kinds of metaprogramming, that is, programs that construct or manipulate other programs. Genericity by stage is only available in a few programming languages and language extensions, as, for instance, in C++ (templates), Haskell (template Haskell), ML (MetaML), and OCaml (MetaOCaml). As genericity by stage is the only form of genericity that does not involve types and type systems, it is beyond the scope of this thesis and we refer to the literature [Gibbons, 2007, Abrahams and Gurtovoy, 2005] for further explanation.

3.2.7 Genericity by shape

Genericity by shape, also called datatype genericity, means that programs can be parameterised by the shape of data types. Consider, for instance, the following functions `list_map` for lists and `tree_map` for trees with empty leaves and data stored in inner nodes (`data Tree a = Leaf | Inner (Tree a) a (Tree a)`):

```
1 list_map :: (a -> b) -> [a] -> [b]
2 list_map f [] = []
3 list_map f (x:xs) = (f x):(list_map f xs)
4
5 tree_map :: (a -> b) -> Tree a -> Tree b
6 tree_map f Leaf = Leaf
7 tree_map f (Inner left x right) = Inner (tree_map f left) (f x) (tree_map f right)
```

These two mapping functions are rather similar: both have a base case for the simple constructor and a recursive case for the recursive constructor. This parallelism is due to the similarity of the data type definitions: both are recursive data types with two mutually exclusive constructors. Genericity by shape is exactly about abstracting over such commonalities.
The common shape of these types can be captured by abstracting over the property that they are recursive data types. One of the major results of category theory is that recursive types can be described as fixpoints of regular functors [Lambek, 1968]. This result can be used in programming by defining a general fixpoint type and shape functors for recursive data types. Shape functors are regular functors that define the structure of a data type. Thus, we can define a generic `map` function in terms of that fixpoint type. This `map` function works for all recursive data types that are defined in terms of the general fixpoint type:

```
1 data Fix s a = In (s a (Fix s a))
2 out :: Fix s a -> s a (Fix s a)
3 out (In x ) = x
4
5 data ListF a b = NilF | ConsF a b
6 type List a = Fix ListF a
7
8 data TreeF a b = LeafF | InnerF b a b
9 type Tree a = Fix TreeF a
10
11 map :: Bifunctor s => (a -> b) -> (Fix s a -> Fix s b)
12 map f = In . bimap f (map f ) . out
```

As a data type generic library is one example of the application of our transformation, we will explore this approach in more detail in Chap. 9.
Data type genericity is a term that refers to any parametrisation of data types and functions by shape, and there are other possibilities to capture the shape of data than the one sketched above [Rodriguez et al., 2008, 2010]. Those approaches usually make use of higher-order type constructors, thus require a higher-order, typed language. As many languages still do not support higher-order type constructors and thus do not provide the possibility of data type genericity, it is one idea of our transformational approach to turn data type genericity into genericity by type and genericity by concepts (as we also do with genericity by function). Thus, we are able to implement libraries that are based on genericity by shape in languages like C++, which do not provide genericity by shape.

3.3 Polymorphism

Polymorphism is a concept related to genericity, providing another classification of adaptable software. While genericity concentrates explicitly on parametrisation (on what structures can be parametrised by), polymorphism emphasises the technical dimension: How can a function behave differently for different types? Polymorphism is not identical to type parametrisation: as we will explain below, there are ways to make a function behave differently for different types that do not require type parametrisation.

A classification of different kinds of polymorphism is given by Cardelli and Wegner [1985], who distinguish between universal polymorphism and ad-hoc polymorphism (see Fig. 3.1). Universal polymorphism is provided using properties of the type system of a language; thus, universal polymorphism embeds genericity into the type system of a language. In contrast, ad-hoc polymorphism is polymorphism that is not provided by the type system.

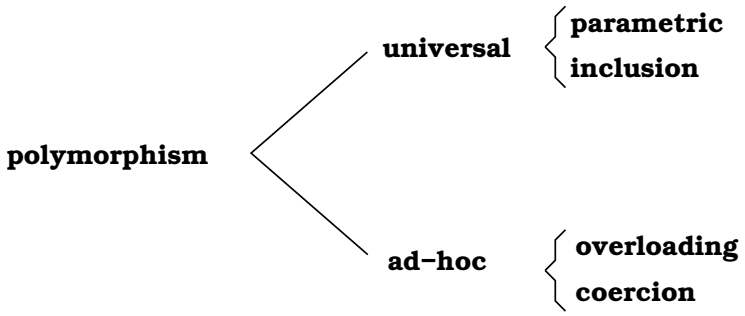

Figure 3.1: Kinds of polymorphism (from [Cardelli and Wegner, 1985])

3.3.1 Universal polymorphism

Universal polymorphism is a fundamental feature of the type system [Cardelli and Wegner, 1985]; universally polymorphic functions usually work on an infinite range of types. According to Cardelli and Wegner [1985] universal polymorphism can be further divided into parametric polymorphism and inclusion polymorphism:

- Parametric polymorphism means that a generic function is implicitly or explicitly parameterised with a type. The function works uniformly for all types, as there are no type-specific operations involved. For instance, a function that computes the length of a list can be implemented in Haskell as follows:

```
1 list_length :: [a] -> Int
2 list_length []     = 0
3 list_length (x:xs) = 1 + list_length xs
```

  This function is parameterised on a type parameter `a`, and it behaves equally for all types as no specific properties are used of the types that `a` represents.

- Inclusion polymorphism means that types are arranged in a type hierarchy. Every type has a supertype (except for the most general type there is), and a value of a specific type

is simultaneously a value of all its supertypes. Thus, functions that take arguments of some type work as well on arguments of its supertypes. Inclusion polymorphism is a key feature of object-oriented languages. Typical applications of inclusion polymorphism are graphical user interfaces and window types. Basic functions like resizing and moving are implemented for a general window type, specialised window types are subtypes of that general window type and can therefore be used as arguments for the basic window functions.

Parametric polymorphism and inclusion polymorphism do not exclude but complement each other. Ada was the first language that offered both, as discussed by Meyer [1986], who calls parametric polymorphism genericity and inclusion polymorphism inheritance. Java and C++ also support both kinds of polymorphism.

3.3.2 Ad-hoc polymorphism

In contrast to universal polymorphism, ad-hoc polymorphism is polymorphism that is not integrated in the type system. Ad-hoc polymorphism is a technical feature of a programming language; it can be seen as a dispatch mechanism: control moving through one named function is dispatched to various other functions without the need to specify the exact function being called [Cardelli and Wegner, 1985]. This kind of polymorphism is divided into overloading and coercion:

- Overloading allows multiple functions taking different types to share their name. The compiler or interpreter automatically chooses the right function to call depending on the type of its actual arguments. Overloading is in many languages used for arithmetic operators as + or *, where different versions for different basic types (integer numbers, floating-point numbers) exist, which take advantage, e.g., of optimised processor operations for these types.

- Coercion means that a function has a unique name, but semantic conversions (such as type casts) are applied to arguments whenever necessary. In coercion, there is just one function, but it might behave differently for different types. In C++, for instance, coercion makes it possible to call functions for floating-point numbers also with integer arguments. Using coercion, the compiler automatically replaces the arguments of type `int` with arguments of type `float`.

While parametric polymorphism is used to implement functions that work for infinitely many types simultaneously, ad-hoc polymorphism is used to implement functions that work differently for different types and might be not implemented for some types. For instance, a function that computes the length of a list can be defined for any type of elements stored in the list and carries out the same computation for every type; therefore, it is implemented using parametric polymorphism. In contrast, an addition operation is not meaningful for all types of arguments. While it is meaningful to define an addition for any numeral type and even for strings, an addition for, say, widgets does not seem to be quite useful. Thus, a polymorphic addition operation will usually be defined using ad-hoc polymorphism.

3.3.3 Concept-controlled polymorphism

Overloading can be combined with parametric polymorphism, this combination is sometimes called concept-controlled polymorphism [Järvi et al., 2003b]. The function `list_sum` from above is in fact a concept-controlled polymorphic function:

```
1 list_sum :: Addable a => [a] -> Int
2 list_sum [] = 0
3 list_sum (x:xs) = a + list_sum xs
```

On the one hand, this function has a type parameter, which indicates that it is parametrically polymorphic. However, as we have seen, not every type can be used to replace the type parameter `a`. Only types that provide an addition operator are allowed, as ensured by the requirement `Addable a`. On the other hand, the different versions of the + operator have to be provided by overloading, and concept mappings are used to resolve the overloading for a specific type. Such a combination of type parameters and concepts is called concept-controlled polymorphism.

Concept-controlled polymorphism also includes the possibility to overload functions on the base of different concepts. That means, functions might have the same signature except for different requirements. The compiler selects the most appropriate version based on the concept mappings that are available for a concrete type. Concept-based overloading is for instance possible with C++ concepts, but not with Haskell type classes.

For the rest of this thesis, concept-controlled polymorphism is the most important form of polymorphism. We will use concept-controlled polymorphism to emulate different kinds of genericity. That will enable us to provide high-level genericity in very different languages, as concept-controlled polymorphism is provided by many languages from different families. We will provide examples in Chap. 7.

3.3.4 Genericity and polymorphism

Genericity and polymorphism emphasise different aspects of generic programming. While genericity concentrates on the theoretical dimension of generic programming, polymorphism captures the technical aspects. Genericity distinguishes different entities that can be parameterised about, polymorphism captures different programming language techniques that can make a function work on arguments of different types. The different kinds of genericity can be achieved by using different kinds of polymorphism. This correspondence is, however, not unique. Genericity by type can, for instance, be achieved by parametric polymorphism as well as by overloading [Gibbons, 2007]. Genericity by concepts captures a mixture between parametric polymorphism and overloading: While functions that are generic by concepts usually have type parameters, the type-dependent functionality is provided by overloading. As described before, this mixture is sometimes referred to as concept-controlled polymorphism. Genericity by function can be implemented by higher-order functions but also by means of concept-controlled polymorphism: the function type is just a special type providing one special operator—the application operator. This application operator is overloaded with different implementations—one for every function that is defined. Genericity by property is of theoretical nature, as the properties the code is parameterised on, in most programming languages can not be expressed. If it is simulated, for instance, by comments in the code that state these properties, this is usually done using concept-controlled polymorphism. Genericity by shape is accomplished by parametric polymorphism and overloading.

The use of overloading in implementing different kinds of genericity already suggests that the most important generic programming technique is overloading, and in fact, most generic functions need some overloaded functionality of their type arguments. A generic sorting function, based on element comparison, for instance, needs overloaded comparison operators. Pure parametric polymorphism, in contrast, is used rather rarely. One class of functions that can be implemented with pure parametric polymorphism is that of functions on generic containers, where no properties of the element type are needed.

3.4 Related Topics and Conclusion

In this chapter, we explained the idea of the generic programming paradigm. Even if this is a widely used term in computer science, a precise definition lacks. We showed how the different definitions have some idea of parameterisation and abstraction in common. Afterwards, we explored two different aspects of generic programming that are important for our transformational approach. While genericity emphasises the theoretical dimension of generic programming, polymorphism captures the technical aspects.

A related programming paradigm is that of generative programming [Czarnecki, 1998]. It emphasises automated generation of specialised versions of software components, based on a model of a software family and a set of customisation requirements. The main difference between generative programming and generic programming is that the former encapsulates differences within a family of abstractions in customisable features, whereas the latter encapsulates their commonalities in concepts. Generative programming and generic programming sometimes meet each other in generic libraries: while algorithms in such libraries are usually generic, library data structures have a generative character, as specialised versions can be generated automatically.

Object-oriented programming is a related paradigm, which builds on connecting data and functionality in objects. A program is a collection of such objects, which interact by invoking each other's functions. Object-oriented programming realises subtype (or inclusion) polymorphism, while generic programming realises parametric polymorphism. Object-oriented polymorphic function calls are resolved at runtime, while in generic programming such calls are resolved at compile time. Nevertheless, object-oriented programming and generic programming can complement each other [Meyer, 1986], and modern object-oriented programming languages like C++, Java, and Scala also provide generic programming capabilities.

Even if generic programming is subject to intensive research, a universal theory of it is still missing. Early generic languages, such as PolyP [Jansson and Jeuring, 1997], have a strong theoretical background based on the algebra of programming [Bird and de Moor, 1997], while later work emphasised the close connection between parameterised data types and category theory [Dos Reis and Järvi, 2005]. Recent work investigates a general categorial foundation of generic programming [Hinze and Wu, 2011].

Chapter 4

Program Transformation

Program transformation is a key operation in computer science, used in many areas of software engineering. A program transformation system is any system that takes a computer program and generates another program; every compiler is, for instance, a program transformation system consisting of a sequence of program transformations. Other applications of program transformation systems include program synthesis, optimisation and refactoring, but also reverse engineering. Even documentation systems, which generate documentation from code, are program transformation systems.

Program transformations are often used for translating programs from a language that is close to natural language, and can be easily understood by programmers, into a language that is machine-executable. Other typical applications are tasks that are in a certain sense mechanical and can therefore easily be automated (such as documentation generation). Our transformation is from the first category: it transforms functional higher-order function signatures, which are easily understandable by programmers, into concept-constrained function object declarations. Even if the latter are harder to read (see Fig. 1.2), they provide the possibility to express functional higher-order function signatures in languages that do not provide functional features such as genericity by function. Expressions that use such features are transformed into expressions based on genericity by concepts. Thus, the result of the transformation can be used in every language that provides concept-controlled polymorphism.

This chapter provides an overview of program transformations and explains a special and important program transformation: defunctionalisation, the process of removing higher-order functions from a program. This program transformation is central to all functional programming languages: whenever functional programs are compiled or interpreted, higher-order functions have to be replaced by first-order functions. Our transformation introduces a defunctionalisation of functional higher-order function signatures by replacing higher-order functions by objects and type parameters that are constrained by a function concept. Furthermore, the chapter outlines the main idea of our transformation and describes how it is embedded in a complete program transformation system and how we position our transformation in the landscape of program transformations.

4.1 Overview

As already mentioned, there is a multitude of application areas of program transformation systems. For an overview, we present a classification motivated by Visser [2001, 2005]. Program transformation systems are divided into two classes, distinguished by the relation between source and target language. If the latter is a subset of the former, the program transformation system is a rephrasing system, otherwise a translation system. Both classes are divided into several sub-classes, where a major criterion is the abstraction level of the source and target language. For translation systems, the following taxonomy results:

- **Program synthesis**
- **Program migration**
- **Reverse engineering**
- **Program analysis**

Program synthesis is a program transformation that performs a translation of a program, usually from a language with a higher level of abstraction into a language with a lower level of abstraction. A common idea for program synthesis systems is that high-level design is traded for increased low-level efficiency. Program synthesis can target machine-executable code, it is then called compilation [Aho et al., 2006], or target another language at the level of non-executable code, in which case it is called refinement [Morgan, 1990]. Compilation of a program is a process that is usually achieved in several phases. First, the high-level language is translated into a non-executable intermediate representation. Afterwards, this intermediate representation is translated into a sequence of machine instructions. In refinement, the source program is usually called a specification, the target program is usually called an implementation. The translation rules are chosen so, that each rule preserves the specified properties. Thus, the resulting implementation as a whole satisfies the specification. In either case, a correct synthesis transformation always preserves the semantics of a program.

Program migration is a program transformation that aims at adapting a program from one programming language to another one that has the same level of abstraction. Often, both languages are different dialects or versions of the same language. For instance, the migration from an older Java version to a newer one or the migration from a C++ program for a GNU compiler to a C++ program for a Microsoft compiler can be done by migration transformation. If the target language is another version of the source language, it could be a newer or an older version of the source language yielding a forward migration in the first case and a backward migration in the second case. Program migration is a transformation that generally preserves semantics.

Reverse engineering goes the opposite direction to program synthesis [Chikofski and Cross, 1990]. A program in a lower-level language is translated into a program in a higher-level language. Like program synthesis, it serves several purposes. Decompilation is the opposite to program compilation; it is a transformation that translates an executable program into high-level code. The resulting and the original program, however, are not necessarily identical: given a program in a high-level language, a compilation followed by a decompilation will usually not result in the original code. Other aspects of reverse engineering are architecture extraction, documentation generation, and software visualisation. In general, reverse engineering transformations are not semantics-preserving.

Program analysis, finally, is a transformation that reduces a program to one aspect of interest. Thus, it can be considered a transformation to a specific sublanguage of the source language. Program analysis might increase or decrease the complexity of the source program. For instance, a program with array bounds checks can be transformed to a program without array bounds checks, or, a program with exceptions can be transformed to a program without exceptions. As the purpose of program analysis is to ascertain information about a program [Reps, 1998], program analysis transformations should preserve the semantics of the property of interest (for instance, if the original program exhibits overflow behaviour, the program transformation must capture that).

Much as translation systems, rephrasing systems can also be further classified. The major classification criterion here is the purpose of the transformation, leading to the following distinction:

- **Program normalisation**
- **Program optimisation**
- **Refactoring**
- **Program renovation**

Program normalisation is a rephrasing transformation that converts a program into a program in a sublanguage with lower syntactic complexity. Normalisations that aim at removing high-level constructs (syntactic sugar) by replacing them with more fundamental constructs are sometimes called desugaring. Defunctionalisation, which will be described in detail below, is a normalisation transformation. A more general kind of normalisation is simplification, which reduces a program to some kind of normal form without necessarily removing syntactic-sugar language constructs. Program normalisation transformations are necessarily semantics-preserving.

Program optimisation is a rephrasing transformation that aims at improving the performance of a program. Performance improvements include improvements in runtime, memory use, or energy use. There are numerous examples of optimisations, such as fusion, inlining, loop unrolling, constant propagation, and elimination of dead code. Ideally, optimisation transformations preserve the semantics of a program, but in practice they do not always do. For example, the highest optimisation level (`-O3`) for the Gnu C++ compiler gcc [GCC] changes the semantics of floating-point computations.

Refactoring is a transformation that also aims at improvement of a program, but in contrast to program optimisation, it concentrates on improving code quality instead of performance. Thus, the goal of refactoring is to improve the structure of a program such that the code becomes easier to understand. Fowler [1999] defines refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour." Thus, refactoring should be semantics-preserving.

Finally, *program renovation* is a rephrasing method that differs from the others as it explicitly changes the semantics of a program. An example is the conversion of a program from computing with one currency to computing with another currency.

It should be noted that other taxonomies exists. Stuikys and Damasevicius [2002] provide an overview of different attempts to define taxonomies of program transformation. Classifica-

tion criteria include the degree of automation or the way transformation rules are expressed. However, we think the taxonomy above gives the best overview of program transformations.

4.2 Defunctionalisation

A challenge in every compiler or interpreter that processes a functional language, is to lower higher-order functions to first-order functions. As Strachey [2000] points out, functions are second-class values in first-order programs, and first-class values in a higher-order program. Second-class values means that they are denotable but not referable as function arguments, whereas first-class values are both. Higher-order functions are actually just syntactic sugar to provide a higher-level of abstraction; in compiled, machine-executable code they cannot be processed. Defunctionalisation is a compile-time transformation that eliminates higher-order functions, replacing them by a single first-order function. One of the challenges for this transformation comes from the fact that higher-order function often are anonymous functions that cannot be referred to with a function name.

There are several methods to transform higher-order functions in first-order languages [Danvy and Nielsen, 2001], summarised under the term defunctionalisation. The first description of this technique was given by Reynolds [1972]. He observed that a program at compile time contains only finitely many higher-order function calls, thus each of them can be replaced by a function call without higher-order function arguments.

In principle, defunctionalisation introduces a *tag type* for every higher-order function. This tag type is defined as an algebraic datatype, its different constructors encode the different argument functions to higher-order function calls. An associated apply-function encapsulates the function calls themselves. This apply function expects the same inputs as the original higher-order function, but in addition an argument of the tag type. The different computations are then encoded using pattern matching over the tag type argument. For instance, in the following Haskell program, there are two higher-order function calls in lines 10 and 11:

```
1  f1 :: (Int -> Int) -> Int
2  f1 f = f 1 + f 10
3
4  inc :: Int -> Int
5  inc x = x + 1
6
7  dec :: Int -> Int
8  dec x = x - 1
9
10 print ( f1 inc )
11 print ( f1 dec )
```

The higher-order function `f1` is called with two different arguments: `inc` and `dec`. To defunctionalise the program, the two calls `f1 inc` and `f1 dec` have to be transformed into calls that do not contain `inc` or `dec` as arguments. This is done by the following transformation procedure: the transformation starts by introducing a new tag type, in this case called `F1_HOF1_Tag`. It has two constructors, `Inc` and `Dec`, and encodes the two different arguments of the higher-order function call:

```
1 data F1_HOF1_Tag = Dec | Inc
2
3 apply :: (F1_HOF1_Tag, Int) -> Int
```

```
4 apply (Inc,x) = x + 1
5 apply (Dec,x) = x - 1
6
7 f1_def :: F1_HOF1_Tag -> Int
8 f1_def t = apply (t, 1) + apply (t, 10)
9
10 print ( f1_def Inc )
11 print ( f1_def Dec )
```

The transformation also introduces an `apply`-function, which inlines the function bodies of the two higher-order function arguments. It works by pattern matching and performs the appropriate computation depending on the tag type input. Finally, the function `f1` is transformed into `f1_def`, which has as domain no longer a higher-order function but the tag type instead. With the steps just described, all higher-order functions (as well as anonymous functions) can be eliminated.

In the simply-typed λ-calculus, defunctionalisation is known to be type-preserving. Defunctionalisation becomes difficult in the case of polymorphic higher-order functions. As obvious in the example above, it is not clear what the type of the `apply` function is. The tag type then has to encode functions of different type, thus the `apply` function has to have a polymorphic type, but it is not clear how to ensure that the type `apply` is applied to match the function encoded in the tag type. A solution is to use guarded algebraic data types to encode the `apply` function [Pottier and Gauthier, 2004]. In this setting, it can be shown that polymorphic defunctionalisation is a type-preserving transformation [Pottier and Gauthier, 2004].

There are many variants of defunctionalisation, for instance *firstification* [Nelan, 1994], *higher-order removal* [Chin and Darlington, 1996], and *closure conversion* [Reynolds, 1972]. While serving the same purpose, they differ in technical details making them more or less suitable in certain settings. For instance, closure conversion can be seen as a special case of defunctionalisation where instead of a tag a code pointer is used to implement the apply function. An advantage of defunctionalisation over closure conversion is that the apply function in case of defunctionalisation is realised by a direct jump, whereas closure conversion is based on an indirect jump.

4.3 Our Transformational Approach

The approach we present in the next chapters defines a program transformation system. Our transformational approach differs from most other approaches insofar that we transform type declarations and not actual code that contains value expressions. The type declarations we transform are function signatures with functional higher-order types in the sense of Chap. 2: they are polymorphic, with higher-order functions and higher-order type constructors. In addition, they can contain constraints on type parameters. We have already argued in Chap. 1 why this restriction makes sense.

The target type system is not functional: higher-order functions and higher-order type constructors are no longer supported. It still provides constrained polymorphism, so higher-order constructs can be expressed with constraints on type parameters. We capture the essentials of functional type systems on the one hand and non-functional but concept-enabled type systems on the other hand in two kernel languages, the functional type kernel language (FTKL) and the concept type kernel language (CTKL). They consist only of function signature declarations and

do not contain any expression that evaluates to a value. Our transformation is defined between these two kernel languages. The whole transformation is summarised in Fig. 4.1. The picture

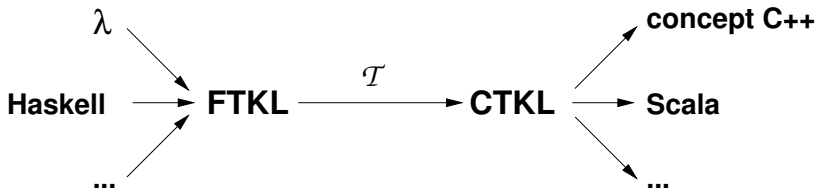

Figure 4.1: Transformation $\mathscr{T}$ and language bindings

makes explicit that the approach actually implements three stages of processing, each of them defining a separate transformation. The first transformation, called frontend, is a transformation between the functional languages and FTKL. As the applications presented in this thesis only use Haskell as functional prototyping language, and, as we will see, FTKL itself resembles a subset of Haskell, the frontend transformation is in our case trivial and therefore neither formalised nor presented.

The second transformation is the core transformation $\mathscr{T}$ and will be described in detail in Chap. 6. In principle, $\mathscr{T}$ realises a defunctionalisation transformation, performed on function signatures at the level of functions and type functions. In terms of the classification above, this transformation is a program normalisation that lowers higher-order constructs (functions and type constructors) from FTKL to CTKL.

The third transformation, called backend, transforms CTKL expressions into expressions of real programming languages. This transformation depends on the particular target language, and we will give examples of transformations to different languages in Chap. 7.

4.4 Related Topics and Conclusion

In this chapter, we gave a short overview of program transformations in order to characterise our transformational approach and to position it in the landscape of transformation systems. We explained a special program transformation in detail, namely defunctionalisation. Afterwards, we sketched our approach, which, in principle, is a defunctionalisation transformation. In contrast to other approaches, our defunctionalisation works on function signatures and therefore acts at the level of types. Further, it works not only on functions but also on type constructors.

The main applications of program transformation systems are in the field of compiler and interpreter construction, where it is applied in many stages of code processing, but there are several other application areas that were not mentioned in this chapter.

Another domain for transformations of functional programs is hardware design or hardware-software codesign. As in the model-based approach that motivated this thesis (Chap. 1), functional languages are used to specify, prototypically implement, and test hardware designs. Due to their clean notation and their abstraction from technical details, functional languages are a suitable tool for specification and prototyping of hardware. Thus, in hardware design the same advantages of functional programming are appreciated as in functional prototyping for scientific models.

Prominent examples of hardware specification languages based on functional programming languages are *SAFL* [Sharp and Mycroft, 2001] and *SASL* [Frankau and Mycroft, 2003]. For these two languages, there are source-to-source compilers available [Mycroft and Sharp, 2001]. *Lava* [Bjesse et al., 1999], based on Haskell, is an example of embedding a hardware specification language in an existing functional language. Lava focuses on the structural representation of hardware and provides functions for related connection patterns. There are transformation tools to translate Lava descriptions into VHDL and there exist interfaces with external theorem provers to verify Lava-designed circuits. *Hydra* [O'Donnell, 2002] is another hardware design language based upon Haskell. Hydra implementations can be transformed into hardware by using functional equational reasoning. Another hardware modelling language based on a functional language is *Hardware ML* (HML) [Li and Leeser, 1995], which builds upon Standard ML and is mainly an improved version of the standard hardware design language VHDL. It comes along with a transformation of language constructs into the corresponding VHDL constructs.

Since all these languages are ultimately translated to C, they provide evidence that a transformational approach as used in this thesis is indeed useful.

Chapter 5

Kernel Languages

As outlined in the previous chapter, we aim at a transformation that converts function signatures. The given function signatures are expressed in a functional, higher-order, typed language as for instance Haskell, and are transformed to function signatures in a higher-order, typed language that is not functional and provides concept-controlled polymorphism.

As this transformation should not depend on particular languages, we abstract the essentials of function signatures in both language families in two kernel languages. These two languages capture the necessary constructs to express higher-order types in a functional language or in a nonfunctional language with concepts. In this chapter we introduce these two languages, define their syntax, and present a semantics for them.

A common characteristics of both languages is that one cannot define types. For example, expressions like `Int` and `Float` are just strings; they have an intended meaning, but that meaning is beyond the scope of the two kernel languages. Types like `Int`, the type of integer numbers, depend strongly on the actual implementation language and the machine the program is compiled on. An integer type in Haskell defines another range of values and another bitwise representation as an integer type in ML, C++, or Java.

As a consequence, it is not possible to define a semantics for types in the usual way. Instead, we aim at a higher level of abstraction when defining a semantics of higher-order, typed function signatures, based on the structure induced by a function signature. Kinds, which are types of types, are useful for such a definition, and we use kinds to define the kind structure as a semantic property of FTKL and CTKL expressions.

5.1 The Functional Type Kernel Language (FTKL)

The functional type kernel language (FTKL) resembles a simplified subset of the functional programming language Haskell, as it comprises functional-style function signatures and concept definitions, where the latter are restricted to concepts that provide only functions as associated entities. However, in functional programming, other associated entities are rarely used, so that this restriction is no major limitation.

The language provides type constructors, type variables, contexts, and type applications to construct complex parameterised types. FTKL, however, does not contain any expression that evaluates to a value and not a type—values are completely unknown in FTKL. Therefore, we call a sequence of FTKL expressions not a program, but an FTKL specification. In the following

subsections we define and explain the syntax and the semantics (kind structure and kind rules) of FTKL.

5.1.1 Syntax

The BNF of the FTKL syntax is presented in Fig. 5.1. An FTKL description is a collection of function signatures and concept definitions, where the latter themselves are collections of function signatures. The root node of the grammar, the node <spec>, accepts a set of function signatures and concept definitions.

Examples of function signatures in FTKL are listed in Fig. 1.1. Function signatures consist of an identifier followed by a double colon, an optional context, and a type. This type is restricted: only function types (<tyfun>) are allowed. Concept definitions (<conceptdef>) begin with the `class` keyword followed by an identifier and one or more type variables. The keyword `class` is chosen for pragmatical reasons, it keeps FTKL close to Haskell. After the `where` keyword the function signatures are listed. The types in FTKL are similar to those from

```
<spec>          :: <spec_item>+

<spec_item>     :: <conceptdef> | <fctsig>

<conceptdef>    :: "class" <conceptname> <tyvar>+ "where" {<fctsig>}
<conceptname>   :: <regular_string>

<fctsig>        :: <ident> "::" [ <context> "=>" ] <tyfun>

<ident>         :: <regular_string>

<context>       :: <requirement> {"," <requirement>}
<requirement>   :: <conceptname> <type>+

<kind>          :: "*" | <kind> "->" <kind> | "(" <kind> ")"

<type>          :: <tyfun> | <tyapp> | <tyvar> | <tyvar_kinded> |
                   <tycon> | <tycon_kinded> | <tytuple> | "("<type>")"
<tyfun>         :: <type> "->" <type>
<tyapp>         :: <type> <type>
<tytuple>       :: "(" <type> {"," <type>}+ ")"
<tyvar>         :: <string_beginning_lower_case>
<tyvar_kinded>  :: <tyvar> ":" <kind>
<tycon>         :: <string_beginning_upper_case>
<tycon_kinded>  :: <tycon> ":" <kind>
```

Figure 5.1: The syntax of FTKL

the higher-order polymorphic lambda-calculus: types are built up from type constructors and type variables. Type variables begin with a lower case letter, type constructors with an upper

```
fold :: Bifunctor s => (s a b -> b) -> Fix s a -> b
```

```
<spec>
<spec_item>
<fctsig>
  <ident> "fold" "::"
  <context>
    <requirement>
      <conceptname> "Bifunctor"
      <type>
        <tyvar> "s"
  "=>"
  <tyfun>
    <type>
      <tyfun>
        <type>
          "("
          <type>
            <tyfun>
              <type>
                <tyapp>
                  <type>
                    <tyapp>
                      <type>
                        <tyvar> "s"
                      <type>
                        <tyvar> "a"
                  <type>
                    <tyvar> "b"
              "->"
              <type>
                <tyvar> "b"
          ")"
        "->"
        <type>
          <tyfun>
            <type>
              <tyapp>
                <type>
                  <tyapp>
                    <type>
                      <tycon> "Fix"
                    <type>
                      <tyvar> "s"
                <type>
                  <tyvar> "a"
            "->"
            <type>
              <tyvar> "b"
```

Figure 5.2: Derivation tree of the fold function signature from the Origami library in FTKL

case letter, so that they are syntactically distinguishable: `Int` is a type constructor in FTKL and `int` is a type variable. Both type constructors and type variables can be optionally annotated with a kind. Using the notation from Sect. 2.6, we can, for instance, explicitly state that `Int` is a proper type by writing `Int : *`.

The kinds that FTKL accepts are denoted in the node `<kind>`, they are similar to those introduced in Sect. 2.6. There is a basic kind `*`, one binary kind constructor `->`, and parentheses. Functions at the level of values (`Int -> Int`) and the level of types (`* -> *`) use the same function type symbol. This convention makes the functional nature of FTKL once more explicit: Kinds are functions, just on a higher level. However, in the higher-order polymorphic lambda-calculus, functions and kinds have a different syntactic notation. We stick to the notation $* \Rightarrow *$ wherever it is possible and use the FTKL notation just in FTKL code.

Type application in FTKL is written in prefix notation, thus, the application of a type `t1` to another type `t2` is denoted `t1 t2` (node `<tyapp>`). The grammar contains additional nodes for special type constructors, `<tyfun>` and `<tytuple>`. These nodes provide special syntax for the function type constructor `->` and for tuple type constructors. The function type constructor is a binary type constructor and could be expressed using type application. For example, a function type from `Int` to `Int` could be denoted by `(FunType Int) Int`. Yet, this notation becomes rather complicated for complex function types. The `->` notation shortens function type signatures and makes them much easier to read. In the same sense, the node `<tytuple>` encompasses a whole family of types constructed by the type constructors with different arities. This special syntax makes the denotation of tuples much more convenient.

There are two conventions that are not visible in the grammar in Fig. 5.1. The function type constructor associates to the right, that is, `f -> g -> h` is interpreted as `f -> (g -> h)`, and type application associates to the left, that is, `T1 T2 T3` is interpreted as `(T1 T2) T3`.

Figure 5.2 shows an example of a valid expression and its derivation using the grammar in Fig. 5.1. The function used in this example is the `fold` function for fixpoint datatypes, it is part of the so called Origami-library and we will describe this function in detail in Chap. 9. This form of presenting a syntactic derivation is called derivation tree.

5.1.2 Kind rules

In typed programming languages, every expression that evaluates to a value has to be well-typed. In FTKL, expressions do not evaluate to values but to types, and those expressions do not have a type but a kind. Accordingly, FTKL programs are checked for well-kindedness instead of well-typedness. As is obvious from Fig. 5.1, kinds in FTKL have a simple structure. Fig. 5.3 shows the kinding rules for FTKL. They follow the productions in the grammar. The rule (K-TyClass) declares a type class definition to be of kind `*` if all its type signatures have kind `*`. The rule (K-TySig) declares that a function signature has kind `*` if the function type it declares has kind `*`. (K-TyFun) says that if $\texttt{t}_1$ has kind `*` and $\texttt{t}_2$ has kind `*` the function type $\texttt{t}_1$ `->` $\texttt{t}_2$ also has kind `*`. The rule for the tuple type constructor works similarly (K-TyTuple). The rule (K-TyApp) evaluates the kind of a type application: if type $\texttt{t}_1$ has kind $\texttt{K}_1 \Rightarrow \texttt{K}_2$ and type $\texttt{t}_2$ has kind $\texttt{K}_1$, the type application $\texttt{t}_1$ $\texttt{t}_2$ is well-kinded and has kind $\texttt{K}_2$. There are two base cases for kinded type variables (K-KindedTyVar) and kinded type constructors (K-KindedTyCon).

Whenever we may apply one of these rules to a type expression, we call this expression well-kinded. In particular, we define a well-kinded function signature respectively a well-

$$\frac{\texttt{tysig}_1 \texttt{ : } * \ldots \texttt{tysig}_n \texttt{ : } *}{\texttt{class ident tyvar}_1 \ldots \texttt{tyvar}_m \texttt{ where tysig}_1 \ldots \texttt{ tysig}_n \texttt{ : } *} \quad \text{(K-TyClass)}$$

$$\frac{\texttt{tyfun : } *}{\texttt{ident :: context => tyfun : } *} \quad \text{(K-TySig)}$$

$$\frac{\texttt{t}_1 \texttt{ : } * \quad \texttt{t}_2 \texttt{ : } *}{\texttt{t}_1 \rightarrow \texttt{t}_2 \texttt{ : } *} \quad \text{(K-TyFun)}$$

$$\frac{\texttt{t}_1 \texttt{ : K}_1\texttt{=>K}_2 \quad \texttt{t}_2 \texttt{ : K}_1}{\texttt{t}_1\texttt{t}_2 \texttt{ : K}_2} \quad \text{(K-TyApp)}$$

$$\frac{\texttt{t}_1 \texttt{ : } * \quad \ldots \quad \texttt{t}_n \texttt{ : } *}{(\texttt{t}_1, \ldots, \texttt{t}_n) \texttt{ : } *} \quad \text{(K-TyTuple)}$$

$$\frac{}{\texttt{tyvar : K : K}} \quad \text{(K-KindedTyVar)}$$

$$\frac{}{\texttt{Tycon : K : K}} \quad \text{(K-KindedTyCon)}$$

Figure 5.3: Kind rules in FTKL

kinded type class definition as follows:

Definition 4 *(Well-kindedness in FTKL.) An FTKL function signature is well-kinded if it has kind $*$. An FTKL type class definition is well-kinded if all function signatures it contains are well-kinded. Finally, an FTKL specification is well-kinded if all function signatures and type class definitions it contains are well-kinded.*

According to the kind rules from Fig. 5.3, no kind can be deduced for unkinded type constructors and type variables: since FTKL does not allow type definitions, there is no possibility to deduce the kind of a type constructor from its definition. As a consequence, all kind assignments for unkinded type variables and type constructors in FTKL must be inferred. Another consequence is that the mapping $\mathscr{T}$, which maps FTKL function signatures to CTKL function signatures and will be defined in the next chapter, may focus entirely on preserving the structure of a signature, and need not be concerned with concrete types. In the next two subsections, we further elaborate on each consequence. First we explain how kinds can be inferred, then we introduce the kind structure of a function signature. Later, we will show that the transformation $\mathscr{T}$ preserves this kind structure.

5.1.3 Kind inference and kind checking

As mentioned above, the assignment of kinds to type constructors and type variables in FTKL is optional, which means they can be provided in an FTKL specification but they do not have to. In any case, the kind information is needed by our transformation. Thus, missing kind information has to be inferred. In addition, given kind information has to be checked for consistency.

Input: An arbitrary FTKL function signature `f :: Context => a -> b`, such that every type constructor and every type variable is annotated with a kind.

Output: A boolean value, `true` if the function signature is well-kinded, `false` otherwise.

A1. Check the annotated kinds for consistency. Return `false` if different kinds are given for one type constructor or type variable.

A2. Start with type constructors and type variables, and apply the kind rules bottom-up to all complex type expressions. If there is an expression for which no kind rule can be applied, return `false`.

A3. If the kind rule for the whole function signature evaluates to `true`, return `true`.

Algorithm 5.1: `kind_check`: Kind checking for an FTKL function signature

The basis of kind inference and kind checking are the kind rules presented above (Fig. 5.3). If full kind information is given, the procedure listed in Alg. 5.1 can be applied for kind checking. It starts with a consistency check that just collects all given kind information and checks if it is free from contradictions. Afterwards, the kind rules are applied in a bottom-up way: as the kind of every type constructor and every type variable is given, the rules can be applied to every complex type expression, computing a kind for that expression. If a function signature is well-kinded, eventually the rule for a function signature can be applied. If a function signature is not well-kinded, there is a point where no kind rule is applicable. Fig. 5.4 shows how the `fold` function signature, with kind annotations for every type constructor and every type variable, can be kind-checked according to Alg. 5.1. Type expressions with kind annotations that are known are listed in the first column, the rules that are applied are listed in the third and the conclusion that can be drawn is listed in the second column. As the listing shows, the `fold` function signature is well-kinded. If it there would be occurrences of different kind annotation (e.g., if the type variable `b` would be annotated with kind `*->*`), the kind-checking algorithm would, for instance, if the premise was `(Fix s a: *) -> (b: *->*)`, find no rule that can be applied. Thus the algorithm would stop at this point by noting that the given function signature is not well-kinded.

Kind inference applies the kind rules in the opposite direction: Instead of checking the kind annotations that define the premise and then applying the appropriate rule to get the conclusion,

```
fold :: Bifunctor s => ((s: *->*->*) (a: *) (b: *) -> (b: *))
                       -> (Fix: (*->*->*)->*->*) (s: *->*->*) (a: *) -> (b: *)
```

premise	conclusion	rule
`(s: *->*->*)(a: *)(b: *)`	`s a b: *`	(K-TyApp), twice
`(s a b: *)-> (b: *)`	`s a b -> b: *`	(K-TyFun)
`(Fix: (*->*->*)->*->*)(s: *->*->*)(a: *)`	`Fix s a: *`	(K-TyApp), twice
`(Fix s a: *)-> (b: *)`	`Fix s a -> b: *`	(K-TyFun)
`(Fix s a -> b: *)(s a b -> b: *)`	`(s a b -> b)->` `(Fix s a -> b): *`	(K-TyFun)
`(s a b -> b)-> (Fix s a -> b): *`	`fold :: ... :*`	(K-TySig)

Figure 5.4: Kind checking for the signature of the `fold` function

we start with the assumption that a given type expression is well-kinded. We then choose the appropriate rule and compute the necessary premises to get the conclusion. If, for instance, the function signature `f :: A B -> B` is given, kind inference starts by assuming that the whole signature is well-kinded. With (K-TySig) we can conclude that then the function type `A B -> B` is well-kinded. From rule (K-TyFun), it follows that `A B` and `B` are well-kinded. In conclusion, `B` has to have kind $*$ and `A` has to have kind $* \Rightarrow *$.

Input: An arbitrary FTKL function signature `f :: Context => a -> b`, where type constructors and type variables are optionally annotated with a kind.

Output: A mapping k, which assigns a unique kind to every type variable and every type constructor in `f`, or an error message if the algorithm is not able to find such a mapping.

A1. Compute a kind mapping according to Alg. 5.3:
local_kinds = infer_local_kinds(`a -> b`, $*$)

A2. Construct k out of *local_kinds* in the following way:

1. If $local\_kinds(t)$ for a type variable or type constructor t contains kind $*$ and a complex kind $K_1 \Rightarrow K_2$, abort the algorithm with an error message that states ambiguous kinds for t.
2. If $local\_kinds(t)$ for a type variable or type constructor t contains two complex kinds of different structure, for instance $K_1 \Rightarrow K_2$ and $K_1 \Rightarrow K_2 \Rightarrow K_3$, abort the algorithm with an error message that states ambiguous kinds for t.
3. If $local\_kinds(t)$ for a type variable or type constructor t contains $*$, the kind variable K_t, and nothing else, set $k(t) = *$.
4. If $local\_kinds(t)$ for a type variable or type constructor t contains only the kind variable K_t, set $k(t) = *$.
5. If $local\_kinds(t)$ for a type variable or type constructor t contains the kind variable K_t, a complex kind $K_1 \Rightarrow K_2$, and nothing else, set $k(t) = K_1 \Rightarrow K_2$.
6. Check the mapping *local_kinds* for mutual dependencies. If there are types $t_1, t_2, \ldots, t_n$ such that $local\_kinds(t_1)$ contains the kind variable K_{t_2}, and so on, and $local\_kinds(t_n)$ contains the kind variable K_{t_1}, abort the algorithm and report an error.
7. Replace all kind variables K_t for type variables or a type constructor t that remain in the mapping *local_kinds*: if t is a type variable or a type constructor, and $local\_kinds(t)$ contains only one entity, replace all occurrences of K_t with $local\_kinds(t)$. If t is a type variable or a type constructor, and $local\_kinds(t)$ contains more than one entity, report an error.
8. Replace all kind variables K_t for a composed type t that remain in the mapping *local_kinds*. Reconstruct the kind of t according to the kind rules and the mapping *local_kinds*. If this is not possible, report an error.

A3. return k.

Algorithm 5.2: `infer_kinds`: kind inference for an FTKL signature `f`

The kind rules for FTKL guarantee neither existence nor uniqueness of a mapping from type variables and type constructors to kinds. For the function signature `f :: A B -> A`, for

instance, the well-kindedness assumption induces that `A B` and `A` are well-kinded. In conclusion, `B` has to have kind $*$ and `A` has to have kind $* \Rightarrow *$ and kind $*$, which is a contradiction. This example shows that syntactic correctness does not guarantee well-kindedness.
On the other hand, for the function signature `f :: A B -> C`, there is an infinite number of kind annotations that lead to a well-kinded signature `f`:

`A`	`B`	`C`
$* \Rightarrow *$	$*$	$*$
$* \Rightarrow * \Rightarrow *$	$* \Rightarrow *$	$*$
$* \Rightarrow * \Rightarrow * \Rightarrow *$	$* \Rightarrow * \Rightarrow *$	$*$
...	...	...

To avoid ambiguities, we decide that if there are many possible kind annotations we choose the simplest (containing the least number of $*$ symbols). In the example above, we therefore end up with kind $* \Rightarrow *$ for `A`, kind $*$ for `B`, and kind $*$ for `C`.

The kind inference procedure is implemented in Alg. 5.2 and Alg. 5.3. While Alg. 5.2 acts at a global level, Alg. 5.3 performs a local kind inference. "Local" means, that the algorithm infers kinds for a given type expression without taking into account the kind information that is available from other type expressions. Thus, there might be different kinds that can be deduced for one type constructor or type variable, derived at different positions, and Alg. 5.3 collects them in a list. The inputs for Alg. 5.2 are the type expression that should be analysed and the assumed kind of the type expression. The assumed kind can be either a concrete kind or a kind variable. Such a kind variable states that there are infinitely many possible kinds for the type expression. Algorithm 5.3 applies the kind rules of FTKL in a backward manner. For complex type expressions, it calls itself for all subexpressions. If the complex type expression is a type application `t1 t2`, a kind variable for the second type `t2` is introduced. This kind variable stands for any kind and expresses the fact that the type application is well-kinded if the second type `t2` has an arbitrary kind K_{t2} and the first type has a kind that can be applied to K_{t2}. For instance, if the type application `t1 t2` has to have kind $*$, the kind of `t1` should be $K_{t2} \Rightarrow *$.

Fig. 5.5 sketches the application of Alg. 5.3 to the function type of the `fold` function signature without any kind annotations. It starts with the assumption that the function type `(s a b -> b) -> Fix s a -> b` is well-kinded, therefore the second argument for the call of Alg. 5.3 is $*$. The function type is then divided into domain and codomain and `infer_local_kinds` is called recursively for each. The recursion stops whenever `infer_local_kinds` is called for one of the type variables `a`, `b`, or `s`, or the type constructor `Fix`. The computed possible kinds are collected in a mapping. At the end of the computation, this mapping has collected the following local kinds:

type variable or type constructor t	$kinds(t)$
a	K_a
b	K_b, $*$
s	$K_a \Rightarrow K_b \Rightarrow *$, K_s
Fix	$K_s \Rightarrow K_a \Rightarrow *$

The type variable `a` only occurs in type applications and can therefore be of arbitrary kind. In contrast, `b` occurs as codomain type of a function type, therefore, `b` has to have kind $*$. As `b` also occurs in type applications, the possible kinds for `b` include a kind variable. As `s` and `Fix`

Input: An arbitrary FTKL type `t` and a kind or a kind variable K.

Output: A mapping *kinds*, which assigns a list of all kinds that can be locally deduced to every type variable and every type constructor that appears in `t`.

A1. Case distinction on the type of `t`:

- `t1 -> t2`:
  $kinds_1$ = infer_local_kinds(t1, K)
  $kinds_2$ = infer_local_kinds(t2, K)
  $kinds$ = join($kinds_1$, $kinds_2$)
  return *kinds*
- `(t1, t2, ..., tN)`:
  $kinds_1$ = infer_local_kinds(t1, K)
  $kinds_2$ = infer_local_kinds(t2, K)
  ...
  $kinds_N$ = infer_local_kinds(tN, K)
  $kinds$ = join($kinds_1$, $kinds_2$, ..., $kinds_N$)
  return *kinds*
- `(t)`:
  return infer_local_kinds(t, K)
- `t1 t2`:
  $kinds_1$ = infer_local_kinds(t1, $KV_{T2} \Rightarrow K$)
  $kinds_2$ = infer_local_kinds(t2, KV_{T2})
  $kinds$ = join($kinds_1$, $kinds_2$)
  return *kinds*
- otherwise:
  for a type constructor or type variable `t` that is not annotated with a kind:
  return a mapping *kinds* with *kinds*(`t`) = [K]
  for a type constructor or type variable `t` that is annotated with a kind K_1:
  return a mapping *kinds* with *kinds*(`t`) = [K, K_1]

Algorithm 5.3: `infer_local_kinds`: local kind inference for an FTKL type `t`

play the role of type constructors in type applications, their possible kinds depend on the kinds of `a` and `b`.

At the global level, Alg. 5.2 uses the multi-mapping computed by Alg. 5.3 to reconstruct a unique mapping from type constructors and type variables to kinds. This is done by stripping the function type from a function signature, and then calling Alg. 5.3 to compute the local kinds for type constructors and type variables. Based on the mapping from type constructors and type variables to lists of local kinds as returned by Alg. 5.3, Alg. 5.2 constructs a mapping that maps every type constructor and every type variable to a unique kind. In the example above, the list of local kinds for `b` contains $*$, so `b` has to have $*$. For `a`, every kind is possible and according to the convention to choose the simplest possible kind, `a` is also mapped to $*$. The kinds of `s` and `Fix` can then be constructed accordingly, leading to the mapping:

```
infer_local_kinds ((s a b -> b) -> Fix s a -> b, *)
= join (infer_local_kinds (s a b -> b, *),
        infer_local_kinds (Fix s a -> b, *))
= join (join (infer_local_kinds (s a b, *),
              infer_local_kinds (b, *)),
        join (infer_local_kinds (Fix s a, *),
              infer_local_kinds (b, *)))
= join (join (join (infer_local_kinds (s a, K_b => *),
                    infer_local_kinds (b, K_b)),
              infer_local_kinds (b, *)),
        join (join (infer_local_kinds (Fix s, K_a => *),
                    infer_local_kinds (a, K_a)),
              infer_local_kinds (b, *)))
= join (join (join (join (infer_local_kinds (s, K_a => K_b => *),
                          infer_local_kinds (a, K_a)),
                    infer_local_kinds (b, K_b)),
              infer_local_kinds (b, *)),
        join (join (join (infer_local_kinds (Fix, K_s => K_a => *),
                          infer_local_kinds (s, K_s)),
                    infer_local_kinds (a, K_a)),
              infer_local_kinds (b, *)))
= join (join (join (join (kinds: s ---> [K_a => K_b => *],
                          kinds: a ---> [K_a]),
                    kinds: b ---> [K_b]),
              kinds: b ---> [*]),
        join (join (join (kinds: Fix ---> [K_s => K_a => *],
                          kinds: s ---> [K_s]),
                    kinds: a ---> [K_a]),
              kinds: b ---> [*])
= kinds: s ---> [K_a => K_b => *, K_s],
         a ---> [K_a],
         b ---> [K_b, *],
         Fix ---> [K_s => K_a => *]
```

Figure 5.5: Computation of the local kinds for the fold function signature

type variable or type constructor t	$k(t)$
a	$*$
b	$*$
s	$* \Rightarrow * \Rightarrow *$
Fix	$(* \Rightarrow * \Rightarrow *) \Rightarrow * \Rightarrow *$

These kinds represent a structure of the function signature for `fold`. Whenever the transformation $\mathscr{T}$ is applied to the `fold` signature, we have to ensure that the kinds of the type constructors and type variables are preserved. Whatever, for instance, the type constructor `Fix` is mapped

to under a function signature transformation, it should have the same kind as `Fix` itself. In the next section, we define the kind structure of a function signature. This property will connect the kinds of type constructors and type variables with the structure of the function types involved in a function signature.

5.1.4 Kind structure

The information about the kinds of the type constructors and type variables is useful, but a function signature provides some more information. In particular, the function type involved in a function signature contains information how these type constructors and type variables are composed. To encode the kinds involved in an expression and describe precisely how a type-level expression has been constructed, we therefore introduce the kind structure of a well-kinded expression. It is defined as follows:

Definition 5 *(Kind structure in FTKL.) The kind structure of an FTKL type `t`, ks(`t`), is defined by structural induction:*

1. *For a type constructor `TC` with kind K_{TC}, $ks(\texttt{TC}) = (K_{TC})$.*
2. *For a type variable `TV` with kind K_{TV}, $ks(\texttt{TV}) = (K_{TV})$.*
3. *For a tuple type `(t1, ..., tN)`, it is $(ks(\texttt{t1}), \ldots, ks(\texttt{tN}))$.*
4. *For a type in parentheses `(t)`, it is $(ks(\texttt{t}))$.*
5. *For a function type `t1 -> t2`, it is $ks(\texttt{t1})$ `->` $ks(\texttt{t2})$.*
6. *For a type application `t1 t2`, it is $ks(\texttt{t1})$ $ks(\texttt{t2})$.*

The kind structure of an FTKL function signature `f :: Context => a -> b`, ks(`f`), is given by the kind structure of the function type $ks(\texttt{a -> b})$.

To compute the kind structure, the kinds of all type constructors and type variables have to be available. Therefore, kind inference and kind checking have to be done before the kind structure can be computed. If kind checking shows that the function signature is well-kinded, the kind structure is well-defined. The same holds true for FTKL types in general, stated in the following lemma:

Lemma 1 *If t is a well-kinded FTKL type, its kind structure is well-defined.*

Proof: Routine induction. If t is a type constructor or a type variable, well-kindedness means that there is a unique kind for t and therefore the kind structure is well-defined. If t is a composed type, its kind structure is well-defined if it is well-defined for all its sub-expressions. By the induction hypothesis, all sub-expressions have a well-defined kind structure, therefore t has a well-defined kind structure. □

Based on the kinds inferred for the `fold` function signature in the example above, Fig. 5.6 shows the kind structure of this function signature. Once the kinds for type constructors and type variables are inferred, computation of the kind structure for `fold` is rather simple. The structure of the function type in a function signature is combined with the kind information, yielding a meaningful description for the structure of a function signature.

```
fold :: Bifunctor s => (s a b -> b)-> Fix s a -> b
((* ⇒ * ⇒ *) (*) (*) -> (*)) -> ((* ⇒ * ⇒ *) ⇒ * ⇒ *) (* ⇒ * ⇒ *) (*) -> (*)
```

Figure 5.6: The fold signature in FTKL and its kind structure

5.2 The Concept Type Kernel Language (CTKL)

The language CTKL represents the counterpart to FTKL for non-functional languages. Like FTKL, it centers around the notion of function signatures and provides type variables for parameterised types. However, CTKL does not provide a function type and thus no notion of higher-order functions. Further, type application is limited to cases where the first type, which is applied to the second one, is a type constructor or a type variable. Thus, no notion of partial type application is provided. In contrast to FTKL, kinds are not explicitly available at syntax level in CTKL.

As FTKL, CTKL provides concept definitions, but unlike FTKL, CTKL provides three kinds of predefined concepts: FUNCTION and TYPECONSTRUCTOR concepts for each arity, and the SAMETYPE concept to enforce type parameter bindings. These concepts provide the infrastructure to model functional types at a higher level of abstraction. The FUNCTION concept, for instance, states that every type that provides the three associated entities domain type, codomain type, and application operator can be used as a type for functions. Similarly, the TYPECONSTRUCTOR concept states that a type that provides the possibility of type application is a type constructor.

In the following subsections we present the grammar, the kind rules, and the kind structure for CTKL.

5.2.1 Syntax

The grammar for the syntax of CTKL is shown in Fig. 5.7. A CTKL specification consists of a set of class definitions (concepts) and function signatures. Class definitions are similar to FTKL class definitions, but also comprise class names for the predefined concepts: `SameType`, `FunctionN` (for functions of different arrities), and TYPECONSTRUCTORN (for type constructors of different arrities). Function signatures, as in FTKL, consist of the declaration of domain and codomain types, but classify now the function identifier as a type constructor (<c_tycon>) and always contain an optional list of type variables along with a context. A context pairs up requirements (which then map to concepts) and the types they constrain (see <c_type>). Types in CTKL, in turn, similar to the functional case, can be type variables, type constructors, and tuples. An important difference, however, is that functions are no longer types. This reflects the treatment of higher-order functions in CTKL: functional abstraction and the encapsulation it provides will be realised by concept-based encapsulation. On the other hand, in FTKL there are only unary functions. Functions with more than one argument are represented as unary functions acting on a tuple of inputs or as curried functions. In CTKL, functions with a higher arity are explicitly supported. This illustrates the non-functional nature of CTKL; in languages like C++, Java, Ada, or Scala, functions with higher arities are widely used. Another difference between CTKL and FTKL concerns type application (<c_app>): since there is no partial application or currying, a type application must always yield a complete type, and therefore applies a type variable or type constructor to the list of all its arguments. Figure 5.8 shows an example

```
<c_spec>          :: <c_spec_item>+

<c_spec_item>     :: <c_conceptdef> | <c_fctsig>

<c_conceptdef>    :: "concept" <c_conceptname> {<c_tyvar>}+
                     "where" {<c_fctsig>}
<c_conceptname>   :: <regular_string> | "SameType" |
                     "Function" | "Function2" | ... |
                     "TypeConstructor" | "TypeConstructor2" | ...

<c_fctsig>        :: <c_tycon> {<c_tyvar_list>} "::" {<c_context> "=>"}
                     <c_type> "->" <c_type>

<c_context>       :: <c_requirement> {"," <c_requirement>}
<c_requirement>   :: <c_conceptname> <c_type>+

<c_type_list>     :: <c_type> {"," <c_type>}
<c_tyvar_list>    :: "<" [ <c_tyvar> {"," <c_tyvar>} ] ">"

<c_type>          :: <c_tyapp> | <c_tyvar> | <c_tycon> |
                     <c_tytuple> | "("<c_type>")" | <c_tymem>
<c_tyapp>         :: <c_tyvar> "<" <c_type_list> ">" |
                     <c_tycon> "<" <c_type_list> ">"
<c_tytuple>       :: "(" <c_type> {"," <c_type>}+ ")"
<c_tymem>         :: <c_type> "." <c_type>

<c_tyvar>         :: <small_letter_beginning_string>
<c_tycon>         :: <capital_letter_beginning_string>
```

Figure 5.7: The syntax of CTKL

of a valid expression and its derivation tree using the grammar in Fig. 5.7.

5.2.2 Kind rules

Although kinds are no explicit part of the syntax of CTKL, the structure of kinds in CTKL is similar to that of FTKL. Not surprisingly, however, the kind rules in CTKL (see Fig. 5.9) are different from those in FTKL.

One difference is that there is no longer a single kind rule for functions, as CTKL does not provide a function type. The kind rules for functions (of different arities) are now captured in the rules (K-TyFunParam1), (K-TyFunParam2), and so on. These rules describe when a type parameter that is supposed to model a function type, declared by a `Function` requirement, is well-kinded. The kind rules (K-TyFunParamN) correspond to the (K-TyFun) rule in FTKL, and require for the well-kindedness of a function the well-kindedness of its domain and codomain in much the same way FTKL does for its function type.

```
Fold <s, a, fun1> ::
   Bifunctor s, Function fun1, SameType fun1.Domain1 (s <a, fun1.Codomain> ),
   TypeConstructor2 s
   => fun1 -> Fold_codomain1
```

```
<c_spec>
<c_fctsig>
  <tycon> "Fold"
  <c_tyvar_list>
  "<" <c_type> <c_tyvar> "s" ","
      <c_type> <c_tyvar> "a" ","
      <c_type> <c_tyvar> "fun1"
  ">"
  "::"
  <c_context>
    <c_requirement>
      <c_conceptname> "Bifunctor" <c_type> <c_tyvar> "s"
    <c_requirement>
      <c_conceptname> "Function" <c_type> <c_tyvar> "fun1"
    <c_requirement>
      <c_conceptname> "SameType"
      <c_type>
        <c_tymem>
          <c_type> <c_tyvar> "fun1"
          "."
          <c_type> <c_tycon> "Domain1"
      <c_type>
        <c_tyapp>
          <c_tyvar> "s"
          "<" <c_type_list>
                <c_type> <c_tyvar> "a" ","
                <c_type>
                  <c_tymem>
                    <c_type> <c_tyvar> "fun1"
                    "."
                    <c_type> <c_tycon> "Codomain"
          ">"
    <c_requirement>
      <c_conceptname> "TypeConstructor2"  <c_type> <c_tyvar> "s"
  "=>"
  <c_type> <c_tyvar> "fun1"
  "->"
  <c_type> <c_tycon> "Fold_codomain1"
```

Figure 5.8: Derivation of the fold function signature from the Origami library in CTKL

$$\frac{\mathtt{t_1} : * \quad \mathtt{t_2} : *}{\mathtt{ident} :: \mathtt{Context} => \mathtt{t_1} \to \mathtt{t_2} : *} \quad \text{(K-CTKL-TySig)}$$

$$\frac{\texttt{Function fun} \in \texttt{context} \quad \mathtt{fun.Domain1} : * \quad \mathtt{fun.Codomain} : *}{\mathtt{fun} : *} \quad \text{(K-TyFunParam1)}$$

$$\frac{\texttt{Function2 fun} \in \texttt{context} \quad \mathtt{fun.Domain1} : * \quad \mathtt{fun.Domain2} : * \quad \mathtt{fun.Codomain} : *}{\mathtt{fun} : *} \quad \text{(K-TyFunParam2)}$$

...

$$\frac{\mathtt{TypeConstructor\ t} \in \mathtt{context}}{\mathtt{t} : *=>*} \quad \text{(K-TyCtorParam1)}$$

$$\frac{\mathtt{TypeConstructor2\ t} \in \mathtt{context}}{\mathtt{t} : *=>*=>*} \quad \text{(K-TyCtorParam2)}$$

...

$$\frac{\mathtt{t_1} : \mathtt{K_1} \quad \mathtt{SameType\ t_1\ t_2} \in \mathtt{context}}{\mathtt{t_2} : \mathtt{K_1}} \quad \text{(K-SameType1)}$$

$$\frac{\mathtt{t_2} : \mathtt{K_1} \quad \mathtt{SameType\ t_1\ t_2} \in \mathtt{context}}{\mathtt{t_1} : \mathtt{K_1}} \quad \text{(K-SameType2)}$$

$$\frac{\mathtt{t_2} : \mathtt{K_1}}{\mathtt{t_1.t_2} : \mathtt{K_1}} \quad \text{(K-TyMem)}$$

$$\frac{\mathtt{t} : \mathtt{K_1} => \ldots => \mathtt{K_n} => * \quad \mathtt{t_1} : \mathtt{K_1} \quad \ldots \quad \mathtt{t_n} : \mathtt{K_n}}{\mathtt{t} < \mathtt{t_1}, \ldots, \mathtt{t_n} > \ : *} \quad \text{(K-TyApp)}$$

$$\frac{\mathtt{t_1} : * \quad \ldots \quad \mathtt{t_n} : *}{(\mathtt{t_1}, \ldots, \mathtt{t_n}) : *} \quad \text{(K-TyTuple)}$$

Figure 5.9: Kind rules in CTKL

Each of the other predefined concepts, including arities, has an associated rule of its own, to ensure the kinding that its name suggests (see the rules (K-TyCtorParamN), (K-SameType1), and (K-SameType2)); accordingly, the premises of those rules contain look-ups of the context. Further, since CTKL has no higher-order functions and type application has to be expressed in uncurried form, the kind rule of applications has to be changed to take this fact into account: (K-TyApp) expects a complete list of type arguments. The remaining rules are the top-level rule (K-CTKL-TySig) and the rule for tuples, which resemble the rules in FTKL, and the rule (K-TyMem), which has no counterpart in FTKL, since the notion of member types does not exist there.

5.2.3 Kind structure

In analogy to FTKL, we define the semantics of CTKL expressions in terms of the kind structure. As in FTKL, the notion of kind structure depends on the notion of well-kindedness, therefore this notion is defined first. The definition of well-kindedness is similar to the one in FTKL: we consider a CTKL expression to be well-kinded if it has kind $*$. In particular, we get the following definition for well-kinded CTKL function signatures and concept definitions:

Definition 6 *(Well-kindedness in CTKL.) A CTKL function signature is well-kinded if it has kind $*$. A CTKL concept definition is well-kinded if all function signatures it contains are well-kinded. Finally, a CTKL specification is well-kinded if all function signatures it contains are well-kinded.*

```
Fold_codomain1 <s, a, fun1> ::
   Bifunctor s,
   Function fun1,
   SameType fun1.Domain1 (s <a, fun1.Codomain> )
   TypeConstructor2 s,
                    => Fix <s, a> -> fun1.Codomain
Fold <s, a, fun1> ::
   Bifunctor s,
   Function fun1,
   SameType fun1.Domain1 (s <a, fun1.Codomain> ),
   TypeConstructor2 s
                     => fun1 -> Fold_codomain1
```

$$((* \Rightarrow * \Rightarrow *)\,(*)\,(*) \text{ -> } (*)) \text{ -> } ((* \Rightarrow * \Rightarrow *) \Rightarrow * \Rightarrow *)\,(* \Rightarrow * \Rightarrow *)\,(*) \text{ -> } (*)$$

Figure 5.10: The fold signature in CTKL and its kind structure

As with FTKL, we are interested in the kind structure of a well-kinded expression. Similar to the FTKL kind structure, the kind structure in CTKL combines kind information about type variables and type constructors with information about the structure of the function signature. Not surprisingly, its definition is rather similar to the one in FTKL:

Definition 7 *(Kind structure in CTKL.) The kind structure of a CTKL type for a given context C, ks(t, C), is defined by structural induction:*

1. *For a type constructor TC with kind K_{TC}, ks(TC, C) = (K_{TC}).*

2. *For a type variable* `TV` *with kind* K_{TV}, *ks(*`TV, C`*) =* (K_{TV}).

3. *For a type variable* `TV` *that is constrained by a* `TypeConstructorN` *concept in* `C`, *it is the kind induced by that requirement.*

4. *For two type variables or type constructors* `T1` *and* `T2` *that are constrained by a* `SameType` *requirement in* `C`, *their kind structure is equal: ks(*`T1, C`*) = ks(*`T2, C`*).*

5. *For a type variable* `TV` *that is constrained in* `C` *by an n-ary function concept* `FunctionN` *with domain types* `t1`, ..., `tN` *and codomain type* `t`, *ks(*`TV, C`*) = (ks(*`t1, C`*) -> ... -> ks(*`tN, C`*) -> ks(*`t, C`*)).*

6. *For a type in parentheses* `(t)`, *it is (ks(*`t, C`*)).*

7. *For a tuple type* `(t1, ..., tN)`, *it is (ks(*`t1, C`*), ..., ks(*`tN, C`*)).*

8. *For a type application* `t<t1,..., tN>`, *it is ks(*`t, C`*) ks(*`t1, C`*) ... ks(*`tN, C`*).*

The kind structure of a CTKL *function signature* `f <p1, ..., pN> :: Context => a -> b`, *ks(*`f`*), is given by ks(*`a, Context`*) -> ks(*`b, Context`*).*

For convenience we write the kind structures for functions and type applications (cases (5),(7) in Def. 7) in curried form to allow for easy comparison with kind structures in FTKL.
In analogy to FTKL, well-kindedness and well-defined kind structures in CTKL are related: if a function signature is well-kinded, the kind structure is well-defined. The following lemma generalises for every CTKL type:

Lemma 2 *For a well-kinded* CTKL *type, the kind structure is well-defined.*

Proof: Routine induction as in the case of FTKL. For a type constructor or a type variable, well-kindedness means that there is a unique kind for it and therefore the kind structure is well-defined. For a composite type the kind structure is well-defined if all its parts have a well-defined kind structure. This is true by the induction hypothesis, therefore the kind structure of every composite type is well-defined. □

Figure 5.10 shows the signature of fold in CTKL, along with its kind structure. Comparing Figure 5.6 and Figure 5.10, we can convince ourselves that the kind structure is the same in the two languages.

Chapter 6

Transformation

In the previous chapter, we introduced the two kernel languages that capture the essentials of functional and non-functional type systems, where the latter was equipped with concepts that classify functional types. In this chapter, we turn to the main contribution, the transformation proper, which maps FTKL signatures to CTKL signatures.

As we have already mentioned in the previous chapters, the biggest task of the transformation is the lowering of higher-order functions and higher kinds, called defunctionalisation. The main idea of the transformation is to perform defunctionalisation at the level of types. Higher-order function types in function signatures are defunctionalised as well as curried type applications, which use higher-order functions at the level of types. This approach differs from common defunctionalisation methods, which only deal with function bodies.

The transformation assumes well-kindedness of the FTKL function signature. Further, it requires that all kind information is available, that means kinds of all type variables and all type constructors are explicitly given or inferred by the procedure presented in the previous chapter.

We have broken down the transformation in five major steps that are organised in a pipe-and-filter manner. We start by giving a brief overview of all steps (Sect. 6.1), then we present every step in detail, including statements and proofs about properties of these steps (Sect. 6.2 - Sect. 6.6). Finally, we summarise the complete transformation and its properties (Sect. 6.7).

6.1 Overview

We introduce the transformation by summarising the five major steps. This overview is meant to illustrate the main ideas of every step, and does not aim at an exhaustive presentation. While the upcoming subsections will describe every step in a detailed way, we disregard in this overview details as far as possible. As it is instructive to illustrate the steps with an example, we use the `fold` function from the Origami library, which we already used in the previous chapter, as a running example throughout this chapter:

```
1 fold :: Bifunctor s => (s a b -> b) -> Fix s a -> b
```

The kind information is not written explicitly in the example. The `fold` function signature contains all crucial elements of FTKL: there is a function type both in domain and codomain position, the type parameter `s` represents a type constructor, and `Fix` is a concrete, higher-order

type constructor. Thus, all transformational steps are actually executed in this example and have an impact on the function signature. For other function signatures, some steps might be effectless as the input might not contain the type-level construct which should actually be transformed.

The five steps of the transformation are, in the order in which they are executed:

1. **Uncurry type constructor application**. As CTKL does not support curried type constructor applications, occurrences of these have to be transformed into an uncurried form. In the running example, there are two such applications, `s a b` and `Fix s a`; they are rewritten as follows:

```
fold :: Bifunctor s => (s <a, b> -> b) -> Fix <s, a> -> b
```

   Note that `fold` is no longer syntactically valid FTKL; we thus introduced an intermediate language FTKL' (see below).

2. **Defunctionalisation of codomain types**. As CTKL provides no function type, higher-order functions in a given function signature have to be replaced by functions that do not contain any function type in its domain or codomain type. We introduce a new function signature (and therefore a new type) for every higher-order function type in the codomain of a type signature and replace this higher-order function type by a type constructor for the new type. We call this step *codomain defunctionalisation*. In the running example, we define the function signature (and type) `TC_fold_codomain1` and rewrite the codomain of `fold`:

```
TC_fold_codomain1 :: Bifunctor s => Fix <s, a> -> b
fold :: Bifunctor s => (s <a, b> -> b) -> TC_fold_codomain1
```

3. **Defunctionalisation of domain types**. Function types in the domain of a function signature need to be replaced as well, but in a way different from the codomain case. For each higher-order function type, we introduce a type variable and extend the context by constraining concepts: the type parameter itself is constrained by the `Function` concept, and `SameType` concepts ensure that all domain and codomain types are "initialised" to the original types. We call this step *domain defunctionalisation*. In the example, the function type in the domain of `fold` (`s<a,b> -> b`) is replaced by the type variable `fun1` and the context of `fold` is extended by three requirements on `fun1` and its domain and codomain (`Function fun1` and the two `SameType` requirements on `fun1.Domain1` and `fun1.Codomain1`); this additional context is also passed to `TC_fold_codomain1`.

```
TC_fold_codomain1 <s, a, b, fun1> ::
    Bifunctor s,
    Function fun1,
    SameType fun1.Domain1 (s <a, b> ),
    SameType fun1.Codomain b          =>  Fix <s, a> -> b
fold <s, a, b, fun1> ::
    Bifunctor s,
    Function fun1,
    SameType fun1.Domain1 (s <a, b> ),
    SameType fun1.Codomain b          => fun1 -> TC_fold_codomain1
```

4. **Encoding higher kinds**. While FTKL supports higher kinds of type variables at a syntactical level, in CTKL they have to be encoded in the context. The *type constructor encoding* step checks the kind of all type variables and adds a `TypeConstructorN` requirement for each type variable that is a placeholder for a type constructor; the arity is indicated in a numeral suffix. In the example, only type variable `s` represents a type constructor; since it is a binary type constructor, it is properly constrained by the `TypeConstructor2` concept.

```
1 TC_fold_codomain1 <s, a, b, fun1> ::
2     Bifunctor s,
3     Function fun1,
4     SameType fun1.Domain1 (s <a, b> ),
5     SameType fun1.Codomain b,
6     TypeConstructor2 s                => Fix <s, a> -> b
7 fold <s, a, b, fun1> ::
8     Bifunctor s,
9     Function fun1,
10    SameType fun1.Domain1 (s <a, b> ),
11    SameType fun1.Codomain b,
12    TypeConstructor2 s                => fun1 -> TC_fold_codomain1
```

5. **Optimizing context.** The encoding of higher-order functions and higher kinds in the context of a function signature might lead to redundancies in the requirements; those are removed in the last step. Type parameters are called *redundant* if they can by expressed by a member type of a function type parameter, and requirements are called *redundant* if they encode trivial identities. In the example, we can optimise away the type variable `b` (see `SameType fun1.Codomain1 b`) and, subsequently, one of the two `SameType` requirements in the context of `TC_fold_codomain1`. The result of the optimisation step is the result of the entire transformation:

```
1 TC_fold_codomain1 <s, a, fun1> ::
2     Bifunctor s,
3     Function fun1,
4     SameType fun1.Domain1 (s <a, fun1.Codomain> ),
5     TypeConstructor2 s          => (Fix <s, a> ) -> fun1.Codomain
6 fold <s, a, fun1> ::
7     Bifunctor s,
8     Function fun1,
9     SameType fun1.Domain1 (s <a, fun1.Codomain> ),
10    TypeConstructor2 s          => fun1 -> TC_fold_codomain1
```

Usually, the intermediate results of the transformation are no longer FTKL function signatures. Already after the first step of uncurrying type constructor applications, the function types in the function signatures may contain type applications in CTKL style. As the function types may still contain higher-order functions, the function signatures are also not yet CTKL function signatures. To capture the intermediate results of the transformation, we introduce an intermediate kernel language called FTKL'. It is based on FTKL, but differs from the former by an additional rule for type lists and an accordingly modified type application rule:

```
<typelist>  :: "<" <type> {"," <type>} ">"
<tyapp'>    :: <tyvar> <typelist> | <tycon> <typelist>
```

The type of the first argument of a type application is now constrained to be a type variable or a type constructor, while the second argument has to be a list of types. These new syntax rules are supplemented by a new kinding rule that replaces the (K-TyApp) rule of FTKL:

$$\frac{\mathtt{t} : \mathtt{K}_1 => \ldots => \mathtt{K}_n => * \qquad \mathtt{t}_1 : \mathtt{K}_1 \qquad \ldots \qquad \mathtt{t}_n : \mathtt{K}_n}{\mathtt{t} < \mathtt{t}_1, \ldots, \mathtt{t}_n > \; : *} \quad \text{(K-TyApp)}$$

The definition for well-kinded FTKL' function signatures is the same as that for FTKL function signatures; for the definition of the kind structure we combine the definitions of the kind structure of FTKL and CTKL in the appropriate way.

In the remainder of the chapter, we explain each of the five steps in detail.

6.2 Uncurrying of Type Constructor Applications

The transformation starts by uncurrying type applications. As obvious from Fig. 5.1, FTKL has only one type application rule, which applies one type to another type. Higher-arity type constructors have to be implemented by curried type application. For instance, the binary pair type constructor `Pair` is applied as `Pair a b`, which is equal to `(Pair a) b`. This is an application of `Pair a` to `b` where `Pair a` is an application of `Pair` to `a`. `Pair` is a type constructor with kind $* \Rightarrow * \Rightarrow *$ and `Pair a` itself is a type constructor of kind $* \Rightarrow *$; `Pair a` can be applied to `b` and the whole type application `Pair a b` has kind $*$. Therefore, `Pair` can be seen as a higher-order type-level function as it contains another type-level function in the codomain. Applying `Pair` to `a` does not yield a type but a type-level function again.

CTKL does not allow such curried expressions as in its type application rule only type constructors or type variables are allowed as first type. As a consequence, a type application in CTKL is only well-kinded if it has kind $*$, while in FTKL a well-kinded type application can be of other kinds. The reason behind this restriction, is, as discussed, the non-functional nature of CTKL.

Consequently, to transform FTKL expressions into CTKL expressions, curried type constructor applications have to be transformed into uncurried versions. As explained before, curried type applications introduce type-level higher-order functions, therefore this first step can be seen as a defunctionalisation at the level of type constructors. The defunctionalisation is not complete, as only the codomains of type-level functions are defunctionalised.

Algorithm 6.1 performs the uncurrying of type applications of a given type. To uncurry all type applications in a given FTKL specification, the algorithm is applied to the function type of all function signatures of an FTKL specification. The algorithm works by pattern matching on the input type: if the input type is a type variable or a type constructor, it is just returned. If it is a type composed of simpler types (i.e., function type, tuple type, or type application) the algorithm is applied recursively to its components. In the case of a type application the algorithm does not only perform the recursion step, but also creates a new type by making explicit that the new first type (after the recursion) of the uncurried type application is applied to the second. If the new first type is an uncurried type application itself, the second one has to be inserted into its type argument list.

The type constructor uncurrying algorithm can be characterised by the following lemmas:

Input: An FTKL type `t`

Output: An FTKL' type `t'` where all type constructor applications are uncurried and all other types are unchanged.

A1. Case distinction on the type of `t`:

- `t1 -> t2`:
  return `uncurry_tc(t1) -> uncurry_tc(t2)`
- `(t1, t2, ..., tN)`:
  return `(uncurry_tc(t1), uncurry_tc(t2), ..., uncurry_tc(tN))`
- `(t)`:
  return `(uncurry_tc(t))`
- `t1 t2`:
  set `t1` = uncurry_tc(t1)
  set `t2` = uncurry_tc(t2)
  if `t1` is of type `t1' <t2'_1, ... ,t2'_N>`
  - set `t1 = t1'`
  - set `t2 = <t2'_1, ... t2'_N, t2>`

  else set `t2 = <t2>`
  return `t1 t2`
- otherwise:
  return `t`

Algorithm 6.1: `uncurry_tc`: type constructor uncurrying of an FTKL type `t`

Lemma 3 *Alg. 6.1 terminates for every FTKL type, thus the complete type constructor uncurrying step terminates.*

Proof: In the case distinction of Alg. 6.1, the recursive call of Alg. 6.1 is only applied to sub-expressions of the current input expression. As FTKL only allows finite expressions, termination is guaranteed. □

Lemma 4 *For any well-kinded FTKL type, Alg. 6.1 preserves its well-kindedness.*

Proof: Induction on the type construction of FTKL. In the base case, the type is either a type constructor or a type variable. Then, the type is not transformed and therefore the well-kindedness is preserved. In the inductive step, there are three distinguished cases of term construction. In the case of function types, types in parenthesis, and tuple types, Alg. 6.1 is just recursively applied to the component types of the input type. By induction hypothesis, the types that result from the recursive application of Alg. 6.1 have kind $*$. Then the function type respectively the tuple type has kind $*$, which can be deduced by applying the kind rules (K-TyFun) respectively (K-TyTuple).
The crucial case is that of type application—on the one hand because type application types are changed by the transformation, on the other hand because type applications involve higher kinds. For type application, there are two possible cases: the type application either can be a curried application of a type constructor or just a type constructor application (equivalent to

the two cases in Alg. 6.1). In the first case, the first type of the type application is a partial application of a type constructor, which means type application has, without loss of generality, the form `(t1 t2) t3`. As this type application is well-kinded by assumption, `t3` has an arbitrary kind K_{t3} and `(t1 t2)` has kind $K_{t3} \Rightarrow *$, which means that `t2` has an arbitrary kind K_{t2} and `t1` has kind $K_{t2} \Rightarrow K_{t3} \Rightarrow *$. This type application is transformed into `t1 <t2, t3>`, which has kind $K_1 \Rightarrow K_2 \Rightarrow *$ and is therefore well-kinded according to the rule (K-TyApp) for FTKL'. In the second case, the transformation is very simple: without loss of generality, the type application has then form `t1 t2` and, as it is well-kinded by assumption, if `t2` has an arbitrary kind K_{t2} and `t1` has kind $K_{t2} \Rightarrow *$. This means in particular that `t1` is applicable to `t2`. The type application is transformed into `t1 <t2>`, and according to rule (K-TyApp) for FTKL' well-kindedness is preserved. □

Lemma 5 *For any well-kinded FTKL signature, Alg. 6.1 preserves its kind structure.*

Proof: As only type applications are transformed in this step, it is sufficient to show that the kind structure of type application is preserved. For type application we have, without loss of generality, two cases: it can be a curried type application of the form `(t1 t2) t3` or it can be an uncurried type application of type `t1 t2`. In the first case, the assumed well-kindedness of `(t1 t2) t3` implies that `t1` has kind $K_{t2} \Rightarrow K_{t3} \Rightarrow *$ (note that K_{t2} etc. are the kinds of `t2` etc., which are known by assumption), `t2` has kind K_{t2} and `t3` has kind K_{t3} (repeated application of the kind rule K-TyApp). According to Def. 5, the kind derivation of `(t1 t2) t3` is then $(K_{t2} \Rightarrow K_{t3} \Rightarrow *)\ (K_{t2})\ (K_{t3})$. Alg. 6.1 transforms this type application into `t1 <t2,t3>`, which has, according to the definition of the kind structure for an FTKL' expression (Def. 7, case (7)), kind structure $(K_{t2} \Rightarrow K_{t3} \Rightarrow *)\ (K_{t2})\ (K_{t3})$. In the second case, assumed well-kindedness of `t1 t2` implies kind $*$ for `t1 t2`, therefore `t1` has kind $K_{t2} \Rightarrow *$ and `t2` has kind K_{t2} (rule K-TyApp). Thus, the kind structure of `t1 t2` is $(K_{t2} \Rightarrow *)\ (K_{t2})$. Alg. 6.1 transforms this type application into `t1 <t2>`, which has, according to Def. 7, case (7), kind structure $(K_1 \Rightarrow *)\ (K_1)$. □

6.3 Defunctionalisation of the Codomain of a Function Signature

After uncurrying type constructors, which is a defunctionalisation of type applications, the transformation continues by eliminating higher-order functions at the level of value functions. As the processing of higher-order functions in the domain type and in the codomain type is different, defunctionalisation is divided into two sub-steps; we start with defunctionalisation of the codomain type of a function signature.

This defunctionalisation is done by Alg. 6.2. The algorithm is based on the following pattern: whenever a function type occurs in the codomain of a function signature, this function can be turned into a first-order function by introducing a new function signature that implements the functionality provided by this function. The occurrence of the higher-order function in the codomain can be replaced by a type constructor of the new type, which is defined by the new function signature. As already mentioned, function signatures in CTKL can be thought of as type definitions, and they provide a specification of the types involved in the defined mapping.

In general, if `f :: Context => a -> b` is a function that contains function types in its codomain, the new signature `TC_f_codomain_#N` is introduced, as the codomain type might contain more than one function type (by `N` we denote a counter variable and `#N` evaluates this

Input: An FTKL' function signature `f :: Context => a -> b`

Output: An FTKL' specification *spec* such that each higher-order function in the codomain of `f` has been replaced by a type constructor. In addition, *spec* contains for each higher-order function in the codomain a new function signature that encodes this type.

A1. Create an empty specification *spec*.

A2. Set the counter `N = 0`.

A3. While `find_fct (b)` for `f :: Context => a -> b` is `Just (c -> d)`:

- Set `N = N + 1`.
- Create the function signature `TC_f_codomain_#N :: Context => c -> d`.
- Apply Alg. 6.2 to `TC_f_codomain_#N` and append the result to *spec*.
- Call `replace_type(f, (c -> d), TC_f_codomain_#N)`.

A4. Put `f` into *spec*.

A5. Return *spec*.

Algorithm 6.2: `codomain_defunctionalisation:` defunctionalisation of the codomain type of an FTKL' function signature

variable.) The algorithm is performed recursively to ensure that newly created function types do not have any higher-order functions left in their codomains. As the created names are part of the recursively passed function signatures, they encode the depth of higher-order functions in the codomain:

```
f ::  Context => a -> (b -> (c -> d), d -> (c -> b))
```

is transformed by Alg. 6.2 into

```
TC_f_codomain1_codomain1 :: Context => c -> d
TC_f_codomain1 :: Context => b -> TC_f_codomain1_codomain1
TC_f_codomain2_codomain1 :: Context => c -> b
TC_f_codomain2 :: Context => d -> TC_f_codomain2_codomain1
f :: Context => a -> (TC_f_codomain1, TC_f_codomain2)
```

Thus, there is no possibility for name clashes.

The helper algorithm `find_fct` is used to search for a function type in a given type `t`, it returns `Just(f)` if `f` is the first function type found in `t` (breadth-first search) or `Nothing` if there is no function type in `t`. Another helper algorithm is `replace_type`, which replaces in a type a given type with another given type. These helper algorithms are listed in Appendix A.

The running example contains the function type `Fix <s, a> -> b` in the codomain, so Alg. 6.2 introduces a new function signature `TC_fold_codomain1 :: Bifunctor s => Fix <s, a> -> b`. As this function signature can be thought of to define a new type, and therefore a type constructor, the function type in the codomain of `fold` is replaced by this type constructor. Then, the codomain does not contain a function type anymore, as the new codomain type is a nullary type constructor, which constructs a type `TC_fold_codomain1`:

```
TC_fold_codomain1 :: Bifunctor s => Fix <s, a> -> b
fold :: Bifunctor s => (s <a, b> -> b) -> TC_fold_codomain1
```

The target language of Alg. 6.2 is still FTKL' (more precisely, as not all features of FTKL' are used, it is a subset of FTKL' that contains all specifications that do not contain higher-order functions in the codomains of function types in function signatures.) Alg. 6.2 has the same properties as Alg. 6.1:

Lemma 6 *Alg. 6.2 terminates for every FTKL' function signature, thus the codomain defunctionalisation step terminates.*

Proof: As every function type in the codomain type of a function signature is replaced by a type, and as there are only finitely many function types in the codomain type, the `while` loop terminates. Moreover, the recursive call of Alg. 6.2 is only applied to sub-expressions of the type expression that is the current input. As every type expression is finite, termination is guaranteed. □

Lemma 7 *Alg. 6.2 preserves the kind of all types the input type is constructed from. In particular, it preserves higher kinds of type constructors and type variables.*

Proof: Type constructors and type variables are not transformed by Alg. 6.2, so the claim is necessarily true. □

Lemma 8 *If Alg. 6.2 is applied to a well-kinded FTKL' function signature, its result is a well-kinded FTKL' specification. Thus, the complete codomain defunctionalisation step preserves well-kindedness.*

Proof: We have to show that every newly introduced function signature (and related type constructor) is well-kinded. For the function types that are created in step **A3** of Alg. 6.2 this is clear: the function type that is encoded by this new function signature is the codomain of a well-kinded input function type and therefore well-kinded according to rule (K-TyFun). Furthermore, we have to show that replacing a higher-order function type with a type constructor for the type introduced by the new function signature preserves well-kindedness. As the newly introduced function signature is well-kinded, the type constructor for the type associated with this function signature has kind $*$, and therefore the replacement does not affect well-kindedness. □

Lemma 9 *Applied to a well-kinded FTKL'-signature, Alg. 6.2 preserves kind structure.*

Proof: Algorithm 6.2 transforms codomains of functions, so we have to show that kind structures of codomain types are preserved. Without loss of generality we can assume that the function type is of form `f :: a -> (b -> c)`, which is transformed into

```
1 TC_f_codomain :: b -> c
2 f :: a -> TC_f_codomain
```

The untransformed signature has kind $K_a \Rightarrow K_b \Rightarrow K_c$, and the transformation does not change the kind structure of the domain type, which therefore remains K_a. For the codomain type `(b -> c)` a new function signature `TC_f_codomain :: b -> c` is added to the specification. As this signature (literally) implements the codomain type, its kind structure is equal to that of the codomain type, thus it is $K_b \Rightarrow K_c$. The definition of kind structure for function signatures can be applied to yield that the kind structure of the transformed signature is still $K_a \Rightarrow K_b \Rightarrow K_c$. □

6.4 Defunctionalisation of the Domain of a Function Signature

After the codomain type of a function signature has been defunctionalised, the next step is to defunctionalise the domain type. This domain defunctionalisation is done by Alg. 6.3 and Alg. 6.4. As CTKL treats functions in the domain type differently from those in the codomain type, these algorithms work quite differently from those in the previous sections. The main idea of Alg. 6.3 and Alg. 6.4 is to scan for occurrences of function types in the domain of the function signature. Algorithm 6.3 scans a function signature for occurences of function

Input: An FTKL' function signature `f :: Context => a -> b`

Output: A CTKL function signature `TC_f :: Context' => a' -> b` such that every function type used within a domain type is replaced by a type variable and requirements in `Context'` according to Alg. 6.4.

A1. Set the counter `N = 0`.

A2. If `find_fct (a)` for `f :: Context => a -> b` results in `Just(c -> d)` compute `(a', Context', N') = concept_encode_HOF(a, Context, N)`. Collect all type variables from `Context'`, `a'`, and `b` in the parameter list `<param>`. Then return the CTKL function signature `TC_f <param> :: Context' => a' -> b`.

A3. Otherwise, collect all type variables from `Context`, `a`, and `b` in the parameter list `<param>`. Then return the CTKL function signature `TC_f <param> :: Context => a -> b`.

Algorithm 6.3: `domain_defunctionalisation:` defunctionalisation of the domain type of an FTKL' function signature

types in the domain. For each function type found, Alg. 6.4 is called and introduces a type variable. Then, concepts are used to encode the type of the function that the type variable replaces. Concepts for function types are used to state the requirement that the type variable is a placeholder for function types, type identity requirements encode the domain and the codomain type of the function. Thus, the context contains a complete encoding of the function type that was replaced.

Applied to our example, Alg. 6.3 replaces the function type `s <a, b> -> b` in the domain of `fold` with a type parameter:

```
1  TC_fold_codomain1 <s, a, b, fun1> ::
2     Bifunctor s,
3     Function fun1,
4     SameType fun1.Domain1 (s <a, b> ),
5     SameType fun1.Codomain b
6                                    =>  Fix <s, a> -> b
7  fold <s, a, b, fun1> ::
8     Bifunctor s,
9     Function fun1,
10    SameType fun1.Domain1 (s <a, b> ),
11    SameType fun1.Codomain b
12                                     => fun1 -> TC_fold_codomain1
```

Three requirements are necessary to encode the type of the new type parameter `fun1`: First, the requirement `Function fun1` states that `fun1` is a placeholder for a unary function type. Then, the requirements `SameType fun1.Domain1 (s <a, b>)` and `SameType fun1.Codomain`

Input: An FTKL' type `t`, a context `C`, and a counter `N`.

Output: A triple `(t', C', N')`, consisting of the type `t'`, which is free from function types, a context `C'` that includes `C`, and a counter `N'`. Every function type from `t` is replaced by a type variable. In addition `C'` is extended with requirements that encode the signature function type by using function concepts and type identity concepts. Further, `N'` is the sum of `N` and the number of replaced function types.

A1. Case distinction on the type of `t`:

- `t1 -> t2`:
  `(t', C', N')` = `concept_encode_HOF (t1, C, N)`
  `(t", C", N")` = `concept_encode_HOF (t2, C', N')`
  Create a type variable `fun#N"` and replace `t1 -> t2` with it.
  Let *D* be the number of arguments of `t1`. Add the requirement `Function#D fun#N"` to `C"`.
  If `t1` is a tuple of type `(t_1, ..., t_T)` for every type `t_i` add the requirement `SameType fun#N".Domain#i t_i` to `C"`. Otherwise, add the requirement `SameType fun#N".Domain1 t1` to `C"`.
  Return `(fun#N", C", (N"+1))`.
- `(t_1, t_2, ..., t_T)`:
  for i=1, ..., T
  - Compute `(t', C', N')` = `concept_encode_HOF (t_i, C, N)`.
  - Set `t_i := t'`.
  - Set `C := C'`.
  - Set `N := N'`.

  Return (`(t_1, t_2, ... ,t_T)`, C, N).
- `(t)`:
  Return `concept_encode_HOF (t, C, N)`.
- `t1 <t2_1, ... ,t2_T>`:
  for i=1, ..., T
  - Compute `(t', C', N')` = `concept_encode_HOF (t2_i, C, N)`.
  - Set `t2_i = t'`.
  - Set `C = C'`.
  - Set `N = N'`.

  Return (`t1 <t2_1, ... ,t2_T>`, C, N).
- otherwise:
  Return `(t, C, N)`.

Algorithm 6.4: `concept_encode_HOF`: encoding function types in the domain of a function signature with concept-constrained type variables

`b` encode the signature of the function type replaced by `fun1`. The resulting function type definition no longer contains any function types; they are all replaced by type constructors or

type variables. Again, Alg. 6.3 and Alg. 6.4 have the properties of the algorithms presented before. We only have to prove them for Alg. 6.4, as they are then trivially fulfilled for Alg. 6.3.

Lemma 10 *Algorithm 6.3 and Alg. 6.4 terminate for every FTKL' function signature, thus the domain defunctionalisation step terminates.*

Proof: All recursive calls of Alg. 6.4 are applied to proper sub-expressions of the current input expression. Therefore, termination is guaranteed. □

Lemma 11 *Applied to a well-kinded FTKL' expression Alg. 6.3 and Alg. 6.4 preserve the kind of all types the input type is constructed from. In particular, they preserve kinds of type constructors and type variables, thus higher kinds.*

Proof: Type constructors and type variables are not transformed by either algorithm. □

Lemma 12 *If Alg. 6.3 and Alg. 6.4 are applied to a well-kinded FTKL' function signature, the result is a well-kinded CTKL specification. Thus, the domain flattening step preserves well-kindedness.*

Proof: Alg. 6.3 and Alg. 6.4 transform domain types of functions, so we have to show that well-kindedness is preserved for the transformed domain types. Without loss of generality we can assume that the function type has the form `f :: (a -> b) -> c`. We can further assume that `c` does not contain any function types as they have been removed before by Alg. 6.2. The well-kindedness of the input implies that `a -> b` is of kind $*$ and `c` is of kind $*$. The transformation transforms the function signature `f :: (a -> b)-> c` into

```
1 f <a,b,c,fun1> ::
2     Function fun1,
3     SameType fun1.domain1  a,
4     SameType fun1.codomain b,
5                     => fun1 -> c
```

The newly introduced type variable `fun1` encodes the function signature `a -> b` in the context. The requirements state that `fun1.Domain1` is equal to `a` and `fun1.Codomain` is equal to `b`. Therefore, `fun1.Domain1` and `fun1.Codomain` have kind $*$ according to rule (K-SameType2). Then, according to rule (K-TyFunParam1), `fun1` is well-kinded, and thus, according to rule (K-CTKL-TySig), the transformed function signature is well-kinded. □

Lemma 13 *For any well-kinded FTKL'-signature, Alg. 6.3 and Alg. 6.4 preserve its kind structure.*

Proof: Alg. 6.3 and Alg. 6.4 transform domains of functions, so we have to show that kind structures for them are preserved. Without loss of generality we can assume again that the function type is of form `f :: (a -> b)-> c`, which is transformed as listed above. The untransformed signature has kind structure $(K_a \Rightarrow K_b) \Rightarrow K_c$. The transformation does not change the kind structure of the codomain type, which therefore remains K_c. For the domain function type `(a -> b)` a type variable `fun1` is introduced, and the type `(a -> b)` is encoded with requirements in the context. Therefore, according to Def. 7, case (4) and case (5), the kind structure of `fun1` is given by $K_a \Rightarrow K_b$. Thus, the kind structure of the transformed function signature remains $(K_a \Rightarrow K_b) \Rightarrow K_c$. □

6.5 Encoding Higher Kinded Type Variables

CTKL requires one to express higher kinds of type variables in the context, i.e., in the form of requirements. To encode higher kinds, the requirements `TypeConstructorN` are used. The actual encoding is then straightforward: the kinds of all type variables are given (that was an assumption for the whole transformation), and if they are different from $*$, an appropriate requirement is added.

Input: A function signature `f<p> c :: a -> b` *that does not contain higher-order function types.*

Output: A function signature `f<p> c' :: a -> b` *such that all higher kinds of type variables are expressed in the context.*

A1. Set `c'` = `encode_higher_kinds_in_type(a,c)`.

A2. Set `c''` = `encode_higher_kinds_in_type(b,c')`.

A3. Return `f<p> c'' :: a -> b`.

Algorithm 6.5: `encode_higher_kinds`: encode higher kinds of type variables in a CTKL function signature

Applied to the running example, Alg. 6.5 adds a requirement on the type parameter `s`:

```
1  TC_fold_codomain1 <s, a, b, fun1> ::
2     Bifunctor s,
3     Function fun1,
4     SameType fun1.Domain1 (s <a, b> ),
5     SameType fun1.Codomain b,
6     TypeConstructor2 s              => Fix <s, a> -> b
7  fold <s, a, b, fun1> ::
8     Bifunctor s,
9     Function fun1,
10    SameType fun1.Domain1 (s <a, b> ),
11    SameType fun1.Codomain b,
12    TypeConstructor2 s              => fun1 -> TC_fold_codomain1
```

The important properties of Alg. 6.5 and Alg. 6.6 are:

Lemma 14 *Alg. 6.5 and Alg. 6.6 terminate for every CTKL function signature.*

Proof: Alg. 6.6 contains only recursive calls that refer to subterms of the input term. Termination of Alg. 6.5 follows immediately. □

Lemma 15 *Alg. 6.5 and Alg. 6.6 preserve kinds and therefore well-kindedness.*

Proof: Both algorithms do not transform any type in the function signature and therefore can not change any kind. □

Lemma 16 *Alg. 6.5 and Alg. 6.6 preserve the kind structure of a CTKL type.*

Proof: Both algorithms do not transform any type and therefore do not change any kind. Thus, the kind structure is preserved. □

Input: A CTKL type `t` *and an context* `c`

Output: A context `c'` *that contains* `c` *and* `TypeConstructor` *requirements for all higher kinded type variables in* `t`*.*

A1. Case distinction on the type of `t`:

- `t1 -> t2`:
  `c = encode_higher_kinds_in_type(t1, c)`
  `c = encode_higher_kinds_in_type(t2, c)`
  return `c`
- `(t_1, t_2, ..., t_T)`:
  for i=1, ..., T
  - `c = encode_higher_kinds_in_type(t_i, c)`

  return `c`
- `(t)`:
  - `c = encode_higher_kinds_in_type(t, c)`

  return `c`
- `t1 <t2_1, ... ,t2_T>`:

```
c = encode_higher_kinds_in_type(t1, c)
for i=1, ..., T
    c = encode_higher_kinds_in_type(t_i, c)
return c
```

- otherwise:
  if `t` is a type variable:
  - if `t` has a higher kind (that means a kind different from $*$), add an appropriate `TypeConstructor` requirement to `c`.
  - return `c`

  if `t` is a type constructor:
  - return `c`

Algorithm 6.6: `encode_higher_kinds_in_type`: encode higher kinds of type variables in a CTKL type

6.6 Optimising Context

Alg. 6.4 in the domain flattening step encodes higher-order function types in the domain of a function in a straightforward manner, by simply introducing a type parameter for every type variable. In general, however, this causes some redundancy, as the type variable could be replaced by a member type of a function type variable. In the running example, the type parameter `b` is redundant, as it can be replaced with the `Codomain` member type of `fun1`.
Alg. 6.7 is applied to remove such redundant type parameters and requirements; it can be applied to any CTKL function signature.

This algorithm scans the requirements of a generic function for occurrences of type identity

Input: A CTKL function signature `f<p> Context :: a -> b`.

Output: A CTKL function signature `f<p'> Context' :: a -> b` where the requirements and the type parameters are optimised such that `<p'>` does not contain any type variable that can be substituted by a member type of a function type variable. Furthermore, `Context'` contains neither duplicates nor trivial identities.

A1. Let `r` iterate over the requirements in `Context`.

- If `r` is a type identity requirement (`SameType`) between a type parameter `x` and the same type parameter `x`:
  - Remove `r` from the requirements in `Context`.
- If `r` is a type identity requirement between a type parameter `x` and a function type member `y`:
  - Remove `x` from the type parameter list `p`.
  - Replace all occurrences of `x` in the domain type `a` with `y`.
  - Replace all occurrences of `x` in the codomain type `b` with `y`.
  - Remove `r` from the requirements in `Context`.

A2. Remove duplicate requirements in `Context`.

A3. Return `Context`.

Algorithm 6.7: `optimise_requirements`: optimise the context (requirements and type parameters) of a CTKL function type signature

requirements between type variables and member types of functions. For such requirements the type parameter and the requirement itself are redundant as the type parameter can be replaced with the member type of the function variable, and the requirement then becomes trivial. Finally, all duplicates are removed from the requirements. Duplicates may occur, as in the encoding of higher kinds the same higher-kinded type variable might be decoded several times.

Applied to the running example, Alg. 6.7 removes the type parameter `b` and replaces its occurences with `fun1.Codomain`:

```
1  TC_fold_codomain1 <s, a, fun1> ::
2     Bifunctor s,
3     Function fun1,
4     SameType fun1.Domain1 (s <a, fun1.Codomain> ),
5     TypeConstructor2 s            => (Fix <s, a> ) -> fun1.Codomain
6  fold <s, a, fun1> ::
7     Bifunctor s,
8     Function fun1,
9     SameType fun1.Domain1 (s <a, fun1.Codomain> ),
10    TypeConstructor2 s            => fun1 -> TC_fold_codomain1
```

By this replacement, the type identity constraint `SameType fun1.Codomain b` becomes trivial and is therefore removed from the requirements.

Alg. 6.7 has the following properties:

Lemma 17 *Alg. 6.7 terminates for every CTKL function signature.*

Proof: There is only one iteration over the finite list of requirements. □

Lemma 18 *Alg. 6.7 preserves kinds and therefore well-kindedness.*

Proof: The algorithm does not transform any type in the function signature and therefore can not change any kind. □

Lemma 19 *Alg. 6.7 preserves the kind structure of CTKL type.*

Proof: The algorithm does not transform any type and therefore does not change any kind. Thus, the kind derivation is preserved. □

Lemma 20 *For the input and the output of Alg. 6.7 the following relations are valid:* `<p'>` $\subseteq$ `<p>` *and* `Context'` $\subseteq$ `Context`.

Proof: For both `p` and `Context`, elements can only be deleted, but not inserted. □

Lemma 21 *The list of type parameters* `<p'>` *that is returned by Alg. 6.7 contains only type variables that can not be replaced by member types of function type variables. In that sense,* `<p'>` *is optimal.*

Proof: By construction, all type variables that can be replaced by a member type of a function type variable are replaced. □

6.7 The Complete Transformation and its Properties

For a given functional specification in FTKL, which consists of function signatures and type class definitions, the steps described above are performed in the presented order. First function signatures are transformed and then function signatures that are members of type classes. For every function signature, the codomain type is defunctionalised, yielding a list of function

Input: A well-kinded expression `f` *in FTKL.*
Output: A well-kinded expression `F` *in CTKL such that:*

- *Type variables are mapped to type variables with the same name.*
- *Type constructors are mapped to type constructors with the same name.*
- *Higher-order functions in the codomain are mapped to type constructors. Definitions for these types are introduced.*
- *Higher-order functions in the domain are mapped to type variables. Their signature is encoded in the context using the concepts* `FunctionN` *and* `SameType`.
- *Higher kinds are encoded in the context using the concepts* `TypeConstructorN`.
- *Concept definitions are mapped to concept definitions, concept names are preserved. Member functions of the concept are transformed into member types of the concept according to the transformation.*

Figure 6.1: Specification of transformation $\mathscr{T}$

signatures. For every element of this list the domain is defunctionalised, higher-kinded type variables are encoded, and type parameters are optimized. This finally yields a list of function signature in CTKL, which can then be translated to concept-enabled programming languages. The input and output parameters of the transformation $\mathscr{T}$, including its functional specification, are listed in Fig. 6.1. The transformation $\mathscr{T}$ has some crucial properties that follow immediately from the properties we have already shown for each step. Therefore, the proofs are trivial. In particular:

Theorem 1 *The transformation $\mathscr{T}$ terminates for every well-kinded FTKL function signature.*

Proof: Follows immediately from Lemma 3, Lemma 6, Lemma 10, Lemma 14, and Lemma 17. □

Theorem 2 *The transformation $\mathscr{T}$ preserves well-kindedness. Thus, it transforms well-kinded FTKL function signatures into well-kinded CTKL function signatures.*

Proof: Follows immediately from Lemma 4, Lemma 8, Lemma 12, Lemma 15, and Lemma 18. □

Theorem 3 *The transformation $\mathscr{T}$ preserves the kind structure. Thus, the CTKL function signatures that are output of $\mathscr{T}$ have the same kinds as the input FTKL function signatures.*

Proof: Follows immediately from Lemma 5, Lemma 9, Lemma 13, Lemma 16, and Lemma 19. □

Theorem 4 *The transformation $\mathscr{T}$ produces contexts that are optimal in the sense of Lemma 21.*

Proof: Follows immediately from Lemma 21.

Chapter 7

Backends

The transformation described in the previous chapter turns FTKL function signatures into CTKL function signatures. As both languages are kernel languages that do not provide any capabilities to actually implement computations, implementations have to be done in actual programming languages. However, implementations are guided by function signatures, and CTKL function signatures can be automatically translated into several programming languages. Such translations are called backends of our transformational approach, and there are different backends for different languages.

Since CTKL function signatures are based on concepts, good candidates for target languages are those that support concept-controlled genericity. Such support is provided by languages from very different programming paradigms, and it is not clear how to implement technical details in different paradigms. Thus, we start this chapter with some general considerations of useful language features for target languages (Sect. 7.1). Afterwards, we discuss three backends for three very different languages. The existence of these three backends shows that concept-based functional higher-order, typed signatures can be expressed in object-oriented as well as hybrid and functional languages.

The first backend we provide is for ConceptC++, an extension of C++ that provides explicit support for concept-controlled polymorphism. The backend demonstrates how CTKL can be translated into a language with object-orientation and with generics that are orthogonal to object-orientation, that means, generics that are not integrated in the object system of the language. In C++, templates are a feature that is completely independent from object-oriented features. Thus, concepts in ConceptC++ are also independent from object-oriented features, and in particular concepts are not embedded in the subtyping hierarchy of C++ classes. The ConceptC++ backend can be seen as a blueprint for other languages where generics are available that are independent of object-orientation. Among others, Ada and D are languages of that kind. However, as we will discuss in Sect. 7.1.4, of these languages, ConceptC++ is the only one that provides explicit support for concepts and concept-controlled polymorphism.

Our second backend is for Scala, and it shows that the transformational approach can be used with a pure object-oriented language as backend (we use Scala in a purely object-oriented way). Generics, with concepts and concept modellings, are embedded directly in the subtyping hierarchy of such pure object-oriented languages. In particular, concept modelling is implemented by inheritance, and requirements are implemented by subtyping constraints. This has severe technical implications, which we will discuss in Sect. 7.3. However, as these limitations are recognised by the Scala developers, there is a separate implementation of concepts and

concept maps that we will also discuss. The Scala backend might seem of limited practical use as Scala already provides syntax for functional features. However, there is a good reason why the Scala backend is useful. It can be seen as blueprint for backends in other languages that provide object-oriented generics. Examples of such languages are Java, C#, and Eiffel. Among these languages, which are all object-oriented without functional syntax extensions, we have chosen Scala because it provides the best support for generic programming. In fact, as we will discuss in Sect. 7.1.4, all these other languages miss essential features of generics.

The third backend shows that it is possible to define backends for non-object-oriented languages that provide sufficient support for generic programming. The main question in these languages is how to implement a function if not as a function type or a function class. We show that it is possible to define a function using a concept (type class) and algebraic datatypes. Thus, this backend is a blueprint for backends in other non-object-oriented languages like standard-ML.

Every backend consists of three parts. The first part is a rewriting transformation that translates CTKL function signatures into the actual language. The second part of every backend is a library containing the language constructs of CTKL that are not provided by the target language itself. The library includes in particular the predefined concepts that are used to encode higher-order constructs in CTKL, that is, the function concepts and the type constructor concepts. Further, the backend library usually provides an implementation of tuple types, as they are available in CTKL but not sufficiently supported by most non-functional languages (for instance, C++ provides just a `pair` type, but no type for tuples of higher arity). The third part of the backend transformation is an optional user-defined type constructor mapping. As mentioned in Sect. 6.7, the transformation from FTKL to CTKL preserves type constructor names. In the backend transformation, the user has the possibility to provide a specific mapping for these names. For instance, if the type constructor `Int` in Haskell should be mapped to the C++ type constructor `long int` in a specific application of the transformation system, this can be declared in the mapping. Further, type constructors can also be mapped to type variables with optional requirements. In C++, for instance, the type constructor `List` is usually mapped to a type variable that is constrained to be a `std::Container`. Fig. 7.1 shows an example of such a mapping. The specific mappings are written line-wise and declare the type constructor that should be mapped, followed by a syntactic separation (`- -`), and the target of the mapping, using `tc:`, `tv:`, and `req:` to declare the kind of construct the type constructor is mapped to. Type constructors that are not found in this mapping are mapped to type constructors with identical names.

```
Bool  -- tc:  bool
Nat   -- tc:  int
Int   -- tc:  long int
List  -- tv:  Cont
List  -- req: std::Container<Cont>
```

Figure 7.1: A user-defined type constructor mapping

7.1 Language Features

As our approach builds on generic programming with concepts, suitable target languages for backends are those that provide generic programming with support for concepts. In particular, CTKL features can be easily expressed in such languages. As generics are increasingly popular and important in software development, nowadays many programming languages provide basic support for them. However, for our approach we need languages that provide better support than elementary generics. In terms of the classification of genericity from Chap. 3 we need languages that support generic programming beyond genericity by type. In particular, the transformation turns higher-order constructs into concept-based expressions, thus, target languages need to provide genericity by structure.

Beyond basic generics, we need some language constructs to constrain type parameters with concepts. With concepts, or their replacements, it should be possible to constrain multiple types simultaneously; such concepts are called multi-type concepts. Moreover, it should be possible to place more than one constraint on a type parameter. This feature is known as multiple constraints. For the expression of type constructors as concepts we need some form of type computations. Associated types have proved valuable in such type computations, thus, a language that serves as a backend language should provide associated types in concepts and in types. Further, constraints on associated types should be allowed and a mechanism for creating shorter names for types should be provided.

As discussed by Garcia et al. [2003, 2007] such an extensive support for generic programming is still rare. In particular, there are three languages that provide outstanding support, so that we decided to use them as backend languages for our approach. Table 7.1 gives an overview of the support of concept-controlled programming in the three backend languages. Below we will discuss how the necessary language features are implemented within these languages. We will use the same minimal example to explain the syntax and application of generic programming for all these languages.

Language	ConceptC++	Scala	Haskell
generic programming	templates	generics	type parameters
concept	concept	trait	type class
constraint	requirement	type parameter bounds	context
modelling	concept mapping	inheritance from trait	instance declaration
modelling paradigm	retroactive	proactive	retroactive
associated types	typedef	type (keyword)	type (keyword)

Table 7.1: Support of generic programming in different languages

7.1.1 ConceptC++

Generic programming in C++ is based on C++'s template mechanism [Vandervoorde and Josuttis, 2002] and has a long tradition. The most influential generic library in C++ is the Standard Template Library (STL) [Stepanov and Lee, 1994]. In the design of the STL, concepts played a big role from the beginning. Iterators, which are used to traverse through container data structures, are no concrete entities. Instead, they implicitly define a set of requirements on

types that may be used as iterators. Any instance of, say, `InputIterator`, must provide certain operations: a test for equality, an incrementation operator, a dereference operator, and so on.

However, concepts are not part of the C++ language; there is no way to declare a concept in a program, or to declare that a particular type is a model of a concept. In standard C++, concept modelling has to be done implicitly by providing the necessary entities with appropriate names. Nevertheless, concepts are an extremely important part of the STL, as the whole library is centred around an implicit concept taxonomy defining different iterators.

C++ concepts [Gregor et al., Gregor and Stroustrup, 2006] are an extension to C++ and its template system that introduces language support for concepts. C++ concepts were proposed for inclusion in the C++0x standard [Gregor et al., 2008]. However, the C++0x committee decided to remove concepts from the draft standard, as they were not considered to be mature enough for C++0x[Gregor, b]. They are still a candidate for inclusion in future standards, and a rudimentary version of C++-concepts has been implemented as an experimental compiler [Gregor, a].

```
1  concept Invertable<class T> {
2      typename inverse_type;
3      inverse_type invert(T x);
4  }
5
6  template<class T>
7  requires
8      Invertable<T>
9  inverse_type invert(T x) {
10     return inverse(x);
11 }
12
13 struct Apple { int r; }
14 struct Orange { float r; }
15
16 concept_map Invertable<Apple> {
17     typedef Orange inverse_type;
18     inverse_type invert(Apple a) {
19         Orange o;
20         o.r = 1/a.r;
21         return o;
22     }
23 };
24
25 Apple a1.r = 4;
26 Orange o1 = invert(a1);
```

Figure 7.2: Generic inversion in ConceptC++

Fig. 7.2 shows an example of ConceptC++ code. A concept, as the concept `Invertable`, is declared using the `concept` keyword followed by the list of its type parameters (in the example `T`). The keywords `class` and `typename` in the parameter list are freely exchangeable. In its body, the concept lists associated entities. In the example above, there are two associated entities: the associated type `inverse_type` and the associated operation `invert`. Generic functions (or classes) are provided by templates, which are defined using the keyword `template`. Type parameters in templates can be constrained using the `requires` keyword. In the example, the

`invert` function is generic as it is parameterised by the type parameter `T` that is constrained with the requirement `Invertable<T>`. Finally, a model of a concept is declared via a `concept_map`, which defines the associated entities of the concept. In the example, we declare `Apple` to be a model of the concept `Invertable`.

Concepts are an experimental feature, since they are not included in the C++ standard yet and therefore not supported by most compilers. We use traits [Myers, 1995] to simulate concepts. Like concepts, traits are template classes that provide bindings for associated entities by specialisations for particular types. Unlike concepts, however, they do not provide the possibility to type check templates without instantiation.

C++ templates are actually code generators: whenever instantiated with a type, a copy of the template code is created. This copy of the code is internally visible, but hidden to the programmer. In standard C++, only instantiated templates can be type-checked. Concepts enable templates to be type-checked separately without instantiation [Zalewski, 2008], and thus improve the readability of compiler error messages. If a template is instantiated with a type that does not provide the required interface, the compiler generates an error message. This error message usually contains template parameters spelled out in full, and in addition error messages often do not immediately refer to the actual location of the error, but list the full instantiation hierarchy instead. As a consequence, error messages become rather long and verbose. Concepts avoid such complicated error messages, as they enable the compiler to precisely state which associated entities are missing for which type.

7.1.2 Scala

Scala [Odersky et al., 2004] is one of the few object-oriented languages that provide support for higher-order functions and higher kinds, even if support for the latter was added only recently [Moors et al., 2007, 2008, Moors, 2009]. Thus, Scala is sometimes referred to as an object-functional language. However, at its core Scala is a pure object-oriented language in the sense that every value is an object. Data types and behaviour of objects are described by classes and traits, and functional syntax is added as syntactic sugar on top of this object-oriented structure.

Functions are defined by the `def` keyword. For example, the increment function on integer values could be written:

```
1 def inc (x : Int) : Int = x + 1
```

This function definition comprises a function signature: the `inc` function has the signature `inc : Int -> Int` (in mathematical notation). Functions in Scala are first-class values, so higher-order functions are supported. In addition, currying and anonymous functions are supported.

Generic programming in Scala is supported by so-called generics. Functions and classes can be parameterised by types. For example, function composition can be implemented as follows:

```
1 def comp[a,b,c] (f : b => c) (g : a => b) (x : a) : c = f (g (x))
```

Type parameters can be constrained using traits, a mechanism that was originally introduced to enable multiple inheritance in object hierarchies [Schärli et al., 2003]. Traits implement the idea of concepts as they consist of abstract methods and associated types. In contrast to Java interfaces, for instance, traits allow for concrete methods. These implementations can be inherited from a trait. To constrain type parameters Scala provides two possibilities.

The first one is to use traits and subtyping, and is thus a clean object-oriented implementation of generics that is similar to Java generics with interfaces. Fig. 7.3 shows an implementation of generic inversion in object-oriented Scala generics. Concepts are modelled by traits, using the `trait` keyword followed by a list of the type parameters of the trait. Associated entities of the concept are listed in the body of the trait. Generic functions (or classes) are defined by listing their type parameters in squared brackets. Constraints on type parameters are expressed directly in the type parameter list, using Scala's subtyping capabilities. The notion `T <: Invertable[T]` in the example means that `T` can only be replaced by types that are subtypes of the trait `Invertable` instantiated with `T`.

A major difference between C++ templates and object-oriented Scala generics is that in C++, models of a concept can be declared retroactively, while in Scala model declarations have to be integrated proactively in the design of class hierarchies. In C++, a concept and a class that is a model of that concept can be developed independently. A concept mapping is used to define name bindings for the associated entities and thus glues together the concept and the class. For a Scala class to model a concept, this has to be declared directly in the class definition. The class has to extend the trait (using the `extends` keyword) that represent the concepts the class is intended to model. Because inheritance has to be declared when a class definition is given, retroactive modeling is not possible. In the example of Fig. 7.3 the declaration `class Apple(rating : Int)extends Invertable[Apple]` posits that the type `Apple` extends the trait `Invertable[Apple]` and is therefore a subtype of `Invertable[Apple]`. The associated entities of the trait `Invertable[Apple]` have to be provided by overloading—they are defined in the body of the class `Apple`. Another disadvantage of object-oriented generics is that they do not support multi-parameter type classes.

```
1  trait Invertable[T] {
2      type inverse_type;
3      def invert(x : T) : inverse_type;
4  }
5
6  class Orange(rating : Float) {
7      val r = 1/rating;
8  }
9
10 class Apple(rating : Int) extends Invertable[Apple] {
11     val r=rating;
12     type inverse_type = Orange;
13     def invert(x : Apple) : inverse_type = {
14         new Orange(x.r)
15     }
16 }
17
18 def invert[T <: Invertable[T]] (x : T) : T#inverse_type = {
19     x.invert(x);
20 }
21
22 val a1 = new Apple(4);
23 val o1 = invert(a1);
```

Figure 7.3: Generic inversion in Scala—with subtyping

```
1 trait Invertable[T] {
2     type inverse_type;
3     def invert(x : T) : inverse_type;
4 }
5
6 class Orange(rating : Float) {
7     val r = 1/rating;
8 }
9
10 class Apple(rating : Int) {
11     val r = rating;
12 }
13
14 implicit object invertableApple extends Invertable[Apple] {
15     type inverse_type = Orange;
16     def invert(x : Apple) : inverse_type = {
17         val o = new Orange(x.r)
18         o
19     }
20 }
21
22 def invert[T](x: T)(implicit inv_x: Invertable[T]): Invertable[T]#inverse_type = {
23     inv_x.invert(x);
24 }
25
26 val a1 = new Apple(4);
27 val o1 = invert(a1);
```

Figure 7.4: Generic inversion in Scala—with implicits

To overcome the limitations and disadvantages of object-oriented Scala generics, Scala 2.0 introduced implicit parameters, which implement the idea of implicitly passed dictionaries. Implicit parameters allow the compiler to deduce some parameters implicitly; and they can therefore be used to emulate concepts ([Odersky, 2006]). Fig. 7.4 shows another implementation of the generic inversion example in Scala, this time with implicit parameters. The concept `Invertable` is implemented as before, but now modeling can be done retroactively. The class `Apple` does not extend the trait. Instead, an implicit object `invertableApple` that extends `Invertable[Apple]` is defined. This implicit object defines the associated entities of the trait `Invertable`. The generic method `invert` now expects one more parameter. This is an implicit one, declared by `implicit inv_x: Invertable[T]`. This parameter has not been given explicitly but can be inferred by the compiler. It passes the dictionary that is created by the implicit object definition.

Another major difference between C++ templates and Scala generics (in both versions) is the compilation model. As mentioned before, C++ templates generate separate code for every instance of the template. In contrast, Scala generics generate polymorphic byte code that is instantiated with type arguments at run time.

7.1.3 Haskell

Generic functions in Haskell are implemented by means of parametric polymorphism; every function can be parameterised over types. Every generic function can be supplemented with

a context stating requirements on type parameters. Fig. 7.5 shows our example in Haskell.

```
1 module Main where
2
3 class Invertable t where
4     type Inverse_type t
5     inverse :: t -> Inverse_type t
6
7 data Orange = OrangeC Float
8
9 data Apple = AppleC Float
10
11 instance Invertable Apple where
12     type Inverse_type Apple = Orange
13     inverse (AppleC r) = OrangeC(1/r)
14
15 invert :: (Invertable t) => t -> Inverse_type t
16 invert x = inverse x
17
18 a1 = AppleC 4
19 o1 = invert a1
```

Figure 7.5: Generic inversion in Haskell

Concepts are implemented as type classes using the `class` keyword. Models are declared as instances using the `instance` keyword.

Generic programming in Haskell is somewhat similar to generic programming in ConceptC++. Models of a concept can be declared retroactively; in the example, the type `Apple` is first defined and afterwards declared to be an instance of the concept `Invertable`. In this model declaration, the associated entities are provided by overloading, and in the generic function those overloaded entities are chosen accordingly. As in ConceptC++, the concept system of Haskell enables type checking of generic code separately from its instantiation. However, an important difference to ConceptC++ is the compilation model: Haskell enables separate compilation, generic algorithms can be compiled to polymorphic code that can be instantiated with type arguments at run time.

7.1.4 Other languages

Generic programming is supported by many programming languages. However, many languages provide support that lacks important features. In this subsection we consider some of these languages. We do not provide backends for them, but sketch the design and limitations of such backends.

Generic programming in Java is provided by Java generics [Gosling et al., 2005], which are integrated with object-orientation. Concepts can be expressed using interfaces, and modeling corresponds to implementing interfaces. Constraints on type parameters are given via interfaces. However, interfaces do not provide the possibility to define entities other than methods. Thus, associated types can not be expressed in Java interfaces. This limitation makes the implementation of our example cumbersome. As illustrated in Fig. 7.6, the main problem is that the type for the inverse cannot be declared as an associated type. Thus, `Invertable` is no longer a predicate on a type, but a binary relation between two types. This leads to long type parameter

```
1 public interface Invertable<T, Inverse_Type> {
2     Inverse_Type inverse();
3 }
4
5 public class Orange {
6     public double r;
7     public Orange(double rating) {
8         r = rating;
9     }
10 }
11
12
13 public class Apple implements Invertable<Apple, Orange> {
14     public double r;
15     public Apple(double rating) {
16         r = rating;
17     }
18     public Orange inverse() {
19         Orange o = new Orange(1/r);
20         return o;
21     }
22
23 }
24
25 public static <IT, T extends Invertable<T, IT> > IT inverse (T x) {
26     return x.inverse();
27 }
28
29 public static void main (String argv[]) {
30     Apple lApple   = new Apple (4);
31     Orange lOrange = inverse(lApple);
32 }
```

Figure 7.6: Generic inversion in Java

lists, which are difficult to read.
For CTKL backend languages, associated types are an essential feature. They allow realising type computations, and these type computations are the basis of our implementation of type constructor concepts. Type constructors are types that can be applied to other types, and this application is realised by storing the argument as an associated type. The situation in C# and Eiffel is quite similar to that in Java. Both support generics that are embedded in the object system of the language, but neither provides the possibility to define associated types. Thus, all these language are not suitable backend languages for CTKL. For each of these languages, we could provide a backend that captures function concepts, but we would fail to provide one with type constructor concepts.

The D programming language is an object-oriented, imperative, multi-paradigm, system programming language. While originating from a re-engineering of C++, it is meanwhile an independent language even though it is mainly influenced by C++. In D, there is a template mechanism that is based on C++ templates. Even if D templates allow a simple form of constraints on type parameters, these constraints follow a rather pragmatic approach. A constraint is expressed as a template itself, but a template that contains code that is never executed but

only compiled. If the type does not meet the requirements expressed by the template, compilation fails. This way of expressing constraints allows a simulation of concepts. The running example can be implemented in D as shown in Fig. 7.7. The code in template `isInvertable` is

```
template isInvertable(T) {
    const isInvertable =
    __traits(compiles, (T t) {
        T x;
        T.inverse_type x_inv;
        x_inv = x.inverse();
    });
}

struct Orange { double r; }

struct Apple {
    alias Orange inverse_type;
    double r;
    Orange inverse() {
        Orange lO;
        lO.r= 1/r;
        return lO;
    }
}

T.inverse_type inverse (T)(T t)
    if (isInvertable!(T))
{
    return t.inverse();
}

Apple app;
app.r = 4;
Orange ora = inverse(app);
```

Figure 7.7: Generic inversion in D

compiled but never executed. If a template parameter is to be constrained, this is done by the compile-time if statement `if (isInvertable!(T))`. If the type that is provided for the type parameter `T` does not provide all types and operations used in the code of `isInvertable`, the instantiation `isFunction(T)` fails to compile for that type, and the compiler states that. However, these constraints do not allow separate compilation or type checking, they are rather a form of static assertion. Otherwise, generic programming in D is rather similar to C++, so we can provide a D backend for CTKL if needed.

Ada provides generics in the form of generic units, which are packages or subprograms that are parameterised on one or more generic formal parameters. Such generic formal parameters can be values, variables, constants, types, subprograms, or even an instance of another, designated generic unit. Formal parameters can be constrained with function signatures, these function signatures can be used to express concepts [Duret-Lutz, 2001]. However, Ada does not provide the possibility to bundle different signatures into explicit concepts. For every parameter, every required signature has to be given explicitly, which becomes rather inconvenient especially in the case of complex concepts with many associated entities.

7.2 ConceptC++ Backend

To use ConceptC++ as a backend language, we have to provide the concepts for higher-order constructs. In Fig. 7.8 we list the concepts for unary and binary functions, in Fig. 7.9 those for unary and binary type constructors.

```
concept Function<class Fun> {
  typename Domain1;
  typename Codomain;

  Codomain operator () (Fun, Domain1);
};

concept Function2<class Fun> {
  typename Domain1;
  typename Domain2;
  typename Codomain;

  Codomain operator () (Fun, Domain1, Domain2);
};

...
```

Figure 7.8: Function concepts in C++

```
concept TypeConstructor<class T> {
  template<class X>
  class Apply;

  typename Inner;

  requires std::SameType<Apply<Inner>, T>;
}

concept TypeConstructor2<class T> {
  template<class X1, class X2>
  class Apply;

  typename Inner1;
  typename Inner2;

  requires std::SameType<Apply<Inner1, Inner2>, T>;
}
...
```

Figure 7.9: TypeConstructor concepts in C++

There is one function concept for every arity, as shown in Fig. 7.8, and each function concept provides three types of members:

- associated domain types (called `Domain1`, ...) for every domain,
- an associated codomain type (called `Codomain`),

- and an application operator mapping elements of the domain(s) into the codomain.

The function call operator is written in confix notion; therefore the syntax of a function call is `f(x)` where `f` is a unary function and `x` an argument of its domain type. The three provided entities capture what makes a type a function type. The type `X -> Y` in Haskell or even in mathematics states that objects of these types provide exactly one operation: a mapping of elements of type `X` to elements of type `Y`. The domain and codomain type are explicitly given, but not explicitly named in this notation. The concept makes explicit all entities a function type has to provide. With type constructors, the situation is similar. C++ does not provide support for first-class type constructors, and Fig. 7.9 lists the type constructor concepts with which we implement first-class type constructors. An alternative might be to use template template parameters, which provide the possibility to parameterise templates on other templates. However, due to technical limitations, they are rarely used by C++ programmers [Vandervoorde and Josuttis, 2002]. Accordingly, we represent type constructors (of kind $* \Rightarrow *$) with a `TypeConstructor` concept, as presented in Fig. 7.9. The concept is parametrised by a plain template parameter and provides three associated entities: a template that represents type constructor application, an associated type that represents the type the constructor is applied to, and an associated requirement ensuring the consistency of all the ingredients. In the same style, we provide concepts of type constructors with a higher arity.

With these concepts available, we can translate CTKL into ConceptC++ code. We present the rewriting rules for types in Table 7.2, for function signatures in Table 7.3 and Table 7.4, and for concepts in Table 7.5. Rewriting of a CTKL type according to Table 7.2 is straightforward. Type variables are just printed as strings, for type constructors their image under the user-given type constructor mapping m is printed as string. For complex types, the rewriting adds the type-specific syntax and is recursively called on the components of the complex type. The notation `[|x|]` stands for the ConceptC++-syntax of a CTKL type `x`. Type application is denoted by angle brackets, where in the case of a type variable the `Apply` method is used. Tuple types in C++ are implemented as templates, so they are a special case of type application. Member types can be accessed by the membership resolution operator `::`.

CTKL type `x`	ConceptC++-syntax `[|x|]`
type variable `t`	`t`
type constructor `t`	*m*(`t`)
a type in parentheses `(t)`	`( [|x|] )`
`(t1, t2, ..., tN)`	`TupleN<[|t1|], ..., [|tN|]>`
`t<t1, t2, ..., tN>`	`t::Apply<[|t1|], ..., [|tN|]>`
where `t` is a type variable	
`t<t1, t2, ..., tN>`	`[|t1|] <[|t1|], ..., [|tN|]>`
where `t` is not a type variable	
`t1::t2`	`[|t1|]::[|t2|]`

Table 7.2: Rewriting of a CTKL type in ConceptC++

In the same fashion, Table 7.3 and Table 7.4 show the processing for CTKL function signatures in C++. This processing is divided into two procedures, the first step is a rewriting of the function signature head, followed by the generation of parts of the function signature body. The head of a function signature is rewritten as a C++ class (using the `struct` keyword). Func-

tion signatures with type parameters are written as template classes, the context is rewritten accordingly. It starts with the keyword `requires`, followed by the list of constraints with their type parameters in angle brackets.

Afterwards, the function signature body is generated. That means, within the class for the function signature, all domain types and the codomain type are defined using the `typedef` keyword. Furthermore, we write the signature of the application operator in C++ syntax.

CTKL function signature `f`	ConceptC++ syntax *head_syntax* (`f`)
`TC_f :: A -> B`	`struct TC_f`
`TC_f <a,b> :: a -> b`	`template<class a, class b>` `struct TC_f`

`TC_f <a,b> ::`
`  SomeReq a,`
`  SomeOtherReq b`
`  => a -> b`
	`template<class a, class b>`
	`requires`
	`  SomeReq<a>,`
	`  SomeOtherReq<b>`
	`struct TC_f`

Table 7.3: Rewriting of a CTKL function signature head in ConceptC++

CTKL function signature `f`	ConceptC++ syntax *body_syntax*(`f`)

`TC_f :: A -> B`	`{`
	`  typedef [|A|] Domain1;`
	`  typedef [|B|] Codomain;`
	
	`  Codomain operator()(Domain1 x1) { }`
	`};`

`TC_f :: (A1,A2) -> B`	`{`
	`  typedef [|A1|] Domain1;`
	`  typedef [|A2|] Domain2;`
	`  typedef [|B|] Codomain;`
	
	`  Codomain operator()`
	`  (Domain1 x1, Domain2 x2) { }`
	`};`

Table 7.4: Rewriting of a CTKL function signature body in ConceptC++

Finally, Table 7.5 contains the rules for rewriting a CTKL concept. The `class` expression from CTKL is replaced by the `concept` keyword, the type parameters of the concept are enclosed in angle brackets. Afterwards, the rewritten function signature heads are listed, and every function signature head is terminated with a semicolon.

7.3 Scala Backend

As in the case of ConceptC++, for the Scala backend we first have to provide the concepts (as traits) for higher-order constructs. In Fig. 7.10, we list the traits for unary and binary function

CTKL concept	ConceptC++ syntax

`class Name a b where`
`  f1`
`  f2`
`  ...`
	`concept Name<a, b> {`
	`  head_syntax(f1);`
	`  head_syntax(f2);`
	`  ...`
	`};`

Table 7.5: Rewriting of a CTKL concept in ConceptC++

interfaces, in Fig. 7.11 those for unary and binary type constructors.

```
1 trait FunctionInterface {
2     type Domain1
3     type Codomain
4
5     def apply(x: Domain1) : Codomain
6 }
7
8 trait Function2Interface {
9     type Domain1
10    type Domain2
11    type Codomain
12
13    def apply(x1: Domain1, x2: Domain2) : Codomain
14 }
15
16 ...
```

Figure 7.10: Function concepts in Scala

```
1 trait TC {
2     type A1
3 }
4
5 trait TC2 {
6     type A1
7     type A2
8 }
9
10 ...
```

Figure 7.11: Type constructor concepts in Scala

As in the case of ConceptC++ function concepts, we have different traits for functions of different arities, each providing the usually three types of members. The application operator is provided as an associated method apply, as Scala does not allow one to overload the application operator.

Higher-order type constructors are represented using traits `TC` respectively `TC2`. However, these traits are simpler than their C++ counterparts, as they just have associated types for the

arguments to the type constructor. There is no `apply` method, as the associated types can be assigned using the = notion.

With the auxiliary concepts implemented as traits, we can translate CTKL into Scala. As before, we present the different rewriting rules in tables, those for types in Table 7.6, those for function signatures in Table 7.7 and Table 7.8, and those for concepts in Table 7.9. Rewriting of a CTKL type according to Table 7.6 is, again, straightforward and does not significantly differ from ConceptC++ rewriting. Type variables are just printed as a strings, for type constructors their image under the user-given type constructor mapping m is printed as string. For complex types, the rewriting adds the type-specific syntax and is recursively called on the components of the complex type, where the notion `/x/` stands for the Scala syntax of a CTKL type `x`. Type application is denoted by square brackets, except for type application where the first type is a type variable. Then, type application is denoted by braces with explicit assignment of types to the inner types of the type constructor (which thus has to be declared as a type constructor using the appropriate type constructor trait). Membership resolving is denoted by # in Scala.

CTKL type `x`	Scala syntax `/x/`
type variable `t`	`t`
type constructor `t`	*m*(`t`)
a type in parentheses `(t)`	`( /x/ )`
`(t1, t2, ..., tN)`	`TupleN[ /t1/, ..., /tN/ ]`
`t<t1, t2, ..., tN>` where `t` is a type variable	`t{A1 = /t1/, ..., AN = /tN/}`
`t<t1, t2, ..., tN>` where `t` is not a type variable	`/t1/ [/t1/, ...,/tN/]`
`t1::t2`	`/t1/#/t2/`

Table 7.6: Rewriting of a CTKL type in Scala

Building on this rewriting of types, Table 7.7 and Table 7.8 show the translation of CTKL function signatures into Scala. Again, this step is divided into two procedures, the function signature head rewriting and the (partial) function body generation. The head of a function signature is rewritten as a class; if the function signature has type parameters, they are attached in squared brackets. The rewriting of the context is not as straightforward as in C++. As we have shown above, in Scala there are two ways to constrain type parameters with concepts. To provide a backend that uses generics that are integrated in the object system of the language, we constrain type parameters with concepts by subtyping. Thus, the backend can be used as a sketch for other backends in pure object-oriented languages. In the Scala backend, there is no equivalent to the keyword `requires`—instead, requirements have to be expressed directly in the parameter list using the subtype relation `<:`. However, as constraining type parameters with concepts by subtyping does not allow for multi-parameter concepts, we also use implicits to model multi-parameter concept requirements. For type identities, there is a special syntax in Scala: the requirement `SameType a b` from CTKL is written as `a =:= b`.

Afterwards, the function body is generated. This means that within the class for the function signature all domain types and the codomain type are defined using the `type` keyword and the assignment operator. Furthermore, a syntax skeleton of the `apply` method is generated.

Finally, Table 7.9 contains the rules for rewriting a CTKL concept. The `concept` expression from CTKL is expressed as a trait, the type parameters of the concept are enclosed in squared

CTKL function signature f	Scala syntax *head_syntax* (f)
`TC_f :: A -> B`	`class TC_f extends FunctionInterface`
`TC_f <a,b> :: a -> b`	`class TC_f[a,b] extends FunctionInterface`

`TC_f <a,b> ::`
`SomeReq a,`
`SomeOtherReq b`
`=> a -> b`
	`class TC_f[a <: SomeReq,`
	`b <: SomeOtherReq]`
	`extends FunctionInterface`

`TC_f <a,b> ::`
`SomeReq a,`
`SomeOtherReq a b`
`=> a -> b`
	`class TC_f[a <: SomeReq]`
	`(implicit r1: SomeOtherReq[a,b])`
	`extends FunctionInterface`

`TC_f <a,b> ::`
`SomeReq a,`
`SameType a b`
`=> a -> b`
	`class TC_f[a <: SomeReq]`
	`(implicit r1: a =:= b)`
	`extends FunctionInterface`

Table 7.7: Rewriting of a CTKL function signature head in Scala

CTKL function signature f	Scala syntax *body_syntax*(f)

`TC_f :: A -> B`	`{`
	`type Domain1 = /A/`
	`type Codomain = /B/`
	
	`def apply(x1: Domain1)`
	`: Codomain = { }`
	`}`

`TC_f :: (A1,A2) -> B`	`{`
	`type Domain1 = /A1/`
	`type Domain2 = /A2/`
	`type Codomain = /B/`
	
	`def apply(x1: Domain1, x2: Domain2)`
	`: Codomain = { }`
	`}`

Table 7.8: Rewriting of a CTKL function signature body in Scala

brackets. The body of the concept is generated by listing the function signature heads of the contained methods.

7.4 Generic Haskell Backend

To illustrate that our approach can be used with very different backends, we show that it is also possible to rewrite CTKL in Haskell. For this we need to show how a function type can be replaced by a function concept and how higher-order type constructors are expressed using concepts. Such a rewriting is only possible because of recent additions for associated types to

CTKL concept c	Scala syntax

`class Name a b where`
`  f1`
`  f2`
`  ...`
	`trait Name[a,b] {`
	`  head_syntax(f1);`
	`  head_syntax(f2);`
	`  ...`
	
	`}`

Table 7.9: Rewriting of a CTKL concept in Scala

the GHC Haskell compiler [Chakravarty et al., 2005, Schrijvers et al., 2008]. These associated types are neccesary to express the member types of the auxiliary concepts.

We start with the function concept as a Haskell type class, presented in Fig. 7.12. As Haskell does not allow overloading the application operator, we use an infix operator `(!)` instead. In the signature of the `(!)` operator we still use higher-order functions and currying,

```
1  class Function fun where
2    type Domain fun
3    type Codomain fun
4    (!) :: fun -> (Domain fun -> Codomain fun)
5
6  class Function2 fun where
7    type Domain1 fun
8    type Domain2 fun
9    type Codomain fun
10   (!) :: fun -> ((Domain1 fun, Domain2 fun) -> Codomain fun)
11
12 ...
```

Figure 7.12: Function concepts in Haskell

but this is only to provide a convenient notion of the application operator. The actual function arguments are represented by tuples.

As functions in our framework are represented by a type for each function, this approach is directly applied to the Haskell backend: for every function, a new type is defined. As Haskell does not provide the possibility to define members of types, the newly defined type just provides a single constructor. Domain and codomain types as well as the application operator are associated with the type by stating explicitly that this type is a function type, using an `instance` declaration:

```
1 data IncFunc = IncFuncC
2
3 instance Function (IncFuncC) where
4     type Domain1 = Int
5     type Codomain = Int
6     (fun ! x) = (x + 1)
```

The increment function in this example is defined as a type `IncFunc`, and this type is declared as a model of the function concept. In this declaration, domain and codomain types are defined and the application operator is implemented accordingly.

Next, we move to the `TypeConstructor` concept where we use GHC's support for infix type constructors to make their definitions a bit more readable (Fig. 7.13). As in the C++ case,

```
1 class TypeConstructor con where
2   data (:@) con :: * -> *
3   type Apply = (:@)
4
5 class TypeConstructor2 con2 where
6   data (:@) con2 :: * -> * -> *
7   type Apply2 = (:@)
8
9 ...
```

Figure 7.13: Type constructor concepts in Haskell

this concept introduces an `Apply` method at the level of types. Type constructor application produces a wrapped value, the associated type `Apply` corresponds to the `Apply` template in the C++ version of this concept.

Due to the fact that this Haskell backend is more a proof of concept that our approach also works with a non-object-oriented backend language than a real backend, we do not provide specific rewriting rules here.

7.5 Conclusion

In this chapter we discussed how CTKL can be mapped to different existing programming languages. We started with a discussion of general generic programming features that are necessary for CTKL backend languages, and showed how these features map to different languages. Surprisingly, there are just a few languages that provide sufficient support for generic programming with concepts. Concepts themselves are only recently integrated in a few programming languages, and even when supported by a language, this support often is rudimentary. However, there are at least three languages that provide sufficient support for generic programming with concepts: ConceptC++, Scala, and Haskell. For these three languages we have shown how to provide higher-order constructs as concepts and how to map CTKL expressions to particular language syntax. As the three languages are of very different nature, our three concrete backends prove that CTKL is a kernel language that can be translated into languages from different programming paradigms. Of course, more than these three backends are possible, although writing and using them might be more verbose than for ConceptC++, Scala, and Haskell.

Chapter 8

Implementation

In the previous chapters, we described the kernel languages, the transformation that is defined between them, and different backends for the target language of the transformation. In this chapter, we describe how to turn these three building blocks into a software system that implements an FTKL compiler with different backends.

We start by giving an overview of the architecture of the system (Sect. 8.1) and then describe its main part: the compiler (Sect. 8.2) that translates FTKL function signatures into CTKL and then generates from CTKL target language code. This compiler is available from the project web page [Lincke]. Afterwards, we show that this compiler can be used to produce highly efficient ConceptC++ code (Sect. 8.3). As mentioned in Chap. 1, the method presented in this thesis is motivated by the use of functional prototypes in scientific modeling. In such models, performance is an important factor, as their complexity leads to long computation times. Efficiency, especially at the level of elementary operations, can decrease these computation times considerably. To show that the compiler output can be the basis of efficient code, we use the compiler to implement a library for functional operations such as function composition, currying, and uncurrying. We implement test programs that are based on higher-order functions and compare the run-time efficiency of these programs with that of programs implemented by using libraries that provide higher-order function workarounds. Finally we present a test suite that is used to test signatures generated by the compiler. This test suite is generated manually and captures all essential elements of FTKL.

8.1 Architecture

The transformation system is embedded in a workflow for developing programs or libraries from functional prototypes using our approach. This workflow, depicted in Fig. 8.1, is as follows: the developer in the application domain writes a functional prototype of the library or the model that should be implemented. This prototype is used for testing and iterative improvement of the library or model. If this prototype has been finalised, parts of it can be transformed into FTKL specifications. In particular, the function signatures and concept definitions are extracted and then transformed into FTKL. This step is not explicitly integrated in our transformation system, mainly because we aim at Haskell as functional prototyping language. As FTKL is basically a subset of Haskell, the translation from Haskell function signatures into FTKL function signatures is straightforward and can be done with a text-replacement script. For other func-

tional languages, one would have to provide an explicit transformation from function signatures in that language to FTKL.

In the next step, the extracted FTKL specifications are the input for our compiler, which is described in Sect. 8.2. The compiler performs two transformations: first, from FTKL to CTKL

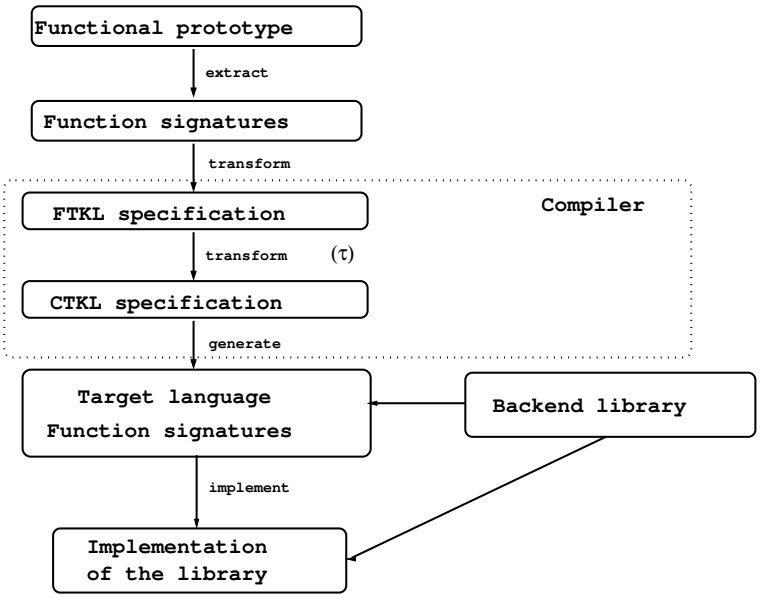

Figure 8.1: The architecture of our transformation system

according to the transformation that is described in Chap. 6. Secondly, CTKL is transformed into function type declarations in the target language, according to the backend transformation described in Chap. 7.

The generated target language contains function signatures expressed as function type definition in the implementation language. These function type definitions are constrained with requirements provided by the backend library and their functionality (application operators) has to be implemented manually. Application operators are not included in FTKL specifications, and are therefore not in the scope of automated transformation. The same is true for datatypes, which also have to be implemented manually.

As the implemented program or library depends on the backend library that provides the concept definitions for functions and type constructors, the backend library has to be included in the implementation, that is, compilation without the backend library is not possible.

8.2 Compiler

The compiler that implements the transformation from FTKL to CTKL and from CTKL to the target language is depicted in Fig. 8.2. The compiler consists of several phases, working in a pipe-and-filter manner. As the compiler only accepts FTKL as input, the first stage of compilation is a parser for FTKL. The FTKL representation has to be given in a separate file, the filename is an input argument for the compiler. As FTKL is a subset of Haskell, the parser in the compiler system is based on a Haskell parser provided in the package “Haskell-Source with

Extensions (haskell-src-exts)" by Broberg, and implemented in Haskell itself. It parses FTKL and provides an abstract syntax tree (AST) representation. The AST representation is passed on and further processed by the transformer, which is implemented in C++. The transformer implements the transformation $\mathscr{T}$, or more precisely, the five steps we described in Chap. 6 according to the algorithms given there. It produces an abstract syntax tree of CTKL function signatures.

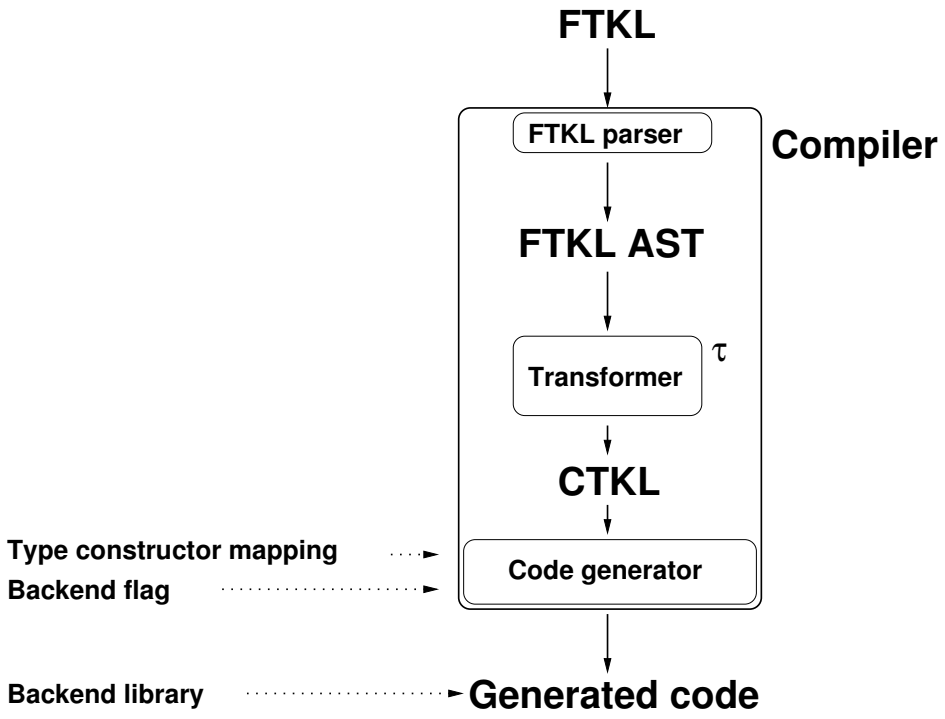

Figure 8.2: The compiler and its stages

These CTKL function signatures are finally transformed into actual programming language code. This transformation is integrated in the compiler and gets two additional inputs: a flag that chooses the target language and a user-provided type constructor mapping. As mentioned in Chap. 7, such a mapping contains a list of type constructors and their replacements. If a type constructor is not found in this list, it is replaced by a type constructor with an identical name. The backend transformation is, in principle, organised as a rewriting system, that is, expressions from CTKL are successively replaced by respective language constructs in the target language. Finally, the generated code is written into an output file. The filename is given as an argument to the compiler.

The whole compiler is implemented in C++ using the Standard Template Library (STL) for generic container types and comprises about 5500 lines of code.

8.3 Performance

As already mentioned in the introduction, there are alternatives to our approach. We simulate higher-order functions and higher-order type constructors by the use of concepts, but

such higher-order constructs could also be simulated with other language constructs. We already explained why we prefer our approach, but with the implemented transformation system we can show that our approach can also produce code that is highly performant in terms of computing time. As test, we use programs that make use of higher-order functions and basic functional operations like function composition and currying. For these programs we compare the concept-based implementations with alternative representations of functions. Especially in C++, there are many ways to provide higher-order functions, making use of different programming techniques that represent the different programming paradigms C++ supports. We evaluate the runtime efficiency of several possible representations of a function type:

- as a function-pointer wrapper;
- as an abstract class;
- as `Boost.function`;
- as a FC++ function; or
- as a function concept.

All approaches are illustrated in Fig. 8.3. The implementations of the incrementation functions `increment` (as a usual C++-function) and `Increment` (function-object) used in this figure are already presented in the introduction of this thesis, the function concept is the one presented in Chap. 7. The function pointer approach builds on the usual imperative way to define a pointer that points to the address of executable code and not to data. Function pointers are the only technique for using functions as arguments to other functions that is directly supported in the language. We define a type that wraps a function pointer as shown in Fig. 8.3; this figure also shows how this type can be used (Lines 16–17).

Representing the function type as an abstract base class for which concrete functions can be derived is the standard design in object-oriented programming. Fig. 8.3 shows the abstract base class and its function call operator, which is declared `virtual` ("abstract") so that all subclasses are forced to provide an implementation. A concrete function class has then to be derived from that class, as shown in Fig. 8.3 (Lines 20–34).

The `Boost.function` library is a higher-order function library that is part of the Boost [Boo] collection of de-facto standardised libraries; it combines function pointers, object-oriented programming, and C++ metaprogramming techniques [Abrahams and Gurtovoy, 2005]. The library `Boost.function` contains a family of class templates that are function object wrappers. The usage of such a wrapper type is shown in Fig. 8.3 (Lines 38–40).

FC++ [McNamara and Smaragdakis, 2004] is another, historically earlier, library for functional programming in C++, and it is still in use today. The function datatype from FC++ works similar to the one from `Boost.function`, with the difference that it requires a function pointer for construction (Lines 44–45).

The concept-based design is the approach we followed in this thesis: functions are defined as function classes, which are types themselves, and higher-order functions are realised as type parameters constrained with a function concept. As we have showed in Chap. 6 and in Chap. 7, our transformation and the C++ backend can be used to implement libraries based on this approach.

For each of the five possibilities to represent a function type, we implemented a little library of operations that are widely used in programming with functions:

```
// function-pointer wrapper
template<class A, class B>
class FP_Function {
  public:
    typedef B (*pt2Func)(A);

    FP_Function(pt2Func f) {my_f=f;}
    B operator () (A x) {
        return my_f(x);
    }

  private:
    pt2Func my_f;
};

FP_Function<float,float> fp_inc(increment);
float f = fp_inc(1.1);

// oo-solution
template <class A, class B>
class OO_Function {
  public:
    virtual B operator()(A)=0;
};

class Increment_Function : public OO_Function<float, float> {
  public:
    float operator() (float x) {
        return increment(x);
    }
};

Increment_Function oo_inc;
float f = oo_inc(1.1);

// Boost.function library
#include <boost/function.hpp>
boost::function<float (float x)>  boost_inc1;
boost_inc1 = Increment();
float f = boost_inc1(1.1);

// FC++ library
#include <function.h>
fcpp::Fun1<float,float> fcpp_inc = makeFun1( ptr_to_fun (&increment));
float f = fcpp_inc(1.1);

// generic programming with concepts
concept_map Function<Increment>  {
     typedef float Domain;
     typedef float Codomain;
}

Increment inc;
float f = inc(1.1);
```

Figure 8.3: Function concept and function datatypes in C++

- Function application: the application of a function is the main operation for functions. Ideally, a higher-order function library should not introduce any performance penalty for the application operator.
- Higher-order functions: the application of a higher-order function should not cause any performance penalty compared to ordinary functions.
- Function composition: functions should be composable with each other. Given two functions `f,g` with signatures `f: A -> B` and `g: B -> C`, function composition returns a function `g.f: A -> C`. Function composition should be an efficient operation, it should not introduce any performance penalty for application of a composed function.
- Partial application: for functions with tuple types as domain, it should be possible to transform them into a function that can be partially applied to fewer arguments. This transformation and the partial application should be efficient.

To test the performance of different function type implementations, we implemented test procedures for all four operations listed above. All tests were performed using the following setting: Dell Latitude notebook with an Intel Pentium III mobile, 1.86 GHz, OpenSuSe Linux 10.2, kernel version 2.6.18.8-0.13. We used the conceptgcc compiler [Gregor, a], version 4.3.0 to compile the code, and all code was compiled with two different optimisation level: no optimisation at all, and the highest optimisation level (enabled by the compiler flag `-O3`).

For the function application test, we implemented the following test procedure: We used a wrapper class (`MyClass`) to wrap the built-in type float. The purpose of using such a wrapper is to complicate compiler optimizations. Then, we defined a test function with signature `MyClass -> MyClass`, which implements an incrementation on the underlying floating point data. The different function types were used to define five test functions, whose application operators was then successively called in a loop. The execution time of this loop was measured using the internal C++ clock, and Table 8.1 shows the results for this test. In every iteration step, the function application operator was called 10^7 times, the results were averaged over 20 iterations of the test procedure. Furthermore, the results were scaled to the fastest implementation, which was the compiler-optimized concept-based implementation. In the table, "FP" stands for function-pointer wrapper; the other abbreviations are obvious. As the table shows, the concept-

	no optimisation	optimisation level 3 (-O3)
FP	5.37	2.56
OO	3.84	1.06
Boost	8.91	8.59
FC++	12.42	5.24
Concept	3.73	1.00

Table 8.1: Performance of function application, scaled to the fastest implementation (concepts) with -03

based and the object-oriented function implementation perform nearly equally. In those two approaches, the compiler can inline function bodies (and does, if we set an appropriate optimization level). In contrast, function pointers always suffer from the costs of indirection, which

explains why the function pointer wrapper approach is much slower. The function datatypes provided by the functional extension libraries FC++ and `Boost` are even slower, which is due to the fact that these datatypes invoke a big machinery, which involves virtual function calls (in the case of FC++) and pointer indirections.

The performance of higher-order functions, implemented by the different approaches, can be tested in a way similar to the test of function application. We can use the same functions and implement a simple higher-order function test case: a binary function that takes as arguments

	no optimisation	optimisation level 3 (-O3)
FP	17.18	5.45
OO	13.82	6.18
Boost	25.19	16.76
FC++	34.50	10.55
Concept	12.92	1.00

Table 8.2: Performance of higher-order functions

a function and a value to which this function should be applied. As before, the application of the functions was embedded in a loop and the execution time for this loop was measured. The results were averaged over 20 iterations of this procedure. Table 8.2 shows the result for this test, where the number of iterations in the loop was chosen so that the total number of function applications was equal to the test for function application. Therefore, we can normalise the results to the fastest implementation of the previous function application test. Impressively, for the compiler-optimized concept-based approach there is no overhead at all, which means that the compiler inlines all function calls so that there is no difference between function application and higher-order function application. The object-oriented approach now has some overhead, caused by the fact that a higher-order function in this setting has to be declared as a reference (or pointer) to the abstract base class. The virtual function application operator has to be resolved at run time. Again, the two library datatypes are much slower than the other approaches, in the highest optimization level more than one decimal order of magnitude.

We also tested the performance of function composition and currying with partial applications, the results are listed in Table 8.3 and Table 8.4. Both tests are similar to the test of Table 8.2, as in both cases we iterate the invocation of composed respectively curried functions. All composition and currying operations for the function representations had to be

	no optimisation	optimisation level 3 (-O3)
FP	-	-
OO	10.77	8.14
Boost	26.97	20.03
FC++	39.14	13.00
Concept	10.51	1.00

Table 8.3: Performance of the application of composed functions, scaled to the fastest implementation (concept-based approach)

	no optimisation	optimisation level 3 (-O3)
FP	-	-
OO	59.41	43.05
Boost	284.25	123.74
FC++	88.63	72.86
Concept	11.60	1.00

Table 8.4: Performance of the application of partially applied, curried functions, scaled to the fastest implementation

implemented by ourselves, none of them was provided, not even for the FC++ and `Boost` function types. For a detailed description of these implementations we refer to Lincke and Schupp [2009].

In all experiments, the concept-based approach produces the most efficient code. Especially in the last two experiments, there are huge factors in the execution times. They are caused by the fact that, except for the concept-based implementation, all implementations have to resolve the application operator of function values at runtime. Function pointers always suffer from the costs of indirection, which explains its performance penalty. The object-oriented approach comes with a runtime overhead caused be the resolution of virtual function calls; and the function datatypes provided by the functional extension libraries FC++ and Boost are even slower, which, as already said, is due to the fact that these datatypes invoke a big machinery involving virtual function calls (in the case of FC++) and pointer indirections. In contrast, using a separate class for every function, together with a function concept, allows the compiler to resolve all function call operators at compile time. If optimisation is enabled, all function call operators can be inlined, increasing the runtime efficiency drastically. This is one major advantage of our implementation of higher-order functions: an object-oriented approach, for instance, does not allow for inlining at all, as virtual function calls can not be inlined.

The experiments show that with our approach we can generate highly efficient C++ code. This statement is C++ specific, as in C++ every generic function or class is instantiated at compile time, and the instantiated code is compiled. Thus, overloaded methods are replaced with their actual implementations at compile time, and there is no runtime overhead for resolving overloading. Other languages handle generic programming based on concepts differently. When concepts are integrated in the type hierarchy of a language, as in Java or Scala, overloading is usually resolved at runtime. Such languages produce polymorphic executable code and type parameters are replaced at runtime. Thus, method overloading is resolved at runtime leading to an overhead. In such languages, we cannot expect similar performance gains from the generated code. On the other hand, Java or Scala today are typically not used for performance-critical code anyway.

8.4 Test systems

Software testing is an important phase of software development, and it is in particular important for safety-critical programs such as compilers. The user of a compiler expects that the compiler implementation works correctly, that is, it produces target code that is executable and has the

```
-- simple mapping between two types
f  :: Int -> Int

-- generic function
f  :: a -> a

-- generic and higher order function, curried
f  :: a -> a -> a

-- generic function, uncurried
f  :: (a,a) -> a

-- with context
f  :: Ord a => a -> a -> a

-- type identities
f  :: (a -> b) -> b

-- higher kinded type variable
g  :: x -> f x

-- higher kinded type variables, type constructor concepts
inpsys  :: (Monad m, Functor m) => ((x,a) -> m x) -> List a  -> x -> m x

-- weired structure
f  :: ((a -> c,b) -> c, b, a) -> (c,(a -> b))

-- something cool
f  :: ((b -> (b,c)), ((a,b) -> b)) -> a -> c

-- fold from Origami, including a higher-order type constructor
fold    :: Bifunctor s => (s a b -> b) -> Fix s a -> b

-- explicitly kinded fold
fold    :: Bifunctor s => ((s :: * -> * -> *) (a :: *) (b :: *) -> b)
                       -> (Fix :: (* -> * -> *) -> * -> *) s a -> b

-- extensive type constructor test
f  :: (H (F I (J a b c))) c -> K (G I)

-- higher-order type constructor variable
f  :: f g -> g a
```

Figure 8.4: The test cases for the compiler

same semantics as the source code. Whether a compiler works correctly can usually not be proven, instead test suites are used in a testing process. Such test cases are often generated by tools [Burgess, 1994, Boujarwah and Saleh, 1997]. The test cases are compiled, and afterwards the results are compared with the expected results that is specified in the test suite.

We tested our compiler with the test suite presented in Fig. 8.4. It was not generated automatically, but written by hand instead. It captures all essential elements of FTKL: generic functions, higher-order functions in both domain and codomain, higher-order type constructors, and higher-kinded type variables. We used this test cases to test the compiler and its different backends, and computed manually the expected results. For the compiler with ConceptC++ backend the generated signatures are listed in Appendix B.

Chapter 9

Applications

In the previous chapters, we introduced the two kernel languages, the transformation, and different backends, and described how the whole system is implemented. In this chapter, we turn to the application side and present three applications of our transformational approach. All three applications use Haskell as functional prototyping language and ConceptC++ as implementation language, however, the nature of the examples and the usage of the transformation is quite different. In Table 9.1 we summarise the code statistics of all three libraries. The table contains the number of classes and concepts, the number of generated lines of code, and the number of lines of code the library contains in total. We will analyse these numbers in the sections that refer to the different libraries.

In our first example (Sect. 9.1), we use the transformation to implement the library FCPPC (Functional C++ with and without Concepts). It is a foundational library and extends the ConceptC++ backend, as it provides basic functional operations, such as composition and currying. These operations are implemented for function objects, and their signatures are specified using the function concept. Further, this library extends the concepts contained in the backend with a number of useful concepts for special type constructors, such as Monad, Functor, and Bifunctor. The functional prototype for this library is the Haskell standard library itself, and the function signatures that are transformed are taken directly from there.

In the second example (Sect. 9.2), the transformation is used to reengineer a generic Haskell library in ConceptC++. The so called *Origami* library centers around a generic fixed-point data type and generic recursion operators for it. The library defines a fixed-point type and recursion operators, and we apply the transformation to the function signatures of these recursion

Library	Description	Classes	Gen.	Total	Ratio
FCPPC	Higher-order functions	$\simeq 60$	575	4200	0.15
Origami++	Fixed points and recursion	$\simeq 40$	290	1250	0.23
MonSys++	Monadic dynamic systems	$\simeq 25$	580	1485	0.39

Table 9.1: Functional C++-libraries: number of classes (and concepts), generated code, and total size (LOC). The ratio between these two is computed as the ratio between the lines of code generated and the lines of code of the whole library (including parts that are outside the transformation, like data-type definitions).

operators to generate the function signatures of the *Origami++* library.

In our third example (Sect. 9.3), we use Haskell as an intermediate language for vulnerability modelling. We used a functional prototype of a vulnerability modelling library developed in a related thesis [Ionescu, 2008], extracted the function signatures from this prototype and used the transformation to generate function signatures for a library for vulnerability modelling with monadic dynamical systems. In contrast to the other applications, in this case the code generated by the transformation is further optimised. Certain combination of concepts are summarised to new concepts to increase clarity and understandability of concept-constrained functions.

The C++ code of our examples is tested with the gcc compiler [GCC], version 4.3, and the concept code is partly tested with the ConceptGCC compiler [Gregor, a] (version alpha 7). The presented libraries are available from the project web page [Lincke].

```
1  Compose  :: ((b -> c),(a -> b)) -> a -> c
2  Compose2 :: ((b -> c),((a1,a2) -> b)) -> (a1,a2) -> c
3  Compose3 :: ((b -> c),((a1,a2,a3) -> b)) -> (a1,a2,a3) -> c
4  Compose4 :: ((b -> c),((a1,a2,a3,a4) -> b)) -> (a1,a2,a3,a4) -> c
5  Compose5 :: ((b -> c),((a1,a2,a3,a4,a5) -> b)) -> (a1,a2,a3,a4,a5) -> c
6
7  Curry    :: ((a1, a2) -> b) -> a1 -> a2 -> b
8  Curry2   :: ((a1, a2, a3) -> b) -> a1 -> (a2,a3) -> b
9  Curry3   :: ((a1, a2, a3, a4) -> b) -> a1 -> (a2,a3,a4) -> b
10 Curry4   :: ((a1, a2, a3, a4, a5) -> b) -> a1 -> (a2,a3,a4,a5) -> b
11 Curry5   :: ((a1, a2, a3, a4, a5, a6) -> b) -> a1 -> (a2,a3,a4,a5,a6) -> b
12
13 Uncurry :: (a1 -> a2 -> b) -> (a1, a2) -> b
14 Uncurry2 :: (a1 -> a2 -> a3 -> b) -> (a1, a2, a3) -> b
15 Uncurry3 :: (a1 -> a2 -> a3 -> a4 -> b) -> (a1, a2, a3, a4) -> b
16 Uncurry4 :: (a1 -> a2 -> a3 -> a4 -> a5 -> b) -> (a1, a2, a3, a4, a5) -> b
17 Uncurry5 :: (a1 -> a2 -> a3 -> a4 -> a5 -> a6 -> b) -> (a1, a2, a3, a4, a5, a6) -> b
18
19 class Functor f where
20   fmap   ::  ((a -> b), f a) -> f b
21
22 class Bifunctor f where
23   bimap     :: ((a -> c), (b -> d), f a b) -> f c d
24
25 class Monad m where
26   bind   ::  (m a, (a -> m b)) -> m b
27   return ::  a -> m a
```

Figure 9.1: The Haskell function signatures we transform for the FCPPC library

9.1 Higher-order Functions

In the first application we use the transformation to provide some additional functionality for the C++ backend library, called FCPPC (*Functional C++ with and without Concepts*). This library contains, besides the concepts for functions and type constructors presented in Chap. 7, concepts for functors, bifunctors, and monads. Further, the library provides some operations

for functional programming based on function objects and the function concept, in particular function composition, currying and uncurrying.

The Haskell function signatures that describe the functions and additional concepts of the library are listed in Fig. 9.1. They are extracted from the Haskell standard library, and for convenience they are used in uncurried form. Afterwards, we used the transformation to generate ConceptC++ versions of these function signatures. The corresponding application operators are implemented manually, and the resulting ConceptC++ function composition operation is shown in Fig. 9.2. It contains the function objects `Compose` and `Compose_codomain1`; their signatures

```
1  template<class fun1, class fun2>
2  requires
3    Function<fun1>, Function<fun2>,
4    SameType<fun1::Codomain, fun2::Domain1>
5  struct Compose_codomain1 {
6
7      typedef   typename fun1::Domain1  Domain1;
8      typedef   typename fun2::Codomain  Codomain;
9
10     // fun1 :: A -> B,  fun2 :: B-> C
11     Compose_codomain1(fun1 const& f, fun2 const& g) : my_f(f), my_g(g) {}
12
13     inline
14     Codomain operator() (Domain1 const& x) const {
15         return my_g(my_f(x));
16     }
17
18   private:
19     fun1 my_f;
20     fun2 my_g;
21 };
22
23
24 template<class fun1, class fun2>
25 requires
26   Function<fun1>, Function<fun2>,
27   SameType<fun1::Codomain, fun2::Domain1>
28 struct Compose {
29
30     typedef   fun1  Domain1;
31     typedef   fun2  Domain2;
32     typedef   Compose_codomain1<fun1,fun2>Codomain;
33
34     Codomain operator() (Domain1 f, Domain2 g) {
35         return Compose_codomain1<fun1,fun2> (f, g);
36     }
37
38 };
```

Figure 9.2: The C++ implementation of function composition

are encoded in the generated contexts. The function object `Compose_codomain1` encapsulates the composite function, its application operator implements the successive application of the two functions that are composed. `Compose` implements the actual composition operation, it is

applied to two functions and returns the composition as a function object.

The whole FCPPC library contains about 4.200 lines of code, and about 575 of them are generated. This ratio seems to be quite low, however, there are parts of the library for which nothing can be generated at all. The library code can roughly be divided in six groups: the concepts for functions and type constructors we have seen already (Chap. 7), the concepts for functor and monad we have generated, classes that implement parameterised tuple types, the classes for composition, currying, and uncurrying, a number of predefined functions and their modellings of the function concept, and, finally, a number of instance declarations for functors, monads and bifunctors. While the function signatures for composition etc. as well as the member-function signatures for the concepts can be generated, all instance declarations have to be completely implemented manually. That is why, as Table 9.1 shows, almost all the code in this library is handwritten. Taking into account only the parts of the library for which code is generated, the ratio is 575 to 1475 lines of code, which shows the efficiency gained by using the transformation in developing this library.

9.2 Recursive Data Types as Fixed Points

The next application, the Origami++ library, illustrates the use of the transformation, using the function signatures of an existing datatype-generic library in Haskell. The library builds on recursive patterns as folds and unfolds, and a generic fix-point view which allows giving a single generic definition of each recursive pattern used for all datatypes that can be expressed using this fix-point view. This approach is known as *Origami Programming* [Gibbons and O. de Moor, 2003] and because of its theoretical foundation it is of special interest in generic programming. The library defines a fixed-point type with the constructor `In` and the canonical isomorphism `out` so that exactly the same identities hold as, for example, established in Haskell:

```
1 FIX f a = In(f a (FIX f a))
2 out :: FIX f a -> f a (FIX f a)
3 out(In x) = x
```

`In` and `out` define an isomorphism, thus every instance of `FIX f a` defines a fixpoint of the binary type constructor `f`. If `f` is a bifunctor, instead of just a binary type constructor, fundamental results for fixpoints hold that allow for a generic definition of canonical `fold` and `unfold` operations [Lambek, 1968]. The C++ version of the generic fixpoint type looks rather similar:

```
1 template<class F, class A>
2 requires
3   TypeConstructor2<F>
4 struct FixP {
5     typedef F::Apply<A, FixP<F,A> >  F_A_FixP_type;
6     F_A_FixP_type In_;
7
8     FixP(F_A_FixP_type const& In) : In_(In) {}
9 };
```

The type provides a wrapper constructor similar to the one of the Haskell version, together with the canonical isomorphism `FixP_Out`.

```
map    :: Bifunctor s => (a -> b) ->  (Fix s a ->  Fix s b)
fold   :: Bifunctor s => ((s a b -> b), Fix s a) -> b
unfold :: Bifunctor s => ((b -> s a b), b) -> Fix s a
hylo   :: Bifunctor s => ((b -> s a b), (s a c -> c), b) -> c
```

Figure 9.3: Function signatures of the recursion operators of the Origami library

```
binConcat     :: FList a (List a) -> List a
Concat        :: (List a, List a) -> List a
ListLength_F  :: FList a int -> int
ListLength    :: List a -> int
ListAccu_F    :: FList a a -> a
ListAccu      :: List a -> a
ListReverse_F :: FList a (List a) -> List a
ListReverse   :: List a -> List a
ListFilter_F  :: (a -> bool) -> FList a (List a) -> List a
ListFilter    :: (a -> bool) -> (List a) -> (List a)
```

Figure 9.4: Function signatures of the list algebra implemented in Origami++

Fig. 9.3 lists the function signatures of the four operators of the Origami library. They explicitly rely on higher-order functions and type constructors. The fixpoint datatype is parameterised over a functor `s`, called the shape functor of a recursive datatype. All four function signatures are rewritten in uncurried form, and all four contain a function as first argument, `hylo` even contains a second function as argument. The library also provides data-type definitions for lists and trees in terms of this fixed-point type along with the typical constructors for those types, i.e., cons and empty respectively leaf and node. In addition, the library provides elementary functions for these types. The algebra for lists that is provided is summarised in Fig. 9.4.

The transformation generates ConceptC++ function object signatures out of these Haskell function signatures, while the data type declarations are implemented manually. The generated ConceptC++ signatures are listed in Appendix C. As the four Origami functions map, fold, unfold, and hylo depend on functions as well as on the bifunctor concept, Origami++ builds on top of the FCPPC-library presented above.

9.3 A Domain Specific Language for Vulnerability Modelling

The last library, Monsys, is a library for vulnerability assessment. This library differs from the previous ones because it is a domain-specific library; as expected, the ratio of interfaces to implementation changes here in favour of interfaces. In fact, the ratio of generated code is as high as 39%. On the other hand, in this library, the output of the transformation is used only as a preliminary stage of type specification and further transformed by hand, as we will see below, mostly to improve readability.

As this library was a major motivation for the entire thesis, we will provide a detailed description of the formalisation, the functional prototype, and the C++ implementation of the library.

9.3.1 Vulnerability formalisation and functional prototype

In the past decade, the concept of "vulnerability" has played an important role in fields such as climate change, food security, and natural hazard studies. Vulnerability studies have been useful in alerting policymakers to the possibility of precarious situations in the future.

However, definitions of vulnerability vary: there seem to be almost as many definitions as case studies conducted. Wolf et al. [2008] , for instance, analyse the usage of 20 definitions of vulnerability. An allegory frequently used to describe the terminology is the Babylonian confusion, and the need for a common understanding has repeatedly been expressed in the literature [Brooks, 2003, Janssen and Ostrom, 2006].

While definitions of vulnerability are usually imprecise, Ionescu [2008] proposes a mathematically defined framework that underlies the definitions and the usage of the term *vulnerability*. The framework of Ionescu [2008] applies to those studies that "project future conditions" by using computational tools and that can thus be designated as computational vulnerability assessments. This common framework for vulnerability assessment consists of three primitive concepts: an *entity* (which is subject to possible future harm), *possible future evolutions* (of the entity), and *harm*. Next, we briefly summarise the development of Ionescu [2008]. First, we discuss the primitive concepts and, then, their extension to *monadic dynamical systems*.

An entity, its current situation, and its environment are described by a state. In vulnerability computations, the current state of an entity is only a starting point; it is the *evolution* of the state over time that is the main focus of interest. A single evolution path of an entity is called a *trajectory*. In a first, simple vulnerability model, the evolution of an entity is computed in a single step:

```
1 possible :: State -> F Evolution
```

The `possible` function maps a state into an `F`-structure of future evolutions. Two types and one type constructor are involved: `State` is the type of the state of an entity, `Evolution` is the type of the evolution of an entity, for simplicity we take `Evolution = [State]`, and `F` is the type constructor representing the structure of evolutions (e.g., a set). The constructor `F` is a functor that describes the nature of the future computation. For example, if the future of an entity is deterministic, one would have `F = Id`, the identity functor, giving a single evolution. If a more sophisticated model is used to predict the future evolution, the outcome might be a list of future evolutions or a probability distribution over them. In these cases, the list functor `F = []` or a probability distribution functor `F = SimpleProb` (see [Lincke et al., 2009]) would be used. The second ingredient of a vulnerability computation is a harm judgement function:

```
1 harm :: Evolution -> Harm
```

The `harm` function maps evolutions into values of type `Harm`, which describes the nature of the harm assessment. A very simply harm assessment is a threshold measurement, in such cases, harm is simply a boolean value (`Harm = Bool`). In other models, the harm value for each trajectory may be measured as a real value (`Harm` = $\mathbb{R}$). A more sophisticated model might measure multiple dimensions of harm, all expressed as real values, which would lead to `Harm` = $\mathbb{R}^N$.

The `harm` function measures the harm of a single evolution, but a model produces an `F`-structure of evolutions, given some implementation of the function `possible`. The harm function must then be mapped to the evolutions, using the `fmap` function of the functor `F`. A measure

function takes the resulting F-structure of harm values and summarises them into an overall measurement:

```
measure :: F Harm -> V
```

The codomain of the measure function could be, for example, the real numbers (to get the result as a single number, a way of expression policymakers prefer) or a multidimensional space (to express various dimensions of vulnerability instead of collapsing it into a single number). Common examples of the measure function are measure = maximum in the case of F = [] and Harm = Bool, or measure = expectation in the case of F = SimpleProb and Harm = $\mathbb{R}$.

Combining these three functions gives the vulnerability computation:

```
vulnerability :: State -> V
vulnerability = measure . fmap harm . possible
```

This computation captures the idea behind many vulnerability definitions: we *measure* the *harm* that might be *possible*.

The simple model of vulnerability presented above has two main disadvantages. First, it is not very generic: we have to write a new function for every vulnerability computation. This can easily be fixed by using higher-order functions to make possible, harm, and measure inputs to the vulnerability computation. Second, the model is inflexible: the evolution function of the simple model only consumes the initial state of a system as the input. In practice, however, an evolution function often has more inputs. In particular, an evolution may depend on auxiliary control arguments that are not part of the initial state of a system.

Ionescu [2008] explores possible structures that might provide the required flexibility. On the one hand the multitude of systems that might be involved has to be represented, on the other hand we want an easy way of computing iterations. It turns out that the notion of a *monadic dynamical system* (MDS) captures both aspects.

An MDS is a mapping with the signature t -> (x -> m x), where m is a monad. An MDS takes a value of type t (e.g., t = Nat) to a *(monadic) transition function* of type x -> m x. Every MDS must fulfil some further conditions; for instance, the type t has to form a monoid structure (with a zero z and an operation (+)) and the MDS must preserve that structure (must be a monoid morphism—see [Ionescu, 2008] for details).

An MDS need not be directly defined by the user. Often it is built by constructor functions, given a transition function. There are two important constructors for MDSs, one for discrete MDSs taking as input a transition function of the form x -> m x, and another one that constructs an MDS with external control out of a given transition function i -> x -> m x. Their signatures are (see [Ionescu, 2008] for their implementations):

```
discrete_sys :: (Monad m) => (x -> m x) -> (Nat -> x -> m x)
input_system :: (Monad m) => (i -> x -> m x) -> ([i] -> x -> m x)
```

Furthermore, given an MDS, two iteration functions can be defined: the macro trajectory function, macro_trj, computes the evolution of an M-structure of states, and the micro trajectory, micro_trj, computes the m-structure of evolutions (sequences of states). However, it turns out that the micro_trj computation is more useful for vulnerability assessment, as the micro trajectory records the single evolution paths including all intermediate states. The function signature of the micro_trj computation is:

```
micro_trj :: (Monad m) => (i -> (x -> m x)) -> [i] -> x -> m [x]
```

```
-- system Constructors
discrete_sys :: (Monad m) => (x -> m x) -> (Int -> x -> m x)
input_system :: (Monad m) => (i -> x -> m x) -> [i] -> x -> m x

-- trajectory computation
addHist :: (Monad f) => (x -> f x) -> [x] -> f [x]
micro_trj :: (Monad m) =>  (i -> (x -> m x)) -> [i] -> x -> m [x]

-- system combination
pr :: Functor f => (f a, b) -> f (a, b)
pl :: Functor f => (b, f a) -> f (b, a)

syspar :: (Functor m, Functor n) =>
          (t1 -> x -> m x, t2 -> y -> n y) -> (t1, t2) -> ((x,y) -> PairM m n (x,y))

sysser :: (Monad m) => (t1 -> x -> m x, t2 -> x -> m x) ->
                       [(t1, t2)] -> x -> m x
detserl :: (Monad m) => (t1 -> x -> Id x, t2 -> x -> m x) ->
                        [(t1, t2)] -> x -> m x
detserr :: (Monad m) => (t1 -> x -> m x, t2 -> x -> Id x) ->
                        [(t1, t2)] -> x -> m x
embedMDS :: (Monad m) => (t -> a1 -> Id a) -> t -> a1 -> m a

-- vulnerability computation
vulnerability :: (Functor m, Monad m) =>
                 (i -> x -> m x) -> ([x] -> h) -> (m h -> v) -> [i] -> x -> v
```

Figure 9.5: The function signatures of the vulnerability model prototype

Its implementation is shown in Fig. 1.1 in Chap. 1, for further details see [Ionescu, 2008].

To tie back to vulnerability assessment, the `micro_trj` computation for an MDS fits the signature of the `possible` function in a vulnerability computation. Thus, `possible` can be computed as the structure of micro-trajectories of an MDS. Taking this into account we can compute vulnerability as:

```
vulnerability :: (Functor m, Monad m) => (i -> x -> m x) -> ([x] -> h) -> (m h ->
    v) -> [i] -> x -> v
vulnerability sys harm measure ts = measure . fmap harm . micro_trj sys ts
```

Many computational models used in climate impact research already implicitly implement the structure of an MDS. For instance, the climate model CLIMBER [Petoukhov et al., 2000] has an internal state which represents the current configuration of certain physical parameters. In addition, CLIMBER takes as an input the concentration of greenhouse gases. The CLIMBER model can be described as an MDS built from a family of transition functions with signature `climber :: GHG -> (Climber_st -> F Climber_st)`, where `GHG` is the type of values representing concentration of greenhouse gases, and `F = Id` is the identity monad, since CLIMBER is deterministic.

One monadic system is rarely used alone, at least in vulnerability assessment. It is, in fact, a common task to build a model of a complex system from building blocks. This is in particular characteristic for the type of modelling needed in vulnerability studies. There, the systems of interest are often combined from a physical part (the climate system) and a socio-ecological part (economic or environmental model). These building blocks are often represented in very

different ways: physical systems might be described by deterministic or stochastic systems, while the social systems are usually described as non-deterministic or fuzzy.

Unfortunately, it turns out that the combination of monadic dynamical systems is rather difficult, the main reason for that is that there is no unique way of combining monads to yield monads [King and Wadler, 1992]. The same two monads may be combined differently, yielding different results. Accordingly, Ionescu [2008] presents those combinations of systems which are most likely to be useful in the practice of computational vulnerability assessment. These combination methods include parallel combination and two serial combinations (see Fig. 9.5). Both serial combination methods both require the two systems to be combined to share the same state space. In addition, both methods require that one of the involved systems is a deterministic one. For the discussion of implementational details and the cases where no deterministic system is involved, we refer to Ionescu [2008].

Fig. 9.5 summarises the function signatures of the complete functional prototype. This prototype consists of the system constructors, the trajectory computations, system combination functions (`syspar` for parallel combination, `sysser` for serial combination of systems that share the same monad, `detserl` and `detserr` for parallel combination of a deterministic system with an arbitrary system), and the generic vulnerability computation. Next, we explain how the transformation was used in the process of implementing a ConceptC++ library.

9.3.2 Transformation and ConceptC++ implementation

We use the transformation to process the function signatures presented in Fig. 9.5. The outcome is a translation into ConceptC++ along the lines of the transformation, which serves as a basis for the implementation of the library. Function types are encoded in a straightforward way, using several requirements. To illustrate this, we revisit the type `i -> (x -> m x)`, which is frequently used to encode higher-order functions in Fig. 9.5. The transformation generates concept-based requirements to encode this type in the following way:

```
1 template<class m, class fun1>
2 requires
3   Function<fun1>,
4   Function<fun1::Codomain>,
5   TypeConstructor<m>,
6   Monad<m>,
7   SameType<fun1::Codomain::Codomain,
8            m::Apply<fun1::Codomain::Domain1> >
```

The generation of such function signatures saves indeed a lot of coding that would otherwise have to be done manually. However, on the downside, the transformation has a major disadvantage: it is rather hard to read, as the function signature is encoded by a collection of five requirements. Further, it needs two type parameters to encode the type `i -> (x -> m x)`, but actually one would like to express the whole type as one type parameter with appropriate constraints. A concept that encodes such a type would also provide a domain-specific notion at the level of concepts.

On the other hand, such verbosity is easy to deal with: one can just encapsulate entire function types that are often used within the library. Except for finding a name, such an encapsulation can be done automatically. In this thesis, however, we do it manually. The type `i -> (x -> m x)` is such a type that is often used as argument for higher-order functions within

```
template <class Sys, class Harm, class Meas, class Crtl>
requires
  MonadicSystem<Sys>, Function<Harm>, Function<Meas>,
  Monad<Meas::Domain1>,
  std::SameType<Harm::Domain1, std::vector<Sys::Codomain::Domain1> >,
  std::SameType<Meas::Domain1, Meas::Domain1::Apply<Harm::Codomain> >,
  std::Container<Crtl>,
  std::SameType<Crtl::element_type, Sys::Domain1>
typename Meas::Codomain
struct vulnerability {

    typedef   Sys                       Domain1;
    typedef   Harm                      Domain2;
    typedef   Meas                      Domain3;
    typedef   typename Sys::Domain2     Domain4;
    typedef   Crtl                      Domain5;
    typedef   typename Meas::Codomain   Codomain;

    Codomain operator() (Domain1 x1, Domain2 x2, Domain3 x3,
                         Domain4 x4, Domain5 x5) {

        typedef Sys::Codomain::Apply<std::vector<typename Sys::Domain2> >
            micro_trj_type;
        micro_trj_type micro_trj;

        micro_trajectories(sys, x, crtl.begin(), crtl.end(), micro_trj);
        return m( fmap<Harm, micro_trj_type> (h, micro_trj) );

    }

};
```

Figure 9.6: The vulnerability computation in ConceptC++

the library: it encodes a monadic system, thus we introduced a concept called `MonadicSystem` that encapsulates the entire type by bundling the requirements:

```
concept MonadicSystem<class Sys> {

  requires
    Function<Sys>,
    Function<Sys::Codomain>,
    TypeConstructor<Sys::Codomain::Codomain>,
    Monad<Sys::Codomain::Codomain>,
    SameType<Sys::Codomain::Codomain,
             Sys::Codomain::Codomain::Apply<Sys::Codomain::Domain1> >

};
```

This concept still contains a type-encoding that can be itself factored out. In this case, the type `i -> (x -> m x)` contains the type `x -> m x` that is frequently used to encode argument functions in Fig. 9.5. This type can itself be encoded in a concept called `MonadicCoalgebra`. The `MonadicSystem` concept then becomes simpler:

```
concept MonadicSystem<class Sys> {
```

```
2
3   requires
4     Function<Sys>,
5     MonadicCoalgebra<Sys::Codomain>
6
7 };
```

The requirements for a function containing the type `i -> (x -> m x)` now become much more readable:

```
1 template<class MonSys>
2 requires
3   MonadicSystem<MonSys>
```

For the development of the library we used the output of the transformation as a base to develop a hierarchy of concepts that encode different function types of particular interest for vulnerability computations with monadic dynamical systems. In addition, the library provides the constructors, trajectory computations, system combinators, and the generic vulnerability function from Fig. 9.5. The ConceptC++ output of the transformation for the vulnerability computation function is already shown in Fig. 1.3 in Chap. 1. This generated code is further processed manually, yielding the implementation shown in Fig. 9.6. To improve readability, we renamed the type parameters and used the concept `MonadicSystem` as introduced above. Further, we made the vulnerability computation more flexible by introducing a type parameter for the list of controls. We constrain this type parameter to be a `std::Container`, which means that it can be replaced by the usual STL parameterised container types. As mentioned in Chap. 7, this can be provided as an input to the transformation. The actual vulnerability computation is then similar to the Haskell version: we measure the harm that is possible along the different trajectories of the evolution of the monadic system.

Further details of the formalisation as well as the development process of the library are described elsewhere. In [Wolf et al.], we describe the formalisation process and how it helps to clarify the usage of the term vulnerability, in [Lincke et al., 2008] and [Lincke et al., 2009] we illustrate the evolution of the development process of the library.

Chapter 10

Discussion, Outlook, and Conclusion

In this thesis, we introduced a transformational approach to generic software development that is based on higher-order, typed functional signatures. The motivation for this method is the use of intermediate models and functional prototypes in scientific modelling. However, in Chap. 9 we showed that the method can also be applied to reengineer generic programming libraries.

The key idea of this thesis is to express certain types at the level of concepts. In particular, we describe function types with a function concept and type constructors with a type constructor concept. We then define a transformation that maps higher-order, typed function signatures into function signatures that encode higher-order constructs as type parameters constrained with these concepts. The introduced transformation performs a defunctionalisation at the level of types and lifts type constructors from the level of types to the level of concepts.

There are other possibilities to reimplement a functional prototype in an object-oriented language. For instance, we could have chosen a particular library that provides a specific function datatype, and could have translated function types from functional programming directly. However, such an approach has several disadvantages. First of all, it is language-dependent, as libraries for functional programming and higher-order functions are available in C++, but usually not in other object-oriented languages. Second, it is library-dependent, as a translation into code that depends on a specific library is usually not adaptable to other libraries. Third, it is not easily applicable to higher-order constructs other than higher-order functions. Higher-order type constructors are usually not provided by libraries. Our translation of higher-order constructs into concepts and our transformation of functional, higher-order types signatures into concept-constrained function object signatures overcomes these disadvantages. As we have shown in Chap. 7, an implementation of concepts is nowadays available in many programming languages. Concepts are independent from any particular language family: they are available in imperative, object-oriented, and functional languages. Thus, our approach is very flexible. By lifting the transformation from the level of concrete languages to the level of the two intermediate languages FTKL and CTKL (Chap. 5), the transformation is independent of any concrete programming language. In addition, these two intermediate languages enable us to introduce a semantics for function signatures, and to prove that this semantics is preserved by the transformation (Chap. 6).

However, there are technicalities that are not handled in this thesis. One example is the kind-inference procedure in Chap. 5. We do not provide any formal correctness or completeness statement for this procedure, as we see this procedure as an optional support for the transformation that is not necessary since FTKL function signatures can be annotated with explicit

kinds. The formal verification of kind inference of FTKL is a topic for a separate project.

Another topic that is not handled in this thesis is how one can organise concepts in hierarchies. Object-oriented languages as well as Haskell provide the possibility to organise concepts in hierarchies such that one concept extends another one. For instance, the concepts for functors and monads we presented in Chap. 7 and Chap. 9 are both unary type constructors. Therefore, the concepts `Functor` and `Monad` could extend the `TypeConstructor` concept. Such concept hierarchies are not generated by the transformation; they can not even be expressed in FTKL. Future research could include concept hierarchies in our approach.

Our method is restricted to function signatures and does not process function bodies. This restriction is quite reasonable: besides the fact that full automatic translation of function bodies is hardly possible [Haeri and Schupp, 2011], it is unlikely that, at implementation level, one wanted to keep the code structure. In the scenario of climate-change modeling that motivated us originally, the functional front end serves the purpose of specifying dependencies between subsystems at type level, while the actual implementation of a climate-change model is performance-critical and has at least to be hand-tuned. For instance, recursion from functional programming is typically replaced right away by iterative constructs.

The method presented in this thesis can be extended in various directions. A first possible extension is to transform not only function signatures but also data-type definitions. In functional programming, data types are usually defined as algebraic data types. A research question is how to transform such algebraic datatypes into non-functional languages. However, such a transformation would be indeed language- and library-dependent. In C++, for instance, there exists attempts to implement algebraic data types based on meta-programming libraries like Boost ([Boo]). The Boost libraries provide the parameterised `variant` type that encapsulates a disjoint union type. In general, generic algorithms make use of pattern matching over the constructors of a concrete type, but the property that an arbitrary type is an algebraic data type is not used in generic functions, i.e., there is no pattern matching over an unknown algebraic data type. As for pattern matching the concrete constructors of a data type have to be known, pattern matching over an general algebraic data type specified with a concept is not possible. Thus, a translation of algebraic data types into type parameter specified with an algebraic data type concept is not desirable.

A second extension is to include higher-rank types in the transformation. According to Kfoury and Tiuryn [1992], the rank of a type describes the depth at which universal quantifiers appear contravariantly. An example of a function with rank-2 type is the following:

```
1 g :: (forall a. a -> a) -> (Bool, Char)
2 g f = (f True, f 'a')
```

Here, `g` is not only a higher-order function, it is a function which has an argument that is a parameterised function itself. Using such higher-rank functions with our approach requires a concept for polymorphic functions instead of the monomorphic function concept we present in Chap. 7.

Both extensions (by algebraic datatype and higher-rank types) of our approach would need extensions of all parts of our system: FTKL and CTKL would have to be extended with the appropriate data type definitions or higher-rank types, transformation rules for these constructs would have to be defined, and the backends would have to be extended accordingly.

The motivation for the method presented in this thesis is the use of intermediate models and functional prototypes in scientific modelling. We have shown that such functional prototypes

allow executable computational descriptions to be developed in an exploratory way. For the vulnerability model in the context of climate change, we showed how the development of the functional prototype provided a better understanding of the structure of vulnerability computations. Finally, we used the method presented in this thesis to implement a generic library for vulnerability modelling.

A next application will be the development of an agent-based model of exchange economies from a functional prototype. For such a model, the situation is similar to that of the vulnerability model: a precise, established notion of agent-based models is missing, the structure of underlying computations such as price equilibria computations are not well understood. We are in the process of developing a functional prototype for such a model [Botta et al., 2011], and will, once the prototype is finalised, use the transformation to implement a generic library for such models in C++.

Bibliography

Boost C++ Libraries. `http://www.boost.org/`. Accessed: Nov 28, 2011.

GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`. Accessed: Nov 28, 2011.

D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming. Concepts, Tools, and Techniques from Boost and Beyond*. C++ In-Depth Series. Addison-Wesley, Amsterdam, 2005.

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2 edition, 2006.

T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06*, New York, NY, USA, 2006. ACM.

R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In D. S. Swierstra, P. R. Henriquez, and J. N. Olivieira, editors, *Lecture Notes in Computer Science*, volume 1608, pages 28–115. Springer-Verlag, 1999.

J. Backus. The history of Fortran I, II, and III. In R. L. Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132. UNESCO, 1959.

A. H. Bagge and M. Haveraaen. Axiom-based transformations: Optimisation and testing. *Electronic Notes in Theoretical Computer Science*, 238(5):17–33, 2009.

A.H. Bagge, V. David, and M. Haveraaen. Testing with concepts and axioms in C++. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 773–774, New York, NY, USA, 2008. ACM.

H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

P. Becker. Working Draft, Standard for Programming Language C++. Technical Report N2798=08-0308, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++,, 2011.

J. Bernardy, P. Jansson, M. Zalewski, S. Schupp, and A. Priesnitz. A comparison of C++ concepts and Haskell type classes. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP '08*, pages 37–48, New York, NY, USA, 2008. ACM.

J. Bernardy, P. Jansson, M. Zalewski, S. Schupp, and A. Priesnitz. Generic programming with C++ concepts and Haskell type classes: A comparison. *Journal of Functional Programming*, 20(3-4): 271–302, 2010.

R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1):174–184, 1999.

N. Botta and C. Ionescu. Relation-based computations in a monadic BSP model. *Parallel Computing*, 33(12):795–821, 2007.

N. Botta, C. Ionescu, C. Linstead, and R. Klein. Structuring Distributed Relation-Based Computations With SCDRC. PIK Report 103, Potsdam Institute For Climate Impact Research, 2006.

N. Botta, A. Mandel, C. Ionescu, M. Hofmann, D. Lincke, S. Schupp, and C. Jaeger. A functional framework for agent-based models of exchange. *Applied Mathematics and Computation*, 218(8): 4025–4040, 2011.

A.S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39:617–625, 1997.

N. Broberg. The haskell-src-exts package. `http://hackage.haskell.org/package/haskell-src-exts`. Accessed: Nov 28, 2011.

N. Brooks. Vulnerability, risk and adaptation: A conceptual framework. *Tyndall Center Working Paper*, 38, 2003.

C. J. Burgess. The automated generation of test cases for compilers. *Software testing, Verification and Reliability*, 4(2):81–99, 1994.

L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–523, 1985.

M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *Proceedings tenth ACM SIGPLAN International Conference on Functional Programming, ICFP '05*, pages 241–253. ACM, 2005.

E. J. Chikofski and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7:13–17, 1990.

W. Chin and J. Darlington. A higher-order removal method. *Lisp and Symbolic Computation*, 9(4): 287–322, December 1996.

A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 268–279. ACM, 2000.

V. Cremet and P. Altherr. Adding type constructor parameterization to Java. *Journal of Object Technology*, 7(5):25–65, 2008.

H. B. Curry and R. Feys. *Combinatory Logic: Volume 1*. North Holland, 1958.

K. Czarnecki. *Generative Programming - Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technical University of Ilmenau, 1998.

O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 162–174, New York, NY, USA, 2001. ACM.

J. de Guzman and D. Marsden. Fusion library homepage. `http://www.boost.org/libs/fusion`. Accessed: Nov 28, 2011.

G. Dos Reis and J. Järvi. What is generic programming? In A. Lumsdaine, S. Schupp, D. Musser, and J. Siek, editors, *Proceedings of the First International Workshop of Library-Centric Software Design, LCSD '05. An OOPSLA '05 workshop*, 2005.

A. Duret-Lutz. Expression templates in ada. In *Proceedings of the 6th Ade-Europe International Conference Leuven on Reliable Software Technologies*, Ada Europe '01, pages 191–202, London, UK, 2001. Springer-Verlag.

M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

S. Frankau and A. Mycroft. Stream processing hardware from functional language specifications. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences, HICSS '03 - Track 9*, Washington, DC, USA, 2003. IEEE Computer Society.

R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '03*, pages 115–134. ACM Press, 2003.

R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17:145–205, 2007.

J. Gibbons. Datatype-generic programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 1–72. Springer-Verlag, 2007.

J. Gibbons and O. de Moor, editors. *The Fun of Programming*. Cornerstones in Computing. Palgrave, 2003.

J. Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

J. Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The 3rd Edition*. Addison-Wesley Professional, 2005.

D. Gregor. ConceptGCC. `http://www.generic-programming.org/software/ConceptGCC/`, a. Accessed: Nov 28, 2011.

D. Gregor. What happened in Frankfurt? C++-Next, The next generation of C++, `http://cpp-next.com/archive/2009/08/what-happened-in-frankfurt/`, b. Accessed: Nov 28, 2011.

D. Gregor and B. Stroustrup. Concepts (Revision 1). Technical report, 2006. Document number: N2081=06-0151, Project: Programming Language C++, Evolution Working Group.

D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06*, pages 291–310.

D. Gregor, B. Stroustrup, J. Widman, and J. Siek. Proposed Wording for Concepts (Revision 8). Technical Report N2741=08-0251, ISO/IEC JTC1/SC22/WG21 - C++, Aug 2008.

S. H. Haeri and S. Schupp. Functional metaprogramming in C++ and cross-lingual development with Haskell. In *Proceedings of the 23rd Symposium on Implementation and Application of Functional Languages, IFL2011*, 2011.

R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

R. Hinze and N. Wu. Towards a categorical foundation for generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '11, pages 47–58, New York, NY, USA, 2011. ACM.

W. A. Howard. The formulas-as-types notion of construction. In J.P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

P. Hudak and M.P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity. Research Report YALEU/DCS/RR-1049, Department of Computer Science, Yale University, New Haven, CT, 1994.

C. Ionescu. *Vulnerability Modelling and Monadic Dynamical Systems*. PhD thesis, Freie Universität Berlin, 2008.

M. A. Janssen and E. Ostrom. Resilience, vulnerability and adaptation: A cross-cutting theme of the international human dimensions programme on global environmental change. *Global Environmental Change*, 16(3):237–239, 2006. Editorial.

P. Jansson and J. Jeuring. Polyp—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97*, pages 470–482, New York, NY, USA, 1997. ACM.

J. Järvi, G. Powell, and A. Lumsdaine. The lambda library: unnamed functions in C++. *Software: Practice and Experience*, 33:259–291, 2003a.

J. Järvi, J. Willcock, and A. Lumsdaine. Concept-controlled polymorphism. In F. Pfennig and Y. Smaragdakis, editors, *Generative Programming and Component Engineering, volume 2830 of LNCS*, pages 228–244. Springer Verlag, 2003b.

S.P. Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1–82, 2007.

N. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 1999.

A. J. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order lambda calculus. *Information and Computation*, 98:228–257, June 1992.

D. King and P. Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Glasgow Workshop on Functional Programming*, pages 134–143. Springer, 1992.

J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 26–37. ACM Press, 2003.

K. Läufer. A framework for higher-order functions in C++. In *Proceedings of the USENIX Conference on Object-Oriented Technologies on USENIX Conference on Object-Oriented Technologies, COOTS '95*, pages 103–116, Berkeley, CA, USA, 1995. USENIX Association.

D. Leivant. Polymorphic type inference. In *Proceedings of the 10th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '83*, pages 88–98. ACM, 1983.

Y. Li and M. Leeser. HML: An innovative hardware description language and its translation to VHDL. In *Proceedings of the International Conference on Computer Hardware Description Languages and their Applications, CHDL '95*, pages 691–696, 1995.

D. Lincke. A transformational approach to generic programming. `http://www.pik-potsdam.de/members/lincke/tagp`. Accessed: Jun 26, 2012.

D. Lincke and S. Schupp. The Function Concept in C++ - An Empirical Study. In *Proceedings of the ACM SIGPLAN workshop on Generic programming, WGP '09*, New York, NY, USA, 2009. ACM.

D. Lincke, C. Ionescu, and N. Botta. A generic library for earth system modelling based on monadic systems. In *Digital Earth Summit on Geoinformatics*, pages 188–194. UNESCO, 2008.

D. Lincke, P. Jansson, M. Zalewski, and C. Ionescu. Generic libraries in C++ with concepts from high-level domain descriptions in Haskell. A domain-specific library for computational vulnerability assessment. In *Proceedings of the IFIP Working Conference on Domain Specific Languages*, pages 236–261, 2009.

D. MacQueen. Adaptation in HOT languages: Comparing polymorphism, modules, and objects. In *Engineering theories of software construction*, pages 47–96. IOS Press, 2001.

B. McNamara. *Multiparadigm programming: Novel devices for implementing functional and logic programming constructs in C++*. PhD thesis, Georgia Institute of Technology, 2004.

B. McNamara and Y. Smaragdakis. Functional programming with the FC++ library. *Journal of Functional Programming*, 14(4):429–472, 2004.

B. Meyer. Genericity versus inheritance. In *Proceedings of the 3rd ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 391–405, New York, NY, USA, 1986. ACM.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10:470–502, July 1988.

A. Moors. *Type Constructor Polymorphism for Scala: Theory and Practice*. PhD thesis, Katholieke Universiteit Leuven, 2009.

A. Moors, F. Piessens, and M. Odersky. Towards equal rights for higher-kinded types. In *6th International Workshop on Multiparadigm Programming with Languages at the European Conference on Object-Oriented Programming (ECOOP)*, pages 1–4, 2007.

A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '08, pages 423–438, New York, NY, USA, 2008. ACM.

C. Morgan. *Programming from specifications*. Prentice Hall International, Hertfordshire, 1990.

A. Mycroft and R. Sharp. Hardware/software co-design using functional languages. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 236–251, London, UK, 2001. Springer-Verlag.

N. Myers. A new and useful template technique: "Traits". *C++ Report*, 7(5):32–35, 1995.

P. Naur. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6:1–17, 1963.

G. Nelan. *Firstification*. PhD thesis, Department of Computer Science, Arizona State University, 1994.

J. Nordlander. Polymorphic subtyping in O'Haskell. *Science of Computer Programming*, 43(2-3): 93–127, 2002.

M. Odersky. Poor man's type classes., 2006. http://lamp.epfl.ch/∼odersky/talks/wg2.8-boston06.pdf.

M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97*, pages 146–159, 1997.

M. Odersky and P. Wadler. Leftover curry and reheated pizza: How functional programming nourishes software reuse. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 2–11, Washington, DC, USA, 1998. IEEE Computer Society.

M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. Mcdirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the scala programming language (second edition). Technical report, Swiss Federal Institute of Technology in Lausanne, 2006.

J. O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, Washington, DC, USA, 2002. IEEE Computer Society.

V. Petoukhov, A. Ganopolski, V. Brovkin, M. Claussen, A. Eliseev, C. Kubatzki, and S. Rahmstorf. CLIMBER-2 : a climate system model of intermediate complexity. Part I : model description and performance for present climate. *Climate dynamics*, 16:1–17, 2000.

T. Petricek and J. Skeet. *Real-World Functional Programming. With examples in F# and C#*. Manning, 2009.

S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.

F. Pottier and N. Gauthier. Polymorphic typed defunctionalization. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '04*, pages 89–98. ACM Press, 2004.

F. P. Ramsey. The foundations of mathematics. *Proceedings of the London Mathematical Society, Series 2*, 25(5):338–384, 1925.

D. Remy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89*, pages 40–53, 1997.

T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40:5–19, 1998.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, pages 717–740, New York, NY, USA, 1972. ACM.

J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag.

A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN Symposium on Haskell*, pages 111–122, New York, NY, USA, 2008. ACM.

A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing datatype generic libraries in Haskell. *Journal of Functional Programming*, 2010. accepted for publication.

N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *In Proc. European Conference on Object-Oriented Programming*, pages 248–274. Springer, 2003.

T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP '08: Proc. 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62. ACM, 2008.

R. Sharp and A. Mycroft. A higher-level language for hardware synthesis. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME '01*, pages 228–243, London, UK, 2001. Springer-Verlag.

J. Siek, L.Q. Lee, and A. Lumsdaine. *The Boost Graph Library.* Addison-Wesley, 2002.

I. Sommerville. *Software Engineering*. Addison-Wesley, 2006.

A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, 1994.

C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1-2):11–49, April 2000.

B. Stroustrup. *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley, 2000.

V. Stuikys and R. Damasevicius. Taxonomy of the program transformation processes. *Information Technology and Control*, 22:39–52, 2002.

D. A. Turner. Functional programs as executable specifications. In *Proceedings of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, pages 29–54, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

D. Vandervoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, Amsterdam, 2002.

E. Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57(2), 2001.

E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

P. L. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89*, pages 60–76, New York, NY, USA, 1989. ACM.

A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1925–1927.

S. Wolf, J. Hinkel, M. Hofmann, A. Bisaro, D. Lincke, C. Ionescu, and R. J.T. Klein. Vulnerability definitions and methodologies - a clarification by formalisation. *Ecology and Society*. submitted.

S. Wolf, D. Lincke, J. Hinkel, C. Ionescu, and S. Bisaro. A formal framework of vulnerability. Final deliverable to the ADAM project. FAVAIA working paper 8, Potsdam Institute for Climate Impact Research, Potsdam, Germany, 2008.

M. Zalewski. *Generic Programming with Concepts*. PhD thesis, Chalmers University of Technology, Gothenburg, 2008.

M. Zalewski, A. Priesnitz, C. Ionescu, N. Botta, and S. Schupp. Multi-language library development. From Haskell type classes to C++ concepts. In J. Striegnitz, editor, *Proceedings 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2007.

Marcin Zalewski and Sibylle Schupp. A semantic definition of separate type checking in C++ with concepts. *Journal of Object Technology*, 8(5):105–132, 2009.

Appendix A

Algorithms

Below we present the two helper algorithms `find_fct` and `replace_type` used in Chap. 6. Both algorithms are presented for FTKL inputs, but also used with FTKL' and CTKL inputs. The versions for FTKL' and CTKL inputs only contain minor differences.

A.1 Algorithm: find_fct

Input: An FTKL type `t`.

Output: An element of type `Maybe x -> y` containing the first function type found, or `Nothing` if there is no function type in `t`.

A1. Case distinction on the type of `t`:

- `t1 -> t2`:
  Return `Just (t1 -> t2)`.
- `(t1, t2, ..., tN)`:
  for i=1, ..., N
  - If `find_fct(t1) != Nothing` return `find_fct(t1)`.

  Return `Nothing`.
- `(t)`:
  Return `find_fct(t)`.
- `t1 t2`:
  If `find_fct(t1) != Nothing` return `find_fct(t1)`.
  Return `find_fct(t2)`.
- otherwise:
  Return `Nothing`.

Algorithm A.1: `find_fct`: find the first function type in an FTKL type

A.2 Algorithm: replace_type

Input: Arbitrary FTKL types `t`, `t1`, and `t2`.
Output: `t`, where every occurrence of `t1` is replaced by `t2`.

A1. If `t == t1` return `t2`.

A2. Case distinction on the type of `t`:

- `tx -> ty`:
  - Compute `(t', C', N')` = `concept_encode_HOF (t2_i, C, N).`
  - Set `tx' = replace_type(tx,t1,t2).`
  - Set `ty' = replace_type(ty,t1,t2).`
  - Return `tx' -> ty'`.
- `(t_1, ..., t_N)`:
  for i=1, . . . , N
  - Set `t_i = replace_type(t_i,t1,t2).`

  Return `(t_1, ..., t_N)`.
- `(t)`:
  Return `t = replace_type(t,t1,t2).`
- `tx ty`:
  Return `replace_type(tx,t1,t2) replace_type(ty,t1,t2).`
- Otherwise:
  Return `t`.

Algorithm A.2: `replace_type`: replace in an FTKL type one type by another type

Appendix B

Test Output for ConceptC++

```
f :: Int -> Int

struct F {

    typedef   Int  Domain1;
    typedef   Int  Codomain;

    Codomain operator() (Domain1 x1) {
    }

};


f :: a -> a

template<class a>
struct F {

    typedef   a  Domain1;
    typedef   a  Codomain;

    Codomain operator() (Domain1 x1) {
    }

};


f :: a -> a -> a

template<class a>
struct F_codomain1 {

    typedef   a  Domain1;
    typedef   a  Codomain;

    Codomain operator() (Domain1 x1) {
    }

};

template<class a>
```

```
42 struct F {
43
44     typedef   a  Domain1;
45     typedef   F_codomain1  Codomain;
46
47     Codomain operator() (Domain1 x1) {
48     }
49
50 };
51
52
53 f :: (a,a) -> a
54
55 template<class a>
56 struct F {
57
58     typedef   a  Domain1;
59     typedef   a  Domain2;
60     typedef   a  Codomain;
61
62     Codomain operator() (Domain1 x1, Domain2 x2) {
63     }
64
65 };
66
67
68 f :: Ord a => a -> a -> a
69
70 template<class a>
71   requires
72     Ord<a>
73 struct F_codomain1 {
74
75     typedef   a  Domain1;
76     typedef   a  Codomain;
77
78     Codomain operator() (Domain1 x1) {
79     }
80
81 };
82
83 template<class a>
84   requires
85     Ord<a>
86 struct F {
87
88     typedef   a  Domain1;
89     typedef   F_codomain1  Codomain;
90
91     Codomain operator() (Domain1 x1) {
92     }
93
94 };
95
96
97 f :: (a -> b) -> b
98
99 template<class fun1>
```

```
100   requires
101 Function<fun1>
102 struct F {
103
104     typedef   fun1  Domain1;
105     typedef   fun1::Codomain  Codomain;
106
107     Codomain operator() (Domain1 x1) {
108     }
109
110 };
111
112
113 g :: x -> f x
114
115 template<class x, class f>
116   requires
117     TypeConstructor<f>
118 struct G {
119
120     typedef   x  Domain1;
121     typedef   f::Apply<x>  Codomain;
122
123     Codomain operator() (Domain1 x1) {
124     }
125
126 };
127
128
129 inpsys  :: (Monad m, Functor m) => ((x,a) -> m x) -> List a  -> x -> m x
130
131 template<class m, class fun1>
132   requires
133     Monad<m>,
134     Functor<m>,
135     Function2<fun1>,
136     SameType<fun1::Codomain, m::Apply<fun1::Domain1>>,
137     TypeConstructor<m>
138 struct Inpsys_codomain1_codomain1 {
139
140     typedef   fun1::Domain1  Domain1;
141     typedef   m::Apply<fun1::Domain1>  Codomain;
142
143     Codomain operator() (Domain1 x1) {
144     }
145
146 };
147
148 template<class m, class fun1>
149   requires
150     Monad<m>,
151     Functor<m>,
152     Function2<fun1>,
153     SameType<fun1::Codomain, m::Apply<fun1::Domain1>>,
154     TypeConstructor<m>
155 struct Inpsys_codomain1 {
156
157     typedef   std::list<fun1::Domain2>  Domain1;
```

```
158     typedef   Inpsys_codomain1_codomain1  Codomain;
159
160     Codomain operator() (Domain1 x1) {
161     }
162
163 };
164
165 template<class m, class fun1>
166   requires
167     Monad<m>,
168     Functor<m>,
169     Function2<fun1>,
170     SameType<fun1::Codomain, m::Apply<fun1::Domain1>>,
171     TypeConstructor<m>
172 struct Inpsys {
173
174     typedef   fun1  Domain1;
175     typedef   Inpsys_codomain1  Codomain;
176
177     Codomain operator() (Domain1 x1) {
178     }
179
180 };
181
182
183 f :: ((a -> c,b) -> c, b, a) -> (c,(a -> b))
184
185 template<class fun2>
186   requires
187     Function<fun2::Domain1>
188     Function2<fun2>
189     SameType<fun2::Codomain, fun2::Domain1::Codomain>
190 struct F_codomain1 {
191
192     typedef   fun2::Domain1::Domain1  Domain1;
193     typedef   fun2::Domain2  Codomain;
194
195     Codomain operator() (Domain1 x1) {
196     }
197
198 };
199
200 template<class fun2>
201   requires
202     Function<fun2::Domain1>,
203     Function2<fun2>,
204     SameType<fun2::Codomain, fun2::Domain1::Codomain>
205 struct F {
206
207     typedef   fun2  Domain1;
208     typedef   fun2::Domain2  Domain2;
209     typedef   fun2::Domain1::Domain1  Domain3;
210     typedef   Tuple2<fun2::Domain1::Codomain, F_codomain1>  Codomain;
211
212     Codomain operator() (Domain1 x1, Domain2 x2, Domain3 x3) {
213     }
214
215 };
```

```
216
217
218 f :: ((b -> (b,c)), ((a,b) -> b)) -> a -> c
219
220 template<class c, class fun1, class fun2>
221   requires
222     Function<fun1>,
223     SameType<fun1::Codomain, Tuple2<fun1::Domain1, c>>,
224     Function2<fun2>,
225     SameType<fun2::Domain2, fun1::Domain1>,
226     SameType<fun2::Codomain, fun1::Domain1>
227 struct F_codomain1 {
228
229     typedef   fun2::Domain1  Domain1;
230     typedef   c  Codomain;
231
232     Codomain operator() (Domain1 x1) {
233     }
234
235 };
236
237 template<class c, class fun1, class fun2>
238   requires
239     Function<fun1>,
240     SameType<fun1::Codomain, Tuple2<fun1::Domain1, c>>,
241     Function2<fun2>,
242     SameType<fun2::Domain2, fun1::Domain1>,
243     SameType<fun2::Codomain, fun1::Domain1>
244 struct F {
245
246     typedef   fun1  Domain1;
247     typedef   fun2  Domain2;
248     typedef   F_codomain1  Codomain;
249
250     Codomain operator() (Domain1 x1, Domain2 x2) {
251     }
252
253 };
254
255
256 fold    :: Bifunctor s => (s a b -> b) -> Fix s a -> b
257 fold    :: Bifunctor s => ((s :: * -> * -> *) (a :: *) (b :: *) -> b)
258                         -> (Fix :: (* -> * -> *) -> * -> *) s a -> b
259
260 template<class s, class a, class fun1>
261   requires
262     Bifunctor<s>,
263     Function<fun1>,
264     SameType<fun1::Domain1, s::Apply<a, fun1::Codomain>>,
265     TypeConstructor2<s>
266 struct Fold_codomain1 {
267
268     typedef   Fix<s, a>  Domain1;
269     typedef   fun1::Codomain  Codomain;
270
271     Codomain operator() (Domain1 x1) {
272     }
273
```

```
274 };
275
276 template<class s, class a, class fun1>
277   requires
278     Bifunctor<s>,
279     Function<fun1>,
280     SameType<fun1::Domain1, s::Apply<a, fun1::Codomain>>,
281     TypeConstructor2<s>
282 struct Fold {
283
284     typedef   fun1  Domain1;
285     typedef   Fold_codomain1  Codomain;
286
287     Codomain operator() (Domain1 x1) {
288     }
289
290 };
291
292
293 f :: (H (F I (J a b c))) c -> K (G I)
294
295 template<class a, class b, class c>
296 struct F {
297
298     typedef   H<F<I, J<a, b, c>>, c>  Domain1;
299     typedef   K<G<I>>  Codomain;
300
301     Codomain operator() (Domain1 x1) {
302     }
303
304 };
305
306
307 f :: f g -> g a
308
309 template<class f, class g, class a>
310   requires
311     TypeConstructor<f>,
312     TypeConstructor<g>
313 struct F {
314
315     typedef   f::Apply<g>  Domain1;
316     typedef   g::Apply<a>  Codomain;
317
318     Codomain operator() (Domain1 x1) {
319     }
320 };
```

Appendix C

Transformation Output for Origami++

Below we present the generated ConceptC++ function signatures for one of our applications of the transformation, more precisely the generated function signatures for the four operators of the Origami++ library.

C.1 Map

```
template<class s, class fun1>
  requires
    Bifunctor<s>,
    Function<fun1>
struct Map_codomain1 {

    typedef   Fix<s, fun1::Domain1>   Domain1;
    typedef   Fix<s, fun1::Codomain>  Codomain;

    Codomain operator() (Domain1 x1) {
    }

};

template<class s, class fun1>
  requires
    Bifunctor<s>,
    Function<fun1>
struct Map {

    typedef   fun1           Domain1;
    typedef   Map_codomain1  Codomain;

    Codomain operator() (Domain1 x1) {
    }

};
```

Figure C.1: The generated function signatures of the map function

C.2 Fold

```
template<class s, class a, class fun1>
  requires
    Bifunctor<s>,
    Function<fun1>,
    SameType<fun1::Domain1, s::Apply<a, fun1::Codomain>>,
    TypeConstructor2<s>
struct Fold_codomain1 {

    typedef   Fix<s, a>       Domain1;
    typedef   fun1::Codomain  Codomain;

    Codomain operator() (Domain1 x1) {
    }

};

template<class s, class a, class fun1>
  requires
    Bifunctor<s>,
    Function<fun1>,
    SameType<fun1::Domain1, s::Apply<a, fun1::Codomain>>,
    TypeConstructor2<s>
struct Fold {

    typedef   fun1            Domain1;
    typedef   Fold_codomain1  Codomain;

    Codomain operator() (Domain1 x1) {
    }

};
```

Figure C.2: The generated function signatures of the fold function

C.3 Unfold

```
template<class s, class a, class fun1>
  requires
    Bifunctor<s>,
    Function<fun1>,
    SameType<fun1::Codomain, s::Apply<a, fun1::Domain1>>,
    TypeConstructor2<s>
struct Unfold_codomain1 {

    typedef   fun1::Domain1  Domain1;
    typedef   Fix<s, a>      Codomain;

    Codomain operator() (Domain1 x1) {
    }

};

template<class s, class a, class fun1>
  requires
    Bifunctor<s>,
    Function<fun1>,
    SameType<fun1::Codomain, s::Apply<a, fun1::Domain1> >,
    TypeConstructor2<s>
struct Unfold {

    typedef   fun1              Domain1;
    typedef   Unfold_codomain1  Codomain;

    Codomain operator() (Domain1 x1) {
    }

};
```

Figure C.3: The generated function signatures of the unfold function

C.4 Hylo

```
template<class s, class fun1, class a, class fun2>
  requires
    Bifunctor<s>,
    Function<fun1>,
    SameType<fun1::Codomain, s::Apply<a, fun1::Domain1>>,
    Function<fun2>,
    SameType<fun2::Domain1, s::Apply<a, fun2::Codomain>>,
    TypeConstructor2<s>
struct Hylo_codomain1_codomain1 {

    typedef   fun1::Domain1   Domain1;
    typedef   fun2::Codomain  Codomain;

    Codomain operator() (Domain1 x1) {
    }
};

template<class s, class fun1, class a, class fun2>
  requires
    Bifunctor<s>,
    Function<fun1>,
    SameType<fun1::Codomain, s::Apply<a, fun1::Domain1>>,
    Function<fun2>,
    SameType<fun2::Domain1, s::Apply<a, fun2::Codomain>>,
    TypeConstructor2<s>
struct Hylo_codomain1 {

    typedef   fun2                      Domain1;
    typedef   Hylo_codomain1_codomain1  Codomain;

    Codomain operator() (Domain1 x1) {
    }
};

template<class s, class fun1, class a, class fun2>
  requires
    Bifunctor<s>,
    Function<fun1>,
    SameType<fun1::Codomain, s::Apply<a, fun1::Domain1>>,
    Function<fun2>,
    SameType<fun2::Domain1, s::Apply<a, fun2::Codomain>>,
    TypeConstructor2<s>
struct Hylo {

    typedef   fun1  Domain1;
    typedef   Hylo_codomain1  Codomain;

    Codomain operator() (Domain1 x1) {
    }
};
```

Figure C.4: The generated function signatures of the hylo function

Acknowledgements

I would like to thank Prof. Dr. Sibylle Schupp, my supervisor, for various reasons. Firstly, for being an excellent supervisor, teaching me the secrets of good scientific practice. Moreover, I would like to thank Sibylle for always finding the time to help me with my research, even when the fact that the supervision was an external one made it sometimes difficult to coordinate.

I would further like to thank Prof. Dr. Friedrich Mayer-Lindenberg for being the second referee of this thesis, and for organising the less academic aspects of my visits at the TU Hamburg-Harburg I thank Ulrike Hantschmann.

All of this work has been greatly facilitated by the environment provided by the Potsdam Institute for Climate Impact Research, especially by the research domain Transdisciplinary Concepts and Methods, whose members, from doctoral students to senior scientists and professors, constitute an amazing reservoir of knowledge, brilliance, and wisdom. This especially includes my colleagues from the FAVAIA working group: Cezar Ionescu, Jochen Hinkel, Richard Klein, who were conviced that hiring me as a PhD-student is a good idea. Further, this includes Carlo Jaeger and the Lagom group (Antoine Mandel, Steffen Fürst, Wiebke Lass, Sarah Wolf) who patiently tried to understand the basics of my work. A pleasant atmosphere is an important factor for such a thesis and I would like to thank my officemates, Sarah Wolf, Sandy Bisaro, Alexandra Beckstein, Mareen Hofmann and Elke Henning (chronological order) for many interesting exchanges and a pleasant working environment in general.

On a final note, I would like to acknowledge my family, my parents Volker and Angelika Lincke and my sister Manja Schramm. Furthermore, there are many friends that provided a lot of fun outside research, special thanks go to Adam Ailsby, Ina Dunst, Marek Haustein, Stefan Keichel, Christian Pape and Konstanze Schwedka. Thanks, guys!