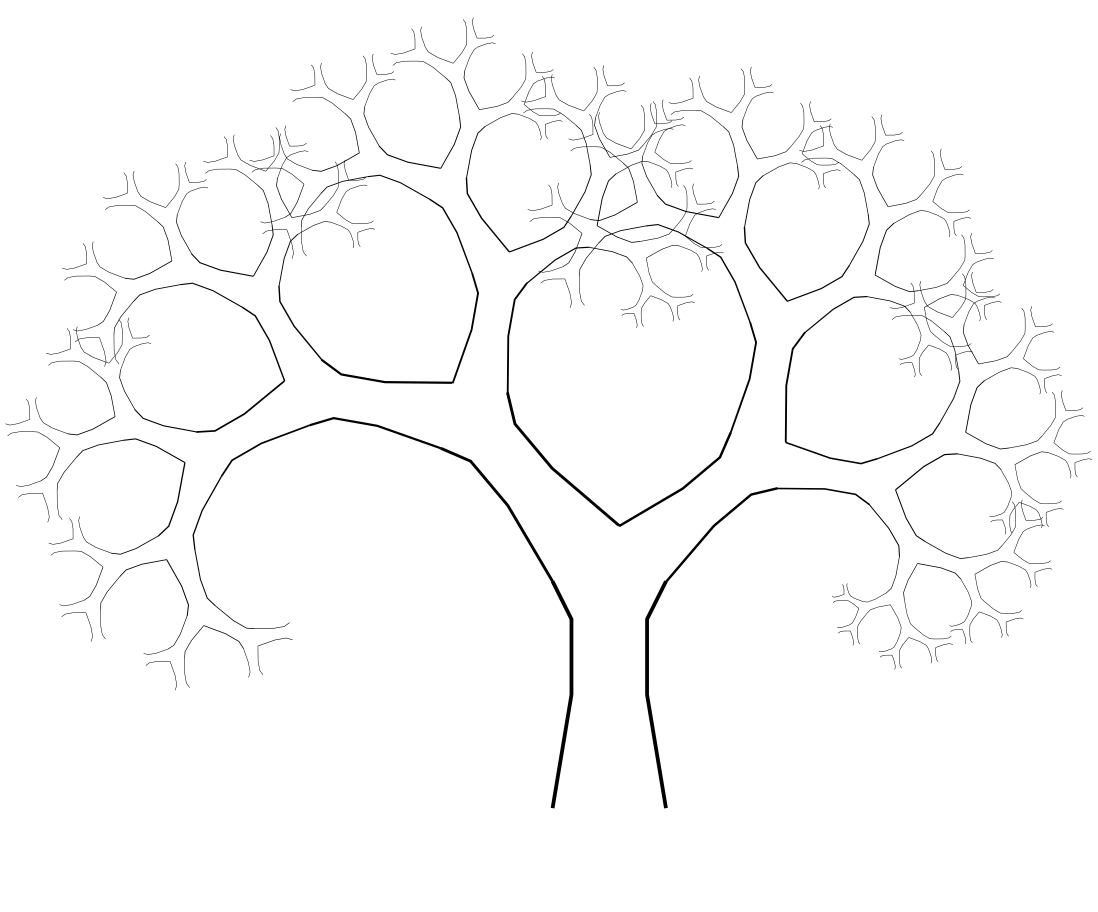
Proof Support for Type Classes

Master's thesis

Ron van Kesteren Februari 2005



Colophon

Date:	Februari 2005
Name:	Ron van Kesteren (`R.vanKesteren@student.science.ru.nl`)
Student number:	9902295
Study:	Computing Science (Foundations of Computing Science)
Thesis number:	535
Supervisor:	Marko van Eekelen
Assistant supervisor:	Maarten de Mol
Referent:	Sjaak Smetsers

Nijmegen Institute for Computing and Information Sciences
Faculty of Science
Radboud University Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

Abstract

Proving properties about a program can significantly help creating high quality reliable software. Many general purpose proof assistants exist, but these are hard to use when applied to more practical domains such as actual programming languages. Because of this, special purpose proof assistants have been developed. In this thesis, the focus is on Sparkle; a proof assistant for the functional programming language Clean.

A feature that is commonly found in functional programming languages is overloading structured by type classes. Type classes essentially are groups of types, the class instances, for which certain operations are implemented. These implementations are created from the available instance definitions and may be different for each instance. An important observation regarding type classes is that, in general, the defined instances should be semantically related. For example, all instances of the equality operator usually implement an equivalence relation.

In this project, proof support for type classes is developed and added to Sparkle. Properties about type classes are specified as class constrained properties. A new proof rule and an effective tactic are presented for proving class constrained properties by proving them for the available instance definitions. This is not straightforward, because instance definitions may depend on each other. The proof assistant Isabelle handles this problem for single parameter type classes by structural induction on types. However, this does not suffice for an effective tactic for multi-parameter classes. This is solved using an induction scheme derived from the instance definitions.

The tactic is implemented in the proof assistant Sparkle, but the result is general and can be used for other programming languages and proof assistants as well.

This work was presented at the Fifth Symposium on Trends in Functional Programming 2004 (TFP 2004) at the Ludwig-Maximilians University in Munich, Germany. Based on the comments an improved version was published as a technical report (NIII-R05001) and submitted for review for the TFP 2004 selected papers, to be published by Intellect.

Preface

"What's in a name? That which we call a rose by any other word would smell as sweet; So Romeo would, were he not Romeo call'd, retain that dear perfection which he owes without that title."

This excerpt from Romeo and Juliet by William Shakespeare appears to be perfectly true in its dramatic context, but one should be careful not to interpret it too literally. Although something does not lose its properties when it is given another name and a name may not define all properties, two things having the same name does suggest that both have some properties in common: both red and yellow roses smell sweet, any chair can be used to sit on, and every language is used for communication. One could say that a name represents a class of which all instances share some properties. In some cases, these properties are even sufficient to define the class. Hence, without doubt, the name has a connection to certain properties.

This observation lies at the heart of this thesis. Here, the classes considered are classes in a functional programming language. In most functional languages, several functions that have the same name can be defined, each operating on different types (overloading). Often, these functions share some important properties, otherwise there would be very little reason to give them the same name. This thesis deals with proving these properties for the entire class at once, instead of separately for all class members.

The application of formal reasoning and theory for a new result is what interested me the most in this project. A computing science student gets a lot of exercise in understanding and working with existing definitions. However, creating new definitions yourself introduces a form of uncertainty. For example, there was a large number of possible choices for the level at which to formalize the solution. In some way, publishing this work as an article confirmed the choices made and furthermore increased my understanding of what I have actually done.

I would like to express my gratitude to everyone who contributed to this thesis. First of all, my supervisor Marko van Eekelen for help on formalization and generalization and his remarks on scientific research in general. Secondly, Maarten de Mol, the author of SPARKLE, for sharing his experience on SPARKLE and his insight in the problem domain, and Sjaak Smetsers for clarifying how Clean implements type classes. Of course, I would also like to thank the Nijmegen Institute for Computing and Information Sciences (NIII) for making it possible to present this work at the Fifth Symposium on Trends in Functional Programming (TFP 2004) at the Ludwig-Maximilians University in Munich, Germany. I should not forget to mention Chris Heunen for various hints on mathematical notation and for playing music and many games of table football before he went to Canada. And finally, I would like to thank my family and friends for their support and encouragement during the time I was working on this thesis.

Thanks!
Ron van Kesteren, Februari 2005

Contents

Introduction 1

In this chapter, the subject of this project and the problem that is solved are explained. The problem is formulated as a research goal and a research question. The research goal states the practical result that is accomplished by the project. The research question is the scientific question that is answered in this thesis.

Section 1.1 introduces the programming language CLEAN and SPARKLE; a proof assistant for CLEAN programs. SPARKLE's main advantage is explained in section 1.2. This thesis solves one of the shortcomings of SPARKLE, proving properties about overloaded expressions, which is presented in section 1.3. The kind of properties we would like to prove, class constrained properties, is presented in section 1.4. Section 1.5 summarizes the goal of the project and the research question. Finally, section 1.6 gives the outline of this thesis.

1.1 Clean and Sparkle

This thesis is about proof support for CLEAN programs that use type classes. CLEAN [17] is a pure and lazy functional programming language developed at the Software Technology department of the Radboud University Nijmegen. It is available free of charge at: `http://www.cs.ru.nl/~clean`. An important property of a pure functional language is referential transparency, which means that the result of a function only depends on the function parameters. This makes it relatively easy to reason about the behavior of a CLEAN program.

To bring together programming and reasoning, the proof assistant SPARKLE was developed [5]. SPARKLE is specialized for CLEAN and can be used to prove properties about (parts of) arbitrary CLEAN programs. Before a property about a program can be proven, the program must be translated to the specification language of the proof assistant. Then, the proof can be conducted on the translated program. SPARKLE uses a subset of CLEAN, called CORE, as its specification language, which makes the translation relatively straightforward.

Properties can be specified in the well known first order predicate logic extended with equality of expressions. By selecting one of the more than forty (parameterized) tactics, the property is transformed into (a number of) the properties, the *proof goals* or *proof obligation*, that are hopefully easier to prove. The property is proven when all remaining proof goals are trivial. The tactics are specifically tailored to functional languages; `Reduction` is a very useful tactic that can be used to reduce all expressions in the current goal to root normal form, `Induction` performs structural induction on expression variables.

Example 1.1.1 (example property)

A property that can be proven using Sparkle is:

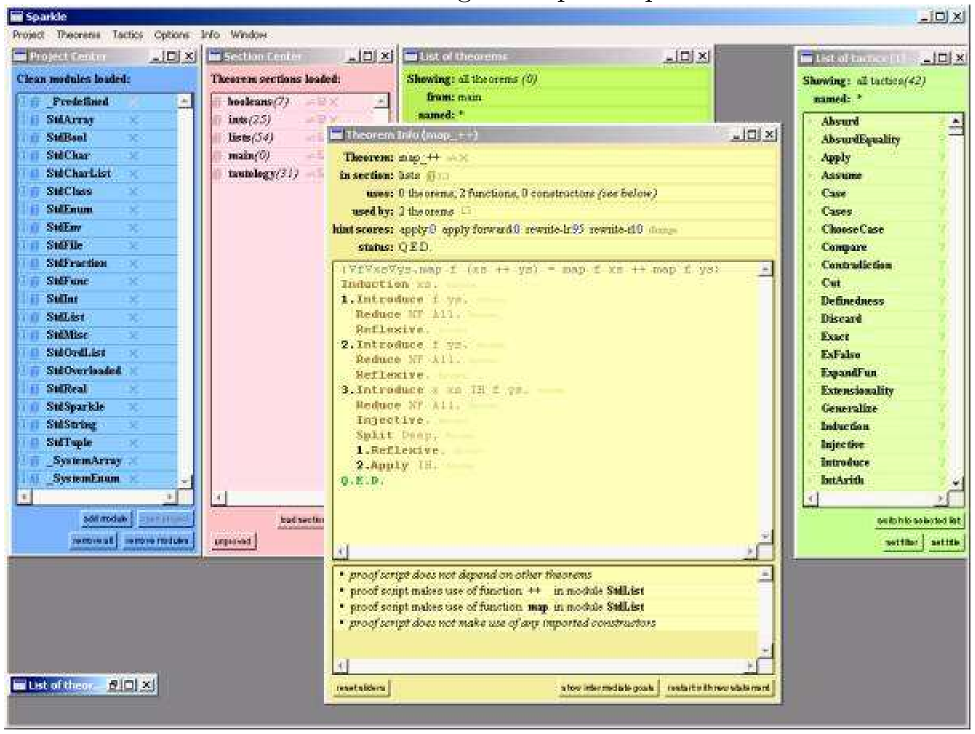
$\forall_{\texttt{n:Int}}[\texttt{n} \neq \bot \Rightarrow \forall_{\texttt{a}}\forall_{\texttt{xs:[a]}}[\texttt{take n xs ++ drop n xs} = \texttt{xs}]]$

This property can be proven using the `Induction` tactic on `n` and `xs`.

The key in proving a property is selecting the right tactics. This requires knowledge of the tactics and proofs in general. Fortunately, Sparkle features a hint mechanism that aids in selecting tactics. Based on the current goal, tactics are assigned a probability rating of 1 to 100 indicating how likely the application of the tactic will help proving the goal. By setting a threshold for automatical application, Sparkle can also try to automatically build a proof.

Example 1.1.2 (user interface)

The Sparkle user interface showing a completed proof:



Theorem: map_++
in section: lists
uses: 0 theorems, 2 functions, 0 constructors *(see below)*
used by: 2 theorems
hint scores: apply 0 apply forward 0 rewrite-lr 95 rewrite-rl 0
status: Q.E.D.

```
{∀f∀xs∀ys.map f (xs ++ ys) = map f xs ++ map f ys}
Induction xs.
1.Introduce f ys.
  Reduce NF All.
  Reflexive.
2.Introduce f ys.
  Reduce NF All.
  Reflexive.
3.Introduce x xs IH f ys.
  Reduce NF All.
  Injective.
  Split Deep.
  1.Reflexive.
  2.Apply IH.
Q.E.D.
```

- *proof script does not depend on other theorems*
- proof script makes use of function ++ in module **StdList**
- proof script makes use of function **map** in module **StdList**
- *proof script does not make use of any imported constructors*

1.2 Sparkle's strength

The use of the intermedial language Core, instead of a more mathematically oriented specification language, is a major advantage of Sparkle over most other proof assistants. Programmers usually understand their programs very well, but for reasoning a good understanding of the *translated* program is required. This might be a problem if the specification language differs too much from the programming language. Three types of differences can be distinguished [5]:

Differences in semantics Understanding the translated program may be difficult because of unfamiliar concepts in the specification language. Furthermore, the understanding of the original program may not transfer to the translation. Fortunately, both CORE and CLEAN use a lazy term-graph rewriting system. The only difference is that overflow and rounding errors that occur in CLEAN are disregarded in CORE.

Differences in notational expressivity The notational expressivity of the specification language may be less than that of the programming language. Some CLEAN concepts have to be translated to simpler ones in CORE. CORE has all the basic concepts, but pattern matches are translated to case distinctions, overloading translates to dictionaries and concrete functions, and dot-dot expressions and list comprehensions are translated to functions. In SPARKLE, some, but not all, of these translations are reversed or hidden from the user.

Differences in syntax Differences in syntax are often not so serious. Nevertheless, they are unnecessary and should be avoided as much as possible. Therefore, where possible, SPARKLE uses CLEAN syntax for specifying the programs as well as the properties.

It is important that proving properties about functional programs is as easy as possible. Verifying properties can greatly increase the correctness and thus the security and robustness of a program, the importance of which is generally accepted. However, proving properties is very time consuming work and will often be considered not worth the effort. Even with the progress made by SPARKLE, it is only suitable for the most critical parts of a program. However, the easier it is to prove properties, the more it will be done and the more secure and robust programs will be. A worthy goal to pursue.

1.3 Sparkle's weakness

One of the differences in expressivity between CORE and CLEAN is that the latter supports overloading, also known as ad hoc polymorphism, and CORE does not. The term overloading is used for the specification of a group of equally named functions (instances), each operating on a different, possibly overlapping, range of types. These functions can depend on each other, allowing the overloaded use of the function name in its own definition.

It is important to note that these functions, at least in most cases, implement the same kind of operation. Therefore, all instances will have some properties in common. In fact, overloading is probably used because of these common properties. For example, all instances of the equality operator in example 1.3.1 are symmetric and transitive. It would be useful to be able to prove that these properties hold for an overloaded function, regardless of the concrete type of the parameters. Currently, this is not possible in SPARKLE.

Example 1.3.1 (type class)

This is an example type class `Eq`, of which the equality operator `(==)` is the only member. Type classes are explained in more detail in section 2.3.

```
class Eq a where
   (==) :: a a -> Bool

instance Eq Int where
   (==) :: Int Int -> Bool
   (==) x y = eqInt x y

instance Eq Char where
   (==) :: Char Char -> Bool
   (==) x y = eqChar x y

instance Eq [a] | Eq a where
   (==) :: [a] [a] -> Bool | Eq a
   (==) []     []     = True
   (==) []     [y:ys] = False
   (==) [x:xs] []     = False
   (==) [x:xs] [y:ys] = (x == y) && (xs == ys)
```

As briefly mentioned before, overloading is resolved in the translation from Clean to Core. When an overloaded function is used, based on the inferred type, a specific implementation is created from the available definitions and passed around as a dictionary. This way, the property has to be proven separately for each closed type for which the function is used, because only in these cases the overloading can be completely removed. This is very inconvenient and does not align with a basic idea of overloading, which is to treat groups of functions that perform equivalent operations as one. Furthermore, the programmer knows the program at the level that uses overloading, not at the level of dictionaries.

Example 1.3.2 (current situation)

Currently, it is possible to prove symmetry of equality for a specific type, for instance lists of integers, in Sparkle:

$\forall_{\mathtt{x}\in[\mathtt{Int}]}\forall_{\mathtt{y}\in[\mathtt{Int}]}[\mathtt{x == y} \Rightarrow \mathtt{y == x}]$

However, it is not possible to prove, or even specify, symmetry of equality in general:

$\forall_{\mathtt{a|Eq\ a}}\forall_{\mathtt{x}\in\mathtt{a}}\forall_{\mathtt{y:a}}[\mathtt{x == y} \Rightarrow \mathtt{y == x}]$

1.4 Class constrained properties

The problem noted in the previous section is that we would like to be able to prove properties in which an overloaded function is used, for all instances of that overloaded function at once. These properties are only welltyped if it is assured that there is an instance of these functions for the type required. In Clean, this is enforced by adding class constraints to the types. Example 1.4.1 shows how such properties might be defined.

Example 1.4.1 (example properties)

Several properties about overloaded expressions:

$\forall_{\texttt{a} \mid \texttt{Eq a}}$	$\forall_{\texttt{x}\in\texttt{a}}$	`x == x`
$\forall_{\texttt{a} \mid \texttt{Eq a}}$	$\forall_{\texttt{x,y,z}\in\texttt{a}}$	$\texttt{x == y} \wedge \texttt{y == z} \Rightarrow \texttt{x == z}$
$\forall_{\texttt{a} \mid \texttt{Eq a}, + \texttt{a}}$	$\forall_{\texttt{x}\in\texttt{a}}$	`x + x == x + x`
$\forall_{\texttt{a} \mid < \texttt{a}, - \texttt{a}, * \texttt{a}}$	$\forall_{\texttt{x,y}\in\texttt{a}}$	`(x - 1) * y <= x * y`
$\forall_{\texttt{a,b} \mid \texttt{Eq b}, \texttt{Coerce a b}}$	$\forall_{\texttt{x}\in\texttt{a}}$	`(coerce x) == (coerce x)`

Adding class constrains can influence if a property is true or not. Consider for example the property $\texttt{x <= y} \wedge \texttt{y <= z} \Rightarrow \texttt{x == y} \vee \texttt{y == z}$ from example 1.4.2.

Example 1.4.2 (a false property)

Consider the following property:

$\forall_{\texttt{a} \mid \texttt{Eq a}, < \texttt{a}}\forall_{\texttt{x,y}\in\texttt{a}}[\ \texttt{x <= y} \wedge \texttt{y <= z} \Rightarrow \texttt{x == y} \vee \texttt{y == z}\ ]$

We assume there are instances of `<=` and `==` for integers. If `x`, `y`, and `z` have values `1`, `2`, and `3` respectively, we have a counterexample for the property. Thus, the property is false.

This property is not true for all instances of `<=` and `==`. However, it is true for types that have only two values. Say `Flip` is a class with instances for exactly those types. By adding this class constraint to the property from example 1.4.3, it becomes true.

Example 1.4.3 (class constraints influence properties)

Say there is a `Flip` class with instances for binary types. The member function `flip` flips the value:

```
:: Coin = Heads | Tails

class Flip a where
   flip :: a -> a

instance Flip Coin where
   flip :: Coin -> Coin
   flip Heads = Tails
   flip Tails = Heads

instance Flip Bool where
   flip :: Bool -> Bool
   flip True  = False
   flip False = True
```

If the property from example 1.4.2 is constrained to this class, it is true.

$\forall_{\texttt{a} \mid \texttt{Eq a}, < \texttt{a}, \texttt{Flip a}}\forall_{\texttt{x,y}\in\texttt{a}}\ \texttt{x <= y} \wedge \texttt{y <= z} \Rightarrow \texttt{x == y} \vee \texttt{y == z}$

Hence, class constraints can influence the truth of a property.

When the `Flip` class is extended with an instance for integers, the property is no longer true.

```
instance Flip Int where
   flip :: Int -> Int
   flip x  = -x
```

This illustrates that properties need to be proven again when instances are added.

The essential addition to the properties caused by overloading are the class constraints. Overloaded functions cannot be applied without class constraints, because then it is unknown if there is an instance available. Class constraints allow overloaded function applications, but can also be specified when no overloaded functions are applied. Therefore, this class of properties will be called *class constrained properties.*

The problem in proving class constrained properties is that instance definitions may depend on each other. For example, the instance of equality for lists in example 1.3.1 depends on the instance for the list members. Hence, the instance for lists is only an equivalence relation if the instance for the list members is so as well. This requires some induction method that allows this hypothesis to be assumed. In general, we would like to be able to assume hypotheses for all *sensible* dependencies; the dependencies that preserve type correctness.

1.5 Research goal

The main problem noted in the previous sections, is that Sparkle cannot be used to prove class constrained properties. In this project, this problem will be solved by creating an extension of the proof assistant Sparkle that allows the user to conduct these proofs. It would be nice if this extension, besides solving the problem, stays close to Clean's syntax and semantics, at least to the user, since that is one of the major advantages of Sparkle over other proof assistants.

Research goal: Extend Sparkle with the ability to conduct proofs of class constrained properties.

Reaching this goal requires a number of interesting theoretical and practical issues to be solved. These issues can be formulated as three objectives.

Objective 1 (Specification): First of all, it should be investigated how class constrained properties can be specified in Sparkle and how they can be specified using Clean syntax. This will require an extension of the Core mathematical framework on which Sparkle is based.

Objective 2 (Reasoning): The second objective is to investigate how one can reason about overloaded expressions in order to prove the kind of properties arrived at by objective 1. The result will be the formalization of valid proof rules and tactics.

Objective 3 (Implementation): Lastly, we should allow the properties from objective 1 to be specified and the tactics from objective 2 to be applied in Sparkle. This involves changing and adding code to Sparkle; one of the larger Clean programs written (approximately 130.000 lines of code, including libraries).

These three objectives naturally lead to the research question that is answered in this thesis.

Research question: How can properties about overloaded expressions be specified and proven?

1.6 Outline

This thesis is structured as follows:

- Chapter 2 reviews several ways to implement overloading, most concerning type classes, and presents the approach taken in Clean and, very interesting, in the proof assistant Isabelle.
- Chapter 3 contains the most important contributions. It extends the Core framework to be able to handle class constrained properties. Definitions for type classes are added to the specification language, it is specified how class constrained properties can be defined in the logic language, and, to be able to prove class constrained properties, special proof rules and tactics are presented.
- Chapter 4 discusses the interesting parts of the implementation of the specification of class constrained properties and the proof rules in Sparkle.
- Chapter 5 presents the conclusion and provides suggestions for further research.
- Appendix A introduces the mathematical notation and concepts used in this thesis.
- Appendix B introduces Sparkle's foundations, the Core mathematical framework, on which the formalization in this thesis is built.
- Appendix C provides proofs and proof sketches for the proof rules and tactics given in section 3.3.
- Appendix D contains a summary for laymen that explains in Dutch, and in popular style, what this thesis is about and why this work is useful.
- Appendix E contains the paper written about this work submitted to, and presented at, the Fifth Symposium on Trends in Functional Programming 2004.

Overloading 2

In the past, there have been several approaches to structure overloading. Whereas the first attempts were not very flexible or elegant, let alone considered standard, the situation improved with the introduction of type classes [19]. Nowadays, type classes have become a common means of abstraction for overloading in functional programming languages, such as Haskell and Clean. In this chapter, several approaches to overloading, most concerning type classes, are presented. Naturally, much of the literature on this topic discusses theoretical issues such as decidability of type inference and coherence of semantics. Since we are working with a system for which these problems have already been solved, we will mostly skip these subjects and look at overloading from the perspective of the programmer.

Section 2.1 explains the difference between parametric polymorphism and overloading. Then, some (early) approaches to overloading are reviewed in section 2.2. Section 2.3 introduces the system of type classes. After that, several extensions of type classes are presented in sections 2.4, 2.5, 2.6, and 2.7. In section 2.8, the translation of type classes to a language without type classes is discussed. In conclusion, sections 2.9 and 2.10 present the use of overloading and type classes in Clean and Isabelle respectively.

2.1 Polymorphism

Overloading is a form of polymorphism. Polymorphism means that a single symbol is used to denote operations on values of various types. For example, in mathematics the equality symbol $=$ can be used to denote equality between, for instance, natural, real, and rational numbers. This is done for reasons of convenience. Less symbols are required, which means that the symbols can be kept simple. Moreover, in general the same symbol is used for semantically equivalent operations. Here, polymorphism is considered in the context of programming languages, where two main forms of polymorphism can be distinguished: *parametric* and *ad hoc* polymorphism (overloading) [16].

Parametric polymorphism is a very regular form of polymorphism: a single function definition is used to describe a function that operates on various types. To achieve this, the function type contains parameters. The function is defined for all types that are a substitution of the function type. Unfortunately, this implies that in the function body no type specific operations can be used on the values that have an unknown type. Parametric polymorphism is illustrated by the `length` function in example 2.1.1.

Example 2.1.1 (parametric polymorphism)

The `length` function is defined by:

```
length :: [a] -> Int
length []     = 0
length [x:xs] = 1 + length xs
```

The `length` function computes the length of a list by counting the elements. This operation is independent of the type of the list elements because no special operations are performed on these elements; they are just counted. Therefore, the `length` function can be specified for all possible element types by a single definition using a parameter for the type of the list elements. For example:

```
length [1, 2, 3] = 3
length [('a', 'b'), ('c', 'd')] = 2
```

Overloading, or ad hoc polymorphism, as implied by its name, is much less systematic than parametric polymorphism. Instead of using a single function definition, several definitions are provided for the same symbol, each operating on a different set of types. This way, type specific operations can be used, as shown in example 2.1.2. Note that, although ad hoc, the definitions are semantically related in general. For example, all definitions of the equality operator `==` should denote an equivalence relation.

Example 2.1.2 (ad hoc polymorphism)

Equality on integers and characters can be defined by (`eqInt` and `eqChar` are predefined):

```
(==) :: Int Int -> Bool
(==) x y = eqInt x y

(==) :: Char Char -> Bool
(==) x y = eqChar x y
```

This way, the symbol `==` can be used for comparing both integers and characters. In more complex systems, equality on lists can be defined by:

```
(==) :: [a] [a] -> Bool
(==) []     []     = True
(==) [x:xs] []     = False
(==) []     [y:ys] = False
(==) [x:xs] [y:ys] = x == y && xs == ys
```

This definition assumes that `==` is defined for the list elements as well. Hence, this defines `==` for integers, characters, lists of integers, lists of characters, lists of lists of integers, and so on.

Because this research concerns type classes, which are a way to structure overloading, the following discussion only contains approaches to overloading.

2.2 Approaches to overloading

Several ways to support overloading have been investigated. Possibly the simplest approach is to only overload the basic (predefined) operations, such as addition, multiplication, and equality. Then, expressions like `2 == 2` (true) and `'a' == 'c'` (false) are defined, but functions defined in terms of basic operations cannot be overloaded.

Example 2.2.1 (limitations of basic overloading)
If only the basic operations are overloaded, it is *not* possible to define:

```
isMember []     x = False
isMember [y:ys] x = x == y || isMember x ys
```

such that one can write:

```
isMember [1,2,3,5,7] 2
isMember ['a', 'b', 'c'] 'd'
```

This is not a very general approach, since it does not use the fact that the definition of `isMember` can be seen as the definition of two overloaded functions, one for the type `Int` and one for the type `Char`, by using the definitions of `==` for `Int` and `Char` respectively. However, when functions as `isMember` are considered overloaded as well, the number of overloaded functions increases exponentially with the number of parameters (example 2.2.2).

Example 2.2.2 (exponential increase)
Consider the following function definition:

```
doubles x y z = (x + x, y + y, z + z)
```

If `+` is overloaded for types `Int` and `Real`, `doubles` can be considered overloaded as well. In that case, there would be eight overloaded versions of `doubles`, because each of the three parameters can have either type `Int` or `Real`. In general, the increase is exponential in the number of parameters.

A more general approach is to allow a limited form of polymorphism in which type variables can be restricted to a subset of types, for example types for which a basic operation, say `(==)`, is defined.

Example 2.2.3 (limited polymorphism)
Let `a(==)` be a type variable that ranges only over types for which equality is implemented. Then `==` has type:

```
(==) :: a(==) a(==) -> Bool
```

and the `isMember` function from example 2.2.1 is of type:

```
isMember :: [a(==)] a(==) -> Bool
```

Hence, the `isMember` function can be applied to all the basic types for which equality is defined.

Without specific support from the programming language, overloading can be simulated using dictionaries; records in which the implementations of overloaded functions can be looked up. Unfortunately, this is not very efficient and syntactically not very nice, because the dictionaries have to be passed around as parameters.

Example 2.2.4 (simulated overloading)

Given the add and subtract functions for types `Int` (`addint` and `subint` respectively) and `Real` (`addreal` and `subreal` respectively) the following dictionaries are created:

```
:: Arith a = { add :: a a -> a, sub :: a a -> a }

ArithInt  = { add = addint , sub = subint  }
ArithReal = { add = addreal, sub = subreal }
```

The exponential increase shown in example 2.2.2 is prevented by defining the `doubles` function as:

```
doubles ::  (Arith a) (Arith b) (Arith c) a b c -> a b c
doubles ax ay az x y z = (ax.add x x, ay.add y y, az.add z z)
```

Of the presented attempts to structure overloading, none is really satisfying. Ironically, dictionaries, which do not require special support from the programming language, provide the most flexibility. Unfortunately, they are also syntactically the least appealing. In the next section, a solution is described that is flexible and has a nice syntax as well.

2.3 Type classes

To add more structure to ad hoc polymorphism, Wadler and Blott [19] proposed a new technique, called *type classes*. Type classes are used to group and name a set of overloaded symbols, which are called *class members*. For each member the overloaded name and overloaded type are specified in the *type class definition*. An *instance* of a class is a concrete implementation of the members for a certain type. This type does not have to be a basic type, but may be an abstract type and contain variables as well. The range of these *type variables* can be limited to types for which there is an instance of a certain class.

Type classes require an extension of the type system. Wadler and Blott do this by adding predicates over the type variables that require that there is an instance of a certain class for the type the variable stands for. Example 2.3.1 shows an example of a type class (predicates are written at the end of the type).

Example 2.3.1 (type classes)

The class `Eq`, of which the equality operator `==` is the only member, is defined by:

```
class Eq a where
   (==) :: a a -> Bool
```

Instances for integers, characters, and lists are defined by (`eqInt` and `eqChar` are predefined):

```
  instance Eq Int where
     (==) :: Int Int -> Bool
     (==) x y = eqInt x y

  instance Eq Char where
     (==) :: Char Char -> Bool
     (==) x y = eqChar x y

  instance Eq [a] | Eq a where
     (==) :: [a] [a] -> Bool | Eq a
     (==) []     []     = True
     (==) []     [y:ys] = False
     (==) [x:xs] []     = False
     (==) [x:xs] [y:ys] = (x == y) && (xs == ys)
```

The instance for lists is special, because it is defined in terms of a class member. It is only defined for lists if an instance of equality exists for the type of the list elements. Because of this requirement, the equality operator can be safely used to compare the elements in the function body. This way, the equality operator is defined for types `Int`, `Char`, `[Int]`, `[Char]`, `[[Int]]`, `[[Char]]`, ...

Overloaded functions can be defined in terms of class members or other overloaded functions. Consider for example the `isMember` function (example 2.3.2), which is defined for all types `a` as long as `==` is defined for `a`.

Example 2.3.2 (overloaded function)

The `isMember` function is defined by:

```
  isMember :: [a] a -> Bool | Eq a
  isMember []     a = False
  isMember [x:xs] y = x == y || isMember xs y
```

It is defined for all types `a` for which an instance of `Eq` is defined.

Classes can also be defined in terms of other classes, which are called *subclasses*. New members can be defined, but this is not required. This way, for example, a collection of classes can be grouped under a single name. Such a class, that only groups other classes, is called a *compound* class. Of course, cyclic class dependencies are not allowed; a class cannot be a subclass of itself.

Example 2.3.3 (classes defined in terms of classes)

The `Arith` class groups the `+` and `-` class:

```
  class Arith a | +, - a
```

An essential property of this system is that it is a generalization of the Hindley-Milner polymorphic type system and that all type declarations can be inferred. However, to determine which instance of an overloaded function to use, the most specific instance definition has to be determined. Unfortunately, sometimes this is not possible. Such an overloaded expression is called *ambiguously overloaded*. Often, this can be solved by splitting the expression up and explicitly typing the parts (example 2.3.4).

Example 2.3.4 (ambiguous overloading)

Assume there are instances of `Read` and `Write` for types `Int` and `Bool`:

```
class Read  a where read  :: a -> String
class Write a where write :: String -> a

instance Read  Int  where read  = readInt
instance Read  Bool where read  = readBool
instance Write Int  where write = writeInt
instance Write Bool where write = writeBool
```

Then, the expression:

```
f :: String -> String
f x = read (write x)
```

can either mean:

```
f x :: String -> String
f x = read (write2 x)
where
   write2 :: String -> Int
   write2 x = write x
```

or:

```
f x :: String -> String
f x = read (write2 x)
where
   write2 :: String -> Bool
   write2 x = write x
```

When Wadler and Blott introduced type classes, they already mentioned that it would be natural to specify properties that each instance must satisfy. Their idea was to add the properties to the class definition as assertions. These assertions could then be verified for each instance, and used in proofs. They used informal comments to illustrate their idea. More formally, the assertion that == is a symmetric and transitive relation might be specified as in example 2.3.5. It is clear that these class assertions can be expressed as class constrained properties as well.

Example 2.3.5 (class assertions)

In the `Eq` class, the equality function == is defined. This should be a reflexive, symmetric and transitive relation, which can be enforced by adding assertions to the class definition:

```
class Eq a where
   (==) :: a a -> Bool
```

$$\begin{array}{ll} \forall_{x \in a} & \texttt{x == x} \\ \forall_{x,y \in a} & \texttt{x == y} \Rightarrow \texttt{y == x} \\ \forall_{x,y,z \in a} & \texttt{x == y} \wedge \texttt{y == z} \Rightarrow \texttt{x == z} \end{array}$$

2.4 Constructor classes

In 1995, Jones proposed a natural generalization of type classes, called *constructor classes* [9]. It was developed because the system of type classes could not always sufficiently capture the structure of a function's type. Consider for example the `map` function, that applies a function to all elements of a list.

Example 2.4.1 (map function)

The `map` function is defined by:

```
map :: (a -> b) [a] -> [b]
map f []     = []
map f [x:xs] = [f x: map f xs]
```

The `map` function can also be implemented for trees, in which case its type would be `map :: (a -> b) (Tree a) -> (Tree b)`. Hence, the general type for the `map` function is `map :: (a -> b) (c a) -> (c b)` where `c` is a constructor. This structure cannot be expressed using type classes, because class variables stand for types whereas `c` is a constructor. The straightforward solution is to allow class variables to stand for both variables and constructors.

Example 2.4.2 (constructor classes)

The `functor` class and instances are defined by:

```
:: Tree a = Leaf a | Node (Tree a) (Tree a)

class Functor c where
   map :: (a -> b) (c a) -> (c b)

instance Functor [] where
   map :: (a -> b) [a] -> [b]
   map f []     = []
   map f [x:xs] = [f x: map f xs]

instance Functor Tree where
   map :: (a -> b) (Tree a) -> (Tree b)
   map f (Leaf x)   = Leaf (f x)
   map f (Node l r) = Node (map f l) (map f r)
```

To ensure that all class members have a well-formed type, the notion of *kind* is introduced. The set of kinds is inductively defined where the kind of all types is denoted by $*$ and the kind of a constructor that expects something of kind k_1 and returns something of kind k_2 is written as $k_1 \rightarrow k_2$. Hence, type classes are a special case of constructor classes, where the class variable has kind $*$. In the case of the `Functor` class, `c` can only stand for a constructor that expects a type as an argument. Since `c` also returns a type, it has kind $* \rightarrow *$.

Example 2.4.3 (examples of kinds)

Constructor/type	Kind
`Int, Char, [Float]`	$*$
`Tree, []`	$* \rightarrow *$
`->`	$* \rightarrow (* \rightarrow *)$

2.5 Multi-parameter type classes

One of the most natural extensions of type classes is to allow a class to have more than one parameter [11]. This extension has various useful applications, for example coercion, isomorphisms, and collections (example 2.5.1).

Example 2.5.1 (collections)

The `Collection` class that has two type class variables is defined by:

```
class Collection c a where
   empty  :: (c a)
   insert :: a (c a) -> (c a)
   member :: a (c a) -> Bool

instance Collection [] a where
   empty  = []
   insert = insertList
   member = memberList

instance Collection TreeSet a where
   empty  = emptyTreeSet
   insert = insertTreeSet
   member = memberTreeSet
```

Unfortunately, this example of collections is a bit restrictive. For example, it does not allow a collection to be represented by a characteristic function. Allowing this, however, makes the definition too general, as shown in the next section. Fortunately, the next section also contains a solution.

2.6 Functional dependencies

One of the most recently proposed extensions of type classes are functional dependencies [10]. Functional dependencies are a way to restrict the number of possible instance definitions of a class. They can be used to indicate that an instantiation of a class parameter uniquely determines another. For example, consider an alternative definition of the `Collection` class (example 2.6.1).

Example 2.6.1 (more general collections)

The `Collection` class can be defined by:

```
class Collection ca a | ca ⇝ a where
   empty  :: ca
   insert :: a ca -> ca
   member :: a ca -> Bool

instance Collection (a -> Bool) Bool where
   empty  = emptyFunc
   insert = insertFunc
   member = memberFunc
```

Without the `ca` $\rightsquigarrow$ `a`, the class definition in example 2.6.1 would be too general: the type of `empty` would be `ca | Collection ca a` where the a occurs only in the class constraint. This is ambiguous, because there could be more than one instance definition applicable (different instantiations of `a`). Fortunately, `ca` $\rightsquigarrow$ `a`, which means that `a` is uniquely determined by `ca`, excludes exactly that possibility. This allows a more general definition of the `Collections` class.

2.7 Parametric type classes

Another way to restrict types, parametric type classes, was presented by Chen, Hudak, and Odersky [3]. To make the intended relation between the two class parameters explicit, the class parameters are split into two groups: the placeholder variables and the parameter variables. No two instance definitions of a class may have the same instantiation of the placeholder variables, hence, the types of the placeholder variables uniquely identify which instance to use. By implementing collections using one placeholder variable and one parameter variable, essentially the same restriction can be enforced as in example 2.6.1.

2.8 Translation of type classes

Wadler and Blott provided rules that translate a program that uses type classes into an equivalent one that does not [19]. This translation can be straightforwardly extended for the presented extensions. Essentially, the translation uses dictionaries in the same way as the simulated overloading presented in section 2.2. For every class instance, a dictionary can be created using the instance definitions. Dictionaries are explicitly passed around to both overloaded functions and instance functions.

Example 2.8.1 (translation of type classes)

The `Eq` class from example 2.3.1 and instances are translated to:

```
:: eqDict a = { == :: a a -> Bool }

eqDictInt :: eqDict Int
eqDictInt = { == = eqInt }

eqDictReal :: eqDict Real
eqDictReal = { == = eqReal }

eqDictList :: (eqDict a) -> eqDict [a]
eqDictList d = { == = eqList d }

eqList :: (eqDict a) [a] [a] -> Bool
eqList d []     []     = True
eqList d [x:xs] []     = False
eqList d []     [y:ys] = False
eqList d [x:xs] [y:ys] = d.== x y && eqList d xs ys
```

The `isMember` function (example 2.3.2) is translated to:

```
isMember' :: (eqDict a) [a] a -> Bool
isMember' d []     y = False
isMember' d [x:xs] y = d.== x y || isMember d xs y
```

In the translation of the `Eq` class, a dictionary is created that contains only one entry, the `==` function. `eqDictInt` and `eqDictChar` are the dictionaries for types `Int` and `Char` respectively. They use the predefined functions `eqInt` and `eqChar`. The function `eqList` uses the dictionary for type `a` to create a dictionary for lists of that type (`[a]`).

Example 2.8.2 (translation of expressions)

Given the definitions and translations of `==` and `isMember` in examples 2.3.1, 2.3.2, and 2.8.1, the expressions:

```
3.4 == 12.45
[[1,2],[3,4],[5]] == [[1],[2]]
isMember [1.2,3.4,4.5] 1.2
isMember [[1,2],[3]] [1]
```

are respectively translated to:

```
eqDictReal.== 3.4 12.45
eqList (eqDictList eqDictInt) [[1,2],[3,4],[5]] [[1],[2]]
isMember' eqDictReal [1.2,3.4,4.5] 1.2
isMember' (eqDictList eqDictInt) [[1,2],[3]] [1]
```

The application of `==` is translated to an application of `eqInt`, `eqChar`, or `eqList`. In the case of `eqList`, the appropriate dictionary is created and passed as a parameter.

From these examples, it seems that programs that use type classes can always be translated to an equivalent program that is typeable in the Hindley-Milner system. This is not the case however: when the dictionary contains a polymorphic function, in some cases, polymorphic types of rank ≥ 2 are required, which are not part of the Hindley-Milner type system[1].

Example 2.8.3 (untypeable translation)

Consider the `Copy` class that contains the polymorphic member `f`:

```
class Copy a where
   f :: a b -> b

instance Copy Bool where
   f :: Bool b -> b
   f False y = abort
   f True  y = y
```

The translation is perfectly typeable:

```
:: dictCopy a = { f :: a b -> b }

fbool :: Int b -> b
fbool False y = abort
fbool True  y = y
```

[1] Clean, but not Sparkle, supports rank 2 and will support rank n polymorphism.

However, problems arise when `f` is used in a function:

```
g :: (Int, Char)
g = (f True 2, f True 'a')
```

The function `g` is translated to:

```
g d = (d.f True 2, d.f True 'a')
```

Now `g` has type `g :: { f ::` $\forall_b$ `Bool b -> b} -> (Int, Char)`. The universal quantifier is required, because otherwise `b` has to be of the same type for both applications of `d.f`. This type requires rank 2 polymorphism, which is not part of the Hindley-Milner type system.

In practice this is not a problem, because the original program can be typed and type checked. The translation preserves type correctness and hence does not have to be typed again.

2.9 Overloading in Clean

The examples of type classes were presented in Clean syntax. However, not all of the extensions are available in Clean. The implementation of overloading in Clean is fully called *multi parameter type constructor classes*. This is essentially a combination of the constructor and multi-parameter extensions. This section discusses a number of specific details, some of which are illustrated by examples: overlapping instances, non-flat instance types, derived members, and a syntactic shorthand.

In Clean, overlapping instance definitions are allowed. In case two or more definitions are applicable, the compiler will always choose the most specific one. Especially, generic instances that operate on all types can be defined. Of course, identical instances are not allowed. In fact, instance definitions where all outermost constructors are equal (head-equal) are not allowed.

Example 2.9.1 (overlapping instances)

Two overlapping instances of the `Coerce` class can be defined:

```
class Coerce a b where coerce :: a -> b

instance Coerce Int Bool where ...
instance Coerce Int Real where ...
```

Even generic instances are possible:

```
instance Coerce a b where ...
instance Eq a where ...
```

However, two head-equal instances *cannot* be defined:

```
instance Eq [Int] where ...
instance Eq [Real] where ...
```

In Clean, the type pattern in an instance definition is not restricted. However, a common limitation is to only allow flat types. A flat type is a constructor followed by type variables.

Currently, Clean allows type variables to occur more than once within the same class constraint. Unfortunately, this can cause cyclic dependencies where an instance depends on itself. In the current implementation of the compiler, this is not noted and results in a stack overflow. It might be best to add a restriction to the language to exclude these situations.

Example 2.9.2 (cyclic dependencies)

In Clean, the following classes and instances can be defined:

```
class cl a b where f :: a b -> Bool

instance cl Int Int where ...
instance cl [a] b | cl b b where ...
instance cl (Tree a) (Tree b) | cl a b where ...
```

Unfortunately, the second instance definition causes cyclic dependencies: the instance for `[Int] [Int]` depends on itself.

A class definition can contain members that are expressed in terms of other members. For example, `<>` is always equal to `not ==`. In Clean, this is expressed by macros (example 2.9.3).

Example 2.9.3 (derived members)

In the `Eq` class, `<>` can be defined in terms of `==`:

```
class Eq a where
    (==) :: a a -> Bool

    (<>) :: a a -> Bool | Eq a
    (<>) x y :== not ((==) x y)
```

Finally, a shorthand is provided for defining classes that contain only one member. Then, the class name is used for the member as well.

Example 2.9.4 (shorthand)

The class containing only the `==` function can be defined by:

```
class (==) a :: a a -> Bool
```

Although there are some differences, the Clean compiler essentially translates type constructor classes as presented in section 2.8. However, when possible, Clean generates specialized functions that are more efficient. In the next few chapters, derived members and specialized functions are left out of the discussion. They show up again in chapter 4, where the implementation is considered.

2.10 Overloading in Isabelle

Another environment that supports overloading is Isabelle [15]. Isabelle is a generic proof assistant developed by Lawrence C. Paulson and Tobias Nipkow that can, among other calculi, work with Higher-Order Logic (a version of Gordon's HOL [6]). Isabelle's version of HOL includes a Haskell like type system with ordered-sorted type classes [14] and an extension called axiomatic type classes [20]. Using this, it can be used to prove properties containing overloaded function applications. This is very interesting, because we might be able to use a method similar to Isabelle's for our project.

Since 1991, Isabelle's type system allows *overloaded definitions*. Overloaded definitions can be specified by giving a function a polymorphic type and providing instances.

Example 2.10.1 (overloaded definitions in Isabelle)

Consider the following overloaded definition of $\leq$:

`const` $\leq\ :: \alpha \rightarrow \alpha \rightarrow \mathit{prop}$

`defs` $x_{nat} \leq y_{nat} \equiv \mathit{nat\_le}$

`defs` $x_{\alpha\times\beta} \leq y_{\alpha\times\beta} \equiv \mathit{fst}\ \ x_{\alpha\times\beta} \leq \mathit{fst}\ \ y_{\alpha\times\beta} \wedge \mathit{snd}\ \ x_{\alpha\times\beta} \leq \mathit{snd}\ \ y_{\alpha\times\beta}$

This defines the type of the $\leq$ operator and two instances of it; one for natural numbers and one for tuples.

Overloaded definitions are allowed as long as no two definitions have overlapping instances. In the right hand side of the definition, the operator itself can be used as long as it is for a structurally smaller type (primitive recursion over types). Note that only one parameter type can be used and only flat instance types are supported.

On top of this, a system called *axiomatic type classes* is defined. Here, type classes are classes of types that meet certain properties instead of types for which certain operations are defined (this would be impossible; in HOL it cannot be expressed if objects are declared or meaningful).

Example 2.10.2 (axiomatic type classes in Isabelle)

The class *ord* contains only types that have an ordering relation defined:

`class` *ord*

reflexive	$x_\alpha \leq x_\alpha$
transitive	$x_\alpha \leq y_\alpha \wedge y_\alpha \leq z_\alpha \Rightarrow x_\alpha \leq z_\alpha$
antisymmetric	$x_\alpha \leq y_\alpha \wedge y_\alpha \leq x_\alpha \Rightarrow x_\alpha = y_\alpha$

The class *ord* can be used as a type predicate that states that $\leq$ is an ordering relation. Concrete instances of the class *ord* are required to have the operator specified such that the properties are derivable.

Example 2.10.3 (instance definitions in Isabelle)

Consider the following instance definitions of *ord* for natural numbers and tuples:

`instance` $\mathit{nat} :: \mathit{ord}$

`instance` $\times :: (\mathit{ord}, \mathit{ord})\ \ \mathit{ord}$

At the definition, the class axioms have to be proven for the instances. The property may be assumed for structurally smaller types that are in the same class. Again, definitions may have only a single parameter type which has a flat type pattern and may not overlap.

Subclasses are supported for both classes and instances.

Example 2.10.4 (subclasses in Isabelle)

A class can be defined as a subclass of other classes, meaning it inherits the axioms from the parent classes:

`class` $c_1 \preceq c_2, c_3$

An instance is a subclass of another instance if all instances of the class can be proven to be an instance of the parent class:

`instance` $c_1 \preceq c2$

Of course, classes can be used as type constraints by requiring that a type is an instance of that class.

In summary, ignoring the differences in use of type classes and other minor details, Isabelle supports the notions of overloading and type classes for single parameter classes where instance definitions are not allowed to overlap and use flat type patterns. Constructor classes are not supported. Proofs can use assumptions for structurally smaller types, hence a form of structural induction on types is used as a proof rule. All of this is achieved by adding a partially ordered layer of ”sorts” on top of the types. In section 3.3.7, the approach taken in Isabelle is compared with our approach.

Specifying and proving class constrained properties 3

In this chapter, class constrained properties are formalized by extending the CORE framework (appendix B). Programs and properties are considered at the CORE level where type classes have been translated to dictionaries. This is the most straightforward approach because it requires only few extensions of the existing CORE framework. Of course, special proof rules and tactics for class constrained properties have to be defined.

The definitions presented here are CORE specific and take all of CLEAN's details and extensions into account. For a less detailed and more general version, see the paper included in appendix E.

In section 3.1, the CORE programming language is extended with class and instance definitions and the relevant parts of the translation from CLEAN to CORE programs are explained. Section 3.2 explains how class constrained properties can be specified in CORE's logic language. The proof language is not altered. Special proof rules and tactics for class constrained properties are defined in section 3.3.

3.1 Programming language definitions

In CORE, overloading is considered explicitly in translated form. In this section this translation is explained and class and instance definitions are added to the CORE framework to allow reasoning with class constraints. First, the translation of overloaded function and member applications is explained in section 3.1.1. Then, sections 3.1.2, 3.1.3, and 3.1.4 respectively add class constrained types, class and instance definitions, and programs with type classes to the CORE programming language. The definition of the set of instances of a class and the creation of dictionaries are defined in sections 3.1.5 to 3.1.8.

3.1.1 Overloaded functions and expressions

As introduced in section 2.8, overloading is made explicit using dictionaries. Dictionaries are records that contain concrete implementations of the class members for a certain instance. For every class, there is a specific record type for its dictionaries. Before the translation is explained further, these dictionary types are formally defined.

Example 3.1.1 (a dictionary type and value)
Consider the `PlusMin` class defined by:

```
class PlusMin a where
   plus :: a a -> a
   min  :: a a -> a
```

The translation creates a specific dictionary type:

```
:: plusmindict a = { plus :: a a -> a, min :: a a -> a }
```

A possible value (for an instance for integers) is:

```
{ plus = +int, min = -int }
```

Class symbols are used as a reference to a class. In most examples, the name of the class will be used as its class symbol. The arity of a class symbol is the number of parameters of the class that it refers to.

Definition 3.1.2 (class symbols)
$\mathcal{S}_{Cl}$ is the set of class symbols.
The function $Arity :: \mathcal{S}_{Cl} \to \mathbb{N}$ returns the arity of the class symbol.

In Core, record types are just algebraic types and record values are represented by a constructor followed by the record field values. The functions $DictType_\psi$ and $DictConstr_\psi$ respectively return the type and constructor of the dictionary for a given class. The construction of dictionary values is treated in section 3.1.8.

Assumption 3.1.3 (dictionary type)
The function $DictType_\psi :: \mathcal{S}_{Cl} \to \mathcal{S}_a$ is defined such that $DictType_\psi(c)$ returns the algebraic type symbol of the dictionary type of class c. $Arity(DictType_\psi(c))$ gives the number of class parameters.

Assumption 3.1.4 (dictionary constructor)
The function $DictConstr_\psi :: \mathcal{S}_{Cl} \to \mathcal{S}_c$ is defined such that $DictConstr_\psi(c)$ returns the constructor of the dictionary of class c and $Arity(DictConstr_\psi(c))$ is the number of subclasses plus the number of class members.

Overloaded functions, which would have a qualified type in Clean, are translated to functions that have a normal type in Core. For every class constraint in the type of the original function, the translated function expects a dictionary for the correct instance as a parameter. Hence, an overloaded function of class constrained type $\langle \vec{\alpha}_1 :: c_1, \ldots, \vec{\alpha}_n :: c_n \rangle \Rightarrow \tau \to \sigma$ is represented in Core by a function of type $(DictType(c_1)\ \vec{\alpha}_1) \to \ldots \to (DictType(c_n)\ \vec{\alpha}_n) \to \tau \to \sigma$.

In expressions the member functions are applied by selecting them from the dictionary. This dictionary can be a function argument or, when the required instance is known at compile time, created on the spot. The member definitions are treated as ordinary (overloaded) functions and given a unique name. A formalization of these translations is provided elsewhere [19].

Example 3.1.5 (translation of an overloaded function)

The `isMember` function is an overloaded function:

```
isMember :: [a] a -> Bool | Eq a
isMember []     a = False
isMember [x:xs] y = x == y || isMember xs y
```

The translation uses a dictionary for the equality operator (`==`):

```
:: eqDict a = { (==) :: a a -> Bool }

isMember' :: (eqDict a) [a] a -> Bool
isMember' d []     y = False
isMember' d [x:xs] y = d.== x y || isMember d xs y
```

3.1.2 Class constrained types

Representing type classes requires an extension of the set of types with class constraints (see example 3.1.11). Qualified types, introduced by Jones mainly for this purpose [8], will be used for this. A qualified type $\vec{\pi} \Rightarrow \vec{\sigma}$ only denotes the instances of type pattern $\vec{\sigma}$ that satisfy predicates $\vec{\pi}$. Here, the predicates are class constraints that state that the type is an instance of a certain class.

The set of types $\mathcal{T}$ is defined in the Core framework (definition B.1.7). As a convention, type variables are denoted by α and β, b denotes a basic type, and τ, σ and μ are used to denote any type.

A class constraint $\vec{\tau} :: c$ states that there is an instance of class c for type pattern $\vec{\tau}$. Class constraints of which the type pattern consists of type variables only, are called *simple class constraints*.

Definition 3.1.6 (class constraints)
The set of *class constraints* $\mathcal{P}$ is defined by:
$\mathcal{P} = \{\vec{\tau} :: c \mid \vec{\tau} \in \langle\mathcal{T}\rangle, c \in \mathcal{S}_{Cl}, \mathit{Arity}(c) = |\vec{\tau}|\}$

Definition 3.1.7 (simple class constraints)
The set of simple class constraints $\mathcal{P}_s \subset \mathcal{P}$ is defined by:
$\mathcal{P}_s = \{\vec{\alpha} :: c \mid \vec{\alpha} \in \langle\mathcal{T}\rangle, (\vec{\alpha} :: c) \in \mathcal{P}\}$

Class constrained types are constructed by adding the class constraints, also called *context*, to the types. The type variables in the class constraints must all occur in the type they are added to.

Definition 3.1.8 (class constrained types)
The set of class constrained types $\mathcal{T}_q$ is defined by:
$\mathcal{T}_q = \{\vec{\pi} \Rightarrow \tau \mid \vec{\pi} \in \langle\mathcal{P}\rangle, \tau \in \mathcal{T}, TV(\vec{\pi}) \subseteq TV(\tau)\}$
Instead of $\langle\rangle \Rightarrow \tau$ we will simply write τ.

The function TV can be straightforwardly extended to return the free variables in (sequences of) predicates and class constrained types. The functions *Type* and *Context* are used to respectively retrieve the type and the context of a class constrained type.

Definition 3.1.9 (type of a qualified type)
The function $\mathit{Type} :: \mathcal{T}_q \rightarrow \mathcal{T}$ is defined by:
$\mathit{Type}(\vec{\pi} \Rightarrow \tau) = \tau$

Definition 3.1.10 (context of a qualified type)
The function $Context :: \mathcal{T}_q \rightarrow \langle \mathcal{P} \rangle$ is defined by:
$Context(\vec{\pi} \Rightarrow \tau) = \vec{\pi}$

Class constrained types can occur in function and class definitions. The untranslated `isMember` function, for example, has a class constrained type.

Example 3.1.11 (`isMember` function type)
The `isMember` function is defined by:

```
isMember :: [a] a -> Bool | Eq a
isMember []     a = False
isMember [x:xs] y = x == y || isMember xs y
```

The formal class constrained type of the `isMember` function is:
$(\mathsf{a} :: \mathsf{Eq}) \Rightarrow$ `([a] a -> Bool)`

3.1.3 Class and instance definitions

In this section, class and instance definitions are added to Core. Although Core uses dictionaries instead of classes, these definitions are required to create the dictionaries. Special syntactical features, such as derived members, do not require explicit formalization.

Member symbols are the symbols for the class members. They cannot be used directly in expressions, but occur in the class and instance definition.

Definition 3.1.12 (member symbols)
$\mathcal{S}_m$ is the set of member symbols.

In Clean, a class definition consists of a class name, one or more class type variables, an optional class context, and a number of member definitions. These member functions consist of a function name and a type. This is formalized by the set of *class definitions*.

Definition 3.1.13 (class definitions)
The set of class definitions $\mathcal{C}_d$ is defined by:

$$\mathcal{C}_d = \{\mathsf{class}\ \vec{\pi} \Rightarrow \pi'\ \mathsf{where}\ \langle (m_1 :: \tau_1), \ldots, (m_n :: \tau_n) \rangle \mid \vec{\pi} \in \langle \mathcal{P}_s \rangle, \pi' \in \mathcal{P}_s, \tau_i \in \mathcal{T}, m_i \in \mathcal{S}_m, TV(\vec{\pi}) \subseteq TV(\pi')\}$$

Definition 3.1.14 (class context)
The function $Context :: \mathcal{C}_d \rightarrow \langle \mathcal{P} \rangle$ is defined by:
$Context(\mathsf{class}\ \vec{\pi} \Rightarrow \pi'\ \mathsf{where}\ \langle (m_1 :: \tau_1), \ldots, (m_n :: \tau_n) \rangle) = \vec{\pi}$

Definition 3.1.15 (class members)
The function $Members :: \mathcal{C}_d \rightarrow \langle \mathcal{S}_m \rangle$ is defined by:
$Members(\mathsf{class}\ \vec{\pi} \Rightarrow \pi'\ \mathsf{where}\ \langle (m_1 :: \tau_1), \ldots, (m_n :: \tau_n) \rangle) = \langle m_1, \ldots, m_n \rangle$

Example 3.1.16 (example class definition)
Consider this definition of the `Eq` class:

```
class Eq a where
    (==) :: a a -> Bool
```

The corresponding Core class definition is:
$\mathsf{class}\ (\mathsf{a} :: \mathsf{Eq})\ \mathsf{where}$ `(==)` $::$ `a a` $\rightarrow$ `Bool`

An instance definition consists of a class name, an instance head, an optional class context, and all member definitions. The member definitions consist of the type declaration and the function body. This is formalized by the set of *instance definitions.* Because the concrete instance functions are treated as normal (overloaded) functions in CORE, with uniquely created symbols, only these symbols are required in the instance definition; the implementations are already included in the program.

Definition 3.1.17 (instance definitions)
The set of instance definitions $\mathcal{I}_d$ is defined by:

$$\mathcal{I}_d = \{\textsf{instance } \vec{\pi} \Rightarrow \pi' \textsf{ where } \langle(m_1 = f_1 :: \tau_1), \ldots, (m_n = f_n :: \tau_n)\rangle \mid \pi \in \mathcal{P}_s, \pi' \in \mathcal{P}, m_i \in \mathcal{S}_m, f_i \in \mathcal{S}_f, \tau_i \in \mathcal{T}_q, TV(\vec{\pi}) \subseteq TV(\pi')\}$$

if $\pi \in \mathcal{P}_s$ then it is required that $\vec{\pi} = \langle\rangle$. $\mathcal{S}_f$ is the set of function symbols (definition B.1.12).

Definition 3.1.18 (instance context)
The function $Context :: \mathcal{I}_d \rightarrow \langle\mathcal{P}\rangle$ is defined by:

$$Context(\textsf{instance } \vec{\pi} \Rightarrow \pi' \textsf{ where } \vec{m}) = \vec{\pi}$$

Definition 3.1.19 (instance head)
The function $Head :: \mathcal{I}_d \rightarrow \langle\mathcal{T}\rangle$ is defined by:

$$Head(\textsf{instance } \vec{\pi} \Rightarrow (\vec{\tau} :: c) \textsf{ where } \vec{f}) = \vec{\tau}$$

Definition 3.1.20 (instance members)
The function $Members :: \mathcal{I}_d \rightarrow \langle\mathcal{T}\rangle$ is defined by:

$$Members(\textsf{instance } \vec{\pi} \Rightarrow \tau \textsf{ where } \vec{m}) = \vec{m}$$

Two instances are head-equal if all outermost constructors in their instance heads are equal. No two instance definitions of a class may be head-equal.

Definition 3.1.21 (head-equality)
The function $HeadEqual :: \langle\mathcal{T}\rangle\ \langle\mathcal{T}\rangle \hookrightarrow Bool$ is defined as:

$$\begin{aligned}
&HeadEqual(\langle\tau_1, \ldots, \tau_n\rangle, \langle\sigma_1, \ldots, \sigma_n\rangle) &&= \text{true iff } \forall_{1 \leq i \leq n}[HeadEqual(\tau_i, \sigma_i)]\\
&HeadEqual(\alpha, \sigma) &&= \text{true iff } \sigma = \alpha'\\
&HeadEqual(b, \sigma) &&= \text{true iff } \sigma = b'\\
&HeadEqual(c\ \vec{\tau}, \sigma) &&= \text{true iff } \sigma = c\ \vec{\varphi}\\
&HeadEqual(\tau \rightarrow \tau', \sigma) &&= \text{true iff } \sigma = \varphi \rightarrow \varphi'\\
&HeadEqual(\alpha\ \vec{\tau}, \sigma) &&= \text{true iff } \sigma = \alpha'\ \vec{\varphi}
\end{aligned}$$

Example 3.1.22 (example instance definitions)
Consider these instances of the `Eq` class:

```
instance Eq Int where
   (==) :: Int Int -> Bool
   (==) x y = eqInt x y

instance Eq Char where
   (==) :: Char Char -> Bool
   (==) x y = eqChar x y
```

```
instance Eq [a] | Eq a where
   (==) :: [a] [a] -> Bool | Eq a
   (==) []     []     = True
   (==) []     [y:ys] = False
   (==) [x:xs] []     = False
   (==) [x:xs] [y:ys] = (x == y) && (xs == ys)
```

Assuming the translation presented in example 2.8.1, the corresponding Core instance definitions are:

$\mathsf{instance}\ (\mathtt{Int} :: \mathtt{Eq})\ \mathsf{where}\ (\mathtt{==}) = \mathtt{eqInt} :: \mathtt{Int}\ \mathtt{Int} \rightarrow \mathtt{Bool}$
$\mathsf{instance}\ (\mathtt{Char} :: \mathtt{Eq})\ \mathsf{where}\ (\mathtt{==}) = \mathtt{eqChar} :: \mathtt{Char}\ \mathtt{Char} \rightarrow \mathtt{Bool}$
$\mathsf{instance}\ \langle \mathsf{a} :: \mathtt{Eq}\rangle \Rightarrow (\mathtt{[a]} :: \mathtt{Eq})\ \mathsf{where}$
$\quad (\mathtt{==}) = \mathtt{eqList} :: \langle \mathsf{a} :: \mathtt{Eq}\rangle \Rightarrow (\mathtt{[a]})\ (\mathtt{[a]}) \rightarrow \mathtt{Bool}$

3.1.4 Programs

In this section, Core programs are extended with class and instance definitions. Access functions for function, class and instance definitions are defined. Furthermore, the constraints on class and instance definitions that a valid program must satisfy are specified.

The set of programs Ψ is a partial function that maps class and function symbols to their corresponding definitions.

Definition 3.1.23 (programs)
The set of programs Ψ is defined by:
$\Psi = \{\mathsf{prog}\ p\ c \mid p \in (\mathcal{S}_a \cup \mathcal{S}_f \hookrightarrow \mathcal{A} \cup \mathcal{F}_d), c \in (\mathcal{S}_{Cl} \hookrightarrow \mathcal{C}_d \times \langle \mathcal{I}_d \rangle)\}$
where $\mathcal{S}_a$ (definition B.1.4), $\mathcal{A}$ (definition B.1.18), and $\mathcal{F}_d$ (definition B.1.16) are the sets of algebraic type constructors, algebraic type definitions, and function definitions respectively.

Definition 3.1.24 (access function definition)
The function $\mathit{FuncDef}_\psi :: \mathcal{S}_f \hookrightarrow \langle \mathcal{F}_d \rangle$ is defined by:
$\mathit{FuncDef}_\psi(f) = p(f)$ if $\psi = \mathsf{prog}\ p\ c$

Definition 3.1.25 (access class definition)
The function $\mathit{ClassDef}_\psi :: \mathcal{S}_{Cl} \hookrightarrow \mathcal{C}_d$ is defined by:
$\mathit{ClassDef}_\psi(c) = d$ if $\psi = \mathsf{prog}\ p\ c'$ and $c'(c) = (d, \vec{\imath})$

Definition 3.1.26 (access instance definitions)
The function $\mathit{InstDefs}_\psi :: \mathcal{S}_{Cl} \hookrightarrow \langle \mathcal{I}_d \rangle$ is defined by:
$\mathit{InstDefs}_\psi(c) = \vec{\imath}$ if $\psi = \mathsf{prog}\ p\ c'$ and $c'(c) = (d, \vec{\imath})$

A class without members is called a *compound* class because it only groups its subclasses. The notion of subclasses is formalized by the subclass relation.

Definition 3.1.27 (compound classes)
The predicate $\mathit{Compound}_\psi \subseteq \mathcal{C}_d$ is defined by:
$\mathit{Compound}_\psi(c) \Leftrightarrow \mathit{Members}(\mathit{ClassDef}_\psi(c)) = \langle\rangle$

Definition 3.1.28 (subclass relation)
The subclass relation on classes $\subset_\psi \subseteq \mathcal{S}_{Cl} \times \mathcal{S}_{Cl}$ is defined as the transitive closure of the relation:

$$\{(c_i, c) \mid \mathit{ClassDef}_\psi(c) = (\mathsf{class}\ \langle \vec{\alpha}_1 :: c_1, \ldots, \vec{\alpha}_n :: c_n\rangle \Rightarrow \pi\ \mathsf{where}\ \vec{m})\}$$

There are several constraints regarding the class and instance definitions that a valid program has to satisfy.

- Some obvious welltypedness requirements have to be satisfied. Of course, every instance definition has to define the same member functions as the corresponding class definition. The instance types have to be a valid substitution of the types defined in the class definition. Furthermore, the instance functions have to have the type corresponding to the one in the instance definition.
- No cyclic class dependencies are allowed: $\forall_{c \in \mathcal{S}_{Cl}}\ [c \not\subset_\psi c]$
- No two instance definitions for the same class may be head-equal:
  $\mathit{InstDefs}_\psi(c) = \langle i_1, \ldots, i_n\rangle \Rightarrow$
  $\forall_{j,k \in \{1,\ldots,n\}}[j \neq k \Rightarrow \neg \mathit{HeadEqual}(\mathit{Head}(i_j), \mathit{Head}(i_k))]$

3.1.5 Selection of instance definitions

Dictionary construction requires selecting an instance definition. Which instance definition is chosen depends on which one is applicable given the types at the application. However, CLEAN supports overlapping instance definitions, which means that more than one instance definition may be applicable. A mechanism is required that decides which instance definition to use in that case.

The CLEAN reference manual [17] states that the compiler will always choose the most specific instance definition that matches the required instance type. In cases in which it is not clear what the most specific matching instance definition is, lexicographic order is used. This mechanism is formalized by the $\mathit{ApplyInstance}_\psi$ function.

First, an order on types is defined. A single type τ is more specific than type σ if σ is a type variable and τ is not, or if τ is a symbol type and σ an application type.

Definition 3.1.29 (order on types)
The relation $< \subseteq \mathcal{T} \times \mathcal{T}$ is defined by:

$$< = \{\ (\alpha, b), (\alpha, a\ \vec{\tau}), (\alpha, \sigma \rightarrow \tau), (\alpha, \beta\ \tau), (\alpha\ \vec{\tau}, a\ \vec{\sigma})\}$$

An instance definition i is more specific than i' if its instance head contains more types that are more specific than the corresponding types in the instance head of i' than vice versa. If both instances are equal in this respect, lexicographical order is used.

Definition 3.1.30 (specificity order on type patterns)
The function $Spec :: \langle \mathcal{T} \rangle\ \langle \mathcal{T} \rangle \hookrightarrow \mathbb{N}$ is defined by:

$$Spec(\langle \tau_1, \ldots, \tau_n \rangle, \langle \sigma_1, \ldots, \sigma_n \rangle) = \sum_{1 \le i \le n} [Spec(\tau_i, \sigma_i)]$$

$$Spec(\tau, \sigma) = \begin{cases} 1 & \text{if } \tau < \sigma \\ -1 & \text{if } \sigma < \tau \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.1.31 (lexicographical order on type patterns)
The function $< :: \langle \mathcal{T} \rangle\ \langle \mathcal{T} \rangle \hookrightarrow Bool$ is defined by:

$$\langle \tau_1, \ldots, \tau_n \rangle < \langle \sigma_1, \ldots, \sigma_n \rangle \Leftrightarrow min_i(\tau_i < \sigma_i) < min_i(\sigma_i < \tau_i)$$

Definition 3.1.32 (order on instance definition)
The relation $< \subseteq \mathcal{I}_d \times \mathcal{I}_d$ is defined by:

$$\begin{aligned} i < i' \Leftrightarrow\ & Spec(Head(i), Head(i')) < 0\ \vee \\ & Spec(Head(i), Head(i')) = 0\ \wedge\ Head(i) < Head(i') \end{aligned}$$

The $<$ on instance definitions can now be used to specify what instance definition is selected. First, the set of matching instance definitions is defined.

Definition 3.1.33 (matching instance definitions)
For all programs $\psi \in \Psi$, the set of instance definitions of class c that match types $\vec{\tau}$ is defined by:

$$Matching_\psi(\vec{\tau} :: c) = \{ i \mid i \in InstDefs_\psi(c), \exists_{* \in \circledast}[*(Head(i)) = \vec{\tau}] \}$$

The most specific matching instance definition is the most specific instance definition from the set of matching instance definitions.

Definition 3.1.34 (most specific matching instance definition)
For all programs $\psi \in \Psi$, the function $MostSpecific_\psi :: \mathcal{P} \rightarrow \mathcal{I}_d$ is defined by:

$$\begin{aligned} MostSpecific_\psi(\vec{\tau} :: c) = i \Leftrightarrow\ & i \in Matching_\psi(\vec{\tau} :: c)\ \wedge \\ & \forall_{i' \in Matching_\psi(\vec{\tau}::c)}[i = i' \vee i' < i] \end{aligned}$$

Example 3.1.35 (instance definition selection)
Some examples will illustrate the selection mechanism. Consider the next instance definitions:

```
instance Class Int [a] where ...
instance Class Int a   where ...
instance Class a   Int where ...
```

When an instance definition has to be selected for type `(Int [Char])`, the first two instance definitions are applicable. The first one will be selected because it is more specific (`[a]` is more specific than `a`).
When an instance definition has to be selected for type `(Int Int)`, the last two instance definitions are applicable. They are both equally specific, hence lexicographical order is used and the instance definition `(Int a)` is selected.

The $ApplyInstance_\psi$ function yields the most specific instance definition. In case the class has no members, and therefore no instance definitions, it returns a generic instance definition. This way, we do not have to make exceptions for compound classes later on.

Definition 3.1.36 (instance definition selection)
For all programs $\psi \in \Psi$, the function $\mathit{ApplyInstance}_\psi :: \mathcal{P} \rightarrow \mathcal{I}_d$ is defined by:

$$\mathit{ApplyInstance}_\psi(\vec{\tau} :: c) = \begin{cases} \textsf{instance } \langle\rangle \Rightarrow \vec{\alpha} \textsf{ where } \langle\rangle & \text{if } \mathit{Compound}_\psi(c) \\ \mathit{MostSpecific}_\psi(\vec{\tau} :: c) & \text{otherwise} \end{cases}$$

3.1.6 Dependencies

A useful and intuitive notion for defining the set of instances and dictionaries is the *dependencies* of an instance; all instances the instance depends on. This includes both the subclasses and the instance definition context. Note that $*_{(\vec{\tau} \leftrightarrow \vec{\sigma})}$ (definition A.2.4) denotes the most general unifier of types $\vec{\tau}$ and $\vec{\sigma}$.

Definition 3.1.37 (dependencies)
For all programs $\psi \in \Psi$ and classes c, the function $\mathit{Deps}_\psi :: \mathcal{P}\ \mathcal{I}_d \rightarrow \langle\mathcal{P}\rangle$ is defined by:

$$\mathit{Deps}_\psi(\vec{\tau} :: c, i) = \mathit{SubClasses}_\psi(\vec{\tau} :: c) \circ *_{(\mathit{Head}(i) \leftrightarrow \vec{\tau})}(\mathit{Context}(i))$$

When i is omitted, $\mathit{ApplyInstance}_\psi(\vec{\tau} :: c)$ is assumed for it.

Definition 3.1.38 (subclasses)
For all programs $\psi \in \Psi$ and classes c, the function $\mathit{SubClasses}_\psi :: \mathcal{P} \rightarrow \langle\mathcal{P}\rangle$ is defined by:

$$\mathit{SubClasses}_\psi(\vec{\tau} :: c, i) = \langle *_{(\vec{\alpha} \leftrightarrow \vec{\tau})}(\pi_1), \ldots, *_{(\vec{\alpha} \leftrightarrow \vec{\tau})}(\pi_n)\rangle$$
$$\Leftrightarrow \mathit{ClassDef}_\psi(c) = \textsf{class } \langle\pi_1, \ldots, \pi_n\rangle \Rightarrow \vec{\alpha} \textsf{ where } \vec{m}$$

Example 3.1.39 (dependencies)
Some simple examples of this can be shown using the `Eq` class (example 2.3.1):

$$\begin{aligned} \mathit{Deps}_\psi(\texttt{Int} :: \texttt{Eq}) &= \langle\rangle \\ \mathit{Deps}_\psi(\texttt{Char} :: \texttt{Eq}) &= \langle\rangle \\ \mathit{Deps}_\psi(\texttt{[Int]} :: \texttt{Eq}) &= \langle\texttt{Int} :: \texttt{Eq}\rangle \\ \mathit{Deps}_\psi(\texttt{[[Char]]} :: \texttt{Eq}) &= \langle\texttt{[Char]} :: \texttt{Eq}\rangle \end{aligned}$$

3.1.7 Instances

A dictionary exist for types for which an instance of the class exists and for every instance there is exactly one dictionary. An instance can be identified by an instantiation of the class parameters with closed types. In this section, this is used to define the set of instances.

An instance is defined by an instance definition if all of the definition's dependencies can be satisfied and it is the one that will be applied given the type at the application. This can be easily formalized. However, to prevent cyclic dependencies (see 2.9), an instance $\vec{\tau}$ may only depend on instances that can be created without depending on itself. This is solved by an iterative definition.

Definition 3.1.40 (subsets of instances)
For all programs $\psi \in \Psi$, classes $c \in \mathcal{C}_d$, the set $\mathit{Instances}_\psi(c, n) \in \wp(\langle\mathcal{T}^{\textsf{closed}}\rangle)$ is defined by:

$$\begin{aligned}
\mathit{Instances}_\psi(c,0) &= \emptyset \\
\mathit{Instances}_\psi(c,n+1) &= \{\vec{\tau} \mid i \in \mathit{InstDefs}_\psi(c), * \in \circledast, \vec{\tau} \in \langle \mathcal{T}^{\mathsf{closed}} \rangle \\
&\qquad \vec{\tau} = *(\mathit{Head}(\mathit{ApplyInstance}_\psi(\vec{\tau} :: c))) \\
&\qquad \forall_{(\vec{\tau}' :: c') \in \mathit{Deps}_\psi(\vec{\tau} :: c)}[\vec{\tau}' \in \mathit{Instances}_\psi(c', n)]
\end{aligned}$$

Definition 3.1.41 (instances)
For all programs $\psi \in \Psi$ and classes $c \in \mathcal{C}_d$, the set $\mathit{Instances}_\psi(c) \in \wp(\langle \mathcal{T}^{\mathsf{closed}} \rangle)$ is defined by:
$\mathit{Instances}_\psi(c) = \bigcup_{n \in \mathbb{N}}[\mathit{Instances}_\psi(c,n)]$

Example 3.1.42 (instances of the `Eq` class)
The set of instances of the `Eq` class defined in example 2.3.1 is:
$\mathit{Instances}_\psi(\texttt{Eq}) = \{\texttt{Int}, \texttt{Char}, \texttt{[Int]}, \texttt{[Char]}, \texttt{(Int, Int)}, \texttt{(Char, Int)}, \texttt{(Int, Char)}, \texttt{(Char, Char)}, \texttt{[[Int]]}, \texttt{[[Char]]}, \texttt{[(Int, Int)]}, \ldots\}$

3.1.8 Dictionary creation

Dictionary types and constructors were introduced in section 3.1.1. In this section, the creation of dictionary values is defined.

A dictionary for a predicate $\vec{\tau} :: c$ is a record that contains the dictionaries for the subclasses of c and implementations of its member functions at instance $\vec{\tau}$. The function Dict_ψ creates these dictionaries.

Definition 3.1.43 (dictionary creation)
For all programs $\psi \in \Psi$, the function $\mathit{Dict}_\psi :: \mathcal{P} \hookrightarrow \mathcal{E}$ that creates the dictionary for a class constraint is defined by:
$\mathit{Dict}_\psi(\langle \pi_1, \ldots, \pi_n \rangle) = \langle \mathit{Dict}_\psi(\pi_1), \ldots, \mathit{Dict}_\psi(\pi_n) \rangle$
$\mathit{Dict}_\psi(\vec{\tau} :: c) = \mathit{DictConstr}_\psi(c)\ \mathit{Dict}_\psi(\mathit{SubClasses}_\psi(\vec{\tau} :: c)) \circ \langle e_1, \ldots, e_n \rangle$
iff:
$\mathit{ApplyInstance}_\psi(\vec{\tau} :: c) = \mathsf{instance}\ \vec{\pi} \Rightarrow (\vec{\sigma} :: c)\ \mathsf{where}\ \langle m_1, \ldots, m_n \rangle$
$\forall_{1 \le i \le n}[e_i = \mathit{Member}_\psi(*_{(\vec{\sigma} \leftrightarrow \vec{\tau})}, m_i)]$

Definition 3.1.44 (creation of a sequence of dictionaries)
For all programs $\psi \in \Psi$, the function $\mathit{Dict}_\psi :: \langle \mathcal{P} \rangle \hookrightarrow \langle \mathcal{E} \rangle$ that creates the dictionaries for a sequence of class constraints is defined by:

The concrete implementation of a member function is created by selecting an instance function from the instance definition and applying it to the dictionaries required by it.

Definition 3.1.45 (member creation)
For all programs $\psi \in \Psi$, the function Member_ψ that creates the concrete implementations of a member function for a dictionary of a class constraint is defined by:
$\mathit{Member}_\psi(*, o = f :: \vec{\pi} \Rightarrow \tau) = \mathsf{func}\ f\ \mathit{Dict}_\psi(*(\vec{\pi}))$

Example 3.1.46 (dictionary creation)

Again, consider the `Eq` class (example 2.3.1 and 2.8.1):

$Dict_\psi(\texttt{Int} :: \texttt{Eq}) = \{$ `== = eqInt` $\}$
$Dict_\psi(\texttt{Char} :: \texttt{Eq}) = \{$ `== = eqChar` $\}$
$Dict_\psi(\texttt{[Int]} :: \texttt{Eq}) = \{$ `== = eqList` $\{$ `== = eqInt` $\}$ $\}$
$Dict_\psi(\texttt{[[Char]]} :: \texttt{Eq}) = \{$ `== = eqList` $\{$
`== = eqList` $\{$ `== = eqChar` $\}$ $\}$ $\}$

3.2 Specifying class constrained properties

In this section, it is shown how class constrained properties can be defined in Core. Using a natural assumption, properties can be straightforwardly defined by quantification over dictionaries. It is also shown that this assumption is not required when an additional predicate is introduced.

It has been repeatedly stated that in the translation from Clean to Core, class constraints are translated to dictionaries. It will be no surprise that the same mechanism can be used for properties. Every class constraint can be replaced by the introduction of a dictionary for the corresponding class and type parameters. Example 3.2.1 shows this for symmetry of equality.

Example 3.2.1 (translation of properties)

Consider the class constrained property that states symmetry of equality, defined in section 1.4:

$\forall_{\texttt{a|Eq a}}\forall_{\texttt{x}\in\texttt{a}}\forall_{\texttt{y}\in\texttt{a}}[\texttt{x == y} \Rightarrow \texttt{y == x}]$

By replacing the class constraint with a universal quantor over the corresponding dictionary and translating the expressions, we get:

$\forall_{\texttt{a}}\forall_{\texttt{d}\in\texttt{dicteq a}}\forall_{\texttt{x}\in\texttt{a}}\forall_{\texttt{y}\in\texttt{a}}[\texttt{x d.== y} \Rightarrow \texttt{y d.== x}]$

However, this is not a correct translation of the property as such. In section 3.1.1, dictionary types where added to the Core language. Unfortunately, this type does not really capture the notion of a valid dictionary. The type is just a record type that can be filled with any values of the correct type. Valid dictionaries are much more restricted; they only contain expressions (the member instances) that are generated from the class and instance definitions.

Example 3.2.2 (dictionary type too general)

Consider the dictionary type for the `Eq` class (example 2.8.1):

```
:: eqdict a = { (==) :: a a -> Bool }
```

Some values of the type `eqdict` are:

```
{ (==) = ==int }
{ (==) = eqlist ==int }
```

but also:

```
{ (==) = or }
```

that is not a valid dictionary for the `Eq` class.

There are two ways to solve this inconvenience. The first is to just assume that the dictionary type only contains valid dictionaries: $DictType_\psi(c)\ \vec{\tau}$ contains only the value $Dict_\psi(\vec{\tau} :: c)$. Thus, the expression `{ (==) = or }` in example 3.2.2 is not considered a value of type `eqdict Bool`. This is a reasonable assumption, because dictionary values are created by the compiler and therefore will always be valid dictionaries. In this case, example 3.2.1 does show how class constrained properties can be defined using dictionaries.

The second option is to add a predicate to the logic language that is only true when the dictionary value is a valid dictionary.

Example 3.2.3 (restricting dictionary values)
The predicate $\mathit{Valid}(d, \pi)$ is true iff $d = \mathit{Dict}_\psi(\pi)$:
$\forall_{\mathtt{a}}\forall_{\mathtt{d}\in\mathtt{dicteq\ a}}[\mathit{Valid}(\mathtt{d}, \mathtt{Eq\ a}) \Rightarrow \forall_{\mathtt{x}\in\mathtt{a}}\forall_{\mathtt{y}\in\mathtt{a}}[\mathtt{x\ d.==\ y} \Rightarrow \mathtt{y\ d.==\ x}]]$

In this project, for reasons of convenience, the first option will be used. In chapter 4, which discusses the implementation, we come back to this issue.

Assumption 3.2.4 (dictionaries of a class)
For all programs $\psi \in \Psi$, classes $c \in \mathcal{C}_d$, and types $\vec{\tau} \in \langle \mathcal{T}^{\mathsf{closed}} \rangle$, we assume:

$$\mathit{Type}_\psi(\mathit{DictConstr}_\psi(c)\ \vec{e}, \phi) = (\mathit{DictType}_\psi(c)\ \vec{\tau})$$
$$\Leftrightarrow (\mathit{DictConstr}_\psi(c)\ \vec{e}) = \mathit{Dict}_\psi(\vec{\tau} :: c)$$

3.3 Proof rules and tactics

To prove class constrained properties, additional proof rules are required; a member application cannot be rewritten when the value of the dictionary it is selected from is unknown. The value of a valid dictionary depends on the instance required at the application. Since the set of types (instances) is inductively defined, the most natural approach is induction on the set of types (instances). This allows us to assume the property for smaller types. In general, we would like to be able to assume hypotheses for all sensible dependencies.

In section 3.3.1, a definition that executes one step of the dictionary generation is given. This will be used to specify the tactics later on, but can by itself be used for a less advanced tactic as well. Because the succeeding proof rules and tactics are based on induction, well-founded induction is explained in section 3.3.2. In section 3.3.3 structural induction on types is explained and it is argued why this does not really suffice. An induction scheme based on the instance definitions of a class is defined in section 3.3.4 and extended to multiple class constraints in 3.3.5. Section 3.3.6 briefly describes how non-simple class constraints can be handled. In conclusion, section 3.3.7 compares the proof rules and tactics to related work on Isabelle.

3.3.1 Dictionary expansion

Because we have defined that a dictionary type contains only one value, the valid dictionary, it is possible to generate part of the dictionary value when part of the dictionary type is known, by executing one step of the Dict_ψ function (definition 3.1.43). Instead of a recursive creation of dictionaries, expression variables are introduced for the dependencies. To allow more hypotheses to be assumed later on, compound subclasses are always expanded.

It turns out to be useful to have a shorthand for quantifying over dictionary variables. This is denoted by quantification over a predicate. Note that expression variables are introduced that are not explicitly written at the quantor.

Definition 3.3.1 (shorthand for dictionary introduction)
A shorthand for quantifying over dictionaries is defined by:

$$\forall_{\langle \vec{\tau}_1 :: c_1, \ldots, \vec{\tau}_n :: c_n \rangle} \equiv \forall_{(d_{(\vec{\tau}_1 :: c_1)} \in \mathit{DictType}_\psi(c_1)\ \vec{\tau}_1)} \cdots \forall_{(d_{(\vec{\tau}_n :: c_n)} \in \mathit{DictType}_\psi(c_n)\ \vec{\tau}_n)}$$

The introduced expression variables are of the form $d_{(\pi)}$. These are normal expression variables, but their name suggests that a dictionary for predicate π is assigned to it. By $d_{(\langle \pi_1, \ldots, \pi_n \rangle)}$ the vector $\langle d_{(\pi_1)}, \ldots, d_{(\pi_n)} \rangle$ is denoted.

Definition 3.3.2 (dictionary creation step)
For all programs $\psi \in \Psi$, the function $\mathit{Dict}^1_\psi :: \mathcal{P} \cup \langle \mathcal{P} \rangle\ \mathcal{I}_d \hookrightarrow \mathcal{E}$ that creates dictionaries is defined by:

$$\begin{aligned} \mathit{Dict}^1_\psi(\langle \pi_1, \ldots, \pi_n \rangle, i) &= \langle \mathit{Dict}^1_\psi(\pi_1, i), \ldots, \mathit{Dict}^1_\psi(\pi_n, i) \rangle \\ \mathit{Dict}^1_\psi(\vec{\tau} :: c, i) &= \mathit{DictConstr}_\psi(c)\ \langle s'_1, \ldots, s'_{n'} \rangle \circ \langle m'_1, \ldots, m'_n \rangle \end{aligned}$$

iff:

$i = \mathsf{instance}\ \vec{\pi} \Rightarrow (\vec{\sigma} :: c)\ \mathsf{where}\ \langle m_1, \ldots, m_n \rangle$
$\mathit{SubClasses}_\psi(\vec{\tau} :: c) = \langle s_1, \ldots, s_{n'} \rangle$
$\forall_{1 \le i \le n}[m'_i = \mathit{Member}^1_\psi(*_{(\vec{\sigma} \leftrightarrow \vec{\tau})}, m_i)]$
$\forall_{1 \le i \le n'}[s'_i = \mathit{SubClass}^1_\psi(s_i)]$
When i is omitted, $\mathit{ApplyInstance}_\psi(\vec{\tau} :: c)$ is assumed for it.

Definition 3.3.3 (member creation step)
For all programs $\psi \in \Psi$, the function Member^1_ψ, which creates the concrete implementation of a member function for a dictionary, is defined by:

$$\mathit{Member}^1_\psi(*, o = f :: \vec{\pi} \Rightarrow \tau) = \mathsf{func}\ f\ d_{(*(\vec{\pi}))}$$

Definition 3.3.4 (subclass creation step)
For all programs $\psi \in \Psi$, the function $\mathit{SubClass}^1_\psi :: \mathcal{P} \hookrightarrow \mathcal{E}$ is defined by:

$$\mathit{SubClass}^1_\psi(\vec{\tau} :: c) = \begin{cases} \mathit{DictConstr}_\psi(c)\ \langle s'_1, \ldots, s'_n \rangle & \text{if } \mathit{Compound}_\psi(c) \\ d_{(\vec{\tau} :: c)} & \text{otherwise} \end{cases}$$

iff:

$\mathit{SubClasses}_\psi(\vec{\tau} :: c) = \langle s_1, \ldots, s_n \rangle$
$\forall_{1 \le i \le n}[s'_i = \mathit{SubClass}^1_\psi(s_i)]$

We now introduce a new form of dependencies; the instances that a value created by Dict^1_ψ depends on. From now on, with dependencies, this definition is meant.

Definition 3.3.5 (single step dependencies)
For all programs $\psi \in \Psi$ and classes c, the function $\mathit{Deps}^1_\psi :: \mathcal{P} \times \mathcal{I}_d \to \langle \mathcal{P} \rangle$ is defined by:

$$\mathit{Deps}^1_\psi(\pi, i) = \mathit{ExpComp}_\psi(\mathit{Deps}_\psi(\pi, i))$$

Definition 3.3.6 (expand compounds)
For all programs $\psi \in \Psi$ and classes c, the function $\mathit{ExpComp}_\psi :: \mathcal{P} \cup \langle\mathcal{P}\rangle \rightarrow \langle\mathcal{P}\rangle$ is defined by:

$$\mathit{ExpComp}_\psi(\langle \pi_1, \ldots, \pi_n \rangle) = \mathit{ExpComp}_\psi(\pi_1) \circ \ldots \circ \mathit{ExpComp}_\psi(\pi_n)$$
$$\mathit{ExpComp}_\psi(\pi) = \begin{cases} \mathit{ExpComp}_\psi(\mathit{SubClasses}_\psi(\pi)) & \text{if } \mathit{Compound}_\psi(c) \\ \langle \pi \rangle & \text{otherwise} \end{cases}$$

The relation between dictionary creation and single step dictionary creation is very straightforward.

Assumption 3.3.7 (relation between dictionary creation functions)
For all programs $\psi \in \Psi$ and all predicates $\pi \in \mathcal{P}$ for which $FV(\pi) = \langle\rangle$:

$$\forall_{\mathit{Deps}^1_\psi(\pi)}[\mathit{Dict}^1_\psi(\pi) = \mathit{Dict}_\psi(\pi)]$$

Using this one step dictionary expansion function, a tactic can be created. The single step dictionary creation uses one of the instance definitions. Hence, if the step is done for all instance definitions, the conjunction of the resulting goals implies the original property. In fortunate cases where not all instance definitions are applicable, less goals are generated.

Tactic 3.3.8 (dictionary expansion)
For all programs $\psi \in \Psi$, types $\vec{\tau} \in \langle\mathcal{T}\rangle$, and classes $c \in \mathcal{S}_{Cl}$, the **(expand)** rule is defined by:

$$\frac{\begin{array}{l} i \in \mathit{InstDefs}_\psi(c) \\ \quad [\ \mathit{ApplyInstance}_\psi(\pi) = i\ \wedge \\ \quad\quad \forall_{TV(\pi)}[\forall_{\mathit{Deps}^1_\psi(\pi)}[P(\mathit{Dict}^1_\psi(\pi))]] \\ \quad] \end{array}}{\forall_{TV(\pi)}[\forall_\pi[P(d_\pi)]]}$$

Example 3.3.9 (dictionary expansion)
Given the `Eq` class from examples 2.3.1 and 2.8.1 and the **(expand)** tactic, symmetry of equality:

$$\forall_{\mathtt{a}}\forall_{\mathtt{a::Eq}}\forall_{\mathtt{x}\in\mathtt{a}}\forall_{\mathtt{y}\in\mathtt{a}}[\mathtt{d}_{\mathtt{(a::Eq)}}\mathtt{.==\ x\ y} \Rightarrow \mathtt{d}_{\mathtt{(a::Eq)}}\mathtt{.==\ y\ x}]$$

is implied by three goals, one for each instance definition:

- $\forall_{\mathtt{x}\in\mathtt{Int}}\forall_{\mathtt{y}\in\mathtt{Int}}\ [\ \{\ \mathtt{==\ =\ eqInt}\ \}\mathtt{.==\ x\ y}$
  $\quad \Rightarrow \{\ \mathtt{==\ =\ eqInt}\ \}\mathtt{.==\ y\ x}\ ]$

- $\forall_{\mathtt{x}\in\mathtt{Real}}\forall_{\mathtt{y}\in\mathtt{Real}}\ [\ \{\ \mathtt{==\ =\ eqReal}\ \}\mathtt{.==\ x\ y}$
  $\quad \Rightarrow \{\ \mathtt{==\ =\ eqReal}\ \}\mathtt{.==\ y\ x}\ ]$

- $\forall_{\mathtt{a}}\forall_{\mathtt{a::Eq}}\forall_{\mathtt{x}\in\mathtt{[a]}}\forall_{\mathtt{y}\in\mathtt{[a]}}\ [\ \{\ \mathtt{==\ =\ eqList\ d}_{\mathtt{(a::Eq)}}\ \}\mathtt{.==\ x\ y}$
  $\quad \Rightarrow \{\ \mathtt{==\ =\ eqList\ d}_{\mathtt{(a::Eq)}}\ \}\mathtt{.==\ y\ x}\ ]$

Note that the third goal is not provable without the assumption that equality on the list members is symmetric.

3.3.2 Well-founded induction

In this section, the proof method of *well-founded induction* is explained. To introduce the basic idea, we begin with an intuitive example of well-founded induction. Consider a long row of dominoes. It can be concluded that all will fall from two statements: 1. The first domino falls. 2. When a domino falls, its neighbor will also fall.

Well-founded induction is a generalization of this idea for well-founded sets for which a partial order relation is defined.

Definition 3.3.10 (partial order)
A binary relation $\leq$ on a set P is a partial order if:

$$\begin{array}{lll} \forall_{x \in P} & x \leq x & \text{(reflexivity)} \\ \forall_{x,y,z \in P} & x \leq y \wedge y \leq z \Rightarrow x \leq y & \text{(transitivity)} \\ \forall_{x,y \in P} & x \leq y \Rightarrow y \leq x \Rightarrow x = y & \text{(antisymmetry)} \end{array}$$

Definition 3.3.11 (well-founded relation)
A well-founded relation is an order relation R on a set P where every descending chain starting at $x \in P$ is finite.

The well-founded induction theorem states that a property can be proven for all elements by proving it for all elements while assuming it holds for all smaller elements.

Theorem 3.3.12 (well-founded induction)
If S is a set, $P(x)$ a property, $\leq$ is a well-founded partial order on S, and $<$ is defined such that $x < y \Leftrightarrow x \leq y \wedge x \neg = y$:

$$\frac{\forall_{x \in S}\,[\forall_{s < x}\,[P(s)] \Rightarrow P(x)]}{\forall_{x \in S}[P(x)]}$$

A special case of well-founded induction is natural induction, A property is proven by proving it for 0 and for $n + 1$ assuming it holds for n.

Theorem 3.3.13 (natural induction)
If $\mathbb{N}$ is the set of natural numbers and $P(x)$ a property then:

$$\frac{[P(0) \wedge \forall_{n \in \mathbb{N}}\,[P(n) \Rightarrow P(n+1)]]}{\forall_{n \in \mathbb{N}}[P(n)]}$$

This is what happens in case of the dominoes; label the dominoes in the row, starting with zero and imagine the property “domino n will fall” for $P(n)$. In the next three sections, the well-founded induction theorem will be used to specify useful proof rules for class constrained properties.

3.3.3 Structural induction on types

First, the most straightforward induction scheme, structural induction on types, is investigated. The partial order on types required for structural induction follows directly from the definition of types.

Definition 3.3.14 (subexpression relation on closed types)
For all programs $\psi \in \Psi$, the relation $<^1 \subseteq \mathcal{T}^{\mathsf{closed}} \times \mathcal{T}^{\mathsf{closed}}$ is defined by:

$$\begin{array}{ll} \mu <^1 \sigma \to \tau & \Leftrightarrow \mu = \sigma \vee \mu = \tau \\ \sigma <^1 a\ \tau_1 \ldots \tau_n & \Leftrightarrow \sigma = \tau_1 \vee \ldots \vee \sigma = \tau_n \end{array}$$

The partial order $\leq$ is the reflexive transitive closure of $<^1$.

Definition 3.3.15 (structural partial order on closed types)
For all programs $\psi \in \Psi$, the relation $\leq \subseteq \mathcal{T}^{\mathsf{closed}} \times \mathcal{T}^{\mathsf{closed}}$ is defined by:

$$\begin{array}{ll} \forall_{\tau \in \mathcal{T}^{\mathsf{closed}}} & \tau \leq \tau \\ \forall_{\tau, \sigma \in \mathcal{T}^{\mathsf{closed}}} & \tau <^1 \sigma \Rightarrow \tau \leq \sigma \\ \forall_{\tau, \sigma, \mu \in \mathcal{T}^{\mathsf{closed}}} & \tau \leq \sigma \wedge \sigma \leq \mu \Rightarrow \tau \leq \mu \end{array}$$

Using the well-founded induction theorem, the structural induction theorem for types is straightforwardly defined.

Definition 3.3.16 (structural induction on closed types)
For all programs $\psi \in \Psi$ and properties P, structural induction on closed types **(struct-rule)** is defined by:

$$\frac{\forall_{\tau \in \mathcal{T}^{\mathsf{closed}}} [\forall_{\sigma <^1 \tau} [P(\sigma)] \Rightarrow P(\tau)]}{\forall_{\tau \in \mathcal{T}^{\mathsf{closed}}} [P(\tau)]}$$

By applying this proof rule, hypotheses may be assumed in the generated goals for structurally smaller types. Hence, this seems to be a useful proof rule. Unfortunately, structural induction on types does not allow all desired hypotheses to be assumed. The hypotheses will be used to assume the property for dependencies. This requires that the dependencies are of a structurally smaller type than the dictionary itself. This is not necessarily the case for multi-parameter classes, as is shown in example 3.3.17. This problem can be solved by deriving the induction scheme from the instance definitions.

Example 3.3.17 (dependencies are not always structurally smaller)
Consider the next class and instance definitions:

```
class Eq2 a b where
   eq2 :: a b -> Bool where

instance Eq2 Int Int where
   eq2 x y = x == y

instance Eq2 Char Char where
   eq2 x y = x == y

instance Eq2 (a, b) [c] | Eq2 a c & Eq2 b c where
   eq2 (x, y) [u, v] = eq2 x u && eq2 y v
   eq2 x y           = False

instance Eq2 [c] (a, b) | Eq2 a c & Eq2 b c where
   eq2 x y = eq2 y x
```

The instance of `Eq2` for `[Int] (Char, Char)` depends on the instance for `Char Int`, which is not structurally smaller because `Char` is not structurally smaller than `[Int]`, and `Int` is not structurally smaller than `(Char, Char)`. Hence, the hypothesis cannot be assumed for this dependency.

3.3.4 Induction on instances

The problem with structural induction can be avoided when an order is used that is based on the structure in the set of instances (dictionaries) of a class.

First, a partial order is defined on the set of instances of a class. Then, well-founded induction on instances is defined. It is used to derive a tactic that follows the structure of the instances (dictionaries).

Because the idea is to base the order on the instance definitions, an instance $\vec{\sigma}$ is considered one step smaller than $\vec{\tau}$ if creating the dictionary for $\vec{\tau}$ requires the dictionary for $\vec{\sigma}$, that is, if $\vec{\sigma} :: c$ is a dependency of the most specific instance definition for $\vec{\tau} :: c$.

Definition 3.3.18 (order on instances)
For all programs $\psi \in \Psi$ and classes $c \in \mathcal{S}_{Cl}$, the one step order on instances $<^1_{(\psi,c)} \subseteq \mathit{Instances}_\psi(c) \times \mathit{Instances}_\psi(c)$ is defined by:

$$\vec{\sigma} <^1_{(\psi,c)} \vec{\tau} \Leftrightarrow (\vec{\sigma} :: c) \in \mathit{Deps}^1_\psi(\vec{\tau} :: c)$$

Example 3.3.19 (order on instances)
For the `Eq` class (example 2.3.1) we have for example:

$\texttt{Int} <^1_{(\psi,\texttt{Eq})} \texttt{[Int]}$

$\texttt{[Char]} <^1_{(\psi,\texttt{Eq})} \texttt{[[Char]]}$

The partial order $\leq_{(\psi,c)}$ is the reflexive transitive closure of $<^1_{(\psi,c)}$ (see C.1).

Definition 3.3.20 (partial order on instances)
For all programs $\psi \in \Psi$ and classes $c \in \mathcal{S}_{Cl}$, the partial order on instances of c, $\leq_{(\psi,c)} \subseteq \mathit{Instances}_\psi(c) \times \mathit{Instances}_\psi(c)$, is defined by:

$$\begin{array}{ll}
\forall_{\vec{\tau} \in \mathit{Instances}_\psi(c)} & \vec{\tau} \leq_{(\psi,c)} \vec{\tau} \\
\forall_{\vec{\tau},\vec{\sigma} \in \mathit{Instances}_\psi(c)} & \vec{\tau} <^1_c \vec{\sigma} \Rightarrow \vec{\tau} \leq_{(\psi,c)} \vec{\sigma} \\
\forall_{\vec{\tau},\vec{\sigma},\vec{\mu} \in \mathit{Instances}_\psi(c)} & \vec{\tau} \leq_{(\psi,c)} \vec{\sigma} \wedge \vec{\sigma} \leq_{(\psi,c)} \vec{\mu} \Rightarrow \vec{\tau} \leq_{(\psi,c)} \vec{\mu}
\end{array}$$

Using the $<^1_{(\psi,c)}$ and $\leq_{(\psi,c)}$ relations, well-founded induction can be performed on the set of instances.

Proof rule 3.3.21 (well-founded induction on instances)
For all programs $\psi \in \Psi$, classes $c \in \mathcal{S}_{Cl}$, and properties P, well-founded induction on $\mathit{Instances}_\psi(c)$ **(inst-rule)** is defined by:

$$\frac{\forall_{\vec{\tau} \in \mathit{Instances}_\psi(c)}[\forall_{\vec{\sigma} <^1_{(\psi,c)} \vec{\tau}}[P(\vec{\sigma})] \rightarrow P(\vec{\tau})]}{\forall_{\vec{\alpha} \in \mathit{Instances}_\psi(c)}[P(\vec{\alpha})]}$$

The final tactic is a combination of induction on instances and dictionary expansion (see C.3). Given a property with a simple class constraint, a goal is generated for every instance definition of that class and the property is assumed for all instances that are one step smaller than the assumed instance.

Tactic 3.3.22 (induction on instances)
For all programs $\psi \in \Psi$ and classes $c \in \mathcal{S}_{Cl}$, the following holds **(inst-tactic)**:

$$\frac{\begin{array}{l}\forall_{i \in \mathit{InstDefs}_\psi(c)} \forall_{TV(\mathit{Head}(i) \in \langle \mathcal{T}^{\mathsf{closed}} \rangle)} \forall_{\mathit{Deps}(\mathit{Head}(i)::c,i)} \\
\quad [\ \forall_{(\vec{\sigma}::c') \in \mathit{Deps}_\psi(\mathit{Head}(i)::c,i)}[c = c' \Rightarrow \exists_*[*(\vec{\alpha}) = \vec{\sigma}] \Rightarrow P(d_{(\vec{\sigma}::c)}, \vec{\sigma})] \\
\quad\quad \Rightarrow P(\mathit{Dict}^1_\psi(\mathit{Head}(i) :: c, i), \mathit{Head}(i))] \\
\quad]\end{array}}{\forall_{\vec{\alpha} \in \langle \mathcal{T}^{\mathsf{closed}} \rangle} \forall_{\vec{\alpha}::c}[P(d_{(\vec{\alpha}::c)}, \vec{\alpha})]}$$

where it is assumed that all $i \in \mathit{InstDefs}_\psi(c)$ contain fresh variables only (this can be assured by alpha conversion).

Example 3.3.23 (induction on instances)

Given the `Eq` class from examples 2.3.1 and 2.8.1 and using **(inst-tactic)**, symmetry of equality:

$\forall_{\mathtt{a}}\forall_{\mathtt{a::Eq}}\forall_{\mathtt{x}\in\mathtt{a}}\forall_{\mathtt{y}\in\mathtt{a}}[\mathtt{d}_{(\mathtt{a::Eq})}\mathtt{.==\ x\ y} \Rightarrow \mathtt{d}_{(\mathtt{a::Eq})}\mathtt{.==\ y\ x}]$

is implied by three goals, one for each instance definition:

- $\forall_{\mathtt{x}\in\mathtt{Int}}\forall_{\mathtt{y}\in\mathtt{Int}}\ [\ \{\ \mathtt{==\ =\ eqInt}\ \}\mathtt{.==\ x\ y}$
  $\Rightarrow \{\ \mathtt{==\ =\ eqInt}\ \}\mathtt{.==\ y\ x}\ ]$

- $\forall_{\mathtt{x}\in\mathtt{Real}}\forall_{\mathtt{y}\in\mathtt{Real}}\ [\ \{\ \mathtt{==\ =\ eqReal}\ \}\mathtt{.==\ x\ y}$
  $\Rightarrow \{\ \mathtt{==\ =\ eqReal}\ \}\mathtt{.==\ y\ x}\ ]$

- $\forall_{\mathtt{a}}\forall_{\mathtt{a::Eq}}\ [\ \forall_{\mathtt{x}\in\mathtt{a}}\forall_{\mathtt{y}\in\mathtt{a}}[\mathtt{d}_{(\mathtt{a::Eq})}\mathtt{.==\ x\ y} \Rightarrow \mathtt{d}_{(\mathtt{a::Eq})}\mathtt{.==\ y\ x}]$
  $\Rightarrow \forall_{\mathtt{x}\in[\mathtt{a}]}\forall_{\mathtt{y}\in[\mathtt{a}]}\ [\ \{\ \mathtt{==\ =\ eqList\ d}_{(\mathtt{a::Eq})}\ \}\mathtt{.==\ x\ y}$
  $\Rightarrow \{\ \mathtt{==\ =\ eqList\ d}_{(\mathtt{a::Eq})}\ \}\mathtt{.==\ y\ x}\ ]$
  $]$

Note that this is almost the same as dictionary expansion **(expand)** (proof rule 3.3.8, example 4.2.4) except for the important induction hypothesis in the third goal. This induction hypothesis makes it possible to prove the goal using the tactics already available in Sparkle.

Example 3.3.23 provides a clear example of the tactic, but it does not show the advantage over structural induction. Therefore, example 3.3.24 shows that hypotheses can be assumed for types that are not structurally smaller. This makes this tactic useful for more type classes than a tactic based on structural induction on types.

Example 3.3.24 (induction on instances)

Given the `Eq2` class from example 3.3.17, the following property can be defined:

$\forall_{\mathtt{a,b}}\forall_{\langle\mathtt{a,b}\rangle\mathtt{::Eq2}}\forall_{\langle\mathtt{b,a}\rangle\mathtt{::Eq2}}\forall_{\mathtt{x}\in\mathtt{a}}\forall_{\mathtt{y}\in\mathtt{b}}[\mathtt{d}_{(\langle\mathtt{a,b}\rangle\mathtt{::Eq2})}\mathtt{.==\ x\ y} \Rightarrow \mathtt{d}_{(\langle\mathtt{b,a}\rangle\mathtt{::Eq2})}\mathtt{.==\ y\ x}]$

Using the **(inst-tactic)** results in four proof obligations, one for each instance definition. In the proof goal for the fourth instance definition, a hypothesis is assumed for a type that is not structurally smaller (`eq2list` is the translation of the fourth instance definition):

$\forall_{\mathtt{a,b,c}}\forall_{\langle\mathtt{b,a}\rangle\mathtt{::Eq2}}\forall_{\langle\mathtt{c,a}\rangle\mathtt{::Eq2}}$
$[\ \forall_{\mathtt{x}\in\mathtt{b}}\forall_{\mathtt{y}\in\mathtt{a}}[\mathtt{d}_{(\langle\mathtt{b,a}\rangle\mathtt{::Eq2})}\mathtt{.eq2\ x\ y} \Rightarrow \mathtt{d}_{(\langle\mathtt{a,b}\rangle\mathtt{::Eq2})}\mathtt{.eq2\ y\ x}]$
$\Rightarrow \forall_{\mathtt{x}\in\mathtt{c}}\forall_{\mathtt{y}\in\mathtt{a}}[\mathtt{d}_{(\langle\mathtt{c,a}\rangle\mathtt{::Eq2})}\mathtt{.eq2\ x\ y} \Rightarrow \mathtt{d}_{(\langle\mathtt{a,c}\rangle\mathtt{::Eq2})}\mathtt{.eq2\ y\ x}]$
$\Rightarrow \forall_{\langle(\mathtt{b,c}),[\mathtt{a}]\rangle\mathtt{::Eq2}}\forall_{\mathtt{x}\in[\mathtt{a}]}\forall_{\mathtt{y}\in(\mathtt{b,c})}$
$[\ \mathtt{eq2list\ d}_{(\langle\mathtt{b,a}\rangle\mathtt{::Eq2})}\ \mathtt{d}_{(\langle\mathtt{c,a}\rangle\mathtt{::Eq2})}\ \mathtt{x\ y} \Rightarrow \mathtt{d}_{(\langle(\mathtt{b,c}),[\mathtt{a}]\rangle\mathtt{::Eq2})}\ \mathtt{y\ x}$
$]$
$]$

3.3.5 Multiple class constraints

In the previous section, only one class constraint was considered at a time. However, properties can contain multiple class constraints that may share type variables. Because of this sharing, it is not always possible to prove a property with multiple class constraints by performing induction for each class constraint separately using **(inst-tactic)**, as shown in example 3.3.25. To solve this problem, a proof rule and tactic that take multiple class constraints into account are defined.

Example 3.3.25 (inst-tactic does not always suffice)

Consider the next two class definitions:

```
:: Tree a = Leaf | Node a (Tree a) (Tree a)

class f a where f :: a -> Bool

instance f Int where
   f x = x == x

instance f [a] | g a where
  f []     = True
  f (x:xs) = g x == g x

instance f (Tree a) | f a & g a where
   f Leaf         = True
   f (Node x l r) = f x == g x

class g a where g :: a -> Bool

instance g Int where
   g x = x == x

instance g (Tree a) | f a where
   g Leaf         = True
   g (Node x l r) = f x == f x

instance g [a] | g a & f a where
   g []     = True
   g (x:xs) = g x == f x
```

Given the property:

$\forall_{a}\forall_{a::f}\forall_{a::g}\forall_{x\in a}[\mathtt{d}_{(a::f)}\mathtt{.f\ x} = \mathtt{d}_{(a::g)}\mathtt{.g\ x}]$

(inst-tactic) can be applied, which yields among others the proof goal:

$\forall_{a}\forall_{[a]::g}\forall_{x\in[a]}[\{\ \mathtt{f = flist\ d}_{(a::g)}\ \}\mathtt{.f\ x} = \mathtt{d}_{(a::g)}\mathtt{.g\ x}]$

This goal has a non-simple class constraint, which can only be removed by dictionary expansion **(expand)**, resulting in:

$\forall_{a}\forall_{a::f}\forall_{a::g}\forall_{x\in[a]}\ [\ \{\ \mathtt{f = flist\ d}_{(a::g)}\ \}\mathtt{.f\ x}$
$\quad = \ \{\ \mathtt{g = glist\ d}_{(a::f)}\ \mathtt{d}_{(a::g)}\ \}\mathtt{.g\ x}\ ]$

By some reduction steps this can be transformed into:

$\forall_{a}\forall_{a::f}\forall_{a::g}\forall_{x\in[a]}[\mathtt{d}_{(a::g)}\ \mathtt{x == d}_{(a::g)}\ \mathtt{x} = \mathtt{d}_{(a::f)}\ \mathtt{x == d}_{(a::g)}\ \mathtt{x}]$

This proof obligation is true when $\mathtt{d}_{(a::f)}\ \mathtt{x} = \mathtt{d}_{(a::g)}\ \mathtt{x}$. Unfortunately, the induction scheme did not allow this hypothesis to be assumed. This is caused by the fact that type variables occur in multiple class constraints.

In the ideal situation, it would be possible to derive an induction scheme on the set of types that satisfy all class constraints. Unfortunately, this is impossible. In such an induction scheme we would, of course, never create a branch where there are no types that satisfy the class constraints. This, however, is an instantiation of the constraint set satisfiability problem (CS-SAT), which is checking if the set of types that satisfy a set of (class) constraints is empty. The general CS-SAT problem is undecidable. This leaves two options.

One possibility is to work for a restrictive case. Dennis Volpano [18] presented an algorithm for the CS-SAT problem that models classes as tree automata and computes the intersection, essentially combining multiple classes into one. Unfortunately, although the result would be very elegant, this only works for single-parameter, non-mutually recursive classes. In that case we are better off using structural induction on types.

The second approach is to just try all possible combinations of instance definitions for all class constraints. This is more ad-hoc (for example, the resulting goal may contain non-simple predicates) but works in all cases and yields good results in the most common cases (flat instance heads). Unfortunately, as we are unable to solve the CS-SAT problem, sometimes one has to prove properties for a set of class constraints that has no instances (example 3.3.35).

The latter approach is comparable to a more recently proposed algorithm by Camarão, Figueredo, and Vasconcellos [2] to solve CS-SAT without restrictions (most importantly multi-variable classes and mutually recursive instances are allowed). Essentially, it tries to generate a type by trying all possible combinations of instance definitions for all class constraints. To cope with the undecidability, a limit on the number of recursive calls is used. Thus, it sometimes fails to give an answer.

Here, a tactic that uses this method will be defined. First, the set of types that satisfy multiple class constraints is defined. Then, a partial order on these types is defined. Finally, the proof rule and tactic are given.

First, the set of type sequences that are instances of all classes that occur in a list of class constraints $\vec{\pi}$ is defined. $\vec{\tau}$ is a member of this set if all class constraints $\vec{\pi}$ are satisfied when all variables $TV(\vec{\pi})$ are replaced by the corresponding type from $\vec{\tau}$. It is assumed here that $TV(\bar{\pi})$ is a linearly ordered, for example lexicographically, sequence and that the elements of $\bar{\tau}$ are in the corresponding order.

Definition 3.3.26 (types that satisfy multiple class constraints)
For all programs $\psi \in \Psi$ and simple predicates $\vec{\pi} \in \langle \mathcal{P}_s \rangle$, the set of types that satisfy a set of class constraints the function $\mathit{SetInstances}_\psi :: \langle \mathcal{P}_s \rangle \hookrightarrow \langle \mathcal{T}^{\mathsf{closed}} \rangle$ is defined by:

$$\mathit{SetInstances}_\psi(\vec{\pi}) = \{\vec{\tau} \mid \forall_{(\vec{\alpha}' :: c) \in \vec{\pi}}[*_{(TV(\vec{\pi}) \leftrightarrow \vec{\tau})}(\vec{\alpha}') \in \mathit{Instances}_\psi(c)]\}$$

Example 3.3.27 (types that satisfy multiple class constraints)
Consider the set of instances of both classes `f` and `g` (example 3.3.25):

$$\mathit{SetInstances}_\psi(\mathtt{a} :: \mathtt{f}, \mathtt{a} :: \mathtt{g}) = \{\mathtt{Int}, \mathtt{[Int]}, \mathtt{Tree\ Int}, \mathtt{[[Int]]}, \ldots\}$$

The order on this set is an extension of the order for single class constraints to sets. A sequence of types $\vec{\sigma}$ is considered one step smaller than $\vec{\tau}$

if $*_{(TV(\vec{\pi})\leftrightarrow\vec{\sigma})}(\vec{\pi})$ is a subset of the dependencies of $*_{(TV(\vec{\pi})\leftrightarrow\vec{\tau})}(\vec{\pi})$.

Definition 3.3.28 (partial order for multiple class constraints)
For all programs $\psi \in \Psi$ and simple predicates $\vec{\pi} \in \langle \mathcal{P}_s \rangle$, the partial order $<^1_{(\psi,\vec{\pi})} \subseteq \mathit{SetInstances}_\psi(\vec{\pi}) \times \mathit{SetInstances}_\psi(\vec{\pi})$ is defined by:

$$\vec{\sigma} <^1_{(\psi,\vec{\pi})} \vec{\tau} \Leftrightarrow *_{(TV(\vec{\pi})\leftrightarrow\vec{\sigma})}(\vec{\pi}) \subseteq \bigcup_{\pi\in\vec{\pi}}[\mathit{Deps}^1_\psi(*_{(TV(\vec{\pi})\leftrightarrow\vec{\tau})}(\pi))])$$

Example 3.3.29 (partial order for multiple class constraints)
For the classes in example 3.3.25 we have for example:

$$\texttt{[Int]} <^1_{(\psi,\langle \texttt{a::f},\texttt{a::g}\rangle)} \texttt{[[Int]]}$$

because:

$$\{\texttt{[Int]} :: \texttt{f}, \texttt{[Int]} :: \texttt{g}\} \subseteq \mathit{Deps}^1_\psi(\texttt{[[Int]]} :: \texttt{g}) \cup \mathit{Deps}^1_\psi(\texttt{[[Int]]} :: \texttt{f})$$

Again, the reflexive transitive closure of this order, $\leq_{(\psi,\vec{\pi})}$, is a well-founded partial order (see C.2).

Definition 3.3.30 (well-founded induction on multiple instances)
For all programs $\psi \in \Psi$ and simple predicates $\vec{\pi} \in \langle \mathcal{P}_s \rangle$, well-founded induction on $\mathit{SetInstances}_\psi(\vec{\pi})$ is defined by:

$$\frac{\forall_{\vec{\tau}\in \mathit{SetInstances}_\psi(\vec{\pi})}[\forall_{\vec{\sigma}<^1_{(\psi,\vec{\pi})}\vec{\tau}}[P(\vec{\sigma})] \rightarrow P(\vec{\tau})]}{\forall_{\vec{\tau}\in \mathit{SetInstances}_\psi(\vec{\pi})}[P(\vec{\tau})]}$$

Because multiple class constraints are involved, defining the final tactic is a bit more complicated. Instead of all instance definitions, every combination of instance definitions, one for each class constraint, has to be tried. All of these instance definitions make assumptions about the types of the type variables, and these assumptions should be unifiable. Therefore, we define the most general unifier that takes the sharing of type variables across class constraints into account:

Definition 3.3.31 (most general unifier for sets of class constraints)
For all programs $\psi \in \Psi$, the function $\mathit{SetMgu} :: \langle \mathcal{P}_s \rangle \langle \mathcal{T}^{\mathsf{closed}} \rangle \hookrightarrow \circledast$ is defined by:

$$\begin{aligned} &\mathit{SetMgu}(\langle \vec{\alpha}_1 :: c_1, \ldots, \vec{\alpha}_n :: c_n\rangle, \langle \vec{\tau}_1, \ldots, \vec{\tau}_n\rangle) = * \Leftrightarrow \\ &\qquad \forall_{1\leq i\leq n}[*(\vec{\alpha}_i) = \vec{\tau}_i] \;\wedge\; \forall_{*'}[\forall_{1\leq i\leq n}[*'(\vec{\alpha}_i) = \vec{\tau}_i] \Rightarrow \exists *''[*' = *'' \circ *]] \end{aligned}$$

Furthermore, for readability of the final tactic, some straightforward extensions of existing definitions to sequences are used.

Definition 3.3.32 (extensions for sequences)
Sequence versions of instance definitions, instance head, and dependencies are defined by:

$$\begin{aligned} \mathit{InstDefs}_\psi(\langle \pi_1, \ldots, \pi_n\rangle) &= \{i_1, \ldots, i_n \mid i_j \in \mathit{InstDefs}_\psi(\pi_j)\} \\ \mathit{Head}(\langle i_1, \ldots, i_n\rangle) &= \langle \mathit{Head}(i_1), \ldots, \mathit{Head}(i_n)\rangle \\ \mathit{Deps}_\psi(\langle \pi_1, \ldots, \pi_n\rangle, \langle i_1, \ldots, i_n\rangle) &= \langle \mathit{Deps}_\psi(\pi_1, i_1), \ldots, \mathit{Deps}_\psi(\pi_n, i_n)\rangle \end{aligned}$$

Finally, using the presented definitions, evidence creation, class constrained properties, and the proof rule, the tactic can be defined (see C.4).

Tactic 3.3.33 (multi-tactic)
For all programs $\psi \in \Psi$, simple predicates $\pi \in \langle \mathcal{P}_s \rangle$, **(multi-tactic)** is defined by:

$$\begin{array}{l}
\forall_{\vec{\imath} \in \mathit{InstDefs}_\psi(\vec{\pi})} \exists_{* \mid * = \mathit{SetMgu}(\vec{\pi}, \mathit{Head}(\vec{\imath}))} \forall_{*(\mathit{Head}(\vec{\imath})) \in \langle \mathcal{T}^{\mathsf{closed}} \rangle} \forall_{\mathit{Deps}^1_\psi(*(\vec{\pi}), \vec{\imath})} \\
\quad [\ \forall_{*' \mid *'(\vec{\pi}) \subseteq \mathit{Deps}_\psi(*(\vec{\pi}), \vec{\imath})} [P(d_{(*'(\vec{\pi}))}, *'(TV(\vec{\pi})))] \\
\quad\ \ \Rightarrow P(\mathit{Dict}^1_\psi(*(\vec{\pi}), \vec{\imath}), *(\mathit{Head}(\vec{\imath}))) \\
\quad] \\
\hline
\forall_{TV(\vec{\pi})} \forall_{\vec{\pi}} [P(d_{(\vec{\pi})}, TV(\vec{\pi}))]
\end{array}$$

Example 3.3.34 (problem solved)
Using the newly defined tactic the property from example 3.3.25 can be proven.

$\forall_{\mathtt{a}} \forall_{\mathtt{a::f}} \forall_{\mathtt{a::g}} \forall_{\mathtt{x} \in \mathtt{a}} [\mathtt{d}_{(\mathtt{a::f})}\mathtt{.f\ x} = \mathtt{d}_{(\mathtt{a::g})}\mathtt{.g\ x}]$

Applying **(multi-tactic)** yield three proof goals, one for every unifiable combination of instance definitions:

- $\forall_{\mathtt{x} \in \mathtt{Int}} [\mathtt{fint\ x} = \mathtt{gint\ x}]$
- $\begin{array}{l} \forall_{\mathtt{a}} \forall_{\mathtt{a::f}} \forall_{\mathtt{a::g}}\ [\ \forall_{\mathtt{x} \in \mathtt{a}} [\mathtt{d}_{(\mathtt{a::f})}\mathtt{.f\ x} = \mathtt{d}_{(\mathtt{a::g})}\mathtt{.g\ x}] \\ \qquad\qquad \Rightarrow \forall_{\mathtt{x} \in [\mathtt{a}]}\ [\ \{\ \mathtt{f = flist\ d}_{(\mathtt{a::g})}\ \}\mathtt{.f\ x} \\ \qquad\qquad\qquad\qquad = \{\ \mathtt{g = glist\ d}_{(\mathtt{a::f})}\ \mathtt{d}_{(\mathtt{a::g})}\ \}\mathtt{.g\ x}\ ] \\ \qquad\quad] \end{array}$
- $\begin{array}{l} \forall_{\mathtt{a}} \forall_{\mathtt{a::f}} \forall_{\mathtt{a::g}}\ [\ \forall_{\mathtt{x} \in \mathtt{a}} [\mathtt{d}_{(\mathtt{a::f})}\mathtt{.f\ x} = \mathtt{d}_{(\mathtt{a::g})}\mathtt{.g\ x}] \\ \qquad\qquad \Rightarrow \forall_{\mathtt{x} \in \mathtt{Tree\ a}}\ [\ \{\ \mathtt{f = ftree\ d}_{(\mathtt{a::f})}\ \mathtt{d}_{(\mathtt{a::g})}\ \}\mathtt{.f\ x} \\ \qquad\qquad\qquad\qquad\quad = \{\ \mathtt{g = gtree\ d}_{(\mathtt{a::f})}\ \}\mathtt{.g\ x}\ ] \\ \qquad\quad] \end{array}$

The third goal now contains an hypothesis that can be used to prove the property.

Note that using this proof rule might result in a goal that contains non-simple class constraints. However, if all instance definitions have flat instance types (which is very likely) and there are no subclasses, all created class constraints are simple. Another problem might be that it is not noted if there are no types that satisfy all class constraints (example 3.3.35).

Example 3.3.35 (empty intersection)
Consider the next class definitions:

```
class A a :: a -> a
instance A Int       where A = AInt
instance A [a] | A a where A = AList

class B a :: a -> a
instance B Real      where B = BReal
instance B [a] | B a where B = BList
```

Although both classes have an instance definition for lists, the classes `A` and `B` have no common instances. The **(multi-tactic)** will not note this and generate a proof goal for lists when applied to:

$\forall_{\mathtt{a}} \forall_{\mathtt{a::A}} \forall_{\mathtt{a::B}} [P(\mathtt{a})]$

3.3.6 Handling non-simple class constraints

The proof rules in the previous two subsections only work for properties with simple class constraints. For non-simple class constraints another solution has to be found. The most straightforward solution is introducing fresh variables in the class constraints. Predicates are added to enforce that the new type variables can be unified with the types they are substituted for.

Example 3.3.36 (handling non-simple class constraints)

A property with a non-simple class constraint:

$\forall_{\mathtt{a}}\forall_{\mathtt{[a]::c}}[P(\mathtt{d}_{(\mathtt{[a]::c})})]$

can be written as:

$\forall_{\mathtt{a,b}}\forall_{\mathtt{b::c}}[\mathtt{b\ =\ [a]} \Rightarrow P(\mathtt{d}_{(\mathtt{[a]::c})})]$

where `b` = `[a]` is true if and only if `b` and `[a]` can be unified.

This way, the non-simple class constraints can be transformed into simple class constraints and the proof rules from the previous subsection can be applied. This will not be formalized, because it adds a new concept and it is only required in the rare cases where induction on multiple class constraints generates goals with non-simple class constraints.

3.3.7 Comparison with Isabelle

The solution for proving overloaded properties in the proof assistant Isabelle was presented in section 2.10. Here, that solution is compared to the proposal in this chapter on three aspects: the type system, the induction scheme, and the tactic.

To support overloading and type classes, Isabelle's type system was explicitly extended with a partially ordered layer of sorts. Core's type system was not explicitly altered; predicated types were only used to represent class and instance definitions. Nevertheless, in section 4.2 it is explained how Core's type inference function had to be extended when implementing the tactic.

The induction scheme used in Isabelle is structural induction on types. Here, it was shown that this does not really suffice for multi-parameter classes. Instead, an induction scheme on (a subset of) types is used that is derived from the instance definitions.

Isabelle only supports single parameter classes with non-overlapping instances and flat instance heads. The tactics defined in this chapter support multi-parameter classes, overlapping instances, and non-flat instance heads. This complicated the tactic a bit; the tactic should explicitly generate meaningful proof goals and hence expand all dictionaries at least one step.

In summary, the **(multi-tactic)** presented here is more general and supports more extensions of type classes than the one used in Isabelle.

4 Implementation in Sparkle

In this chapter, the implementation of the specification and tactics in SPARKLE is discussed. There were three distinct parts of SPARKLE that had to be extended. First, the specification mechanism was generalized to handle class constrained properties. Then, the type system was extended to handle properties with universally quantified dictionaries well. Finally, the tactic for class constrained properties was implemented and added to the hints mechanism.

In section 4.1, 4.2, and 4.3, the three parts of the implementation are explained. For each part it is described what has been implemented, why this is necessary, and how it has been implemented. Section 4.4 evaluates the implementation and the implementation process.

4.1 Specifying properties

The first implementation task was to enable the specification of class constrained properties in SPARKLE. This turned out to be relatively easy.

SPARKLE already used the original class and instance definitions to create a dictionary when a member or overloaded function was applied. However, this was only allowed when the entire dictionary value could be generated at the time of specification. Hence, no variables occurred in the dictionary values and quantification over dictionaries never occurred.

For cases in which the dictionary cannot be fully generated an extension was implemented. When a dictionary or dependency is required that cannot be generated, a variable is introduced for it and universally quantified over. Afterwards, if two or more of the introduced variables are of the same type, and hence represent the same dictionary, one of them is replaced by the other. Furthermore, care is taken that in all non class constrained properties SPARKLE behaves exactly as before.

Considering this was the first extension, and thus included the overhead of getting to know SPARKLE's code, the implementation was very straightforward and took only a few days. The only real problem was finding out where the dictionaries were generated.

Until here, we have assumed that a dictionary type only contains valid dictionaries. In section 3.2 this assumption was motivated. However, it is also described that an additional predicate can remove the need for this assumption. This predicate has not been implemented, because it would have taken too much time for a proof of concept and clutters the properties. Nevertheless, either it should be verified that the assumption does not yield any unanticipated problems or the predicate should be implemented, and maybe hidden from the user.

Example 4.1.1 (specifying properties)
The example from section 1.3 can now be viewed from another perspective. Before implementing the extension in Sparkle, the following property could be defined:

$\forall_{\texttt{x}\in[\texttt{Int}]}\forall_{\texttt{y}\in[\texttt{Int}]}[\texttt{eqlist eqint x y} \Rightarrow \texttt{eqlist eqint y x}]$

However, it was not possible to specify, symmetry of equality in general:

$\forall_{\texttt{a}|\texttt{Eq a}}\forall_{\texttt{x}\in\texttt{a}}\forall_{\texttt{y}\in\texttt{a}}[\texttt{x == y} \Rightarrow \texttt{y == x}]$

because the dictionary of `Eq` for type `a` cannot be generated yet. The extension generates universal quantification over the possible dictionaries in this case. Hence, when specifying `x == y => y == x`, the following property is generated:

$\forall_{\texttt{a}}\forall_{\texttt{d}\in\texttt{dicteq a}}\forall_{\texttt{x}\in\texttt{a}}\forall_{\texttt{y}\in\texttt{a}}[\texttt{d.== x y} \Rightarrow \texttt{d.== y x}]$

4.2 Typing dictionaries

Most of the implementation time was spent on altering the type inference algorithm in Sparkle. The types of expressions in class constrained properties, more specifically the types of the dictionary variables, are required for applying the tactic. Beforehand, it seemed a major problem that Sparkle throws away all typing information. Adding types to the internal representation would have been a huge undertaking. Other tactics, for example the `Induction` tactic, infer typing information when required. Our tactic could also do this but, as discussed in section 2.8, typing dictionaries sometimes requires rank-2 polymorphism, which is currently not available in Sparkle. Fortunately, it turned out to be possible to hide the higher-level polymorphism in an available abstract datatype (example 4.2.1).

Example 4.2.1 (hiding rank-2 polymorphism for dictionaries)
In example 2.8.3, a dictionary expressions is shown where function `g` should have type `({ f :: ∀`$_{\texttt{b}}$` Bool b -> b} -> (Int, Char))`. This type requires rank-2 polymorphism, but can also be expressed using an abstract datatype for the dictionary `(dictCopy Bool)`. If all cases in which the rank-2 polymorphism may be required can be adjusted to work with the abstract datatype, rank-2 polymorphism is not required. The type of `g`, for example, would be `((dictCopy Bool) -> (Int, Char))`.

Three operations on dictionaries were identified for which the type inference system was altered to derive a type using the abstract datatype and the class definition: dictionary creation, dictionary selection, and reduced dictionary selection.

4.2.1 Dictionary creation

The first situation for which the type system was altered is the creation of dictionaries. Dictionaries are created by a dictionary constructor followed by the values of the members and subclasses. Whenever the typing system encounters a dictionary constructor, we know the result of the expression is the abstract dictionary datatype. The types of the values for members and subclasses can be derived from the class definition.

Example 4.2.2 (typing dictionary creation)

Consider the creation of a dictionary of the `Eq` class:

```
create_dictionary_==; eqint
```

From the class definition of `Eq` it can be derived that the type of this expression is of the form `(dicteq a)` and that the type of `eqint` is of the form `(a a -> Bool)`. From this, it will be inferred that `a` should be an `Int`.

4.2.2 Dictionary selection

The second case for which an addition was required is the selection of a field from a dictionary. A field is selected from the dictionary by applying a dictionary selector function to the dictionary value (which can be a variable). Whenever a dictionary selector is encountered, we know that the following value is a dictionary. The type of the selected function or subclass can then be derived from the class definition.

Example 4.2.3 (typing dictionary selection)

Consider the selection of the `==` field from a dictionary of the `Eq` class:

```
_dictionary_Eq;_select_== d x y
```

From the name of the selector function it can be derived that `d` is a dictionary of the `Eq` class and thus has a type of the form (`eqdict a`). Furthermore, since the `==` field is selected, the type of `(_dictionary_Eq;_select_== d)` will be of the form `(a a -> Bool)`. Because `d` is a variable, the type that `a` stands for cannot be derived from this expression.

4.2.3 Reduced dictionary selection

The selection functions are reduced to a pattern match. In that case, essentially the member and subclass values are assigned to new symbols and used in an expression. The types of these symbols can be derived from the class definition just like for the dictionary selection function.

Example 4.2.4 (typing reduced dictionary selection)

The selection of the `==` field from a dictionary of the instance for `Int` of `Eq` can be reduced to:

```
case (create_dictionary_==; eqint) of
    { ==s } -> ==s x y
```

Again, from the name of the selector function it can be derived that `eqint` is a dictionary of the `Eq` class and thus has a type of the form `(eqdict a)`. Furthermore, since this value is assigned to the `==s` symbol, the type of `==s` is of the form `(eqdict a)` as well. From this, it will be inferred that `a` should be an `Int`.

4.3 Tactic

The tactic implemented in Sparkle is a version of the multi-tactic that is called `TypeClassInduction`. By default, it uses all universally quantified dictionaries. The tactic is added to the hints mechanism and preferred over the `Induction` tactic, because the `Induction` tactic would not work for dictionaries anyway. One might argue that the tactic should be combined with the `Induction` tactic, because it can also be considered induction in dictionaries. However, eventually we would like to hide the dictionaries and present them as class constraints.

Implementing the tactic was very straightforward, because its structure was equal to the structure of the available `Induction` tactic. Also, Leonard Lensink had previously implemented a tactic for Sparkle [13] and left marks at the places were modifications were required.

4.4 Evaluation

The implementation took less time than anticipated. The total amount of time spent on it is about three weeks. Most of the time was lost in fixing mistakes in the additions to the type inference function. Despite its 130.000 lines of code (including libraries) the code remains clear and easy to understand.

One of the aspects of this work was to stay close to Clean syntax, hence type classes. For the specification, this has been achieved. Properties can be specified using Clean syntax. When proving such a property however, the user works directly with dictionaries. Special options in Sparkle to disable the hiding of dictionaries, creator and selector functions, and instance suffixes have to be used. Ideally, the use of dictionaries should be completely hidden from the user. However, since the proof rule is directly linked to the dictionaries, an alternative representation of dictionaries that looks more like type classes has to be created. Furthermore, when dictionaries are hidden, the dictionary selection should be reduced automatically whenever possible. There was no time to investigate this further.

In section 2.9 it was mentioned that Clean supports derived members and specialized functions. The derived members are just macros that are always expanded, also when specifying a property. Hence, no special care had to be taken of them. Specialized functions are functions that implement a class member for a specific instance, built for reasons of efficiency. Once a property has been proven for all class instances, it also holds for the specialized instances. Currently, this cannot be derived, hence the property has to be proven for each specialized function again.

The tactic has been used to prove, amongst others, the examples in this paper. Hence, this tactic is a useful addition to Sparkle. Nevertheless, further experience with proofs of class constrained properties, made possible by this tactic, may very well indicate that it needs to be altered to better suit some unanticipated proofs. The implementation, including the stored proofs, is available at: `http://www.student.kun.nl/ronvankesteren/SparkleGTC.zip`

Conclusion 5

Proving properties about a program can significantly help creating high quality reliable software. In this project we set out to solve one of problems that are encountered when proving a property of a function program; how can properties about overloaded functions, called class constrained properties, be specified and proven.

In the presented solution, class constrained properties are specified at the CORE level; SPARKLE's existing formal framework in which overloading has been translated to dictionaries. To be able to formalize these dictionaries, the CORE framework was extended with class and instance definitions.

A new proof rule for class constrained properties and an effective tactic based on it were presented. Although structural induction on types is theoretically powerful enough, it was shown that for an effective tactic, one that allows all sensible hypotheses to be assumed, an induction scheme should be used that is based on the instance definitions. The proof rule and tactic were first defined for single class constraints and then generalized to properties with multiple class constraints. The resulting tactic is general; although specialized for CLEAN, it can also be used for proving properties about overloaded expressions in other languages, such as HASKELL and ISABELLE.

As a proof of concept, the resulting tactic has been implemented in the proof assistant SPARKLE. The implementation took less time than anticipated and has been used to prove a number of properties. These proofs are available together with the implementation. However, there still remains some work to be done to make the tactic more user friendly.

We have compared our solution to the proof rule in ISABELLE, which uses structural induction on types. If ISABELLE supported multi-parameter type classes, it would be useful to implement the tactic in ISABELLE as well. Currently however, this work does not have an advantage for ISABELLE because ISABELLE only supports single parameter type classes, for which structural induction suffices.

5.1 Future work

As a direct continuation of this work, the implementation of the tactic could be improved to, to the user, operate more at the level of type classes. Furthermore, it should be investigated how the tactic behaves in practice. Maybe another tactic is required to prove all sensible properties.

As mentioned in section 2.10, the general proof assistant Isabelle [15] supports overloading and single parameter type classes. Isabelle's notion of type classes is somewhat different from Clean's in that it represents types that satisfy certain properties instead of types for which certain values are defined. Nevertheless, the problems to be solved are equivalent. Isabelle uses a proof rule based on structural induction on types [14, 20], which suffices for the supported type classes. However, if Isabelle would support more extensions, most importantly multi parameter classes, it would be useful to define our proof rule and a corresponding tactic in Isabelle.

Essentially, the implementation of the tactic we proposed extends the induction techniques available in Sparkle. Leonard Lensink proposed and implemented extensions of Sparkle for induction and co-induction for mutually recursive functions and data types [13]. The main goal was to ease proofs by making the induction scheme match the structure of the program. Together with this work this significantly increases the applicability of Sparkle.

Because generics is often presented as an extension of type classes [7], it would be nice to extend this work to generics as well. Currently, in Clean generics are translated to normal type classes where instances are created for every available data type [1]. There is a library for Haskell that generates classes with boilerplate code for every available data type [12]. The tactic presented here can already be used to prove properties about generic functions by working on these generated type classes. However, the property is only proven for the data types that are used in the program and a separate proof is required for each data type. That is, after all, the main difference between normal type classes and generics. Hence, it remains useful to define a proof rule specifically for generics.

Bibliography

[1] A. Alimarine and R. Plasmeijer. A generic programming extension for Clean. In Th. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers*, LNCS 2312, pages 168–185, Älvsjö, Sweden, September 24-26 2001. Springer.

[2] C. Camarão, L. Figueiredo, and C. Vasconcellos. Constraint-set satisfiability for overloading. In *International Conference on Principles and Practice of Declarative Programming*, Verona, Italy, August 2004.

[3] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 170–181. ACM Press, 1992.

[4] M. de Mol. *Sparkle*. PhD thesis, Radboud University Nijmegen, (in progress).

[5] M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - SPARKLE: A functional theorem prover. In Th. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers*, LNCS 2312, pages 55–71, Älvsjö, Sweden, September 2001.

[6] M. J. C. Gordan and T. F. Melham. *Introduction to HOL: A theroem proving environment for higher order logic*. Cambridge University Press, 1993.

[7] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2000.

[8] M. P. Jones. A theory of qualified types. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582, pages 287–306. Springer, New York, N.Y., 1992.

[9] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, N.Y., 1993. ACM Press.

[10] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*, LNCS 1782, pages 230–244. Springer, 2000.

[11] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Proceedings of the Second Haskell Workshop*, Amsterdam, June 1997.

[12] R. Lämmel and S. Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37, New Orleans, Januari 2003. ACM.

[13] L. Lensink and M. van Eekelen. Induction and Co-induction in Sparkle. In Hans-Wolfgang Loidl, editor, *Fifth Symposium on Trends in Functional Programming (TFP 2004)*, pages 273–293. Ludwig-Maximilians Universität, München, November 2004.

[14] T. Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. CUP, 1993.

[15] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer, 2002.

[16] C. Stratchey. Fundamental concepts in programming languages. In *Lecture notes for International Summer SChool in Computer Programming*, Copenhagen, 1967.

[17] M. van Eekelen and R. Plasmeijer. *Concurrent Clean Language Report (version 2.0).* University of Nijmegen, December 2001.

[18] D. M. Volpano. Haskell-style overloading is NP-hard. In *Proceedings of the International Conference on Computer Languages*, pages 88–95, Toulouse, France, 1994.

[19] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[20] M. Wenzel. Type classes and overloading in higher-order logic. In E. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, pages 307–322, Murray Hill, New Jersey, 1997.

Mathematical conventions A

Most of the notation and conventions used in this thesis are equivalent or based on the ones used in the description of the mathematical framework of Sparkle [4]. The most important ones are listed in section A.1. Section A.2 defines substitutors and most general unifiers.

A.1 Notation

Set theoretical notation Functions, predicates and relations are often defined using a set theoretical notation:

- Functions f from X to Y are defined by:
  $\{(x, y) \mid x \in X, y \in Y, f(x) = y\}$
- A predicate P on X is defined by:
  $\{x \mid x \in X, P(x)\}$
- A relation R on $X_1 \times \ldots \times X_n$ is defined by:
  $\{(x_1, \ldots, x_n) \mid x_1 \in X_1, \ldots, x_n \in X_n, R(x_1, \ldots, x_n)\}$
- The powerset of X is denoted by: $\wp(X)$

Overloading This thesis is about overloading, but for reasons of convenience overloading will also be used in the thesis itself. Functions $f_1, \ldots, f_n$ with domains $X_1, \ldots, X_n$ and ranges $Y_1, \ldots, Y_n$ are considered a function $f = f_1 \cup \ldots \cup f_n$ with domain $X_1 \cup \ldots \cup X_n$ and range $Y_1 \cup \ldots \cup Y_n$.

Ordered sequences Ordered sequences are denoted by $\langle x_1, \ldots, x_n \rangle$. An ordered sequence of zero length is denoted by $\langle\rangle$. The type of ordered sequences is denoted as $\langle \mathcal{X} \rangle$ where $\mathcal{X}$ is the type of the sequence elements.

- For all sets S, sequence concatenation $\circ :: \langle S \rangle\ \langle S \rangle \rightarrow \langle S \rangle$ is defined by:
  $\langle x_1, \ldots, x_n \rangle \circ \langle y_1, \ldots, y_m \rangle = \langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$
- For all sets S, the relation $\in\ \subseteq S \times \langle S \rangle$ is defined by:
  $x \in \langle x_1, \ldots, x_n \rangle \Leftrightarrow \exists_{1 \leq i \leq n}[x_i = x]$

- For all sets S, the relation $\subseteq\, \subseteq \langle S\rangle \times \langle S\rangle$ is defined by:
  $X \subseteq Y \Leftrightarrow \forall_{x\in S}[x \in X \Rightarrow x \in Y]$

Minimal value For all predicates $P(i)$, the function min_i is defined by:
$min_i(P(i)) = j \Leftrightarrow P(j) \wedge \forall_{k\in\mathbb{N}}[P(k) \Rightarrow j < k]$

A.2 Substitutors

A substitutor is an operation that substitutes types for type variables.

Definition A.2.1 (substitutor on types)
A substitutor $* :: \mathcal{T} \rightarrow \mathcal{T}$ is an operation such that:

$$\begin{array}{ll} *(\sigma \rightarrow \tau) & = *(\sigma) \rightarrow *(\tau) \\ *(\mathsf{basic}\ b) & = \mathsf{basic}\ b \\ *(c\ \langle \tau_1, \ldots, \tau_n\rangle) & = c\ \langle *(\tau_1), \ldots, *(\tau_n)\rangle \\ *(\alpha\ \langle \tau_1, \ldots, \tau_n\rangle) & = *(\alpha)\ \langle *(\tau_1), \ldots, *(\tau_n)\rangle \end{array}$$

The *support* of $*$ is defined by $sup(*) = \{\alpha | *(\alpha) \neq \alpha\}$. Usually the support is finite, in which case we write $* = [\alpha_1 := *(\alpha_1), \ldots, \alpha_n := *(\alpha_n)]$, where $\{\alpha_1, \ldots, \alpha_n\}$ is the support of $*$.

A substitutor can also be straightforwardly applied to sequences or sets of types, predicates, class constraints, and class constrained properties.

Definition A.2.2 (substitutor on lists of types)
The set of all substitutors is denoted by $\circledast$. Hence, $* \in \circledast$ denotes that $*$ is a substitutor.

A special kind of substitutor is the *most general unifier*.

Definition A.2.3 (unifier)
A *unifier* for types $\vec{\sigma}$ and $\vec{\tau}$ is a substitutor $* \in \circledast$ such that $*(\vec{\sigma}) = *(\vec{\tau})$.

Definition A.2.4 (most general unifier)
The *most general unifier* for types $\vec{\sigma}$ and $\vec{\tau}$ is a substitutor $* \in \circledast$ such that:

(*i*) $*(\vec{\sigma}) = *(\vec{\tau})$, and
(*ii*) $\forall_{*'\in\,\circledast}[*'(\vec{\sigma}) = *'(\vec{\tau}) \Rightarrow \exists_{*''\in\,\circledast}[*' = *'' \circ *]]$.

Definition A.2.5 (most general unifier shorthand)
The most general unifier for types $\vec{\sigma}$ and $\vec{\tau}$ is denoted by $*_{(\vec{\sigma}\leftrightarrow\vec{\tau})}$.

The Core framework B

This appendix presents the formal mathematical framework called Core that formalizes the reasoning in Sparkle. Core consists of a programming language, a logic language, and a proof language.

The *programming language*, presented in section B.1, formally describes Clean programs. Program properties are stated in a *logic language*, which is described in section B.2. The *proof language*, used to describe the reasoning steps performed in proving a property, is defined in section B.3.

B.1 Programming language

This section formally introduces the programming language of the Core framework. Only the essential Clean concepts are included in Core, hence a translation from Clean to Core is required. Fortunately, the Clean compiler uses an internal language similar to Core, so the compiler can be used for this translation, adding to the validity of Sparkle proofs.

Here we use a simplified version of the specification by Maarten de Mol [4]; some parts are left out, because they are not necessary for our purpose. First, the types and values available in Core are defined. Then, expressions that operate on these values are specified. After that, the set of programs is defined. Finally, the semantics and welltypedness are defined.

B.1.1 Types and values

Clean includes the predefined basic types `Int`, `Real`, `Char`, `Bool`, and `String`. These basic types are also available in Core.

Definition B.1.1 (basic types)
The set of basic types B_t is defined by:
$B_t = \{\mathsf{tbool}, \mathsf{tint}, \mathsf{tchar}, \mathsf{treal}, \mathsf{tstring}\}$

Definition B.1.2 (basic values)
B_v is the set of values of a basic type.

In both Clean and Core it is possible to define algebraic datatypes. However, Clean's (predefined) tuple, list, array and dictionary types are all considered algebraic datatypes in Core. In the next example, an algebraic data type is defined.

Example B.1.3 (algebraic datatype)

In Clean, the `Tree` algebraic datatype is defined by:

```
:: Tree a = Node (Tree a) (Tree a)
          | Leaf a
```

In example B.1.3, the symbol `Tree` is an *algebraic type symbol*, which can be used to construct types from smaller types. Each algebraic type symbol expects a fixed number of argument types, called its *arity*. `Tree` expects one argument (`a`) and therefore has arity one.

Definition B.1.4 (algebraic type symbols)
$\mathcal{S}_a$ is the set of algebraic type symbols.
The function $Arity :: \mathcal{S}_a \to \mathbb{N}$ returns the arity of an algebraic type symbol.

What algebraic type symbols are for types, *constructor symbols* are for values. New values can be created by combining a constructor and a number of simpler values. Constructors also have a fixed arity. In example B.1.3, `Node` and `Leaf` are constructors of arity two and one respectively.

Definition B.1.5 (constructor symbols)
$\mathcal{S}_c$ is the set of constructor symbols.
The function $Arity :: \mathcal{S}_c \to \mathbb{N}$ returns the arity of a constructor symbol.

The `a` appearing in example B.1.3 is a *type variable*. Type variables are placeholders for which a meaningful type is to be substituted later. In formal definitions, they are generally denoted by Greek letters. Instead of having an arity, a type variable is of a certain *kind*, which is the number of arguments that it must be applied to.

Definition B.1.6 (type variables)
$\mathcal{V}_t$ is the set of type variables.
The function $Kind :: \mathcal{V}_t \to \mathbb{N}$ returns the kind of a type variable.

Now, the set of types supported by Core can be defined. The *variable types* are type variables of kind 0. The *symbol types* are constructed by applying an algebraic type symbol to a list of types. *Function types* consist of the type of the argument and the result type of a function. Type variables of kind > 0 applied to a list of arguments are called *application types*. These five kinds of types define the complete set of types available in Core.

Definition B.1.7 (types)
The set of types $\mathcal{T}$ is defined by:

$$\begin{array}{lll}
\mathcal{T} = \{\alpha & \mid \alpha \in \mathcal{V}_t, Kind(\alpha) = 0\} & \text{(variable types)} \\
\quad \cup\ \{b & \mid b \in B_t\} & \text{(basic types)} \\
\quad \cup\ \{a\ \vec{\tau} & \mid a \in \mathcal{S}_a, \vec{\tau} \in \langle \mathcal{T} \rangle, 0 < |\vec{\tau}| \leq Arity(a)\} & \text{(symbol types)} \\
\quad \cup\ \{\sigma \to \tau & \mid \sigma \in \mathcal{T}, \tau \in \mathcal{T}\} & \text{(function types)} \\
\quad \cup\ \{\alpha\ \vec{\tau} & \mid \alpha \in \mathcal{V}_t, \vec{\tau} \in \langle \mathcal{T} \rangle, 0 < |\vec{\tau}| \leq Kind(\alpha)\} & \text{(application types)}
\end{array}$$

Definition B.1.8 (free variables in types)
The function $TV :: \mathcal{T} \rightarrow \wp(\mathcal{V}_t)$ is defined by:

$$\begin{aligned} &TV(\alpha) &&= \{\alpha\} \\ &TV(b) &&= \emptyset \\ &TV(a\ \langle\tau_1, \ldots, \tau_n\rangle) &&= TV(\tau_1) \cup \ldots \cup TV(\tau_n) \\ &TV(\sigma \rightarrow \tau) &&= TV(\sigma) \cup TV(\tau) \\ &TV(\alpha\ \langle\tau_1, \ldots, \tau_n\rangle) &&= \{\alpha\} \cup TV(\tau_1) \cup \ldots \cup TV(\tau_n) \end{aligned}$$

Definition B.1.9 (free variables in lists of types)
The function $TV :: \langle\mathcal{T}\rangle \rightarrow \wp(\mathcal{V}_t)$ is defined by:

$$TV(\langle\tau_1, \ldots, \tau_n\rangle) = TV(\tau_1) \cup \ldots \cup TV(\tau_n)$$

Definition B.1.10 (closed types)
The set of closed types $\mathcal{T}^{\mathsf{closed}} \subseteq \mathcal{T}$ is defined by:

$$\mathcal{T}^{\mathsf{closed}} = \{\tau \mid \tau \in \mathcal{T}, TV(\tau) = \emptyset\}$$

Using these definitions all types and values in Clean can be constructed. Consider lists and trees (example B.1.3) for example.

Example B.1.11 (lists and trees)
List and tree types can be constructed by an algebraic type symbols with arity one:

$\mathcal{S}_a = \{\mathsf{tlist}, \mathsf{tree}\}$,
$Arity(\mathsf{tlist}) = 1, Arity(\mathsf{tree}) = 1$

List values require constructor symbols for the empty list and for list construction. The tree equivalents are nodes and leaves:

$\mathcal{S}_c = \{\mathsf{nil}, \mathsf{cons}, \mathsf{node}, \mathsf{leaf}\}$,
$Arity(\mathsf{nil}) = 0, Arity(\mathsf{cons}) = 2, Arity(\mathsf{node}) = 2, Arity(\mathsf{leaf}) = 1$

B.1.2 Expressions

Function symbols are used as references to functions. Normally, the name of the function will be used as the function symbol. The arity of a function symbol is the number of arguments the referenced function can be applied to.

Definition B.1.12 (function symbols)
$\mathcal{S}_f$ is the set of function symbols.
The function $Arity :: \mathcal{S}_f \rightarrow \mathbb{N}$ returns the arity of a function symbol.

The set of expressions consists of *variables*, *basic values*, *constructed data*, *function applications*, *applications* of expressions, and *case distinctions*. Sharing is expressed using *lazy* and *strict let definitions*. In the strict let definition, the shared computation may not be recursive and can only be used by reduction when the computation is known to be defined. The erroneous expression is represented by *bottom*. Note that these expressions are only the basic expressions required to express Clean programs. Hence, type classes are not available but translated as explained in section 2.8.

Definition B.1.13 (expressions)
The set of expressions $\mathcal{E}$ is defined by:

$$\begin{array}{lll}
\mathcal{E} = \{\mathsf{var}\ x & \mid x \in \mathcal{V}_e\} & \text{(variable)} \\
\quad \cup\ \{\mathsf{basic}\ b & \mid b \in B_v\} & \text{(basic value)} \\
\quad \cup\ \{\mathsf{constr}\ c\ \vec{e} & \mid c \in \mathcal{S}_c, \vec{e} \in \langle\mathcal{E}\rangle, \mathit{Arity}(c) \geq |\vec{e}|\} & \text{(data construction)} \\
\quad \cup\ \{\mathsf{func}\ f\ \vec{e} & \mid f \in \mathcal{S}_f, \vec{e} \in \langle\mathcal{E}\rangle, \mathit{Arity}(f) \geq |\vec{e}|\} & \text{(function application)} \\
\quad \cup\ \{\mathsf{apply}\ e\ e' & \mid e \in \mathcal{E}, e' \in \mathcal{E}\} & \text{(application)} \\
\quad \cup\ \{\mathsf{case}\ e'\ \mathsf{of}\ \vec{a} = \vec{e} & \mid e' \in \mathcal{E}, \vec{e} \in \langle\mathcal{E}\rangle, \vec{a} \in \mathit{casealts}\} & \text{(case distinction)} \\
\quad \cup\ \{\mathsf{let!}\ x = e\ \mathsf{in}\ e' & \mid x \in \mathcal{V}_e, e \in \mathcal{E}, e' \in \mathcal{E}\} & \text{(strict sharing)} \\
\quad \cup\ \{\mathsf{let}\ \vec{x} = \vec{e}\ \mathsf{in}\ e' & \mid \vec{x} \in \langle\mathcal{V}_e\rangle, \vec{e} \in \langle\mathcal{E}\rangle, e' \in \mathcal{E}\} & \text{(lazy sharing)} \\
\quad \cup\ \{\mathsf{bottom}\} & & \text{(bottom)}
\end{array}$$

where the set of case alternatives *casealts* is assumed.

Assumption B.1.14 (free variables in expressions)
The function $FV :: \mathcal{E} \rightarrow \wp(\mathcal{V}_e)$ returns the set of free expression variables in an expression.

Definition B.1.15 (closed expressions)
The set of closed expressions $\mathcal{E}^{\mathsf{closed}} \subseteq \mathcal{E}$ is defined by:

$$\mathcal{E}^{\mathsf{closed}} = \{e \mid e \in \mathcal{E}, FV(e) = \emptyset\}$$

In Core, basic functions for basic types (like the arithmetic operations on integers and reals) are called *delta functions*. Their implementations and properties are predefined. This formal detail is not important here; we will just assume they are normal functions.

B.1.3 Programs

A Core program is executed by rewriting the `Start` function. The rewriting process often makes use of the other function definitions. A program is the collection of functions and definitions of algebraic data types that is available.

A function definition defines the types of the input parameters, the variables to which the output expressions are assigned, the result type, and the function expression.

Definition B.1.16 (function definitions)
The set of function definitions $\mathcal{F}_d$ is defined by:

$$\mathcal{F}_d = \{\mathsf{function}\ f\ \vec{x}\ e\ \tau \mid f \in \mathcal{S}_f, \vec{x} \in \langle\mathcal{V}_e\rangle, e \in \mathcal{E}, \tau \in \mathcal{T}\}$$

An algebraic data type definition (like `Tree` in example B.1.3) is a collection of alternatives, where each alternative is a constructor followed by the types of the values.

Definition B.1.17 (alternatives)
The set of alternatives *Alts* is defined by:

$$\mathit{Alts} = \{(c, \vec{\sigma}) \mid c \in \mathcal{S}_c, \vec{\sigma} \in \langle\mathcal{T}\rangle, \mathit{Arity}(c) = |\vec{\sigma}|\}$$

Definition B.1.18 (algebraic data type definitions)
The set of algebraic data type definitions $\mathcal{A}$ is defined by:

$$\mathcal{A} = \{\mathsf{algdef}\ t\ \vec{a} \mid t \in \mathcal{S}_a, \vec{a} \in \langle\mathit{Alts}\rangle\}$$

A program assigns function definitions to function symbols and algebraic type definitions to algebraic type symbols.

Definition B.1.19 (programs)
The set of programs Ψ is defined by:
$\Psi = \{\mathsf{prog}\ \psi \mid \psi \in (\mathcal{S}_a \cup \mathcal{S}_f) \hookrightarrow (\mathcal{A} \cup \mathcal{F}_d)\}$

The set of programs includes also programs that are not welltyped. Welltypedness is defined in the next section, after which only welltyped programs will be considered.

B.1.4 Semantics

The programming language CORE and its semantics are based on term rewriting, whereas CLEAN is based on graph rewriting. Nevertheless, the expressivity and semantics are equivalent.

The operational semantics of the programming language is defined by a reduction mechanism which describes the evaluation of expressions in the context of a program. This mechanism uses 21 conditional rules that define how an expression may be rewritten. It supports sharing, enforces strict evaluation of strict let definitions, is correct with respect to CLEAN, and is useful for reasoning. It is a single-step system in which any redex (hence, any subexpression) can be reduced. Thus, the order in which the redexes are reduced can be chosen freely. Additionally, every step preserves welltypedness. The rewrite function will not be specified here (see [4] for a detailed specification).

B.1.5 Welltypedness

In CLEAN, all valid programs must be welltyped. The type inference function, that can be used to type expressions, will be assumed (as in [4]).

Definition B.1.20 (type contexts)
The set of type contexts Γ^t is defined by:
$\Gamma^t = \mathcal{V}_e \hookrightarrow \mathcal{T}$

Assumption B.1.21 (type inference function)
The function $\mathit{Type}_\psi :: \mathcal{E} \times \Gamma^t \hookrightarrow \mathcal{T} \times \Gamma^t$ has the following properties:

- It corresponds to standard Mycroft typing
- All free variables from the expression are in the resulting typing context
- The resulting type is the most specific type possible
- The returned context is the input context with variable type bindings added and types made more specific
- The domain represents all tuples of welltyped expressions and typing contexts. Hence, an expression is welltyped in a context iff they are a member of the domain of Type_ψ.

Using the type inference function, welltypedness can be defined for expressions, function definitions and programs. Only welltyped programs will be considered valid.

An expression is welltyped if and only if a type can be inferred for it.

Definition B.1.22 (welltypedness of expressions in a type context)
The relation $\mathit{WellTyped}_\psi \subseteq \mathcal{E} \times \Gamma^t$ is defined by:
$\mathit{WellTyped}_\psi(e, \gamma) \Leftrightarrow (e, \gamma) \in \mathit{Dom}(\mathit{Type}_\psi)$

Definition B.1.23 (welltypedness of functions)
The relation $\mathit{WellTyped}_\psi \subseteq \mathcal{F}_d$ is defined by:
$\mathit{WellTyped}_\psi(\textsf{function } f\ \langle x_1, \ldots, x_n \rangle\ e\ (\tau_1 \to \ldots \to \tau_n \to \sigma)) \Leftrightarrow$
$\mathit{Type}_\psi(e, \{(x_1, \tau_1), \ldots, (x_n, \tau_n)\}) = (\sigma, \vec{\tau})$ (modulo α-conversion)

Finally, a program is welltyped if and only if all function definitions it provides are welltyped.

Definition B.1.24 (welltypedness of programs)
The relation $\mathit{WellTyped} \subseteq \Psi$ is defined by:
$\mathit{WellTyped}(\psi) \Leftrightarrow \forall_{f \in \mathcal{F}_d}[f \in \mathit{Dom}(\psi) \Rightarrow \mathit{WellTyped}_\psi(\psi(f))]$

B.2 Logic language

The purpose of Sparkle is to prove *properties* about Clean programs. In the previous section, a mathematical model of Clean programs was presented. This section defines the set of properties about these programs.

First, the set of propositions is defined. Then welltypedness is extended from expressions to propositions. Finally, the semantics of propositions is defined.

B.2.1 Propositions

In Sparkle, properties are defined in a first-order predicate logic extended with equality of expressions. Some examples are shown in example B.2.1.

Example B.2.1 (example properties)
Two example properties are:
$\forall_{\texttt{n}\in\texttt{Int}}\forall_{\texttt{a}}\forall_{\texttt{xs}\in[\texttt{a}]}[\texttt{n} \neq \textsf{bottom} \Rightarrow \texttt{take n xs ++ drop n xs} = \texttt{xs}]$
$\forall_{\texttt{a,b}}\forall_{\texttt{f}\in\texttt{a}\to\texttt{b}}\forall_{\texttt{xs}\in[\texttt{a}]}\forall_{\texttt{ys}\in[\texttt{a}]}[\texttt{map f (append xs ys)}$
$= \texttt{append (map xs) (map ys)}]$

For quantification of propositions, a set of proposition variables is required.

Definition B.2.2 (proposition variables)
$\mathcal{V}_p$ is the set of proposition variables.

The set of quantors is composed of universal quantors and existential quantors.

Definition B.2.3 (universal quantors)
The set of universal quantors $\mathcal{Q}_u$ is defined by:

$$\begin{aligned} \mathcal{Q}_u = {} & \{\textsf{forall types } x \mid x \in \mathcal{V}_t\} \\ \cup {} & \{\textsf{forall exprs } x\ \sigma \mid x \in \mathcal{V}_e, \sigma \in \mathcal{T}\} \\ \cup {} & \{\textsf{forall props } x \mid x \in \mathcal{V}_p\} \end{aligned}$$

Definition B.2.4 (existential quantors)
The set of existential quantors $\mathcal{Q}_e$ is defined by:

$$\begin{aligned} \mathcal{Q}_e \cup\ & \{\mathsf{exists\ expr}\ x\ \sigma \mid x \in \mathcal{V}_e, \sigma \in \mathcal{T}\} \\ \cup\ & \{\mathsf{exists\ prop}\ x \mid x \in \mathcal{V}_p\} \end{aligned}$$

Definition B.2.5 (quantors)
The set of quantors $\mathcal{Q}$ is defined by:

$$\mathcal{Q} = \mathcal{Q}_u \cup \mathcal{Q}_e$$

The set of properties is a formalization of a first-order predicate logic extended with equality of expressions.

Definition B.2.6 (properties)
The set of properties *Props* is defined by:

$$\begin{array}{lll} \mathit{Props} = \{\mathsf{var}\ \alpha & \mid \alpha \in \mathcal{V}_p\} & \text{(variable)} \\ \quad \cup \{e\ \mathsf{equals}\ e' & \mid e \in \mathcal{E}, e' \in \mathcal{E}\} & \text{(equality)} \\ \quad \cup \{\mathsf{true}\} & & \text{(true)} \\ \quad \cup \{\mathsf{false}\} & & \text{(false)} \\ \quad \cup \{\mathsf{not}\ p & \mid p \in \mathit{Props}\} & \text{(negation)} \\ \quad \cup \{p\ \mathsf{and}\ q & \mid p \in \mathit{Props}, q \in \mathit{Props}\} & \text{(conjunction)} \\ \quad \cup \{p\ \mathsf{or}\ q & \mid p \in \mathit{Props}, q \in \mathit{Props}\} & \text{(disjunction)} \\ \quad \cup \{p\ \mathsf{implies}\ q & \mid p \in \mathit{Props}, q \in \mathit{Props}\} & \text{(implication)} \\ \quad \cup \{p\ \mathsf{iff}\ q & \mid p \in \mathit{Props}, q \in \mathit{Props}\} & \text{(equivalence)} \\ \quad \cup \{\mathsf{quantor}\ q\ p & \mid p \in \mathit{Props}, q \in \mathcal{Q}\} & \text{(quantification)} \end{array}$$

Assumption B.2.7 (free variables in propositions)
The function $PV :: \mathit{Props} \to \wp(\mathcal{V}_p)$ returns the set of free variables in a proposition.

Definition B.2.8 (closed propositions)
The set of closed propositions $\mathit{Props}^{\mathsf{closed}} \subseteq \mathit{Props}$ is defined by:

$$\mathit{Props}^{\mathsf{closed}} = \{p \mid p \in \mathit{Props}, PV(p) = \emptyset\}$$

B.2.2 Semantics

All valid properties are closed, they do not contain any free variables. Hence, only the semantics of closed properties has to be defined.

The semantics of proposition variables is the semantics of the proposition it stands for. Except for equality of expressions, standard first-order logic semantics is used. Two expressions are said to be equal, when replacing one by the other in any program does not change the observed behavior of the program (see [4]). This equality is designed to handle infinite (lazy) and undefined expressions well.

Assumption B.2.9 (equality of expressions)
For all programs $\psi \in \Psi$, the relation $\mathit{Equal}_\psi \subseteq \mathcal{E}^{\mathsf{closed}} \times \mathcal{E}^{\mathsf{closed}}$ is defined such that $\mathit{Equal}_\psi(e, e')$ is true iff e and e' are considered equal.

First, the sets over which the quantors in a property quantify are defined.

Definition B.2.10 (variables)
The set of all variables $\mathcal{V}$ is defined by:

$$\mathcal{V} = \mathcal{V}_t \cup \mathcal{V}_e \cup \mathcal{V}_p$$

Definition B.2.11 (closed terms)
The set of closed terms $\mathcal{X}^{\mathsf{closed}}$ is defined by:

$$\mathcal{X}^{\mathsf{closed}} = \mathcal{T}^{\mathsf{closed}} \cup \mathcal{E}^{\mathsf{closed}} \cup \mathit{Props}^{\mathsf{closed}}$$

Definition B.2.12 (instances of quantors)
For all programs $\psi \in \Psi$, the function $\mathit{Instances}_\psi :: \mathcal{Q} \rightarrow \wp(\mathcal{V} \times \mathcal{X}^{\mathsf{closed}})$ is defined by:

$$\begin{array}{lll}
\mathit{Instances}_\psi(\mathsf{forall\ types}\ \alpha) & = \{(\alpha,\sigma) & \mid \sigma \in \mathcal{T}^{\mathsf{closed}}\} \\
\mathit{Instances}_\psi(\mathsf{forall\ exprs}\ x\ \sigma) & = \{(x,e) & \mid e \in \mathcal{E}^{\mathsf{closed}}, \mathit{Type}_\psi(e,\emptyset) = \sigma\} \\
\mathit{Instances}_\psi(\mathsf{exists\ exprs}\ x\ \sigma) & = \{(x,e) & \mid e \in \mathcal{E}^{\mathsf{closed}}, \mathit{Type}_\psi(e,\emptyset) = \sigma\} \\
\mathit{Instances}_\psi(\mathsf{forall\ props}\ p) & = \{(x,p) & \mid p \in \mathit{Props}^{\mathsf{closed}}\} \\
\mathit{Instances}_\psi(\mathsf{exists\ props}\ p) & = \{(x,p) & \mid p \in \mathit{Props}^{\mathsf{closed}}\}
\end{array}$$

Using the previous definitions, the semantics of properties is straightforwardly defined.

Definition B.2.13 (semantics of closed properties)
For all programs $\psi \in \Psi$, the predicate $[\![\cdot]\!]_\psi \subseteq \mathit{Props}^{\mathsf{closed}}$ is defined by:

$$\begin{array}{ll}
[\![e\ \mathsf{equals}\ e']\!]_\psi & \Leftrightarrow \mathit{Equal}_\psi(e,e') \\
[\![\mathsf{true}]\!]_\psi & \Leftrightarrow \text{true} \\
[\![\mathsf{false}]\!]_\psi & \Leftrightarrow \text{false} \\
[\![\mathsf{not}\ p]\!]_\psi & \Leftrightarrow \neg[\![p]\!]_\psi \\
[\![p\ \mathsf{and}\ q]\!]_\psi & \Leftrightarrow [\![p]\!]_\psi \wedge [\![q]\!]_\psi \\
[\![p\ \mathsf{or}\ q]\!]_\psi & \Leftrightarrow [\![p]\!]_\psi \vee [\![q]\!]_\psi \\
[\![p\ \mathsf{implies}\ q]\!]_\psi & \Leftrightarrow ([\![p]\!]_\psi \Rightarrow [\![q]\!]_\psi) \\
[\![p\ \mathsf{iff}\ q]\!]_\psi & \Leftrightarrow ([\![p]\!]_\psi \Leftrightarrow [\![q]\!]_\psi) \\
[\![\mathsf{quantor}\ q\ p]\!]_\psi & \Leftrightarrow \begin{cases} \forall_{(x,X)\in \mathit{Instances}_\psi(q)}[[\![p_{x\mapsto X}]\!]_\psi] & \text{if } q \in \mathcal{Q}_u \\ \exists_{(x,X)\in \mathit{Instances}_\psi(q)}[[\![p_{x\mapsto X}]\!]_\psi] & \text{if } q \in \mathcal{Q}_e \end{cases}
\end{array}$$

B.2.3 Welltypedness

A property is welltyped if every two expressions to be compared have the same type given the type context. The type context is built by the surrounding quantors.

Definition B.2.14 (equally typed expressions in a type context)
For all programs $\psi \in \Psi$, the relation $\mathit{EquallyTyped}_\psi \subseteq \Gamma_t \times \mathcal{E} \times \mathcal{E}$ is defined by:

$$\mathit{EquallyTyped}_\psi(\gamma,e,e') \Leftrightarrow \exists_{\sigma\in\mathcal{T}}[\mathit{Type}_\psi(\gamma,e) = \sigma \wedge \mathit{Type}_\psi(\gamma,e') = \sigma]$$

Definition B.2.15 (add quantor to type context)
For all programs $\psi \in \Psi$, the function $\mathit{Add}_\psi :: \mathcal{Q} \times \Gamma^t \rightarrow \Gamma^t$ is defined by:

$$\begin{array}{l}
\mathit{Add}(\mathsf{forall\ types}\ x,\quad \gamma) = \gamma \\
\mathit{Add}(\mathsf{forall\ exprs}\ x\ \sigma, \gamma) = \gamma_{x\mapsto\sigma} \\
\mathit{Add}(\mathsf{exists\ expr}\ x\ \sigma, \gamma) = \gamma_{x\mapsto\sigma} \\
\mathit{Add}(\mathsf{forall\ props}\ x,\quad \gamma) = \gamma \\
\mathit{Add}(\mathsf{exists\ prop}\ x,\quad \gamma) = \gamma
\end{array}$$

Definition B.2.16 (welltypedness of propositions)
For all programs $\psi \in \Psi$, the relation $\mathit{WellTyped}_\psi \subseteq \mathit{Props} \times \Gamma^t$ is defined by:

$$\begin{array}{lcl}
\mathit{WellTyped}_\psi(\gamma, \mathsf{var}\ \alpha) & \Leftrightarrow & \alpha \in \mathcal{V}_p \\
\mathit{WellTyped}_\psi(\gamma, e\ \mathsf{equals}\ e') & \Leftrightarrow & \mathit{EquallyTyped}_\psi(\gamma, e, e') \\
\mathit{WellTyped}_\psi(\gamma, \mathsf{true}) & \Leftrightarrow & \text{true} \\
\mathit{WellTyped}_\psi(\gamma, \mathsf{false}) & \Leftrightarrow & \text{false} \\
\mathit{WellTyped}_\psi(\gamma, \mathsf{not}\ p) & \Leftrightarrow & \mathit{WellTyped}_\psi(\gamma, p) \\
\mathit{WellTyped}_\psi(\gamma, p\ \mathsf{and}\ q) & \Leftrightarrow & \mathit{WellTyped}_\psi(\gamma, p) \wedge \mathit{WellTyped}_\psi(\gamma, q) \\
\mathit{WellTyped}_\psi(\gamma, p\ \mathsf{or}\ q) & \Leftrightarrow & \mathit{WellTyped}_\psi(\gamma, p) \wedge \mathit{WellTyped}_\psi(\gamma, q) \\
\mathit{WellTyped}_\psi(\gamma, p\ \mathsf{implies}\ q) & \Leftrightarrow & \mathit{WellTyped}_\psi(\gamma, p) \wedge \mathit{WellTyped}_\psi(\gamma, q) \\
\mathit{WellTyped}_\psi(\gamma, p\ \mathsf{iff}\ q) & \Leftrightarrow & \mathit{WellTyped}_\psi(\gamma, p) \wedge \mathit{WellTyped}_\psi(\gamma, q) \\
\mathit{WellTyped}_\psi(\gamma, \mathsf{quantor}\ q\ p) & \Leftrightarrow & \mathit{WellTyped}_\psi(\mathit{Add}(\gamma, q), p)
\end{array}$$

B.3 Proof language

In this section, the language that describes proofs is defined. In general, a proof is a syntactical derivation of a logical statement. Every derivation step transforms a logical statement in a conjunction of logical statement that imply the original statement. The result, a complete proof, is a tree of logical statements where the leaves are trivially true.

In Sparkle, the logical statements are the properties and the steps allowed are described by *tactics*. Instead of using trees, Sparkle uses a representation that adds more structure to the proofs. Respectively, the language, the semantics, and the welltypedness are defined.

B.3.1 Proofs

A proof goal corresponds to a logical statement, the property that is to be proven. To keep track of the introductions performed by the user, a proof goal is further split into introduced annotated quantors, introduced hypotheses, and a target proposition ($\forall_{q_1,\ldots,q_n}[H_1 \Rightarrow H_2 \Rightarrow \cdots \Rightarrow H_m \Rightarrow P]$).

Definition B.3.1 (closed proof goals)
The set of closed proof goals $\mathit{Goal}^{\mathsf{closed}}$is defined by:

$$\mathit{Goal}^{\mathsf{closed}} = \{\mathsf{prove}\ p\ \mathsf{with}\ \vec{q}\ \vec{h} \mid p \in \mathit{Props}, \vec{q} \in \langle \mathcal{Q}_u \rangle, \vec{h} \in \langle \mathit{Props}^{\mathsf{closed}} \rangle\}$$

A *proof state* is a list of proof goals corresponding to a conjunction of properties, which are essentially all the leaves in a (partial) proof tree.

Definition B.3.2 (closed proof states)
The set of closed proof states $\mathit{State}^{\mathsf{closed}}$is defined by:

$$\mathit{State}^{\mathsf{closed}} = \{\mathsf{goals}\ \vec{g} \mid \vec{g} \in \langle \mathit{Goal}^{\mathsf{closed}} \rangle\}$$

A tactic is a function from a goal to a list of goals. All tactics have to be sound with respect to the semantics. That is, the created goals have to logically imply the original goal. Applying a tactic changes the proof state in one that is hopefully easier to prove.

Definition B.3.3 (tactic)
A tactic is a function $f :: \mathit{Goal} \hookrightarrow \langle \mathit{Goal} \rangle$ such that:
$\forall_{g \in \mathit{Goal}}[g \in \mathit{Dom}(f) \Rightarrow [f(g) \Rightarrow g]]$.

B.3.2 Semantics

Since proof states and proof goals correspond to (a conjunction of) properties, their semantics is defined by the semantics of properties. This differs from the more advanced definition by Maarten de Mol [4], which explicitly takes alpha conversion into account.

First, a translation from proof goals and states to propositions is defined.

Definition B.3.4 (translate goal to proposition)
The function $\mathit{ToProp} :: \mathit{Goal}^{\mathsf{closed}} \to \mathit{Props}^{\mathsf{closed}}$ that translates a proof goal to a property is defined by:
$\mathit{ToProp}(\mathsf{prove}\ p\ \mathsf{with}\ \langle q_1, \ldots, q_n \rangle\ \langle h_1, \ldots, h_m \rangle) =$
$\mathsf{quantor}\ q_1\ \ldots\ \mathsf{quantor}\ q_n\ [h_1\ \mathsf{implies}\ \ldots\ \mathsf{implies}\ h_m\ \mathsf{implies}\ p]$

Definition B.3.5 (translate proof state to proposition)
The function $\mathit{ToProp} :: \mathit{State}^{\mathsf{closed}} \to \mathit{Props}^{\mathsf{closed}}$ that translates a proof state to a property is defined by:
$\mathit{ToProp}(\mathsf{goals}\ \langle g_1, \ldots, g_n \rangle) = q_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ q_n$

The semantics of proof goals and states is defined as the semantics of the properties to which they can be translated.

Definition B.3.6 (semantics of proof goals)
For all programs $\psi \in \Psi$, the meaning predicate $[\![\cdot]\!]_\psi \subseteq \mathit{Goal}^{\mathsf{closed}}$ is defined by:
$[\![g]\!]_\psi \Leftrightarrow [\![\mathit{ToProp}(g)]\!]_\psi$

Definition B.3.7 (semantics of proof states)
For all programs $\psi \in \Psi$, the meaning predicate $[\![\cdot]\!]_\psi \subseteq \mathit{State}^{\mathsf{closed}}$ is defined by:
$[\![s]\!]_\psi \Leftrightarrow [\![\mathit{ToProp}(s)]\!]_\psi$

B.3.3 Welltypedness

Just as the semantics, welltypedness of proof goals and states can be defined in terms of the properties to which they can be translated.

Definition B.3.8 (welltypedness of proof goals)
For all programs $\psi \in \Psi$, the relation $\mathit{WellTyped}_\psi \subseteq \mathit{Goal}^{\mathsf{closed}} \times \Gamma^t$ is defined by:
$\mathit{WellTyped}_\psi(\gamma, g) \;\Leftrightarrow\; \mathit{WellTyped}_\psi(\gamma, \mathit{ToProp}(g))$

Definition B.3.9 (weltypedness of proof states)
For all programs $\psi \in \Psi$, the relation $\mathit{WellTyped}_\psi \subseteq \mathit{State}^{\mathsf{closed}} \times \Gamma^t$ is defined by:
$\mathit{WellTyped}_\psi(\gamma, s) \;\Leftrightarrow\; \mathit{WellTyped}_\psi(\gamma, \mathit{ToProp}(s))$

Proof sketches C

In this appendix, proof sketches are given for the proof rules and tactics from section 3.3. Being sketches, some steps in the proofs are rather large. Nevertheless, they are correct and can be verified with relatively little work.

Section C.1 and C.2 show that the set of instances and the set of multiple instances are well-founded and that the defined orders are indeed partial orders. In sections C.3 and C.4, the proofs of the **(inst-tactic)** and **(multi-tactic)** are sketched.

C.1 Order on instances

In this section it is shown that $\leq_{(\psi,c)}$ is a partial order by showing that a natural number can be assigned to types such that if one type is smaller than another the number is as well.

First, a natural number is assigned to a class constraint.

Definition C.1.1 (assign number to class constraints)
The function $Num :: \mathcal{P} \hookrightarrow \mathbb{N}$ is defined by:
$Num(\vec{\tau} :: c) = min_n(\vec{\tau} \in Instances_\psi(c, n))$

The first lemma shows that if a predicate is a dependency of another, it will have a smaller number assigned.

Lemma C.1.2 (dependencies have a smaller number)
For all predicates π we have:
$\forall_{\pi' \in Deps_\psi(\pi)}[Num(\pi') < Num(\pi)]$
Proof:
Assume $Num(\pi) = n$. Then by definition of the sets of instances (3.1.40), if $(\vec{\tau} :: c) \in Deps_\psi(\pi)$ then $\vec{\tau} \in \bigcup_{1 \leq i \leq n-1}[Instances_\psi(c, i)]$. Hence, $Num(\pi') < n$.

The second lemma shows that expanding compound classes never creates predicates with a larger number.

Lemma C.1.3 (expanding compounds does not increase number)
For all predicates $\vec{\tau} :: c$ we have:
$\forall_{\pi \in ExpComp_\psi(\vec{\tau}::c)}[Num(\pi) \leq Num(\vec{\tau} :: c)]$

Proof:

The definition of $\mathit{ExpComp}_\psi$ (3.3.6) distinguishes two cases.

Case $\neg \mathit{Compound}_\psi(c)$:

By definition of $\mathit{ExpComp}_\psi$ we have to prove $\mathit{Num}(\vec{\tau} :: c) \leq \mathit{Num}(\vec{\tau} :: c)$, which is trivially true.

Case $\mathit{Compound}_\psi(c)$:

By definition of $\mathit{ExpComp}_\psi$ we have to prove:

$\forall_{\pi \in \mathit{ExpComp}_\psi(\mathit{SubClasses}_\psi(\vec{\tau}::c))}[\mathit{Num}(\pi) \leq \mathit{Num}(\vec{\tau} :: c)]$

The proof proceeds by induction on $\mathcal{S}_{Cl}$ using the subclass relation.

Base cases:

c has no subclasses, thus $\mathit{SubClasses}_\psi(\vec{\tau} :: c) = \langle\rangle$. Hence we should prove $\forall_{\pi \in \langle\rangle}[\mathit{Num}(\pi) \leq \mathit{Num}(\vec{\tau} :: c)]$, which is trivially true.

Induction Step:

Assume that $\mathit{SubClasses}_\psi(\vec{\tau} :: c) = \langle \vec{\tau}_1 :: c_1, \ldots, \vec{\tau}_n :: c_n \rangle$ and $\vec{\pi}' = \mathit{ExpComp}_\psi(\vec{\tau}_1 :: c_1) \circ \ldots \circ \mathit{ExpComp}_\psi(\vec{\tau}_n :: c_n)$. Then, we have to prove: $\forall_{\pi \in \vec{\pi}'}[\mathit{Num}(\pi) \leq \mathit{Num}(\vec{\tau} :: c)]$

By the induction hypothesis we know that for all $1 \leq i \leq n$:

$\forall_{\pi_i \in \mathit{ExpComp}_\psi(\vec{\tau}_i::c_i)}[\mathit{Num}(\pi_i) \leq \mathit{Num}(\vec{\tau}_i :: c_i)]$

By lemma C.1.2 we have $\mathit{Num}(\vec{\tau}_i :: c_i) \leq \mathit{Num}(\vec{\tau} :: c)$. Hence, $\mathit{Num}(\pi_i) \leq \mathit{Num}(\vec{\tau} :: c)$.

Using these lemmas, it can be shown that the number decreases as types get smaller.

Lemma C.1.4 (smaller types ($<^1_{(\psi,c)}$) have a smaller number)

For all classes $c \in \mathcal{S}_{Cl}$ and types $\vec{\tau}$ and $\vec{\sigma}$, the following holds:

$\vec{\sigma} <^1_{(\psi,c)} \vec{\tau} \Rightarrow \mathit{Num}(\vec{\sigma} :: c) < \mathit{Num}(\vec{\tau} :: c)$

Proof:

$\vec{\sigma} <^1_{(\psi,c)} \vec{\tau}$ by definition (3.3.18) means that $(\vec{\sigma} :: c) \in \mathit{Deps}^1_\psi(\vec{\tau} :: c)$. By definition (3.3.5), this implies $(\vec{\sigma} :: c) \in \mathit{ExpComp}_\psi(\mathit{Deps}_\psi(\vec{\tau} :: c))$. Then, by lemma C.1.2 and C.1.3, we have $\mathit{Num}(\vec{\sigma} :: c) < \mathit{Num}(\vec{\tau} :: c)$.

This lemma can be used to show that $\leq_{(\psi,c)}$ is a partial order and that the set $\mathit{Instances}_\psi(c)$ is well-founded.

Proposition C.1.5 ($\leq_{(\psi,c)}$ is a partial order)

For all classes $c \in \mathcal{S}_{Cl}$, $\leq_{(\psi,c)}$ is a partial order.

Proof:

$\leq_{(\psi,c)}$ is reflexive and transitive by definition. Anti-symmetry and well-foundedness are guaranteed by lemma C.1.4.

That $\leq_{(\psi,c)}$ is a partial order also proves **(inst-rule)** (3.3.21), since that was required by the well-founded induction theorem.

C.2 Order on multiple instances

In this section, it is shown that $\leq_{(\psi,\vec{\pi})}$ is a partial order by showing that a natural number can be assigned to types such that if one type is smaller than another the number is as well.

First, a natural number is assigned to a sequence of class constraints.

Definition C.2.1 (assign number to multiple class constraints)
The function $\mathit{Num} :: \langle\mathcal{P}\rangle \hookrightarrow \mathbb{N}$ is defined by:

$\mathit{Num}(\langle\pi_1, \ldots, \pi_n\rangle) = \mathit{max}(\mathit{Num}(\pi_1), \ldots, \mathit{Num}(\pi_n))$

Using the lemmas from the previous section, it can be shown that this number decreases as types get smaller.

Lemma C.2.2 (smaller types ($<^1_{(\psi,\vec{\pi})}$) have a smaller number)
For all predicates $\vec{\pi} \in \langle\mathcal{P}\rangle$ and types $\vec{\tau}$ and $\vec{\sigma}$, the following holds:

$\vec{\sigma} <^1_{(\psi,\vec{\pi})} \vec{\tau} \Rightarrow \mathit{SetNum}(*_{(TV(\vec{\pi})\leftrightarrow\vec{\sigma})}(\vec{\pi})) < \mathit{SetNum}(*_{(TV(\vec{\pi})\leftrightarrow\vec{\tau})}(\vec{\pi}))$

Proof:

$\vec{\sigma} <^1_{(\psi,\pi)} \vec{\tau}$ by definition (3.3.28) implies:

$*_{(TV(\vec{\pi})\leftrightarrow\vec{\sigma})}(\vec{\pi}) \subseteq \bigcup_{\pi\in\vec{\pi}}[\mathit{Deps}^1_\psi(*_{(TV(\vec{\pi})\leftrightarrow\vec{\tau})}(\pi))]$

Expanding Deps^1_ψ (3.3.5) gives:

$*_{(TV(\vec{\pi})\leftrightarrow\vec{\sigma})}(\vec{\pi}) \subseteq \bigcup_{\pi\in\vec{\pi}}[\mathit{ExpComp}_\psi(\mathit{Deps}_\psi(*_{(TV(\vec{\pi})\leftrightarrow\vec{\tau})}(\pi)))]$

By lemma C.1.2 and C.1.3 we know that for all $\pi' \in \vec{\pi}$:

$\mathit{Num}(*_{(TV(\pi)\leftrightarrow\vec{\sigma})}(\pi')) < \mathit{Num}(*_{(TV(\pi)\leftrightarrow\vec{\tau})}(\pi'))$

Hence, $\mathit{SetNum}(*_{(TV(\vec{\pi})\leftrightarrow\vec{\sigma})}(\vec{\pi})) < \mathit{SetNum}(*_{(TV(\vec{\pi})\leftrightarrow\vec{\tau})}(\vec{\pi}))$

This lemma can be used to show that $\leq_{(\psi,\vec{\pi})}$ is a partial order and that the set $\mathit{SetInstances}_\psi(\vec{\pi})$ is well-founded.

Proposition C.2.3
For all predicates $\vec{\pi} \in \langle\mathcal{P}\rangle$, $\leq_{(\psi,\vec{\pi})}$ is a partial order.

Proof:

$\leq_{(\psi,\vec{\pi})}$ is reflexive and transitive by definition. Anti-symmetry and well-foundedness are guaranteed by lemma C.2.2.

That $\leq_{(\psi,\vec{\pi})}$ is a partial order also proves **(multi-rule)** (3.3.30), since that was required by the well-founded induction theorem.

C.3 Proof of (inst-tactic)

In this section the **(inst-tactic)** is proven correct. That is, it is proven to be sound; the top property logically implies the bottom one. This is shown in three steps, starting with the sound **(inst-rule)**.

Proposition C.3.1 ((inst-tactic) is sound)
The **(inst-tactic)** is sound.

Proof:

We start with the **(inst-rule)**:

$$\frac{\forall_{\vec{\tau}\in \mathit{Instances}_\psi(c)}[\forall_{\vec{\sigma}<^1_{(\psi,c)}\vec{\tau}}[P(\vec{\sigma})] \Rightarrow P(\vec{\tau})]}{\forall_{\vec{\alpha}\in \mathit{Instances}_\psi(c)}[P(\vec{\alpha})]}$$

By definition of $\mathit{Instances}_\psi$ (3.1.41) this implies:

$$\frac{\begin{array}{l}\forall_{i\in \mathit{InstDefs}_\psi(c)}\forall_{\mathit{Head}(i)\in \mathit{Instances}_\psi(c)}\\ \quad[\ i = \mathit{ApplyInstance}_\psi(\mathit{Head}(i) :: c)\\ \quad\ \Rightarrow \forall_{\vec{\sigma}<^1_{(\psi,c)}\vec{\tau}}[P(\vec{\sigma})] \Rightarrow P(\vec{\tau})\\ \quad]\end{array}}{\forall_{\vec{\alpha}\in \mathit{Instances}_\psi(c)}[P(\vec{\alpha})]}$$

By the definition of $<^1_{(\psi,c)}$ (3.3.18) this implies:

$$\frac{\begin{array}{l}\forall_{i \in \mathit{InstDefs}_\psi(c)} \forall_{\mathit{Head}(i) \in \mathit{Instances}_\psi(c)} \\ \quad [\ i = \mathit{ApplyInstance}_\psi(\mathit{Head}(i) :: c) \\ \quad\quad \Rightarrow \forall_{(\sigma::c') \in \mathit{Deps}^1_\psi(\mathit{Head}(i)::c,i)}[c = c' \Rightarrow \exists_*[*(\vec{\alpha}) = \vec{\sigma}] \Rightarrow P(\vec{\sigma})] \Rightarrow P(\vec{\tau}) \\ \quad]\end{array}}{\forall_{\vec{\alpha} \in \mathit{Instances}_\psi(c)}[P(\vec{\alpha})]}$$

By making dictionaries explicit this implies the **(inst-tactic)**:

$$\frac{\begin{array}{l}\forall_{i \in \mathit{InstDefs}_\psi(c)} \forall_{\mathit{TV}(\mathit{Head}(i) \in \langle \mathcal{T}^{\mathsf{closed}} \rangle)} \forall_{\mathit{Deps}(\mathit{Head}(i)::c,i)} \\ \quad [\ \forall_{(\vec{\sigma}::c') \in \mathit{Deps}_\psi(\mathit{Head}(i)::c,i)}[c = c' \Rightarrow \exists_*[*(\vec{\alpha}) = \vec{\sigma}] \Rightarrow P(d_{(\vec{\sigma}::c)}, \vec{\sigma})] \\ \quad\quad \Rightarrow P(\mathit{Dict}^1_\psi(\mathit{Head}(i) :: c, i), \mathit{Head}(i))] \\ \quad]\end{array}}{\forall_{\vec{\alpha} \in \langle \mathcal{T}^{\mathsf{closed}} \rangle} \forall_{\vec{\alpha}::c}[P(d_{(\vec{\alpha}::c)}, \vec{\alpha})]}$$

C.4 Proof of (multi-tactic)

In this section the **(multi-tactic)** is proven correct. That is, it is proven to be sound; the top property logically implies the bottom one. This is shown in three steps, starting with the (sound) **(multi-rule)**.

Proposition C.4.1 ((multi-tactic) is sound)
The **(multi-tactic)** is sound.
Proof:

We start with the **(multi-rule)**:

$$\frac{\forall_{\vec{\tau} \in \mathit{SetInstances}_\psi(\vec{\pi})}[\forall_{\vec{\sigma} <^1_{(\psi,\vec{\pi})} \vec{\tau}}[P(\vec{\sigma})] \Rightarrow P(\vec{\tau})]}{\forall_{\vec{\tau} \in \mathit{SetInstances}_\psi(\vec{\pi})}[P(\vec{\tau})]}$$

By definition of $\mathit{SetInstances}_\psi$ (3.3.26) this implies:

$$\frac{\begin{array}{l}\forall_{\vec{\imath} \in \mathit{InstDefs}_\psi(\vec{\pi})} \exists_{* \mid * = \mathit{SetMgu}(\vec{\pi}, \mathit{Head}(\vec{\imath}))} \forall_{*(\mathit{Head}(\vec{\imath})) \in \langle \mathcal{T}^{\mathsf{closed}} \rangle} \\ \quad [\ \vec{\imath} \in \mathit{ApplyInstance}_\psi(*(\vec{\pi})) \\ \quad\quad \Rightarrow \forall_{\vec{\sigma} <^1_{(\psi,\vec{\pi})} \vec{\tau}}[P(\vec{\sigma})] \Rightarrow P(\vec{\tau}) \\ \quad]\end{array}}{\forall_{\vec{\tau} \in \mathit{SetInstances}_\psi(\vec{\pi})}[P(\vec{\tau})]}$$

By the definition of $<^1_{(\psi,\vec{\pi})}$ (3.3.28) this implies:

$$\frac{\begin{array}{l}\forall_{\vec{\imath} \in \mathit{InstDefs}_\psi(\vec{\pi})} \exists_{* \mid * = \mathit{SetMgu}(\vec{\pi}, \mathit{Head}(\vec{\imath}))} \forall_{*(\mathit{Head}(\vec{\imath})) \in \langle \mathcal{T}^{\mathsf{closed}} \rangle} \\ \quad [\ \vec{\imath} \in \mathit{ApplyInstance}_\psi(*(\vec{\pi})) \\ \quad\quad \Rightarrow \forall_{*' \mid *'(\vec{\pi}) \subseteq \mathit{Deps}_\psi(*(\vec{\pi}), \vec{\imath})}[P(\vec{\sigma})] \Rightarrow P(\vec{\tau}) \\ \quad]\end{array}}{\forall_{\vec{\tau} \in \mathit{SetInstances}_\psi(\vec{\pi})}[P(\vec{\tau})]}$$

By making dictionaries explicit this implies the **(multi-tactic)**:

$$\frac{\begin{array}{l}\forall_{\vec{\imath} \in \mathit{InstDefs}_\psi(\vec{\pi})} \exists_{* \mid * = \mathit{SetMgu}(\vec{\pi}, \mathit{Head}(\vec{\imath}))} \forall_{*(\mathit{Head}(\vec{\imath})) \in \langle \mathcal{T}^{\mathsf{closed}} \rangle} \forall_{\mathit{Deps}^1_\psi(*(\vec{\pi}), \vec{\imath})} \\ \quad [\ \forall_{*' \mid *'(\vec{\pi}) \subseteq \mathit{Deps}_\psi(*(\vec{\pi}), \vec{\imath})}[P(d_{(*'(\vec{\pi}))}, *'(\mathit{TV}(\vec{\pi})))] \\ \quad\quad \Rightarrow P(\mathit{Dict}^1_\psi(*(\vec{\pi}), \vec{\imath}), *(\mathit{Head}(\vec{\imath}))) \\ \quad]\end{array}}{\forall_{\mathit{TV}(\vec{\pi})} \forall_{\vec{\pi}}[P(d_{(\vec{\pi})}, \mathit{TV}(\vec{\pi}))]}$$

Bewijsassistent verbeterd D

Het lijkt wel of tegenwoordig overal een chip en een computerprogramma in zit. Pinautomaten, magnetrons, wasmachines, horloges, scheerapparaten. Al die apparaten moet goed werken. Altijd. Je wilt bijvoorbeeld niet dat een pinautomaat teveel geld van je rekening afschrijft. Er mogen dus ook geen fouten in de computerprogramma's zitten.

Helaas. We weten maar al te goed dat er in computerprogramma's juist vrijwel altijd fouten zitten. Vastlopers van de PC zijn aan de orde van de dag, mobiele telefoons zijn inmiddels gemeengoed maar crashen voortdurend en ook auto's hebben steeds vaker last van 'softwareproblemen'. Technologische vooruitgang gaat gepaard met ongemakken.

Hoe komt dit nou? Worden de programmeurs niet goed genoeg betaald? Wordt de software niet goed uitgetest? Misschien, maar dit zijn zeker niet de enige oorzaken. Software is namelijk al snel te complex voor een mens om helemaal te begrijpen en te overzien, en het is vaak onmogelijk om alle denkbare situaties uit te testen, laat staan alle ondenkbare. Hoe kunnen we er dan ooit zeker van zijn dat het programma doet wat het moet doen?

De enige mogelijkheid om dit echt zeker te weten is door het te bewijzen. Met bewijzen wordt bedoeld dat je laat zien, volgens de regels van de logica, dat het niet anders kan zijn dan dat het programma goed werkt. Er bestaan al vrij lang computerprogramma's die hierbij kunnen helpen: de bewijsassistenten. Helaas zijn deze bewijsassistenten gericht op wiskundigen en logici, en niet op programmeurs. Men kan dingen bewijzen over een programma, maar dan wel in een andere taal dan de programmeertaal. Programmeren in het Nederlands, bewijzen in het Chinees. Geen praktische combinatie dus. Er wordt dan ook niet zo gek veel bewezen.

Om dit te verhelpen is er in Nijmegen een bewijsassistent ontwikkeld die wel een voor programmeurs bekende taal spreekt. Bijna dan. Programmeren in het Nederlands, bewijzen in het Surinaams zogezegd. Zo ontbraken er nog een aantal niet essentiele maar wel ontzettend handige eigenschappen van de programmeertaal. Één daarvan is "overloading", wat staat voor de mogelijkheid om een stuk code dat voor verschillende situaties steeds herhaald moet worden, terug te brengen tot een veel kleiner stuk voor een beperkt aantal gevallen waaruit de code voor andere gevallen automatisch gegenereerd kan worden. Het gevolg was dat de programmeur de code wel kort kon opschrijven, maar het bewijzen toch voor alle mogelijke situaties afzonderlijk moest doen. Dat kon niet de bedoeling zijn.

Goed nieuws! In dit project is dit gemis verholpen. Het bewijzen kan nu ook via de korte weg: in één keer voor alle mogelijke situaties. Dit kan zelfs zo uitgebreid worden dat de bewijzer niet eens ziet dat we hiervoor iets speciaals doen. Het bewijzen kan dan in het Vlaams in plaats van het Surinaams. Nog niet perfect, maar wel een stuk beter. De kans dat de programmeur de moeite neemt iets over zijn programma te bewijzen neemt hiermee toe. En zo komt de wereld waarin de mobiele telefoon het altijd doet, de PC nooit crasht en de wasmachine weer gewoon wast, een heel klein stukje dichterbij.

E Paper for TFP 2004

Paper for TFP 2004 E

Proof Support for General Type Classes

Ron van Kesteren Marko van Eekelen Maarten de Mol

Nijmegen Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
`rkestere@sci.ru.nl, {M.vanEekelen, M.deMol}@niii.ru.nl`

Abstract

We present a proof rule and an effective tactic for proving properties about HASKELL type classes by proving them for the available instance definitions. This is not straightforward, because instance definitions may depend on each other. The proof assistant ISABELLE handles this problem for single parameter type classes by structural induction on types. However, this does not suffice for an effective tactic for more complex forms of overloading. We solve this using an induction scheme derived from the instance definitions. The tactic based on this rule is implemented in the proof assistant SPARKLE.

Keywords: Functional Programming, Theorem Proving, Type Classes

E.1 Introduction

It is often stated that formulating properties about programs increases robustness and safety, especially when formal reasoning is used to prove these properties. Robustness and safety are becoming increasingly important considering the current dependence of society on technology. Research on formal reasoning has spawned many general purpose proof assistants, such as COQ [5], ISABELLE [14], and PVS [16]. Unfortunately, these general purpose tools are geared towards mathematicians and are hard to use when applied to more practical domains such as actual programming languages.

Because of this, proof assistants have been developed that are geared towards specific programming languages. This allows proofs to be conducted

```
class Eq a where
  (==) ::  a -> a -> Bool

instance Eq Int where
  x == y = predefinedeqint x y

instance Eq Char where
  x == y = predefinedeqchar x y

instance (Eq a) => Eq [a] where
  []     == []     = True
  (x:xs) == []     = False
  []     == (y:ys) = False
  (x:xs) == (y:ys) = x == y && xs == ys
```

Figure E.1: A type class for equality in Haskell

on the source program using specifically designed proof rules. Functional languages are especially suited for formal reasoning because they are referentially transparent. Examples of proof assistants for functional languages are Evt [15] for Erlang [2], Sparkle [4] for Clean [17], and Era [20] for Haskell [8].

E.1.1 Type classes

A feature that is commonly found in functional programming languages is overloading structured by *type classes* [18]. Type classes essentially are groups of types, the class *instances*, for which certain operations, the class *members*, are implemented. These implementations are created from the available instance definitions and may be different for each instance. The type of an instance definition is called the *instance head.* The equality operator will be used as a running example throughout this paper (figure E.1).

In the most basic case, type classes have only one parameter and instance heads are flat, that is, a single constructor applied to a list of type variables. Furthermore, no two instance definitions may overlap.

Several significant extensions have been proposed, such as multiple parameters [9], overlapping instances, and instantiation with constructors [7], that have useful applications such as collections, coercion, isomorphisms and mapping. In this paper, the term *general type classes* is used for systems of type classes that support these extensions and non-flat instance heads. Figure E.2 shows a multi parameter class for the symmetric operation `eq2`.

An important observation regarding type classes is that, in general, the defined instances should be semantically related. For example, all instances of the equality operator usually implement an equivalence relation. These properties can be proven for all instances at once by proving them for the available instance definitions. Unfortunately, this is not straightforward because the instance definitions may depend on each other and hence so will the proofs. For example, equality on lists is only symmetric if equality on the list members is

```
class Eq2 a b where
  eq2 ::  a -> b -> Bool where

instance Eq2 Int Int where
  eq2 x y = x == y

instance Eq2 Char Char where
  eq2 x y = x == y

instance (Eq2 a c, Eq2 b c) => Eq2 (a, b) [c] where
  eq2 (x, y) [u, v] = eq2 x u && eq2 y v
  eq2 x y           = False

instance (Eq2 a c, Eq2 b c) => Eq2 [c] (a, b) where
  eq2 x y = eq2 y x
```

Figure E.2: A multi parameter class in Haskell

so as well.

E.1.2 Contributions

The only proof assistant with special support for overloading that we know of is Isabelle [13, 19], which essentially supports single parameter type classes and a proof rule for it based on structural induction on types. However, we show that for general type classes, an effective tactic is not easily derived when structural induction is used. We use an induction scheme on types based on the instance definitions to solve this problem. Using this induction scheme, a proof rule and tactic are defined that are both strong enough and effective.

As a proof of concept, we have implemented the tactic in the proof assistant Sparkle for the programming language Clean. The results, however, are generally applicable and can, for example, also be used for Haskell and Isabelle, if Isabelle would support the specification of general type classes. In fact, the examples here are presented using Haskell syntax. Sparkle is dedicated to Clean, but can also be used to prove properties about Haskell programs by translating them to Clean using the Hacle translator [12].

E.1.3 Outline

The rest of this paper is structured as follows. First, the proof assistant Sparkle is presented (section E.2). Then, basic definitions for instance definitions, evidence values, and class constrained properties are introduced (section E.3). After showing why structural induction does not suffice (section E.4), the proof rule and tactic based on the instance definitions are defined (section E.5) and extended to multiple class constraints (section E.6). We end with a discussion of the implementation (section E.7), related and future work (section E.8), and a summary of the results (section E.9).

E.2 Sparkle

The need for this work arose whilst improving the proof support for type classes in SPARKLE. SPARKLE is a proof assistant specifically geared towards CLEAN, which means that it can reason about CLEAN concepts using rules based on CLEAN's semantics. Properties are specified in a first order predicate logic extended with equality on expressions. An example of this, using a slightly simplified syntax, is:

example: $\forall_{\mathtt{n:Int|n}\neq\bot}\forall_{\mathtt{a}}\forall_{\mathtt{xs:[a]}}[\mathtt{take\ n\ xs\ ++\ drop\ n\ xs\ =\ xs}]$

These properties can be proven using *tactics*, which are user friendly operations that transform a property into a number of logically stronger properties, the *proof obligations* or *goals*, that are easier to prove. A tactic is the implementation of (a combination of) theoretically sound *proof rules*. Whereas in general a proof rule is theoretically simple but not very prover friendly, a tactic is prover friendly but often theoretically more complex. The proof is complete when all remaining proof obligations are trivial. Some useful tactics are, for example, reduction of expressions, induction on expression variables, and rewriting using hypotheses.

In SPARKLE, properties that contain member functions can only be proven for specific instances of that function. For example:

sym$_{\mathtt{[Int]}}$: $\forall_{\mathtt{x:[Int]}}\forall_{\mathtt{y:[Int]}}[\mathtt{x\ ==\ y} \rightarrow \mathtt{y\ ==\ x}]$

can be easily proven by induction on lists using symmetry of equality on integers. Proving that something holds for *all* instances, however, is not possible in general. Consider for example symmetry of equality:

sym: $\forall_{\mathsf{a}}[\mathsf{Eq} :: \mathsf{a} \Rightarrow \forall_{\mathtt{x:a}}\forall_{\mathtt{y:x}}[\mathtt{x\ ==\ y} \rightarrow \mathtt{y\ ==\ x}]]$

where Eq :: a denotes the, previously not available, constraint that equality must be defined for type a. This property can be split into a property for every instance definition, which gives among others the property for the instance for lists:

sym$_{\mathsf{[a]}}$: $\forall_{\mathsf{a}}[\mathsf{Eq} :: \mathsf{a} \Rightarrow \forall_{\mathtt{x:[a]}}\forall_{\mathtt{y:[a]}}[\mathtt{x\ ==\ y} \rightarrow \mathtt{y\ ==\ x}]]$

It is clear that this property is true as long as it is true for instance a. Unfortunately, this hypothesis is not available. Using an approach based on induction, however, we may be able to assume the hypotheses for all instances the instance definition depends on, and hence will be able to prove the property.

Internally, SPARKLE translates type classes to *evidence values* or *dictionaries* [18], that make the use of overloading explicit. The evidence value for a class constraint c :: a is the evidence that there is an (implementation of the) instance of class c for type a. Hence, an evidence value exists if and only if the class constraint is satisfied. As usual, we will use the implementation itself

```
eqint ::  Int -> Int -> Bool
eqint = predefinedeqint

eqchar ::  Char -> Char -> Bool
eqchar = predefinedeqchar

eqlist ::  (a -> a -> Bool) -> ([a] -> [a] -> Bool)
eqlist ev []     []     = True
eqlist ev (x:xs) []     = False
eqlist ev []     (y:ys) = False
eqlist ev (x:xs) (y:ys) = ev x y && eqlist ev xs ys
```

Figure E.3: Translation of figure E.1

as the evidence value. A program is translated by converting all instance definitions to functions (distinct names are created by suffixes). In expressions, the evidence value is substituted for member applications. When functions require certain classes to be defined, the evidence values for these constraints are passed as a parameter. Figure E.3 shows an example of the result of the translation of the equality class from figure E.1.

E.3 Preliminaries

Instead of defining a proof rule that operates on the example properties from section E.2, we define both instances and properties at the level that explicitly uses evidence values. In this section, basic definitions for instance definitions, evidence values, and class constrained properties are given.

E.3.1 Instance definitions

Because we intend to support constructor classes, types are formalized by a language of constructors [7]:

$$\tau ::= \alpha \mid \mathcal{X} \mid \tau\ \tau'$$

where α and $\mathcal{X}$ range over a given set of type variables and type constructors respectively. For example, τ can be `Int`, `[Int]`, and `Tree Char`, but also the `[]`, `Tree`, and `->` constructors that take types as an argument and yield a list, tree, or function type respectively. $TV(\tau)$ denotes the set of type variables occurring in τ. The set of closed types $\mathcal{T}^c$ is the set of types for which $TV(\tau)$ is empty.

Predicates are used to indicate that an instance of a certain class exists. An instance can be identified by an instantiation of the class parameters. The predicate $c :: \vec{\tau}$ denotes that there is an instance of the class c for instantiation $\vec{\tau}$ of the class parameters. For example, `Eq` :: `(Int, Int)` and `Eq` :: `[Int]` denote that there is an instance of the `Eq` class for types `(Int, Int)` and `[Int]` respectively:

$$\pi ::= c :: \vec{\tau}$$

Because these predicates are used to constrain types to a certain class, they are called *class constraints*. Class constraints in which only type variables occur in the type, for example $\texttt{Eq} :: \texttt{a}$, are called *simple*. For reasons of simplicity, it is assumed that all type variables that occur in a class constraint are distinct.

Without loss of generality, throughout this paper we restrict ourselves to type classes that have only one member and no subclasses. Multiple members and subclasses can be supported using records of expressions for the evidence values. An instance definition:

$$\mathsf{inst}\ \vec{\pi} \Rightarrow c :: \vec{\tau} = e$$

defines an instance $\vec{\tau}$ of class c for types that satisfy class constraints $\vec{\pi}$. The instance definition provides the translated expression e for the class member c. The functions $\mathit{Head}(\mathsf{inst}\ c :: \vec{\pi} \Rightarrow \tau = e) = \tau$ and $\mathit{Context}(\mathsf{inst}\ c :: \vec{\pi} \Rightarrow \tau = e) = \vec{\pi}$ will be used to retrieve the instance head and context respectively.

The program context ψ, that contains the function and class definitions, also includes the available instance definitions. The function $\mathit{Idefs}_{\psi}(c)$ returns the set of instance definitions of class c defined in program ψ.

E.3.2 Evidence values

From the translation from type classes to evidence values, as briefly summarized in section E.2, the rule for evidence creation is important for our purpose. Two definitions are required before it can be defined.

Firstly, because instance definitions are allowed to overlap, a mechanism is needed that chooses between them. Since the exact definition is not important for our purpose, we assume that the function $\mathit{Ai}_{\psi}(c :: \vec{\tau})$ determines the most specific instance definition applicable for instance $\vec{\tau}$ of class c. Ai_{ψ} is also defined for types that contain variables as long as it can be determined which instance definition should be applied.

Secondly, the *dependencies* of an instance are the instances it depends on:

$$\mathit{Deps}(c :: \vec{\tau}, i) = *_{\mathit{Head}(i) \to \vec{\tau}}(\mathit{Context}(i))$$

where $*_{\vec{\tau} \to \vec{\tau}'}$ denotes the substitutor that maps the type variables in $\vec{\tau}$ such that $*(\vec{\tau}) = \vec{\tau}'$. When i is not provided, $\mathit{Ai}_{\psi}(c :: \vec{\tau})$ is assumed for it.

Evidence values are now straightforwardly created by applying the expression of the most specific instance definition to the evidence values of its dependencies:

$$\frac{\begin{array}{c}\mathit{Deps}(\pi) = \langle \pi_1, \ldots, \pi_n \rangle \\ \mathit{Ai}_{\psi}(\pi) = \mathsf{inst}\ c :: \vec{\pi}' \Rightarrow \vec{\tau}' = e\end{array}}{\mathit{Ev}_{\psi}(\pi) = e\ \mathit{Ev}_{\psi}(\pi_1)\ \ldots\ \mathit{Ev}_{\psi}(\pi_n)}$$

In proofs, evidence values will be created assuming the evidence values for the dependencies are already assigned to expression variables:

$$\frac{\begin{array}{c}\mathit{Deps}(\pi, i) = \langle \pi_1, \ldots, \pi_n \rangle \\ i = \mathsf{inst}\ c :: \vec{\pi}' \Rightarrow \vec{\tau}' = e\end{array}}{\mathit{Ev^p}_{\psi}(\pi, i) = e\ \texttt{ev}_{\pi_1}\ \ldots\ \texttt{ev}_{\pi_n}}$$

assuming that the evidence for π is assigned to the variable $\texttt{ev}_\pi$. A specific instance definition i can be provided, because $Ai_\psi(\pi)$ might not be known in proofs.

E.3.3 Class constrained properties

In Sparkle, properties are formalized by a first order predicate logic extended with equality on expressions. The equality on expressions is designed to handle infinite and undefined expressions well.

We extend these properties with class constraints, that can be used to constrain types to a certain class. These properties will be referred to as *class constrained properties*. For example, consider symmetry and transitivity of equality:

$$\textbf{sym:} \quad \forall_{\texttt{a}}[\texttt{Eq} :: \texttt{a} \Rightarrow \forall_{\texttt{x,y:a}}[\texttt{ev}_{\texttt{Eq::a}}\ \texttt{x}\ \texttt{y} \rightarrow \texttt{ev}_{\texttt{Eq::a}}\ \texttt{y}\ \texttt{x}]]$$

$$\textbf{trans:} \quad \forall_{\texttt{a}}[\texttt{Eq} :: \texttt{a} \Rightarrow \forall_{\texttt{x,y,z:a}}[\texttt{ev}_{\texttt{Eq::a}}\ \texttt{x}\ \texttt{y} \rightarrow \texttt{ev}_{\texttt{Eq::a}}\ \texttt{y}\ \texttt{z} \rightarrow \texttt{ev}_{\texttt{Eq::a}}\ \texttt{x}\ \texttt{z}]]$$

The property $c :: \vec{\tau} \Rightarrow p$ means that in property p it is assumed that $\vec{\tau}$ is an instance of class c and the evidence value for this class constraint is assigned to $\texttt{ev}_{c::\vec{\tau}}$. Thus, the semantics of the property $\pi \Rightarrow p$ is defined as $p_{[\texttt{ev}_\pi \mapsto Ev_\psi(\pi)]}$.

E.4 Structural induction

The approach for proving properties that contain overloaded identifiers taken in Isabelle essentially is structural induction on types. In this section it is argued that the proof rule for general type classes should use another induction scheme.

Structural induction on types seems an effective approach because it gives more information about the type of an evidence value. This information can be used to expand evidence values. For example, $\texttt{ev}_{\texttt{Eq::[a]}}$ can be expanded to `eqlist` $\texttt{ev}_{\texttt{Eq::a}}$ (see figure E.3).

$$\frac{\begin{array}{c} Ai_\psi(\pi) = i \\ \forall_{TV(\pi)}[Deps(\pi) \Rightarrow p(Ev^p{}_\psi(\pi))] \end{array}}{\forall_{TV(\pi)}[\pi \Rightarrow p(\texttt{ev}_\pi)]} \quad \textbf{(expand)}$$

More importantly, structural induction allows the property to be assumed for structurally smaller types. Ideally the hypothesis should be assumed for all dependencies on the same class. Unfortunately, structural induction does not always allow this for multi parameter classes.

Consider for example the multi parameter class in figure E.2. The instance of `Eq2` for `[Int] (Char, Char)` depends on the instance for `Char Int`, which is not structurally smaller because `Char` is not structurally smaller than `[Int]`, and `Int` is not structurally smaller than `(Char, Char)`. Hence, the hypothesis cannot be assumed for this dependency. This problem can be solved by basing the induction scheme on the instance definitions.

E.5 Induction on instances

The induction scheme proposed in the previous section can be used on the set of defined instances of a class. In this section, a proof rule and tactic that use this scheme are defined and applied to some examples.

E.5.1 Proof rule and tactic

We first define the set of instances of a class and an order based on the instance definitions on it. The well-founded induction theorem applied to the defined set and order yields the proof rule. Then, the tactic is presented that can be derived from this rule.

Remember that the instances of a class are identified by sequences of closed types. $\vec{\tau}$ is an instance of class c if an evidence value can be generated for the class constraint $c :: \vec{\tau}$. Hence, the set of instances of class c can be defined as:

$$Inst_\psi(c) = \{\vec{\tau} \mid \forall_{c'::\vec{\tau}' \in Deps(c::\vec{\tau})}[\vec{\tau}' \in Inst_\psi(c')]\}$$

For example, $Inst_\psi(\texttt{Eq}) = \{\texttt{Int}, \texttt{Char}, \texttt{[Int]}, \texttt{[Char]}, \texttt{[[Int]]}, \ldots\}$.

An order on this set is straightforwardly defined. Because the idea is to base the order on the instance definitions, an instance $\vec{\tau}'$ is considered one step smaller than $\vec{\tau}$ if the evidence for $\vec{\tau}$ depends on the evidence for $\vec{\tau}'$, that is, if $c :: \vec{\tau}'$ is a dependency of the most specific instance definition for $c :: \vec{\tau}$. For example, $\texttt{Int} <^1_{(\psi,\texttt{Eq})} \texttt{[Int]}$ and $\texttt{[Char]} <^1_{(\psi,\texttt{Eq})} \texttt{[[Char]]}$.

$$\vec{\tau} <^1_{(\psi,c)} \vec{\tau}' \Leftrightarrow c :: \vec{\tau}' \in Deps(c :: \vec{\tau})$$

Note that there is a specific set of instances for each class and therefore also a specific order for each class.

Well-founded induction requires a well-founded partial order, for which we use the reflexive transitive closure of $<^1_{(\psi,c)}$. It can be easily derived from the way evidence values are generated that this is indeed a well-founded partial order. Applying this order, $\leq_{(\psi,c)}$, to the well-founded induction theorem yields the following proof rule:

$$\frac{\forall_{\vec{\tau} \in Inst_\psi(c)}[\forall_{\vec{\tau}' \leq_{(\psi,\texttt{C})} \vec{\tau}}[p(\vec{\tau}')] \rightarrow p(\vec{\tau})]}{\forall_{\vec{\alpha} \in Inst_\psi(c)}[p(\vec{\alpha})]} \quad \textbf{(inst-rule)}$$

Rewriting the proof rule using the definitions of $Inst_\psi(c)$, $\leq_{(\psi,c)}$, evidence creation, and class constrained properties results in a tactic that can be directly applied to class constrained properties. For all class constraints $c :: \vec{\alpha}$:

$$\frac{\begin{array}{l}\forall_{i\in \mathit{Idefs}_\psi(c)}\forall_{\mathit{Head}(i)\in\langle\mathcal{T}^c\rangle}\\ \quad[\ \mathit{Deps}(c :: \mathit{Head}(i), i)\\ \qquad\Rightarrow \forall_{c'::\vec{\tau}'\in \mathit{Deps}(c::\mathit{Head}(i),i)}[c = c' \Rightarrow p(\mathtt{ev}_{c::\vec{\tau}'}, \vec{\tau}')]\\ \qquad\rightarrow p(\mathit{Ev}^p{}_\psi(c :: \mathit{Head}(i), i), \mathit{Head}(i))\\ \quad]\end{array}}{\forall_{\vec{\alpha}\in\langle\mathcal{T}^c\rangle}[c :: \vec{\alpha} \Rightarrow p(\mathtt{ev}_{c::\vec{\alpha}}, \vec{\alpha})]}\quad\textbf{(inst-tactic)}$$

where it is assumed that all variables in *Head*(i) are fresh. When the tactic is applied to a class constrained property, it generates a proof obligation for every available instance definition with hypotheses for all dependencies on the same class.

E.5.2 Results

The result is both a proof rule and a user friendly tactic. The tactic is nicely illustrated by symmetry of equality (figure E.1 and E.3). When **(inst-tactic)** is applied to:

sym: $\forall_{\mathsf{a}}[\mathsf{Eq} :: \mathsf{a} \Rightarrow \forall_{\mathtt{x}:\mathsf{a}}\forall_{\mathtt{y}:\mathsf{a}}[\mathtt{ev}_{\mathtt{Eq::a}}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{ev}_{\mathtt{Eq::a}}\ \mathtt{y}\ \mathtt{x}]]$

it generates the following three proof obligations (one for each instance definition):

sym$_{\mathtt{Int}}$: $\forall_{\mathtt{x:Int}}\forall_{\mathtt{y:Int}}[\mathtt{eqint\ x\ y} \rightarrow \mathtt{eqint\ y\ x}]$

sym$_{\mathtt{Char}}$: $\forall_{\mathtt{x:Char}}\forall_{\mathtt{y:Char}}[\mathtt{eqchar\ x\ y} \rightarrow \mathtt{eqchar\ y\ x}]$

sym$_{[\mathsf{a}]}$:
$$\begin{array}{l}\forall_{\mathsf{a}}\ [\ \mathsf{Eq} :: \mathsf{a}\\ \quad\Rightarrow \forall_{\mathtt{x}:\mathsf{a}}\forall_{\mathtt{y}:\mathsf{a}}[\mathtt{ev}_{\mathtt{Eq::a}}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{ev}_{\mathtt{Eq::a}}\ \mathtt{y}\ \mathtt{x}]\\ \quad\rightarrow \forall_{\mathtt{x}:[\mathsf{a}]}\forall_{\mathtt{y}:[\mathsf{a}]}[\mathtt{eqlist}\ \mathtt{ev}_{\mathtt{Eq::a}}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{eqlist}\ \mathtt{ev}_{\mathtt{Eq::a}}\ \mathtt{y}\ \mathtt{x}]\\ \quad]\end{array}$$

which are easily proven using the already available tactics.

The previous step could also have been taken using a tactic based on structural induction on types. However, **(inst-tactic)** can also assume hypotheses for dependencies that are possibly not structurally smaller. Consider for example the symmetry of `eq2` in figure E.2:

sym2:
$$\begin{array}{l}\forall_{\mathsf{a,b}}\ [\ \mathsf{Eq2} :: \mathsf{a}\ \mathsf{b} \Rightarrow \mathsf{Eq2} :: \mathsf{b}\ \mathsf{a}\\ \quad\Rightarrow \forall_{\mathtt{x}:\mathsf{a}}\forall_{\mathtt{y}:\mathsf{b}}[\mathtt{ev}_{\mathtt{Eq2::a\ \ b}}\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{ev}_{\mathtt{Eq2::b\ \ a}}\ \mathtt{y}\ \mathtt{x}]\\ \quad]\end{array}$$

Applying **(inst-tactic)** to this property generates a proof obligation for every instance definition, including one for the fourth instance of `Eq2` in figure E.2, where `eq2list` is the translation of that instance definition:

$$
\begin{aligned}
\textbf{sym2}_{[\mathsf{a}]}\textbf{:}\quad & \forall_{\mathsf{a,b,c}} \\
& [\ \mathsf{Eq2} :: \mathsf{b\ a} \Rightarrow \mathsf{Eq2} :: \mathsf{c\ a} \\
& \quad \Rightarrow [\mathsf{Eq2} :: \mathsf{a\ b} \Rightarrow \forall_{\mathsf{x:b}}\forall_{\mathsf{y:a}}[\mathtt{ev}_{\mathsf{Eq2::b\ \ a}}\ \mathtt{x\ y} \rightarrow \mathtt{ev}_{\mathsf{Eq2::a\ \ b}}\ \mathtt{y\ x}]] \\
& \quad \rightarrow [\mathsf{Eq2} :: \mathsf{a\ c} \Rightarrow \forall_{\mathsf{x:c}}\forall_{\mathsf{y:a}}[\mathtt{ev}_{\mathsf{Eq2::c\ \ a}}\ \mathtt{x\ y} \rightarrow \mathtt{ev}_{\mathsf{Eq2::a\ \ c}}\ \mathtt{y\ x}]] \\
& \quad \rightarrow \mathsf{Eq2} :: (\mathsf{b,c})\ [\mathsf{a}] \Rightarrow \forall_{\mathsf{x:[a]}}\forall_{\mathsf{y:(b,c)}}[\\
& \qquad\qquad\qquad \mathtt{eq2list}\ \mathtt{ev}_{\mathsf{Eq2::b\ \ a}}\ \mathtt{ev}_{\mathsf{Eq2::c\ \ a}}\ \mathtt{x\ y} \\
& \qquad\qquad\qquad \rightarrow \mathtt{ev}_{\mathsf{Eq2::(b,c)\ \ [a]}}\ \mathtt{y\ x}] \\
&]
\end{aligned}
$$

In this proof obligation, the hypotheses could not have been assumed when using structural induction on types (see section E.4), hence our tactic is useful in more cases.

E.6 Multiple class constraints

The proof rule and tactic presented in the previous section work well when the property has only one class constraint. In case of multiple class constraints, however, the rules might not be powerful enough. In this section it is shown that this problem does indeed occur. Therefore, a more general proof rule and tactic are defined and applied to some examples.

The problem

Consider the two class definitions in figure E.4. The translated instance definitions are respectively called `fint`, `flist`, `ftree`, `gint`, `gtree`, and `glist` at the level of dictionaries. Given the property:

$$\textbf{same:}\quad \forall_{\mathsf{a}}[\mathtt{f} :: \mathsf{a} \Rightarrow \mathsf{g} :: \mathsf{a} \Rightarrow \forall_{\mathsf{x:a}}[\mathtt{ev}_{\mathsf{f::a}}\ \mathtt{x}\ =\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]]$$

Applying **(inst-tactic)** yields among others the goal:

$$\textbf{same}_{[\mathsf{a}]}\textbf{f:}\quad \forall_{\mathsf{a}}[\mathsf{g} :: [\mathsf{a}] \Rightarrow \forall_{\mathsf{x:[a]}}[\mathtt{flist}\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}\ =\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]]$$

This goal has a non-simple class constraint, which can only be removed by evidence expansion **(expand)**, resulting in:

$$
\begin{aligned}
\textbf{same}_{[\mathsf{a}]}\textbf{f':}\ \forall_{\mathsf{a}}[\mathtt{f} :: \mathsf{a} \Rightarrow \mathsf{g} :: \mathsf{a} \Rightarrow \forall_{\mathsf{x:[a]}}[&\mathtt{flist}\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x} \\
&=\ \mathtt{glist}\ \mathtt{ev}_{\mathsf{f::a}}\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]]
\end{aligned}
$$

After some reduction steps, this can be transformed into:

$$
\begin{aligned}
\textbf{same}_{[\mathsf{a}]}\textbf{f'':}\ \forall_{\mathsf{a}}[\mathtt{f} :: \mathsf{a} \Rightarrow \mathsf{g} :: \mathsf{a} \Rightarrow \forall_{\mathsf{x:[a]}}[&\mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}\ \mathtt{==}\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x} \\
&=\ \mathtt{ev}_{\mathsf{f::a}}\ \mathtt{x}\ \mathtt{==}\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}]]
\end{aligned}
$$

This proof obligation is true when $\mathtt{ev}_{\mathsf{f::a}}\ \mathtt{x}\ =\ \mathtt{ev}_{\mathsf{g::a}}\ \mathtt{x}$. Unfortunately, the induction scheme did not allow us to assume this hypothesis. Since this problem

```
data Tree a = Leaf | Node a (Tree a) (Tree a)

class f a where f ::  a -> Bool

instance f Int where
  f x = x == x

instance (g a) => f [a] where
  f []     = True
  f (x:xs) = g x == g x

instance (f a, g a) => f (Tree a) where
  f Leaf         = True
  f (Node x l r) = f x == g x

class g a where g ::  a -> Bool

instance g Int where
  g x = x == x

instance (f a) => g (Tree a) where
  g Leaf         = True
  g (Node x l r) = f x == f x

instance (g a, f a) => g [a] where
  g []     = True
  g (x:xs) = g x == f x
```

Figure E.4: Problematic class definitions

is caused by the fact that the type variables occur in more than one class constraint, the natural solution is to take multiple class constraints into account in the induction scheme.

E.6.1 Proof rule and tactic

We take the same approach as in the previous section. We first define the set of instances, the order, the proof rule and the tactic. Then, in section E.6.2, it is shown that the new tactic solves the problem.

First, the set of type sequences that are instances of all classes that occur in a list of class constraints is defined. $\vec{\tau}$ is a member of the set if all class constraints $\vec{\pi}$ are satisfied when all variables $TV(\vec{\pi})$ are replaced by the corresponding type from $\vec{\tau}$. We assume here that $TV(\bar{\pi})$ is a linearly ordered, for example lexicographically, sequence and that the elements of $\bar{\tau}$ are in the corresponding order. For example, $\mathit{SetInst}_\psi(\texttt{f} :: \texttt{a}, \texttt{g} :: \texttt{a}) = \{\texttt{Int}, \texttt{[Int]}, \texttt{Tree Int}, \texttt{[[Int]]}, \ldots\}$.

$$\mathit{SetInst}_\psi(\vec{\pi}) = \{\vec{\tau} \mid \forall_{c::\vec{\alpha'} \in \vec{\pi}}[*_{TV(\vec{\pi}) \to \vec{\tau}}(\vec{\alpha'}) \in \mathit{Inst}_\psi(c)]\}$$

The order on this set is an extension of the order for single class constraints to sets. A sequence of types $\vec{\tau}$ is considered one step smaller than $\vec{\tau'}$ if $*_{TV(\vec{\pi}) \to \vec{\tau}}(\vec{\pi})$ is a subset of the dependencies of $*_{TV(\vec{\pi}) \to \vec{\tau}}(\vec{\pi})$. For example, $\texttt{[Int]} <^1_{(\psi, \langle \texttt{f::a,g::a} \rangle)} \texttt{([[Int]])}$ because $\{\texttt{f} :: \texttt{[Int]}, \texttt{g} :: \texttt{[Int]}\}$ is a subset of $\mathit{Deps}(\texttt{g} :: \texttt{[[Int]]}) \cup \mathit{Deps}(\texttt{f} :: \texttt{[[Int]]})$. Here, sequences of class constraints are lifted to sets when required:

$$\vec{\tau} <^1_{\psi, \vec{\pi}} \tau' \Leftrightarrow *_{TV(\vec{\pi}) \to \vec{\tau}}(\vec{\pi}) \subseteq \bigcup_{\pi \in \vec{\pi}} [\mathit{Deps}(*_{TV(\vec{\pi}) \to \vec{\tau}'}(\pi))])$$

Again, it can be derived from the evidence creation that the reflexive transitive closure of this order, $\leq_{(\psi, \vec{\pi})}$, is a well-founded partial order.

Applying the well-founded induction theorem to this set and order yields the proof rule for multiple class constraints. For every sequence of simple class constraints $\vec{\pi}$:

$$\frac{\forall_{\vec{\tau} \in \mathit{SetInst}_\psi(\vec{\pi})}[\forall_{\vec{\tau}' \leq_{(\psi, \vec{\pi})} \vec{\tau}}[p(\vec{\tau}')] \to p(\vec{\tau})]}{\forall_{\vec{\tau} \in \mathit{SetInst}_\psi(\vec{\pi})}[p(\vec{\tau})]} \quad \textbf{(multi-rule)}$$

Because multiple class constraints are involved, defining the final tactic is a bit more complicated. Instead of all instance definitions, every combination of instance definitions, one for each class constraint, has to be tried. All of these instance definitions make assumptions about the types of the type variables, and these assumptions should be unifiable. Therefore, we define the most general unifier that takes the sharing of type variables across class constraints into account:

$$\begin{aligned}&\mathit{SetMgu}(\langle c_1 :: \vec{\alpha}_1, \ldots, c_n :: \vec{\alpha}_n \rangle, \langle \tau_1, \ldots, \tau_n \rangle) = * \Leftrightarrow \\ &\qquad \forall_{1 \leq i \leq n}[*(\vec{\alpha}_i) = \tau_1] \ \wedge \ \forall_{*'}[\forall_{1 \leq i \leq n}[*'(\vec{\alpha}_i) = \tau_i] \Rightarrow \exists *''[*' = *'' \circ *]]\end{aligned}$$

Furthermore, for readability of the final tactic, some straightforward extensions of existing definitions to vectors are used:

$$\begin{array}{lcl}
\mathit{Idefs}_{\psi}(\langle \pi_1, \ldots, \pi_n \rangle) & = & \{ i_1, \ldots, i_n \mid i_j \in \mathit{Idefs}_{\psi}(\pi_j) \} \\
\mathit{Head}(\langle i_1, \ldots, i_n \rangle) & = & \langle \mathit{Head}(i_1), \ldots, \mathit{Head}(i_n) \rangle \\
\mathit{Ev^p}_{\psi}(\langle \pi_1, \ldots, \pi_n \rangle, \langle i_1, \ldots, i_n \rangle) & = & \langle \mathit{Ev^p}_{\psi}(\pi_1, i_1), \ldots, \mathit{Ev^p}_{\psi}(\pi_n, i_n) \rangle \\
\mathtt{ev}_{\langle \pi_1, \ldots, \pi_n \rangle} & = & \langle \mathtt{ev}_{\pi_1}, \ldots, \mathtt{ev}_{\pi_n} \rangle \\
\mathit{Deps}(\langle \pi_1, \ldots, \pi_n \rangle, \langle i_1, \ldots, i_n \rangle) & = & \langle \mathit{Deps}(\pi_1, i_1), \ldots, \mathit{Deps}(\pi_n, i_n) \rangle
\end{array}$$

Finally, using the presented definitions, evidence creation, class constrained properties, and the proof rule, the tactic can be defined. For every sequence of simple class constraints $\vec{\pi}$:

$$\frac{\begin{array}{l}
\forall_{\vec{\imath} \in \mathit{Idefs}_{\psi}(\vec{\pi})} \exists_{* \mid * = \mathit{SetMgu}(\vec{\pi}, \mathit{Head}(\vec{\imath}))} \forall_{*(\mathit{Head}(\vec{\imath})) \in \langle \mathcal{T}^c \rangle} \\
\quad [\ \mathit{Deps}(*(\vec{\pi}), \vec{\imath}) \\
\qquad \Rightarrow \forall_{*' \mid *'(\vec{\pi}) \subseteq \mathit{Deps}(*(\vec{\pi}), \vec{\imath})} [p(\mathtt{ev}_{*'(\vec{\pi})}, *'(TV(\vec{\pi})))] \\
\qquad \rightarrow p(\mathit{Ev^p}_{\psi}(*(\vec{\pi}), \vec{\imath}), *(\mathit{Head}(\vec{\imath}))) \\
\quad]
\end{array}}{\forall_{TV(\vec{\pi})} [\vec{\pi} \Rightarrow p(\mathtt{ev}_{\vec{\pi}}, TV(\vec{\pi}))]} \quad \textbf{(multi-tactic)}$$

Note that applying this tactic may result in non-simple class constraints when non-flat instance types are used. For non-simple class constraints, the induction tactics cannot be applied, but the **(expand)** rule might be used. However, in practice most instance definitions will have flat types.

This solution for multiple class constraints has some parallels to the constraint set satisfiability problem (CS-SAT), the problem of determining if there are types that satisfy a set of class constraints. The general CS-SAT problem is undecidable. However, recently, an algorithm was proposed [3] that essentially tries to create a type that satisfies all constraints by trying all combinations of instance definitions, as we have been doing in our tactic.

E.6.2 Results

In this section, we have generalized the proof rule and tactic from section E.5 to multiple class constraints. In case of a single class constraint, the new rules behave exactly the same as **(inst-rule)** and **(inst-tactic)**. However, now we can apply **(multi-tactic)** to multiple class constraints at once. Given the previously problematic property:

same: $\forall_{\mathsf{a}}[\mathtt{f} :: \mathsf{a} \Rightarrow \mathtt{g} :: \mathsf{a} \Rightarrow \forall_{\mathtt{x}:\mathsf{a}}[\mathtt{ev}_{\mathtt{f}::\mathsf{a}}\ \mathtt{x} \ = \ \mathtt{ev}_{\mathtt{g}::\mathsf{a}}\ \mathtt{x}]]$

this yields three proof obligations, one for every unifiable combination of instance definitions:

$$\textbf{same}_{\texttt{Int}}\textbf{:}\quad \forall_{\texttt{x:Int}}[\texttt{fint x} = \texttt{gint x}]$$

$$\begin{aligned}\textbf{same}_{\texttt{[a]}}\textbf{:}\quad & \forall_{\texttt{a}}[\texttt{f} :: \texttt{a} \Rightarrow \texttt{g} :: \texttt{a} \Rightarrow \forall_{\texttt{x:a}}[\texttt{ev}_{\texttt{f::a}}\ \texttt{x} = \texttt{ev}_{\texttt{g::a}}\ \texttt{x}] \\ & \quad \rightarrow \forall_{\texttt{x:[a]}}[\texttt{flist ev}_{\texttt{g::a}}\ \texttt{x} = \texttt{glist ev}_{\texttt{f::a}}\ \texttt{ev}_{\texttt{g::a}}\ \texttt{x}]]\end{aligned}$$

$$\begin{aligned}\textbf{same}_{\texttt{Tree a}}\textbf{:}\quad & \forall_{\texttt{a}}[\texttt{f} :: \texttt{a} \Rightarrow \texttt{g} :: \texttt{a} \Rightarrow \forall_{\texttt{x:a}}[\texttt{ev}_{\texttt{f::a}}\ \texttt{x} = \texttt{ev}_{\texttt{g::a}}\ \texttt{x}] \\ & \quad \rightarrow \forall_{\texttt{x:Tree a}}[\texttt{ftree ev}_{\texttt{f::a}}\ \texttt{ev}_{\texttt{g::a}}\ \texttt{x} = \texttt{gtree ev}_{\texttt{f::a}}\ \texttt{x}]\end{aligned}$$

The goal $\textbf{same}_{\texttt{[a]}}$ (and $\textbf{same}_{\texttt{Tree a}}$) now has a hypothesis that can be used to prove the goal using the already available tactics. Hence, by taking multiple class constraints into account the problem is solved.

E.7 Implementation

As a proof of concept, we have implemented the **(multi-tactic)** tactic extended for multiple members and subclasses in Sparkle. Because of the similarity to the already available induction tactic and the clearness of the code, the implementation of the tactic took very little time. However, to implement the tactic, the typing rules had to be extended. The translation of type classes to dictionaries is only typeable in general using rank-2 polymorphism, which is currently not supported by Sparkle. This was worked around by handling the dictionary creation and selection in a way that hides the rank-2 polymorphism. Ideally, the use of dictionaries should be completely hidden from the user as well.

The tactic has been used to prove, amongst others, the examples in this paper. The implementation is available at:
`http://www.student.kun.nl/ronvankesteren/SparkleGTC.zip`

E.8 Related and future work

As mentioned in section E.1, the general proof assistant Isabelle [14] supports overloading and single parameter type classes. Isabelle's notion of type classes is somewhat different from Haskell's in that it represents types that satisfy certain properties instead of types for which certain values are defined. Nevertheless, the problems to be solved are equivalent. Isabelle [13, 19] uses a proof rule based on structural induction on types, which suffices for the supported type classes. However, if Isabelle would support more extensions, most importantly multi parameter classes, it would be useful to define our proof rule and a corresponding tactic in Isabelle.

Essentially, the implementation of the tactic we proposed extends the induction techniques available in Sparkle. Leonard Lensink proposed and implemented extensions of Sparkle for induction and co-induction for mutually recursive functions and data types [11]. The main goal was to ease proofs by making the induction scheme match the structure of the program. Together with this work this significantly increases the applicability of Sparkle.

Because generics is often presented as an extension of type classes [6], it would be nice to extend this work to generics as well. Currently, in Clean generics are translated to normal type classes where classes are created for every available data type [1]. There is a library for Haskell that generates classes with boilerplate code for every available data type [10]. The tactic presented here can already be used to prove properties about generic functions by working on these generated type classes. However, the property is only proven for the data types that are used in the program and a separate proof is required for each data type. That is, after all, the main difference between normal type classes and generics. Hence, it remains useful to define a proof rule specifically for generics.

E.9 Conclusion

In this paper, we have presented a proof rule for class constrained properties and an effective tactic based on it. Although structural induction on types is theoretically powerful enough, we showed that for an effective tactic an induction scheme should be used that is based on the instance definitions. The tactic is effective, because, using the defined proof rule, it allows all sensible hypotheses to be assumed. The rule and tactic were first defined for single class constraints and then generalized to properties with multiple class constraints.

As a proof of concept, the resulting tactic is implemented in Sparkle for the programming language Clean, but it can also be used for proving properties about Haskell programs. This is, to our knowledge, the first implementation of an effective tactic for general type classes. If Isabelle would support extensions for type classes, the tactic could be implemented in Isabelle as well.

Acknowledgements

We would like to thank Sjaak Smetsers for his suggestions and advice concerning this work, especially on the semantics of type classes in Clean, and Fermín Reig for a valuable discussion on generic proofs at the TFP2004 workshop.

References

[1] A. Alimarine and R. Plasmeijer. A generic programming extension for Clean. In Th. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers*, LNCS 2312, pages 168–185, Älvsjö, Sweden, September 24-26 2001. Springer.

[2] J. L. Armstrong and R. Virding. Erlang – An Experimental Telephony Switching Language. In *XIII International Switching Symposium*, Stockholm, Sweden, May 27 – June 1, 1991.

[3] C. Camarão, L. Figueiredo, and C. Vasconcellos. Constraint-set satisfiability for overloading. In *International Conference on Principles and Practice of Declarative Programming*, Verona, Italy, August 2004.

[4] M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - SPARKLE: A functional theorem prover. In Th. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers*, LNCS 2312, pages 55–71, Älvsjö, Sweden, September 2001.

[5] The Coq development team. *The Coq proof assistant reference manual (version 8.0).* LogiCal Project, 2004.

[6] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2000.

[7] M. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, N.Y., 1993. ACM Press.

[8] S. Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.

[9] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Proceedings of the Second Haskell Workshop*, Amsterdam, June 1997.

[10] R. Lämmel and S. Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37, New Orleans, Januari 2003. ACM.

[11] L. Lensink and M. van Eekelen. Induction and Co-induction in Sparkle. In Hans-Wolfgang Loidl, editor, *Fifth Symposium on Trends in Functional Programming (TFP 2004)*, pages 273–293. Ludwig-Maximilians Universität, München, November 2004.

[12] M. Naylor. Haskell to Clean translation. University of York, 2004.

[13] T. Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. CUP, 1993.

[14] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer, 2002.

[15] T. Noll, L. Fredlund, and D.Gurov. The EVT Erlang verification tool. In *Proceedings of the 7th international Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS 2031, pages 582–585, Stockholm, 2001. Springer.

[16] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[17] M. van Eekelen and R. Plasmeijer. *Concurrent Clean Language Report (version 2.0).* University of Nijmegen, December 2001.

[18] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[19] M. Wenzel. Type classes and overloading in higher-order logic. In E. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, pages 307–322, Murray Hill, New Jersey, 1997.

[20] N. Winstanley. *Era User Manual (version 2.0).* University of Glasgow, 1999.