T U M

I N S T I T U T F Ü R I N F O R M A T I K

A Polymorphic Sort System for Axiomatic Specification Languages

Dieter Nazareth

TUM-I9515
Mai 1995

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-05-1995-I9515-150/1.-FI

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

Fakultät für Informatik
der Technischen Universität München

A Polymorphic Sort System for Axiomatic Specification Languages

Dieter Nazareth

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Jürgen Eickel
Prüfer der Dissertation:	
1.	Univ.-Prof. Tobias Nipkow, Ph.D.
2.	Univ.-Prof. Dr. Manfred Broy

Die Dissertation wurde am 25. Januar 1995 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10. Mai 1995 angenommen.

Abstract

This thesis presents a novel polymorphic sort system for axiomatic specification languages. The sort system itself can be dynamically specified by a separate specification language. This approach enables a flexible sort system that can be dynamically adapted to different application areas. In particular, this sort system allows the modelling of different kinds of polymorphism. The thesis investigates the syntactic as well as the semantic treatment of axiomatic specification languages based on such a sort system.

The sort system is based on the concept of qualified sorts. It allows one to abstract from concrete sorts by using sort variables and to qualify these sort variables by sort predicates. The specification language is structured into two levels. The description of functions on the term level is based on classical first order predicate logic. It allows one to use polymorphic identifiers and to declare unsorted variables. The properties of the sort predicates are described by a separate specification language based on Horn clauses.

To check the well-sortedness of specifications we give a sort inference calculus. In addition, this calculus is used to compute the instantiation of polymorphic identifiers and the sorts of variables declared without sort information. There can be several derivations for one specification. A first attempt at defining a model takes all sort derivations into consideration. Afterwards, the semantic relations between different sort derivations of a specification are investigated. It is shown that syntactically more general derivations have less models. This result allows us to define an equivalent model concept based on some most general sort derivation of a specification.

Finally, we investigate the implementation of the sort inference calculus. We present an inference algorithm which splits into two phases. The first phase checks the well-formedness of the specification and computes the necessary sort predicate restrictions. In addition, a principal sort derivation is computed for the specification. The second phase uses resolution to prove the computed restrictions to be correct with respect to the sort specification. We show that the inference algorithm is correct and complete with respect to the sort inference calculus.

In general, the sort inference algorithm terminates only for well-sorted specifications because the problem is only semi-decidable for general sort specifications. If particular polymorphism concepts, as for example the sort class system of Haskell, are modelled the problem becomes decidable, and the given sort inference algorithm terminates for all specifications.

Zusammenfassung

Die vorliegende Arbeit stellt ein neuartiges polymorphes Sortensystem für axiomatische Spezifikationssprachen vor. Das Sortensystem selbst kann durch eine eigene Spezifikationssprache dynamisch spezifiziert werden. Dieser Ansatz ermöglicht ein flexibles Sortensystem das dynamisch an unterschiedliche Anwendungsgebiete angepaßt werden kann. Insbesondere erlaubt dieses Sortensystem unterschiedliche Polymorphiekonzepte zu modellieren. Die Arbeit untersucht die syntaktische und semantische Behandlung von axiomatischen Spezifikationssprachen basierend auf einem derartigen Sortenkonzept.

Das Sortensystem basiert auf dem Konzept der eingeschränkten Sorten. Es erlaubt mit Hilfe von Sortenvariablen von konkreten Sorten zu abstrahieren und die Sortenvariablen dann durch Sortenprädikate einzuschränken. Die verwendete Spezifikationssprache gliedert sich in zwei Ebenen. Die Beschreibung der Funktionen auf der Termebene basiert auf einer klassischen Prädikatenlogik 1. Stufe. Sie erlaubt die Verwendung von polymorphen Bezeichnern und die Deklaration von unsortierten Variablen. Die Eigenschaften der Sortenprädikate werden mit Hilfe einer eigenen Spezifikationssprache basierend auf Horn-Klauseln beschrieben.

Um die Wohlsortiertheit von Spezifikationen zu überprüfen, wird ein Sorteninferenzkalkül angegeben. Dieser Kalkül berechnet darüberhinaus die Instantiierung von polymorphen Bezeichnern sowie die Sortierung von unsortiert deklarierten Variablen. Dabei kann es zu einer Spezifikation mehrere Herleitungen geben. Ein erster Modellbegriff wird deshalb unter Betrachtung aller möglichen Sortenherleitungen definiert. Anschließend werden die semantischen Beziehungen zwischen unterschiedlichen Sortenherleitungen einer Spezifikation untersucht. Es wird gezeigt, daß syntaktisch allgemeinere Herleitungen weniger Modelle besitzen. Dieses Ergebnis erlaubt uns einen äquivalenten Modellbegriff zu definieren der nur auf einer allgemeinsten Sortenherleitung einer Spezifikation basiert.

Abschließend wird die Implementierbarkeit des Sorteninferenzkalküls untersucht. Es wird ein Inferenzalgorithmus vorgestellt, der sich in zwei Abschnitte gliedert. Im ersten Abschnitt wird die Wohlgeformtheit der Spezifikation untersucht und es werden die dazu notwendigen Sortenprädikatbeschränkungen berechnet. Darüberhinaus wird eine allgemeinste Sortenherleitung für die Spezifikation berechnet. Im zweiten Abschnitt werden dann mit Hilfe eines Resolutionsverfahrens die berechneten Beschränkungen gegenüber der Sortenspezifikation als korrekt bewiesen. Es wird gezeigt, daß der Inferenzalgorithmus korrekt und vollständig ist gegenüber dem Sorteninferenzkalkül.

Der Sorteninferenzalgorithmus terminiert im allgemeinen nur für wohlsortierte Spezifikationen, da das Problem für allgemeine Sortenspezifikationen lediglich semientscheidbar ist. Modelliert man jedoch spezielle Polymorphiekonzepte, wie beispielsweise das Sortenklassenkonzept von Haskell, dann wird das Problem entscheidbar und der angegebene Sorteninferenzalgorithmus terminiert für alle Spezifikationen.

Acknowledgements

In the first place, my thanks go to all those people that have helped me, either directly or indirectly with the development of this thesis.

Particular thanks to my advisors Tobias Nipkow and Manfred Broy. Manfred Broy enabled me to join his research group and suggested the subject of this thesis. Tobias Nipkow spent a lot of time in technical discussions. Without the freedom granted by both of them, this thesis could not have been finished.

Thanks also to all my colleagues for years of discussion. In particular I would like to thank Bernhard Schätz, Konrad Slind, Franz Regensburger, and Bernhard Rumpe for carefully reading drafts of this thesis.

On a personal note, I would like to thank my parents for their love and support. A final and special thank-you to my wife, Margit, for reminding me of the better things in life and for believing in me when I didn't.

Contents

Chapter 1

Introduction

1.1 A Sort System for a Specification Language

The world of formal languages is split into sorted and unsorted languages. Generally defined, a *sort* is a collection of objects with common properties. An unsorted language can be viewed as a sorted language where all objects belong to one large sort. This sort is often called the *universe of objects*. In a sorted language the universe is partitioned into smaller collections.

In this thesis we present a novel sort system for an axiomatic specification language. We start with a discussion of the advantages and drawbacks of sorts. We ask the question why one should bother with sorts in specifications at all. Simultaneously, we collect requirements for a desirable sort system.

Sorts are an important, universal *ordering principle*. They are used to classify objects of a universe according to their usage and behaviour. Classification is important because it forces the user to structure the objects used in a specification. This decreases the error rate and makes specifications better readable and more understandable. Thus, sorts can be seen as a simple form of documentation. Knowing the sort of an object even allows one to deduce some properties about its behaviour (see [Wad89]).

Furthermore, sorts are an important *abstraction principle*. They allow us to make propositions about a collection of objects, which are valid for each member of this collection. Thus, we can abstract in propositions from individual objects. At the same time the proposition is restricted to the objects of a sort. Therefore, in axiomatic specification languages sorts serve as a basis for restricting properties to a subset of the universe of objects.

Besides the classical ordering and abstraction principles there are further reasons for using sorts in specification languages. The most common one is the attempt to avoid

partial functions. By specifying the *functionality* of an object we fix its domain and codomain. This allows us to forbid terms where functions are applied to unintended arguments. Thus, sorts can be used to identify two classes of terms. A term with no unintended use of a function is called *well-sorted*. Otherwise, it is called *non-well-sorted*.

In implementations of specification languages, we are interested in the detection of non well-sorted terms. This can be achieved by analyzing specifications *syntactically*. The detection of an unintended use of a function with respect to the specified functionality is the most important advantage of a sort system. Simple specification mistakes can be detected by a syntactic analysis of the specification, before a time-consuming development towards an executable program is started.

To enable a syntactic analysis of specifications we need, however, not only a semantic notion of sort, but also a syntactic one. We need a formal language to describe the sorts of object identifiers and a formal calculus defining the well-sorted terms. The combination of both is called a *sort system*. A sort system that enables us to detect all errors with respect to the calculus is called a *strong sort system*.

Only a strong sort system really helps to decrease the error rate, because sort errors are always detected and reported to the user. Because axiomatic specifications are in general not executable, sort correctness must be checked by a static analysis. In the framework of axiomatic specification a strong sort system must therefore be a *static sort system*, i.e. a sort system where well-sortedness can be determined by static analysis. This condition strongly restricts the choice of suitable sort systems for specification languages. Furthermore, the sort system should be intuitive and not too complex in that the user should be able to predict whether a given specification is well-sorted or not.

Note that static sort systems are not suited to avoid partial functions in general. Some partial functions, e.g. the predecessor function on natural numbers, cannot be modelled adequately by totalizing them artificially. Moreover, partial functions are used to model non-terminating computations. Thus, a sorted specification language should also provide partial functions.

The sort system of a language can also be used to improve efficiency. In term rewriting, for example, the search space can be dramatically reduced by using many sorted matching. The same holds for automatic theorem proving if sorted unification is used.

Up to now we have enumerated only the advantages of sorts, but of course there are also disadvantages. The main one is that a sort system may hinder the flexibility and expressibility of the user because it imposes too many restrictions. In some cases a problem must be adapted to fit a specific sort system. This often leads to artificial, not problem oriented, specifications.

A further problem of many traditional sort systems is that they force the user to write a lot of redundant sort information. The sort of an identifier can often be uniquely

determined from the context. However, the sort system must also allow one to write explicit sort annotations. In the framework of axiomatic specification, explicitly written sort annotations are not only helpful documentation but may also influence the semantics of a specification.

Specification languages are commonly used for formal software development. The target of the development is a program for an executable language. The last step of the development is the formal transition from the specification to the executable program. The syntactic as well as the semantic transition is much easier if the sort systems of both languages go together. Thus, the sort system of a specification language should be oriented towards the sort system of the intended target language.

We summarize the requirements for a sort system for a specification language as follows. A sort system should

- be static, i.e., allow static analysis of specifications for their sorting properties,
- allow the omission of unnecessary sort information,
- restrict the user as little as possible,
- be intuitive and easy to understand,
- and be related with the sort system of the executable target language.

1.2 Polymorphism

Most traditional specification languages provide a so-called *many-sorted sort system*. Such sort systems allow one to name, by user-chosen identifiers, the sorts that partition a universe of objects. In first order languages, the functionality of a function can be described by assigning a sort identifier to each parameter position and to the argument position of the function identifier.

In higher order languages, in addition to the user-chosen sort identifiers, usually the identifiers "$\rightarrow$" and "$\times$" are provided. They denote the function space constructor, and the constructor for Cartesian products, respectively. The functionality of an object can be described with the help of a simple sort term language.

Terms are defined to be well-sorted if at each function application the sorts of the arguments coincide with the argument sorts specified for this function. Such sort systems are based on the idea that every object belongs to one and only one sort. They are said to be *monomorphic*. Monomorphic sort systems obviously do not fulfil the requirements

stated in Section 1.1 because they are too rigid. They are, e.g., not flexible enough to describe generic functions that are applicable on a range of sorts. A flexible sort system must allow an object to belong to more than one sort. Such sort systems are said to be *polymorphic*.

The specification language presented in this thesis is used to develop functional programs. Most functional languages provide some notion of polymorphism in order to weaken the restrictiveness of monomorphic sort systems. Because of the last requirement stated in Section 1.1, we therefore take a look at the most common notions of polymorphism used in these languages. We explain the different concepts informally by using typical examples. A more detailed and formal introduction can be found in [CW85].

1.2.1 Ad-hoc Polymorphism

One of the oldest means of achieving polymorphism is so-called *ad-hoc polymorphism* [Str67], better known as *overloading*. Overloading allows the use of the same identifier for completely different functions. A typical example is the infix identifier + which is not only used for addition of numbers, but sometimes also for the concatenation of lists. In a language that provides overloading the user can specify both functions, i.e.

$$+ : \texttt{Nat} \times \texttt{Nat} \to \texttt{Nat}$$
$$+ : \texttt{List} \times \texttt{List} \to \texttt{List}$$

and use them both in one term. From the sort context, normally by static program analysis, it is decided which + must be used, i.e. the overloading is resolved. Therefore, strictly speaking, this is not a kind of polymorphism, because each term again has exactly one sort. If not, the overloading cannot be resolved uniquely. Normally such terms are rejected as not well-sorted. Because overloaded identifiers are syntactically resolved, ad-hoc polymorphism does not have to be dealt with in the semantics. Furthermore, ad-hoc polymorphism can be simulated with the help of the class concept described in Section 1.2.3. Thus we do not occupy ourselves with ad-hoc polymorphism in this thesis.

1.2.2 Hindley/Milner Polymorphism

The identifiers "$\to$" and "$\times$", when used to describe the functionality of an object, denote *functions on the sort level*. They take a tuple of sorts as an argument and yield a sort as a result. Thus, a natural extension of many-sorted sort systems is to allow user-definable identifiers for arbitrary sort functions. A typical example is a function taking a sort as argument and yielding the sort of lists containing elements of the given sort as a result. This function can then, e.g., be denoted by an identifier `List`.

A further step towards a more flexible sort term language is to allow *sort variables* in sort terms. Variables are usually used to abstract from a concrete object. Therefore, by sort variables we can achieve abstraction from individual sorts in sort terms. The most well-known polymorphism concept based on sort variables is called, due to Hindley [Hin69] and Milner [Mil78], *Hindley/Milner polymorphism*.

Let us consider a function `length` that takes a list as argument and returns the length of the list as result. This function can be defined by the following two equations:

```
length emptylist = 0
length(append(x,l)) = succ(length l)
```

The definition of the function `length` function does not depend on the elements of the list (represented by `x`) and is therefore independent of the sort of the list elements. Monomorphic sort systems are not flexible enough to express this independence. They force the user to define a length function for each list sort, though the bodies of these functions do not differ. In the Hindley/Milner sort system, however, we can abstract from the element sort and assign the following sort scheme to the function `length`:

$$\texttt{length} : \Pi\alpha.\ \texttt{List}\ \alpha \rightarrow \texttt{Nat}$$

The identifier α is used as a sort variable and Π as universal quantification for this sort variable[1]. The quantifier indicates that α can be replaced by an arbitrary sort. `List` is a unary function on the sort level, called a *sort constructor*. Hindley/Milner polymorphism allows the application of the function `length` to a list of arbitrary sort. Let for example the variables `ln: List Nat`, `lln: List (List Nat)` and `lb: List Bool` be defined. Then `length` can be applied to all those variables, i.e. `length(ln)`, `length(lln)` and `length(lb)` are well-sorted terms. Thus `length` is called a *polymorphic function*.

Following Strachey [Str67] this abstraction mechanism is often described as *parametric polymorphism*. The sort variable α can be seen as a formal sort parameter of the function `length`. At application the formal parameter is then "invisibly" replaced by the actual sort of the list elements. We will adopt this view when defining the semantics of polymorphic functions. Note, however, that Hindley/Milner polymorphism is only the simplest form of parametric polymorphism. It allows the binding of sort variables only at the outermost level of a functionality. Thus, it is sometimes also called *shallow polymorphism* and the quantifier Π is usually omitted. We will have a brief look at higher order parametric polymorphism concepts in Section 1.2.6.

During his work on the formal treatment of polymorphism Milner observed that, in many cases, identifiers can be declared without sort information, since this information can be inferred from the application context of the identifier. He gave a *sort inference algorithm*

[1] Note that the binding Π is usually omitted in concrete functional languages.

calculating the omitted sort annotations, while checking the well-formedness of programs statically. The concept of *sort inference* is thus closely connected with Hindley/Milner polymorphism. Both were originally developed by Milner for the functional language ML [Har86]. Most modern functional languages have adopted these concepts. Thus a specification language used to formally develop functional programs should at least provide Hindley/Milner polymorphism.

1.2.3 Sort Classes

During the development of the functional programming language Haskell [HJW92], the developers realized that the then-existing kinds of polymorphism were not well-suited to model some kind of functions. They noticed that between ad-hoc polymorphism and parametric polymorphism there is, in some sense, a gap.

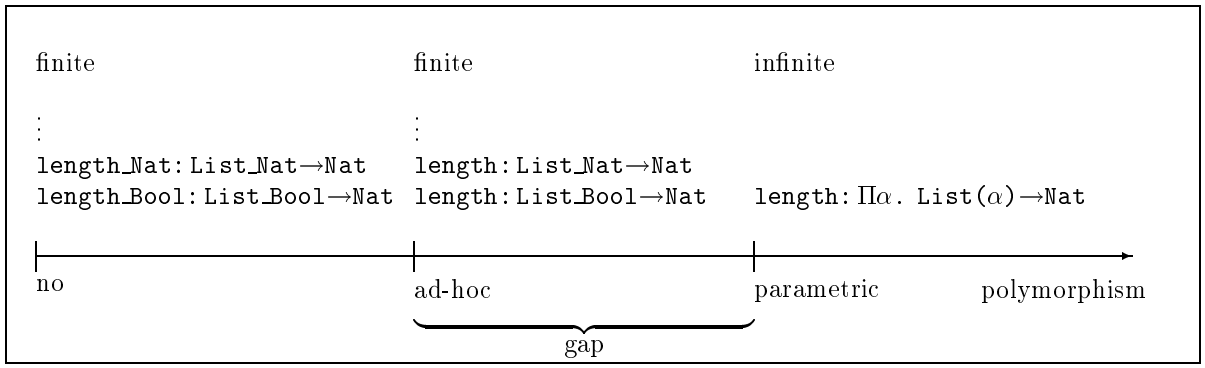

Figure 1.1: Polymorphism Gap

Figure 1.1 shows how the `length` function can be handled with different notions of polymorphism. The difference between no polymorphism and ad-hoc polymorphism is only a syntactic one, because overloading only allows one to use the same function identifier several times in one scope. But we are not allowed to overload sort identifiers: e.g., we have to use a different identifier for each kind of list. Thus, in both cases we have a separate function for each sort and we can only define a finite number of `length` functions. Parametric polymorphism, in contrast, allows us to define one `length` function which can be applied to lists of every sort, because the sort variable α can be instantiated by an arbitrary sort. This yields an infinite number of instantiations.

The problem is that there are a lot of functions that should be applicable on an infinite number of sorts, but not on all sorts, i.e. an infinite subset of the universe of sorts. A typical example is equality between objects. The equality function == should be applicable on nearly every sort, except, for example, on functional sorts, since the equality of functions is undecidable. In a program, the application of == to functions leads to

a run-time error which should be avoided by the sort system. However, this can be achieved neither by ad-hoc, nor parametric polymorphism.

In the functional programming language Haskell this problem was solved by using *sort classes*[2] [WB89]. This approach generalizes the equality type variables of ML (see [Pau92] for details) and closes the gap between ad-hoc and parametric polymorphism. A similar concept was independently developed by Kaes [Kae88]. The common idea is to structure sorts by typing them. In Haskell these types are called *classes*. The classes are used to restrict the bound sort variables in polymorphic functions by tagging them with classes. This leads to a typed abstraction mechanism. The equality function == can then be defined in the following way[3]:

$$==: \Pi\alpha\texttt{::EQ}.\ \alpha \times \alpha \rightarrow \texttt{Bool}$$

This means that == is only available on sorts which are in the class `EQ`. The following two definitions define some sorts belonging to class `EQ`:

```
Nat::EQ
List::(EQ)EQ
```

The first line states that `Nat` must be in class `EQ`. The second line means that `List` is a function taking a sort of class `EQ` and yielding a sort of class `EQ`. In other words, if sort `A` is in class `EQ`, the sort `List A` is also in class `EQ`. These declarations allow us to apply the function == on an infinite, but restricted number of sorts. In our case == can be applied to all elements of sort `Nat`, and to all nested lists of sort `Nat`, i.e. to elements of sort `Nat`, `List Nat`, `List (List Nat)`, etc., but it cannot be applied to elements of other sorts.

There are a lot of other examples where we want to use a function on an infinite, but restricted number of sorts. See [HJW92] or [BFG+93a] for more examples.

Another reason for the introduction of the class concept in Haskell was to make "ad-hoc polymorphism less ad-hoc", as Wadler and Blott stated in [WB89]. In other words, the sort class concept can be used to model ad-hoc polymorphism in a more structured way.

1.2.4 Subsorting

Sorts are used to classify the universe of objects. There is no obvious reason for classifying these objects uniquely. It is quite natural for an object to belong not only to one sort, but to many different sorts. Though most many-sorted approaches semantically allow overlapping sorts, a particular overlapping cannot be specified explicitly.

[2] In the functional community they are called *type* classes, of course.

[3] We will not use Haskell syntax in this section.

Subsort polymorphism, or, in short, *subsorting*, allows one to specify a restricted form of overlapping. It allows the definition of a partial order on sorts, called a *subsort relation*, which is interpreted as set inclusion. The theory of subsorting was developed by Goguen [Gog78] together with Meseguer [GM87]. The development was closely linked to that of OBJ [GW88], a logical programming language providing subsort polymorphism.

Subsorting allows us to apply functions to elements of a subsort of the function's parameter sort. Let us assume that the sort `Nat` is a subsort of the sort `Int`, written `Nat`$\subseteq$`Int`. If we have a function + with functionality `Int`$\times$`Int`$\rightarrow$`Int`, the application of + to a pair of natural numbers yields a well-sorted term, because the subsort relation states that each object of sort `Nat` is also an object of sort `Int`. The result is, however, an element of sort `Int` though adding two natural numbers always yields a natural number. To express this the subsort polymorphism allows to overload identifiers. Thus, we can additionally specify + to be of sort `Nat`$\times$`Nat`$\rightarrow$`Nat`. As a result, we get two addition functions which must, however, coincide on the common subsort `Nat`.

The ability to specify subsort relations supports the systematic treatment of exception propagation. Each sort may be enriched by error values, including them in a supersort. Furthermore, some partial functions can be avoided by restricting them to appropriate subsorts. We will pick up this subject again in Section 6.3.

Subsorting is also closely related to *inheritance*. In the example above one could also argue that the natural numbers inherit the function + from the integers. In Section 2.2 we will demonstrate on a larger example how to model inheritance by subsorting.

1.2.5 Qualified Sorts

In the last sections we gave an overview of the most important notions of polymorphism. We saw that Hindley/Milner polymorphism, subsorting and sort classes are very useful means of weakening the restrictiveness of traditional sort systems. They all help to describe the task of a system in a problem oriented way. Furthermore, these concepts allow the well-formedness of terms to be decided by static program analysis. It is therefore desirable to unify all these concepts by a general approach.

Mark P. Jones extended the notion of sort class to n-ary sort predicates [Jon92]. In his approach, bound sort variables can be restricted not only by sort classes, but by arbitrary n-ary sort predicates. This allows the definition of functions having the following sort scheme:

$$f : \Pi\alpha_1, \ldots, \alpha_n.\ P_1[\alpha_1, \ldots, \alpha_n], \ldots, P_m[\alpha_1, \ldots, \alpha_n] \Rightarrow \tau[\alpha_1, \ldots, \alpha_n]$$

τ is a sort expression, possibly containing the sort variables α_1,...,α_n. The sort variables can be restricted by predicates $P_1, \ldots, P_m$ of arbitrary arities. The "," between the

predicates is interpreted as a logical $\wedge$. Jones calls these restricted sorts *qualified sorts*. They form the basis for a general approach to polymorphism. The above sort scheme subsumes all sort expressions needed for the polymorphism concepts mentioned in the last sections.

Hindley/Milner polymorphism can be expressed trivially by qualified sorts, because we do not need any restricting predicate. For sort classes, a class is simply a unary sort predicate. In the functionality of ==, class `EQ` is replaced by a sort predicate `EQ`. As a result we get the following, in some sense equivalent, qualified sort:

$$==: \Pi\alpha.\ \texttt{EQ}(\alpha) \Rightarrow \alpha \times \alpha \rightarrow \texttt{Bool}$$

The handling of subsort polymorphism is more complicated. The concept described in Section 1.2.4 provided an *implicit* subsorting mechanism. Implicit, because besides the subsort relation we do not need any other mechanism to use the subsorting. That means we do not need any kind of *coerce* function, coercing elements of a sort to elements of a supersort. The coercion is done implicitly because the subsort relation is interpreted as set inclusion. This implicit subsorting, however, cannot be expressed with qualified sorts. We can only provide some kind of *explicit* subsorting.

Let us look at the addition function + from Section 1.2.4. In a first step we abstract from the concrete sort `Int` and get the following function:

$$+: \Pi\alpha.\ \alpha \times \alpha \rightarrow \alpha$$

However, this function is too general, because now + can be applied to elements of any sort. Thus, we must restrict the sort variable α by an appropriate predicate. We only want to apply + on subsorts of `Int`. This can be achieved by the following function:

$$+: \Pi\alpha.\ \alpha\subseteq\texttt{Int} \Rightarrow \alpha \times \alpha \rightarrow \alpha$$

The binary sort predicate $\subseteq$ is used to restrict the sort variable α to all subsorts of `Int`. This is, however, only the intended semantics behind the purely syntactic identifier $\subseteq$. The relation $\subseteq$ is not interpreted as set inclusion on sorts, but only as an "arbitrary" binary predicate. In Section 2.2 we will see how to specify certain properties for this binary predicate, e.g. reflexivity.

Qualified sorts offer only a restricted kind of subsorting, because every function that wants to use the subsort relation must explicitly use the restricting predicate in its functionality. Explicit subsorting achieved by qualified sorts, however, enables us to directly express the dependence of the result sort on an argument sort. In our example, adding two natural numbers yields again a natural number and not an integer, as in the implicit version. With implicit subsorting this can only be achieved by overloading the identifier +, or by using so-called *retract* functions. See [Mos91, Naz93a] for a closer look at the problems that arise, when using subsorting in an axiomatic specification language.

Besides sort classes and subsorting, there are other interesting applications of qualified sorts, e.g. extensible records. See [Jon92] for more examples. A sort system based on qualified sorts is provided by Gofer [Jon93a], a Haskell-like functional programming language.

1.2.6 Further Concepts

In this section we briefly introduce some instances of polymorphism not captured by qualified sorts.

An important generalization of Hindley/Milner polymorphism is included in the so-called *Girard-Reynolds polymorphic λ-calculus* [Gir72, Rey74]. In Hindley/Milner polymorphism the sort variables occurring in a sort term are implicitly or, as in the preceding chapters, explicitly bound at the outermost level of the sort term. The binding Π's cannot occur within a sort term.

In the polymorphic λ-calculus, in contrast, the binding Π's can be arbitrarily nested. In this calculus, however, sort abstraction and application are explicit. A term of the form $\Lambda\alpha.e$ is called *sort abstraction* while a term of the form $e\{\tau\}$ is called *sort application*.

Hindley/Milner polymorphism is limited compared to the polymorphism permitted in the polymorphic λ-calculus. The latter does not need a separate let-mechanism to define polymorphic functions since polymorphic abstraction is achieved by the usual λ-abstraction. Therefore, polymorphic values are handled as first-class citizens. They can be passed to and returned by functions as demonstrated by the following example:

$$\lambda f : (\Pi\alpha.\alpha \to \beta).(f\{\tau_1\}(e_1), f\{\tau_2\}(e_2))$$

The function takes a polymorphic function as the argument and applies it to the types of each of the components of a given pair. In languages providing only Hindley/Milner polymorphism, such as ML, this function cannot be defined.

Besides this obvious advantage the polymorphic λ-calculus has some serious drawbacks. Sort abstraction and application must be handled explicitly because sort inference is undecidable in general [KT90]. Furthermore, the semantics of this calculus is more involved than the one of the simply typed λ-calculus. There is some discussion whether there exists a set-theoretic model of the polymorphic λ-calculus [Pit87] or not [Rey84].

The polymorphic λ-calculus described above was called F_2 by Girard. Like Hindley/Milner polymorphism, it only allows abstraction from basic sorts. A further generalization leads to the system F_ω, which allows abstracting from arbitrary sort constructors (see also Section 7.2.1).

In the polymorphism concepts described so far the binding Π is used as universal quantifier for sort variables. For this reason some authors use $\forall$ instead of Π. By analogy, we can try to give meaning to existentially quantified sorts. By expression[4]

$$c : \exists\alpha.\tau[\alpha]$$

we specify that c has sort $\tau[\alpha]$ for some sort α. Existential sorts can be seen as *abstract sorts*. The sort variable α hides a concrete representation. In [CW85] it was demonstrated how existential sorts may be used to model data abstraction. Läufer [Läu94] integrated some restricted form of existential sorts into Haskell and showed that the combination with sort classes enables an interesting abstraction mechanism.

1.3 A Polymorphic Specification Language

In this thesis a novel sort system for a formal specification language is presented. In Section 1.1 we have worked out requirements for a desirable sort system. Conventional many-sorted sort systems only partially fulfil these requirements; they are easy to understand and sort correctness can be proved by static specification analysis. The main disadvantage of these sort systems, however, is that they are too rigid and, as a consequence, restrict the user. In Section 1.2 we have shown that polymorphism helps overcome this deficiency. Thus, polymorphism should also be integrated into the sort systems of formal specification languages.

Since we do not want to design our specification language for a particular functional target language, we are interested in supporting different kinds of polymorphism within one sort system. A very general polymorphism approach is the concept of qualified sorts. In Section 1.2.5 we have already sketched that this approach can be used to model different kinds of polymorphism. Therefore we will base our sort system on qualified sorts.

Qualified sorts, however, only form the basis of a flexible sort system. The key for flexibility lies in the sort predicates qualifying the sort variables occurring in the functionality of an identifier. For different notions of polymorphism, different sort predicates with different properties are needed. Thus, we need a language to describe these properties. To model, e.g., the sort classes of Haskell we need a language to describe class membership of basic sorts, class membership propagation to non-basic sorts, and subclass relations. However, if we want to model subsorting we need a language to describe subsort relations between basic sorts and subsort relation propagation to non-basic sorts. In addition, the subsort relation must be reflexive and transitive.

[4]The square brackets denote that the sort variable α occurs in τ.

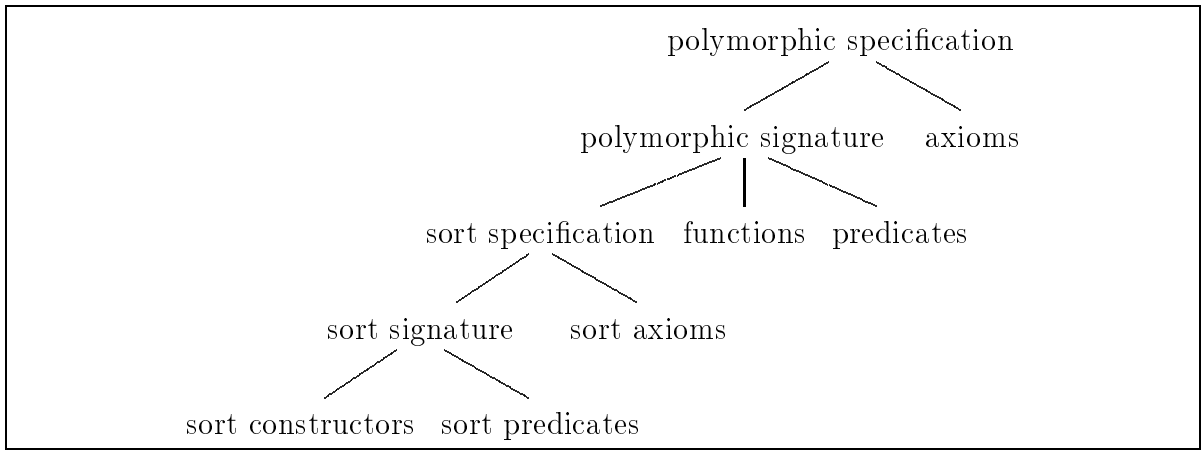

Figure 1.2: Structure of polymorphic specifications

In functional languages, specifically restricted languages are used to describe a particular sort predicate system. For example, Gofer provides qualified sorts, but subsorting cannot be modelled. As a consequence, a flexible sort system does, in addition to qualified sorts, require a flexible language to specify the properties of sort predicates. Therefore, the presented polymorphic specification language provides a separate specification language for specifying the properties of sort predicates.

Polymorphic specifications consist of two specifications on different levels. The structure is shown in Fig. 1.2. The sort specification is part of the polymorphic signature. The sort signature contains the sort constructors and the sort predicates of the polymorphic specification. In the sort axioms the properties of the sort predicates are described. In addition to the sort specification the polymorphic signature contains polymorphic functions and predicates. The functionality of both can be specified by qualified sorts. In the axioms, as usual, the properties of constants and functions are specified.

From a theoretical point of view the sort axioms may consist of arbitrary first order formulae. The well-sortedness of a polymorphic specification, however, depends on these axioms. Because we want to detect non-well-sorted specifications by static analysis, these axioms must be executable. Therefore, we use Horn-clause logic in sort axioms. It is at once expressive enough to specify arbitrary sort predicates and executable via resolution. In the axioms of a polymorphic specification we use, in contrast, classical full first-order logic.

At first glance, such a two-level specification language seems complicated and difficult to understand. Looking at it again, however, we see that the same technique is used on both specification levels. On the sort level as well as on the object level we work with signatures and axioms. In both cases the properties of functions and predicates are specified by axioms. Thus, we do not have to learn syntax and semantics of a specific

language specifying sort predicates.

Furthermore, polymorphic functions and predicates are applied like conventional monomorphic functions. As in functional languages, polymorphic identifiers are instantiated implicitly. In addition, variables may be declared without sort annotations. As a consequence, the user can write sort-free axioms as in unsorted specification languages. Nevertheless, sort errors with respect to the sort system are always detected. Our specification language therefore combines the convenience of unsorted languages with the advantages of static sort systems.

1.4 Outline of Thesis

This thesis presents a novel polymorphic sort system for axiomatic specification languages. The sort system itself can be dynamically specified by a separate specification language. This approach enables a flexible sort system that can be dynamically adapted to different application areas. In particular, this sort system allows the modelling of different notions of polymorphism. The thesis investigates the syntactic as well as the semantic treatment of axiomatic specification languages based on such a sort system. Since the presented approach is very general, our results can be immediately applied to syntactically as well as semantically more restricted languages. Thus, this thesis makes a contribution to the formal foundation of polymorphic axiomatic specifications. The detailed contributions are summarized in Section 7.1.

Chapter 2 gives an informal introduction to the two-level specification approach. We use the concrete specification language PolySpec to show the benefits of polymorphic specifications. In particular, we show how to model sort classes and inheritance within our general polymorphism approach.

In Chapter 3 we define a syntactic framework for two-level polymorphic specifications. The framework consists of a formal language for both sort specifications and object specifications. We give a calculus defining an entailment relation on sort predicates and an inference calculus to compute sort derivations for formulae. The inference calculus is an extension of the calculus given by Jones. In addition, our extension contains rules to handle explicit sort annotations as well as explicit sort predicate qualifications. With the help of both calculi, we define a notion of well-formedness for polymorphic specifications.

Chapter 4 investigates the semantics of polymorphic specifications. In Section 4.1 we define a concept of model for sort specifications and a semantic consequence relation for sort predicates, which is sound and complete with respect to the entailment relation. Furthermore, we give an interpretation for qualified sorts.

Section 4.2 starts with a definition of polymorphic algebras. Because of missing sort declarations and implicitly instantiated polymorphic identifiers, the meaning of formulae cannot be defined directly. Thus, we give an interpretation for sort derivations containing the lacking information. We present a concept of model for polymorphic specifications taking all possible sort derivations into consideration. Therefore, we neither need an intermediate language nor a particular sort inference algorithm to define the models of polymorphic specifications.

In Section 4.3 we present a further notion of model based on a principal sort derivation. We define a syntactic ordering "more general" on sort derivations. We show that if a polymorphic algebra satisfies a sort derivation of a formula, it satisfies all less general sort derivations. If we slightly change the sort inference calculus without changing the concept of well-formedness, we can show that each specification has a greatest sort derivation with respect to this ordering. This greatest sort derivation, called the principal sort derivation, is used to define a further notion of model. We show this to be slightly stronger than the one defined in Section 4.2.

In Chapter 5 we focus on the implementation of polymorphic specifications. We give a sort inference algorithm testing the well-formedness of polymorphic specifications. The algorithm is based on an extension of algorithm W [DM82] and a resolution prover. The prover is used to test the entailment relation on sort predicates. We show that the sort inference algorithm is sound and complete with respect to the sort inference calculus. While testing the well-formedness of specifications, the algorithm constructs a sort derivation for the specification. We prove this derivation to be a principal sort derivation for the specification. Finally, we show that the problem of well-sortedness is only semi-decidable in our sort system and discuss this fact.

In Chapter 6 the styles of polymorphism presented in Section 1.2 are discussed once again. We study the differences between using a particular kind of polymorphism in an executable language and in an axiomatic specification language.

Chapter 7 reviews the results and outlines several ideas for further work.

1.5 Related Work

The comparison with other work is twofold. On the one hand we compare our sort system with sort systems mainly used in functional languages. On the other hand we compare our polymorphic specification language to specification languages incorporating polymorphism.

1.5.1 Related Sort Systems

As we already noted, the sort system developed in this thesis is based firmly on qualified types presented by M.P. Jones in [Jon92]. Qualified types again are a generalization of Hindley/Milner polymorphism [Hin69, Mil78, DM82]. Neither [DM82] nor [Jon92], however, allow one to explicitly constrain terms by sort annotations. In both papers, a simple "mini" language based on the unsorted λ-calculus is used to investigate the sort system. In the framework of axiomatic specification, however, explicit sort annotations are essential since they may influence the semantics of a specification. For the same reason our sort system must allow one to explicitly constrain sort variables by qualifying sort predicates. Thus, in contrast to [Jon92], we additionally investigate the formal treatment of explicit sort annotations as well as explicit qualifying sort predicates.

In his thesis Jones abstracts from a concrete sort predicate entailment. He shows by a few examples how to instantiate his general framework with specific sort predicate entailments. We, in contrast, do not abstract from a concrete predicate entailment but provide a specification language to specify arbitrary sort predicate systems. Therefore, we do not have to instantiate our sort system for a concrete implementation. The desired sort predicate system can be dynamically specified by the user. This approach enables a maximum of flexibility. Moreover, the properties of the sort predicates are described within a standardized simple logical language. Thus, we do not have to design a specific language for each particular sort predicate system, and the user does not have to learn different languages. The implementation of sort inference remains always the same and does not have to be adapted to specific kinds of polymorphism.

In [Kae92] Kaes presents a concept of *constrained type* which is very similar to qualified types. Much of his work is concerned with establishing decidability of particular sort predicate systems. A set of rewrite rules is used to describe the entailment relation. He gives sufficient conditions for the rewrite rules to guarantee the decidability of the entailment relation. Most of his results can be transferred to our Horn clause-based approach.

Sort inference in the presence of constraints on sorts has been widely studied in the particular case of subsorting, e.g. in [Mit91] or [FM88, FM89, FM90]. In contrast to our approach these sort systems provide an implicit subsorting concept. From the viewpoint of sort inference there is, however, no difficulty to incorporate a rule in the sort system that allows implicit subsorting (see Section 6.3 in [Jon92]).

The sort system presented in [FM88] allows one to only specify subsort relations between basic sorts. Furthermore, suitable "structural" rules can be fixed to define the subsort relation between structured sorts in terms of their components. In particular, subsort relations between different sort constructors cannot be specified. Our approach,

in contrast, allows one to specify subsort relations between arbitrary structured sorts. This fact enables us, e.g., to treat values as one-element lists by specifying $\alpha \subseteq \texttt{List}(\alpha)$. As a consequence, all functions for lists can also be applied to arbitrary non-list values without converting them explicitly to one-element lists. This is not possible in [FM88].

Another interesting example of sort inference in the presence of constraints are sort isomorphisms. Thatte [Tha91] describes an extension of the Hindley/Milner sort system allowing one to declare isomorphisms between sorts. Isomorphism classes denote sorts, elements of which are interconvertible. In particular, this is a useful concept to handle different representations of the same abstract structure. As for subsorting, we support only explicit isomorphism. The user may specify arbitrary isomorphism relations between sorts. These relations are only regarded if a polymorphic function is explicitly constrained to some isomorphism class. Our approach, however, allows the handling of isomorphism between sorts as a special case of subsorting. We can specify two sorts to be isomorphic if and only if they are in a mutual subsort relation.

1.5.2 Related Specification Languages

In axiomatic specification languages polymorphism concepts have been ignored for a long time. In languages as CLEAR [BG80], ASL [SW83] or ACT-ONE [EM85, EM90] sort abstraction can only be achieved by using explicit parameterization mechanisms. They allow one to designate some part of a specification as formal parameters. The specification abstraction obtained is called *parameterized* and can be subsequently instantiated with actual specifications "fitting" the formal ones. Parametric polymorphic specifications are closely related to parameterized specifications. In [GN94] it is shown that in many cases parameterized specifications can be replaced by more elegant polymorphic specifications.

As part of the BMFT[5] project KORSO [HLR93, BW93], the specification language Spectrum was developed at the TUM[6]. One of the goals of the development of Spectrum was to investigate polymorphism in the framework of axiomatic specification. Version 0.3 of Spectrum [BFG+92] provided Hindley/Milner polymorphism in combination with implicit subsorting [Naz92]. The sort system was based on the work of [FM88] and thus enabled only a restricted form of subsorting (see Section 1.5.1).

In Version 1.0 of Spectrum [BFG+93a, BFG+93b] subsorting was replaced by a sort class system similar to the one used in the functional language Haskell [HJW92] and in the generic theorem prover Isabelle [Pau94]. The syntactic treatment of the sort class system

[5] German Ministry of Research and Technology

[6] Technische Universität München.

is based on [NP93]. An important aspect of this sort system is that only predicates of the form[7] $\mathtt{P}(\alpha)$ may appear in qualified sorts as well as in the predicate context of the sort inference calculus. Any context $\mathtt{P}(\tau)$, where τ is some structured sort, must be reduced to that simpler form. This reduction process can be thought of as a partial attempt to check for satisfiability of a sort predicate. It requires the sort signature to be coregular (see [NP93]). Otherwise, the reduction process is not uniquely defined. Our approach, in contrast, permits, as does the functional language Gofer [Jon93a], predicates over arbitrary structured sorts. Predicates over structured sorts need not be reduced. Hence, we do not need any restrictions for sort specifications. As a consequence, our concept of well-formedness is slightly less restrictive with respect to sort classes than the one defined for SPECTRUM. In the framework of axiomatic specification, however, there is no obvious reason for a more restricted well-formedness concept.

The semantics of SPECTRUM, described in [GR94], is defined indirectly via a second language. This extended language does not allow one to declare variables without sort information. Furthermore, it provides a mechanism to explicitly instantiate polymorphic identifiers. The extended specification is obtained by computing a principal sort derivation for the given specification and translating this derivation into a specification of the extended language. The whole translation process is, however, missing in [GR94]. Moreover, this method seems to be rather ad-hoc since the semantics is based on an algorithm computing a principal sort derivation; this is, however, a purely syntactic notion. In particular, any sort derivation could have been used to define the semantics of specifications. In this thesis, in contrast, we present a notion of model that is not based on any syntactic notion of a principal sort derivation. In addition, we show that the method used for SPECTRUM was adequate by proving the equivalence of both semantic approaches.

From the viewpoint of polymorphism, there is no fundamental difference between the object language used in SPECTRUM and the one used in our approach. SPECTRUM, however, in addition provides a language (inspired by ASL [SW83]), to structure specifications in the large. The semantic treatment of this language is described in [GR94].

In [Reg94] the classical higher order logic HOL [Gor85] was extended with a sort class concept similar to the one of Isabelle, giving HOLC. As in [GR94] the semantics of a specification is defined indirectly via a translation mechanism. Thus, the comments given for SPECTRUM hold for [Reg94], too. In contrast to [GR94], however, in [Reg94] a specification is not translated into a specification of an extended language but into a fully sorted specification of the same language. Moreover, the translation is formally defined.

[7] An exact comparison is difficult because different terminologies are used in [BFG+93a] as well as in [NP93].

As in Haskell, in HOLC the member (characteristic) functions of a class are distinguished from normal polymorphic functions[8]. The member functions are also reflected in the semantics of sort classes. In [Reg94] a class semantically does not only consist of a set of carriers but of a set of tuples containing a carrier together with a semantic value for each member function of the class. This is in contrast to SPECTRUM, Isabelle and our approach. In all these languages no polymorphic function is explicitly marked as a member function of a particular class. In SPECTRUM and in our approach a class semantically only consists of set of carriers. The more complex class semantics in [Reg94] is needed to support a notion of conservative extension of theories.

While the Hindley/Milner sort system has become the basis of the sort systems of many functional languages, in axiomatic specification languages attention was mainly devoted to subsorting concepts. Goguen [Gog78] was the first to present a framework for order-sorted algebras (OSA). The development of OSAs was closely linked to that of the OBJ system [GW88]. A quite similar approach to OSA was developed by Gogolla [Gog84] and Poigné [Poi84]. All these approaches have in common that the subsort order imposes a partial subset relation on the domains. This is in contrast to our approach because we only enable an explicit subsorting concept. Thus, our polymorphic algebras do not extend order-sorted algebras.

Qian [Qia91] extended order-sorted algebras with higher order functions and Hindley/Milner polymorphism. Subsorting on functional sorts is not modelled by set inclusion but by function restriction. This guarantees extensionality in contrast to the set inclusion approach presented in [Mit84] or [Car84]. The semantics of a polymorphic specification is defined as the semantics of its induced non-polymorphic specification. The induced specification is obtained by gradually replacing the sort variables occurring in the signature and in the axioms by all ground sort terms. This leads to an infinite set of function symbols in the signature and an infinite set of axioms. As a consequence of this purely syntactic treatment of polymorphism, the axioms must only be valid for the elements of domains generated by a sort term. We, in contrast, present a semantic approach to polymorphism. Polymorphic axioms must be valid for the elements of all domains of the universe. Because the universe of domains is not necessarily sort term generated, both approaches differ. In particular, enrichment or refinement of existing specifications cannot be treated as simple model set inclusion in the syntactic approach. This is only possible if we combine, as in our approach, a semantic treatment of polymorphism with a loose model concept. Besides this semantic difference the language in [Qia91] is restricted to equational logic. Furthermore, sort inference is not investigated. It is assumed that polymorphic functions are instantiated explicitly.

[8] Furthermore, the notion of characteristic axioms is defined in HOLC. Characteristic axioms specify the properties of characteristic functions.

Polymorphism has also been investigated in the framework of logic programming language [Smo89, Han89, HT90, Han91]. In [Han91] a two-level logic programming language is presented. As in our approach the language consists of a specification on the sort level and a specification on the object level. On both levels a Horn clause logic is used. In contrast to our approach the sort clause specification can only be used to specify the properties of the subsort relation $\subseteq$. The object language is restricted to Horn clause logic with first order functions. The declarative semantics of the two-level approach is based on [Poi86]. For a given polymorphic signature a dependent overloaded signature is derived. In contrast to [Qia91], however, the derived signature is obtained by interpreting the functionalities with respect to some sort algebra satisfying the sort axioms. The result is an infinitely overloaded signature in which object identifiers are sorted by semantic sorts. In the object algebra each (possibly overloaded) identifier is then interpreted by one "unsorted" function. A polymorphic function is therefore an object belonging to infinitely many different sorts. It can be viewed as being an element of the intersection of all sorts that are instances of the polymorphic sort. In our approach, in contrast, each polymorphic identifier is interpreted as one object belonging to exactly one polymorphic sort. See [Sok89] for a discussion of these two views.

Chapter 2

An Informal Introduction to Polymorphic Specifications

In this chapter we briefly sketch our polymorphic specification language. We show, using examples, how to model sort classes and inheritance. We do not go into syntactic or semantic details. These areas are treated in Chapters 3 and 4. Hence, we do not use the formal language defined in Chapter 3. This language is defined to investigate the formal treatment of polymorphic specifications and is thus very abstract. Instead, we use PolySpec, a concrete polymorphic specification language. The concrete syntax of PolySpec is given in Appendix B. The translation from the concrete language to the abstract one is straightforward. Since the main task of this chapter is to introduce our polymorphic sort system, the axiomatization of the specified functions is sometimes incomplete. In particular, we do not deal with definedness properties in this introductory chapter. For convenience we assume all quantified variables to range only over defined values.

PolySpec is a two-level specification language. A specification is divided into a specification on the sort level, called the *sort specification*, and a specification on the object level, called the *object specification*, where the latter depends on the former. The specification on the object level, as usual, defines the behaviour of the constants and functions. The sort level specification defines the properties of the sort predicates used in the qualified sorts of the object level.

On the object level we use a classical first order logic with higher order functions. Because the sort specification is used to define the well-sortedness of the object specification, sort specifications must be executable in some sense. To achieve this, we use a Horn-Clause logic on the sort level. This language is general enough to describe the properties of arbitrary predicates and is executable via resolution.

In Section 2.1 we introduce the language and show how to model sort classes. In Section 2.2 we demonstrate on a larger example that we can also model inheritance with our approach.

2.1 Modelling Sort Classes

In [GN94] we showed that in the framework of axiomatic specification sort classes can be used to avoid explicit parameterization. In many cases parameterized specifications can be replaced by more elegant polymorphic specifications. This was one of the main benefits of the specification language SPECTRUM. In this section we show that sort classes can be also modelled within our approach.

The original motivation for the development of the sort class concept was the uncomputability of the equality function. Sort classes can be used to syntactically restrict the application of an equality function to non-functional sorts. We will therefore start the description of our sort concept with Specification 2.1 which shows an equality function `eq` restricted in such a way[1].

`%` starts a one line comment. The specification is split into a sort level and an object level. Both can be mixed arbitrarily. As usual in the framework of axiomatic specification, both levels consist of a signature and an axioms part.

On the sort level, the signature consists of the sort functions, called *sort constructors*, and the sort predicates. The number assigned to each identifier expresses its arity. In our example we define a null-ary sort constructor `Bool`, i.e. a sort constant or *basic sort* and a unary sort predicate `EQ`. We do not fix properties for the predicate `EQ` in this specification. Thus, the axioms part of the sort level is missing.

In the signature part of the object level we declare the polymorphic function `eq`. The sort predicate `EQ` is used to restrict the sort variable α. Thus, the application of the function `eq` is restricted to those sorts for which the sort predicate `EQ` holds. In class terminology `eq` is called a *member function* of class `EQ`.

The signature of the object level is the only place where we can declare a polymorphic identifier. If we allowed to declare a polymorphic identifier by λ-abstraction, we would get a higher order function with respect to polymorphism, i.e. we would get a function that takes a polymorphic function as its argument. This is not allowed in our approach, because we restrict our language to shallow polymorphism. Neither do we allow declaring polymorphic identifiers by the quantifiers $\forall$ and $\exists$, since we do not want to make propositions about a set of polymorphic objects. We only want to describe the behaviour of

[1] Do not confuse `eq` with $=$. The identifier `eq` denotes a function while the built-in identifier $=$ denotes the identity predicate.

```
% specification on the sort level
% sort constructor
cons      Bool:0;

% sort predicate
sortpred  EQ:1;

% specification on the object level
% function
fun       eq: Πα. EQ(α) ⇒ α × α → Bool;

% first−order axioms
axioms EQ(α) ⇒ a:α, b:α, c:α in
     eq(a,a) = true;
     eq(a,b) = true ∧ eq(b,c) = true ⇒ eq(a,c) = true;
     eq(a,b) = eq(b,a);
endaxioms
```

Specification 2.1: Specification of function `eq`

a particular polymorphic object, like `eq`. Thus, the signature of a specification replaces the let-construct used in functional languages to define polymorphic functions.

In the axioms part of the object level we describe the laws of the function `eq` by using traditional first order logic. The axioms define `eq` to be an equivalence relation. The variables `a`, `b` and `c` are universally quantified. These variables are declared to be of sort α. But note that by doing this we do not declare polymorphic variables. The sort variable α is not bound by a Π at the outermost level of the sort expression, which is not allowed for local identifiers. Thus, the variables have the same sort at each occurrence in the axioms, and therefore they are not polymorphic. All sort variables are automatically universally quantified at the outermost level of the object specification.

Like in the signature, we restrict the sort variable α by the sort predicate `EQ`, because `eq` can only be applied to variables whose sorts fulfil the predicate `EQ`. By using sort variables in the axioms part we can demand the laws not only for one sort. The laws must be valid for all instantiations of the sort variables which fulfil the restricting predicate.

Our sort system allows one to declare variables without sort information. Thus, we could omit the sort variable α as well as the restriction `EQ(`α`)`, since they can be inferred from the applications in the axioms. Note that usually this sort information is not uniquely determined. The sort inference algorithm always selects the "most general"

sort information. We will discuss this in detail when defining the formal semantics of polymorphic specifications.

Now we want to define the sort predicate EQ to hold for some sorts. Speaking in terms of the functional language Haskell we want to define *instances* of the class EQ, namely for natural numbers and for lists. Specification 2.2 shows the respective axioms.

```
cons Nat:0;
     List:1;

fun  0: Nat;
     succ: Nat → Nat;
     []: Πα. List(α);
     append: Πα. α × List(α) → List(α);

axioms ... in
     % axioms defining Nat and List(α)
endaxioms

% axioms on the sort level
% instantiation
sortaxioms α in
     EQ(Nat);
     EQ(α) ⇒ EQ(List(α));
endaxioms

% specific properties of instances
axioms n:Nat, m:Nat in
     eq(0,succ n) = false;
     eq(succ n,succ m) = eq(n,m);
endaxioms

axioms EQ(α) ⇒ x:α, y:α, k:List(α), l:List(α) in
     eq([],append(x,l)) = false;
     eq(append(x,k),append(y,l))=true ⇔ eq(x,y)=true ∧ eq(k,l)=true;
endaxioms
```

Specification 2.2: Instances for EQ

In the first part of the specification we define a sort `Nat` together with its constructors `0` and `succ`. Furthermore, we define a unary sort constructor `List` together with the polymorphic constructors [] and `append`.

In the axioms part of the sort level we define the desired properties of our predicate `EQ`. Speaking in terms of classes, the first axiom states that sort `Nat` belongs to class `EQ`. The second axiom states that if a sort α belongs to class `EQ` then sort `List` α belongs to this class, too. Thus, we have defined the same behaviour as in Section 1.2.3. The equality `eq` can be applied to all elements of sort `Nat` and to all nested lists with elements of sort `Nat`.

Up to now we have not fixed any specific properties for the instances of `eq`. Both instances must only fulfil the general equivalence axioms given in Specification 2.1. Only by the axioms on the object level do we require the usual properties for both instances. Both axioms blocks are only well-sorted because of the two given sort axioms. In the first axioms block we apply the equality function to terms of sort `Nat`. This application is well-sorted because `EQ(Nat)` is entailed by the sort specification. In the second axioms block we apply the same function to terms of sort `List`(α). This application is well-sorted, too, because `EQ(List(`α`))` is entailed by the sort specification under the given assumption `EQ(`α`)`. Note that we have to show that the instances of `eq` fulfil the general requirements given in Specification 2.1. Otherwise the specification becomes inconsistent.

The sort class concept also supports a mechanism for defining hierarchies of classes using a notion of *subclass*. Specification 2.3 shows that this mechanism is also available in our approach. In the specification we define a new unary sort predicate `ORD`. We specify that if sort predicate `ORD` holds for a particular sort then sort predicate `EQ` also holds for this sort. This is exactly the semantics of "class `ORD` is a subclass of class `EQ`". Function `le` is restricted by the new sort predicate and the axioms require `le` to be a partial order. We instantiate `ORD` with sort `Nat` and uniquely fix the ordering by the given axiom.

2.2 Modelling Inheritance

In the last example we showed only a simple application of our polymorphic specification language. The example can, in a similar way, already be written in the specification language SPECTRUM V1.0 [BFG+93a], which provides sort classes. The difference, however, is that SPECTRUM V1.0 only provides sort classes, whereas in our approach sort classes are only one of many possible applications. In this section we will give an example of a completely different application of our universal polymorphism approach. We will show that our language is also well-suited to model inheritance in a smart way.

```
sortpred  ORD:1;

sortaxioms α in
     ORD(α) ⇒ EQ(α);
endaxioms

fun  le: Πα. ORD(α) ⇒ α × α → Bool;

% general properties of le
axioms a, b, c in
     le(a,a) = true;
     le(a,b) = true ∧ le(b,c) = true ⇒ le(a,c) = true;
     le(a,b) = true ∧ le(b,a) = true ⇒ eq(a,b);
endaxioms

% instantiation
sortaxioms in
     ORD(Nat);
endaxioms

% specific properties of the instance
axioms n in
     le(n,succ n) = true;
endaxioms
```

Specification 2.3: Subclass `ORD`

The example deals with the specification of a graphics system, one of the first application areas of inheritance techniques. A similar specification can be found in [Bre91] and [Naz93a]. In [Bre91] an implicit subsorting concept with overloading is used to model inheritance. [Naz93a] uses a combination of implicit and explicit subsorting provided by the specification language SPECTRUM V0.3 [BFG+92].

The specification of the graphics system is based on a specification providing a sort `Coord` for coordinates together with a constructor `mk_coord` as well as a few operations. In Specification 2.4 we only show the signature of the used operations. Function `len_coord` yields the length of the vector described by the coordinate. Functions `get_bl_coord` and `get_tr_coord` yield the bottom-left and top-right coordinates of the rectangle described by the two given coordinates. The meaning of the other functions is straightforward.

```
cons Coord:0;

fun  mk_coord: Real × Real → Coord;
     add_coord: Coord × Coord → Coord;
     sub_coord: Coord × Coord → Coord;
     len_coord: Coord → Real;
     get_bl_coord: Coord × Coord → Coord;
     get_tr_coord: Coord × Coord → Coord;
```

Specification 2.4: Signature for coordinate functions

We use a sort `Real` for the real numbers together with the usual operations.

A graphics system generally provides a set of graphic objects together with some operations to manipulate them. In our specification we consider only the moving of objects. This operation should be available on all kinds of objects. Therefore, we use a polymorphic function and restrict the operation to graphic objects. Graphic objects are typically hierarchically ordered by a so-called *is-a* relation. For example, a point is a line and a line is a rectangle and a rectangle is a graphic object. This is-a relation is used to specialize objects. A graphic object in our example is the most vague term with the least properties. The terms rectangle, line, and point are more and more concrete and are only special cases of the others.

The is-a relation is strongly connected with the concept of inheritance. A specialized object usually inherits all properties from a more general object. For example points, lines and rectangles inherit all general properties of a graphic object.

In Specification 2.5 the is-a relation is modelled by a binary sort predicate `IS_A`. In the axioms part of the sort specification we define the laws of the new sort predicate, namely reflexivity and transitivity. Remember that defining sort predicates together with rules does not influence the domains. In particular, `IS_A` is a freely chosen identifier and does not impose a subsort order on domains. Thus, we provide a function `coerce` converting objects along the `IS_A` relation. The laws of the function reflect the reflexivity and transitivity property of the `IS_A` relation.

The first axioms block looks a bit strange. It seems like the function `coerce` yields different values for the same argument. But the first look deceives: there are always different instances of the function applied. The instances can be uniquely deduced from the context, because the built-in predicate = can only compare values of the same sort. In the second axioms block we specify the injectivity of the coerce function. We cannot add this axiom to the first axioms block. The axiom would not be well-sorted because variables `a` and `b` must be of the same sort. Otherwise, they cannnot be compared by

```
sortpred  IS_A:2;

sortaxioms α,β,γ in
     IS_A(α,α);
     IS_A(α,β), IS_A(β,γ) ⇒ IS_A(α,γ);
endaxioms

fun  coerce: Πα,β.IS_A(α,β) ⇒ α → β;

axioms IS_A(α,β), IS_A(β,γ) ⇒ a:α, b:β, c:γ in
     coerce a = a;
     coerce a = b ∧ coerce b = c ⇒ coerce a = c;
endaxioms

axioms a, b in
     coerce a = coerce b ⇒ a = b;
endaxioms
```

Specification 2.5: Specification of sort predicate `IS_A`

the built-in predicate =. In the second block we omitted explicit sort information. This is allowed in our language because the lacking information can be inferred from the sort context. The fully sorted axioms block looks as follows:

```
axioms IS_A(α,β) ⇒ a:α, b:α in
     (coerce a):β = (coerce b):β ⇒ a = b;
endaxioms
```

This example demonstrates that omitting explicit sort information increases the readability of specifications.

In Specification 2.6 we define general properties of graphic objects. On the sort level we define a new basic sort `Graphic_Object`. In our specification each object is surrounded by an invisible rectangular frame. The functions `bl` and `tr` yield the respective bottom-left and top-right coordinates of this frame. The move function is specified by describing its effect on the surrounding frame. All functions are polymorphic and restricted by the predicate `IS_A(`α`,Graphic_Object)`. All functions can therefore be applied only on sorts which are graphic objects. We do not describe each single axiom. Again we only look at the polymorphism aspect.

The properties described in Specification 2.6 are very general and must be valid for all graphic objects. Therefore, we use the sort variable α for the identifier `g` and restrict

```
cons Graphic_Object:0;

fun  bl: Πα. IS_A(α,Graphic_Object) ⇒ α → Coord;
     tr: Πα. IS_A(α,Graphic_Object) ⇒ α → Coord;
     move: Πα. IS_A(α,Graphic_Object) ⇒ α × Coord → α;

axioms IS_A(α,Graphic_Object) ⇒ g:α, c:Coord, d:Coord in
     (move(g,c) = move(g,d)) ⇔ (c = d);
     move(g,bl g) = g;
     move(move(g,c),d) = move(g,d);

     bl(move(g,c)) = c;
     tr(move(g,c)) = add_coord(bl(move(g,c)),sub_coord(tr g,bl g));
endaxioms

axioms IS_A(α,β), IS_A(β,Graphic_Object) ⇒ g:α, g':β in
     coerce(g) = g' ⇒ bl g = bl g' ∧
                      tr g = tr g';
endaxioms
```

Specification 2.6: General properties of graphic objects

α by the predicate `IS_A(`α`,Graphic_Object)`. Note that giving the identifier `g` the sort `Graphic_Object` would yield a completely different semantics. In this case all axioms would only be valid for the elements of sort `Graphic_Object` and not e.g. for the elements of sort `Line`, defined in the next specification. Thus, the concrete graphic objects would not inherit the stated general laws. This is a consequence of our explicit approach.

In the last axioms block we specify that coercing an object along the `IS_A` relation does not influence the result of applying functions `tr` and `bl`. This property usually holds in implicit subsort systems, too. Overloaded functions must coincide on a common subsort. Note that we do not demand this property to hold for the function `move`. This function may behave differently on different sorts. It must only fulfil the general requirements specified in the first axioms block. Of course, we can specify the coincidence property for the function `move`, too. Thus, our use of axioms allows control over the degree of inheritance.

Now we add the concrete graphic objects `Line`, `Circle` and `Point` to our specification and claim specific properties for them. Fig. 2.7 shows the desired `IS_A` relation between graphic objects.

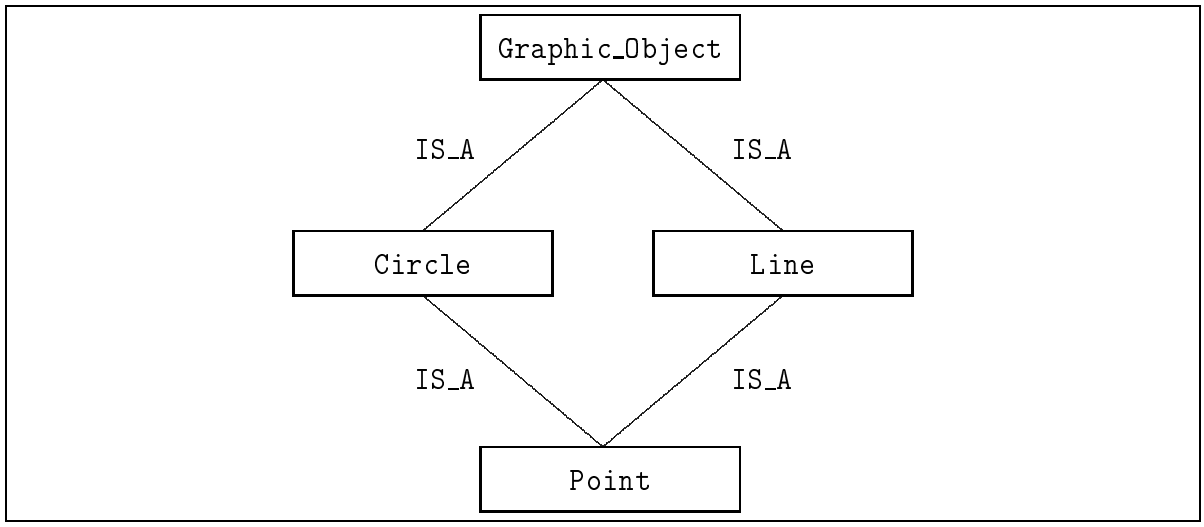

Figure 2.7: IS_A relation between graphic objects

In Specification 2.8 we define the sort Line together with the desired IS_A relation. We define an object constructor mk_line and an additional function length computing the length of objects that are lines. In the axioms part we specialize the polymorphic functions bl, tr and move. These specializations must, however, observe the general properties demanded in Specification 2.6. Otherwise the specification becomes inconsistent, i.e. it has no model. In the last axioms block we specify that all sorts that are lines inherit the properties of the length function given for lines.

Specification 2.9 adds a sort Circle to the graphics system. As in Specification 2.8, we specialize the general functions bl, tr and move.

Finally, we add points in Specification 2.10. The sort Point is specified to be a Line as well as a Circle. Then, by transitivity of the IS_A relation, IS_A(Point,Graphic_Object) holds, too. The sort Point inherits functions tr and bl both from sort Line and from sort Circle. Thus, this is an example for *multiple inheritance*. The sort Point inherits laws for the functions tr and bl from multiple sources. Therefore, we neither need to define axioms for function tr nor for bl. We must only fix how to coerce points to lines and to circles. Note, however, that the multiply inherited functions tr, and bl, respectively, must coincide on points. Otherwise the specification becomes inconsistent. In addition, function length is inherited from sort Lines. Points, however, do not inherit function move from lines or circles. Thus, we must also specialize this function for points.

```
cons Line;

sortaxioms in
     IS_A(Line, Graphic_Object);
endaxioms

fun  mk_line: Coord × Coord → Line;
     length: Πα.IS_A(α,Line) ⇒ α → Real;

axioms ⇒ line:Line, c1:Coord, c2:Coord, c3:Coord, c4:Coord in
     ∃c1. ∃c2. line = mk_line(c1,c2);
     mk_line(c1,c2) = mk_line(c3,c4) ⇔ (c1,c2) = (c3,c4) ∨
                                       (c1,c2) = (c4,c3);

     bl(mk_line(c1,c2)) = get_bl_coord(c1,c2);
     tr(mk_line(c1,c2)) = get_tr_coord(c1,c2);

     move(mk_line(c1,c2),c3) = mk_line(
          add_coord(c1,sub_coord(c3,bl(mk_line(c1,c2)))),
          add_coord(c2,sub_coord(c3,bl(mk_line(c1,c2)))));

     length(mk_line(c1,c2)) = len_coord(sub_coord(c1,c2));
endaxioms

axioms IS_A(α,β), IS_A(β,Line) ⇒ g:α, g':β in
     coerce g = g' ⇒ length g = length g';
endaxioms
```

Specification 2.8: Properties of lines

```
cons Circle;
sortaxioms in
     IS_A(Circle, Graphic_Object);
endaxioms

fun  mk_circle: Coord × Real → Circle;

axioms circle:Circle, c1:Coord, c2:Coord, r1:Real, r2:Real in
     ∃c.∃r. circle = mk_circle(c,r);
     mk_circle(c1,r1) = mk_circle(c2,r2) ⇔ (c1,r1) = (c2,r2);

     bl(mk_circle(c1,r1)) = sub_coord(c1,mk_coord(r1,r1));
     tr(mk_circle(c1,r1)) = add_coord(c1,mk_coord(r1,r1));

     move(mk_circle(c1,r1),c2) = mk_circle(
          add_coord(c2,sub_coord(c1,bl(mk_circle(c1,r1)))),r1);
endaxioms
```

Specification 2.9: Properties of circles

```
cons Point;
sortaxioms in
     IS_A(Point,Line);
     IS_A(Point,Circle);
endaxioms

fun  mk_point: Coord → Point;

axioms point:Point, c1:Coord, c2:Coord in
     ∃c. point = mk_point c;
     mk_point c1 = mk_point c2 ⇔ c1 = c2;

     coerce(mk_point c1) = mk_line(c1,c1);
     coerce(mk_point c1) = mk_circle(c1,0);

     move(mk_point c1,c2) = mk_point c2;
endaxioms
```

Specification 2.10: Properties of points

Chapter 3

A Core Language for Polymorphic Specifications

For the introductory examples in Chapter 2 we used the specification language PolySpec. To investigate the syntactic and semantic treatment of polymorphism in a specification language we use a more abstract formal language, which is restricted to the essential syntactic constructs. It can be therefore used as an abstract core language. More concrete and extended languages can be formally defined by translating them to the presented core language.

In Section 3.1 we present the language of sort specifications, which consist of a sort signature and a set of sort axioms. We define the notion of polymorphic sort terms and give the inference rules defining the entailment relation on sort predicates. We prove this relation to be monotonic, closed with respect to transitivity, and closed with respect to sort variable substitution.

Section 3.2 introduces the polymorphic specification language. Polymorphic specifications consist of a polymorphic signature and a set of axioms. We define the context-free syntax of the axioms language and give a sort inference calculus. This calculus is used to define a concept of well-sortedness for polymorphic specifications.

3.1 The Language of Sort Specifications

As usual in the framework of axiomatic specification, a sort specification consists of a signature and a set of axioms. The signature contains the sort functions, called sort constructors, and the sort predicates. In our language constructors and predicates are unsorted. Using a sorted language would result in a third abstraction level. This approach can be found, e.g., in the specification language SPECTRUM, and in the theorem

prover Isabelle [Pau93]. Both systems provide an order "meta-sorted" version of Hindley/Milner polymorphism (see [GR94] or [NP93] for details). But neither system allows arbitrary sort predicates or axioms on the sort level. They are only suited to express the classical Haskell sort class system. If allowing n-ary sort predicates, however, it is easier to treat meta-sorts as unary sort predicates.

Furthermore, we do not provide higher order sort constructors or predicates. Both can only be applied to a tuple of sorts and not to sort constructors. The functional language Gofer allows higher order sort predicates, i.e. predicates can be applied to sort constructors. In Gofer these predicates are called *constructor classes*. Details about the theory and the application of constructor classes may be found in [Jon93a]. However, Gofer does not allow to specify arbitrary axioms on the sort level. Furthermore, higher order sort predicates can be simulated by using n-ary first order predicates. In Section 7.2 we discuss the integration of higher order sort variables into our approach.

Formally, a sort signature is defined as follows:

Definition 3.1.1 *The set of sort signatures SSIG*

$(SC, SP) \in$ SSIG iff

- $SC = \{SC_n\}_{n \in \mathbb{N}}$ is an arity indexed set of identifiers, called *sort constructors*
- $SP = \{SP_n\}_{n \in \mathbb{N} \setminus 0}$ is an arity indexed set of identifiers, called *sort predicates*
- the sort constructors as well as the sort predicates must be unique, i.e. we do not allow overloaded sort constructors or sort predicates
- $\rightarrow \in SC_2$

□

A sort constructor sc of arity 0, i.e. $sc \in SC_0$ is called a *basic sort*. The arrow $\rightarrow$ is the constructor for the continuous function space. It is the only predefined sort constructor. As it is an element of every sort signature we will omit it when declaring a sort signature.

Example 3.1.1 *Sort signature*

The tuple

$$(\{\{\texttt{Bool},\texttt{Nat},\texttt{Int}\}_0,\{\texttt{List},\texttt{Set}\}_1,\{\times\}_2\},\{\{\texttt{EQ},\texttt{PO}\}_1,\{\texttt{IS\_A}\}_2\})$$

is an example for a sort signature. □

Next we turn to the definition of the sort axioms language given by a sort signature and a set of sort variables. Because we do not allow higher order sort constructors and sort

predicates we only need first order sort variables for building sort terms. Let Φ be a set of sort variables different from any possible sort constructor and sort predicate identifier. In the following we will use Greek lower case letters for sort variables.

Definition 3.1.2 *The set of monomorphic sort terms* $T_\Omega(\chi)$

Let $\Omega = (\{SC_n\}_{n\in\mathbb{N}}, \{SP_n\}_{n\in\mathbb{N}\setminus 0})$ be a sort signature. Let $\chi \subseteq \Phi$ be a set of sort variables. $\mathrm{T}_\Omega(\chi)$ is the set $<sortterm>$ defined by the following EBNF grammar[1]:

$<sortterm>$ ::=	χ	*sort variable*
|	SC_0	*basic sort*
|	$SC_n(\{<sortterm>//,\}^n)$	*sort constructor application, n≥1*

We use the notation $\mathrm{T}_\Omega(\chi)$ to declare the sort variables χ which are allowed to occur freely in the sort terms. If χ is the empty set, i.e. $\tau \in \mathrm{T}_\Omega(\emptyset)$, then τ is called a *ground sort term*. The set of sort variables occurring in a monomorphic sort term τ is denoted by $FSV(\tau)$. □

Note that, in contrast to ML or Cambridge LCF, we use prefix notation for constructor application. Moreover, we always use parenthesis when applying a sort constructor to some term. For better readability we will use the predefined sort constructor $\rightarrow$ in right associative infix notation. We will use parenthesis to achieve other bindings.

Example 3.1.2 *Monomorphic sort terms*

Let Ω be the sort signature defined in Ex. 3.1.1. The following expressions are elements of $\mathrm{T}_\Omega(\{\alpha, \beta\})$:

$$\times(\texttt{Nat},\texttt{Nat}) \rightarrow \texttt{Bool}$$
$$\texttt{List}(\alpha) \rightarrow \alpha$$
$$(\alpha \rightarrow \beta) \rightarrow \texttt{List}(\alpha) \rightarrow \texttt{List}(\beta)$$

□

Definition 3.1.3 *Sort variable substitution*

A *sort variable substitution* is a total function $\sigma : \Phi \rightarrow \mathrm{T}_\Omega(\Phi)$ mapping sort variables to monomorphic sort terms. The identity substitution is denoted by ε, i.e. $\forall\alpha \in \Phi.\ \varepsilon(\alpha) = \alpha$. $\sigma_1\sigma_2$ denotes the composition of two substitutions σ_1 and σ_2, i.e. $\forall\alpha \in \Phi.\ (\sigma_1\sigma_2)(\alpha) = \sigma_1(\sigma_2(\alpha))$. We write $\sigma.[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ for a substitution mapping each α_i to τ_i, $1\leq i\leq n$, and any other sort variable α to $\sigma(\alpha)$. $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ is an abbreviation for $\varepsilon.[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$. Furthermore, we define $DOM(\sigma) := \{\alpha | \sigma(\alpha) \neq \alpha\}$, $COD(\sigma) := \bigcup_{\alpha\in DOM(\sigma)} FSV(\sigma(\alpha))$ and $FSV(\sigma) := DOM(\sigma) \cup COD(\sigma)$. □

The concept of sort variable substitution is used throughout the whole thesis. We will extend sort variable substitutions to several other structures containing sort variables.

[1] See Appendix B.1 for a definition of the notation.

Definition 3.1.4 *Sort term substitution*

A *sort term substitution* is an Ω homomorphism $\sigma^* : \mathrm{T}_\Omega(\Phi) \to \mathrm{T}_\Omega(\Phi)$, extending a sort variable substitution σ to monomorphic sort terms in the following way:

$$\begin{array}{ll} \sigma^*(c) := c & c \in SC_0 \\ \sigma^*(\alpha) := \sigma(\alpha) & \alpha \in \Phi \\ \sigma^*(sc(\tau_1, \ldots, \tau_n)) := sc(\sigma^*(\tau_1), \ \ldots \ , \sigma^*(\tau_n)) & n \geq 1 \end{array}$$

□

The properties of the sort constructors of a specification are fixed by sort clauses supplied by the user. The definition of clauses is, as usual, based on sort terms and atomic sort formulae. We extend sort variable substitutions to these syntactic structures.

Definition 3.1.5 *The set of atomic sort formulae* $AF_\Omega(\chi)$

Let $\Omega = (\{SC_n\}_{n \in \mathbb{N}}, \{SP_n\}_{n \in \mathbb{N} \setminus 0})$ be a sort signature, and let $\chi \subseteq \Phi$ be a set of sort variables. $p(\tau_1, \ldots, \tau_n) \in \mathrm{AF}_\Omega(\chi)$ iff

- $p \in SP_n$ is an n-ary sort predicate
- $\tau_1, \ldots, \tau_n \in \mathrm{T}_\Omega(\chi)$ are monomorphic sort terms, $n \geq 1$

If $A \in \mathrm{AF}_\Omega(\emptyset)$ then A is called a *ground atomic sort formula*. $\mathrm{AF}^{ng}_\Omega(\chi) := \mathrm{AF}_\Omega(\chi) \setminus \mathrm{AF}_\Omega(\emptyset)$ is called the set of *non-ground atomic sort formulae*. These sort formulae must at least contain one sort variable. The set of sort variables occurring in an atomic sort formula A is denoted by $FSV(A)$. □

Definition 3.1.6 *Sort formula substitution*

A *sort formula substitution* is an Ω homomorphism $\sigma^* : \mathrm{AF}_\Omega(\Phi) \to \mathrm{AF}_\Omega(\Phi)$ extending a sort variable substitution σ to atomic sort formulae in the following way:

$$\sigma^*(p(\tau_1, \ldots, \tau_n)) := p(\sigma^*(\tau_1), \ldots, \sigma^*(\tau_n)) \quad n \geq 1$$

□

Definition 3.1.7 *The set of sort clauses* $C_\Omega(\chi)$

Let Ω be a sort signature, and let $\chi \subseteq \Phi$ be a set of sort variables. $B_1, \ldots, B_n \Rightarrow H \in \mathrm{C}_\Omega(\chi)$ iff

- $n \geq 0$
- $\{B_1, \ldots, B_n, H\} \subseteq \mathrm{AF}_\Omega(\chi)$ is a set of atomic sort formulae

H is called the *head* and $B_1,\ldots,B_n$ is called the *body* of the sort clause. If $n = 0$ we will omit the delimiter $\Rightarrow$. If $C \in \mathrm{C}_\Omega(\emptyset)$ then C is called a *ground sort clause*. The set of sort variables occurring in a sort clause C is denoted by $FSV(C)$. □

Definition 3.1.8 *Sort clause substitution*

A *sort clause substitution* is an Ω homomorphism $\sigma^* : \mathrm{C}_\Omega(\Phi) \rightarrow \mathrm{C}_\Omega(\Phi)$ extending a sort variable substitution σ to sort clauses in the following way:

$$\sigma^*(B_1,\ldots,B_n \Rightarrow H) := \sigma^*(B_1),\ldots,\sigma^*(B_n) \Rightarrow \sigma^*(H) \quad n \geq 0$$

□

All substitutions are extended naturally to sets of sort terms, sort formulae or sort clauses by applying the substitutions to each element of the set. In the sequel we will omit the asterisk denoting the extension of a sort variable substitution to some structure containing sort variables.

Definition 3.1.9 *The set of sort specifications SSPEC*

$(\Omega, SA) \in$ SSPEC iff

- Ω is a sort signature
- $SA \subseteq \mathrm{C}_\Omega(\Phi)$ is a set of sort clauses

□

Example 3.1.3 *Sort specification*

Let Ω be the sort signature defined in Example 3.1.1.

$$\begin{array}{l}(\Omega,\ \{\ \texttt{PO}(\alpha) \Rightarrow \texttt{EQ}(\alpha),\\ \qquad\quad \texttt{EQ}(\alpha) \Rightarrow \texttt{EQ(List}(\alpha)),\\ \qquad\quad \texttt{PO(Nat), EQ(Bool)}\ \})\end{array}$$

is an example for a sort specification. □

The sort specification is used to describe the properties of the sort constructors and the sort predicates. Ground sort clauses with an empty body are specifying the properties of the basic sorts. Non-ground sort clauses are usually used to define dependencies between sort predicates or to propagate properties to non-basic sorts.

Before giving a semantics for sort specifications we will define the whole syntactic framework of our language. In the signature of an object specification we want to declare polymorphic identifiers. For this purpose we need the concept of polymorphic sort terms. These build the basis of our polymorphism approach.

Definition 3.1.10 *The set of polymorphic sort terms* T_Ω^Π

$\Pi\alpha_1, \ldots, \alpha_n.\ A_1, \ldots, A_m \Rightarrow \tau \in \mathrm{T}_\Omega^\Pi$ iff

- $n \geq 1$, $m \geq 0$
- $\{A_1, \ldots, A_m\} \subseteq \mathrm{AF}_\Omega^{ng}(\{\alpha_1, \ldots, \alpha_n\})$, i.e. A_i are atomic sort formulae containing at least one sort variable
- $\tau \in \mathrm{T}_\Omega(\{\alpha_1, \ldots, \alpha_n\})$ where $FSV(\tau) = \{\alpha_1, \ldots, \alpha_n\}$, i.e. each α_i, $1 \leq i \leq n$, must occur in τ

The atomic sort formulae A_i are called *qualifying sort predicates*. The monomorphic sort expression τ is called the *body* of the polymorphic sort term. If $m = 0$ we will omit the separator $\Rightarrow$. These sort terms are called *pure polymorphic sort terms*. $\mathrm{T}_\Omega^\Pi(\Phi) := \mathrm{T}_\Omega(\Phi) \cup \mathrm{T}_\Omega^\Pi$ is called the *set of sort terms*. $\mathrm{T}_\Omega := \mathrm{T}_\Omega(\emptyset) \cup \mathrm{T}_\Omega^\Pi$ is called the *set of closed sort terms*. □

In the sequel a list of syntactic objects $s_1, \ldots, s_n$ is abbreviated by $\overline{s_n}$. For instance $\Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau$ is equivalent to $\Pi\alpha_1, \ldots, \alpha_n.\ A_1, \ldots, A_m \Rightarrow \tau$.

Example 3.1.4 *Polymorphic sort terms*

Let Ω be the sort signature defined in Example 3.1.1.

$$\Pi\alpha.\mathtt{EQ}(\alpha) \Rightarrow \times(\alpha, \alpha) \rightarrow \mathtt{Bool}$$
$$\Pi\alpha, \beta.\mathtt{IS\_A}(\alpha, \beta) \Rightarrow \alpha \rightarrow \beta$$

are two polymorphic sort terms. □

Each qualifying sort predicate must at least contain one sort variable. Since sort predicates are used to restrict the instances of the sort variables, ground sort predicates are senseless. Moreover, each Π-bound sort variable of a polymorphic sort term must occur in the body. This restriction simplifies the theoretical treatment of polymorphism because the instances of the sort variables are determined by the application context of the identifier. For pure polymorphic sort terms this restriction is even sensible. The Π-binding allows abstracting from a sort in a sort term. But abstraction only makes sense if the sort really occurs in the sort term. In combination with qualifying n-ary sort predicates, however, it is sometimes sensible if a sort variable occurs in a qualifying predicate, but not in the body of the polymorphic sort term. In this case an equivalent behaviour can be achieved by using an auxiliary sort predicate not containing the sort variable.

Our definition of polymorphic sort terms allows only the usage of first-order polymorphic objects. The binding Π's can not be nested. A Π can only occur at the outermost level of

a polymorphic sort term. As a consequence, polymorphic functions taking a polymorphic object as an argument cannot be specified. This is, e.g., possible in the polymorphic λ-calculus (see Section 1.2.6).

In polymorphic sort terms sort variables may be qualified by sort predicates. The task of the sort specification is to define the sorts satisfying a particular sort predicate. When instantiating these sort variables with a particular sort, it must be proven whether this sort satisfies the predicates with respect to the sort specification. This proof plays a central part in the sort inference process at the object level. We must therefore provide a calculus to compute whether a given atomic sort formula is entailed by a sort specification. This entailment relation will be used in the sort inference calculus to define the well-sortedness of terms.

Definition 3.1.11 *Entailment relation* $\vdash$

An *entailment relation* $\vdash \subseteq (\mathcal{P}(\mathrm{C}_\Omega(\Phi)) \times \mathcal{P}(\mathrm{AF}_\Omega(\Phi)) \times \mathrm{AF}_\Omega(\Phi))$ is a set of triples (SA, Δ, A) where:

- SA is a set of sort clauses
- $\Delta \subseteq \mathrm{AF}_\Omega(\Phi)$ is a set of atomic sort formulae called the *sort predicate assumption*
- $A \in \mathrm{AF}_\Omega(\Phi)$ is an atomic sort formula

We write $\Delta \vdash_{SA} A$ instead of $(SA, \Delta, A) \in \vdash$ and read "the sort predicate assumption Δ entails the atomic sort formula A with respect to the sort axioms SA". $\Delta \vdash_{SA} A$ holds iff there is a finite proof tree using the inference rules below that ends with $\Delta \vdash_{SA} A$. A particular finite proof tree ending with $\Delta \vdash_{SA} A$ is denoted by $\Delta \bigtriangledown_{SA} A$. In the rules we omit the set brackets and write a comma for the union of sets on the lefthandside of $\vdash_{SA}$.

$$\frac{}{\Delta, A \vdash_{SA} A}\ (assumption)$$

$$\frac{\Delta \vdash_{SA} \sigma(B_1) \quad \ldots \quad \Delta \vdash_{SA} \sigma(B_n)}{\Delta \vdash_{SA} \sigma(H)}\ (mp) \begin{cases} B_1, \ldots, B_n \Rightarrow H \in SA \\ \sigma : \Phi \to \mathrm{T}_\Omega(\Phi) \end{cases}$$

We extend the relation to sets of atomic sort formulae Δ_1 and Δ_2 in the following way:

$$\Delta_1 \vdash_{SA} \Delta_2 \qquad \text{iff} \qquad \Delta_1 \vdash_{SA} A \qquad \text{for each } A \in \Delta_2$$

□

The sort clauses in SA are treated as universally quantified. In rule (mp) the sort variables occurring in a sort clause can be arbitrarily instantiated by a sort variable

substitution. The atomic sort formulae in the assumption Δ, in contrast, are not treated as universally quantified. These sort variables denote a fixed sort.

In [Jon92] Jones presents a sort system for a functional language based on qualified sorts. His sort system is independent of a particular entailment relation $\vdash$. The relation is only assumed to be monotonic, closed with respect to transitivity, and closed with respect to sort variable substitution. The following propositions establish these properties for our entailment relation.

Proposition 3.1.1 *Monotonicity of* $\vdash$

Let Δ_1 and Δ_2 be two sets of atomic sort formulae.

$$\text{if} \quad \Delta_2 \subseteq \Delta_1 \quad \text{then} \quad \Delta_1 \vdash_{SA} \Delta_2$$

□

Proposition 3.1.2 *Substitution Closure Property of* $\vdash$

Let Δ be a set of atomic sort formulae, and let A be an atomic sort formula. Let $\sigma : \Phi \to \mathrm{T}_\Omega(\Phi)$ be a sort variable substitution.

$$\text{if} \quad \Delta \vdash_{SA} A \quad \text{then} \quad \sigma(\Delta) \vdash_{SA} \sigma(A)$$

The proposition can be extended to sets of atomic sort formulae Δ_1 and Δ_2:

$$\text{if} \quad \Delta_1 \vdash_{SA} \Delta_2 \quad \text{then} \quad \sigma(\Delta_1) \vdash_{SA} \sigma(\Delta_2)$$

□

Proposition 3.1.3 *Transitivity closure property of* $\vdash$

Let A_1 and A_2 be two atomic sort formulae, and let Δ be a set of atomic sort formulae.

$$\text{i)} \quad \text{if} \quad \Delta \vdash_{SA} A_1 \quad \text{and} \quad A_1 \vdash_{SA} A_2 \quad \text{then} \quad \Delta \vdash_{SA} A_2$$

The proposition can be extended to sets atomic sort formulae Δ_1, Δ_2, and Δ_3.

$$\text{ii)} \quad \text{if} \quad \Delta_1 \vdash_{SA} \Delta_2 \quad \text{and} \quad \Delta_2 \vdash_{SA} \Delta_3 \quad \text{then} \quad \Delta_1 \vdash_{SA} \Delta_3$$

□

Example 3.1.5 *Proof of an entailment relation*

Let $S = (\Omega, SA)$ be the sort specification given in Example 3.1.3. The proof tree in Fig. 3.1 shows that the sort predicate assumption $\{\texttt{PO}(\beta)\}$ entails the sort clause $\texttt{EQ}(\beta)$ with respect to the sort axioms SA. □

$$\dfrac{\dfrac{}{\mathtt{PO}(\beta)\vdash_{SA}\mathtt{PO}(\beta)}\,(assumption)}{\mathtt{PO}(\beta)\vdash_{SA}\mathtt{EQ}(\beta)}\,(mp) \qquad \begin{cases} \mathtt{PO}(\alpha)\Rightarrow\mathtt{EQ}(\alpha)\in SA \\ \sigma=[\beta/\alpha] \end{cases}$$

Figure 3.1: Proof tree for $\mathtt{PO}(\beta)\vdash_{SA}\mathtt{EQ}(\beta)$

3.2 The Language of Polymorphic Specifications

On the object level we use a classical first-order predicate logic for specifying the properties of our objects. The formula language only provides a minimal set of constructs necessary for first order predicate logic. The pure implicitly sorted λ-calculus with constants forms the term language of our object language. Thus, the specification language is not suited for practical use. This minimal syntactic framework was chosen to keep the definitions, propositions and proofs as simple as possible, without losing expressive power. The object language can be used as a core language for an extended, more user-oriented specification language. The semantics of the extended constructs can be defined by a translation to the core language.

Because the language is restricted to the pure λ-calculus, it does not feature any built-in concept of a program state. As a consequence the specification style is oriented towards functional programming. This restriction is fundamental for this thesis. The theoretical results can not simply be transferred to a state-oriented specification language. The restriction to first order logic, however, is not serious. The results can be easily applied to a higher order logic.

As on the sort level, polymorphic specifications are divided into a signature and a set of axioms. In classical many-sorted approaches, the signature of a specification contains the sort identifiers as well as the object and predicate identifiers of the current specification. Our signatures, called *polymorphic signatures*, contain a complete sort specification instead of a set of sort identifiers. Moreover, both predicates and functions may be declared polymorphically.

Definition 3.2.1 *The set of polymorphic signatures PSIG*

$(S, P, F) \in$ PSIG iff

- $S = (\Omega, SA)$ is a sort specification
- $P = \{P_\tau\}_{\tau\in T_\Omega} \cup P_\epsilon$ is an indexed set of predicate identifiers, where P_ϵ contains the identifiers for constant predicates
- $F = \{F_\tau\}_{\tau\in T_\Omega}$ is an indexed set of object identifiers

- the predicate identifiers as well as the object identifiers must be unique, i.e. we do not allow overloaded object identifiers or predicate identifiers in a polymorphic signature.

Instead of writing $p \in P_\tau$ or $f \in F_\tau$ we will sometimes write $p{:}\tau \in P$ and $f{:}\tau \in F$. We will read "p is of sort τ" and "f is of sort τ". Because we do not allow overloaded identifiers, P and F can be interpreted as functions assigning a sort term to an identifier. An identifier id is called *polymorphic* iff $id \in P_\tau \cup F_\tau$, where $\tau \in \mathrm{T}^{\Pi}_{\Omega}$ is a polymorphic sort. All other identifiers are called *monomorphic*. If it is not important for understanding, we will not distinguish between the identifier and the denoted object. We will, for example, use the word "predicate" instead of "predicate identifier". □

Example 3.2.1 *Polymorphic signature*

Let S be the sort specification given in Example 3.1.3.

$$
\begin{array}{l}
(\ S, \\
\quad \{\{\texttt{isin}\}_{\Pi\alpha.EQ(\alpha)\Rightarrow\times(\alpha,List(\alpha))},\ \{\texttt{=}\}_{\Pi\alpha.\times(\alpha,\alpha)}\}, \\
\quad \{\{\texttt{pair}\}_{\Pi\alpha,\beta.\alpha\to\beta\to\times(\alpha,\beta)}, \\
\qquad \{\texttt{0}\}_{Nat},\ \{\texttt{succ}\}_{Nat\to Nat}, \\
\qquad \{\texttt{empty}\}_{\Pi\alpha.List(\alpha)},\ \{\texttt{append}\}_{\Pi\alpha.\alpha\to List(\alpha)\to List(\alpha)}\}\ )
\end{array}
$$

is an example for a polymorphic signature. □

Our conception of signature differs from the usual notion of signature, by allowing the declaration of polymorphic functions and predicates. The signature is the only place to declare polymorphic identifiers. In the axioms part all variables declared by a binding operator must be monomorphic. Note that the sets of predicate and object identifiers can only be indexed by closed sort terms. This allows the usage of sort variables only if they are bound by a Π. Thus, monomorphic identifiers are classically sorted identifiers without sort variables.

The axioms part of our object language is not restricted to equational or conditional-equational axioms. The language allows full first-order formulae. Of course, the language can be specialized if desired. As usual in traditional logics, we strictly distinguish between terms and formulae. The results, however, can be transfered to a language mixing both layers, like the specification language SPECTRUM.

The definition of terms and formulae is performed in two steps. In the first step we give a context-free grammar in EBNF style to define the raw structure of terms and formulae. However, a context-free description can not completely define a statically sorted language. Sorting constraints are typically context sensitive. Therefore, we will define the well-sorted terms and formulae using a logical calculus.

Definition 3.2.2 *The set of pre-terms* $PT_\Sigma(\chi)$

Let $\Sigma = (S, P, F)$ be a polymorphic signature. Let $\chi \subseteq \Phi$ be a set of sort variables. Let $<id>=\bigcup_{\tau \in T_\Omega} F_\tau$ be the set of all object identifiers from the signature. Let $<var>$ be an arbitrary set of variables different from $<id>$, i.e. $<var> \cap <id>=\emptyset$. $\mathrm{PT}_\Sigma(\chi)$ is the set $<term>$ defined by the following EBNF grammar[2]:

$<term>$	::=	$<id>$	*object identifier*
	|	$<var>$	*variable*
	|	$<term>$ $<term>$	*application*
	|	λ $<svar>$. $<term>$	*abstraction*
	|	$<term>$:$<sortterm>$	*sort constrained term*
$<svar>$	::=	$<var>$	*unsorted variable*
	|	$<var>$:$<sortterm>$	*sorted variable*

The application of terms associates to the left. The set of sort variables occurring in a term t is denoted by $FSV(t)$. □

Definition 3.2.3 *The set of pre-formulae* $PF_\Sigma(\chi)$

Let Σ, χ, $<id>$ and $<var>$ be defined as in Def. 3.2.2. Let $<pid>=\bigcup_{\tau \in T_\Omega} P_\tau \cup P_\epsilon$ be the set of all predicate identifiers from the signature. $\mathrm{PF}_\Sigma(\chi)$ is the set $<formula>$ defined by the following EBNF grammar:

$<formula>$	::=	$<pid>$	*predicate identifier*
	|	$<pid>$ $<term>$	*predicate application*
	|	$\forall$ $<svar>$. $<formula>$	*universal quantification*
	|	$\neg$ $<formula>$	*negation*
	|	$<formula>$ $\vee$ $<formula>$	*disjunction*

The set of sort variables occurring in a formula f is denoted by $FSV(f)$. □

Example 3.2.2 *Pre-Formula*

Let Σ be the polymorphic signature defined in Example 3.2.1.

$$\forall \texttt{ x. } \neg \texttt{isin(pair x empty)}$$

is an example for a pre-formula from $\mathrm{PF}_\Sigma(\Phi)$. □

Let us examine the introduced language in more detail. Since it can be derived from the grammar, locally bound variables can be declared without sort annotation. A correct sort annotation can be inferred from the application context of the variable. This feature

[2] See Appendix B.1 for a definition of the notation.

enables one to write sort-free axioms. As a result we achieve the convenience of an unsorted language but keep the advantage of a static sort system. With the help of the sort inference calculus given in Def. 3.2.7 and 3.2.9 the well-formedness of pre-terms and pre-formulae can be proven. This calculus determines a sort for every local variable. In most cases, however, there is not only one possible sort for a variable, but many different sorts, to get a well-formed term or formula.

In a functional language with pure parametric polymorphism, e.g. ML, the semantics of a term does not depend on the way it is sort-checked and therefore does not depend on the sort chosen for a λ-bound variable. This property is called *coherence* (see [BTCGS89]). As we will see this property does not hold for a polymorphic specification language. The sort of a locally bound variable severely influences the semantics of a specification.

To restrict the sort of a variable to the desired sort, it is sometimes necessary to explicitly annotate locally declared variables by a sort expression. Therefore, the language allows one to declare unsorted variables as well as sorted variables. Furthermore, it allows one to explicitly constrain entire terms. This is also necessary to influence the semantics of a specification. It can be used to choose a concrete instantiation of a polymorphic identifier.

Note that in a functional language with type classes, e.g. Gofer, the sort of a term can also influence the semantics of the term. This feature is known as the *coherence problem* and is discussed in Chapter 5 of [Jon92]. We will also discuss this problem in Section 6.2.

Local variables or entire terms can only be annotated by monomorphic sort terms. The sort terms may contain sort variables, but they must not be bound by a Π. Therefore, the language does not allow one to declare local polymorphic identifiers. Though sort terms may contain sort variables, they are not polymorphic. A sort variable occurring in a formula denotes an arbitrary, but fixed sort that must not change within a formula. We will explain the different syntactic treatment of polymorphic identifiers and monomorphic identifiers with sort variables after giving the calculi defining well-sorted terms and formulae. The different semantic treatment will be explained in Chapter 4.

If we only provided our language with ML-polymorphism, we would need no explicit binding mechanism for sort variables in formulae. The free sort variables can be seen as universally quantified at the outermost level of a formula. However, the use of sort predicates in the functionality of polymorphic identifiers, means it is sometimes necessary to explicitly constrain sort variables in formulae by sort predicates. The reason is the same as for the explicit annotation of local variables or terms: the explicit restriction of sort variables is used to influence the semantics of a specification. Thus, we define the concept of qualified formulae. The concrete semantic difference between formulae and qualified formulae will be explained in Chapter 4.

Definition 3.2.4 *The set of qualified formulae QF_Σ*

Let Σ be a polymorphic signature. $\forall\alpha_1,\ldots,\alpha_n.\ A_1,\ldots,A_m \Rightarrow f \in \mathrm{QF}_\Sigma$ iff

- $n \geq 0, m \geq 0$
- $\{A_1, \ldots, A_m\} \subseteq \mathrm{AF}^{ng}_\Omega(\{\alpha_1,\ldots,\alpha_n\})$, i.e. A_i are non-ground atomic sort formulae
- $f \in \mathrm{PF}_\Sigma(\{\alpha_1,\ldots,\alpha_n\})$ is a pre-formula in which only the sort variables $\alpha_1,\ldots,\alpha_n$ occur

The atomic sort formulae A_i are called the *sort predicate context* of a qualified formula. In case of $n = 0$ we omit the quantifier $\forall$. This also implies an empty sort predicate context, because each atomic sort formulae must at least contain one sort variable. If, however, the sort predicate context is empty, i.e. $m = 0$, we do not omit the separator $\Rightarrow$, even if both $n = 0$ and $m = 0$. In particular, we want to distinguish between a qualified formula with an empty sort predicate context and a formula without any sort predicate context. We will explain the difference after defining the well-formedness of formulae and qualified formulae. Note that a qualified formula is always closed with respect to sort variables, i.e. no free sort variables occur in a qualified formula. □

Example 3.2.3 *Qualified formulae*

Let Σ be the polymorphic signature defined in Example 3.2.1.

$$\forall\alpha.\ \texttt{EQ}(\alpha) \Rightarrow \forall\ \texttt{x}{:}\alpha.\ \neg\,\texttt{isin(pair x empty)}$$
$$\forall\alpha.\ \Rightarrow \forall\ \texttt{x}{:}\alpha.\ \neg\,\texttt{isin(pair x empty)}$$
$$\Rightarrow \forall\ \texttt{x}.\ \neg\,\texttt{isin(pair x empty)}$$

are examples for qualified formulae. □

In the sort predicate context no ground sort formulae are allowed. The sort predicate context is used to explicitly qualify sort variables used in formulae. There is no obvious reason to allow ground atomic sort formulae which do not qualify any sort variable. Furthermore, facts in a sort predicate context do not influence the models at the sort level.

To define the context sensitive well-formedness of terms, formulae, and qualified formulae, we use calculi with axioms and inference rules, called *sort inference calculi*. The calculi are used to derive a possible sort for a given term. As result, the sorts of unsorted variables and the instances of polymorphic identifiers are also determined. The deduction oriented formulation is a convenient way to keep track of varying assumptions which arise from various binding mechanisms. The first calculus below is an extension of [Jon92]

and is used to define well-sorted terms. The other two calculi are extensions of the first one and define well-formed non-qualified formulae, and qualified formulae, respectively. To keep track of the varying local variables declared in terms and formulae we use a variable assumption defined in the following way.

Definition 3.2.5 *Variable assumption*

A variable assumption Γ is a finite set of tuples $(x, \tau) \in <var> \times \mathrm{T}_\Omega(\Phi)$ in which no variable appears more than once. We write $x{:}\tau \in \Gamma$ instead of $(x, \tau) \in \Gamma$. Furthermore, we write $\Gamma.x{:}\tau$ as an abbreviation for $(\Gamma \setminus \{x{:}\tau'\}) \cup \{x{:}\tau\}$ for some τ'. We will use this notation to replace an old sort assignment to a variable by a new one. Because each variable appears only once in a variable assumption, it can be interpreted as a function which assigns monomorphic sort expressions to variables, i.e. $\Gamma \in <var> \to \mathrm{T}_\Omega(\Phi)$. In particular, if $x{:}\tau \in \Gamma$ we also write $\Gamma(x){=}\tau$. The set of sort variables occurring in Γ is denoted by $FSV(\Gamma) := \bigcup_{x:\tau\in\Gamma} FSV(\tau)$ □

Definition 3.2.6 *Substitution of variable assumptions*

Let $\sigma{:}\Phi \to \mathrm{T}_\Omega(\Phi)$ be a sort variable substitution. We extend σ to variable assumptions in the following way:

$$\sigma(\Gamma) := \{x{:}\sigma(\tau) \mid x{:}\tau \in \Gamma\}$$

□

Definition 3.2.7 *Term judgement* $\rhd$

The term judgement $\rhd \subseteq \mathrm{PSIG} \times \mathcal{P}(\mathrm{AF}_\Omega(\Phi)) \times (<var> \to \mathrm{T}_\Omega(\Phi)) \times \mathrm{PT}_\Sigma(\Phi) \times \mathrm{T}_\Omega(\Phi)$ is a set of quintuples $(\Sigma, \Delta, \Gamma, e, \tau)$ where:

- $\Sigma = (S, P, F) \in \mathrm{PSIG}$ is a polymorphic signature
- $\Delta \subseteq \mathrm{AF}_\Omega(\Phi)$ is a set of atomic sort formulae, called *sort predicate assumption*
- $\Gamma \in <var> \to \mathrm{T}_\Omega(\Phi)$ is a variable assumption. By using a function, it is guaranteed that no variable occurs twice in an assumption. This prohibits overloading of variables in a scope.
- $e \in \mathrm{PT}_\Sigma(\Phi)$ is the pre-term to be checked
- $\tau \in \mathrm{T}_\Omega(\Phi)$ is the derived sort of e

We write $\Delta,\ \Gamma \rhd_\Sigma e :: \tau$ instead of $(\Sigma, \Delta, \Gamma, e, \tau) \in \rhd$ and read "under sort predicate assumption Δ, and variable assumption Γ, term e is a well-formed term of sort τ with respect to the polymorphic signature Σ". $\Delta,\ \Gamma \rhd_\Sigma e :: \tau$ holds iff there is a finite

proof tree using the inference rules below that ends with $\Delta,\ \Gamma\ \rhd_\Sigma\ e\ ::\ \tau$. A proof tree ending with $\Delta,\ \Gamma\ \rhd_\Sigma\ e\ ::\ \tau$ is called a *sort derivation for e under Δ and Γ with respect to Σ*. A particular finite proof tree ending with $\Delta,\ \Gamma\ \rhd_\Sigma\ e\ ::\ \tau$ is denoted by $\Delta,\ \Gamma\ \bigtriangledown_\Sigma\ e\ ::\ \tau$. The set of sort variables occurring in a sort derivation D is denoted by $FSV(D) := \bigcup_{all\ nodes\ \Delta,\Gamma\ \rhd_\Sigma\ e::\tau\ in\ D} FSV(\tau)$.

Axioms

$$\frac{}{\Delta,\Gamma.x{:}\tau \rhd_\Sigma x\ ::\ \tau}\ (var)$$

$$\frac{}{\Delta,\Gamma \rhd_\Sigma c\ ::\ \tau}\ (id)\ \begin{cases} F(c) = \tau \\ \text{where } \tau \in \mathrm{T}_\Omega(\emptyset) \end{cases}$$

$$\frac{}{\Delta,\Gamma \rhd_\Sigma c\ ::\ \sigma(\tau)}\ (polyid)\ \begin{cases} F(c) = \Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau \\ \sigma(A_i) \in \Delta, 1 \le i \le m \\ \text{for some } \sigma : \Phi \to \mathrm{T}_\Omega(\Phi) \end{cases}$$

Inference Rules

$$\frac{\Delta,\Gamma \rhd_\Sigma e_1\ ::\ \tau_2 \longrightarrow \tau_1 \quad \Delta,\Gamma \rhd_\Sigma e_2\ ::\ \tau_2}{\Delta,\Gamma \rhd_\Sigma e_1 e_2\ ::\ \tau_1}\ (\to E)$$

$$\frac{\Delta,\Gamma.x{:}\tau_1 \rhd_\Sigma e\ ::\ \tau_2}{\Delta,\Gamma \rhd_\Sigma \lambda x.e\ ::\ \tau_1 \longrightarrow \tau_2}\ (\to I_u) \qquad \frac{\Delta,\Gamma.x{:}\tau_1 \rhd_\Sigma e\ ::\ \tau_2}{\Delta,\Gamma \rhd_\Sigma \lambda x{:}\tau_1.e\ ::\ \tau_1 \longrightarrow \tau_2}\ (\to I_s)$$

$$\frac{\Delta,\Gamma \rhd_\Sigma e\ ::\ \tau}{\Delta,\Gamma \rhd_\Sigma e{:}\tau\ ::\ \tau}\ (constrained)$$

□

Definition 3.2.8 *Well-formedness of a pre-term*

A pre-term $e \in \mathrm{PT}_\Sigma(\Phi)$ is called *well-formed with respect to a signature* Σ iff there exists a sort predicate assumption Δ, a variable assumption Γ, and a monomorphic sort term τ such that $\Delta,\ \Gamma\ \rhd_\Sigma\ e\ ::\ \tau$ and $\emptyset \vdash_{SA} \Delta \cap \mathrm{AF}_\Omega(\emptyset)$. □

Definition 3.2.9 *Formula judgement* $\rhd$

The formula judgement $\rhd \subseteq \mathrm{PSIG} \times \mathcal{P}(\mathrm{AF}_\Omega(\Phi)) \times (<var> \to \mathrm{T}_\Omega(\Phi)) \times \mathrm{PF}_\Sigma(\Phi)$ is a set of quadruples $(\Sigma, \Delta, \Gamma, f)$ where:

- Σ, Δ and Γ have the same meaning as in definition 3.2.7.
- $f \in \mathrm{PF}_\Sigma(\Phi)$ is the pre-formula to be checked

We write $\Delta,\ \Gamma\ \rhd_\Sigma\ f$ instead of $(\Sigma,\Delta,\Gamma,f)\in\rhd$ and read "under sort predicate assumption Δ and variable assumption Γ formula f is well-formed with respect to the polymorphic signature Σ". $\Delta,\ \Gamma\ \rhd_\Sigma\ f$ holds iff there is a finite proof tree using the inference rules below that ends with $\Delta,\ \Gamma\ \rhd_\Sigma\ f$. A proof tree ending with $\Delta,\ \Gamma\ \rhd_\Sigma\ f$ is called a *sort derivation for f under* Δ *and* Γ *with respect to* Σ. A particular finite proof tree ending with $\Delta,\ \Gamma\ \rhd_\Sigma$ f is denoted by $\Delta,\ \Gamma\ \bigtriangledown_\Sigma\ f$. Note that the variable assumption is only used for object variables. In a first order logic there are no local binding mechanisms for predicates. All predicates used in a specification must be defined in the signature. Thus, we need no dynamic context for predicates.

Axioms

$$\frac{}{\Delta,\Gamma\rhd_\Sigma p}\ (const)\left\{\ p\in P_\epsilon\right.$$

$$\frac{\Delta,\Gamma\rhd_\Sigma t::\tau}{\Delta,\Gamma\rhd_\Sigma p\,t}\ (appl)\left\{\begin{array}{l}P(p)=\tau\\ \text{where } \tau\in \mathrm{T}_\Omega(\emptyset)\end{array}\right.$$

$$\frac{\Delta,\Gamma\rhd_\Sigma t::\sigma^*(\tau)}{\Delta,\Gamma\rhd_\Sigma p\,t}\ (pappl)\left\{\begin{array}{l}P(p)=\Pi\alpha_1,\ldots,\alpha_n.A_1,\ldots,A_m\Rightarrow\tau\\ \sigma(A_i)\in\Delta, 1\le i\le m\\ \text{for some } \sigma:\Phi\to \mathrm{T}_\Omega(\Phi)\end{array}\right.$$

Inference Rules

$$\frac{\Delta,\Gamma.x{:}\tau\rhd_\Sigma f}{\Delta,\Gamma\rhd_\Sigma\forall x.f}\ (univ_u)\qquad \frac{\Delta,\Gamma.x{:}\tau\rhd_\Sigma f}{\Delta,\Gamma\rhd_\Sigma\forall x{:}\tau.f}\ (univ_s)$$

$$\frac{\Delta,\Gamma\rhd_\Sigma f}{\Delta,\Gamma\rhd_\Sigma\neg f}\ (not)\qquad \frac{\Delta,\Gamma\rhd_\Sigma f_1\quad\Delta,\Gamma\rhd_\Sigma f_2}{\Delta,\Gamma\rhd_\Sigma f_1\vee f_2}\ (or)$$

□

Definition 3.2.10 *Well-formedness of a pre-formula*

A pre-formula $f\in \mathrm{PF}_\Sigma(\Phi)$ is called *well-formed with respect to a signature* Σ iff there exists a sort predicate assumption Δ and a variable assumption Γ, such that $\Delta,\ \Gamma\ \rhd_\Sigma\ f$ and $\emptyset\vdash_{SA}\Delta\cap \mathrm{AF}_\Omega(\emptyset)$. □

The most interesting inference rules of the calculi are the rules *(polyid)* and *(pappl)*, handling polymorphic identifiers. Both rules immediately instantiate the polymorphic

identifiers, i.e. they apply a suitable sort variable substitution to the body of the polymorphic sort. The instantiation yields a monomorphic sort term for the used identifier. Thus, a term can never be polymorphically sorted though polymorphic identifiers are applied in the term. This mechanism avoids higher order polymorphic functions, whose drawbacks were already discussed in Section 1.2.6. Note, however, that a polymorphic identifier can be instantiated differently at each occurrence. This was the reason for the introduction of polymorphism.

Our handling of polymorphic identifiers does not differ from the usual handling in functional languages. There, polymorphic identifiers usually can only be defined by the let-construct or a similar mechanism. If the identifier is applied in the body of the let-construct, it is instantiated to a monomorphic identifier. In our specification language a polymorphic identifier can only be declared in the signature. Thus, the signature plays the role of the let-construct of a functional language with respect to polymorphic identifiers.

All variables defined by a λ-abstraction or the universal quantifier $\forall$, though possibly containing sort variables, are monomorphic variables. They are handled by rule (var) of Def. 3.2.7. This rule does not, in contrast to rule $(polyid)$, instantiate the sort term of the variable at application. Therefore, the sort of a variable remains the same at each occurrence of the variable[3]. As a consequence, polymorphic identifiers cannot be deleted from the signature and instead added to the axioms by an existential quantifier at the outermost level. This is possible in most non-polymorphic specification languages supporting higher order functions.

Our definition of well-formedness only demands the existence of an arbitrary sort derivation with the additional restriction that all ground atomic sort formulae occurring in the sort predicate assumption must be entailed by the empty assumption. Ground atomic sort formulae representing facts may only be entailed by the sort specification of the signature. Without this restriction the qualifying sort predicates of the polymorphic identifiers could never influence the well-formedness of terms and formulae. The sort predicate assumption is only used by the rules *(polyid)* and *(pappl)*. These rules can only be applied if the instantiated qualifying sort predicates A_i of the polymorphic identifier are elements of the sort predicate assumption Δ. Without this restriction, each possible instantiation of a polymorphic identifier would be a well-formed term.

Our concept of well-formedness is very liberal. There are other, more restricted approaches. The functional language Haskell[4] allows only sort predicate assumptions of the form $P(\alpha)$, where P is a unary sort predicate and α is a sort variable. This restriction can be found in the specification language SPECTRUM and in the theorem prover

[3] Of course only within the same binding.

[4] Note that an exact comparison is difficult because the terminology does not go together.

Isabelle, too. In contrast, Gofer allows, as we do, arbitrary sort terms. The advantages and drawbacks of both approaches are discussed in Chapter 7.1 of [Jon92]. This discussion, however, is very specific to a functional programming language. In the framework of an axiomatic specification language there is no obvious reason for this more restricted approach.

The most restrictive well-formedness approach would be to allow only sort predicate assumptions that are completely entailed by the empty assumption. In the setting of sort classes, such a concept of well-formedness is obviously too rigid. In that case, a sort class can only be used if there exists at least one sort belonging to this class.

In the setting of subsorts, however, such a rigid concept of well-formedness might prevent unintended function applications. Our very liberal approach allows, e.g., a sort predicate context $\{$`IS_A(`α`,Bool),IS_A(`α`,Nat)`$\}$. That means, sort variable α must be a subsort of sort `Bool` as well as of sort `Nat`. Usually no sort satisfies this context. Such a context probably results from an unintended use of some function. As we will see, in our approach such sort predicate assumptions can be avoided by using qualified formulae.

Example 3.2.4 *Well-formedness of a pre-formula*

We show that the pre-formula $\forall$ `x.` $\neg$ `isin(pair x empty)` from Ex. 3.2.2 is well-formed with respect to the signature from Ex. 3.2.1 by giving a proof tree for `EQ(`α`),`$\emptyset$ $\rhd_\Sigma$ $\forall$ `x.` $\neg$ `isin(pair x empty)` in Fig. 3.2[5].

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{EQ}(\alpha),\texttt{x:}\alpha\rhd_\Sigma\texttt{pair::}\alpha\to\texttt{List}(\alpha)\to\times(\alpha,\texttt{List}(\alpha))}\,(polyid) \quad \dfrac{}{\texttt{EQ}(\alpha),\texttt{x:}\alpha\rhd_\Sigma\texttt{x::}\alpha}\,(var)}{\texttt{EQ}(\alpha),\texttt{x:}\alpha\rhd_\Sigma\texttt{pair x::List}(\alpha)\to\times(\alpha,\texttt{List}(\alpha))}\,(\to E) \quad \dfrac{}{\texttt{EQ}(\alpha),\texttt{x:}\alpha\rhd_\Sigma\texttt{empty::List}(\alpha)}\,(polyid)}{\texttt{EQ}(\alpha),\texttt{x:}\alpha\rhd_\Sigma\texttt{pair x empty::}\times(\alpha,\texttt{List}(\alpha))}\,(\to E)}{\texttt{EQ}(\alpha),\texttt{x:}\alpha\rhd_\Sigma\texttt{isin(pair x empty)}}\,(pappl)}{\texttt{EQ}(\alpha),\texttt{x:}\alpha\rhd_\Sigma\neg\texttt{isin(pair x empty)}}\,(not)}{\texttt{EQ}(\alpha),\emptyset\rhd_\Sigma\forall\texttt{x. }\neg\texttt{isin(pair x empty)}}\,(univ_u)$$

Figure 3.2: Sort derivation for $\forall$ `x.` $\neg$ `isin(pair x empty)`

This is not the only sort derivation proving the well-formedness of the formula. If we replace each occurrence of the sort variable α in the proof tree by the basic sort `Bool` we get another possible sort derivation for the formula. Because `EQ(Bool)` is a fact from the sort specification of the signature we can prove $\emptyset\vdash_{SA}$`EQ(Bool)`.

[5]For reasons of space the derivation tree is split into two parts.

However, replacing α by `Bool→Bool` does not yield a sort derivation proving the well-formedness of the formula, because we can not prove $\emptyset \vdash_{SA} \texttt{EQ(Bool→Bool)}$. □

Definition 3.2.11 *Qualified formula judgement* $\rhd$

The qualified formula judgement $\rhd \subseteq \text{PSIG} \times \mathcal{P}(\text{AF}_\Omega(\Phi)) \times (<var> \rightarrow \text{T}_\Omega(\Phi)) \times \text{QF}_\Sigma$ is a set of quadruples $(\Sigma, \Delta, \Gamma, qf)$ where:

- Σ, Δ and Γ have the same meaning as in Def. 3.2.7.
- $qf \in \text{QF}_\Sigma$ is the qualified formula to be checked

Again, we write $\Delta, \Gamma \rhd_\Sigma qf$ instead of $(\Sigma, \Delta, \Gamma, qf) \in \rhd$ and read "under sort predicate assumption Δ and variable assumption Γ the qualified formula qf is well-formed with respect to the polymorphic signature Σ". $\Delta, \Gamma \rhd_\Sigma qf$ holds iff there is a finite proof tree using the rule below that ends with $\Delta, \Gamma \rhd_\Sigma qf$. A proof tree ending with $\Delta, \Gamma \rhd_\Sigma qf$ is called a *sort derivation for qf under* Δ *and* Γ *with respect to* Σ. A particular finite proof tree ending with $\Delta, \Gamma \rhd_\Sigma qf$ is denoted by $\Delta, \Gamma \bigtriangledown_\Sigma qf$.

$$\frac{\Delta, \Gamma \rhd_\Sigma f}{\Delta, \Gamma \rhd_\Sigma \forall \alpha_1, \ldots, \alpha_n.\ A_1, \ldots, A_m \Rightarrow f} \ (\textit{qualification}) \left\{ \ \{A_1, \ldots, A_m\} \vdash_{SA} \Delta \right.$$

□

The sort predicate context of qualified formulae is used to explicitly constrain sort variables by sort predicates. The given context, however, may only be stronger, i.e. more restrictive, than the sort predicate assumption of the derivation. This is achieved by the side condition of the rule (*qualification*).

Definition 3.2.12 *Well-formedness of a qualified formula*

A qualified formula $qf \in \text{QF}_\Sigma$ is called *well-formed with respect to a signature* Σ iff there exists a sort predicate assumption Δ and a variable assumption Γ such that $\Delta, \Gamma \rhd_\Sigma qf$. □

Note that in the definition above we do not have an additional restriction for the sort predicate assumption Δ. The explicitly given sort predicate context is used to restrict the assumption Δ.

Example 3.2.5 *Well-formedness of a qualified formula*

We show that the qualified formula $\forall\alpha.\ \texttt{EQ}(\alpha) \Rightarrow \forall \texttt{x}{:}\alpha.\ \neg\,\texttt{isin(pair x empty)}$ from Ex. 3.2.3 is well-formed by giving a proof tree for $\texttt{EQ}(\alpha), \emptyset \rhd_\Sigma \forall\alpha.\ \texttt{EQ}(\alpha) \Rightarrow \forall \texttt{x}{:}\alpha.\ \neg\,\texttt{isin(pair x empty)}$. Fig. 3.3 shows the proof tree, where D is the proof tree from Ex. 3.2.4 but without the application of the last proof step ($univ_u$).

$$\dfrac{\dfrac{D}{\texttt{EQ}(\alpha), \emptyset \rhd_\Sigma \forall \texttt{x}{:}\alpha.\ \neg\,\texttt{isin(pair x empty)}}\ (univ_s)}{\texttt{EQ}(\alpha), \emptyset \rhd_\Sigma \forall\alpha.\ \texttt{EQ}(\alpha) \Rightarrow \forall \texttt{x}{:}\alpha.\ \neg\,\texttt{isin(pair x empty)}}\ (\textit{qualification})\quad \left\{\{\mathrm{EQ}(\alpha)\} \vdash_{SA} \mathrm{EQ}(\alpha)\right.$$

Figure 3.3: Sort derivation for $\forall\alpha.\ \texttt{EQ}(\alpha) \Rightarrow \forall \texttt{x}{:}\alpha.\ \neg\,\texttt{isin(pair x empty)}$

□

As we have already noted, there is an important difference between non-qualified formulae and qualified formulae with empty sort predicate context. A sort derivation proves the well-formedness of a non-qualified formula if the ground atomic sort formulae of the sort predicate assumption are entailed by the empty assumption. For qualified formulae with empty sort predicate context, however, the sort predicate assumption must be completely entailed by this empty context. We have already discussed this very rigid concept of well-formedness. In our approach it can be achieved by explicitly qualifying a formula by the empty sort predicate context.

Example 3.2.6 *Empty sort predicate context*[6]

The qualified formula $\emptyset \Rightarrow \forall \texttt{x}.\ \neg\,\texttt{isin(pair x empty)}$ is only well-formed if there exists a ground sort term τ such that $\emptyset \vdash_{SA} \texttt{EQ}(\tau)$. The sort derivation below proves the well-formedness with respect to the signature from Ex. 3.2.1. D' is the proof tree obtained by replacing sort variable α in the derivation tree displayed in Fig. 3.2 by sort `Bool`.

$$\dfrac{D'}{\texttt{EQ(Bool)}, \emptyset \rhd_\Sigma \emptyset \Rightarrow \forall \texttt{x}{:}\alpha.\ \neg\,\texttt{isin(pair x empty)}}\ (\textit{qualification})\quad \left\{\emptyset \vdash_{SA} \texttt{EQ(Bool)}\right.$$

□

We now unify the concept of well-formed formulae and well-formed qualified formulae by defining the set of all well-formed Σ-formulae.

Definition 3.2.13 *The set of well-formed Σ-formulae $FORM_\Sigma(\Gamma)$*

Let Σ and Γ be defined as in Def. 3.2.7.

$$\begin{aligned}\mathrm{FORM}_\Sigma(\Gamma) \quad := \quad & \{f \in \mathrm{PF}_\Sigma(\Phi) \mid \exists \Delta \in \mathrm{AF}_\Omega(\Phi).\ \Delta, \Gamma \rhd_\Sigma f \text{ and } \emptyset \vdash_{SA} \Delta \cap \mathrm{AF}_\Omega(\emptyset)\} \cup \\ & \{qf \in \mathrm{QF}_\Sigma \mid \exists \Delta \in \mathrm{AF}_\Omega(\Phi).\ \Delta, \Gamma \rhd_\Sigma qf\}\end{aligned}$$

□

[6] We explicitly denote the empty sort predicate context by $\emptyset$.

Definition 3.2.14 *The set of closed well-formed Σ-formulae SEN_Σ*

Let Σ be a polymorphic signature.

$$\mathrm{SEN}_\Sigma := \mathrm{FORM}_\Sigma(\emptyset)$$

□

As usual, a polymorphic specification consists of a signature Σ and a set of closed well-formed Σ formulae, containing both normal formulae and qualified formulae.

Definition 3.2.15 *Polymorphic specification*

A tuple (Σ, A) is called a *polymorphic specification* iff

- $\Sigma \in \mathrm{PSIG}$ is a polymorphic signature
- $A \subset \mathrm{SEN}_\Sigma$ is a set of closed well-formed Σ-formulae

The elements of A are called the *axioms* of a polymorphic specification. □

We close the syntactic definition of our polymorphic specification language by a brief example specification. To simplify the example we use the bi-implication $\Leftrightarrow$ as an additional logical connective. For the first axiom we have already proved the well-formedness. Also the second axiom can be proved to be well-formed.

Example 3.2.7 *Polymorphic specification*

$$\begin{array}{l}
((((\{\{\mathtt{List}\}_1,\ \{\times\}_2\},\ \{\{\mathtt{EQ}\}_1\}),\\
\quad \{\mathtt{EQ}(\alpha) \Rightarrow \mathtt{EQ}(\mathtt{List}(\alpha))\}),\\
\quad \{\{\mathtt{isin}\}_{\Pi\alpha.EQ(\alpha)\Rightarrow\times(\alpha,List(\alpha))},\ \{\mathtt{eq}\}_{\Pi\alpha.EQ(\alpha)\Rightarrow\times(\alpha,\alpha)}\},\\
\quad \{\{\mathtt{empty}\}_{\Pi\alpha.List(\alpha)},\ \{\mathtt{append}\}_{\Pi\alpha.\alpha\to List(\alpha)\to List(\alpha)},\ \{\mathtt{pair}\}_{\Pi\alpha,\beta.\alpha\to\beta\to\times(\alpha,\beta)}\}),\\
\quad \{\forall \mathtt{x}.\ \neg\, \mathtt{isin(pair\ x\ empty)},\\
\quad\ \forall \mathtt{x}.\forall \mathtt{y}.\forall \mathtt{l}.\ \mathtt{(isin(pair\ x\ (append\ y\ l)))} \Leftrightarrow\\
\qquad\qquad \mathtt{(eq(pair\ x\ y)} \vee \mathtt{\ isin(pair\ x\ l))}\})
\end{array}$$

□

Chapter 4

A Semantic Framework for Polymorphic Specifications

In the last chapter we defined the language of polymorphic specifications. In this chapter we assign a meaning to those specifications. Polymorphic specifications are in general not executable. Hence, we cannot define an operational semantics but only a declarative one. We use algebraic structures for the interpretation of polymorphic specifications. As usual, the models of a polymorphic specification are those algebraic structures satisfying all axioms of the specification.

The definition of our models differ markedly from conventional approaches. First of all, we have to deal with two-level specifications. Thus, we require models for sort specifications as well as polymorphic specifications. The model of sort specifications can be defined in a conventional way. We interpret sort specifications by conventional algebras, called *sort algebras*.

For the interpretation of polymorphic specifications, however, we need two-level algebraic structures. We call these structures *polymorphic algebras*. Polymorphic algebras consist of two levels because they contain a sort algebra, plus an interpretation for functions and predicates. This sort algebra must be a model of the sort specification of the polymorphic specification.

In contrast to many-sorted signatures, polymorphic signatures may contain identifiers for polymorphic functions and predicates. Thus, we need an interpretation for polymorphic sort terms. A polymorphic algebra assigns to each polymorphic identifier a value in the interpretation of its polymorphic sort term.

The main difference with conventional approaches, however, arises because polymorphic identifiers are instantiated implicitly at application and because variables may be declared without sort annotation. The problem is now to define models for partially

unsorted specifications. Obviously, we need this information to interpret a polymorphic specification. Hence, we cannot directly define the interpretation of polymorphic specifications.

The particular instances of polymorphic identifiers as well as the sorts of variables declared without sort annotation are inferred with the help of the sort inference calculus. A sort derivation for a polymorphic specification contains all information needed to interpret this specification. Thus, we give an interpretation for sort derivations.

Unfortunately, for some polymorphic specifications there exists not only one sort derivation but sometimes even infinitely many. Which one should be used to interpret these specifications? The most direct and restrictive approach is to consider all possible sort derivations of a polymorphic specification. Therefore, we start with a model concept where each particular sort derivation of a specification has to be satisfied by a polymorphic algebra.

This definition, at first glance natural, has a serious drawback: if we want to reason about properties of a polymorphic specification we must consider all possible sort derivations. In practice, however, we cannot handle infinitely many sort derivations in a proof system. Hence, we define a further notion of model based on single sort derivations. We prove both notions of model to be equivalent.

Since our main focus of attention lies on the polymorphism concept of our language, we will not deal with domain theory in this thesis. Polymorphism is independent of the underlying domain theory. We can use ordinary sets as well as complete partial orders as carriers, and we can use the full function space as well as continuous functions to interpret our function space constructor.

Thus, this chapter only describes a semantic framework for a polymorphic specification language. For a concrete specification language the framework must be instantiated by a suitable domain theory. In the following we will use the word 'domain' for the carrier sets of our algebras without defining it in more detail.

Section 4.1 introduces models for sort specifications. We define sort algebras and give an interpretation for sort terms and sort clauses. If all axioms of a sort specification are valid in a sort algebra, it is defined to be a model of the specification. In addition we define a semantic consequence relation on sort predicates. We prove the syntactic entailment relation defined in Section 3.1 to be sound and complete with respect to the consequence relation. Based on dependent products, we finally give an interpretation for polymorphic sort terms.

Section 4.2 presents models for polymorphic specifications. We start by defining polymorphic algebras. Formulae at the object level may contain unsorted variables. Furthermore, polymorphic identifiers are applied without explicit instantiation. Because of this

omitted information, we cannot define the meaning of a formula directly. Instead, we give an interpretation for sort derivations containing the omitted information. We show that the interpretation function is well-defined. Based on this interpretation, we define a notion of satisfaction for formulae by considering all sort derivations of a formula. A polymorphic algebra is defined to be a model of a specification if all axioms are satisfied in the algebra.

Section 4.3 investigates another conception of model based on a single principal sort derivation. We will discuss the proceeding more precisely in the introductory part of Section 4.3.

4.1 The Semantics of Sort Specifications

A sort specification consists of a signature and a set of axioms that are restricted to Horn clauses. Therefore, the semantics is also defined in a conventional way by defining the concept of a sort algebra. Every sort algebra satisfying the axioms is a model of the sort specification.

Definition 4.1.1 *Sort algebra*

Let $\Omega = (SC, SP)$ be a sort signature. A triple $\mathcal{SA} = (\mathcal{U}, \mathcal{DC}, \mathcal{DP})$ is called an Ω *sort algebra* iff the following properties hold:

- $\mathcal{U}$ is a set of domains and is called the *domain universe*
- $\mathcal{DC}$ is a mapping with the following properties:
  - $\mathcal{DC}$ assigns each $sc \in SC_0$ an element in $\mathcal{U}$
  - $\mathcal{DC}$ assigns each $sc \in SC_n$, $n \geq 1$, a totally defined function from $\mathcal{U}^n$ to $\mathcal{U}$, called a *domain constructor*.
- $\mathcal{DP}$ is a mapping with the following properties: $\mathcal{DP}$ assigns each $sp \in SP_n$, $n \geq 1$ a totally defined predicate from $\mathcal{U}^n$ to the truth values $\{true, false\}$, called a *domain predicate*.
- $\mathcal{DC}(\rightarrow)$ is the function space constructor, i.e. $\mathcal{DC}(\rightarrow)(d_1, d_2)$ is the set of all totally defined functions from the domain d_1 to the domain d_2.

In order to simplify notations we write $sc^{\mathcal{SA}}$ and $sp^{\mathcal{SA}}$ instead of $\mathcal{DC}(sc)$ and $\mathcal{DP}(sp)$, unless ambiguous. □

Instead of the full function space, $\rightarrow$ can also be interpreted by a subset of this, i.e. by the continuous functions. We only require this subset to be closed with respect

to application and λ-abstraction. Furthermore, the function space may only contain totally defined functions, i.e. if $f \in \rightarrow^{\mathcal{SA}}(d_1, d_2)$ then $f(x) \in d_2$ for each value $x \in d_1$. Nevertheless, our framework allows modelling partial functions by adding an undefined value to each domain. This is the usual way to model partiality in programming and specification languages.

Now we define the meaning of a sort term relative to a sort algebra and a sort variable assignment. The sort variable assignment assigns domains to those sort variables which may occur in the sort term.

Definition 4.1.2 *Sort variable assignment*

Let $\Omega{=}(SC, SP)$ be a sort signature and $\mathcal{SA} = (\mathcal{U}, \mathcal{DC}, \mathcal{DP})$ be an Ω sort algebra. A function $\nu : \chi \rightarrow \mathcal{U}$ is called a *sort variable assignment*. We write $\nu.[\alpha \mapsto d]$ to update a sort variable assignment at point α. More formally,

$$\nu.[\alpha \mapsto d](\beta) := \begin{cases} d & \text{if } \alpha = \beta \\ \nu(\beta) & \text{otherwise} \end{cases}$$

□

Definition 4.1.3 *Interpretation of monomorphic sort terms* $T_\Omega(\Phi)$

Let $\Omega{=}(SC, SP)$ be a sort signature and $\mathcal{SA} = (\mathcal{U}, \mathcal{DC}, \mathcal{DP})$ be an Ω sort algebra. Let $\nu : \chi \rightarrow \mathcal{U}$ be a sort variable assignment. The interpretation of a monomorphic sort term $\tau \in \mathrm{T}_\Omega(\Phi)$ in $\mathcal{SA}$ with respect to ν, written $\mathcal{SA}[\![\tau]\!]_\nu$, yields an element in $\mathcal{U}$ and is defined on the structure of τ as follows:

$$\begin{aligned}
&\mathcal{SA}[\![c]\!]_\nu := c^{\mathcal{SA}} && \text{if } c \in SC_0 \\
&\mathcal{SA}[\![\alpha]\!]_\nu := \nu(\alpha) && \text{if } \alpha \in \chi \\
&\mathcal{SA}[\![sc(\tau_1, \ldots, \tau_n)]\!]_\nu := sc^{\mathcal{SA}}(\mathcal{SA}[\![\tau_1]\!]_\nu, \ \ldots \ , \mathcal{SA}[\![\tau_n]\!]_\nu) && n \geq 1
\end{aligned}$$

□

The interpretation of monomorphic sort terms is well-defined. Because each domain constructor is totally defined, the interpretation indeed yields an element in $\mathcal{U}$. If τ is a ground sort term, i.e. $\tau \in \mathrm{T}_\Omega(\emptyset)$, the interpretation does not depend on the actual sort variable assignment. Therefore, we will use the shorter notation $\tau^{\mathcal{SA}}$ to denote the interpretation of τ in the sort algebra $\mathcal{SA}$.

Based on the interpretation of sort terms we can now assign a truth value to sort clauses.

Definition 4.1.4 *Interpretation of sort clauses* $C_\Omega(\Phi)$

Let $\Omega = (SC, SP)$ be a sort signature and $\mathcal{SA} = (\mathcal{U}, \mathcal{DC}, \mathcal{DP})$ be an Ω sort algebra. Let $\nu : \chi \to \mathcal{U}$ be a sort variable assignment. The interpretation of a sort clause $C \in \mathrm{C}_\Omega(\Phi)$ in $\mathcal{SA}$ with respect to ν, written $\mathcal{SA}[\![C]\!]_\nu$, yields a truth value from $\{true, false\}$ and is defined on the structure of C as follows:

$$\mathcal{SA}[\![sp(\tau_1, \ldots, \tau_n)]\!]_\nu := sp^{\mathcal{SA}}(\mathcal{SA}[\![\tau_1]\!]_\nu, \ \ldots \ ,\mathcal{SA}[\![\tau_n]\!]_\nu) \qquad n \geq 1$$

$$\mathcal{SA}[\![B_1, \ \ldots \ ,B_n \Rightarrow H]\!]_\nu := \begin{cases} false & \text{if } \mathcal{SA}[\![H]\!]_\nu = false \text{ and } \mathcal{SA}[\![B_i]\!]_\nu = true,\ 1 \leq i \leq n \\ true & \text{otherwise} \end{cases}$$

□

In contrast to the usual approaches, our sort specification consists not only of a sort signature, but also of a set of sort axioms specifying the properties of the sort constructors. Therefore, we are only interested in those algebras that satisfy the sort clauses given as axioms.

Definition 4.1.5 *Satisfaction, Validity*

Let $\Omega = (SC, SP)$ be a sort signature and $\mathcal{SA} = (\mathcal{U}, \mathcal{DC}, \mathcal{DP})$ be an Ω sort algebra. Let $\nu : \chi \to \mathcal{U}$ be a sort variable assignment.

- ν *satisfies* a sort clause $C \in \mathrm{C}_\Omega(\Phi)$ in $\mathcal{SA}$, written

$$\models_{\mathcal{SA},\nu} C \qquad \text{iff} \qquad \mathcal{SA}[\![C]\!]_\nu = true$$

- C is called *valid* in $\mathcal{SA}$, written

$$\models_{\mathcal{SA}} C \qquad \text{iff} \qquad \models_{\mathcal{SA},\nu} C \quad \text{for each } \nu : \chi \to \mathcal{U}$$

- Both definitions can be naturally extended to sets of sort clauses Θ:

$$\models_{\mathcal{SA},\nu} \Theta \qquad \text{iff} \qquad \models_{\mathcal{SA},\nu} C \quad \text{for each } C \in \Theta$$

$$\models_{\mathcal{SA}} \Theta \qquad \text{iff} \qquad \models_{\mathcal{SA}} C \quad \text{for each } C \in \Theta$$

□

Definition 4.1.6 *Model*

Let $S = (\Omega, SA)$ be a sort specification. An Ω sort algebra $\mathcal{SA}$ is called a *model* of S iff

$$\models_{\mathcal{SA}} SA$$

□

Note that the models of a sort specification are not restricted to term generated sort algebras. The universe of a model may contain domains not generated by a ground sort term.

We now define a logical consequence relation on sort clauses. This relation forms the counterpart to the syntactic entailment relation.

Definition 4.1.7 *Logical Consequence*

Let $\Omega = (SC, SP)$ be a sort signature and $\mathcal{SA} = (\mathcal{U}, \mathcal{DC}, \mathcal{DP})$ be an Ω sort algebra. Let $\nu : \chi \to \mathcal{U}$ be a sort variable assignment. Let $\Delta \subseteq \mathrm{AF}_\Omega(\Phi)$ be a set of atomic sort formulae and $A \in \mathrm{AF}_\Omega(\Phi)$ be an atomic sort formula.

- A is called a *logical consequence of* Δ *in* $\mathcal{SA}$ *under* ν, written

$$\Delta \models_{\mathcal{SA},\nu} A \qquad \text{iff} \qquad \text{if } \models_{\mathcal{SA},\nu} \Delta \text{ then } \models_{\mathcal{SA},\nu} A$$

- A is called a *logical consequence of* Δ *in* $\mathcal{SA}$, written

$$\Delta \models_{\mathcal{SA}} A \qquad \text{iff} \qquad \Delta \models_{\mathcal{SA},\nu} A \qquad \text{for all } \nu : \chi \to \mathcal{U}$$

- Both definitions can be naturally extended to sets of atomic sort formulae Δ_1 and Δ_2:

$$\Delta_1 \models_{\mathcal{SA},\nu} \Delta_2 \qquad \text{iff} \qquad \Delta_1 \models_{\mathcal{SA},\nu} A \qquad \text{for each } A \in \Delta_2$$

$$\Delta_1 \models_{\mathcal{SA}} \Delta_2 \qquad \text{iff} \qquad \Delta_1 \models_{\mathcal{SA}} A \qquad \text{for each } A \in \Delta_2$$

□

We must show that the semantic consequence relation $\models$ coincides with the syntactic entailment relation $\vdash$. This is called the soundness and completeness property of the relation $\vdash$. The proof of this property requires two auxiliary propositions. In both cases the central idea is the same. We can apply a substitution to a syntactic structure and afterwards interpret it with respect to a variable assignment. But we can also update this assignment making it reflect the substitution and interpret the structure immediately with respect to the updated variable assignment. The result of both interpretations is the same. This property, e.g., holds for the terms of the λ-calculus, where it is necessary to prove the soundness of the β-reduction. The first of the following propositions establishes this property for sort terms while the second does so for sort clauses.

Proposition 4.1.1 *Interpretation equivalent sort terms*

Let $\tau \in \mathrm{T}_\Omega(\chi)$ be a monomorphic sort expression, and let σ be a sort variable substitution. Let ν and ν^σ be sort variable assignments.

$$\text{If for each } \alpha \in \chi : \nu^\sigma(\alpha) = \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu \text{ then } \mathcal{SA}[\![\tau]\!]_{\nu^\sigma} = \mathcal{SA}[\![\sigma(\tau)]\!]_\nu.$$

□

Proposition 4.1.2 *Interpretation equivalent sort clauses*

Let $C \in \mathrm{C}_\Omega(\chi)$ be a sort clause, and let Θ be a set of sort clauses. Let σ be a sort variable substitution. Let ν and ν^σ be sort variable assignments.

$$\text{If for each } \alpha \in \chi : \nu^\sigma(\alpha) = \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu \text{ then}$$

$$\text{i)} \quad \models_{\mathcal{SA},\nu^\sigma} C \qquad \text{iff} \qquad \models_{\mathcal{SA},\nu} \sigma(C)$$

$$\text{ii)} \quad \models_{\mathcal{SA},\nu^\sigma} \Theta \qquad \text{iff} \qquad \models_{\mathcal{SA},\nu} \sigma(\Theta)$$

□

Proposition 4.1.3 *Soundness and completeness of entailment relation* $\vdash$

Let $S = (\Omega, SA)$ be a sort specification. Let $\Delta \subseteq \mathrm{AF}_\Omega(\Phi)$ be a set of atomic sort formulae and $A \in \mathrm{AF}_\Omega(\Phi)$ be an atomic sort formula.

1. The entailment relation $\vdash$ is *sound* with respect to the logical consequence relation $\models$, i.e.

$$\text{if} \quad \Delta \vdash_{SA} A \quad \text{then} \quad \Delta \models_{\mathcal{SA}} A \text{ for each model } \mathcal{SA} \text{ of } S$$

2. The entailment relation $\vdash$ is *complete* with respect to the logical consequence relation $\models$, i.e.

$$\text{if} \quad \Delta \models_{\mathcal{SA}} A \text{ for each model } \mathcal{SA} \text{ of } S \quad \text{then} \quad \Delta \vdash_{SA} A$$

3. Both propositions can be extended to sets of atomic sort formulae Δ_1 and Δ_2, i.e.

$$\text{if} \quad \Delta_1 \vdash_{SA} \Delta_2 \quad \text{then} \quad \Delta_1 \models_{\mathcal{SA}} \Delta_2 \text{ for each model } \mathcal{SA} \text{ of } S$$

$$\text{if} \quad \Delta_1 \models_{\mathcal{SA}} \Delta_2 \text{ for each model } \mathcal{SA} \text{ of } S \quad \text{then} \quad \Delta_1 \vdash_{SA} \Delta_2$$

Note that in the completeness proposition it is important that the premise $\Delta \models_{\mathcal{SA}} A$ must be valid for each model. The quantification cannot be moved to the outside of the proposition. In particular, the proposition *for each model* $\mathcal{SA}$ *of* S*: if* $\Delta \models_{\mathcal{SA}} A$ *then* $\Delta \vdash_{SA} A$ does not hold. In the soundness proposition, in contrast, both possibilities are equivalent. □

The next proposition is a variation of Prop. 4.1.2 using the same central idea. It is needed to prove later propositions.

Proposition 4.1.4 *Satisfaction implying atomic sort formulae*

Let $S = (\Omega, SA)$ be a sort specification, and let $\mathcal{SA}$ be a model of S. Let Δ_1, $\Delta_2 \subseteq \mathrm{AF}_\Omega(\chi)$ be two sets of atomic sort formulae. Let σ be a sort variable substitution such that $\Delta_1 \vdash_{SA} \sigma(\Delta_2)$. Let ν and ν^σ be sort variable assignments such that $\nu^\sigma(\alpha) = \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu$ for each $\alpha \in \chi$.

$$\text{If} \quad \models_{\mathcal{SA},\nu} \Delta_1 \quad \text{then} \quad \models_{\mathcal{SA},\nu^\sigma} \Delta_2$$

□

In polymorphic signatures we are allowed to declare polymorphic functions and predicates. For this purpose we introduced polymorphic sort terms. Before we can define polymorphic algebras, we must give an interpretation for polymorphic sort terms.

Definition 4.1.8 *Interpretation of polymorphic sort terms* T^Π_Ω

Let $\Omega = (SC, SP)$ be a sort signature and $\mathcal{SA} = (\mathcal{U}, \mathcal{DC}, \mathcal{DP})$ be an Ω sort algebra. The interpretation of a polymorphic sort term $\sigma \in \mathrm{T}^\Pi_\Omega$ in $\mathcal{SA}$, written $\mathcal{SA}[\![\sigma]\!]$ is defined in the following way:

$$\mathcal{SA}[\![\Pi\alpha_1, \ldots, \alpha_n.A_1, \ldots, A_m \Rightarrow \tau]\!] := \prod_{(d_1,\ldots,d_n)|d_i \in \mathcal{U} \wedge \forall j.1 \le j \le m \Rightarrow \models_{\mathcal{SA},\nu} A_j} \mathcal{SA}[\![\tau]\!]_\nu$$

$$\text{where } \nu := [\alpha_1 \mapsto d_1, \ldots, \alpha_n \mapsto d_n]$$

□

The set $\prod_{x \in S} f(x)$ is called the *dependent product determined by* f. This set consists of functions π such that $\pi(x) \in f(x)$ for each $x \in S$. In our special case of polymorphic sort terms, S is the set of domain tuples such that the qualifying sort predicates A_j are satisfied. The function f is the interpretation of the monomorphic sort term τ, and the actual parameter x is the sort variable assignment ν. To be more precise, the interpretation of a polymorphic sort term $\Pi\alpha_1, \ldots, \alpha_n.A_1, \ldots, A_m \Rightarrow \tau$ is the set of all functions that take a domain tuple $(d_1, \ldots, d_n)$ satisfying the qualifying atomic sort formulae A_j and yield a value in a domain of our universe $\mathcal{U}$. This result domain must be the interpretation of the sort term τ where the domains d_i are assigned to the

corresponding sort variables α_i. The elements of the interpretation of a polymorphic sort term are called *polymorphic functions*. To avoid confusion with the application of normal functions, the application of a polymorphic function is called *instantiation*. We used the same notation for the syntactic sort variable substitution of the body of a polymorphic sort term. In Section 4.2 we will relate the syntactic notation with the semantic one.

Our interpretation of polymorphic sort terms does not have the property known as *parametricity*. This is a formulation of the idea that different instantiations of a polymorphic function are related to one another in a uniform way. This property holds in functional languages providing pure Hindley/Milner polymorphism. In such a language, by construction, each polymorphic identifier defined by a let or similar construct has exactly one body. Thus, at each instantiation the polymorphic function behaves uniformly, because the sort of the parameter cannot be dynamically tested in the body.

The parametricity property, however, does not hold for a functional language providing type classes. The polymorphic functions belonging to a class are instantiated by different functions for each type of the class. These functions need not be related in some way. Thus, the type class system of Haskell can be used for modelling ad-hoc polymorphism. This was indeed the main reason for the development of type classes (see [WB89]). Note that the pure polymorphic functions of Haskell again behave uniformly as in ML. In our framework neither qualified nor pure polymorphic functions must be parametric. Our interpretation of polymorphic sort terms allows a different behaviour for each instantiation of a polymorphic function. A uniform behaviour, however, can easily be achieved by an appropriate axiomatization. Therefore, again, our framework is very liberal and leaves a large degree of freedom for specifications to the user. We will pick up this subject again in Section 6.1.

Note that the interpretation of a polymorphic sort term need not be an element of our universe $\mathcal{U}$. Implicitly we are working with two universes. The first of these, $\mathcal{U}$, is used for interpreting monomorphic sort terms, while the second contains sets that can be the interpretation of polymorphic sort terms. The definition of the second universe is based on the first one. This is the reason for calling our polymorphism concept *predicative*. Because we do not provide any higher order polymorphic functions we do not need more universes based on each other to describe the semantics. Therefore, our polymorphism concept is a very simple predicative one, called *shallow polymorphism*. A definition of a universe is called *impredicative*, in contrast, if the definition is based on the universe itself. Systems providing full higher order polymorphic functions need an impredicative definition of polymorphic sorts. For a general overview on the semantics of polymorphic functions see [Gun92].

Let us illustrate the interpretation of polymorphic sort terms on two examples. The first one shows the interpretation of a pure polymorphic sort term, while the second one

illustrates the more complex interpretation of a polymorphic sort term with qualifying sort predicates.

Example 4.1.1 *Interpretation of polymorphic sort terms*

In Ex. 3.2.1 we gave a signature containing a pure polymorphic pairing function `pair`: $\Pi\alpha,\beta.\alpha \to \beta \to \times(\alpha,\beta)$. Let the sort algebra $\mathcal{SA}$ be a model of the sort specification S used in this signature. Then `pair` is a polymorphic function[1] that can be applied to an arbitrary tuple of domains from the universe $\mathcal{U}$ of $\mathcal{SA}$. If we apply, e.g., `pair` to ($\texttt{Nat}^{\mathcal{SA}}$, $\texttt{(List(Nat))}^{\mathcal{SA}}$), we get an element of the following domain:

$$\mathcal{SA}[\![\alpha \to \beta \to \times(\alpha,\beta)]\!]_{[\alpha\mapsto \texttt{Nat}^{\mathcal{SA}},\beta\mapsto(\texttt{List(Nat)})^{\mathcal{SA}}]} \overset{\text{Prop. 4.1.1}}{=} (\texttt{Nat} \to \texttt{List(Nat)} \to \times(\texttt{Nat},\texttt{List(Nat)}))^{\mathcal{SA}}$$

That means that the result of the instantiation is a higher order monomorphic function that can be applied to elements of domain $\texttt{Nat}^{\mathcal{SA}}$. If we apply `pair` to ($\texttt{Int}^{\mathcal{SA}}$, $\texttt{Bool}^{\mathcal{SA}}$), we get a function of the domain $(\texttt{Int} \to \texttt{Bool} \to \times(\texttt{Int},\texttt{Bool}))^{\mathcal{SA}}$.

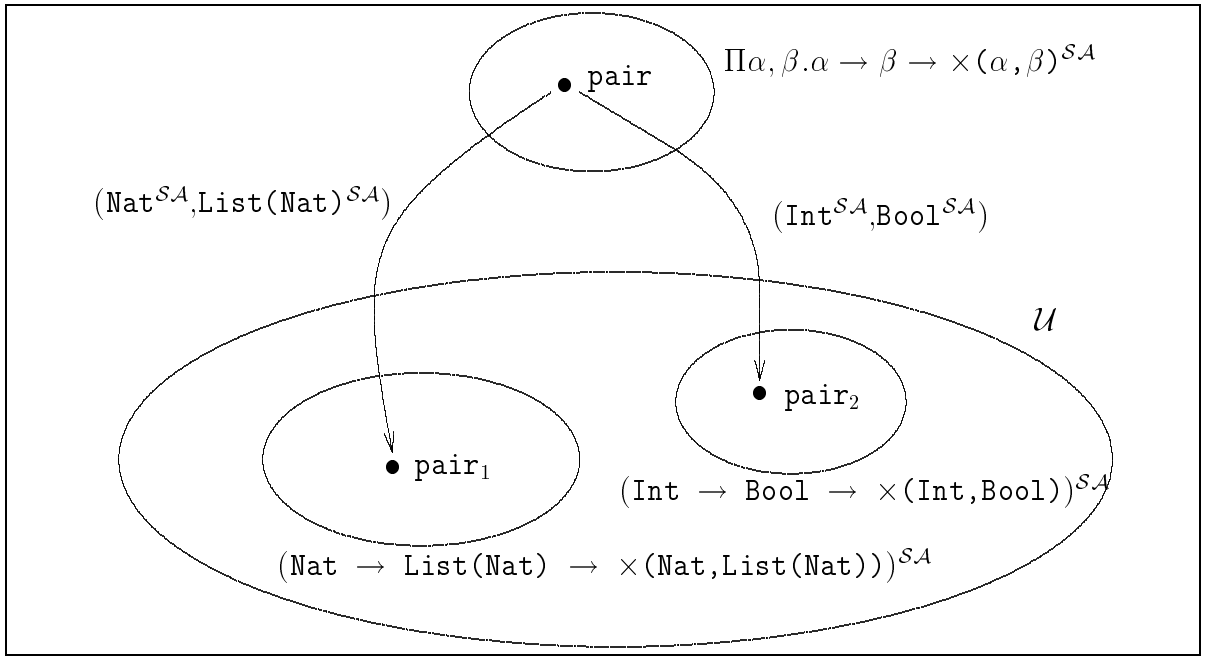

Figure 4.1: Interpretation of $\Pi\alpha,\beta.\alpha \to \beta \to \times(\alpha,\beta)$

Fig. 4.1 illustrates the example. The arrows stand for the respective instantiation of the polymorphic function `pair`. Each instantiation yields different functions $\texttt{pair}_1$ and $\texttt{pair}_2$, respectively, each being an element of a different domain. By default, these two functions do not have common properties, i.e. are not related in some way. Furthermore,

[1] The identifier `pair` is used for the semantic value, too.

the picture shows that the interpretation of the polymorphic sort term is not an element of the universe $\mathcal{U}$.

For the next example we assume a function $\texttt{eq:}\Pi\alpha.\texttt{EQ}(\alpha) \Rightarrow \alpha \to \alpha \to \texttt{Bool}$. Let S and $\mathcal{SA}$ be defined as in the first example. The polymorphic function eq can be applied to sorts satisfying the qualifying sort predicate $\texttt{EQ}(\alpha)$. Because the sort specification S contains axiom EQ(Bool) and $\mathcal{SA}$ is a model of S, EQ(Bool) is valid in $\mathcal{SA}$, i.e.:

$$\models_{\mathcal{SA}} \texttt{EQ(Bool)} \overset{\text{Prop. 4.1.2}}{\Leftrightarrow} \models_{\mathcal{SA},[\alpha \mapsto \texttt{Bool}^{\mathcal{SA}}]} \texttt{EQ}(\alpha)$$

Thus, eq can be instantiated with $\texttt{Bool}^{\mathcal{SA}}$. The result is an element of the following domain:

$$\mathcal{SA}[\![\alpha \to \alpha \to \texttt{Bool}]\!]_{[\alpha \mapsto \texttt{Bool}^{\mathcal{SA}}]} \overset{\text{Prop. 4.1.1}}{\Leftrightarrow} (\texttt{Bool} \to \texttt{Bool} \to \texttt{Bool})^{\mathcal{SA}}$$

The qualifying sort predicate is also satisfied if we assign $\texttt{Nat}^{\mathcal{SA}}$ to α. Thus, eq can also be applied to $\texttt{Nat}^{\mathcal{SA}}$. This instantiation yields an element in $(\texttt{Nat} \to \texttt{Nat} \to \texttt{Nat})^{\mathcal{SA}}$. Fig. 4.2 illustrates both instantiations. Note that eq cannot be applied to e.g. $\texttt{Int}^{\mathcal{SA}}$ if we choose a model $\mathcal{SA}$ of S where $\texttt{EQ}^{\mathcal{SA}}(\texttt{Int}^{\mathcal{SA}})$ does not hold.

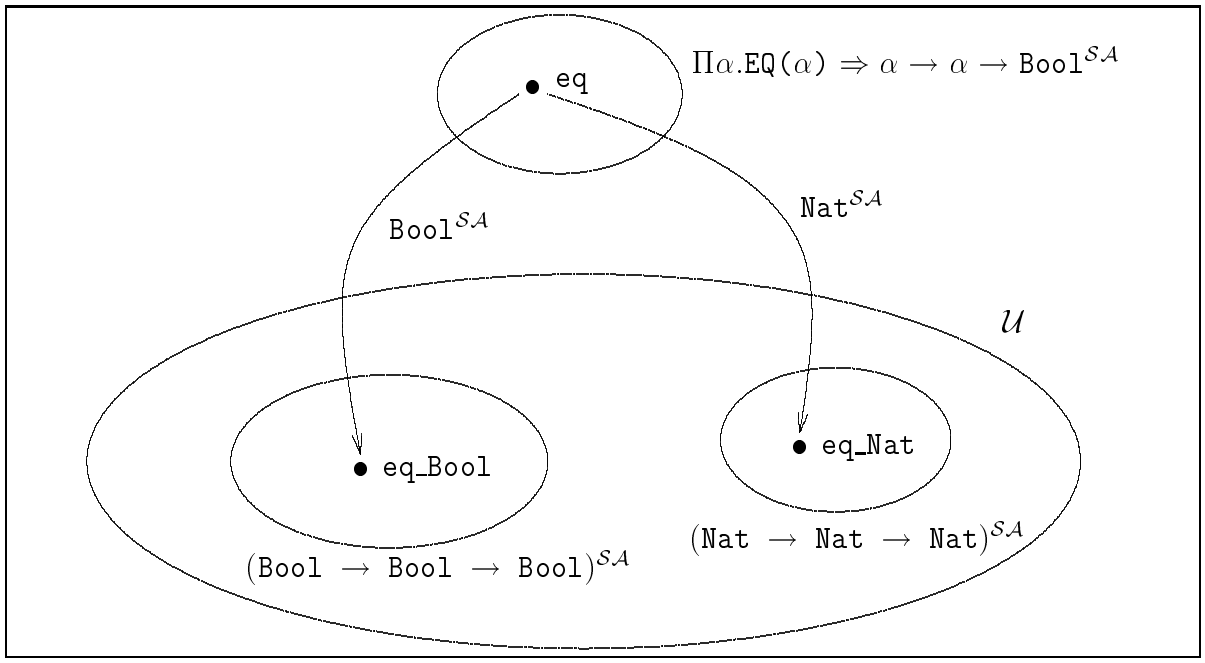

Figure 4.2: Interpretation of $\Pi\alpha.\texttt{EQ}(\alpha) \Rightarrow \alpha \to \alpha \to \texttt{Bool}$

□

The interpretation of polymorphic sort terms does not depend on a certain sort variable assignment because polymorphic sort terms are always closed with respect to sort variables. Therefore, we will use the shorter notation $\sigma^{\mathcal{SA}}$ to denote the interpretation of a polymorphic sort term σ in the sort algebra $\mathcal{SA}$.

4.2 A Semantics for Polymorphic Specifications

Describing the meaning of terms and formulae is more difficult than describing the meaning of sort terms. Thus, we require some further vocabulary and notation. The problem lies in the incompleteness of terms and formulae. Variables may be declared without sort annotation, and polymorphic identifiers cannot be instantiated explicitly. This omitted information is deduced from the context by the sort inference calculus, which checks the well-formedness of terms and formulae. The semantics of the object level, however, can only be defined with the help of this omitted information. For defining, e.g., the meaning of the formula ∀x. p(x), we must know the sort of the variable x. The semantics can only be defined together with the deduced omitted information. For this reason the meaning of terms and formulae is defined indirectly via a sort derivation. This technique is due to [Mit90]. The derivation tree contains all the omitted information we need to define the meaning in a unique way. Unfortunately, as we saw in Section 3.2, there exists not just one derivation tree for a term, but sometimes even infinitely many. Furthermore, the meaning of a term or formula depends on the chosen derivation tree. In this chapter the problem is solved by taking all possible sort derivations into consideration when defining the models of a polymorphic specification. Besides this problem, the mechanism is similar to the sort level. We define a concept of polymorphic algebras, which are models of a polymorphic specification if all axioms are satisfied.

Definition 4.2.1 *Polymorphic Σ-algebras*

Let $\Sigma = ((\Omega, SA), P, F)$ be a polymorphic signature. A triple $\mathcal{A} = (\mathcal{SA}, \mathcal{P}, \mathcal{F})$ is called a Σ-*algebra* iff the following properties hold:

- $\mathcal{SA} = (\mathcal{U}, \mathcal{DC}, \mathcal{DP})$ is a model of the sort specification (Ω, SA), where $\mapsto^{\mathcal{SA}}$ is the full function space constructor, and $\mathrm{TV}^{\mathcal{SA}}$ is the set of truth values $\{true, false\}$.
- $\mathcal{P}$ is a mapping that assigns each $p \in P_\tau$
  - a truth value in $\{true, false\}$ if $\tau = \epsilon$,
  - a totally defined predicate in $((\tau) \mapsto \mathrm{TV})^{\mathcal{SA}}$ if $\tau \in \mathrm{T}_\Omega(\emptyset)$ is a monomorphic sort term,
  - a polymorphic predicate in $(\Pi\alpha_1, \ldots, \alpha_n.A_1, \ldots, A_m \Rightarrow (\tau') \mapsto \mathrm{TV})^{\mathcal{SA}}$ such that each instantiation yields a totally defined predicate, i.e.

    $$\mathcal{P}(p)[s_1, \ldots, s_n](v) \in \mathrm{TV}^{\mathcal{SA}}$$

    for each $s_1, \ldots, s_n$ with $\models_{\mathcal{SA},\nu} A_i$, $1 \leq i \leq m$, and for each $v \in \mathcal{SA}[\![\tau']\!]_\nu$, where $\nu = [\alpha_1 \mapsto s_1, \ldots, \alpha_n \mapsto s_n]$ if $\tau = \Pi\alpha_1, \ldots, \alpha_n.A_1, \ldots, A_m \Rightarrow \tau' \in \mathrm{T}^{\Pi}_{\Omega}$ is a polymorphic sort term.

- $\mathcal{F}$ is a mapping that assigns each $f \in F_\tau$, $\tau \in \mathrm{T}_\Omega$, a — possibly polymorphic — value in $\tau^{\mathcal{SA}}$.

In order to simplify notations we write $p^{\mathcal{A}}$ and $f^{\mathcal{A}}$ instead of $\mathcal{P}(p)$ and $\mathcal{F}(f)$, unless ambiguous. □

Now we define the meaning of a sort derivation for a term inductively on the structure of the derivation tree with respect to a polymorphic algebra. Recall that a sort derivation for a term e ends with $\Delta, \Gamma \rhd_\Sigma e :: \tau$. Term e may contain free object variables. Thus, the interpretation of a sort derivation depends on a variable assignment, assigning values to object variables. The variable assignment must be sort correct, i.e. it must satisfy the variable assumption Γ. Furthermore, the sort derivation may contain sort variables. Therefore, in addition, the interpretation depends on a sort variable assignment, assigning sorts to sort variables. The sort variable assignment must satisfy the sort predicate assumption Δ. That means a sort derivation can only be interpreted with respect to a variable assignment satisfying the variable assumption of the derivation and a sort variable assignment satisfying the sort predicate assumption of the derivation. The next definition precisely defines the satisfaction of a variable assumption.

Definition 4.2.2 *Variable assignment, satisfaction of a variable assumption*

Let $\mathcal{SA}$ be an Ω sort algebra. Let Γ: $<var> \to \mathrm{T}_\Omega(\Phi)$ be a variable assumption, and let $\nu : \chi \to \mathcal{U}$ be a sort variable assignment. A function $\eta : <var> \to \bigcup_{d \in \mathcal{U}} d$, mapping variables to values, is called a *variable assignment*. We say that η *satisfies* Γ in $\mathcal{SA}$ with respect to ν written

$$\eta \models_{\mathcal{SA},\nu} \Gamma \qquad \text{iff} \qquad \text{for each } x{:}\tau \in \Gamma . \eta(x) \in \mathcal{SA}[\![\tau]\!]_\nu$$

We will use the same notation for updating a variable assignment as for sort variable assignments. □

Proposition 4.2.1 *Sort correct updating of a variable assignment*

Let Γ be a variable assumption, and let ν be a sort variable assignment. Let η be a variable assignment such that $\eta \models_{\mathcal{SA},\nu} \Gamma$.

$$d \in \mathcal{SA}[\![\tau]\!]_\nu \qquad \Rightarrow \qquad \eta.[x \mapsto d] \models_{\mathcal{SA},\nu} \Gamma.x{:}\tau$$

□

Definition 4.2.3 *Meaning of a sort derivation for a pre-term*

Let D be a sort derivation for a pre-term e under sort predicate assumption Δ and variable assumption Γ with respect to a polymorphic signature Σ ending with Δ, Γ

$\rhd_\Sigma e :: \tau$. Let $\mathcal{A} = (\mathcal{SA}, \mathcal{P}, \mathcal{F})$ be a polymorphic Σ-algebra. Let ν be a sort variable assignment and η be a variable assignment such that $\models_{\mathcal{SA},\nu} \Delta$ and $\eta \models_{\mathcal{SA},\nu} \Gamma$. The meaning of a sort derivation D in $\mathcal{A}$ under ν and η is $\mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta}$. The interpretation function $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ maps sort derivations D to elements in $\mathcal{SA}[\![\tau]\!]_\nu$ and is recursively defined on the structure of the derivation tree D in the following way:

Base cases

$$\mathcal{A}\left[\!\left[\frac{}{\Delta, \Gamma.x{:}\tau \rhd_\Sigma x :: \tau}\ (var)\right]\!\right]_{\nu,\eta} := \eta(x)$$

$$\mathcal{A}\left[\!\left[\frac{}{\Delta, \Gamma \rhd_\Sigma c :: \tau}\ (id)\right]\!\right]_{\nu,\eta} := c^{\mathcal{A}} \quad \begin{cases} c \in F_\tau \\ \text{where } \tau \in \mathrm{T}_\Omega(\emptyset) \end{cases}$$

$$\mathcal{A}\left[\!\left[\frac{}{\Delta, \Gamma \rhd_\Sigma c :: \sigma(\tau)}\ (polyid)\right]\!\right]_{\nu,\eta} := c^{\mathcal{A}}[\mathcal{SA}[\![\sigma(\alpha_1)]\!]_\nu, \ldots, \mathcal{SA}[\![\sigma(\alpha_n)]\!]_\nu]$$

$$c \in F_{\Pi\alpha_1,\ldots,\alpha_n.A_1,\ldots,A_m \Rightarrow \tau} \qquad \sigma : \Phi \to \mathrm{T}_\Omega(\Phi)$$
$$\sigma(A_i) \in \Delta, 1 \le i \le m$$

Inductive cases

$$\mathcal{A}\left[\!\left[\frac{\Delta,\ \Gamma\ \bigtriangledown_\Sigma\ e_1 :: \tau_2 \rightarrow\!\!\!\!\!\!\rightarrow \tau_1 \quad \Delta,\ \Gamma\ \bigtriangledown_\Sigma\ e_2 :: \tau_2}{\Delta, \Gamma \rhd_\Sigma e_1 e_2 :: \tau_1}\ (\rightarrow E)\right]\!\right]_{\nu,\eta}$$

$$:= \mathcal{A}\left[\!\left[\Delta,\ \Gamma\ \bigtriangledown_\Sigma\ e_1 :: \tau_2 \rightarrow\!\!\!\!\!\!\rightarrow \tau_1\right]\!\right]_{\nu,\eta} (\mathcal{A}\left[\!\left[\Delta,\ \Gamma\ \bigtriangledown_\Sigma\ e_2 :: \tau_2\right]\!\right]_{\nu,\eta})$$

$$\mathcal{A}\left[\!\left[\frac{\Delta,\ \Gamma.x{:}\tau_1\ \bigtriangledown_\Sigma\ e :: \tau_2}{\Delta, \Gamma \rhd_\Sigma \lambda x.e :: \tau_1 \rightarrow\!\!\!\!\!\!\rightarrow \tau_2}\ (\rightarrow I_u)\right]\!\right]_{\nu,\eta} := \text{the unique } f \in \mathcal{SA}[\![\tau_1 \rightarrow\!\!\!\!\!\!\rightarrow \tau_2]\!]_\nu \text{ such that}$$

$$\forall d \in \mathcal{SA}[\![\tau_1]\!]_\nu . f(d) = \mathcal{A}\left[\!\left[\Delta,\ \Gamma.x{:}\tau_1\ \bigtriangledown_\Sigma\ e :: \tau_2\right]\!\right]_{\nu,\eta.[x \mapsto d]}$$

$$\mathcal{A}\left[\!\left[\frac{\Delta,\ \Gamma.x{:}\tau_1\ \bigtriangledown_\Sigma\ e :: \tau_2}{\Delta, \Gamma \rhd_\Sigma \lambda x{:}\tau_1.e :: \tau_1 \rightarrow\!\!\!\!\!\!\rightarrow \tau_2}\ (\rightarrow I_s)\right]\!\right]_{\nu,\eta} := \text{the unique } f \in \mathcal{SA}[\![\tau_1 \rightarrow\!\!\!\!\!\!\rightarrow \tau_2]\!]_\nu \text{ such that}$$

$$\forall d \in \mathcal{SA}[\![\tau_1]\!]_\nu . f(d) = \mathcal{A}\left[\!\left[\Delta,\ \Gamma.x{:}\tau_1\ \bigtriangledown_\Sigma\ e :: \tau_2\right]\!\right]_{\nu,\eta.[x \mapsto d]}$$

$$\mathcal{A}\left[\!\!\left[\frac{\Delta, \Gamma \ \triangledown_\Sigma \ e :: \tau}{\Delta, \Gamma \triangleright_\Sigma e{:}\tau :: \tau} \ (\mathit{constrained})\right]\!\!\right]_{\nu,\eta} := \mathcal{A}\left[\!\!\left[\Delta, \Gamma \ \triangledown_\Sigma \ e :: \tau\right]\!\!\right]_{\nu,\eta}$$

□

Let us examine the interpretation function in a more detailed way. The most interesting point is the interpretation of a polymorphic identifier defined by the case (*polyid*). The identifier is interpreted in our polymorphic algebra, yielding a polymorphic value. This value is immediately applied to the requisite number of domains. As result we get a monomorphic value. Therefore, each occurrence of a polymorphic identifier is interpreted by a monomorphic value. The interpretation, however, depends on the instantiation of the identifier, i.e. on the substitution σ. We made this substitution explicit in the derivation tree by writing $\sigma(\tau)$. Note that this is a bit tricky, because the substitution is not explicitly recorded in the derivation tree. However, because of the side condition that each Π-bound sort variable must occur in the body of the polymorphic sort expression, the substitution can be restored in a unique way for all α_i, $1 \leq i \leq n$. Therefore, the interpretation of a polymorphic function in a sort derivation is uniquely determined. If we omit the side condition, the interpretation of a polymorphic identifier can be ambiguous as shown by the following example.

Example 4.2.1 *Ambiguous interpretation of a polymorphic identifier*

Let `empty`:$\Pi\alpha, \beta$.`Container(`α, β`)` $\Rightarrow \beta$ be a polymorphic identifier in F. The interpretation of the sort derivation

$$\frac{}{\{\texttt{Container(Int,List(Int))},\texttt{Container(Nat,List(Int))}\}, \Gamma \triangleright_\Sigma \texttt{empty::List(Int)}} \ (\mathit{polyid})$$

is ambiguous, because there exist at least two substitutions, σ_1=[`Int`/α,`List(Int)`/β] and σ_2=[`Nat`/α,`List(Int)`/β], such that $\sigma_1(\beta)$=`List(Int)` and $\sigma_2(\beta)$=`List(Int)`. But the interpretation depends on the chosen substitution, because $\texttt{empty}^{\mathcal{A}}[\texttt{Int}^{\mathcal{SA}}, \texttt{List(Int)}^{\mathcal{SA}}]$ may be different from $\texttt{empty}^{\mathcal{A}}[\texttt{Nat}^{\mathcal{SA}}, \texttt{List(Int)}^{\mathcal{SA}}]$. □

We can omit the side condition if we explicitly record the chosen substitution in the derivation tree. This would, however, unnecessarily complicate our derivation trees.

Another remarkable point is the fact that the interpretation of the unsorted λ-abstraction defined in case ($\rightarrow I_u$) does not differ from the interpretation of the sorted λ-abstraction defined in case ($\rightarrow I_s$), though explicit sort annotations were introduced to influence the semantics. The same holds for the case (*constrained*). The interpretation is the same as

the interpretation of the unsorted term. But, of course, this is no contradiction. Explicit sort annotations influence the set of possible sort derivations for a term. And we will see that the semantics of a specification depends on all possible sort derivations.

In the definition we asserted that $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ is a function mapping sort derivations $D = \Delta, \Gamma \bigtriangledown_\Sigma e :: \tau$ to elements in $\mathcal{SA}[\![\tau]\!]_\nu$. To show the well-definedness of our definition we must at first check whether $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ is indeed a function. In other words, we must show that the meaning of a sort derivation is uniquely determined. Furthermore, the result of interpreting D must be an element of $\mathcal{SA}[\![\tau]\!]_\nu$.

Proposition 4.2.2 *The interpretation function* $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ *is well-defined*

Let $D = \Delta, \Gamma \bigtriangledown_\Sigma e :: \tau$ be a sort derivation for a pre-term e. Let ν be a sort variable assignment and η be a variable assignment.

$$\begin{array}{c}\text{If } \models_{\mathcal{SA},\nu} \Delta \text{ and } \eta \models_{\mathcal{SA},\nu} \Gamma \text{ then} \\ \mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta} \text{ yields a uniquely determined result in } \mathcal{SA}[\![\tau]\!]_\nu.\end{array}$$

□

Now we extend the meaning function to sort derivations for pre-formulae.

Definition 4.2.4 *Meaning of a sort derivation for a pre-formula*

Let D be a sort derivation for a pre-formula f under sort predicate assumption Δ and variable assumption Γ with respect to a polymorphic signature Σ. Let $\mathcal{A} = (\mathcal{SA}, \mathcal{P}, \mathcal{F})$ be a polymorphic Σ-algebra. Let ν be a sort variable assignment and η be a variable assignment such that $\models_{\mathcal{SA},\nu} \Delta$ and $\eta \models_{\mathcal{SA},\nu} \Gamma$. The meaning of a sort derivation D for a pre-formula in $\mathcal{A}$ under ν and η is $\mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta}$. The interpretation function $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ maps sort derivations D for pre-formulae to the truth values $\{\mathit{true}, \mathit{false}\}$ and is recursively defined on the structure of the derivation tree D in the following way:

Base cases

$$\mathcal{A}\left[\!\left[\frac{}{\Delta, \Gamma \rhd_\Sigma p}\ (const)\right]\!\right]_{\nu,\eta} := p^{\mathcal{A}} \quad \left\{\ p \in P_\epsilon\right.$$

$$\mathcal{A}\left[\!\left[\frac{\Delta, \Gamma \bigtriangledown_\Sigma t :: \tau}{\Delta, \Gamma \rhd_\Sigma p\, t}\ (appl)\right]\!\right]_{\nu,\eta} := p^{\mathcal{A}}(\mathcal{A}\left[\!\left[\Delta, \Gamma \bigtriangledown_\Sigma t :: \tau\right]\!\right]_{\nu,\eta}) \quad \left\{\begin{array}{l} p \in P_\tau \\ \text{where } \tau \in \mathrm{T}_\Omega(\emptyset)\end{array}\right.$$

$$\mathcal{A}\left[\!\!\left[\frac{\Delta,\,\Gamma\ \bigtriangledown_\Sigma\ t :: \sigma(\tau)}{\Delta,\Gamma \rhd_\Sigma p\,t}\ (appl)\right]\!\!\right]_{\nu,\eta} :=$$

$$\mathrm{p}^{\mathcal{A}}[\mathcal{SA}[\![\sigma(\alpha_1)]\!]_\nu,\ldots,\mathcal{SA}[\![\sigma(\alpha_n)]\!]_\nu](\mathcal{A}\left[\!\!\left[\Delta,\,\Gamma\ \bigtriangledown_\Sigma\ t :: \sigma(\tau)\right]\!\!\right]_{\nu,\eta})$$

$$p \in P_{\Pi\alpha_1,\ldots,\alpha_n.A_1,\ldots,A_m \Rightarrow \tau} \qquad \sigma : \Phi \to \mathrm{T}_\Omega(\Phi)$$
$$\sigma(A_i) \in \Delta, 1 \le i \le m$$

Inductive cases

$$\mathcal{A}\left[\!\!\left[\frac{\Delta,\,\Gamma.x{:}\tau\ \bigtriangledown_\Sigma\ f}{\Delta,\Gamma \rhd_\Sigma \forall x.f}\ (univ_u)\right]\!\!\right]_{\nu,\eta} :=$$

$$\begin{cases} true & \text{if for all } d \in \mathcal{SA}[\![\tau]\!]_\nu\text{: } \mathcal{A}\left[\!\!\left[\Delta,\,\Gamma.x{:}\tau\ \bigtriangledown_\Sigma\ f\right]\!\!\right]_{\nu,\eta[x\mapsto d]} = true \\ false & \text{otherwise} \end{cases}$$

$$\mathcal{A}\left[\!\!\left[\frac{\Delta,\,\Gamma.x{:}\tau\ \bigtriangledown_\Sigma\ f}{\Delta,\Gamma \rhd_\Sigma \forall x{:}\tau.f}\ (univ_s)\right]\!\!\right]_{\nu,\eta} :=$$

$$\begin{cases} true & \text{if for all } d \in \mathcal{SA}[\![\tau]\!]_\nu\text{: } \mathcal{A}\left[\!\!\left[\Delta,\,\Gamma.x{:}\tau\ \bigtriangledown_\Sigma\ f\right]\!\!\right]_{\nu,\eta[x\mapsto d]} = true \\ false & \text{otherwise} \end{cases}$$

$$\mathcal{A}\left[\!\!\left[\frac{\Delta,\,\Gamma\ \bigtriangledown_\Sigma\ f}{\Delta,\Gamma \rhd_\Sigma \neg f}\ (not)\right]\!\!\right]_{\nu,\eta} := \begin{cases} true & \text{if } \mathcal{A}\left[\!\!\left[\Delta,\,\Gamma\ \bigtriangledown_\Sigma\ f\right]\!\!\right]_{\nu,\eta} = false \\ false & \text{otherwise} \end{cases}$$

$$\mathcal{A}\left[\!\!\left[\frac{\Delta,\,\Gamma\ \bigtriangledown_\Sigma\ f_1 \quad \Delta,\,\Gamma\ \bigtriangledown_\Sigma\ f_2}{\Delta,\Gamma \rhd_\Sigma f_1 \vee f_2}\ (or)\right]\!\!\right]_{\nu,\eta} :=$$

$$\begin{cases} true & \text{if } \mathcal{A}\left[\!\!\left[\Delta,\,\Gamma\ \bigtriangledown_\Sigma\ f_1\right]\!\!\right]_{\nu,\eta} = true \quad \text{or} \quad \mathcal{A}\left[\!\!\left[\Delta,\,\Gamma\ \bigtriangledown_\Sigma\ f_2\right]\!\!\right]_{\nu,\eta} = true \\ false & \text{otherwise} \end{cases}$$

□

The statements made for the interpretation of sort derivations for terms can also be applied to the interpretation of sort derivations for formulae. For the same reason as in Def. 4.2.3, the substitution σ explicitly used in case (*appl*) is uniquely determined. Also the interpretation of the unsorted universal quantifier in case ($univ_u$) does not differ from the interpretation of the sorted quantifier in case ($univ_s$). Apart from that, the interpretation of formulae does not differ from conventional two-valued first order logic. Finally, we also have to prove the well-definedness of the interpretation.

Proposition 4.2.3 *The interpretation function* $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ *is well-defined*

Let $D = \Delta, \Gamma \bigtriangledown_\Sigma f$ be a sort derivation for a pre-formula f. Let ν be a sort variable assignment and η be a variable assignment.

$$\begin{array}{c}\text{If } \models_{\mathcal{SA},\nu} \Delta \text{ and } \eta \models_{\mathcal{SA},\nu} \Gamma \text{ then} \\ \mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta} \text{ yields a uniquely determined result in } \{\mathit{true}, \mathit{false}\}.\end{array}$$

□

Now we extend the meaning function to sort derivations for qualified formulae.

Definition 4.2.5 *Meaning of a sort derivation for a qualified formula*

Let D be a sort derivation for a qualified formula qf under sort predicate assumption Δ and variable assumption Γ with respect to a polymorphic signature Σ. Let $\mathcal{A} = (\mathcal{SA}, \mathcal{P}, \mathcal{F})$ be a polymorphic Σ-algebra. Let ν be a sort variable assignment and η be a variable assignment such that $\eta \models_{\mathcal{SA},\nu} \Gamma$. The meaning of a sort derivation D for a qualified formula in $\mathcal{A}$ under ν and η is $\mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta}$. The interpretation function $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ maps sort derivations D for qualified formulae to the truth values $\{\mathit{true}, \mathit{false}\}$ and is defined on the structure of the derivation tree D in the following way:

$$\mathcal{A}\left[\!\left[\frac{\Delta, \Gamma \bigtriangledown_\Sigma f}{\Delta, \Gamma \rhd_\Sigma \forall \alpha_1, \ldots, \alpha_n.\ A_1, \ldots, A_m \Rightarrow f}\ (\mathit{qualification})\right]\!\right]_{\nu,\eta} :=$$

$$\begin{cases} \mathit{true} & \text{if not } \models_{\mathcal{SA},\nu} \{A_1, \ldots, A_m\} \quad \text{or} \quad \mathcal{A}\left[\!\left[\Delta, \Gamma \bigtriangledown_\Sigma f\right]\!\right]_{\nu,\eta} = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}$$

□

The arrow $\Rightarrow$ of the qualified formula is interpreted as the logical implication. The result of interpreting a qualified formula is *true* if either the sort variable assignment does not satisfy the sort predicate context of the qualified formula or the interpretation of the formula f yields *true*. Again we have to prove that this interpretation is a truth-valued function yielding *true* or *false*.

Proposition 4.2.4 *The interpretation function* $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ *is well-defined*

Let $D = \Delta, \Gamma \bigtriangledown_\Sigma \forall \overline{\alpha_n}.\ \overline{A_m} \Rightarrow f$ be a sort derivation for a qualified formula. Let ν be a sort variable assignment and η be a variable assignment.

$$\text{If } \eta \models_{\mathcal{SA},\nu} \Gamma \text{ then } \mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta} \text{ yields a uniquely determined result in } \{\mathit{true}, \mathit{false}\}.$$

□

Based on the meaning of sort derivations for formulae and qualified formulae we now define whether a sort derivation for an arbitrary closed formula is satisfied in an algebra. Because we have no binding mechanism for sort variables, a sort derivation may contain free sort variables. These sort variables are treated as universally quantified at the outermost level of a formula. A sort derivation is only satisfied if, for each sort variable assignment satisfying the sort predicate assumption, the respective interpretation function yields *true*. More precisely:

Definition 4.2.6 *Satisfaction of a sort derivation*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let $D = \Delta, \emptyset \bigtriangledown_\Sigma f$ be a sort derivation for a non-qualified or qualified closed formula f. Let η_0 be an arbitrary variable assignment. D is *satisfied* in $\mathcal{A}$, written

$$\mathcal{A} \models D \quad \text{iff} \quad \mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta_0} = \mathit{true} \quad \text{for each } \nu : \Phi \to \mathcal{U}.\ \models_{\mathcal{SA},\nu} \Delta$$

□

Up to now, we were only working on sort derivations and their interpretations. A specification, however, consists of well-formed formulae and not of sort derivations. We will now make a semantic connection between formulae and their sort derivations. In formulae, polymorphic functions are applied without explicit instantiation and variables may be declared without explicit sort annotations. This omitted information can be deduced from the context. The deduction, however, is not always unique. Normally, there are more possible sort annotations for a bound variable and there are more possible instantiations for a polymorphic function. Each sort derivation for a formula represents exactly one possible deduction of the omitted information. Now the question arises, which sort derivation should be used to define the semantics of a formula. The most intuitive solution is to take all possible sort derivations into consideration. A formula is only satisfied in an algebra if all possible sort derivations for this formula are satisfied in the algebra. This is the most restricting point of view.

Definition 4.2.7 *Satisfaction of a closed well-formed Σ-formula*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let $f \in \mathrm{SEN}_\Sigma$ be a closed well-formed Σ-formula. The formula f is *satisfied* in $\mathcal{A}$, written

$$\mathcal{A} \models f \quad \text{iff} \quad \mathcal{A} \models D \quad \text{for each sort derivation } D \text{ that ends with } \Delta, \emptyset \rhd_\Sigma f \text{ for some sort predicate assumption } \Delta$$

□

Now it becomes clear how explicit sort annotations can influence the semantics of a formula. They decrease the number of possible sort derivations for a formula. This leads to an increasing number of algebras satisfying the formula, because an algebra has to satisfy less sort derivations.

Finally, we define our notion of model.

Definition 4.2.8 *Model*

Let $PS = (\Sigma, A)$ be a polymorphic specification. A polymorphic Σ-algebra $\mathcal{A}$ is called a *model* of PS iff each formula from A is satisfied in the algebra, i.e.

$$\forall f \in A.\ \mathcal{A} \models f$$

□

4.3 A Semantics Based on Principal Sort Derivations

In a polymorphic language providing implicit parametric polymorphism the actual sort parameter of a polymorphic function must be inferred at each application of the function. In addition, for variables declared without sort information the formula judgement chooses a possible sort term for these variables to get a well-formed formula. In both cases there is, however, not only one possible actual sort parameter or sort term, but sometimes infinitely many. Each sort derivation of a formula, representing a proof for the well-formedness of the formula, chooses only one possible sort parameter or sort term. For this reason we had to take all possible sort derivations into consideration when defining the models of a polymorphic specification.

This definition, at first glance natural, has a drawback. If we want to check whether a polymorphic algebra is a model of a given polymorphic specification, we must prove that the algebra satisfies each possible sort derivation of the formula. But in most cases we cannot check all derivations, because there are infinitely many. Moreover, if we want to use a theorem prover to reason about properties of a specification we would have to manage all derivations of this specification. In practice, however, we cannot manage infinitely many sort derivations in a proof system. We must choose one sort derivation.

Of course, we cannot use any arbitrary sort derivation for deriving theorems. Otherwise, we might not be able to derive all theorems of a specification, i.e. the resulting prover might not be complete. Therefore, the question arises whether there exists some kind of principal sort derivation for a formula representing all other possible sort derivations of this formula. The principal sort derivation must have the property that if an algebra satisfies this derivation then the algebra satisfies all possible sort derivations of this formula. We can then use this principal sort derivation in a suitable proof system to derive theorems, without losing completeness.

However, the judgements defined in Section 3.2 do not enable us to find such a principal sort derivation for an arbitrary formula. The following example shows a well-formed formula that possesses no principal sort derivation in the sense discussed above.

Example 4.3.1 *Formula without principal sort derivation*

Let Ω=({{`Bool,Nat`}$_0$},{{`EQ`}$_1$}) be a sort signature. Let S=(Ω, {`EQ(Bool),EQ(Nat)`}) be a sort specification. Let Σ=(S, {`p`: $\Pi\alpha$. `EQ(`α`)` $\Rightarrow$ α},{}) be a polymorphic signature. Let qf = $\emptyset$ $\Rightarrow$ $\forall$ `x. p(x)` be a qualified formula[2]. We first have to prove that qf is a closed well-formed Σ formula at all, i.e. whether $qf \in \mathrm{SEN}_\Sigma$.

[2]We explicitly denote the empty sort predicate context by $\emptyset$.

Proof:

We must show that there exists a sort derivation for qf under some sort predicate assumption Δ and the empty identifier assumption $\emptyset$. Fig. 4.3 shows a sort derivation ending with $\texttt{EQ(Bool)}, \emptyset \rhd_\Sigma qf$ and thus proves the well-formedness of qf. This, however, is not the only sort derivation for qf. Fig. 4.4 shows another derivation tree ending with $\texttt{EQ(Nat)}, \emptyset \rhd_\Sigma qf$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{EQ(Bool),x:Bool} \rhd_\Sigma \texttt{x::Bool}}(var)}{\texttt{EQ(Bool),x:Bool} \rhd_\Sigma \texttt{p(x)}}(appl)}{\texttt{EQ(Bool)},\emptyset \rhd_\Sigma \forall \texttt{x. p(x)}}(univ_u)}{\texttt{EQ(Bool)},\emptyset \rhd_\Sigma \emptyset \Rightarrow \forall \texttt{x. p(x)}}(qualification)\left\{ \emptyset \vdash_{SA} \texttt{EQ(Bool)}\right.$$

Figure 4.3: Sort derivation for $\emptyset \Rightarrow \forall \texttt{x. p(x)}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{EQ(Nat),x:Nat} \rhd_\Sigma \texttt{x::Nat}}(var)}{\texttt{EQ(Nat),x:Nat} \rhd_\Sigma \texttt{p(x)}}(appl)}{\texttt{EQ(Nat)},\emptyset \rhd_\Sigma \forall \texttt{x. p(x)}}(univ_u)}{\texttt{EQ(Nat)},\emptyset \rhd_\Sigma \emptyset \Rightarrow \forall \texttt{x. p(x)}}(qualification)\left\{ \emptyset \vdash_{SA} \texttt{EQ(Nat)}\right.$$

Figure 4.4: Sort derivation for $\emptyset \Rightarrow \forall \texttt{x. p(x)}$

Both trees have the same structure, i.e. the same rules are applied in the same order in both proofs. The trees only differ in their nodes. The second proof tree is obtained by replacing all occurrences of the sort constructor `Bool` by the sort constructor `Nat`.

It is easy to see that these are the only possible relevant[3] sort derivations for qf under the empty identifier assumption with respect to the given signature Σ. However, neither can be taken as a representative. While in the first derivation the predicate `p` must hold for all elements of sort `Bool`, in the second derivation `p` must hold for all elements of sort `Nat`. Thus, for both sort derivations it is easy to find algebras satisfying only one sort derivation, but not the other one. □

The reason for this property is the sort predicate context of the qualified formula. This context forces us to find a proof using a sort predicate assumption that is weaker than the empty context. If we look at the formula $\forall \texttt{x. p(x)}$, i.e. the same formula but without an explicit sort predicate context, we can find more sort derivations proving the well-formedness of this formula. Fig. 4.5 shows a possible sort derivation for $\forall \texttt{x. p(x)}$

[3] Of course we can in both derivations replace the sort predicate assumption by {`EQ(Bool),EQ(Nat)`}. But this replacement does not influence the semantics of the derivations.

under sort predicate assumption $\texttt{EQ}(\alpha)$ and empty identifier assumption with respect to Σ defined in Example 4.3.1.

$$\dfrac{\dfrac{\dfrac{}{\texttt{EQ}(\alpha), \texttt{x}{:}\alpha \ \rhd_\Sigma \ \texttt{x}::\alpha}\,(var)}{\texttt{EQ}(\alpha), \texttt{x}{:}\alpha \ \rhd_\Sigma \ \texttt{p(x)}}\,(appl)}{\texttt{EQ}(\alpha), \emptyset \rhd_\Sigma \forall \texttt{x. p(x)}}\,(univ_u)$$

Figure 4.5: Sort derivation for $\forall\texttt{x. p(x)}$

Other possible derivations proving the well-formedness of $\forall\texttt{x. p(x)}$ are those proof trees in Fig. 4.3 and 4.4 obtained by omitting the application of the rule *(qualification)* in the last step. These are, up to sort variable renaming and stronger sort predicate assumptions, the only possible sort derivations for $\forall\texttt{x. p(x)}$.

Now, however, we can prove the sort derivation in Fig. 4.5 to be a principal sort derivation. Every algebra satisfying this sort derivation satisfies all other sort derivations for $\forall\texttt{x. p(x)}$. This holds because we used a sort variable instead of a concrete sort constructor. An algebra must satisfy the sort derivation for all possible sort variable assignments. Thus, the sort derivation must also be satisfied if we assign the interpretation of `Bool` or `Nat` to the sort variable α. In the next sections we will generalize this result.

In Section 4.3.1 we investigate the concept of principal sort derivations for non-qualified formulae. We start by defining a semantic "more restrictive" ordering on sort derivations comparing the sets of algebras satisfying the sort derivations. Then we define a syntactic "more general" ordering on sort derivations. We prove that if a sort derivation is more general than another it is also more restrictive, i.e. the set of algebras satisfying the more general sort derivation is smaller than the set of algebras satisfying the other sort derivation. Furthermore, we show that each non-qualified formula has a greatest element with respect to the "more general" ordering called principal sort derivation. Finally, we present a definition of satisfaction for non-qualified formulae that is based on principal sort derivations. We show the equivalence of this notion of satisfaction with the one defined in Section 4.2.

In Section 4.3.2 we investigate the concept of principal sort derivations for qualified formulae. We define a new sort inference calculus to allow more abstract sort derivations for qualified formulae. We show that the new calculus does not change the concept of well-formedness. We extend the definitions and propositions made in Section 4.3.1 to qualified formulae. In particular, we show the existence of principal sort derivations for qualified formulae. Furthermore, we present a satisfaction concept for qualified formulae based on a principal sort derivation. We show that this satisfaction concept is not equivalent to the one defined in Section 4.2 but slightly more rigorous. This fact, however, results only

from changing the sort inference calculus. Finally, we define principal sort derivation based models for polymorphic specifications.

4.3.1 Principal Sort Derivations for Non-Qualified Formulae

To simplify our notion of model we are looking for a sort derivation for a formula that can represent all other sort derivations of this formula. We call this sort derivation the *principal sort derivation*. This sort derivation must, in some sense, be the most restrictive one, the one with the least model set. For this reason, we define the following semantic order on sort derivations.

Definition 4.3.1 *Restriction Order*

Let $f \in \mathrm{PF}_\Sigma(\Phi)$ be a closed pre-formula. Let $D_1{=}\Delta_1$, $\emptyset \bigtriangledown_\Sigma f$ and $D_2{=}\Delta_2$, $\emptyset \bigtriangledown_\Sigma f$ be two sort derivations for f. D_1 is called *more restrictive* than D_2, written

$$D_1 \models D_2 \quad \text{iff} \quad \mathcal{A} \models D_1 \Rightarrow \mathcal{A} \models D_2 \ \text{ for all } \Sigma \text{ algebras } \mathcal{A}.$$

□

We now must find syntactic criteria for sort derivations which enable us to determine whether a sort derivation for a formula is more restrictive than another sort derivation for this formula.

In the last section we saw that all sort derivations given as example for a particular formula had the same structure. This was not pure coincidence. We can show this property to hold not only for all non-qualified formulae, but also for all qualified formulae.

Proposition 4.3.1 *Uniqueness of sort derivation structure*

Let $f \in \mathrm{PF}_\Sigma(\Phi) \cup \mathrm{QF}_\Sigma$ be an arbitrary formula. Every sort derivation for f has the same structure according to the judgement rules given in Def. 3.2.7, 3.2.9 and 3.2.11. More precisely, in all sort derivations the same judgement rules are applied in the same order. □

Thus, the sort derivations for a formula can only differ in the nodes of the derivation trees. As already mentioned in the introductory part of this chapter, the cause of this difference lies in the declaration of unsorted variables and in the application of polymorphic functions and predicates. Therefore, we can show the following proposition.

Proposition 4.3.2 *Uniqueness of sort derivations*

Let $f \in \mathrm{SEN}_\Sigma$ be a closed well-formed Σ-formula. If every bound variable is sorted explicitly, and if no polymorphic function or predicate is applied in the formula, then, up to different sort predicate assumptions, there exists only one sort derivation for f. In particular, the sort predicate assumption can be chosen arbitrarily. □

The cause of different sort derivations lies in the rules (*polyid*), ($\rightarrow I_u$), (*appl*) and ($univ_u$) of the calculi given in Def. 3.2.7 and 3.2.9. In the rules (*polyid*) and (*appl*), handling polymorphic identifiers, an arbitrary instantiation may be chosen, and in the rules ($\rightarrow I_u$) and ($univ_u$) an arbitrary sort term for the bound variables can be assumed. Because we are looking for the most restrictive sort derivation we must always choose the most abstract possible instantiation and assumption. Abstraction in sort terms can be achieved by using sort variables. Therefore, the more sort variables a sort term contains, the more abstract the sort term is. The sort derivation with the most abstract instantiations and assumptions will be the most restrictive one, because an algebra must satisfy the sort derivation for all possible sort variable assignments. Based on this thought we now define a syntactic order on sort derivations in the following way:

Definition 4.3.2 *σ-instance relation*

Let $D_1 = \Delta_1, \Gamma_1 \bigtriangledown_\Sigma f$ and $D_2 = \Delta_2, \Gamma_2 \bigtriangledown_\Sigma f$ be two sort derivations for a pre-formula f. Let $\sigma : \Phi \rightarrow \mathrm{T}_\Omega(\Phi)$ be a sort variable substitution. D_1 is called a *σ-instance* of D_2, written $D_1 \leq^\sigma D_2$ iff for each of the following corresponding derivation nodes holds:

- if $\dfrac{}{\Delta_2, \Gamma_2 \rhd_\Sigma c :: \tau_2}$ (*polyid*) occurs in D_2 then
  for the corresponding node $\dfrac{}{\Delta_1, \Gamma_1 \rhd_\Sigma c :: \tau_1}$ (*polyid*) in D_1 holds: $\tau_1 = \sigma(\tau_2)$

- if $\dfrac{\Delta_2, \Gamma_2 . x : \tau_{21} \bigtriangledown_\Sigma e :: \tau_{22}}{\Delta_2, \Gamma_2 \rhd_\Sigma \lambda x.e :: \tau_{21} \rightarrow \tau_{22}}$ ($\rightarrow I_u$) occurs in D_2 then for the corresponding node $\dfrac{\Delta_1, \Gamma_1 . x : \tau_{11} \bigtriangledown_\Sigma e :: \tau_{12}}{\Delta_1, \Gamma_1 \rhd_\Sigma \lambda x.\, e :: \tau_{11} \rightarrow \tau_{12}}$ ($\rightarrow I_u$) in D_1 holds: $\tau_{11} = \sigma(\tau_{21})$

- if $\dfrac{\Delta_2, \Gamma_2 \bigtriangledown_\Sigma t :: \tau_2}{\Delta_2, \Gamma_2 \rhd_\Sigma p\, t}$ (*appl*) occurs in D_2 then
  for the corresponding node $\dfrac{\Delta_1, \Gamma_1 \bigtriangledown_\Sigma t :: \tau_1}{\Delta_1, \Gamma_1 \rhd_\Sigma p\, t}$ (*appl*) in D_1 holds: $\tau_1 = \sigma(\tau_2)$

- if $\dfrac{\Delta_2, \Gamma_2 . x : \tau_2 \bigtriangledown_\Sigma f}{\Delta_2, \Gamma_2 \rhd_\Sigma \forall x.\, f}$ ($univ_u$) occurs in D_2 then
  for the corresponding node $\dfrac{\Delta_1, \Gamma_1 . x : \tau_1 \bigtriangledown_\Sigma f}{\Delta_1, \Gamma_1 \rhd_\Sigma \forall x.\, f}$ ($univ_u$) in D_1 holds: $\tau_1 = \sigma(\tau_2)$

□

We now show that the rules (*polyid*) and $(\rightarrow I_u)$ are indeed the only rules influencing the sort of a term for a fixed signature and variable assumption. We show this by proving that the σ-instance propagates to the sort term of a derivation node.

Proposition 4.3.3 *σ-instances propagate to sort terms*

Let $D_1 = \Delta_1, \Gamma_1 \bigtriangledown_\Sigma e :: \tau_1$ and $D_2 = \Delta_2, \Gamma_2 \bigtriangledown_\Sigma e :: \tau_2$ be two sort derivations for a pre-term e. Let $\sigma : \Phi \rightarrow \mathrm{T}_\Omega(\Phi)$ be a sort variable substitution such that $\Gamma_1 = \sigma(\Gamma_2)$ and $\forall \alpha \in FSV(e).\ \sigma(\alpha) = \alpha$.

$$\text{If } D_1 \leq^\sigma D_2 \text{ then } \tau_1 = \sigma(\tau_2).$$

□

To prove this proposition we need two side conditions. The first one states that the sort variable assumption of D_1 must also be a σ-instance of the sort variable assumption of D_2. The second one states that the substitution may not influence the sort variables occurring in the term. These variables, explicitly written by the specifier, are treated as constants.

We can now easily show that only the rules (*polyid*) and $(\rightarrow I_u)$ influence the sort of a term. Two derivations under the same sort variable assumption and with respect to the same signature, which do not differ in these corresponding rules, always end with the same sort term. Therefore, we call these derivations *sort equivalent derivations*. More precisely:

Proposition 4.3.4 *The sort of a term is only influenced by (polyid) and $(\rightarrow I_u)$*

Let $D_1 = \Delta_1, \Gamma \bigtriangledown_\Sigma e :: \tau_1$ and $D_2 = \Delta_2, \Gamma \bigtriangledown_\Sigma e :: \tau_2$ be two sort derivations for a pre-term e under the same identifier assumption Γ.

$$\text{If } D_1 \text{ and } D_2 \text{ are sort equivalent derivations, i.e. } D_1 \leq^\varepsilon D_2 \text{ then } \tau_1 = \tau_2.$$

Proof: Follows directly from Prop. 4.3.3. □

The syntactic instance order defined so far is not enough to characterize our semantic restriction order. Up to now we have not taken the sort predicate assumption into consideration. This assumption is only used in the rules (*polyid*) and (*appl*),which instantiate polymorphic identifiers. As we have already seen, the predicate assumption does not influence the sort of two sort equivalent derivations. The satisfaction of a sort derivation, however, depends on the sort predicate assumption. The interpretation of the derivation must be *true* for each sort variable assignment satisfying the sort predicate assumption. If the sort predicate assumption is very strict, only a few sort variable assignments satisfy it. Thus, the interpretation of the derivation must be *true* for only a few sort variable assignments. But then many algebras satisfy the sort derivation.

A sort derivation with a strict sort predicate assumption is clearly not restrictive. However, we are looking for the most restrictive one. We must therefore choose the sort derivation with the least strict sort predicate assumption. If no polymorphic identifier with qualifying sort predicates is used in a formula, the least strict sort predicate assumption is always the empty assumption. The following definition formalizes the above reflections. Our informal concept of "more strict" is formalized by the entailment relation $\vdash$.

Definition 4.3.3 *Syntactic order on sort derivations*

Let $f \in \mathrm{PF}_\Sigma(\Phi)$ be a closed formula. A sort derivation $D_2 = \Delta_2, \emptyset \bigtriangledown_\Sigma f$ is said to be *more general* than a sort derivation $D_1 = \Delta_1, \emptyset \bigtriangledown_\Sigma f$, written

$$\begin{array}{lll} D_1 \leq D_2 \quad \text{iff} & \exists \sigma : \Phi \to \mathrm{T}_\Omega(\Phi) & \text{with} \\ & \forall \alpha \in FSV(f).\ \sigma(\alpha) = \alpha & \text{and} \\ & \Delta_1 \vdash_{SA} \sigma(\Delta_2) & \text{and} \\ & D_1 \leq^\sigma D_2 & \end{array}$$

If $D_1 \leq D_2$ and $D_2 \leq D_1$ then D_1 and D_2 are called *equivalent sort derivations.* □

We must now prove that the syntactic criteria defined above indeed characterizes the semantic restriction order. The following proposition shows the connection; it states that if a sort derivation is more general then it is more restrictive. This proposition is of central significance: now we know, that if we choose the most general sort derivation, this is the most restrictive one, i.e. the principal sort derivation.

Proposition 4.3.5 *If a sort derivation is more general then it is more restrictive*

Let $D_1 = \Delta_1, \emptyset \bigtriangledown_\Sigma f$ and $D_2 = \Delta_2, \emptyset \bigtriangledown_\Sigma f$ be two sort derivations for a closed formula $f \in \mathrm{PF}_\Sigma(\Phi)$. If D_2 is more general than D_1 then D_2 is more restrictive than D_1, i.e.

$$D_1 \leq D_2 \quad \Rightarrow \quad D_2 \models D_1$$

□

The proof of this proposition is based on two auxiliary propositions. In fact, proving these two propositions is the main work. We already met this kind of proposition one level higher, namely on the sort level. In Prop. 4.1.1 we proved that the interpretation of two sort terms differing only by a sort variable substitution yields the same result if the substitution is reflected in the sort variable assignment. Now we are proving a similar proposition for sort derivations for terms and formulae. If a sort derivation D_1 is a σ-instance of another sort derivation D_2, the interpretation of both derivations yields the same result in case the sort variable assignment used to interpret D_2 reflects the substitution σ. A precise summary of this relationship is given in the following two propositions.

Proposition 4.3.6 *Interpretation equivalent sort derivations for formulae*

Let $D_1 = \Delta_1, \Gamma_1 \bigtriangledown_\Sigma f$ and $D_2 = \Delta_2, \Gamma_2 \bigtriangledown_\Sigma f$ be two sort derivations for a pre-formula $f \in \mathrm{PF}_\Sigma(\Phi)$. Let σ be a sort variable substitution such that $\forall \alpha \in FSV(f).\ \sigma(\alpha) = \alpha$ and $D_1 \leq^\sigma D_2$. Let ν and ν^σ be sort variable assignments such that $\models_{\mathcal{SA},\nu} \Delta_1$ and $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$. Let η be a variable assignment with $\eta \models_{\mathcal{SA},\nu} \Gamma_1$ and $\eta \models_{\mathcal{SA},\nu^\sigma} \Gamma_2$.

$$\text{If} \quad \nu^\sigma(\alpha) = \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu \text{ for each } \alpha \in \Phi \quad \text{then} \quad \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} = \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta}$$

□

Proposition 4.3.7 *Interpretation equivalent sort derivations for terms*

Let $D_1 = \Delta_1, \Gamma_1 \bigtriangledown_\Sigma e :: \tau_1$ and $D_2 = \Delta_2, \Gamma_2 \bigtriangledown_\Sigma e :: \tau_2$ be two sort derivations for a pre-term $e \in \mathrm{PT}_\Sigma(\Phi)$. Let σ be a sort variable substitution such that $\forall \alpha \in FSV(e).$ $\sigma(\alpha) = \alpha$ and $D_1 \leq^\sigma D_2$. Let ν and ν^σ be sort variable assignments such that $\models_{\mathcal{SA},\nu} \Delta_1$ and $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$. Let η be a variable assignment with $\eta \models_{\mathcal{SA},\nu} \Gamma_1$ and $\eta \models_{\mathcal{SA},\nu^\sigma} \Gamma_2$.

$$\text{If} \quad \nu^\sigma(\alpha) = \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu \text{ for each } \alpha \in \Phi \quad \text{then} \quad \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} = \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta}$$

□

We now precisely define the concept of a principal sort derivation.

Definition 4.3.4 *Principal sort derivation*

Let $f \in \mathrm{PF}_\Sigma(\Phi)$ be a closed formula. A sort derivation $D_1 = \Delta_1, \emptyset \bigtriangledown_\Sigma f$ is called a *principal sort derivation* for f, iff, for each sort derivation $D_2 = \Delta_2, \emptyset \bigtriangledown_\Sigma f$, $D_2 \leq D_1$ holds. □

The principal sort derivation is defined to be a greatest element with respect to the order $\leq$ on sort derivations. We will use this sort derivation to define the satisfaction of a formula. Therefore, the question arises whether there always exists such a sort derivation. Fortunately, we can give an algorithm computing a principal sort derivation for each well-formed formula.

Proposition 4.3.8 *Existence of principal sort derivations*

Let $f \in \mathrm{PF}_\Sigma(\Phi)$ be a closed formula. If there exists a sort derivation for f, then there exists a principal sort derivation for f.

Proof: In Section 5.5 we give an algorithm computing a principal sort derivation for formulae. □

Note that principal sort derivations for formulae are not unique, even up to sort variable renaming. There exist more greatest elements with respect to $\leq$ on sort derivations. The reason is that two different sort predicate assumptions Δ_1 and Δ_2 can be equivalent, i.e. $\Delta_1 \vdash_{SA} \Delta_2$ and $\Delta_2 \vdash_{SA} \Delta_1$. However, we can show that all principal sort derivations for a formula are satisfaction equivalent.

Proposition 4.3.9 *Equivalent sort derivations are satisfaction equivalent*

Let $\mathcal{A}$ be a polymorphic Σ algebra. Let D_1 and D_2 be two sort derivations for a closed formula. If D_1 and D_2 are equivalent sort derivations then D_1 and D_2 are satisfaction equivalent, i.e.

$$\text{if} \quad D_1 \leq D_2 \text{ and } D_2 \leq D_1 \quad \text{then} \quad \mathcal{A} \models D_1 \Leftrightarrow \mathcal{A} \models D_2$$

□

From this proposition we can immediately deduce that principal sort derivations are satisfaction equivalent, because principal sort derivations are clearly equivalent sort derivations. Based on this result we can now define an equivalent satisfaction concept for closed well-formed non-qualified Σ formulae.

Definition 4.3.5 *P-Satisfaction*

Let $\mathcal{A}$ be a polymorphic Σ algebra. Let $f \in \mathrm{PF}_\Sigma(\Phi)$ be a closed well-formed Σ formula, i.e. $f \in \mathrm{SEN}_\Sigma$. Formula f is *p-satisfied in* $\mathcal{A}$, written

$$\mathcal{A} \models_p f \quad \text{iff} \quad \mathcal{A} \models D \text{ for some principal sort derivation } D \text{ for f}$$

□

In contrast to the definition of satisfaction of Def. 4.2.7, where all possible sort derivations for a formula were used to define the concept, in the above definition we only use one sort derivation, namely the principal one, to define the satisfaction of a formula. The satisfaction is unambiguously defined because all principal sort derivations are satisfaction equivalent.

Now testing the satisfaction of a well-formed formula is much easier. We compute a principal sort derivation for this formula and test whether it is satisfied in an algebra. We can further use the principal sort derivation to reason about properties in a suitable proof system.

Finally, we show that both satisfaction concepts for non-qualified formulae are equivalent.

Proposition 4.3.10 *Satisfaction is equivalent to p-satisfaction*

Let $\mathcal{A}$ be a polymorphic Σ algebra. Let $f \in \mathrm{PF}_\Sigma(\Phi)$ be a closed well-formed Σ formula. Formula f is satisfied in $\mathcal{A}$ iff f is p-satisfied in $\mathcal{A}$, i.e.

$$\mathcal{A} \models_p f \quad \Leftrightarrow \quad \mathcal{A} \models f$$

□

4.3.2 Principal Sort Derivations for Qualified Formulae

The results we gained in Section 4.3.1 for non-qualified formulae can not be simply extended to qualified formulae. In the introductory part of Section 4.3 we gave an example for a qualified formula for which we can find a sort derivation but no principal sort derivation representing all other derivations.

The reason for this property lies in the inference rule (*qualification*). This rule, and all other inference rules in Section 3.2, were originally designed to define the well-formedness of terms and formulae. Thus, the rules were kept as easy as possible. The rule (*qualification*) forces us to find a sort derivation for a qualified formula using a sort predicate assumption that is entailed by the explicitly given sort predicate context. Our task is now to find a more liberal variation of the rule (*qualification*) allowing further sort derivations with more liberal sort predicate assumptions. More precisely the rule should fulfil the following requirements:

1. The new rule must not change the language, i.e. the concept of well-formedness must remain the same.
2. For each well-formed qualified formula there must exist a principal sort derivation. All algebras satisfying the principal sort derivation must also satisfy all other sort derivations of this qualified formula.
3. The rule must not change the semantics of our language, i.e. the satisfaction concept for qualified formulae must remain the same.

As we will see, we cannot find a rule completely satisfying all these requirements. The last requirement is too strong. The satisfaction concept for qualified formulae will change in a monotonic way. It will be more rigorous than the old one, i.e. the model class of a specification will be smaller.

Driven by the above requirements we create a new rule (*ext. qual.*). It will replace the old rule (*qualification*). As we saw in Section 4.3.1, the principal sort derivation is the derivation with the most abstract sort terms in the nodes of the derivation tree, i.e. the one with the most sort variables in the sort terms. However, more abstract sort terms also require more abstract sort predicate assumptions. Therefore, we must allow sort predicate assumptions that are more abstract than the given context. Formally, there must exist a substitution σ such that the context is a σ-instance of the assumption. Such a change does not influence the well-formedness property. If there exists a sort derivation under the abstract assumption, there also exists a derivation under the given context. We only have to apply the substitution to all sort terms in the derivation tree.

Definition 4.3.6 *Extended qualified formula judgement* $\rhd'$

The extended qualified formula judgement $\rhd' \subseteq \text{PSIG} \times \mathcal{P}(\text{AF}_\Omega(\Phi)) \times (<var> \rightarrow \text{T}_\Omega(\Phi)) \times \text{QF}_\Sigma$ is a set of quadruples $(\Sigma,\ \Delta,\ \Gamma,\ qf)$ where Σ, Δ, Γ and qf have the same meaning as in definition 3.2.11.

Again, we write $\Delta, \Gamma \rhd'_\Sigma qf$ instead of $(\Sigma, \Delta, \Gamma, qf) \in \rhd'$ and read "under sort predicate assumption Δ and identifier assumption Γ the qualified formula qf is well-formed with respect to the polymorphic signature Σ". $\Delta, \Gamma \rhd'_\Sigma qf$ iff there is a finite proof tree using the rule below that ends with $\Delta, \Gamma \rhd'_\Sigma qf$. A proof tree that ends with $\Delta, \Gamma \rhd'_\Sigma qf$ is called an *extended sort derivation for* qf *under* Δ *and* Γ *with respect to* Σ.

$$\frac{\Delta, \Gamma \rhd_\Sigma f}{\Delta, \Gamma \rhd'_\Sigma \forall \alpha_1, \ldots, \alpha_n . A_1, \ldots, A_m \Rightarrow f}\ (ext.\ qual.) \begin{cases} \exists \sigma : \Phi \rightarrow \text{T}_\Omega(\Phi) \text{ with} \\ \sigma(\alpha_i) = \alpha_i, 1 \leq i \leq n \\ \{A_1, \ldots, A_m\} \vdash_{SA} \sigma(\Delta) \end{cases}$$

□

Example 4.3.2 *Extended sort derivation*

In Ex. 4.3.1 we presented two sort derivations for a qualified formula $\emptyset \Rightarrow \forall$ `x. p(x)`. We saw that the trees shown in Fig. 4.3 and 4.4 are the only relevant sort derivations for this formula under the empty identifier assumption. Both are also valid extended sort derivations if we replace the application of rule *(qualification)* by an application of rule *(ext. qual.)*, because the side condition of the rule *(ext. qual.)* is trivially fulfilled. The more liberal rule *(ext. qual.)*, however, allows another extended sort derivation shown in Fig. 4.6. As we will see later, this is a principal sort derivation for $\emptyset \Rightarrow \forall$ `x. p(x)`.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\texttt{EQ}(\alpha), \texttt{x}{:}\alpha \rhd_\Sigma \texttt{x}::\alpha}\ (var)}{\texttt{EQ}(\alpha), \texttt{x}{:}\alpha \rhd_\Sigma \texttt{p(x)}}\ (pappl)}{\texttt{EQ}(\alpha), \emptyset \rhd_\Sigma \forall \texttt{x. p(x)}}\ (univ_u)}{\texttt{EQ}(\alpha), \emptyset \rhd_\Sigma \emptyset \Rightarrow \forall \texttt{x. p(x)}}\ (ext.\ qual.) \left\{ \emptyset \vdash_{SA} \texttt{[Nat/}\alpha\texttt{]EQ}(\alpha) \right.$$

Figure 4.6: Extended sort derivation for $\emptyset \Rightarrow \forall$`x. p(x)`

□

We now prove that our new rule indeed fulfils the first requirement, namely that the concept of well-formedness does not change if we replace the old qualified formula judgement by the new one defined above.

Proposition 4.3.11 *Equivalence of well-formedness*

Let $qf \in \mathrm{QF}_\Sigma$ be a qualified Σ-formula. There exists an extended sort derivation for qf iff there exists a sort derivation for qf. □

To prove the above proposition we must convert a derivation tree by applying a sort variable substitution to the tree. In a later chapter we will need an extension of this conversion which adds atomic sort formulae to the sort predicate assumptions of the tree. Formally this mapping is defined as follows:

Definition 4.3.7 *Substitution of sort derivations for terms*

Let $\mathrm{D} = \Delta, \Gamma \ \nabla_\Sigma \ t :: \tau$ be a sort derivation for a pre-term $t \in \mathrm{PT}_\Sigma(\Phi)$. Let σ be a sort variable substitution, and let $\Delta' \in \mathcal{P}(\mathrm{AF}_\Omega(\Phi)$ be a set of atomic sort formulae. We define a substitution $\sigma_{\Delta'}(D)$ on a sort derivation D for a pre-term; it extends σ and adds the atomic sort formulae Δ' to the sort predicate assumption Δ in the whole derivation tree.

- $\sigma_{\Delta'}(\dfrac{}{\Delta, \Gamma \ \rhd_\Sigma \ x :: \tau} \ (var)) := \dfrac{}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ x :: \sigma(\tau)} \ (var)$
- $\sigma_{\Delta'}(\dfrac{}{\Delta, \Gamma \ \rhd_\Sigma \ c :: \tau} \ (id)) := \dfrac{}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ c :: \sigma(\tau)} \ (id)$
- $\sigma_{\Delta'}(\dfrac{}{\Delta, \Gamma \ \rhd_\Sigma \ c :: \tau} \ (polyid)) := \dfrac{}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ c :: \sigma(\tau)} \ (polyid)$
- $\sigma_{\Delta'}(\dfrac{D_1 \quad D_2}{\Delta, \Gamma \ \rhd_\Sigma \ e_1 \, e_2 :: \tau} \ (\to E)) := \dfrac{\sigma_{\Delta'}(D_1) \quad \sigma_{\Delta'}(D_2)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ e_1 \, e_2 :: \sigma(\tau)} \ (\to E)$
- $\sigma_{\Delta'}(\dfrac{D_1}{\Delta, \Gamma \ \rhd_\Sigma \ \lambda x. \ e :: \tau_1 \to \tau_2} \ (\to I_u)) :=$
  $\dfrac{\sigma_{\Delta'}(D_1)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ \lambda x. \ e :: \sigma(\tau_1 \to \tau_2)} \ (\to I_u)$
- $\sigma_{\Delta'}(\dfrac{D_1}{\Delta, \Gamma \ \rhd_\Sigma \ \lambda x : \tau. \ e :: \tau \to \tau_2} \ (\to I_s)) :=$
  $\dfrac{\sigma_{\Delta'}(D_1)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ \lambda x : \tau. \ e :: \sigma(\tau \to \tau_2)} \ (\to I_s)$
- $\sigma_{\Delta'}(\dfrac{D_1}{\Delta, \Gamma \ \rhd_\Sigma \ e : \tau :: \tau} \ (constrained)) :=$
  $\dfrac{\sigma_{\Delta'}(D_1)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ e : \tau :: \sigma(\tau)} \ (constrained)$

□

Note that we do not apply the substitution to the sort annotations of the pre-term. We must now prove that this pure syntactic conversion of a derivation tree again yields a valid sort derivation tree. As the next proposition shows, this is only valid if the substitution σ has a certain property.

Proposition 4.3.12 *Substitution preserves sort derivation*

Let $D = \Delta, \Gamma \bigtriangledown_\Sigma t :: \tau$ be a sort derivation for a pre-term $t \in \mathrm{PT}_\Sigma(\Phi)$. Let σ be a sort variable substitution, and let $\Delta' \in \mathcal{P}(\mathrm{AF}_\Omega(\Phi)$ be a set of atomic sort formulae.

$$\text{If } \forall \alpha \in FSV(t).\ \sigma(\alpha){=}\alpha \text{ then } \sigma_{\Delta'}(D) \text{ is a sort derivation for } t.$$

□

We now extend the substitution and the corresponding proposition to sort derivations for formulae.

Definition 4.3.8 *Substitution of sort derivations for non-qualified formulae*

Let $D = \Delta, \Gamma \bigtriangledown_\Sigma f$ be a sort derivation for a non-qualified pre-formula $f \in \mathrm{PF}_\Sigma(\Phi)$. Let σ be a sort variable substitution, and let $\Delta' \in \mathcal{P}(\mathrm{AF}_\Omega(\Phi)$ be a set of atomic sort formulae. We define a substitution $\sigma_{\Delta'}(D)$ on a sort derivation D for a formula; it extends σ and adds the atomic sort formulae Δ' to the sort predicate assumption Δ in the whole derivation tree.

- $\sigma_{\Delta'}(\dfrac{}{\Delta, \Gamma \rhd_\Sigma p}\ (const)) := \dfrac{}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma p}\ (const)$
- $\sigma_{\Delta'}(\dfrac{D_1}{\Delta, \Gamma \rhd_\Sigma p\, t}\ (appl)) := \dfrac{\sigma_{\Delta'}(D_1)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma p\, t}\ (appl)$
- $\sigma_{\Delta'}(\dfrac{D_1}{\Delta, \Gamma \rhd_\Sigma p\, t}\ (appl)) := \dfrac{\sigma_{\Delta'}(D_1)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma p\, t}\ (appl)$
- $\sigma_{\Delta'}(\dfrac{D_1}{\Delta, \Gamma \rhd_\Sigma \forall x.f}\ (univ_u)) := \dfrac{\sigma_{\Delta'}(D_1)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma \forall x.f}\ (univ_u)$
- $\sigma_{\Delta'}(\dfrac{D_1}{\Delta, \Gamma \rhd_\Sigma \forall x : \tau.f}\ (univ_s)) := \dfrac{\sigma_{\Delta'}(D_1)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma \forall x : \tau.f}\ (univ_s)$
- $\sigma_{\Delta'}(\dfrac{D_1}{\Delta, \Gamma \rhd_\Sigma \neg f}\ (not)) := \dfrac{\sigma_{\Delta'}(D_1)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma \neg f}\ (not)$
- $\sigma_{\Delta'}(\dfrac{D_1 \quad D_2}{\Delta, \Gamma \rhd_\Sigma f_1 \vee f_2}\ (or)) := \dfrac{\sigma_{\Delta'}(D_1) \quad \sigma_{\Delta'}(D_2)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma f_1 \vee f_2}\ (or)$

□

Proposition 4.3.13 *Substitution preserves sort derivation*

Let $D = \Delta, \Gamma \bigtriangledown_\Sigma f$ be a sort derivation for a non-qualified formulae $f \in \mathrm{PF}_\Sigma(\Phi)$. Let σ be a sort variable substitution, and let $\Delta' \in \mathcal{P}(\mathrm{AF}_\Omega(\Phi)$ be a set of atomic sort formulae.

If $\forall \alpha \in FSV(f).\ \sigma(\alpha)=\alpha$ then $\sigma_{\Delta'}(D)$ is a sort derivation for f.

□

Next we check the second requirement for our new judgement rule. We want to show that the new rule enables a principal sort derivation along the lines of Section 4.3.1. For this reason we must first transfer those concepts we defined for sort derivations in Chapter 4.3.1 to extended sort derivations. We start by defining the meaning of an extended sort derivation.

Definition 4.3.9 *Meaning of an extended sort derivation for a qualified formula*

Let D be an extended sort derivation for a qualified formula qf under sort predicate assumption Δ and variable assumption Γ with respect to a polymorphic signature Σ. Let $\mathcal{A} = (\mathcal{SA}, \mathcal{P}, \mathcal{F})$ be a polymorphic Σ-algebra. Let ν be a sort variable assignment and η be a variable assignment such that $\models_{\mathcal{SA},\nu} \Delta$ and $\eta \models_{\mathcal{SA},\nu} \Gamma$. The meaning of an extended sort derivation D for a qualified formula in $\mathcal{A}$ under ν and η is $\mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta}$. The interpretation function $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ maps extended sort derivations D for qualified formulae to the truth values $\{true, false\}$ and is defined on the structure of the derivation tree D in the following way:

$$\mathcal{A}\left[\!\left[\frac{\Delta,\ \Gamma\ \bigtriangledown_\Sigma\ f}{\Delta, \Gamma\ \rhd'_\Sigma\ \forall \alpha_1, \ldots, \alpha_n . A_1, \ldots, A_m \Rightarrow f}\ (ext.qual.)\right]\!\right]_{\nu,\eta} :=$$

$$\begin{cases} true & \text{if not } \models_{\mathcal{SA},\nu} \{A_1, \ldots, A_m\} \quad \text{or} \quad \mathcal{A}\left[\!\left[\Delta,\ \Gamma\ \bigtriangledown_\Sigma\ f\right]\!\right]_{\nu,\eta} = true \\ false & \text{otherwise} \end{cases}$$

□

As we can see, the meaning of an extended sort derivation does not differ from the meaning of a sort derivation. Nevertheless, we must check the well-definedness of the definition because the side conditions of the rules differ.

Proposition 4.3.14 *The interpretation function* $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ *is well-defined*

Let $D = \dfrac{\Delta, \Gamma\ \bigtriangledown_\Sigma\ f}{\Delta, \Gamma\ \rhd_\Sigma\ \forall \alpha_1, \ldots, \alpha_n . A_1, \ldots, A_m \Rightarrow f}\ (ext.qual.)$ be an extended sort derivation for a qualified formula. Let ν be a sort variable assignment and η be a variable assignment.

If $\models_{\mathcal{SA},\nu} \Delta$ and $\eta \models_{\mathcal{SA},\nu} \Gamma$
then $\mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta}$ yields a uniquely determined result in $\{true, false\}$.

□

Based on the meaning of an extended sort derivation, we now define what it means for a sort derivation to be satisfied in an algebra. The notion of satisfaction does not differ from the one defined in Def. 4.2.6.

Definition 4.3.10 *Satisfaction of an extended sort derivation*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let $D = \Delta, \emptyset \bigtriangledown'_\Sigma qf$ be an extended sort derivation for a closed qualified Σ-formula qf. Let η_0 be an arbitrary variable assignment. D is *satisfied* in $\mathcal{A}$, written

$$\mathcal{A} \models D \quad \text{iff} \quad \mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta_0} = \mathit{true} \quad \text{for every } \nu : \Phi \to \mathcal{U}.\ \models_{\mathcal{SA},\nu} \Delta$$

□

Next we extend those definitions and propositions given in Section 4.3.1 for non-qualified formula to qualified formula. We start with the definition of a semantic restriction order on extended sort derivations.

Definition 4.3.11 *Restriction Order*

Let $D_1{=}\Delta_1, \emptyset \bigtriangledown'_\Sigma qf$ and $D_2{=}\Delta_2, \emptyset \bigtriangledown'_\Sigma qf$ be two extended sort derivations for a closed qualified formula $qf \in \mathrm{QF}_\Sigma$. D_1 is called *more restricting* than D_2, written

$$D_1 \models D_2 \quad \text{iff} \quad \mathcal{A} \models D_1 \Rightarrow \mathcal{A} \models D_2 \ \text{ for all } \Sigma\text{-algebras } \mathcal{A}.$$

□

Then we extend the syntactic instance order to extended sort derivations.

Definition 4.3.12 *σ-instances of extended sort derivations*

Let $D_1 = \dfrac{\Delta_1, \Gamma_1 \bigtriangledown_\Sigma f}{\Delta_1, \Gamma_1 \rhd_\Sigma qf}\ (ext.qual.)$ and $D_2 = \dfrac{\Delta_2, \Gamma_2 \bigtriangledown_\Sigma f}{\Delta_2, \Gamma_2 \rhd_\Sigma qf}\ (ext.qual.)$
be two extended sort derivations for a qualified formula qf. Let $\sigma : \Phi \to \mathrm{T}_\Omega(\Phi)$ be a sort variable substitution. D_1 is called a *σ-instance* of D_2, written

$$D_1 \leq^\sigma D_2 \quad \text{iff} \quad \Delta_1, \Gamma_1 \bigtriangledown_\Sigma f \leq^\sigma \Delta_2, \Gamma_2 \bigtriangledown_\Sigma f$$

□

Definition 4.3.13 *Syntactic order on extended sort derivations*

Let $qf = \forall\overline{\alpha_n}.\overline{A_m} \Rightarrow f$ be a closed qualified formula. An extended sort derivation $D_2 = \Delta_2, \emptyset \bigtriangledown'_\Sigma qf$ is said to be *more general* than a sort derivation $D_1 = \Delta_1, \emptyset \bigtriangledown'_\Sigma qf$, written

$$
\begin{array}{lll}
D_1 \leq D_2 \quad \text{iff} \quad \exists \sigma : \Phi \to \mathrm{T}_\Omega(\Phi) & & \text{with} \\
\qquad \sigma(\alpha_i) = \alpha_i, 1 \leq i \leq n & & \text{and} \\
\qquad \Delta_1 \vdash_{SA} \sigma(\Delta_2) & & \text{and} \\
\qquad D_1 \leq^{\sigma} D_2 & &
\end{array}
$$

□

As in Prop. 4.3.5, we now show that the syntactic order defined above coincides with the semantic restriction order. Again, the proof is based on an auxiliary proposition showing that the interpretation of two extended sort derivations yields the same result if the derivations are in a σ-instance relation and the sort variable assignment reflects the substitution σ.

Proposition 4.3.15 *If an extended sort derivation is more general then it is more restricting*

Let $D_1{=}\Delta_1, \emptyset \bigtriangledown'_\Sigma qf$ and $D_2{=}\Delta_2, \emptyset \bigtriangledown'_\Sigma qf$ be two extended sort derivations for a closed qualified formula $qf = \forall\overline{\alpha_n}.\overline{A_m} \Rightarrow f$. If D_2 is more general than D_1 then D_2 is more restricting than D_1, i.e.

$$D_1 \leq D_2 \quad \Rightarrow \quad D_2 \models D_1$$

□

Proposition 4.3.16 *Interpretation equivalent extended sort derivations*

Let

$$D_1 = \frac{\Delta_1, \emptyset \bigtriangledown_\Sigma f}{\Delta_1, \emptyset \rhd_\Sigma qf}\ (ext.qual.) \quad \text{and} \quad D_2 = \frac{\Delta_2, \emptyset \bigtriangledown_\Sigma f}{\Delta_2, \emptyset \rhd_\Sigma qf}\ (ext.qual.)$$

be two extended sort derivations for a closed qualified formula $qf = \forall\alpha_1, \ldots, \alpha_n. A_1, \ldots, A_m \Rightarrow f$. Let σ be a sort variable substitution such that $\sigma(\alpha_i) = \alpha_i$, $1{\leq}i{\leq}n$ and $D_1 \leq^{\sigma} D_2$. Let ν and ν^σ be sort variable assignments such that $\models_{\mathcal{SA},\nu} \Delta_1$ and $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$. Let η be an arbitrary variable assignment.

$$\text{If} \quad \nu^\sigma(\alpha){=}\mathcal{SA}[\![\sigma(\alpha)]\!]_\nu \text{ for each } \alpha \in \Phi \quad \text{then} \quad \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} = \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta}$$

□

We now precisely define the concept of a principal extended sort derivation.

Definition 4.3.14 *Principal extended sort derivation*

Let $qf \in \mathrm{QF}_\Sigma$ be a closed qualified formula. A sort derivation $D_1 = \Delta_1, \emptyset \bigtriangledown'_\Sigma qf$ is called a *principal extended sort derivation* for qf, iff for each extended sort derivation $D_2 = \Delta_2, \emptyset \bigtriangledown'_\Sigma qf$ holds that $D_2 \leq D_1$. □

The next proposition establishes the second requirement for extended sort derivations. It does not hold for sort derivations defined in Def. 3.2.11. Only the change of the rule *(qualification)* made this proposition possible.

Proposition 4.3.17 *Existence of principal extended sort derivations*

Let $qf \in \mathrm{QF}_\Sigma$ be a closed qualified formula. If there exists an extended sort derivation for qf, then there exists a principal extended sort derivation for qf.

Proof: In Section 5.5 we give an algorithm computing an extended principal sort derivation for closed qualified formulae. □

For qualified formulae the principal extended sort derivation is not unique either. The reason lies again in different, but equivalent sort predicate assumptions. We can, however, show that all principal extended sort derivations are satisfaction equivalent.

Proposition 4.3.18 *Equivalent extended sort derivations are satisfaction equivalent*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let D_1 and D_2 be two extended sort derivations for a closed qualified formula.

$$\text{if} \quad D_1 \leq D_2 \text{ and } D_2 \leq D_1 \quad \text{then} \quad \mathcal{A} \models D_1 \Leftrightarrow \mathcal{A} \models D_2$$

□

Based on this result we can now define another satisfaction property for closed qualified formulae.

Definition 4.3.15 *P-Satisfaction*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let $qf \in \mathrm{QF}_\Sigma$ be a closed well-formed qualified Σ-formula, i.e. $qf \in \mathrm{SEN}_\Sigma$. Formula qf is *p-satisfied in* $\mathcal{A}$, written

$$\mathcal{A} \models_p qf \quad \text{iff} \quad \mathcal{A} \models D \text{ for some principal extended sort derivation } D \text{ for } qf$$

□

For non-qualified formula we have already shown in Prop. 4.3.10 that satisfaction is equivalent to p-satisfaction. Thus, we also included this proposition into our requirements for the new rule *(ext. qual.)*. However, the p-satisfaction concept based on extended sort derivations does not fulfil this requirement in general. We can only show the following weaker proposition.

Proposition 4.3.19 *P-Satisfaction implies satisfaction*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let $qf \in \mathrm{QF}_\Sigma$ be a closed well-formed qualified Σ-formula. Formula qf is satisfied in $\mathcal{A}$ if qf is p-satisfied in $\mathcal{A}$, i.e.

$$\mathcal{A} \models_p qf \quad \Rightarrow \quad \mathcal{A} \models qf$$

□

We cannot prove the proposition given above in reverse direction. The problem lies in the transition from sort derivations to extended sort derivations. We can prove that if an algebra satisfies every extended sort derivation for a qualified formula, the algebra also satisfies every sort derivation for this formula. We prove this in the proposition below by showing that for each sort derivation D we can find an extended sort derivation D' with the property that if $\mathcal{A} \models D'$ then $\mathcal{A} \models D$. This is easy to show, because every sort derivation is also an extended sort derivation if we replace the rule *(qualification)* by *(ext. qual.)*. Therefore, the set of sort derivations for a qualified formula form a subset of the set of extended sort derivations if we perform the replacement above. This was the method used to prove one direction of Prop. 4.3.11, showing the equivalence of well-formedness.

Proposition 4.3.20 *Existence of satisfaction implying extended sort derivations*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let $qf = \forall \overline{\alpha_n}.\overline{A_m} \Rightarrow f$ be a qualified formula. For each sort derivation $D = \Delta_1, \emptyset \bigtriangledown_\Sigma qf$ there exists an extended sort derivation D'=$\Delta_1, \emptyset \bigtriangledown'_\Sigma qf$ such that

$$\mathcal{A} \models D' \quad \Rightarrow \quad \mathcal{A} \models D$$

□

However, we cannot prove that, if an algebra satisfies every sort derivation for a qualified formula, the algebra also satisfies every extended sort derivation. The reason is the loose model property for sort specifications. Let us explain this fact by an example.

Example 4.3.3 *P-Satisfaction is not equivalent to satisfaction*

In Ex. 4.3.1 we examined the qualified formula $\emptyset \Rightarrow \forall$`x. p(x)`. With respect to the given signature we found two relevant sort derivations shown in Fig. 4.3 and Fig. 4.4, respectively. An algebra satisfies $\emptyset \Rightarrow \forall$`x. p(x)` if it satisfies both sort derivations. That means `p(x)` must hold for all elements of sort `Bool` as well as for all elements of sort `Nat`. In Fig. 4.6 corresponding to Ex. 4.3.2 we showed a more abstract extended sort derivation for this formula. In an algebra satisfying this extended sort derivation `p(x)` must hold for all elements of all sorts satisfying the sort predicate `EQ`. Clearly, the sorts `Nat` and `Bool` satisfy `EQ`. But there may be more sorts in the algebra satisfying `EQ`, though not explicitly specified in the sort specification. But then `p(x)` must also hold for all elements of these sorts, not only for `Nat` and `Bool`. Thus, both notions of satisfaction are not equivalent: p-satisfaction is more rigorous than satisfaction. □

Restricting models to term-generated sort algebras is not enough to achieve equivalence of both satisfaction properties. We must always choose the initial sort algebra as the model, i.e. the *least Herbrand model*. If $P(\tau_1, \ldots, \tau_n)$ holds in the initial model, then

$P(\tau_1, \ldots, \tau_n)$ holds in every model of a particular sort specification. The initial model concept corresponds to the so-called *closed world assumption* that can be found in many logical languages (see [Llo87] for a detailed discussion). This assumption is used to infer negative information that cannot be specified by Horn clauses. Therefore, a special inference rule is used: if a ground atomic sort formula A cannot be inferred from the Horn clause specification then $\neg A$ is inferred. This rule, however, is only correct in the initial model because here the semantic consequence relation is strongly connected to the syntactic entailment relation. We can show the following stronger completeness property for the initial model I of a sort specification S:

$$\Delta \models_I C \quad \Rightarrow \quad \Delta \vdash_{SA} C$$

Together with the soundness property from Prop. 4.1.3 both relations can be shown to be equivalent, i.e.

$$\Delta \models_I C \quad \Leftrightarrow \quad \Delta \vdash_{SA} C$$

By choosing the initial model for the sort specification both satisfaction concepts can be proven equivalent.

We do not go into further details because initial models have a serious drawback: an enrichment or refinement of an existing sort specification cannot be handled semantically by simple model set inclusion. The semantic treatment of these relations between specifications becomes more complicated than in the loose case.

Note that we only lose the equivalence of both satisfaction concepts because of the transition from derivations to extended derivations. In particular, if we already started with the extended sort inference rule in Chapter 4.2, both satisfaction concepts would be equivalent for qualified formulae, too. To show this formally we define a further satisfaction property.

Definition 4.3.16 *X-Satisfaction*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let $qf \in \mathrm{QF}_\Sigma$ be a closed well-formed qualified Σ-formula, i.e. $qf \in \mathrm{SEN}_\Sigma$. Formula qf is *x-satisfied in* $\mathcal{A}$, written

$$\mathcal{A} \models_x qf \quad \text{iff} \quad \begin{array}{l} \mathcal{A} \models D \text{ for each extended sort derivation } D \text{ that ends with} \\ \Delta, \emptyset \rhd qf \text{ for some sort predicate assumption } \Delta \end{array}$$

□

The following proposition establishes the equivalence of x-satisfaction and p-satisfaction.

Proposition 4.3.21 *X-Satisfaction is equivalent to p-satisfaction*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let $qf \in \mathrm{QF}_\Sigma$ be a closed well-formed qualified Σ-formula. Formula qf is x-satisfied in $\mathcal{A}$ iff qf is p-satisfied in $\mathcal{A}$, i.e.

$$\mathcal{A} \models_x qf \quad \Leftrightarrow \quad \mathcal{A} \models_p qf$$

□

Based on the results of this chapter and those of Section 4.3.1 we can now define a new notion of model for polymorphic specifications consisting of non-qualified and qualified formulae. We show that the new notion is more rigorous than the one defined in Def. 4.2.8.

Definition 4.3.17 *P-Model*

Let $PS = (\Sigma, A)$ be a polymorphic specification. A polymorphic Σ-algebra $\mathcal{A}$ is called a *p-model* of PS iff

$$\forall f \in A.\ \mathcal{A} \models_p f$$

□

Proposition 4.3.22 *The p-model property is more rigorous than the model property*

Let $PS = (\Sigma, A)$ be a polymorphic specification. Let $\mathcal{A}$ be a polymorphic Σ-algebra.

$$\text{If} \quad \mathcal{A} \text{ is a p-model of } PS \quad \text{then} \quad \mathcal{A} \text{ is a model of } PS$$

□

The p-model property, however, is only more rigorous because we replaced inference rule (*qualification*) by the more liberal rule (*ext. qual.*). To show this formally, we define a further notion of model again based on all sort derivations. In contrast to the notion defined in Def. 4.2.8, we use the x-satisfaction property for qualified formulae.

Definition 4.3.18 *X-Model*

Let $PS = (\Sigma, A)$ be a polymorphic specification. A polymorphic Σ-algebra $\mathcal{A}$ is called an *x-model* of PS iff

$$\begin{aligned} \forall f \in A. \quad f \in \mathrm{PF}_\Sigma(\Phi) \quad &\Rightarrow \quad \mathcal{A} \models f \\ f \in \mathrm{QF}_\Sigma \quad &\Rightarrow \quad \mathcal{A} \models_x f \end{aligned}$$

□

The following proposition establishes the p-model property to be equivalent to the x-model property.

Proposition 4.3.23 *The p-model property is equivalent to the x-model property*

Let $PS = (\Sigma, A)$ be a polymorphic specification. Let $\mathcal{A}$ be a polymorphic Σ-algebra.

$$\mathcal{A} \text{ is a p-model of } PS \quad \text{iff} \quad \mathcal{A} \text{ is an x-model of } PS$$

□

The last proposition is of major importance. It shows that the semantics of a specification defined by its principal sort derivation coincides with the semantics defined by all possible sort derivations. As a consequence, we can use a principal sort derivation to derive theorems of a polymorphic specification, without losing completeness.

Moreover, we can proceed as follows to define the semantics of a polymorphic specification: We give an algorithm to compute a principal sort derivation for a given specification. Afterwards, we translate the principal sort derivation to a fully sorted specification of an extended specification language. This language must in addition provide a mechanism to instantiate polymorphic identifiers explicitly. It is easy to define a semantics for the extended specification language because this specification contains all sort information needed. We then can define the semantics of a polymorphic specification to be the semantics of its translated principal sort derivation.

This method was used to define the semantics of the specification language SPECTRUM. The sort system of SPECTRUM is only a special case of our polymorphic sort system. Hence, Prop. 4.3.23 can also be applied to that sort system. As a consequence, we now know that this method was adequate because semantically all possible sort derivations of a specification have been taken into consideration.

Chapter 5

Sort Inference for Polymorphic Specifications

In Chapter 3 we defined the syntax of our specification language. Besides a context-free syntax we gave inference rules describing the sort system of our language. A formula is only well-sorted if there exists a sort derivation for the formula using these inference rules. To test whether a given formula is well-sorted we have to construct such a sort derivation. The rules in Def. 3.2.7 and 3.2.9, however, were only designed to *define* the set of well-formed terms and formulae. They cannot be directly executed to *test* the well-formedness of a given formula. The entailment relation $\vdash$ was used to impose further restrictions on well-formed formulae. The rules defining the relation $\vdash$ are not suited for direct execution, either.

In this chapter we investigate the implementation of our sort system. In particular we give an algorithm testing whether a given formula is well-formed. The algorithm is split into two parts, a sort inference algorithm, and a resolution algorithm.

1. The sort inference algorithm implements the calculi defining the relations $\rhd$ for terms and formulae. It is based upon an algorithm by M. P. Jones extending Milner's sort inference algorithm W to support qualified sorts.

2. The resolution algorithm implements the calculus defining the entailment relation $\vdash$ for sort clauses. The algorithm is based on SLD-resolution.

Section 5.1 gives an introduction to unification, which is a central component of both the sort inference algorithm and the resolution algorithm.

In Section 5.2 we present sort inference algorithms for terms, non-qualified formulae, and qualified formulae. The algorithms not only test whether a given term or formula is

well-formed but in addition yield a derivation tree for the terms and formulae. We show the soundness and completeness of the algorithms with respect to the inference calculi.

In Section 5.3 the implementation of the entailment calculus is investigated. Based on unification we present a resolution calculus and show its soundness and completeness with respect to the entailment calculus. We give a resolution refutation algorithm implementing the resolution calculus and show its soundness and completeness.

Based on the results from Section 5.2 and Section 5.3 we present an algorithm in Section 5.4 testing the well-formedness of polymorphic specifications. We show that this algorithm is sound and complete with respect to the concept of well-formedness defined in Section 3.

In Section 5.5 we show that the sort derivations computed by the sort inference algorithms are principal sort derivations.

In Section 5.6 the semi-decidability of the sort system is discussed. Finally, Section 5.7 describes a concrete implementation of the sort system.

5.1 Unification

The goal of unification is to unify two (or more) expressions, i.e. to make the expressions syntactically identical. This can be achieved by some consistent substitution of expressions for variables.

There are algorithms for deciding whether two expressions are unifiable. The original one was given by Robinson in 1965 [Rob65]. His original motivation for describing unification was its role in resolution theorem proving. His work provided the conceptual basis for logic programming languages.

Another important use of unification can be found in sort inference. Hindley [Hin69] was the first one to recognize this. He used the results of Robinson to show the existence of principal types in combinatory logic. Milner [Mil78] rediscovered many of his ideas in the context of functional programming languages and used unification in the type inference algorithm for ML. Meanwhile specific kinds of unification theories were used in different type inference algorithms. For example, in [NS91] a type inference algorithm for a Haskell-like language based on order-sorted unification was presented. In our framework both the sort inference algorithm and the resolution algorithm are based on unification of sort terms. Sort terms themselves are untyped, i.e. they do not belong to some kind of meta sort. Furthermore, sort terms are first-order terms. There is no λ-abstraction in sort terms, and all sort variables are first order variables, i.e. they can only represent basic sorts. Thus, we can use the simple unification theory developed by Robinson.

The core languages used in [Mil78] and [Jon92] do not allow one to write explicit sort annotations. In the framework of axiomatic specification language, however, explicit sort annotations are essential. These annotations may contain sort variables. Those variables may not be specialized by the sort inference or the resolution algorithm. They are treated as constants. Conversely, only the sort variables introduced by the algorithm may be specialized by unification. Therefore, we introduce a set of sort variables $UV \subset \Phi$ called *unification variables*. We assume that UV is disjunct to the set of sort variables explicitly used in formulae. Unification is then defined with respect to a set of unification variables UV in the following way:

Definition 5.1.1 *Unifier with respect to a set of variables UV*

Let τ_1 and τ_2 be sort terms from $\mathrm{T}_\Omega(\Phi)$, and let $UV \subset \Phi$ be a set of unification variables. A substitution σ is called a *unifier of τ_1 and τ_2 w.r.t. UV* iff

$$\sigma(\tau_1) = \sigma(\tau_2) \text{ and } \forall \alpha \in \Phi \setminus UV. \quad \sigma(\alpha) = \alpha$$

σ is called a *most general unifier for τ_1 and τ_2 w.r.t. UV* iff for every unifier ρ of τ_1 and τ_2 there exists a substitution σ' such that $\rho = \sigma'\sigma$. A unifier of two atomic sort formulae A_1 and A_2 from $\mathrm{AF}_\Omega(\Phi)$ is defined analogously. □

The following proposition shown by Robinson [Rob65] establishes the existence of a unification algorithm.

Proposition 5.1.1 *Unification*

There is a unification algorithm whose input is a pair of sort terms τ_1 and τ_2, respectively a pair of atomic sort formula A_1 and A_2, such that either:

- the algorithm fails and there are no unifiers for τ_1 and τ_2, respectively A_1 and A_2 or
- the algorithm succeeds with the most general unifier as result.

In the following, such a unification algorithm is denoted by *mgu*. □

5.2 Sort Inference

The algorithm presented in this section originates from the sort inference algorithm W (well-typing algorithm) by Milner [Mil78]. Originally W was introduced to prove the well-typedness of terms of a simple functional language providing parametric polymorphism.

It was first used in the LCF meta language ML. Since then variants and extensions of W were used in numerous type checkers for different functional languages. Jones [Jon92] extended the algorithm to support qualified types. In addition to W his algorithm computes a sort predicate assumption necessary for well typedness. This is done by collecting the qualifying sort predicates of the applied polymorphic functions. Thus, the algorithm is independent of the underlying entailment relation $\vdash$. Proofs w.r.t. the entailment relation $\vdash$, necessary to establish well-formedness, can be performed in a separate step.

The algorithms described in this section are extensions of Jones algorithm W. Because we strictly separate the formula layer from the term layer, we use different sort inference algorithms for terms and formulae. The sort inference algorithm for terms is described in Section 5.2.1. In Section 5.2.2, and Section 5.2.3, respectively, we discuss sort inference for non-qualified, and qualified formulae, respectively.

5.2.1 Sort Inference for Terms

This section describes sort inference for terms. In contrast to [Jon92] our term language includes explicitly sorted λ-abstraction and sort constrained terms. Thus, we add two cases to the algorithm W presented in [Jon92] to support these explicit sort annotations. Furthermore, we use two different assumptions to distinguish variables from identifiers of the signature. Usually algorithm W is only used to test the well-sortedness of a term. If a term is well-sorted, W results in a valid sort term for the given term as well as the necessary sort predicate assumption. However, we will also use the algorithm to define the semantics of polymorphic specifications. Thus, in addition, W yields a sort derivation for the given term. We will later prove this computed derivation tree to be a principal derivation for the term.

Definition 5.2.1 *Sort inference algorithm W*

We describe the algorithm in an almost[1] functional style as the function $W(F, \Gamma, e) = (\Delta, \sigma, \tau, D)$ defined in Fig. 5.1 where[2]:

- $F \in <id> \rightarrow \mathrm{T}_\Omega$ are object identifiers declared in a signature
- $\Gamma \in <var> \rightarrow \mathrm{T}_\Omega(\Phi)$ is a variable assumption
- $e \in \mathrm{PT}_\Omega(\Phi)$ is the pre-term to be checked

[1] A full functional definition would require a formal treatment of "new variable from UV".

[2] The extended substitution σ_Δ is defined in Def. 4.3.7.

- $\Delta \subseteq \mathrm{AF}_\Omega(\Phi)$ is the computed sort predicate assumption
- $\sigma \in \Phi \to \mathrm{T}_\Omega(\Phi)$ is a variable substitution
- $\tau \in \mathrm{T}_\Omega(\Phi)$ is the inferred sort for e
- D is a sort derivation for e

□

The algorithm W is recursively defined on the term structure. The judgement calculus given in 3.2.7 being syntax-directed, each case of function W corresponds to exactly one calculus rule. Algorithm W fails if

- F does not contain an object identifier occurring in the term e, or
- Γ does not contain a variable occurring free in the term e, or
- a call of the unification function *mgu* fails.

We now show the soundness of W with respect to the term judgement $\rhd$. We must prove that there always exists a sort derivation for a given term if algorithm W terminates successfully for the term. Furthermore, we must show the computed derivation tree to be indeed a sort derivation for the term.

Proposition 5.2.1 *Sort inference algorithm W is sound w.r.t. $\rhd$*

Let $\Sigma = (S, P, F)$ be a polymorphic signature. Let Γ be a variable assumption, and let e be a pre-term. If

$$W(F, \Gamma, e) = (\Delta, \sigma, \tau, D)$$

then

1. $\forall \alpha \notin UV.\ \sigma(\alpha) = \alpha$
2. $\Delta, \sigma(\Gamma) \rhd_\Sigma e :: \tau$
3. D is a sort derivation ending with $\Delta, \sigma(\Gamma) \rhd_\Sigma e :: \tau$

□

Note that successful termination of algorithm W does not prove the well-formedness of the examined term. In addition we must show that $\emptyset \vdash_{SA} \Delta \cap \mathrm{AF}_\Omega(\emptyset)$ (see Def. 3.2.8). In Section 5.3.2 we will present an algorithm testing the membership in the relation $\vdash$.

Apart from soundness we also show the completeness of W with respect to the term judgement $\rhd$. We prove that algorithm W always terminates successfully for a given term if there exists a sort derivation for the term. Furthermore, the sort derivation is always an instance of the computed derivation tree. Formally:

$$W(F,\Gamma,x) = (\emptyset,\varepsilon,\Gamma(x),\ \frac{}{\emptyset,\Gamma\rhd_\Sigma\ x::\Gamma(x)}\ (var))$$

$$W(F,\Gamma,c) = (\emptyset,\varepsilon,F(c),\ \frac{}{\emptyset,\Gamma\rhd_\Sigma\ c::F(c)}\ (id))$$
where $F(c)\in T_\Omega(\emptyset)$

$$W(F,\Gamma,c) = ([\overline{\beta_n/\alpha_n}]\overline{A_m},\varepsilon,[\overline{\beta_n/\alpha_n}]\tau,\ \frac{}{[\overline{\beta_n/\alpha_n}]\overline{A_m},\Gamma\rhd_\Sigma\ c::[\overline{\beta_n/\alpha_n}]\tau}\ (polyid))$$
where $F(c)=\Pi\overline{\alpha_n}.\overline{A_m}\Rightarrow\tau$
$\overline{\beta_n}$ are new variables from UV

$$W(F,\Gamma,e_1\,e_2) = (\Delta,u\sigma_2\sigma_1,u(\alpha),\ \frac{(u\sigma_2)_{u(\Delta_2)}(D_1)\ u_{u\sigma_2(\Delta_1)}(D_2)}{\Delta,u\sigma_2\sigma_1(\Gamma)\rhd_\Sigma\ e_1e_2::u(\alpha)}\ (\to E))$$
where $(\Delta_1,\sigma_1,\tau_1,D_1)=W(F,\Gamma,e_1)$
$(\Delta_2,\sigma_2,\tau_2,D_2)=W(F,\sigma_1(\Gamma),e_2)$
$u=mgu(\sigma_2(\tau_1),\tau_2\to\alpha)$
α is a new variable from UV
$\Delta=u(\sigma_2(\Delta_1)\cup\Delta_2)$

$$W(F,\Gamma,\lambda x.e_1) = (\Delta_1,\sigma_1,\sigma_1(\alpha)\to\tau_1,\ \frac{D_1}{\Delta_1,\sigma_1(\Gamma)\ \rhd_\Sigma\ \lambda x.e_1::\sigma_1(\alpha)\to\tau_1}\ (\to I_u))$$
where $(\Delta_1,\sigma_1,\tau_1,D_1)=W(F,\Gamma.x{:}\alpha,e_1)$
α is a new variable from UV

$$W(F,\Gamma,\lambda x{:}\tau_2.e_1) = (\Delta_1,\sigma_1,\tau_2\to\tau_1,\ \frac{D_1}{\Delta_1,\sigma_1(\Gamma)\ \rhd_\Sigma\ \lambda x{:}\tau_2.e_1::\tau_2\to\tau_1}\ (\to I_s))$$
where $(\Delta_1,\sigma_1,\tau_1,D_1)=W(F,\Gamma.x{:}\tau_2,e_1)$

$$W(F,\Gamma,e_1{:}\tau) = (u(\Delta_1),u\sigma_1,\tau,\ \frac{u_\emptyset(D_1)}{u(\Delta_1),u\sigma_1(\Gamma)\ \rhd_\Sigma\ e_1{:}\tau::\tau}\ (constrained))$$
where $(\Delta_1,\sigma_1,\tau_1,D_1)=W(F,\Gamma,e_1)$
$u=mgu(\tau_1,\tau)$

Figure 5.1: Sort inference algorithm W

Proposition 5.2.2 *Sort inference algorithm W is complete w.r.t.* $\rhd$

Let $\Sigma = ((\Omega, SA), P, F)$ be a polymorphic signature. Let Γ be a variable assumption, and let e be a pre-term. If there exists a sort derivation $\Delta', \sigma'(\Gamma) \bigtriangledown_\Sigma e :: \tau'$ such that $\forall \alpha \notin UV : \sigma'(\alpha) = \alpha$ then

$$W(F, \Gamma, e) = (\Delta, \sigma, \tau, (\Delta, \sigma(\Gamma) \rhd_\Sigma e :: \tau))$$

and there exists a substitution ρ such that

$$\begin{gathered} \forall \alpha \notin UV . \rho(\alpha) = \alpha \\ \rho(\Delta) \subseteq \Delta' \\ \sigma'(\Gamma) = \rho\sigma(\Gamma) \\ \Delta', \sigma'(\Gamma) \bigtriangledown_\Sigma e :: \tau' \leq^\rho \Delta, \sigma(\Gamma) \bigtriangledown_\Sigma e :: \tau \end{gathered}$$

□

5.2.2 Sort Inference for Non-Qualified Formulae

In this section we extend the sort inference algorithm to non-qualified formulae. The algorithm is based on function W testing the well-formedness of a term. We need an additional parameter for the predicate context. Furthermore, the inference of a sort lapses because formulae always yield a truth value as result.

Definition 5.2.2 *Sort inference algorithm V*

Again we describe the algorithm in an almost functional style as the function

$$V(P, F, \Gamma, f) = (\Delta, \sigma, D)$$

defined in Fig. 5.2 where[3]:

- $P \in <id> \to \mathrm{T}_\Omega$ are predicate identifiers declared in a signature
- $f \in \mathrm{PF}_\Omega(\Phi)$ is the formula to be checked
- $F, \Gamma, \Delta, \sigma$ and D have the same meaning as in Def. 5.2.1.

□

Algorithm V is recursively defined on the structure of formulae. Because the judgement calculus given in Def. 3.2.9 is syntax-directed each case of function V corresponds to exactly one calculus rule. Algorithm V fails if

[3]The extended substitution σ_Δ is defined in Def. 4.3.8.

$$
\begin{array}{lcl}
V(P,F,\Gamma,p) & = & (\emptyset, \varepsilon, \ \dfrac{}{\emptyset, \Gamma \rhd_\Sigma p}\ (const)) \\
 & & \textbf{where} \quad P(p) = \varepsilon \\
V(P,F,\Gamma,p\ e) & = & (u(\Delta), u\sigma, \ \dfrac{u_\emptyset(D)}{u(\Delta), u(\Gamma) \rhd_\Sigma p\ e}\ (appl)) \\
 & & \textbf{where} \quad P(p) \in \mathrm{T}_\Omega(\emptyset) \\
 & & \qquad (\Delta, \sigma, \tau, D) = W(F,\Gamma,e) \\
 & & \qquad u = mgu(P(p), \tau) \\
V(P,F,\Gamma,p\ e) & = & (\Delta, u\sigma, \dfrac{u_{u[\overline{\beta_n/\alpha_n}]\overline{A_m}}(D)}{\Delta, u(\Gamma) \rhd_\Sigma p\ e}\ (pappl)) \\
 & & \textbf{where} \quad P(p) = \Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau \\
 & & \qquad (\Delta', \sigma, \tau', D) = W(F,\Gamma,e) \\
 & & \qquad u = mgu(\overline{[\beta_n/\alpha_n]}\tau, \tau') \\
 & & \qquad \Delta = u(\overline{[\beta_n/\alpha_n]}\overline{A_m} \cup \Delta') \\
 & & \qquad \overline{\beta_n} \text{ are new variables from } UV \\
V(P,F,\Gamma,\forall x.f) & = & (\Delta, \sigma, \ \dfrac{D}{\Delta, \sigma(\Gamma) \rhd_\Sigma \forall x.f}\ (univ_u)) \\
 & & \textbf{where} \quad (\Delta, \sigma, D) = V(P,F,\Gamma.x{:}\alpha, f) \\
 & & \qquad \alpha \text{ is a new variable from } UV \\
V(P,F,\Gamma,\forall x{:}\tau.f) & = & (\Delta, \sigma, \ \dfrac{D}{\Delta, \sigma(\Gamma) \rhd_\Sigma \forall x{:}\tau.f}\ (univ_s)) \\
 & & \textbf{where} \quad (\Delta, \sigma, D) = V(P,F,\Gamma.x{:}\tau, f) \\
V(P,F,\Gamma,\neg f) & = & (\Delta, \sigma, \ \dfrac{D}{\Delta, \sigma(\Gamma) \rhd_\Sigma \neg f}\ (not)) \\
 & & \textbf{where} \quad (\Delta, \sigma, D) = V(P,F,\Gamma,f) \\
V(P,F,\Gamma,f_1 \vee f_2) & = & (\sigma_2(\Delta_1) \cup \Delta_2, \sigma_2\sigma_1, \ \dfrac{\sigma_{2_{\Delta_2}}(D_1)\ \varepsilon_{\sigma_2(\Delta_1)}(D_2)}{\sigma_2(\Delta_1) \cup \Delta_2, \sigma_2\sigma_1(\Gamma) \rhd f_1 \vee f_2}\ (or)) \\
 & & \textbf{where} \quad (\Delta_1, \sigma_1, D_1) = V(P,F,\Gamma,f_1) \\
 & & \qquad (\Delta_2, \sigma_2, D_2) = V(P,F,\sigma_1(\Gamma), f_2)
\end{array}
$$

Figure 5.2: Sort inference algorithm V

- P does not contain a predicate identifier occurring in the formula f or
- a non-constant predicate identifier is not applied to a term or
- a constant predicate identifier is applied to a term or
- a call of the unification function *mgu* fails or
- a call of function W fails.

The following proposition establishes the soundness of algorithm V with respect to the formula judgement $\rhd$.

Proposition 5.2.3 *Sort inference algorithm V is sound w.r.t. $\rhd$*

Let $\Sigma = (S, P, F)$ be a polymorphic signature. Let Γ be a variable assumption, and let f be a pre-formula. If

$$V(P, F, \Gamma, f) = (\Delta, \sigma, D)$$

then

$$\Delta, \sigma(\Gamma) \rhd_\Sigma f$$

and D is a sort derivation ending with $\Delta, \sigma(\Gamma) \rhd_\Sigma f$.

Proof: We omit the proof because it is only a variation of the soundness proof of algorithm W. From the point of view of sort inference there is no fundamental difference between terms and formulae. Predicates are handled like functions and the universal quantifier is handled like the λ-abstraction. Furthermore, the logical connectives $\neg$ and $\vee$ can be seen as special identifiers applied to formulae. □

Again, successful termination of algorithm V does not prove the well-formedness of the examined formula. It remains to prove that $\emptyset \vdash_{SA} \Delta \cap \mathrm{AF}_\Omega(\emptyset)$. We now extend the completeness proposition of algorithm W to V.

Proposition 5.2.4 *Sort inference algorithm V is complete w.r.t. $\rhd$*

Let $\Sigma = ((\Omega, SA), P, F)$ be a polymorphic signature. Let Γ be a variable assumption, and let f be a pre-formula. If there exists a sort derivation $\Delta', \sigma'(\Gamma) \bigtriangledown_\Sigma f$ such that $\forall \alpha \notin UV.\ \sigma'(\alpha) = \alpha$ then

$$V(P, F, \Gamma, f) = (\Delta, \sigma, (\Delta, \sigma(\Gamma) \bigtriangledown_\Sigma f))$$

and there exists a substitution ρ such that

$$\forall \alpha \notin UV . \rho(\alpha) = \alpha$$
$$\rho(\Delta) \subseteq \Delta'$$
$$\sigma'(\Gamma) = \rho\sigma(\Gamma)$$
$$\Delta', \sigma'(\Gamma) \bigtriangledown_\Sigma f \leq^\rho \Delta, \sigma(\Gamma) \bigtriangledown_\Sigma f$$

Proof: For the same reason as in Prop. 5.2.3 we omit the proof in this thesis. □

5.2.3 Sort Inference for Qualified Formulae

A qualified formula consists of a formula and a sort predicate context. In this section we extend the sort inference algorithm V to handle this explicit context. In earlier chapters we defined two different judgements for qualified formulae. Both judgements were proven to be well-formedness equivalent. However, because only the extended judgement $\rhd'$ enables principal sort derivations and because we want to use our sort inference algorithm to define the semantics of a specification, we give an implementation of the extended judgement $\rhd'$. For a qualified formula $\forall\overline{\alpha_n}.\overline{A_m} \Rightarrow f$ it allows any assumption Δ to show the well-formedness of f for which we can prove:

$$\exists\sigma : \Phi \to \mathrm{T}_\Omega(\Phi) \text{ with } \sigma(\alpha_i) = \alpha_i, 1 \leq i \leq n \text{ such that}$$
$$\{\overline{A_m}\} \vdash_{SA} \sigma(\Delta)$$

In Section 5.3.2 we present a function *resolve* that computes such a substitution if there exists any. Based on this function we can define the sort inference algorithm for qualified formulae in the following way:

Definition 5.2.3 *Sort inference algorithm U*

The algorithm is described as function $U(SA, P, F, qf) = (\Delta, D)$ in Fig. 5.3 where:

- $SA \subset \mathrm{C}_\Omega(\Phi)$ is a set of sort axioms from a signature
- $qf \in \mathrm{QF}_\Sigma$ is the qualified formula to be checked
- P, F, Δ and D have the same meaning as in Def. 5.2.2

□

$$U(SA, P, F, \forall\overline{\alpha}_n.\overline{A_m} \Rightarrow f) = (\Delta, D)$$
$$\textbf{where} \quad (\Delta, \sigma, D_1) = V(P, F, \emptyset, f)$$
$$\sigma' = resolve(SA, \{\overline{A_m}\}, \Delta)$$
$$D = \frac{D_1}{\Delta, \emptyset \rhd'_\Sigma \forall\overline{\alpha_n}.\overline{A_m} \Rightarrow f}\ (ext.qual.)$$

Figure 5.3: Sort inference algorithm U

The following propositions establish the soundness and completeness of algorithm U with respect to the extended qualified formula judgement $\rhd'$.

Proposition 5.2.5 *Sort inference algorithm U is sound w.r.t.* $\rhd'$

Let $\Sigma = ((\Omega, SA), P, F)$ be a polymorphic signature, and let qf be a qualified formula.

$$\text{If} \quad U(SA, P, F, qf) = (\Delta, D) \quad \text{then} \quad \Delta, \emptyset \ \rhd'_\Sigma \ qf$$

and D is a sort derivation ending with $\Delta, \emptyset \ \rhd'_\Sigma \ qf$. □

Proposition 5.2.6 *Sort inference algorithm U is complete w.r.t.* $\rhd'_\Sigma$

Let $\Sigma = ((\Omega, SA), P, F)$ be a polymorphic signature, and let qf be a qualified formula. If there exists a sort derivation $\Delta', \emptyset \ \triangledown'_\Sigma \ qf$ then

1. $U(SA, P, F, qf) = (\Delta, (\Delta, \emptyset \ \triangledown'_\Sigma \ qf))$ and
2. $\Delta', \emptyset \ \triangledown'_\Sigma \ qf \leq \Delta, \emptyset \ \triangledown'_\Sigma \ qf$

□

5.3 Resolution

To guarantee the well-formedness of a formula (or term) we must, after successful termination of V (or W), show that $\emptyset \vdash_{SA} \Delta \cap \mathrm{AF}_\Omega(\emptyset)$. As we saw in Section 5.2.3 showing the well-formedness of qualified formulae requires an even more complicated entailment proof. The calculus rules defining the relation $\vdash$, however, are not suited for direct execution. This section is concerned with the algorithmic treatment of the relation $\vdash$. We omit all proofs since they can be found in the literature on logic programming, e.g. [Llo87, Pad89]. In Section 5.3.1 we present a resolution calculus and Section 5.3.2 discusses the execution of this calculus.

5.3.1 Resolution Calculus

Derivations via the entailment calculus defined in Def. 3.1.11 proceed bottom-up from the axioms and assumptions to the goal to be proved. From an operational point of view, this approach leads to a completely intractable search space. Therefore, the calculus is not appropriate to execute proofs automatically. For this purpose one should work top-down from the goal to be proved by applying sort clauses backward. The resolution principle due to Robinson [Rob65] embodies such an approach.

This principle can be used to prove the existence of a substitution σ for a set of atomic formulae $\{A_1, \ldots, A_m\}$ such that $\sigma(A_i), 1 \leq i \leq m$ is a logical consequence of some given set of clauses. From a logical point of view it must be shown

$$\exists \alpha_1, \ldots, \alpha_n. \ A_1 \wedge \ldots \wedge A_m$$

where $\overline{\alpha_n}$ are all variables occurring in the atomic formulae $\overline{A_m}$. To prove this the resolution procedure performs a refutation proof, i.e.

$$\neg\exists\alpha_1,\ldots,\alpha_n.A_1 \wedge \ldots \wedge A_m \Leftrightarrow \forall\alpha_1,\ldots,\alpha_n.\neg A_1 \vee \ldots \vee \neg A_m \vee \mathit{false}$$

is assumed. The resolution procedure tries to refute this assumption, called the *goal*, by eliminating all $\neg A_i$. If it is successful it has proved the existence of a substitution σ such that $\sigma(A_i)$, $1 \leq i \leq m$ is a logical consequence of some given clauses. Moreover, the resolution procedure is constructive, i.e. it yields such a substitution as result. Conversely, if there exists no refutation then it has been proved that there exists no substitution σ such that $\sigma(A_i)$ is a logical consequence of the clauses.

There are many different kinds of resolution inference rules used to find a refutation. We use an extension of the well-known SLD-resolution rule. SLD-resolution stands for linear resolution with selection function for definite clauses. A clause is called *definite* if the head consists of exactly one atomic formula. This is correct for sort clauses. Informally the SLD-resolution rule selects one atom from the goal, unifies it with the head of the clause and replaces that atom by the instantiated body of the clause. Thus, the SLD-resolution rule is an inversion of the rule (*mp*) given in Def. 3.1.11.

The rule (*mp*) allows an arbitrary instantiation of the sort clauses from the sort specification. The sort variables occurring in these clauses are implicitly universally quantified. To avoid unintentional bindings the resolution rule must therefore replace all sort variables in a sort clause by new sort variables before unifying the head with an atom from the goal. In the definition below we use a function *new* to carry out this substitution.

In addition to the usual SLD-resolution rule we need a further resolution rule to handle the rule (*assumption*) of the calculus defining the entailment relation $\vdash$. This rule does not instantiate the atomic sort formulae from the assumption. The sort variables occurring in the assumption are not implicitly universally quantified. Therefore, we do not need the function *new* in the corresponding resolution rule. The following resolution calculus incorporates both resolution rules. Note that we diverge from classical presentations by not restricting the unifier in both rules to the most general one.

Definition 5.3.1 *Resolution inference relation* $\vdash^{\mathrm{R}}$

A resolution inference relation $\vdash^{\mathrm{R}} \subseteq ((\mathcal{P}(\mathrm{AF}_\Omega(\Phi)) \times (UV \to \mathrm{T}_\Omega(\Phi))) \times \mathcal{P}(C_\Omega)(\Phi)) \times \mathcal{P}(\mathrm{AF}_\Omega(\Phi)) \times (\mathcal{P}(\mathrm{AF}_\Omega(\Phi)) \times (UV \to \mathrm{T}_\Omega(\Phi)))$ is a set of tuples $((G,\sigma), SA, \Delta, (G',\sigma'))$ where:

- SA and Δ have the same meaning as in Def. 3.1.11
- G and G' are sets of atomic sort formulae called *goals*

- σ and σ' are sort variable substitutions defined on unification variables.

We write $(G,\sigma)\vdash^{\mathrm{R}}_{SA,\Delta}(G',\sigma')$ instead of $((G,\sigma),SA,\Delta,(G',\sigma'))\in\vdash^{\mathrm{R}}$ and read it as "(G,σ) is resolved into (G',σ') with respect to SA and Δ". $(G,\sigma)\vdash^{\mathrm{R}}_{SA,\Delta}(G',\sigma')$ iff there exists a finite proof sequence, called *resolution derivation*, using the inference rules below that starts with (G,σ) and ends with (G',σ').

$$\frac{(G\cup\{A\},\sigma)}{(u(G),u\sigma)}\ (ASS)\begin{cases} P\in\Delta \\ \exists u\in UV\to \mathrm{T}_\Omega(\Phi).u(A)=P\end{cases}$$

$$\frac{(G\cup\{A\},\sigma)}{(u(G\cup\{B_1,\ldots,B_n\}),u\sigma)}\ (SLD)\begin{cases} C\in SA \\ B_1,\ldots,B_n\Rightarrow H=new(C) \\ \exists u\in UV\to \mathrm{T}_\Omega(\Phi).u(A)=u(H)\end{cases}$$

In resolution terminology, A is called the *selected atom* and $u(G)$ respectively $u(G\cup\{B_1,\ldots,B_n\})$ is called *resolvent*. If $(G,\varepsilon)\vdash^{\mathrm{R}}_{SA,\Delta}(\emptyset,\sigma)$ then G is called *resolution refutable (with respect to SA and Δ)*. A corresponding resolution derivation is called a *resolution refutation of G (with respect to SA and Δ)*. The substitution σ is called a *solution of G*. □

We now show the soundness and completeness of the resolution inference relation. In contrast to classical introductions (see e.g. [Llo87]) we do not relate the relation to a model-theoretic level, but compare it directly with the entailment relation defined in Def. 3.1.11. But before proving the soundness we show an example for a resolution refutation.

Example 5.3.1 *Resolution refutation*

In Ex. 3.1.5 we gave a proof tree showing that the sort predicate assumption $\{\mathtt{PO}(\beta)\}$ entails $\mathtt{EQ}(\beta)$ with respect to the sort axioms SA given in Ex. 3.1.3. The resolution refutation in Fig. 5.4 shows that the goal $\mathtt{EQ}(\beta)$ is also resolution refutable with respect to the assumption $\{\mathtt{PO}(\beta)\}$ and the sort axioms SA. □

$$\frac{\dfrac{(\mathtt{EQ}(\beta),\varepsilon)}{(\mathtt{PO}(\beta),[\beta/\tilde{\alpha}]\ \varepsilon)}\ (SLD)}{(\emptyset,[\beta/\tilde{\alpha}]\ \varepsilon)}\ (ASS)\begin{cases} \mathtt{C}\in\mathtt{SA},\ \ new(\mathtt{C})\ \mathtt{=}\ \mathtt{PO}(\tilde{\alpha})\Rightarrow\mathtt{EQ}(\tilde{\alpha}) \\ [\beta/\tilde{\alpha}]\ \ (\mathtt{EQ}(\beta))=[\beta/\tilde{\alpha}]\ \ (\mathtt{EQ}(\tilde{\alpha}))\end{cases}\begin{cases}\mathtt{PO}(\beta)\in\{\mathtt{PO}(\beta)\} \\ \varepsilon(\mathtt{PO}(\beta))=\mathtt{PO}(\beta)\end{cases}$$

Figure 5.4: Resolution refutation of $\mathtt{EQ}(\beta)$

Proposition 5.3.1 *Soundness and completeness of* $\vdash^{\mathrm{R}}$ *w.r.t.* $\vdash$

Let $SA \subseteq \mathrm{C}_\Omega(\Phi)$ be a set of clauses. Let $\Delta \subseteq \mathrm{AF}_\Omega(\Phi)$ and $G \subseteq \mathrm{AF}_\Omega(\Phi)$ be two sets of atomic sort formulae.

1. The resolution inference relation $\vdash^{\mathrm{R}}$ is *sound* with respect to the entailment relation $\vdash$, i.e.
$$\text{if } (G, \varepsilon) \vdash^{\mathrm{R}}_{SA,\Delta} (\emptyset, \sigma) \text{ then } \Delta \vdash_{SA} \sigma(G).$$
2. The resolution inference relation $\vdash^{\mathrm{R}}$ is *complete* with respect to the entailment relation $\vdash$, i.e.
$$\text{if } \Delta \vdash_{SA} \sigma(G) \text{ then } (G, \varepsilon) \vdash^{\mathrm{R}}_{SA,\Delta} (\emptyset, \sigma)$$

Proof: For a proof see [Pad89]. □

Note that the completeness statement is only an inversion of the soundness statement. In classical presentations using a most general unifier in the resolution rule, the statement of completeness is more involved because the answer substitution is always a most general solution.

5.3.2 Resolution Refutation Algorithm

In the last section we defined the resolution inference relation $\vdash^{\mathrm{R}}$ and showed that it is sound and complete with respect to the entailment relation $\vdash$. Now we provide an algorithm proving whether a given goal G is refutable, i.e. whether there exists some substitution σ such that $(G, \varepsilon) \vdash^{\mathrm{R}}_{SA,\Delta} (\emptyset, \sigma)$. The inference rules (ASS) and (SLD) defining the relation $\vdash^{\mathrm{R}}$ are nearly algorithmic. However, they contain three non-deterministic choices:

1. They do not fix which clause from the goal should be the selected atom.
2. They do not fix the unifier u.
3. They do not fix which clause from SA respectively which atomic sort formula from Δ should be used. It may even happen that both calculus rules can be applied in the next step.

The first non-determinism can be removed by choosing a so-called *computation rule*. This rule is used to select an atom in a resolution derivation. Now the question arises, whether the choice of a computation rule influences the completeness of the resolution algorithm. Fortunately, the following independence proposition holds (see [Llo87] for a proof):

Proposition 5.3.2 *Independence of the computation rule*

If $(G, \varepsilon) \vdash^{\mathrm{R}}_{SA,\Delta} (\emptyset, \sigma)$ then for any computation rule C $(G, \varepsilon) \vdash^{\mathrm{R,C}}_{SA,\Delta} (\emptyset, \sigma')$ for some σ' and σ is equivalent to σ' up to sort variable renaming. □

The relation $\vdash^{\mathrm{R,C}}$ is equivalent to $\vdash^{\mathrm{R}}$ except that computation rule C is used to select an atom from the goal. This is an important proposition. It enables us to fix an arbitrary computation rule to find a refutation of a given goal. If the goal is refutable we will always find a refutation using this rule. Therefore, this non-determinism is sometimes called "don't care" non-determinism. In most logic programming systems simply the first atom of the goal is selected[4].

The second non-determinism can be removed by choosing a particular unification algorithm. However, in contrast to the computation rule we cannot choose an arbitrary unification algorithm without losing completeness of $\vdash^{\mathrm{R}}$. Only for the unification algorithm *mgu*, which computes a most general unifier, we can establish the following completeness property.

Proposition 5.3.3 *Mgu preserves completeness*

If $(G, \varepsilon) \vdash^{\mathrm{R}}_{SA,\Delta} (\emptyset, \sigma)$ then $(G, \varepsilon) \vdash^{\mathrm{R,mgu}}_{SA,\Delta} (\emptyset, \sigma')$ for some σ' and there exists a substitution ρ such that $\sigma = \rho\sigma'$. □

The relation $\vdash^{\mathrm{R,mgu}}$ is equivalent to $\vdash^{\mathrm{R}}$ except that the substitution u of rule (*ASS*) and (*SLD*) is computed by *mgu*.

Unfortunately we cannot establish a similar proposition for the third non-determinism. There exists no rule choosing a clause such that a resolution derivation using this rule always leads to a refutation for a refutable goal. Thus, this non-determinism is sometimes called "don't know" non-determinism. We need a search rule which helps us to find a refutation for a given goal. The search space is a certain kind of tree, called *resolution tree*.

Definition 5.3.2 *Resolution tree*

A tree is called a resolution tree for a goal G and a substitution σ iff the tree has the following properties:

- The root node is (G, σ).
- If $G = \emptyset$ the node has no subtrees.

[4] In an implementation lists instead of sets are used for goals.

- Otherwise, let $A \in G$ be the selected atom chosen by some computation rule. Then for each clause $C \in SA$ and for each sort predicate assumption $P \in \Delta$, such that rule (*SLD*) respectively (*ASS*) can be applied resulting in $(G', u\sigma)$, where u is a most general unifier computed by *mgu*, the root has a resolution subtree for G' and $u\sigma$.

□

Each branch of a resolution tree for a goal G and a substitution σ is a resolution derivation for (G, σ). The tree can contain finite as well as infinite branches. Finite branches corresponding to refutations are called *refutation branches*. All other finite branches are called *failure branches*. The problem is now to find a refutation branch in the resolution tree. A *search rule* is a strategy for searching resolution trees to find refutation branches. From a theoretical point of view we are interested in a complete refutation algorithm, i.e. the algorithm should always find a refutation for a given goal if the goal is refutable. This can only be achieved with a so-called *fair* search rule which guarantees that each node of the search tree is visited. Because of efficiency considerations most current PROLOG systems employ a depth-first search rule. For infinite resolution trees, however, this search rule is not fair. Thus, in these systems there is a discrepancy between the declarative and the procedural semantics of the Horn-clause logic. Fair search rules must have a breadth-first component. We now define a simple resolution refutation algorithm with a breadth-first search rule.

Definition 5.3.3 *Resolution refutation algorithm resolve*

The algorithm is described in Fig. 5.5 as function *resolve* $(SA, \Delta, G) = \sigma$ where:

- $SA \in \mathrm{List}(\mathrm{C}_\Omega(\Phi))$ is a list of sort axioms from a signature
- $\Delta \in \mathrm{List}(\mathrm{AF}_\Omega(\Phi))$ is a list of atomic sort formulae
- $G \subseteq \mathrm{AF}_\Omega(\Phi)$ is the goal to be refuted
- $\sigma \in UV \to \mathrm{T}_\Omega(\Phi)$ is the computed sort variable substitution

Instead of sets we use lists for the parameters SA and Δ. In the algorithm we use the constructor [] for the empty list, $[x]$ for the list consisting of the element x and $x : xs$ for the list with the first element x and the rest xs. The infix operation ++ is used to concatenate lists. □

The function *resolve* is based on a function *solve* which puts up the resolution tree for the given goal layer by layer until a refutation is reached. The function *ass* applies the

$resolve(SA, \Delta, G) = solve([(G, \varepsilon)])$
where
$solve([]) = fail$
$solve((\emptyset, \sigma) : rest) = \sigma$
$solve((G, \sigma) : rest) = solve(rest \mathbin{++} children)$

where
$children = ass(comp\_rule(G), \Delta) \mathbin{++} sld(comp\_rule(G), new(SA))$
$ass(A, []) = []$
$ass(A, P : \Delta_{rest}) =$
case $mgu(A, P)$ **of**
$u \rightarrow (u(G \setminus \{A\}), u\sigma) : ass(A, \Delta_{rest})$
$fail \rightarrow ass(A, \Delta_{rest})$
$sld(A, []) = []$
$sld(A, (B_1, \ldots, B_n) \Rightarrow H) : SA_{rest}) =$
case $mgu(A, H)$ **of**
$u \rightarrow (u((G \setminus \{A\}) \cup \{B_1, \ldots, B_n\}), u\sigma) : sld(A, SA_{rest})$
$fail \rightarrow sld(A, SA_{rest})$

Figure 5.5: Algorithm *resolve*

inference rule (*ASS*) for each assumption that can be unified with the selected atom using *mgu*. The function *sld* applies the inference rule (*SLD*) for each sort clause, the head of which can be unified with the selected atom using *mgu*. Selection is done by the computation rule *comp_rule*. Both *ass* and *sld* yield a list of all possible resolvents of a subgoal.

Because the function *resolve* implements a breadth-first search rule the following proposition holds:

Proposition 5.3.4 *Soundness and completeness of resolve*

Let SA be a set of sort clauses. Let Δ and G be sets of atomic sort formulae.

1. The resolution refutation algorithm *resolve* is *sound* with respect to the resolution inference relation $\vdash^{\mathrm{R}}$, i.e.

$$\text{if} \quad resolve(SA, \Delta, G) = \sigma \quad \text{then} \quad (G, \varepsilon) \vdash^{\mathrm{R}}_{SA, \Delta} (\emptyset, \sigma).$$

2. The resolution refutation algorithm *resolve* is *complete* with respect to the resolution inference relation $\vdash^{\mathrm{R}}$, i.e.

$$\text{if} \quad (G, \varepsilon) \vdash^{\mathrm{R}}_{SA,\,\Delta} (\emptyset, \sigma) \quad \text{then} \quad resolve(SA, \Delta, G) = \sigma'$$

□

Because we are only interested whether there exists a refutation our algorithm *resolve* terminates with the computed substitution after having found the first refutation. Thus, we can only show a weak completeness proposition. The algorithm does not find every refutation in the tree. Algorithms used in logic programming systems allow the continuation of the search in the resolution tree for further refutations.

5.4 Well-formed Polymorphic Specifications

Based on the sort inference algorithms U and V and the resolution refutation algorithm *resolve* we now give a function testing the well-formedness of a polymorphic specification. For it, we must check whether all axioms of the specification are closed and well-formed. An axiom can be a non-qualified or a qualified formula. In both cases we must show that there exists a sort derivation under the empty variable assumption. This can be shown with the help of the following function.

Definition 5.4.1 *Analyzing algorithm analyze_ax*

Function $analyze\_ax(SA, P, F, f)$ defined in Fig. 5.6 tests the well-formedness of a non-qualified formula f or a qualified formula qf. SA, P, F have the same meaning as in Def. 5.2.3. □

In the case of a qualified formula the algorithm simply tests whether the sort inference algorithm U terminates successfully or fails. In the case of a non-qualified formula the same is done with sort inference algorithm V. However, if V terminates successfully, yielding a sort predicate assumption, we must in addition prove the closed atomic sort formulae of the computed assumption. This proof is performed by the function *resolve*. The formula to be tested is only well-formed if function *resolve* terminates successfully. For qualified formulae this test is already performed by function U. The following proposition establishes the soundness and completeness of *analyze_ax*.

Proposition 5.4.1 *Algorithm analyze_ax is sound and complete*

Let $\Sigma = ((\Omega, SA), P, F)$ be a polymorphic signature and $f \in \mathrm{PF}_\Sigma(\Phi) \cup \mathrm{QF}_\Sigma$ be an arbitrary formula.

$$
\begin{array}{l}
analyze\_ax(SA, P, F, f) = \\
\quad \textbf{case} \quad V(P, F, \emptyset, f) \quad \textbf{of} \\
\qquad\quad fail \quad\quad \rightarrow \quad non\ well\text{-}formed \\
\qquad\quad (\Delta, \sigma, D) \quad \rightarrow \quad \textbf{case} \quad resolve(SA, \emptyset, \Delta \cap \mathrm{AF}_\Omega(\emptyset)) \quad \textbf{of} \\
\qquad\qquad\qquad\quad fail \quad \rightarrow \quad non\ well\text{-}formed \\
\qquad\qquad\qquad\quad \sigma' \quad\ \rightarrow \quad well\text{-}formed \\
\\
analyze\_ax(SA, P, F, qf) = \\
\quad \textbf{case} \quad U(SA, P, F, qf) \quad \textbf{of} \\
\qquad\quad fail \quad\quad \rightarrow \quad non\ well\text{-}formed \\
\qquad\quad (\Delta, \sigma, D) \quad \rightarrow \quad well\text{-}formed
\end{array}
$$

Figure 5.6: Algorithm *analyze_ax*

1. Algorithm *analyze_ax* is sound, i.e.

$$analyze\_ax(SA, P, F, f) = well\text{-}formed \quad \Rightarrow \quad f \in \mathrm{SEN}_\Sigma$$

2. Algorithm *analyze_ax* is complete, i.e.

$$f \in \mathrm{SEN}_\Sigma \quad \Rightarrow \quad analyze\_ax(SA, P, F, f) = well\text{-}formed$$

□

We now can test the well-formedness of a polymorphic specification with the following function *analyze_spec*.

Definition 5.4.2 *Analyze algorithm analyze_spec*

Function $analyze\_spec(\Sigma, A)$ defined in Fig. 5.7 tests the well-formedness of a polymorphic specification, where Σ and A have the same meaning as in Def. 5.4.1. □

$$
\begin{array}{c}
analyze\_spec(((\Omega, SA), P, F), A) = \\
\left\{ \begin{array}{ll}
well\text{-}formed & \text{if} \quad \forall f \in A. \quad analyze\_ax(SA, P, F, f) = well\text{-}formed \\
non\ well\text{-}formed & \text{otherwise}
\end{array} \right.
\end{array}
$$

Figure 5.7: Algorithm *analyze_spec*

Proposition 5.4.2 *Algorithm analyze_spec is sound and complete*

Let (Σ, A) be a polymorphic specification. (Σ, A) is a well-formed polymorphic specification iff $analyze\_spec(\Sigma, A) = well\text{-}formed$.

Proof: Follows immediately from Prop. 5.4.1. □

5.5 Computing Principal Sort Derivations

In Section 4.3 we defined the notion of principal sort derivations for non-qualified and qualified formulae. We asserted that there exists such a sort derivation for each formula. In this section we want to prove this assertion by showing that V computes a principal sort derivation for non-qualified formulae and that U computes a principal sort derivation for qualified formulae.

Proposition 5.5.1 *V computes a principal sort derivation*

Let $f \in \mathrm{PF}_\Sigma(\Phi)$ be a closed formula. If there exists a sort derivation for f then there exists a principal sort derivation for f and $V(P, F, \emptyset, f) = (\Delta, \sigma, D)$ where D is a principal sort derivation for f. □

Proposition 5.5.2 *U computes a principal sort derivation*

Let $qf \in \mathrm{QF}_\Sigma$ be a closed qualified formula. If there exists an extended sort derivation for qf then there exists a principal extended sort derivation for qf and $U(SA, P, F, qf) = (\Delta, D)$ where D is a principal extended sort derivation for qf.

□

Now the concept of p-satisfaction defined in Def. 4.3.5 and Def. 4.3.15 is well-defined. We can use the sort derivation computed by V respectively U to define the satisfaction of formulae. Furthermore, the sort derivation can be used to reason about properties in a suitable proof system. However, because handling derivation trees in general causes some problems it is much better to work with a textual representation of the derivation tree.

The reason for defining the semantics of polymorphic specifications via sort derivations were the missing sort annotations of bound variables and the missing instantiations of polymorphic identifiers. The derivation tree contains all this information. A textual representation of a sort derivation needs not to preserve the tree structure, but all the computed sort information. Thus, we must extend our specification language with a syntactic mechanism to explicitly instantiate polymorphic identifiers. We can then translate the computed principal sort derivation of a given formula to an extended formula where all bound variables are annotated by a sort expression and where all polymorphic identifiers are instantiated explicitly.

5.6 Decidability of Well-Formedness

A problem is called *decidable* if there exists an algorithm terminating with either *true* or *false* for each instance of the problem. For practical purposes, we are interested in

the question whether the well-formedness of polymorphic specifications is decidable, i.e. whether there exists an effective algorithm terminating with *true* (meaning *well-formed*) or *false* (meaning *non well-formed*) for each polymorphic specification. In Section 5.4 we presented a function *analyze_spec* testing the well-formedness of a specification. We showed that this algorithm is sound with respect to the well-formedness defined in Section 3.2, i.e. if the algorithm terminates with *well-formed* the examined specification is well-formed. We could even show the completeness of the algorithm, i.e. if the examined specification is well-formed then the algorithm terminates with answer *well-formed*. But what happens if the specification is *not* well-formed? Unfortunately, in that case algorithm *analyze_spec* possibly may not terminate. Let us have a closer look at that problem by analyzing the termination behavior of the component algorithms.

Proposition 5.6.1 *Algorithm W and V terminate*

For each finite P, F, Γ and e algorithm W and V terminate successfully or fail.

Proof: From the form of the sort inference algorithms W and V it is clear that the process of determining whether a given term or formula is well-formed under particular assumptions P, F and Γ terminates. Both functions are recursively defined on the structure of terms respectively formulae, i.e. in each recursive call the expression is dismantled to subexpressions. Furthermore, the unification algorithm mgu used in W and V terminates on finite structures as well as the lookup functions needed for the assumptions. □

Thus, the reason for possible non-termination of *analyze_spec* can only be the function *resolve*. This function is used to search for a refutation of a goal in a resolution tree with possible infinite branches. If a resolution tree does not contain a refutation branch, but contains an infinite branch the search process does not stop. It only stops if, as in algorithm *resolve*, a fair search rule is used and if the resolution tree for the examined goal contains at least one refutation branch. The non-termination of function *resolve* is not because we used a "bad" algorithm to show the refutation of a goal. In fact, the problem to show whether there exists a substitution σ such that $(G, \varepsilon) \vdash^{\mathrm{R}}_{SA,\Delta} (\emptyset, \sigma)$ or whether no such substitution exists is undecidable. The problem is, however, *semi-decidable*.

Proposition 5.6.2 $\vdash^{\mathrm{R}}$ *is semi-decidable*

The existence of a substitution σ such that $(G, \varepsilon) \vdash^{\mathrm{R}}_{SA,\Delta} (\emptyset, \sigma)$ is semi-decidable, i.e. there exists an algorithm that correctly outputs *true* if there exists such a substitution but may fail to terminate in cases where *false* should be output.

Proof: It is easy to show that the resolution refutation algorithm *resolve* presented in 5.5 is such an algorithm if a substitution is interpreted as *true* and *fail* is interpreted as *false*. By Prop. 5.3.4 we know that *resolve* is complete, i.e. it alway finds a refutation

tree. The algorithm, however, does not terminate if the tree contains an infinite branch but no refutation branch. □

The semi-decidability of $\vdash^R$ propagates to the problem of deciding whether a given polymorphic specification is well-formed because this problem is based on the former one.

Proposition 5.6.3 *Well-sortedness is semi-decidable*

The question whether a given polymorphic specification is well-formed is semi-decidable. That is, there exists an algorithm answering *well-formed* if the specification is well-formed but may fail to terminate if the specification is non well-formed. □

The algorithm *analyze_spec* presented in Fig. 5.7 is such an algorithm. In Section 5.7 we will deal with the practical consequences of the semi-decidability.

As already mentioned, the non-termination of algorithm *resolve* results from infinite branches in the resolution tree. The problem of finding a refutation becomes decidable if the resolution tree is finite. A sort specification that does not lead to infinite resolution trees is called a *decidable sort specification*.

Thus, if we restrict the sort specification of polymorphic specifications to decidable ones the problem of whether a given specification is well-formed becomes decidable. Moreover, *analyze_spec* solves this problem. That means, if we apply *analyze_spec* to a specification with a decidable sort specification the algorithm terminates with either *well-formed* or *non well-formed*.

Infinite resolution trees arise from recursively defined sort predicates that can be expanded infinitely many times. If all sort predicates are defined by recursion over the structure of sort terms, we get a decidable sort specification.

In [Kae92] the decidability of a very similar entailment relation is investigated. Instead of Horn clauses, rewrite rules are used to describe the properties of the sort predicates. He establishes sufficient conditions for rewrite rules to guarantee the decidability of the entailment relation. His results can be transferred to our approach.

The sort class system of Haskell or SPECTRUM enables only a very restricted specification of class properties. Classes can be ordered by a (non-cyclic) subclass relation and sort constructors can be specified to belong to a particular class if the parameters belong to some class. If we translate such a sort class specification to an equivalent set of sort clauses, as shown in Section 2.1, we get a decidable sort specification.

5.7 Concrete Implementation

If a problem is semi-decidable the corresponding decision function will terminate if the answer is *true*, but may fail to terminate if the answer should be *false*. As long as the

algorithm is running we can never say whether it will perhaps terminate successfully some time or whether it will run forever. For practical reasons, however, we are interested in an algorithm that always terminates. This can only be achieved if we force the algorithm to terminate because of some additional restrictions. Most decision algorithms will terminate for reasons of natural space restrictions, e.g. stack, heap or memory. However, we prefer a controlled termination due to some artificial restrictions.

An obvious restriction is to limit the search depth in the resolution tree. If the maximal search depth is reached the algorithm should terminate. Such an artificial termination influences either the soundness or the completeness of the algorithm. Because we are much more interested in the soundness of our algorithm than in completeness, the algorithm must assume that there exists no refutation below the maximal search depth. Thus, *resolve* must answer *fail* if it terminates due to the search depth. This behavior preserves soundness but leads to incompleteness because there may exist a refutation below the maximal search depth. The incompleteness of *resolve* propagates to *analyze_spec*. Thus, *analyze_spec* will sometimes output *non well-formed* for a well-formed polymorphic specification. However, and this is more important, it will never output *well-formed* if the specification is non well-formed.

Losing the completeness of an algorithm is not really critical. In fact, many completeness results for decision algorithms are only of theoretical interest because of the high complexity of the algorithm. In practice, for big arguments they terminate irregularly with messages like *stack overflow* or *heap overflow* though theoretically the algorithm should terminate with *true*. This is a form of *practical incompleteness*. Experiences with an implementation of the language PolySpec [Gam94] (described in Appendix B) showed that the semi-decidable sort system of the language causes no problems in practice. The implemented specification analyzer is parameterized with respect to the maximal search depth. Furthermore, the implementation of *resolve* distinguishes between a proper failure and a failure because of the maximal search depth was reached. In addition, besides much other information, the goal that failed to be refuted is displayed. If the refutation failed because the maximal search depth was reached there are two possibilities. Either the user recognizes that there indeed exists no refutation for the goal. Then he must search for the error that led to the goal, correct it and start the analyzer again. Or the user assumes that there must exist a refutation for this goal. Then he must start the analyzer again using a bigger maximal search depth.

Note, however, that the semantics is only defined for well-formed polymorphic specifications. As long as the analyzer does not terminate with *well-formed* the specification must be treated as not well-formed and thus has no defined semantics.

Chapter 6

Polymorphism Revisited

In Chapter 1.2 we gave an informal introduction to the most common kinds of polymorphism. In this chapter we want to take a closer look at the semantical differences between these concepts.

In general, polymorphic languages allow, in contrast to monomorphic languages, an object not only to be of one sort but of many different ones. Strachey [Str67] distinguished, informally, between *ad-hoc polymorphism* and *parametric polymorphism*. Cardelli and Wegner [CW85] refined this classification into the structure displayed in Fig. 6.1.

The refined classification arises from the following considerations:

Parametric polymorphism is obtained when a function works uniformly on a possible infinite range of sorts with a common sort structure. The common structure is achieved by abstracting from sorts with the help of sort variables. Abstraction and instantiation may be explicit or implicit.

Inclusion polymorphism provides a framework to capture the concept of implicit subsorting. Sorts may be ordered by an inclusion relation. A supersort contains all objects of its subsort. As in parametric polymorphism each function works uniformly on all subsorts.

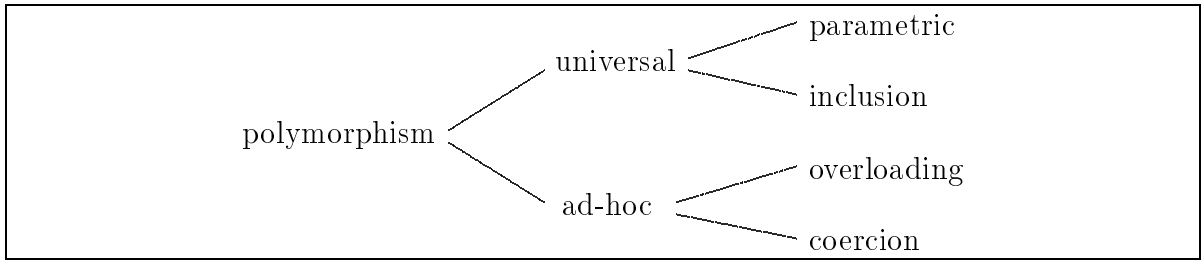

Figure 6.1: Classification of polymorphism due to Cardelli and Wegner

Overloading polymorphism allows to use one identifier to denote different semantic objects. These objects need not be related in some way and thus may behave differently.

Coercion polymorphism is based on an operation converting an argument to the sort expected by a function.

The main distinction between universal and ad-hoc polymorphism is that in universal polymorphism a polymorphic function works uniformly on all admissible sorts. These sorts, sometimes infinite, are related in some way. An ad-hoc polymorphic function only works on a finite set of unrelated sorts. Ad-hoc polymorphism is usually eliminated by preprocessing the program and is not dealt with in the semantics. Thus in [CW85] universal polymorphism is called *true polymorphism* whereas ad-hoc polymorphism is called *apparent polymorphism*.

In the following sections we will point out differences between polymorphism as used in programming languages and as used in axiomatic specification languages.

6.1 Hindley/Milner Polymorphism

Hindley/Milner polymorphism, as used in functional languages, is a kind of parametric polymorphism. Polymorphic functions can only be defined by let-abstraction. Sort abstraction as well as instantiation is done implicitly. Polymorphic functions obtained by let-abstraction work uniformly on all instances. This is guaranteed by the purely syntactic abstraction mechanism. Let us have a closer look at the let-mechanism. The usual let-construct looks as follows[1]:

$$\texttt{let}\ \ f = e_1[f]\ \ \texttt{in}\ \ e_2[f]$$

The identifier f may occur in e_1 as well as in e_2. This is denoted by the brackets. The identifier f is uniquely defined by expression e_1, called the *body* of f. Thus the occurrences of f in e_1 are *defining occurrences*. The occurrences of f in e_2 do not define the function. We call these occurrences *applied occurrences*. By using a sort inference algorithm that computes most general sorts for expressions it is guaranteed that the sort of e_1 is as abstract or polymorphic as possible.

Our axiomatic specification language does not provide a let-construct. There is no difficulty to integrate such a mechanism into our framework. The let-construct is, however, rarely used in axiomatic specifications. Thus it is more convenient not to integrate it

[1] Sometimes the keyword `letrec` is used to indicate a recursive definition.

into the core language but to expand let-defined identifiers by copying their bodies to the application occurrences (see, e.g., Chapter 6.5 of [BFG+93a]).

In our specification language polymorphic identifiers can only be introduced in the signature of a specification. The signature forms the counterpart of the let-construct. In contrast to the let-construct, however, in the signature identifiers are only *declared* but not *defined*. The definition takes place in the axioms by using an identifier. As a consequence, in specifications we cannot distinguish between defining and applied occurrences of an identifier. Each use of an identifier also defines the properties of the function denoted by this identifier.

Therefore, in polymorphic specifications the uniform behaviour of polymorphic functions cannot be guaranteed. A polymorphic function may work completely differently on different sorts. Specification 6.2 shows a non-uniform usage of our Hindley/Milner polymorphism.

We declare a polymorphic infix[2] identifier + and define it do behave as addition on natural numbers and as concatenation on arbitrary lists. The function clearly does not behave uniformly. Of course we have misused our polymorphism to model some kind of ad-hoc polymorphism. However, syntactically we cannot prevent the user from writing such specifications.

Uniform behaviour can only be achieved by a uniform axiomatization. For example, the last axioms block of Specification 6.2 uniformly describes the polymorphic identifier $+$: $\Pi\alpha.\ \texttt{List}(\alpha) \times \texttt{List}(\alpha) \to \texttt{List}(\alpha)$. The axioms must hold for all possible instances of this identifier.

Based on the classification given in the introductory part of this chapter our polymorphism is therefore not a kind of parametric polymorphism. Nevertheless it is not a kind of overloading. Overloading is not dealt with in the semantics. It is eliminated by renaming the different occurrences of the function. In our approach, in contrast, polymorphic identifiers are handled in the semantics. In a polymorphic Σ-algebra a polymorphic value is assigned to each polymorphic identifier of the signature Σ. If applied to a tuple of sorts this value yields a monomorphic value. Therefore in the framework of axiomatic specification we prefer the notion "parametric polymorphism" though it does not coincide with the classification given in [CW85].

In axiomatic specification languages we can then distinguish between two kinds of parametric polymorphism: *ad-hoc polymorphism* and *uniform polymorphism*. Function + of Specification 6.2 is an example of ad-hoc polymorphism. If we restrict the function as described above we get uniform polymorphism. More formally, a parametric polymorphic function is called *uniformly polymorphic* if the same properties hold for all instances

[2]Note that PolySpec does not really provide infix notation.

```
sortpred  CONTAINER:2;

fun  +: Πα. α × α → α;

cons Nat;

fun  0: Nat;
     succ: Nat → Nat;

axioms x,y in
     0 + x = x;
     (succ x) + y = succ(x + y);
endaxioms

cons List:1;

fun  []: Πα. List(α);
     append: Πα. α × List(α) → List(α);

axioms x,l,m in
     [] + l = l;
     append(x,l) + m = append(x,l + m);
endaxioms
```

Specification 6.2: Non-uniform usage of Hindley/Milner polymorphism

of the function. Otherwise it is called *ad-hoc polymorphic*. This is a purely semantic classification.

When developing functional programs all pure polymorphic functions, i.e. polymorphic functions without qualifying sort predicates, should be specified to be uniformly polymorphic. Otherwise, the transition from the specification to the functional program raises problems, since only uniformly defined functions can be implemented by a generic code.

6.2 Sort Classes

The original motivation for the introduction of sort classes in Haskell was "to make ad-hoc polymorphism less ad-hoc"[WB89]. And indeed, due to the classification given in the

introductory part of this chapter, sort classes are not a kind of parametric polymorphism but a kind of ad-hoc polymorphism. It is completely eliminated before run-time by using so-called *dictionaries* (see, e.g., [HB89]).

In contrast to our approach, a Haskell sort class is not just a unary sort predicate. Each sort class has in addition a set of so-called *member functions*. A member function is a purely syntactic notion consisting of an identifier and its functionality. The sort variables in the functionality are restricted by the sort class. For example the class `EQ` defined in Specification 2.1 would be defined as follows in Haskell:

$$\texttt{class EQ } \alpha \texttt{ where}$$
$$\texttt{eq: } \alpha \times \alpha \rightarrow \texttt{Bool}$$

Function `eq` is the only member function of class `EQ`. In contrast to Specification 2.1, Haskell does not allow one to specify the laws of the member function. Other functions that are restricted by the class `EQ` are not member functions of `EQ`. Their definition is, either directly or indirectly, based on the member function `eq`.

When instantiating a class with a particular sort in Haskell, the member functions must be instantiated. That means, a concrete function must be provided for this sort. As a consequence, the member functions are uniquely defined for this particular instance. All other functions restricted by this class need not be instantiated. They are dynamically instantiated by passing a dictionary containing the definitions of the particular member functions.

In our approach there is no notion of a member function and thus no explicit instantiation mechanism for these functions. Of course, e.g., function `eq` of Specification 2.1 can be seen as a member function and by the axioms given in Specification 2.2 `eq` is implicitly instantiated for sort `Nat` and `List(`α`)`. But we do not provide an explicit mechanism. In particular, a class can be instantiated without providing a specific definition for any function. By specifying, e.g., `EQ(Nat)` we only demand that the function `eq` be applicable to elements of sort `Nat`. Furthermore, the `Nat`-instance of `eq` must fulfill the general laws of `eq`.

This very liberal view of sort classes allows one to write abstract requirement specifications. Member functions and concrete instances of member functions must, however, be defined if some kind of executable design specification should be transferred to a Haskell program. We argue that the treatment of member functions should be part of a development method. Such a method step should also include a proof showing that each instance of a member function fulfils the general laws of the function. A formal treatment of sort classes with distinguished member functions and consistency preserving development of theories is investigated in [Reg94].

As mentioned in Section 6.1 the Hindley/Milner polymorphism used in functional languages is a kind of parametric polymorphism: the behaviour of a polymorphic function

does not depend on a specific instantiation. As a consequence, the operational semantics of a functional program using Hindley/Milner polymorphism does not depend on the way it is type checked. It can therefore be defined without taking the type information inferred during type check into consideration. This property is known as *coherence* [BTCGS89].

The sort concept of Haskell is a kind of ad-hoc polymorphism: the member functions of a class are instantiated with different functions for each sort. In classical overloading approaches an overloaded identifier must be uniquely determined by its application context. Ambiguously overloaded terms are rejected as not well-formed. For such terms clearly the coherence property does not hold.

In polymorphic languages such as Haskell overloading resolution is complicated by the presence of sort variables. A dictionary passing mechanism is used to resolve overloaded identifiers at runtime. The operational semantics of a Haskell program is defined by translating it, based on a principal sort derivation, to a functional program without ad-hoc polymorphism. This translation should be well-defined, i.e. for a given principal sort derivation the result of the translation should be uniquely determined. Unfortunately, this does not hold in general. This problem is known as the *coherence problem*. As an example consider the following class definition[3]:

```
class Parsable α where
      parse: String → α
      unparse: α → String
```

Then the expressions

```
unparse (parse "123")
```

is ambiguous. The sort of the intermediate value, i.e. the result of `parse "123"`, is not uniquely determined though the whole expression is uniquely of sort `String`. Because `parse` and `unparse` may behave differently for different instantiations, the semantics of this expression is not uniquely determined.

In [Blo92], [Jon92] and [Che95] sufficient conditions for expressions are defined that guarantee coherence. In our approach, however, the coherence problem does not even arise. We do not need any further conditions on expressions to ensure a well-defined semantics. The reason is that we do not define an operational semantics for specifications but only a declarative one. We will explain the difference in more detail. Fig. 6.3 shows a principal sort derivation[4] for the expression given above where

[3] The example is taken from [Che95].

[4] For reasons of space the derivation tree is split into two parts.

$$\texttt{parse: } \Pi\alpha.\ \texttt{Parsable}(\alpha) \Rightarrow \texttt{String} \rightarrow \alpha;$$
$$\texttt{unparse: } \Pi\alpha.\ \texttt{Parsable}(\alpha) \Rightarrow \alpha \rightarrow \texttt{String};$$
$$\text{“}\texttt{123}\text{”}\texttt{: String};$$

are part of the signature Σ.

$$\dfrac{\dfrac{}{\texttt{Parsable}(\alpha), \emptyset \rhd_\Sigma \texttt{parse} :: \texttt{String} \rightarrow \alpha}\,(polyid) \quad \dfrac{}{\texttt{Parsable}(\alpha), \emptyset \rhd_\Sigma \text{“}\texttt{123}\text{”} :: \texttt{String}}\,(id)}{\texttt{Parsable}(\alpha), \emptyset \rhd_\Sigma \texttt{parse } \text{“}\texttt{123}\text{”} :: \alpha}\,(\rightarrow E)$$

$$\dfrac{\dfrac{}{\texttt{Parsable}(\alpha), \emptyset \rhd_\Sigma \texttt{unparse} :: \alpha \rightarrow \texttt{String}}\,(polyid) \quad \texttt{Parsable}(\alpha), \emptyset \rhd_\Sigma \texttt{parse } \text{“}\texttt{123}\text{”} :: \alpha}{\texttt{Parsable}(\alpha), \emptyset \rhd_\Sigma \texttt{unparse (parse } \text{“}\texttt{123}\text{”}\texttt{)} :: \texttt{String}}\,(\rightarrow E)$$

Figure 6.3: Principal sort derivation for `unparse (parse “123”)`

In Def. 4.2.3 we gave an interpretation function for sort derivations yielding a semantic value. If we apply this interpretation function to the derivation tree given in Fig. 6.3 we get a value in the interpretation of sort `String`. The interpretation, however, depends on a variable assignment as well as on a sort variable assignment. In our example we can ignore the variable assignment because the expression does not contain a variable. Since the sort derivation contains the sort variable α the interpretation depends on the value assigned to α. Different assignments may lead to different semantic values.

From this point of view the coherence property also does not hold in our framework. But this does not result from constraining sort variables by sort classes. The interpretation may even yield different values if we omit the restricting sort predicate `Parsable(`α`)`. As well `parse` as `unparse` need not be defined uniformly (see Section 6.1).

Nevertheless, the notion of model of polymorphic specifications is well-defined. A specification consists of a set of closed formulae. We must therefore apply at least a sort predicate `p` to our example expression. Suppose that `p: String` is a predicate in the signature Σ. Then

$$\dfrac{D}{\texttt{Parsable}(\alpha), \emptyset \rhd_\Sigma \texttt{p(unparse (parse } \text{“}\texttt{123}\text{”}\texttt{))}}\,(appl)$$

is a principal sort derivation for `p(unparse (parse “123”))` where D is the derivation tree displayed in Fig. 6.3. By Def. 4.2.6 the sort derivation is satisfied in an algebra if the interpretation of the sort derivation yields *true* for all sort variable assignments satisfying the sort predicate context `Parsable(`α`)`. As a consequence, the satisfaction of a sort derivation does not depend on a particular sort variable assignment. All sort variables occurring in a sort derivation are treated as universally quantified at the outermost level of a formula. Thus, to be valid in some algebra, predicate `p` must hold for all different values occurring from different assignments to the sort variable α.

Because of this property, our specification language allows even to specify the coherence of parsing and unparsing a string. In Specification 6.4 the axiom states that parsing and unparsing a string yields the original string again.

```
cons      String;

sortpred  Parsable:1;

fun       parse: Πα. Parsable(α) ⇒ String → α;
          unparse: Πα. Parsable(α) ⇒ α → String;

axioms x in
     unparse(parse x) = x;
endaxioms
```

Specification 6.4: Specification of sort class `Parsable`

Note, that this must hold for all possible instances of function `parse` and `unparse`. Regardless of which sort is chosen for the intermediate value, if we unparse the intermediate value we get the original string back.

Although not relevant in the framework of axiomatic specification the coherence problem must be paid attention to if translating an executable specification to a Haskell program.

6.3 Subsorting

Our notion of polymorphism allows only the modelling of an explicit subsorting concept. Sorts cannot be specified to semantically overlap. A subsort mechanism can only be achieved by explicitly constraining a sort variable in the sort term of a function. In Section 2.2 a suitably specified binary sort predicate `IS_A` was used to model subsorting. In addition we used a function `coerce` to specify the coincidence of `IS_A`-restricted functions on common subsorts. Based on the classification given in the introductory part of this chapter, we therefore cannot model inclusion polymorphism but some kind of coercion polymorphism.

In the logical programming language OBJ [GW88] subsorting is used to avoid partial functions. Functions can be made total by restricting the parameter sort to an appropriate subsort. The predecessor function on natural numbers is an example of a partial function because it is undefined on zero.

The following signature shows how the predecessor function `pred` can be made total by using an implicit subsorting concept:

```
cons PosNat; Nat;
     PosNat ⊆ Nat;

fun  succ: Nat → PosNat;
     pred: PosNat → Nat;
```

In the signature above `PosNat` is the subsort of the positive natural numbers. The signature allows us to apply the function `succ` repeatedly, e.g. `succ(succ(0))`. This expression is well-sorted, because the subsort relation `PosNat` $\subseteq$ `Nat` enables us to coerce term `succ(0)` to sort `Nat`. If we look at the expression `pred(pred(succ(succ(0))))` we would expect that the evaluation yields the proper value `0`. This term, however, is not well-sorted, because the sort `Nat` of `pred(succ(succ(0)))` must be coerced to `PosNat`, which is not possible in general.

In a programming language there is a simple solution to this problem. If a value of sort α must be coerced to a subsort β the compiler inserts a so-called *retract function* `r` with functionality $\alpha \to \beta$. This retract function yields a sort error at runtime, if the coercion is not possible. The above example would be changed by the compiler to `pred(r(pred(succ(succ(0)))))`. So parts of the sort check are deferred to the run-time of the program. The price one has to pay is the loss of the static sort system. But retract functions keep the language strongly sorted, because at run-time all sort errors are detected.

In non-executable specification languages, in contrast, there is no such simple solution to this problem: if the specification cannot be executed, parts of the sort check cannot be deferred to run-time. Thus, in non-executable languages a strong sort system must be a static sort system. Furthermore, the problem was solved by using a partial function `r`. But in the above example subsorting was introduced to avoid partial functions. As a consequence, in a specification language the subsorting concept is not suitable to avoid partial functions, and the specification language must itself provide partial functions.

Chapter 7

Conclusion

In this chapter we finally review the results gained in this thesis. Furthermore, we give an outline of future work in the area of polymorphic axiomatic specifications.

7.1 Summary of Contributions

In this thesis we have presented a novel polymorphic sort system for axiomatic specification languages. The sort system itself can be dynamically specified by a separate specification language. This approach enables a flexible sort system that can be dynamically adapted to different application areas. In particular, this sort system allows one to model different notions of polymorphism. We have investigated the syntactic as well as the semantic treatment of axiomatic specification languages based on such a sort system.

From the practical perspective, the main contribution of this thesis is the development of a polymorphic axiomatic specification language the sort system of which

- can be dynamically adopted to specific application areas,
- restricts the user as little as possible in expressibility,
- allows problem-oriented specifications,
- allows one to write sort-free axioms,
- enables the detection of all errors with respect to the sort system by static specification analysis,
- although only semi-decidable causes no problems in practice,

- is intuitive and easy to understand because well-known specification concepts like constants, predicates, and axioms are transferred to the sort level, and
- is well-suited for developing functional programs because it is oriented towards the sort systems of functional programming languages.

From the theoretical perspective, the main contributions of this thesis are:

- The syntactic and semantic foundation of polymorphic specification languages.
- The formulation of a syntactic framework for two-level polymorphic specifications.
- A sort system based on qualified sorts that in addition to [Jon92]
  - allows explicit sort annotations,
  - allows explicit qualifying sort predicates and
  - provides a general approach to specifying the properties of sort predicates.
- A formal interpretation for qualified sorts.
- The definition of two-level polymorphic algebras.
- A definition of model for polymorphic specifications that is neither based on an algorithm nor on any notion of "principal sort".
- The study of semantic relations between different sort derivations of a formula.
- A definition of model for polymorphic specifications based on a principal sort derivation which is proven to be equivalent to the former one.
- An implementation of the sort system, based on a resolution prover. In particular we provide an algorithm proving the well-formedness of polymorphic specifications and computing principal sort derivations.

We have used a general first order logic to study our notion of polymorphism. The results we gained in this thesis can be immediately transfered to syntactically richer and logically more restricted languages. The results should, in principle, also be applicable to higher order logic although we have not investigated this here.

7.2 Future Work

7.2.1 Higher-Order Sort Variables

In [GN94] we have shown that Hindley/Milner polymorphism combined with sort classes is powerful enough to model most of the classical examples for parameterized specifications. However, there are cases in which we want to abstract not only from one sort but from a tuple of sorts. This is not possible in the language SPECTRUM of [GN94] but in our approach by using n-ary sort predicates. There are also cases in which we want to abstract not only from basic sorts but from n-ary sort constructors. A typical example is a container structure with lists or trees as instances. In [GN94] we used a combination of polymorphic and parameterized specification to abstract from a concrete container. In our framework we can model the abstraction by using n-ary sort predicates. As the following example demonstrates, the result is, however, neither easy to understand nor problem oriented.

Let us assume we want to specify a function `member` testing the membership in an arbitrary container structure. Specification 7.1[1] shows a solution based on a binary sort predicate `CONTAINER`. The intended semantics behind `CONTAINER(`α`,`β`)` is that β is an α-container.

```
sortpred  CONTAINER:2;

fun  member: Πα,β.CONTAINER(α,β) ⇒ α × β → Bool;

cons List:1;
     Tree:1;

sortaxioms α in
     CONTAINER(α,List(α));
     CONTAINER(α,Tree(α));
endaxioms
```

Specification 7.1: Abstract `member` function

In the signature of function `member` we abstract from the elements as well as from the container by using sort variables. By the qualifying sort predicate we try to ensure that β is an α-container. In the sort axioms we then specify `List(`α`)` and `Tree(`α`)` to be

[1] Note that the sort variable α in the functionality of the member function should further be restricted by a predicate ensuring an equality function on α.

α-containers. Thus, `member` can be used to test the membership in sets and trees. For this rather simple example the abstraction mechanism seems to work quite well.

Let us look at a more complicated example. We want to define a more general version of the well-known map function. This more general version applies an argument function to all elements of a container and yields the corresponding container as result. Specification 7.2 shows the proceeding.

```
fun  map: Πα,β,γ,δ.CONTAINER(α,γ),CONTAINER(β,δ)
                  ⇒ (α → β) → γ → δ;

sortpred  EQUAL_STRUCTURE:2;

fun  map': Πα,β,γ,δ.CONTAINER(α,γ), CONTAINER(β,δ),
                    EQUAL_STRUCTURE(γ,δ) ⇒ (α → β) → γ → δ;

sortaxioms α,β in
     EQUAL_STRUCTURE(List(α),List(β));
     EQUAL_STRUCTURE(Tree(α),Tree(β));
endaxioms

sortpred  MAP:4;

fun  map'': Πα,β,γ,δ. MAP(α,β,γ,δ) ⇒ (α → β) → γ → δ;

sortaxioms α,β in
     MAP(α,β,List(α),List(β));
     MAP(α,β,Tree(α),Tree(β));
endaxioms
```

Specification 7.2: Abstract `map` function

The functionality of `map` seems to be a natural approach to model the situation. Function `map` takes a function from α to β as an argument and yields a function from γ to δ as a result. By the qualifying sort predicates we achieve that γ is an α-container and that δ is a β-container. But this functionality does not fully describe the desired behaviour. The sort variables γ and δ are not related. Thus, γ and δ can be instantiated by different container structures. But `map` should only apply the argument function without changing the container structure. In the case of the functionality of `map'` we achieve this by using an additional sort predicate `EQUAL_STRUCTURE`. In `map''` we instead use a four place sort

predicate `MAP` to restrict the sort variables in a suitable way. But neither the functionality of `map'` nor the functionality of `map''` adequately describes the intended behaviour.

Abstraction from container structures is not possible in Haskell. Therefore, two approaches have been developed to overcome this deficiency. Chen [Che95] proposes an extension of the Haskell type class concept called *parametric type classes*. Such classes can have type parameters in addition to the placeholder variable which is always present in a class declaration. The parametric type class relates the placeholder variable with the type parameters. Although parametric type classes are syntactically different to our approach, from a theoretical viewpoint they can be seen as restricted n-ary sort predicates. As a consequence, they are able to express function `member` of the former example, but, for the same reason as in our approach, fail to adequately model function `map`.

Another approach to model non-basic sort constructor abstraction are *constructor classes* as proposed by Jones [Jon93b] and implemented in the functional language Gofer [Jon93a]. From a theoretical point of view a constructor class is a sort class over a higher order sort variable. Thus, constructor classes allow one to directly abstract from a non-basic sort constructor. To achieve a similar mechanism we must drop the restriction to first order sort variables and allow higher order sort variables in sort terms.

The sort predicate `CONTAINER` is defined to be a predicate over a sort constructor by the *kind term* $*\rightarrow*$. The asterisks denote the kind of all sorts. The arrow $\rightarrow$ denotes the *kind constructor* constructing the set of all unary sort constructors[2]. Thus, we need a third level to achieve abstraction from non-basic sort constructors.

In the functionality of `map` we now use a second order sort variable φ to abstract from a concrete container. The sort variable is restricted by the sort predicate `CONTAINER`. The relation between the sort of the argument function and the sort of the container elements is now simply achieved by the sort variables α, and β, respectively. The same container structure is guaranteed by the sort variable φ.

In the axioms part we specify the general laws of the map function. By sort axiom `CONTAINER(List)` we fix the sort constructor `List` to construct a container structure. Therefore, `map` can be applied to list structures. In the last axioms block we define the specific behaviour of the map function on lists. Of course, it has to be proven that the `List`-instance of `map` satisfies the general laws specified in the first axioms block.

The results on the syntactic treatment of constructor classes gained in [Jon93b] can be applied to our framework. However, work has to be done to integrate higher order sort variables into our general entailment approach. In particular, we must use a resolution prover handling higher order variables. Furthermore, the semantics gets more involved and thus has to be revised carefully.

[2]Do not confuse the two different ways of using arrow $\rightarrow$.

```
sortpred  CONTAINER: ∗→∗;

fun  map: Πα,β,φ.CONTAINER(φ) ⇒ (α → β) → φ(α) → φ(β);

axioms f, g in
     map id = id;
     comp(map f,map g) = map(comp(f,g));
endaxioms

cons List:1;
fun  []:Πα. List(α);
     append:Πα. α × List(α) → List(α);

sortaxioms in
     CONTAINER(List);
endaxioms

axioms x, l, f in
     map f [] = [];
     map f (append(x,l)) = append(f x,map f l);
endaxioms
```

Specification 7.3: Abstract `map` function by higher order sort variable

7.2.2 Further Areas

Our approach only allows explicit subsorting. An interesting investigation would be to integrate an implicit subsorting concept into our approach. The syntactic handling of implicit subsorting raises no problems. However, the semantics has to be studied carefully. A promising approach seems to be the integration of higher-order order-sorted algebras, as proposed by Qian [Qia91], into our polymorphic algebras.

The syntactic framework used in this thesis only allows specifications "in the small". They are suited to describe small abstract data types. It is, however, more convenient to build large and complex specifications in a structured way by combining and modifying smaller specifications. Structuring can be achieved by using so-called *specification building operators* which map a list of specifications into a result specification. Typical specification building operators are + and `rename` combining two specifications and renaming identifiers of a specification, respectively. An interesting direction for future work is to extend the presented two-level framework by specification building operators

and to study their semantics.

As already mentioned, polymorphic specifications are closely related with parameterized specifications. In [GN94] it was shown that in many cases parameterized specifications can be replaced by more elegant polymorphic ones. The sort class concept used in [GN94], however, is not powerful enough to completely replace parameterized specifications. Furthermore, the comparison is rather informal, being based on examples. Therefore, it would be interesting to compare our more expressive polymorphism approach formally with classical parameterization.

A formal specification language only forms the basis for correctness-oriented program development. In addition, a deduction calculus and a development methodology are required. Both issues are not addressed in this thesis. However, our notion of polymorphism requires methodological support. As already mentioned in some examples, the general laws of a polymorphic function have to be fulfilled by the specific instances. Thus, each instance declaration should be accompanied by a proof.

Though the sort system of our specification language is oriented towards the sort systems used in functional languages, the final transition from a specification to a functional program requires further research. In particular, our approach does not distinguish the member functions of a sort class. In Haskell or Gofer, however, these functions are essentially associated with the class concept. In general, all aspects discussed in Chapter 6 must be paid attention to when leaving the formal framework developed in this thesis. Therefore, deduction as well as methodology are important areas that have to be investigated in the framework of polymorphic specification.

Appendix A

Proofs

Proposition 3.1.1 *Monotonicity of* $\vdash$

Let Δ_1 and Δ_2 be two sets of atomic sort formulae.

$$\text{if} \quad \Delta_2 \subseteq \Delta_1 \quad \text{then} \quad \Delta_1 \vdash_{SA} \Delta_2$$

Proof: Under assumption $\Delta_2 \subseteq \Delta_1$, we have to prove $\Delta_1 \vdash_{SA} \Delta_2 \overset{\text{Def. 3.1.11}}{\Leftrightarrow} \Delta_1 \vdash_{SA} C$ for each $C \in \Delta_2$. Because of $C \in \Delta_1$, this follows immediately by rule $(assumption)$ of Def. 3.1.11. □

Proposition 3.1.2 *Substitution Closure Property of* $\vdash$

Let Δ be a set of atomic sort formulae, and let A be an atomic sort formula. Let $\sigma : \Phi \to \mathrm{T}_\Omega(\Phi)$ be a sort variable substitution.

$$\text{if} \quad \Delta \vdash_{SA} A \quad \text{then} \quad \sigma(\Delta) \vdash_{SA} \sigma(A)$$

The proposition can be extended to sets of atomic sort formulae Δ_1 and Δ_2:

$$\text{if} \quad \Delta_1 \vdash_{SA} \Delta_2 \quad \text{then} \quad \sigma(\Delta_1) \vdash_{SA} \sigma(\Delta_2)$$

Proof: Let $P = \Delta \bigtriangledown_{SA} A$ be an arbitrary proof tree. We show that if we apply a substitution σ to both the assumptions and the atomic sort formula of each proof tree node, we get a valid proof tree, i.e. $\sigma(P) := \sigma(\Delta) \bigtriangledown_{SA} \sigma(A)$ is a valid proof tree. We prove this by induction on the structure of the proof tree:

- $P = \dfrac{}{\Delta, A \vdash_{SA} A}(assumption) \;\Rightarrow\; \sigma(P) := \dfrac{}{\sigma(\Delta), A) \vdash_{SA} \sigma(A)}(assumption) = \dfrac{}{\sigma(\Delta), \sigma(A) \vdash_{SA} \sigma(A)}(assumption)$

  This is a valid application of rule $(assumption)$ and thus yields a valid proof tree.

- $P = \dfrac{\Delta \bigtriangledown_{SA} \sigma'(B_i)\ \ 1 \leq i \leq n}{\Delta \vdash_{SA} \sigma'(H)}(mp) \ \Rightarrow\ \sigma(P) := \dfrac{\sigma(\Delta) \bigtriangledown_{SA} \sigma\sigma'(B_i)\ \ 1 \leq i \leq n}{\sigma(\Delta) \vdash_{SA} \sigma\sigma'(H)}(mp)$

  Because P is a valid proof tree there exists a sort clause $B_1, \ldots, B_n \Rightarrow H$ in SA. Thus, this is a valid application of rule (mp) with substitution $\sigma\sigma'$ and yields a valid proof tree because by induction $\sigma(\Delta) \bigtriangledown_{SA} \sigma\sigma'(B_i)$ are all valid proof trees.

The proof can be trivially extended to a set of atomic sort formulae. □

Proposition 3.1.3 *Transitivity closure property of* $\vdash$

Let A_1 and A_2 be two atomic sort formulae, and let Δ be a set of atomic sort formulae.

$$\text{i)} \quad \text{if} \quad \Delta \vdash_{SA} A_1 \quad \text{and} \quad A_1 \vdash_{SA} A_2 \quad \text{then} \quad \Delta \vdash_{SA} A_2$$

The proposition can be extended to sets atomic sort formulae Δ_1, Δ_2, and Δ_3.

$$\text{ii)} \quad \text{if} \quad \Delta_1 \vdash_{SA} \Delta_2 \quad \text{and} \quad \Delta_2 \vdash_{SA} \Delta_3 \quad \text{then} \quad \Delta_1 \vdash_{SA} \Delta_3$$

Proof:

i) We know that there exists a proof of $A_1 \vdash_{SA} A_2$. If A_2 can be proved without assumption A_1, i.e. there exists no application of rule $(assumption)$ in the proof tree, then A_1 was arbitrarily chosen by some application of rule (mp). In this case we can also use assumption Δ, and thus $\Delta \vdash_{SA} A_2$ holds.

   Otherwise, there must be leaves $\dfrac{}{A_1 \vdash_{SA} A_1}(assumption)$ in the proof tree. We replace these leaves by the proof tree for $\Delta \vdash_{SA} A_1$. Because A_1 is no more needed as assumption, we replace it by Δ in the whole proof tree and get a proof tree for $\Delta \vdash_{SA} A_2$. This replacement does not influence the proof tree because, except for the assumption leaves, it is independent of the assumptions.

ii) The argumentation is similar to i). We only have to replace all used assumptions of Δ_2 by the proof tree for $\Delta_1 \vdash_{SA} \Delta_2$.

□

Proposition 4.1.1 *Interpretation equivalent sort terms*

Let $\tau \in \mathrm{T}_\Omega(\chi)$ be a monomorphic sort expression, and let σ be a sort variable substitution. Let ν and ν^σ be sort variable assignments.

$$\text{If for each } \alpha \in \chi : \nu^\sigma(\alpha) = \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu \text{ then } \mathcal{SA}[\![\tau]\!]_{\nu^\sigma} = \mathcal{SA}[\![\sigma(\tau)]\!]_\nu.$$

Proof: By induction on the structure of τ.

Base cases: • $\tau = \alpha$, $\alpha \in \chi$

$$\mathcal{SA}[\![\alpha]\!]_{\nu^\sigma} \overset{\text{Def. 4.1.3}}{=} \nu^\sigma(\alpha) \overset{\text{hypothesis}}{=} \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu$$

- $\tau = c,\ c \in SC_0$

  $\mathcal{SA}[\![c]\!]_{\nu^\sigma} \overset{\text{Def. 4.1.3}}{=} c^{\mathcal{SA}} \overset{\text{Def. 4.1.3}}{=} \mathcal{SA}[\![c]\!]_{\nu} \overset{\text{Def. 3.1.4}}{=} \mathcal{SA}[\![\sigma(c)]\!]_{\nu}$

Inductive case: $\tau = sc(\tau_1, \ldots, \tau_n),\ sc \in SC_n$

$$\begin{aligned}
\mathcal{SA}[\![sc(\tau_1, \ldots, \tau_n)]\!]_{\nu^\sigma} &\overset{\text{Def. 4.1.3}}{=} sc^{\mathcal{SA}}(\mathcal{SA}[\![\tau_1]\!]_{\nu^\sigma}, \ldots, \mathcal{SA}[\![\tau_n]\!]_{\nu^\sigma}) \\
&\overset{\text{ind. hypoth.}}{=} sc^{\mathcal{SA}}(\mathcal{SA}[\![\sigma(\tau_1)]\!]_{\nu}, \ldots, \mathcal{SA}[\![\sigma(\tau_n)]\!]_{\nu}) \\
&\overset{\text{Def. 4.1.3}}{=} \mathcal{SA}[\![sc(\sigma(\tau_1), \ldots, \sigma(\tau_n))]\!]_{\nu} \\
&\overset{\text{Def. 3.1.4}}{=} \mathcal{SA}[\![\sigma(sc(\tau_1, \ldots, \tau_n))]\!]_{\nu}
\end{aligned}$$

□

Proposition 4.1.2 *Interpretation equivalent sort clauses*

Let $C \in \mathrm{C}_\Omega(\chi)$ be a sort clause, and let Θ be a set of sort clauses. Let σ be a sort variable substitution. Let ν and ν^σ be sort variable assignments.

If for each $\alpha \in \chi : \nu^\sigma(\alpha) = \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu$ then

i) $\models_{\mathcal{SA},\nu^\sigma} C$ iff $\models_{\mathcal{SA},\nu} \sigma(C)$

ii) $\models_{\mathcal{SA},\nu^\sigma} \Theta$ iff $\models_{\mathcal{SA},\nu} \sigma(\Theta)$

Proof:

i) Because of Def. 4.1.5 we have to prove that $\mathcal{SA}[\![C]\!]_{\nu^\sigma} = \mathcal{SA}[\![\sigma(C)]\!]_\nu$. We prove this by case analysis.

- $C = sp(\tau_1, \ldots, \tau_n),\ sp \in SP_n$:

$$\begin{aligned}
\mathcal{SA}[\![sp(\tau_1, \ldots, \tau_n)]\!]_{\nu^\sigma} &\overset{\text{Def. 4.1.4}}{=} sp^{\mathcal{SA}}(\mathcal{SA}[\![\tau_1]\!]_{\nu^\sigma}, \ldots, \mathcal{SA}[\![\tau_n]\!]_{\nu^\sigma}) \\
&\overset{\text{Prop. 4.1.1}}{=} sp^{\mathcal{SA}}(\mathcal{SA}[\![\sigma(\tau_1)]\!]_{\nu}, \ldots, \mathcal{SA}[\![\sigma(\tau_n)]\!]_{\nu}) \\
&\overset{\text{Def. 4.1.4}}{=} \mathcal{SA}[\![sp(\sigma(\tau_1), \ldots, \sigma(\tau_n))]\!]_{\nu} \\
&\overset{\text{Def. 3.1.6}}{=} \mathcal{SA}[\![\sigma(sp(\tau_1, \ldots, \tau_n))]\!]_{\nu}
\end{aligned}$$

- $C = B_1, \ldots, B_n \Rightarrow H$:

$$\mathcal{SA}[\![B_1, \ldots, B_n \Rightarrow H]\!]_{\nu^\sigma} \overset{\text{Def. 4.1.4}}{=} \begin{cases} \mathit{false} & \text{if } \mathcal{SA}[\![H]\!]_{\nu^\sigma} = \mathit{false} \text{ and} \\ & \mathcal{SA}[\![B_i]\!]_{\nu^\sigma} = \mathit{true},\ 1 \le \mathrm{i} \le \mathrm{n} \\ \mathit{true} & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathcal{SA}[\![\sigma(B_1, \ldots, B_n \Rightarrow H)]\!]_{\nu} &\overset{\text{Def. 3.1.8}}{=} \mathcal{SA}[\![\sigma(B_1), \ldots, \sigma(B_n) \Rightarrow \sigma(H)]\!]_{\nu} \\
&\overset{\text{Def. 4.1.4}}{=} \begin{cases} \mathit{false} & \text{if } \mathcal{SA}[\![\sigma(H)]\!]_{\nu} = \mathit{false} \text{ and} \\ & \mathcal{SA}[\![\sigma(B_i)]\!]_{\nu} = \mathit{true},\ 1 \le i \le n \\ \mathit{true} & \text{otherwise} \end{cases}
\end{aligned}$$

By the first case we know that $\mathcal{SA}[\![H]\!]_{\nu^\sigma} = \mathcal{SA}[\![\sigma(H)]\!]_\nu$ and $\mathcal{SA}[\![B_i]\!]_{\nu^\sigma} = \mathcal{SA}[\![\sigma(B_i)]\!]_\nu$. Therefore, $\mathcal{SA}[\![B_1,\ldots,B_n \Rightarrow H]\!]_{\nu^\sigma} = \mathcal{SA}[\![\sigma(B_1,\ldots,B_n \Rightarrow H)]\!]_\nu$

ii) $$\begin{array}{lll} \models_{\mathcal{SA},\nu^\sigma} \Theta & \overset{\text{Def. 4.1.5}}{\Leftrightarrow} & \models_{\mathcal{SA},\nu^\sigma} C \text{ for each } C \in \Theta \\ & \overset{\text{i)}}{\Leftrightarrow} & \models_{\mathcal{SA},\nu} \sigma(C) \text{ for each } C \in \Theta \\ & \overset{\text{Def. 4.1.5}}{\Leftrightarrow} & \models_{\mathcal{SA},\nu} \sigma(\Theta) \end{array}$$

□

Proposition 4.1.3 *Soundness and completeness of entailment relation* $\vdash$

Let $S = (\Omega, SA)$ be a sort specification. Let $\Delta \subseteq \mathrm{AF}_\Omega(\Phi)$ be a set of atomic sort formulae and $A \in \mathrm{AF}_\Omega(\Phi)$ be an atomic sort formula.

1. The entailment relation $\vdash$ is *sound* with respect to the logical consequence relation $\models$, i.e.

$$\text{if} \quad \Delta \vdash_{SA} A \quad \text{then} \quad \Delta \models_{\mathcal{SA}} A \text{ for each model } \mathcal{SA} \text{ of } S$$

2. The entailment relation $\vdash$ is *complete* with respect to the logical consequence relation $\models$, i.e.

$$\text{if} \quad \Delta \models_{\mathcal{SA}} A \text{ for each model } \mathcal{SA} \text{ of } S \quad \text{then} \quad \Delta \vdash_{SA} A$$

3. Both propositions can be extended to sets of atomic sort formulae Δ_1 and Δ_2, i.e.

$$\text{if} \quad \Delta_1 \vdash_{SA} \Delta_2 \quad \text{then} \quad \Delta_1 \models_{\mathcal{SA}} \Delta_2 \text{ for each model } \mathcal{SA} \text{ of } S$$

$$\text{if} \quad \Delta_1 \models_{\mathcal{SA}} \Delta_2 \text{ for each model } \mathcal{SA} \text{ of } S \quad \text{then} \quad \Delta_1 \vdash_{SA} \Delta_2$$

Proof:

1. Under assumption $\Delta \vdash_{SA} A$ we have to prove:

$$\begin{array}{lll} \Delta \models_{\mathcal{SA}} A \quad \text{for each model } \mathcal{SA} \text{ of } S & \overset{\text{Def. 4.1.7}}{\Leftrightarrow} & \\ \Delta \models_{\mathcal{SA},\nu} A \quad \text{for each } \nu : \chi \to \mathcal{U} \quad \text{for each model } \mathcal{SA} \text{ of } S & \overset{\text{Def. 4.1.7}}{\Leftrightarrow} & \\ \models_{\mathcal{SA},\nu} \Delta \;\Rightarrow\; \models_{\mathcal{SA},\nu} A \quad \text{for each } \nu : \chi \to \mathcal{U} \quad \text{for each model } \mathcal{SA} \text{ of } S & & \end{array}$$

Let $\mathcal{SA}$ be an arbitrary model of S, and let ν be an arbitrary sort variable assignment. Because we assumed $\Delta \vdash_{SA} A$, there must be a finite proof tree ending with $\Delta \vdash_{SA} A$. Let $P = \Delta \bigtriangledown_{SA} A$ be such a proof tree. By induction on the structure of the proof tree we show that $\models_{\mathcal{SA},\nu} \Delta \;\Rightarrow\; \models_{\mathcal{SA},\nu} A$ holds under assumption $\Delta \bigtriangledown_{SA} A$.

- $P = \frac{}{\Delta, C \vdash_{SA} C}(assumption)$

  We must show: $\models_{\mathcal{SA},\nu} \Delta \cup \{C\} \;\Rightarrow\; \models_{\mathcal{SA},\nu} C \quad \Leftarrow$
  $\models_{\mathcal{SA},\nu} C \;\Rightarrow\; \models_{\mathcal{SA},\nu} C$

  This is trivially true.

- $P = \frac{\Delta \bigtriangledown_{SA} \sigma(B_i)\ \ 1 \leq i \leq n}{\Delta \vdash_{SA} \sigma(H)}(mp)$

  We must show: $\models_{\mathcal{SA},\nu} \Delta \;\Rightarrow\; \models_{\mathcal{SA},\nu} \sigma(H)$ $\overset{\text{Def. 4.1.5}}{\Leftrightarrow}$
  $\models_{\mathcal{SA},\nu} \Delta \;\Rightarrow\; \mathcal{SA}[\![\sigma(H)]\!]_\nu = true$

  We assume $\models_{\mathcal{SA},\nu} \Delta$ and show $\mathcal{SA}[\![\sigma(H)]\!]_\nu = true$:

  Because $\mathcal{SA}$ is a model of S we know from Def. 4.1.6:

  $$\begin{array}{ll}
  \models_{\mathcal{SA}} SA & \overset{\text{Def. 4.1.5}}{\Leftrightarrow} \\
  \models_{\mathcal{SA}} C \text{ for each } C \in SA & \overset{*)}{\Rightarrow} \\
  \models_{\mathcal{SA}} B_1, \ldots, B_n \Rightarrow H & \overset{\text{Def. 4.1.5}}{\Leftrightarrow} \\
  \models_{\mathcal{SA},\nu} B_1, \ldots, B_n \Rightarrow H \text{ for each } \nu : \chi \to \mathcal{U} & \overset{\text{Def. 4.1.5}}{\Leftrightarrow} \\
  \mathcal{SA}[\![B_1, \ldots, B_n \Rightarrow H]\!]_\nu = true \text{ for each } \nu : \chi \to \mathcal{U} & \overset{\text{Def. 4.1.4}}{\Leftrightarrow} \\
  \mathcal{SA}[\![B_i]\!]_\nu = true, 1 \leq i \leq n \Rightarrow \mathcal{SA}[\![H]\!]_\nu = true \text{ for each } \nu : \chi \to \mathcal{U} & \overset{\nu^\sigma(\alpha) := \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu}{\Rightarrow} \\
  \mathcal{SA}[\![B_i]\!]_{\nu^\sigma} = true, 1 \leq i \leq n \Rightarrow \mathcal{SA}[\![H]\!]_{\nu^\sigma} = true & \overset{\text{Prop. 4.1.2}}{\Leftrightarrow} \\
  \mathcal{SA}[\![\sigma(B_i)]\!]_\nu = true, 1 \leq i \leq n \Rightarrow \mathcal{SA}[\![\sigma(H)]\!]_\nu = true &
  \end{array}$$

  ad $*$) By the side condition of rule *(mp)* $B_1, \ldots, B_n \Rightarrow H \in SA$.
  By induction we know that

  $$\begin{array}{ccc}
  \models_{\mathcal{SA},\nu} \Delta & \Rightarrow & \models_{\mathcal{SA},\nu} \sigma(B_i), 1 \leq i \leq n \\
  & \overset{\text{We assumed } \models_{\mathcal{SA},\nu} \Delta}{\Rightarrow} & \models_{\mathcal{SA},\nu} \sigma(B_i), 1 \leq i \leq n \\
  & \overset{\text{Def. 4.1.5}}{\Leftrightarrow} & \mathcal{SA}[\![\sigma(B_i)]\!]_\nu = true, 1 \leq i \leq n
  \end{array}$$

  Thus, $\mathcal{SA}[\![\sigma(H)]\!]_\nu = true$ follows immediately.

2. We do not need the completeness property in this thesis. Thus, we omit the proof.

3. The proofs can be easily extended to sets of atomic sort formulae.

□

Proposition 4.1.4 *Satisfaction implying atomic sort formulae*

Let $S = (\Omega, SA)$ be a sort specification, and let $\mathcal{SA}$ be a model of S. Let Δ_1, $\Delta_2 \subseteq \mathrm{AF}_\Omega(\chi)$ be two sets of atomic sort formulae. Let σ be a sort variable substitution

such that $\Delta_1 \vdash_{SA} \sigma(\Delta_2)$. Let ν and ν^σ be sort variable assignments such that $\nu^\sigma(\alpha) = \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu$ for each $\alpha \in \chi$.

$$\text{If} \quad \models_{\mathcal{SA},\nu} \Delta_1 \quad \text{then} \quad \models_{\mathcal{SA},\nu^\sigma} \Delta_2$$

Proof:

$$\begin{array}{lll} \Delta_1 \vdash_{SA} \sigma(\Delta_2) & \overset{\text{Prop. 4.1.3}}{\Rightarrow} & \Delta_1 \models_{\mathcal{SA}} \sigma(\Delta_2) \\ & \overset{\text{Def. 4.1.7}}{\Rightarrow} & \Delta_1 \models_{\mathcal{SA},\nu} \sigma(\Delta_2) \\ & \overset{\text{Def. 4.1.7}}{\Leftrightarrow} & \text{if} \ \models_{\mathcal{SA},\nu} \Delta_1 \ \text{then} \models_{\mathcal{SA},\nu} \sigma(\Delta_2) \\ & \overset{\text{Prop. 4.1.2}}{\Leftrightarrow} & \text{if} \ \models_{\mathcal{SA},\nu} \Delta_1 \ \text{then} \models_{\mathcal{SA},\nu^\sigma} \Delta_2 \end{array}$$

□

Proposition 4.2.1 *Sort correct updating of a variable assignment*

Let Γ be a variable assumption, and let ν be a sort variable assignment. Let η be a variable assignment such that $\eta \models_{\mathcal{SA},\nu} \Gamma$.

$$d \in \mathcal{SA}[\![\tau]\!]_\nu \quad \Rightarrow \quad \eta.[x \mapsto d] \models_{\mathcal{SA},\nu} \Gamma.x{:}\tau$$

Proof: Under assumptions $\eta \models_{\mathcal{SA},\nu} \Gamma$ and $d \in \mathcal{SA}[\![\tau]\!]_\nu$ we have to prove:

$$\begin{array}{ll} \eta.[x \mapsto d] \models_{\mathcal{SA},\nu} \Gamma.x{:}\tau & \overset{\text{Def. 4.2.2}}{\Leftrightarrow} \\ \text{for each } x'{:}\tau' \in \Gamma.x{:}\tau.\ \eta.[x \mapsto d](x') \in \mathcal{SA}[\![\tau']\!]_\nu & \overset{\text{Def. 4.2.2}}{\Leftrightarrow} \\ \text{for each } x'{:}\tau' \in \Gamma.x{:}\tau. \begin{cases} d \in \mathcal{SA}[\![\tau']\!]_\nu & \text{if } x = x' \\ \eta(x') \in \mathcal{SA}[\![\tau']\!]_\nu & \text{otherwise} \end{cases} & \end{array}$$

The first case $d \in \mathcal{SA}[\![\tau']\!]_\nu$ follows because $\tau' = \tau$ and we assumed that $d \in \mathcal{SA}[\![\tau]\!]_\nu$. The second case $\eta(x') \in \mathcal{SA}[\![\tau']\!]_\nu$ follows from the assumption $\eta \models_{\mathcal{SA},\nu} \Gamma$. □

Proposition 4.2.2 *The interpretation function* $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ *is well-defined*

Let $D = \Delta, \Gamma \ \triangledown_\Sigma \ e :: \tau$ be a sort derivation for a pre-term e. Let ν be a sort variable assignment and η be a variable assignment.

$$\begin{array}{c} \text{If} \models_{\mathcal{SA},\nu} \Delta \text{ and } \eta \models_{\mathcal{SA},\nu} \Gamma \text{ then} \\ \mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta} \text{ yields a uniquely determined result in } \mathcal{SA}[\![\tau]\!]_\nu. \end{array}$$

Proof: By induction on the structure of D.

Base cases

- $\mathcal{A}\left[\!\left[\dfrac{}{\Delta, \Gamma.x{:}\tau \rhd_\Sigma x :: \tau}\ (var)\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.3}}{=} \eta(\mathrm{x})$

  By Def. 4.2.2 the variable assignment η is a totally defined function. Furthermore, $\eta(\mathrm{x})$ is an element of $\mathcal{SA}[\![\tau]\!]_\nu$ because we assumed $\eta \models_{\mathcal{SA},\nu} \Gamma.x{:}\tau$.

- $\mathcal{A}\left[\!\left[\dfrac{}{\Delta, \Gamma \rhd_\Sigma c :: \tau}\ (id)\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.3}}{=} \mathcal{F}(c)$

  By Def. 4.2.1 $\mathcal{F}$ is a mapping that assigns each $c{:}\tau \in F$ a value in $\mathcal{SA}[\![\tau]\!]_\nu$.

- $\mathcal{A}\left[\!\left[\dfrac{}{\Delta, \Gamma \rhd_\Sigma c :: \sigma(\tau)}\ (polyid)\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.3}}{=} \mathcal{F}(c)[\mathcal{SA}[\![\sigma(\alpha_1)]\!]_\nu, \ldots, \mathcal{SA}[\![\sigma(\alpha_n)]\!]_\nu]$

  Again, by Def. 4.2.1 $\mathcal{F}$ is a mapping yielding an element in $(\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau)^{\mathcal{SA}}$ for each $c{:}\Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau \in F$. As already discussed, the substitution σ is uniquely determined for the sort variables α_i, $1{\le}i{\le}n$. Therefore, the parameters $\mathcal{SA}[\![\sigma(\alpha_i)]\!]_\nu$ are also uniquely determined. Next we have to prove whether the application of the polymorphic function is defined. By Def. 4.1.8 a function in $(\Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau)^{\mathcal{SA}}$ only yields a defined value if

  $$\models_{\mathcal{SA},\nu'} A_j,\ 1 \le j \le m$$
  $$\text{where } \nu' = [\alpha_i \mapsto \mathcal{SA}[\![\sigma(\alpha_i)]\!]_\nu],\ 1{\le}i{\le}n \quad \Leftrightarrow \quad \nu'(\alpha_i){=}\mathcal{SA}[\![\sigma(\alpha_i)]\!]_\nu,\ 1{\le}i{\le}n$$

  Because of the side conditions of rule $(polyid)$ we know:

  $$\sigma(A_j) \in \Delta,\ 1 \le j \le m$$

  Since we assumed $\models_{\mathcal{SA},\nu} \Delta$, we know that $\models_{\mathcal{SA},\nu} \sigma(A_j)$. By Def. 3.1.10 $A_j \in \mathrm{AF}_\Omega(\{\alpha_1, \ldots, \alpha_n\})$. Therefore, we know by Prop. 4.1.2:

  $$\text{if} \quad \nu'(\alpha_i){=}\mathcal{SA}[\![\sigma(\alpha_i)]\!]_\nu \quad \text{then} \quad \models_{\mathcal{SA},\nu} \sigma(A_j) \Leftrightarrow \models_{\mathcal{SA},\nu'} A_j$$

  Thus, the application of the polymorphic function yields a defined value. Finally, it remains to show that the application yields a value in $\mathcal{SA}[\![\sigma(\tau)]\!]_\nu$. By Def. 4.1.8 the result of applying a polymorphic function from $(\Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau)^{\mathcal{SA}}$ to $(\mathcal{SA}[\![\sigma(\alpha_1)]\!]_\nu, \ldots, \mathcal{SA}[\![\sigma(\alpha_n)]\!]_\nu)$ yields an element in $\mathcal{SA}[\![\tau]\!]_{\nu^\sigma}$, where $\nu^\sigma(\alpha_i) = \mathcal{SA}[\![\sigma(\alpha_i)]\!]_\nu$, $1{\le}i{\le}n$. By Def. 3.1.10 $\tau \in \mathrm{T}_\Omega(\{\overline{\alpha_n}\})$. Therefore, by Prop. 4.1.1 $\mathcal{SA}[\![\tau]\!]_{\nu^\sigma} = \mathcal{SA}[\![\sigma(\tau)]\!]_\nu$ holds. This finishes the proof of case $(polyid)$.

Inductive cases

- $\mathcal{A}\left[\!\left[\dfrac{\Delta, \Gamma\ \triangledown_\Sigma\ e_1 :: \tau_2 \rightarrow \tau_1 \quad \Delta, \Gamma\ \triangledown_\Sigma\ e_2 :: \tau_2}{\Delta, \Gamma \rhd_\Sigma e_1 e_2 :: \tau_1}\ (\rightarrow E)\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.3}}{=}$
  $\mathcal{A}\left[\!\left[\Delta, \Gamma\ \triangledown_\Sigma\ e_1 :: \tau_2 \rightarrow \tau_1\right]\!\right]_{\nu,\eta} (\mathcal{A}\left[\!\left[\Delta, \Gamma\ \triangledown_\Sigma\ e_2 :: \tau_2\right]\!\right]_{\nu,\eta})$

  By induction we know that $\mathcal{A}\left[\!\left[\Delta, \Gamma\ \triangledown_\Sigma\ e_1 :: \tau_2 \rightarrow \tau_1\right]\!\right]_{\nu,\eta}$ yields a uniquely determined value in

$$\mathcal{SA}[\![\tau_2 \rightarrow \tau_1]\!]_\nu \overset{\text{Def. } 3.1.4}{=} \mathcal{SA}[\![\tau_2]\!]_\nu \rightarrow^{\mathcal{SA}} \mathcal{SA}[\![\tau_1]\!]_\nu$$

and $\mathcal{A}\left[\!\left[\Delta,\, \Gamma \bigtriangledown_\Sigma e_2 :: \tau_2\right]\!\right]_{\nu,\eta}$ yields a uniquely determined value in $\mathcal{SA}[\![\tau_2]\!]_\nu$. Because $\rightarrow^{\mathcal{SA}}$ only contains totally defined functions, the application yields a uniquely determined value in $\mathcal{SA}[\![\tau_1]\!]_\nu$.

- $\mathcal{A}\left[\!\left[\dfrac{\Delta,\, \Gamma.x{:}\tau_1 \bigtriangledown_\Sigma e :: \tau_2}{\Delta, \Gamma \rhd_\Sigma \lambda x.e :: \tau_1 \rightarrow \tau_2}\ (\rightarrow I_u)\right]\!\right]_{\nu,\eta} \overset{\text{Def. } 4.2.3}{=}$ the unique $f \in \mathcal{SA}[\![\tau_1 \rightarrow \tau_2]\!]_\nu$ such that $\forall d \in \mathcal{SA}[\![\tau_1]\!]_\nu . f(d) = \mathcal{A}\left[\!\left[\Delta,\, \Gamma.x{:}\tau_1 \bigtriangledown_\Sigma e :: \tau_2\right]\!\right]_{\nu,\eta.[x \mapsto d]}$

  Because $d \in \mathcal{SA}[\![\tau_1]\!]_\nu$ and $\eta \models_{\mathcal{SA},\nu} \Gamma$, we know that $\eta.[x \mapsto d] \models_{\mathcal{SA},\nu} \Gamma.x{:}\tau_1$. Therefore, we can apply induction hypothesis. As result we know that $\mathcal{A}\left[\!\left[\Delta,\, \Gamma.x{:}\tau_1 \bigtriangledown_\Sigma e :: \tau_2\right]\!\right]_{\nu,\eta.[x \mapsto d]}$ yields a uniquely determined value in $\mathcal{SA}[\![\tau_2]\!]_\nu$. Because we assumed the extensionality property for our function space constructor, the function f is indeed uniquely determined by its action on the arguments d from $\mathcal{SA}[\![\tau_1]\!]_\nu$. It remains to prove that f is always an element of $\mathcal{SA}[\![\tau_1 \rightarrow \tau_2]\!]_\nu$. If this property holds, the function space $\rightarrow^{\mathcal{SA}}$ is called *closed with respect to λ-abstraction*. If $\rightarrow$ is interpreted as the full function space, as we assumed in Def. 4.1.1, it clearly holds, because $\mathcal{SA}[\![\tau_1 \rightarrow \tau_2]\!]_\nu$ contains all possible functions from $\mathcal{SA}[\![\tau_1]\!]_\nu$ to $\mathcal{SA}[\![\tau_2]\!]_\nu$. However, if we use a subset of the full function space the proposition only holds in case this subset is closed with respect to λ-abstraction.

- $(\rightarrow I_s)$ like $(\rightarrow I_u)$

- $\mathcal{A}\left[\!\left[\dfrac{\Delta,\, \Gamma \bigtriangledown_\Sigma e :: \tau}{\Delta, \Gamma \rhd_\Sigma e{:}\tau :: \tau}\ (\mathit{constrained})\right]\!\right]_{\nu,\eta} \overset{\text{Def. } 4.2.3}{=} \mathcal{A}\left[\!\left[\Delta,\, \Gamma \bigtriangledown_\Sigma e :: \tau\right]\!\right]_{\nu,\eta}$

  By induction $\mathcal{A}\left[\!\left[\Delta,\, \Gamma \bigtriangledown_\Sigma e :: \tau\right]\!\right]_{\nu,\eta}$ is a uniquely determined value from $\mathcal{SA}[\![\tau]\!]_\nu$.

□

Proposition 4.2.3 *The interpretation function $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ is well-defined*

Let $D = \Delta, \Gamma \bigtriangledown_\Sigma f$ be a sort derivation for a pre-formula f. Let ν be a sort variable assignment and η be a variable assignment.

$$\text{If } \models_{\mathcal{SA},\nu} \Delta \text{ and } \eta \models_{\mathcal{SA},\nu} \Gamma \text{ then}$$
$$\mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta} \text{ yields a uniquely determined result in } \{\mathit{true}, \mathit{false}\}.$$

Proof: By induction on the structure of D.

Base cases

- $\mathcal{A}\left[\!\left[\dfrac{}{\Delta, \Gamma \rhd_\Sigma p}\ (const)\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.4}}{=} \mathcal{P}(p)$

  By Def. 4.2.1 $\mathcal{P}$ assigns a uniquely determined truth value from $\{true, false\}$ to each $p \in P_\epsilon$.

- $\mathcal{A}\left[\!\left[\dfrac{\Delta, \Gamma \ \bigtriangledown_\Sigma\ t :: \tau}{\Delta, \Gamma \rhd_\Sigma p\, t}\ (appl)\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.4}}{=} \mathcal{P}(p)(\mathcal{A}\left[\!\left[\Delta, \Gamma \ \bigtriangledown_\Sigma\ t :: \tau\right]\!\right]_{\nu,\eta})$

  By Def. 4.2.1 $\mathcal{P}$ is a mapping that assigns each $p{:}\tau \in P$ a totally defined predicate in $((\tau) \mapsto \mathrm{TV})^{\mathcal{SA}} = \mathcal{SA}[\![\tau]\!]_\nu \mapsto^{\mathcal{SA}} \{true, false\}$. By Prop. 4.2.2 $\mathcal{A}\left[\!\left[\Delta, \Gamma \ \bigtriangledown_\Sigma\ t :: \tau\right]\!\right]_{\nu,\eta}$ yields a uniquely determined value in $\mathcal{SA}[\![\tau]\!]_\nu$. Because $\mapsto^{\mathcal{SA}}$ is the full function space constructor, the application is defined and yields a uniquely determined result in $\{true, false\}$.

- $\mathcal{A}\left[\!\left[\dfrac{\Delta, \Gamma \ \bigtriangledown_\Sigma\ t :: \sigma(\tau)}{\Delta, \Gamma \rhd_\Sigma p\, t}\ (pappl)\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.4}}{=}$
  $\mathcal{P}(p)\ [\mathcal{SA}[\![\sigma(\alpha_1)]\!]_\nu, \ldots, \mathcal{SA}[\![\sigma(\alpha_n)]\!]_\nu]\ (\mathcal{A}\left[\!\left[\Delta, \Gamma \ \bigtriangledown_\Sigma\ t :: \sigma(\tau)\right]\!\right]_{\nu,\eta})$

  Again, by Def. 4.2.1 $\mathcal{P}$ is a mapping that assigns a polymorphic predicate in $(\Pi\overline{\alpha_n}.\overline{B_m} \Rightarrow (\tau) \mapsto \mathrm{TV})^{\mathcal{SA}}$ to p. As in case $(polyid)$ of Prop. 4.2.2 the substitution σ is uniquely determined for all α_i, $1 \leq i \leq n$. Therefore, the parameters of the polymorphic predicate are also uniquely determined. Next we have to check whether the application of the polymorphic predicate to the domains is defined and whether the result is a totally defined predicate in $\mathcal{SA}[\![(\sigma(\tau) \mapsto \mathrm{TV})]\!]_\nu = \mathcal{SA}[\![\sigma(\tau)]\!]_\nu \mapsto^{\mathcal{SA}} \{true, false\}$. The proof is the same as in case $(polyid)$ of Prop. 4.2.2. Finally, by Prop. 4.2.2 $\mathcal{A}\left[\!\left[\Delta, \Gamma \ \bigtriangledown_\Sigma\ t :: \sigma(\tau)\right]\!\right]_{\nu,\eta}$ yields a uniquely determined value in $\mathcal{SA}[\![\sigma(\tau)]\!]_\nu$. Because $\mapsto^{\mathcal{SA}}$ is the full function space constructor, the application is defined and yields a uniquely determined result in $\{true, false\}$.

Inductive cases The inductive cases $(univ_u)$, $(univ_s)$, (not), and (or) can easily be proved by using induction hypothesis. The interpretation is always defined by a case statement yielding either the value *true* or *false*. Furthermore, each condition is uniquely determined and the patterns in the case statements do not overlap. Thus, the interpretation always yields a uniquely determined value in $\{true, false\}$. □

Proposition 4.2.4 *The interpretation function* $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ *is well-defined*

Let $D = \Delta, \Gamma \ \bigtriangledown_\Sigma\ \forall\overline{\alpha_n}.\ \overline{A_m} \Rightarrow f$ be a sort derivation for a qualified formula. Let ν be a sort variable assignment and η be a variable assignment.

If $\eta \models_{\mathcal{SA},\nu} \Gamma$ then $\mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta}$ yields a uniquely determined result in $\{true, false\}$.

Proof: It is clear that the result is either *true* or *false*. We only have to check whether the interpretation defined in Def. 4.2.5 is unambiguous. The interpretation is defined by a case statement. Because we used the word "otherwise" in the second case the cases clearly do not overlap. Therefore, it remains to show that the condition used in the "*true*"-case yields a unique truth value. The satisfaction relation $\models$ is uniquely defined. Furthermore, because of the side condition of rule *(qualification)*:

$$\begin{array}{lll} \overline{A_m} \vdash_{SA} \Delta & \overset{\text{Prop. 4.1.3}}{\Rightarrow} & \overline{A_m} \models_{\mathcal{SA}} \Delta \\ & \Rightarrow & \overline{A_m} \models_{\mathcal{SA},\nu} \Delta \\ & \overset{\text{Def. 4.1.7}}{\Leftrightarrow} & \text{if } \models_{\mathcal{SA},\nu} \overline{A_m} \text{ then } \models_{\mathcal{SA},\nu} \Delta \end{array}$$

If the sort variable assignment does not satisfy the sort predicate context, the whole condition is true. Otherwise, by Prop. 4.2.3 $\mathcal{A}\left[\!\left[\Delta, \Gamma \bigtriangledown_\Sigma f\right]\!\right]_{\nu,\eta}$ yields a uniquely determined truth value because the assumption $\models_{\mathcal{SA},\nu} \Delta$ holds. □

Proposition 4.3.1 *Uniqueness of sort derivation structure*

Let $f \in \mathrm{PF}_\Sigma(\Phi) \cup \mathrm{QF}_\Sigma$ be an arbitrary formula. Every sort derivation for f has the same structure according to the judgement rules given in Def. 3.2.7, 3.2.9 and 3.2.11. More precisely, in all sort derivations the same judgement rules are applied in the same order.

Proof:

The three judgement calculi are all syntax-directed, i.e. for each syntactic unit of the term and formula language there is exactly one inference rule that can be applied. Therefore, the structure of a derivation is uniquely determined by the syntactic structure of the formula and term, respectively. □

Proposition 4.3.2 *Uniqueness of sort derivations*

Let $f \in \mathrm{SEN}_\Sigma$ be a closed well-formed Σ-formula. If every bound variable is sorted explicitly, and if no polymorphic function or predicate is applied in the formula, then, up to different sort predicate assumptions, there exists only one sort derivation for f. In particular, the sort predicate assumption can be chosen arbitrarily.

Proof:

The proposition is straightforward to show. By Prop. 4.3.1 we know that the trees can only differ in their nodes. Because of the restrictions we cannot use rules (*polyid*) and $(\rightarrow I_u)$ from Def. 3.2.7 nor rules (*pappl*) and $(univ_u)$ from Def. 3.2.9 to prove the well-formedness of the formula. But all other rules do not allow to choose between different

sort terms in the derivation. Furthermore, the sort predicate assumption is not involved in these rules and thus can be chosen arbitrarily. □

Proposition 4.3.3 *σ-instances propagate to sort terms*

Let $D_1 = \Delta_1, \Gamma_1 \bigtriangledown_\Sigma e :: \tau_1$ and $D_2 = \Delta_2, \Gamma_2 \bigtriangledown_\Sigma e :: \tau_2$ be two sort derivations for a pre-term e. Let $\sigma : \Phi \to \mathrm{T}_\Omega(\Phi)$ be a sort variable substitution such that $\Gamma_1 = \sigma(\Gamma_2)$ and $\forall \alpha \in FSV(e).\ \sigma(\alpha) = \alpha$.

$$\text{If } D_1 \leq^\sigma D_2 \text{ then } \tau_1 = \sigma(\tau_2).$$

Proof: By induction on the structure of D_1

Base cases:

- $D_1 = \dfrac{}{\Delta_1, \Gamma_1.x{:}\tau_1 \rhd_\Sigma x :: \tau_1}\ (var) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{}{\Delta_2, \Gamma_2.x{:}\tau_2 \rhd_\Sigma x :: \tau_2}\ (var)$

  We assumed that $\Gamma_1.x{:}\tau_1 = \sigma(\Gamma_2.x{:}\tau_2) \overset{\text{Def. 3.2.6}}{\Rightarrow} \tau_1 = \sigma(\tau_2)$

- $D_1 = \dfrac{}{\Delta_1, \Gamma_1 \rhd_\Sigma c :: \tau_1}\ (id) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{}{\Delta_2, \Gamma_2 \rhd_\Sigma c :: \tau_2}\ (id)$

  Because c $\in \mathrm{F}_\tau$ and both derivations are with respect to Σ=(S,P,F) we know that $\tau_1 = \tau_2 = \tau$. Because $\tau \in \mathrm{T}_\Omega(\emptyset)$, τ does not contain any sort variable $\overset{\text{Def. 3.1.4}}{\Rightarrow}$ $\tau = \sigma(\tau)$.

- $D_1 = \dfrac{}{\Delta_1, \Gamma_1 \rhd_\Sigma c :: \tau_1}\ (polyid) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{}{\Delta_2, \Gamma_2 \rhd_\Sigma c :: \tau_2}\ (polyid)$

  $D_1 \leq^\sigma D_2 \overset{\text{Def. 4.3.2}}{\Rightarrow} \tau_1 = \sigma(\tau_2)$

Inductive cases:

- $D_1 = \dfrac{\Delta_1, \Gamma_1.x{:}\tau_{11} \bigtriangledown_\Sigma e :: \tau_{12}}{\Delta_1, \Gamma_1 \rhd_\Sigma \lambda x.\, e :: \tau_{11} \to \tau_{12}}\ (\to I_u) \overset{\text{Prop. 4.3.1}}{\Rightarrow}$

  $D_2 = \dfrac{\Delta_2, \Gamma_2.x{:}\tau_{21} \bigtriangledown_\Sigma e :: \tau_{22}}{\Delta_2, \Gamma_2 \rhd_\Sigma \lambda x.\, e :: \tau_{21} \to \tau_{22}}\ (\to I_u)$

  $$\begin{array}{lll} D_1 \leq^\sigma D_2 & \overset{\text{Def. 4.3.2}}{\Rightarrow} & \tau_{11} = \sigma(\tau_{21}) \\ D_1 \leq^\sigma D_2 & \overset{\text{Def. 4.3.2}}{\Rightarrow} & \Delta_1,\ \Gamma_1.x{:}\tau_{11} \bigtriangledown_\Sigma e :: \tau_{12} \leq^\sigma \Delta_2,\ \Gamma_2.x{:}\tau_{21} \bigtriangledown_\Sigma e :: \tau_{22} \\ \sigma(\Gamma_2.x{:}\tau_{21}) & \overset{\text{Def. 3.2.6}}{=} & \sigma(\Gamma_2).x{:}\sigma(\tau_{21}) \\ & = & \Gamma_1.x{:}\tau_{11} \end{array}$$

  $$\begin{array}{lll} \overset{\text{ind. hypoth.}}{\Rightarrow} & \tau_{12} = \sigma(\tau_{22}) & \\ \Rightarrow & \tau_{11} \to \tau_{12} = & \sigma(\tau_{21}) \to \sigma(\tau_{22}) \\ & \overset{\text{Def. 3.1.4}}{=} & \sigma(\tau_{21} \to \tau_{22}) \end{array}$$

- $D_1 = \dfrac{\Delta_1, \Gamma_1.x{:}\tau_1 \ \nabla_\Sigma \ e :: \tau_{12}}{\Delta_1, \Gamma_1 \rhd_\Sigma \lambda x{:}\tau_1 . \, e \ :: \tau_1 \to \tau_{12}} \ (\to I_s) \quad \overset{\text{Prop. 4.3.1}}{\Rightarrow}$

  $D_2 = \dfrac{\Delta_2, \Gamma_2.x{:}\tau_1 \ \nabla_\Sigma \ e :: \tau_{22}}{\Delta_2, \Gamma_2 \rhd_\Sigma \lambda x{:}\tau_1 . \, e \ :: \tau_1 \to \tau_{22}} \ (\to I_s)$

  $FSV(\tau_1) \subseteq FSV(\lambda x{:}\tau_1 . \, e) \quad \Rightarrow \quad \tau_1 \ = \ \sigma(\tau_1)$

  $D_1 \leq^\sigma D_2 \quad \overset{\text{Def. 4.3.2}}{\Rightarrow} \quad \Delta_1, \Gamma_1.x{:}\tau_1 \ \nabla_\Sigma \ e :: \tau_{12} \leq^\sigma \Delta_2, \Gamma_2.x{:}\tau_1 \ \nabla_\Sigma \ e :: \tau_{22}$

  $\sigma(\Gamma_2.x{:}\tau_1) \quad \overset{\text{Def. 3.2.6}}{=} \quad \sigma(\Gamma_2).x{:}\sigma(\tau_1) = \Gamma_1.x{:}\tau_1$

  $\overset{\text{ind. hypoth.}}{\Rightarrow} \quad \tau_{12} = \sigma(\tau_{22})$

  $\Rightarrow \quad \tau_1 \to \tau_{12} \quad \overset{\text{Def. 3.1.4}}{=} \quad \sigma(\tau_1) \to \sigma(\tau_{22}) = \sigma(\tau_1 \to \tau_{22})$

- $D_1 = \dfrac{\Delta_1, \Gamma_1 \ \nabla_\Sigma \ e :: \tau}{\Delta_1, \Gamma_1 \rhd_\Sigma e{:}\tau :: \tau} \ (constrained) \quad \overset{\text{Prop. 4.3.1}}{\Rightarrow}$

  $D_2 = \dfrac{\Delta_2, \Gamma_2 \ \nabla_\Sigma \ e :: \tau}{\Delta_2, \Gamma_2 \rhd_\Sigma e{:}\tau :: \tau} \ (constrained)$

  $FSV(\tau) \subseteq FSV(e{:}\tau) \ \Rightarrow \ \tau \ = \ \sigma(\tau)$

- $D_1 = \dfrac{\Delta_1, \Gamma_1 \ \nabla_\Sigma \ e_1 :: \tau_{12} \to \tau_{11} \quad \Delta_1, \Gamma_1 \ \nabla_\Sigma \ e_2 :: \tau_{12}}{\Delta_1, \Gamma_1 \rhd_\Sigma e_1 e_2 :: \tau_{11}} \ (\to E) \quad \overset{\text{Prop. 4.3.1}}{\Rightarrow}$

  $D_2 = \dfrac{\Delta_2, \Gamma_2 \ \nabla_\Sigma \ e_1 :: \tau_{22} \to \tau_{21} \quad \Delta_2, \Gamma_2 \ \nabla_\Sigma \ e_2 :: \tau_{22}}{\Delta_2, \Gamma_2 \rhd_\Sigma e_1 e_2 :: \tau_{21}} \ (\to E)$

  $D_1 \leq^\sigma D_2 \quad \overset{\text{Def. 4.3.2}}{\Rightarrow} \quad \Delta_1, \Gamma_1 \ \nabla_\Sigma \ e_1 :: \tau_{12} \to \tau_{11} \leq^\sigma \Delta_2, \Gamma_2 \ \nabla_\Sigma \ e_1 :: \tau_{22} \to \tau_{21}$

  $D_1 \leq^\sigma D_2 \quad \overset{\text{Def. 4.3.2}}{\Rightarrow} \quad \Delta_1, \Gamma_1 \ \nabla_\Sigma \ e_2 :: \tau_{12} \leq^\sigma \Delta_2, \Gamma_2 \ \nabla_\Sigma \ e_2 :: \tau_{22}$

  $\overset{\text{ind. hypoth.}}{\Rightarrow} \quad \tau_{12} \to \tau_{11} \ = \ \sigma(\tau_{22} \to \tau_{21})$

  $\Rightarrow \quad \tau_{11} \ = \ \sigma(\tau_{21})$

□

Proposition 4.3.5 *If a sort derivation is more general then it is more restrictive*

Let $D_1{=}\Delta_1, \emptyset \ \nabla_\Sigma \ f$ and $D_2{=}\Delta_2, \emptyset \ \nabla_\Sigma \ f$ be two sort derivations for a closed formula $f \in \mathrm{PF}_\Sigma(\Phi)$. If D_2 is more general than D_1 then D_2 is more restrictive than D_1, i.e.

$$D_1 \leq D_2 \quad \Rightarrow \quad D_2 \models D_1$$

Proof: Given assumption $D_1 \leq D_2$ we have to show that $D_2 \models D_1$. By Def. 4.3.1 we must prove that for every Σ algebra $\mathcal{A}$ holds:

$\mathcal{A} \models D_2 \ \Rightarrow \ \mathcal{A} \models D_1 \quad \overset{\text{Def. 4.2.7}}{\Leftrightarrow}$

$\mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu,\eta_0} = true$ for every $\nu : \Phi \to \mathcal{U}$. $\models_{\mathcal{SA},\nu} \Delta_2$

$\Rightarrow$

$\mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta_0} = true$ for every $\nu : \Phi \to \mathcal{U}$. $\models_{\mathcal{SA},\nu} \Delta_1$

where η_0 is an arbitrary variable assumption.

Let ν be an arbitrary but fixed sort variable assignment with $\models_{\mathcal{SA},\nu} \Delta_1$. If we can show that there exists a sort variable assignment ν^σ such that $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$ and $\mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta_0}$=$true$ $\Rightarrow$ $\mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta_0}$=$true$, we have proven the proposition. We prove this by constructing a sort variable assignment ν^σ. Because $D_1 \leq D_2$, by Def. 4.3.3 there exists a sort variable substitution σ such that $\forall\alpha \in FSV(f).\ \sigma(\alpha) = \alpha$ and $\Delta_1 \vdash_{SA} \sigma(\Delta_2)$ and $D_1 \leq^\sigma D_2$. We use this substitution to define ν^σ in the following way:

$$\nu^\sigma(\alpha) := \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu \quad \text{for all } \alpha \in \Phi$$

By Prop. 4.3.6 we know that $\mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} = \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta}$ if $\forall\alpha \in FSV(f).\ \sigma(\alpha) = \alpha$ and $D_1 \leq^\sigma D_2$ and $\nu^\sigma(\alpha)$=$\mathcal{SA}[\![\sigma(\alpha)]\!]_\nu$ and η|=$_{\mathcal{SA},\nu}$ Γ_1 and η|=$_{\mathcal{SA},\nu^\sigma}$ Γ_2 and $\models_{\mathcal{SA},\nu} \Delta_1$ and $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$. Thus, it remains to show:

i) η|=$_{\mathcal{SA},\nu}$ Γ_1 and η|=$_{\mathcal{SA},\nu^\sigma}$ Γ_2

ii) $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$

ad i) This is trivially valid because both Γ_1 and Γ_2 are empty.

ad ii) This follows by Prop. 4.1.4 because $\Delta_1 \vdash_{SA} \sigma(\Delta_2)$.

□

Proposition 4.3.6 *Interpretation equivalent sort derivations for formulae*

Let $D_1 = \Delta_1, \Gamma_1 \bigtriangledown_\Sigma f$ and $D_2 = \Delta_2, \Gamma_2 \bigtriangledown_\Sigma f$ be two sort derivations for a pre-formula $f \in \mathrm{PF}_\Sigma(\Phi)$. Let σ be a sort variable substitution such that $\forall\alpha \in FSV(f).\ \sigma(\alpha) = \alpha$ and $D_1 \leq^\sigma D_2$. Let ν and ν^σ be sort variable assignments such that $\models_{\mathcal{SA},\nu} \Delta_1$ and $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$. Let η be a variable assignment with η|=$_{\mathcal{SA},\nu}$ Γ_1 and η|=$_{\mathcal{SA},\nu^\sigma}$ Γ_2.

$$\text{If} \quad \nu^\sigma(\alpha)=\mathcal{SA}[\![\sigma(\alpha)]\!]_\nu \text{ for each } \alpha \in \Phi \quad \text{then} \quad \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} = \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta}$$

Proof: By induction on the structure of D_1.

Base cases:

- $D_1 = \dfrac{}{\Delta_1, \Gamma_1 \rhd_\Sigma p}\ (const) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{}{\Delta_2, \Gamma_2 \rhd_\Sigma p}\ (const)$

  $\mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.4}}{=} p^{\mathcal{A}}$

  $\overset{\text{Def. 4.2.4}}{=} \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta}$

- $D_1 = \dfrac{\Delta_1, \Gamma_1 \bigtriangledown_\Sigma t :: \tau_1}{\Delta_1, \Gamma_1 \rhd_\Sigma p\, t}\ (appl) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{\Delta_2, \Gamma_2 \bigtriangledown_\Sigma t :: \tau_2}{\Delta_2, \Gamma_2 \rhd_\Sigma p\, t}\ (appl)$

$$\begin{array}{lll} \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} & \overset{\text{Def. 4.2.4}}{=} & p^{\mathcal{A}}(\mathcal{A}\left[\!\left[\Delta_1, \Gamma_1 \bigtriangledown_\Sigma t :: \tau_1\right]\!\right]_{\nu,\eta}) \\ & \overset{*)}{=} & p^{\mathcal{A}}(\mathcal{A}\left[\!\left[\Delta_2, \Gamma_2 \bigtriangledown_\Sigma t :: \tau_2\right]\!\right]_{\nu^\sigma,\eta}) \\ & \overset{\text{Def. 4.2.4}}{=} & \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta} \end{array}$$

ad $*$) By Prop. 4.3.7 because

$$\begin{array}{rcll} D_1 & \leq^\sigma & D_2 & \overset{\text{Def. 4.3.2}}{\Rightarrow} \\ \Delta_1, \Gamma_1 \bigtriangledown_\Sigma t :: \tau_1 & \leq^\sigma & \Delta_2, \Gamma_2 \bigtriangledown_\Sigma t :: \tau_2 & \end{array}$$

- $D_1 = \dfrac{\Delta_1, \Gamma_1 \bigtriangledown_\Sigma t :: \tau_1}{\Delta_1, \Gamma_1 \rhd_\Sigma p\, t}\ (pappl) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{\Delta_2, \Gamma_2 \bigtriangledown_\Sigma t :: \tau_2}{\Delta_2, \Gamma_2 \rhd_\Sigma p\, t}\ (pappl)$

where $p{:}\Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau$

and $\tau_1 = \sigma_1(\tau)$ and $\tau_2 = \sigma_2(\tau)$ for some sort variable substitutions σ_1 and σ_2.

$$\begin{array}{ll} \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} & \overset{\text{Def. 4.2.4}}{=} \\ p^{\mathcal{A}}[\mathcal{SA}[\![\sigma_1(\alpha_1)]\!]_\nu, \ldots, \mathcal{SA}[\![\sigma_1(\alpha_n)]\!]_\nu](\mathcal{A}\left[\!\left[\Delta_1, \Gamma_1 \bigtriangledown_\Sigma t :: \tau_1\right]\!\right]_{\nu,\eta}) & \overset{*)}{=} \\ p^{\mathcal{A}}[\mathcal{SA}[\![\sigma_2(\alpha_1)]\!]_{\nu^\sigma}, \ldots, \mathcal{SA}[\![\sigma_2(\alpha_n)]\!]_{\nu^\sigma}](\mathcal{A}\left[\!\left[\Delta_2, \Gamma_2 \bigtriangledown_\Sigma t :: \tau_2\right]\!\right]_{\nu^\sigma,\eta}) & \overset{\text{Def. 4.2.4}}{=} \\ \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta} & \end{array}$$

ad $*$) We must show:

i) $\mathcal{SA}[\![\sigma_1(\alpha_i)]\!]_\nu = \mathcal{SA}[\![\sigma_2(\alpha_i)]\!]_{\nu^\sigma}$, $1 \leq i \leq n$

ii) $\mathcal{A}\left[\!\left[\Delta_1, \Gamma_1 \bigtriangledown_\Sigma t :: \tau_1\right]\!\right]_{\nu,\eta} = \mathcal{A}\left[\!\left[\Delta_2, \Gamma_2 \bigtriangledown_\Sigma t :: \tau_2\right]\!\right]_{\nu^\sigma,\eta}$

ad i) $\begin{array}{lll} D_1 \leq^\sigma D_2 & \overset{\text{Def. 4.3.2}}{\Rightarrow} & \tau_1 = \sigma(\tau_2) \\ & \Leftrightarrow & \sigma_1(\tau) = \sigma(\sigma_2(\tau)) \end{array}$

By Def. 3.1.10 all α_i, $1 \leq i \leq n$, occur in τ. Thus,

$$\begin{array}{lll} \sigma_1(\alpha_i) = \sigma(\sigma_2(\alpha_i)) & \Rightarrow & \mathcal{SA}[\![\sigma_1(\alpha_i)]\!]_\nu \;=\; \mathcal{SA}[\![\sigma(\sigma_2(\alpha_i))]\!]_\nu \\ & \overset{\text{Prop. 4.1.1}}{\Leftrightarrow} & \mathcal{SA}[\![\sigma_2(\alpha_i)]\!]_{\nu^\sigma} \end{array}$$

ad ii) By Prop. 4.3.7 because

$$\begin{array}{rcll} D_1 & \leq^\sigma & D_2 & \overset{\text{Def. 4.3.2}}{\Rightarrow} \\ \Delta_1, \Gamma_1 \bigtriangledown_\Sigma t :: \tau_1 & \leq^\sigma & \Delta_2, \Gamma_2 \bigtriangledown_\Sigma t :: \tau_2 & \end{array}$$

Inductive cases:

- $D_1 = \dfrac{\Delta_1, \Gamma_1.x{:}\tau_1 \bigtriangledown_\Sigma f}{\Delta_1, \Gamma_1 \rhd_\Sigma \forall x.\, f}\ (univ_u) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{\Delta_2, \Gamma_2.x{:}\tau_2 \bigtriangledown_\Sigma f}{\Delta_2, \Gamma_2 \rhd_\Sigma \forall x.\, f}\ (univ_u)$

  $$\mathcal{A}\left[\!\left[D_1 \right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.4}}{=} \begin{cases} true & \text{if for all } d \in \mathcal{SA}[\![\tau_1]\!]_\nu\text{:} \\ & \mathcal{A}\left[\!\left[\Delta_1,\, \Gamma_1.x{:}\tau_1 \bigtriangledown_\Sigma f \right]\!\right]_{\nu,\eta.[x\mapsto d]} = true \\ false & \text{otherwise} \end{cases}$$

  $$\mathcal{A}\left[\!\left[D_2 \right]\!\right]_{\nu^\sigma,\eta} \overset{\text{Def. 4.2.4}}{=} \begin{cases} true & \text{if for all } d \in \mathcal{SA}[\![\tau_2]\!]_{\nu^\sigma}\text{:} \\ & \mathcal{A}\left[\!\left[\Delta_2,\, \Gamma_2.x{:}\tau_2 \bigtriangledown_\Sigma f \right]\!\right]_{\nu^\sigma,\eta.[x\mapsto d]} = true \\ false & \text{otherwise} \end{cases}$$

  It remains to show:

  i) $\mathcal{SA}[\![\tau_1]\!]_\nu = \mathcal{SA}[\![\tau_2]\!]_{\nu^\sigma}$

  ii) For all $d \in \mathcal{SA}[\![\tau_1]\!]_\nu$:
  $\mathcal{A}\left[\!\left[\Delta_1,\, \Gamma_1.x{:}\tau_1 \bigtriangledown_\Sigma f \right]\!\right]_{\nu,\eta.[x\mapsto d]} = \mathcal{A}\left[\!\left[\Delta_2,\, \Gamma_2.x{:}\tau_2 \bigtriangledown_\Sigma f \right]\!\right]_{\nu^\sigma,\eta.[x\mapsto d]}$

  ad i) $\mathcal{SA}[\![\tau_1]\!]_\nu \overset{*)}{=} \mathcal{SA}[\![\sigma(\tau_2)]\!]_\nu \overset{\text{Prop. 4.1.1}}{=} \mathcal{SA}[\![\tau_2]\!]_{\nu^\sigma}$
  ad $*$) $D_1 \leq^\sigma D_2 \overset{\text{Def. 4.3.2}}{\Rightarrow} \tau_1 = \sigma(\tau_2)$

  ad ii) $D_1 \leq^\sigma D_2 \overset{\text{Def. 4.3.2}}{\Rightarrow} \Delta_1, \Gamma_1.x{:}\tau_1 \bigtriangledown_\Sigma f \leq^\sigma \Delta_2, \Gamma_2.x{:}\tau_2 \bigtriangledown_\Sigma f$
  To apply the induction hypothesis we must further show that the new variable assignments satisfy the new variable assumptions, i.e.

  $$\eta.[x \mapsto d] \models_{\mathcal{SA},\nu} \Gamma_1.x{:}\tau_1 \text{ and } \eta.[x \mapsto d] \models_{\mathcal{SA},\nu^\sigma} \Gamma_2.x{:}\tau_2$$

  This is valid by Prop. 4.2.1 because $d \in \mathcal{SA}[\![\tau_1]\!]_\nu \overset{\text{i)}}{=} \mathcal{SA}[\![\tau_2]\!]_{\nu^\sigma}$. Therefore, by induction hypothesis ii) holds.

- $D_1 = \dfrac{\Delta_1, \Gamma_1.x{:}\tau \bigtriangledown_\Sigma f}{\Delta_1, \Gamma_1 \rhd_\Sigma \forall x{:}\tau.\, f}\ (univ_s) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{\Delta_2, \Gamma_2.x{:}\tau \bigtriangledown_\Sigma f}{\Delta_2, \Gamma_2 \rhd_\Sigma \forall x{:}\tau.\, f}\ (univ_s)$

  $$\mathcal{A}\left[\!\left[D_1 \right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.4}}{=} \begin{cases} true & \text{if for all } d \in \mathcal{SA}[\![\tau]\!]_\nu\text{:} \\ & \mathcal{A}\left[\!\left[\Delta_1,\, \Gamma_1.x{:}\tau \bigtriangledown_\Sigma f \right]\!\right]_{\nu,\eta.[x\mapsto d]} = true \\ false & \text{otherwise} \end{cases}$$

  $$\mathcal{A}\left[\!\left[D_2 \right]\!\right]_{\nu^\sigma,\eta} \overset{\text{Def. 4.2.4}}{=} \begin{cases} true & \text{if for all } d \in \mathcal{SA}[\![\tau]\!]_{\nu^\sigma}\text{:} \\ & \mathcal{A}\left[\!\left[\Delta_2,\, \Gamma_2.x{:}\tau \bigtriangledown_\Sigma f \right]\!\right]_{\nu^\sigma,\eta.[x\mapsto d]} = true \\ false & \text{otherwise} \end{cases}$$

  It remains to show:

  i) $\mathcal{SA}[\![\tau]\!]_\nu = \mathcal{SA}[\![\tau]\!]_{\nu^\sigma}$

ii) For all $d \in \mathcal{SA}[\![\tau]\!]_\nu$:
$\mathcal{A}\left[\!\left[\Delta_1, \Gamma_1.x{:}\tau \bigtriangledown_\Sigma f\right]\!\right]_{\nu,\eta.[x\mapsto d]} = \mathcal{A}\left[\!\left[\Delta_2, \Gamma_2.x{:}\tau \bigtriangledown_\Sigma f\right]\!\right]_{\nu^\sigma,\eta.[x\mapsto d]}$

ad i) $\mathcal{SA}[\![\tau]\!]_\nu \overset{(*)}{=} \mathcal{SA}[\![\sigma(\tau)]\!]_\nu \overset{\text{Prop. 4.1.1}}{=} \mathcal{SA}[\![\tau]\!]_{\nu^\sigma}$

ad $*$) because $FSV(\tau) \subseteq FSV(\forall x{:}\tau.\, f)$ and $\sigma(\alpha){=}\alpha\ \forall \alpha \in FSV(\forall x{:}\tau.\, f)$

ad ii) as $univ_u$ ii)

- $D_1 = \dfrac{\Delta_1, \Gamma_1 \bigtriangledown_\Sigma f}{\Delta_1, \Gamma_1 \rhd_\Sigma \neg f}\ (not) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{\Delta_2, \Gamma_2 \bigtriangledown_\Sigma f}{\Delta_2, \Gamma_2 \rhd_\Sigma \neg f}\ (not)$

  $\mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.4}}{=} \begin{cases} true & \text{if } \mathcal{A}\left[\!\left[\Delta_1, \Gamma_1 \bigtriangledown_\Sigma f\right]\!\right]_{\nu,\eta} = false \\ false & \text{otherwise} \end{cases}$

  $\mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta} \overset{\text{Def. 4.2.4}}{=} \begin{cases} true & \text{if } \mathcal{A}\left[\!\left[\Delta_2, \Gamma_2 \bigtriangledown_\Sigma f\right]\!\right]_{\nu^\sigma,\eta} = false \\ false & \text{otherwise} \end{cases}$

  $D_1 \leq^\sigma D_2 \overset{\text{Def. 4.3.2}}{\Rightarrow} \Delta_1, \Gamma_1 \bigtriangledown_\Sigma f \leq^\sigma \Delta_2, \Gamma_2 \bigtriangledown_\Sigma f$. Therefore, by induction hypothesis $\mathcal{A}\left[\!\left[\Delta_1, \Gamma_1 \bigtriangledown_\Sigma f\right]\!\right]_{\nu,\eta} = \mathcal{A}\left[\!\left[\Delta_2, \Gamma_2 \bigtriangledown_\Sigma f\right]\!\right]_{\nu^\sigma,\eta}$.

- Case (or) is similar to (not).

□

Proposition 4.3.7 *Interpretation equivalent sort derivations for terms*

Let $D_1 = \Delta_1, \Gamma_1 \bigtriangledown_\Sigma e :: \tau_1$ and $D_2 = \Delta_2, \Gamma_2 \bigtriangledown_\Sigma e :: \tau_2$ be two sort derivations for a pre-term $e \in \mathrm{PT}_\Sigma(\Phi)$. Let σ be a sort variable substitution such that $\forall \alpha \in FSV(e).\ \sigma(\alpha) = \alpha$ and $D_1 \leq^\sigma D_2$. Let ν and ν^σ be sort variable assignments such that $\models_{\mathcal{SA},\nu} \Delta_1$ and $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$. Let η be a variable assignment with $\eta \models_{\mathcal{SA},\nu} \Gamma_1$ and $\eta \models_{\mathcal{SA},\nu^\sigma} \Gamma_2$.

$$\text{If} \quad \nu^\sigma(\alpha){=}\mathcal{SA}[\![\sigma(\alpha)]\!]_\nu \text{ for each } \alpha \in \Phi \quad \text{then} \quad \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} = \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta}$$

Proof: By induction on the structure of D_1.

Base cases:

- $D_1 = \dfrac{}{\Delta_1, \Gamma_1 \rhd_\Sigma x :: \tau_1}\ (var) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{}{\Delta_2, \Gamma_2 \rhd_\Sigma x :: \tau_2}\ (var)$

  $\mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.3}}{=} \eta(x) \overset{\text{Def. 4.2.3}}{=} \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta}$

- $D_1 = \dfrac{}{\Delta_1, \Gamma_1 \rhd_\Sigma c :: \tau}\ (id) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{}{\Delta_2, \Gamma_2 \rhd_\Sigma c :: \tau}\ (id)$

  $\mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.3}}{=} c^{\mathcal{A}} \overset{\text{Def. 4.2.3}}{=} \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta}$

- $D_1 = \dfrac{}{\Delta_1, \Gamma_1 \rhd_\Sigma c :: \tau_1}\ (polyid) \overset{\text{Prop. 4.3.1}}{\Rightarrow} D_2 = \dfrac{}{\Delta_2, \Gamma_2 \rhd_\Sigma c :: \tau_2}\ (polyid)$

  where $c{:}\Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau$

  and $\tau_1 = \sigma_1(\tau)$ and $\tau_2 = \sigma_2(\tau)$ for some sort variable substitution σ_1 and σ_2.

  $$\begin{array}{lll} \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} & \overset{\text{Def. 4.2.3}}{=} & c^{\mathcal{A}}[\mathcal{SA}[\![\sigma_1(\alpha_1)]\!]_\nu, \ \ldots\ , \mathcal{SA}[\![\sigma_1(\alpha_n)]\!]_\nu] \\ & \overset{*)}{=} & c^{\mathcal{A}}[\mathcal{SA}[\![\sigma_2(\alpha_1)]\!]_{\nu^\sigma}, \ \ldots\ , \mathcal{SA}[\![\sigma_2(\alpha_n)]\!]_{\nu^\sigma}] \\ & \overset{\text{Def. 4.2.3}}{=} & \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta} \end{array}$$

  ad $*$) $D_1 \leq^\sigma D_2 \overset{\text{Def. 4.3.2}}{\Rightarrow} \tau_1 = \sigma(\tau_2) \Leftrightarrow \sigma_1(\tau) = \sigma(\sigma_2(\tau))$

  By Def. 3.1.10 all α_i, $1 \leq i \leq n$, occur in τ. Thus,

  $$\sigma_1(\alpha_i) = \sigma(\sigma_2(\alpha_i)) \Rightarrow \mathcal{SA}[\![\sigma_1(\alpha_i)]\!]_\nu = \mathcal{SA}[\![\sigma(\sigma_2(\alpha_i))]\!]_\nu \overset{\text{Prop. 4.1.1}}{\Leftrightarrow} \mathcal{SA}[\![\sigma_2(\alpha_i)]\!]_{\nu^\sigma}$$

Inductive cases:

- $D_1 = \dfrac{\Delta_1, \Gamma_1 \ \triangledown_\Sigma\ e_1 :: \tau_{12} \to \tau_{11} \quad \Delta_1, \Gamma_1 \ \triangledown_\Sigma\ e_2 :: \tau_{12}}{\Delta_1, \Gamma_1 \rhd_\Sigma e_1 e_2 :: \tau_{11}}\ (\to E) \overset{\text{Prop. 4.3.1}}{\Rightarrow}$

  $D_2 = \dfrac{\Delta_2, \Gamma_2 \ \triangledown_\Sigma\ e_1 :: \tau_{22} \to \tau_{21} \quad \Delta_2, \Gamma_2 \ \triangledown_\Sigma\ e_2 :: \tau_{22}}{\Delta_2, \Gamma_2 \rhd_\Sigma e_1 e_2 :: \tau_{21}}\ (\to E)$

  $$\begin{array}{ll} \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} & \overset{\text{Def. 4.2.3}}{=} \\ \mathcal{A}\left[\!\left[\Delta_1, \Gamma_1 \ \triangledown_\Sigma\ e_1 :: \tau_{12} \to \tau_{11}\right]\!\right]_{\nu,\eta}\ (\mathcal{A}\left[\!\left[\Delta_1, \Gamma_1 \ \triangledown_\Sigma\ e_2 :: \tau_{12}\right]\!\right]_{\nu,\eta}) & \overset{*)}{=} \\ \mathcal{A}\left[\!\left[\Delta_2, \Gamma_2 \ \triangledown_\Sigma\ e_1 :: \tau_{22} \to \tau_{21}\right]\!\right]_{\nu^\sigma,\eta}\ (\mathcal{A}\left[\!\left[\Delta_2, \Gamma_2 \ \triangledown_\Sigma\ e_2 :: \tau_{22}\right]\!\right]_{\nu^\sigma,\eta}) & \overset{\text{Def. 4.2.3}}{=} \\ \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta} & \end{array}$$

  ad $*$) by induction hypothesis because

  $$D_1 \leq^\sigma D_2 \overset{\text{Def. 4.3.2}}{\Rightarrow} \Delta_1, \Gamma_1 \ \triangledown_\Sigma\ e_1 :: \tau_{12} \to \tau_{11} \leq^\sigma \Delta_2, \Gamma_2 \ \triangledown_\Sigma\ e_1 :: \tau_{22} \to \tau_{21}$$
  $$D_1 \leq^\sigma D_2 \overset{\text{Def. 4.3.2}}{\Rightarrow} \Delta_1, \Gamma_1 \ \triangledown_\Sigma\ e_2 :: \tau_{12} \leq^\sigma \Delta_2, \Gamma_2 \ \triangledown_\Sigma\ e_2 :: \tau_{22}$$

- $D_1 = \dfrac{\Delta_1, \Gamma_1.x{:}\tau_{11} \ \triangledown_\Sigma\ e :: \tau_{12}}{\Delta_1, \Gamma_1 \rhd_\Sigma \lambda x.\, e\ :: \tau_{11} \to \tau_{12}}\ (\to I_u) \overset{\text{Prop. 4.3.1}}{\Rightarrow}$

  $D_2 = \dfrac{\Delta_2, \Gamma_2.x{:}\tau_{21} \ \triangledown_\Sigma\ e :: \tau_{22}}{\Delta_2, \Gamma_2 \rhd_\Sigma \lambda x.\, e\ :: \tau_{21} \to \tau_{22}}\ (\to I_u)$

  $$\mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.3}}{=} \text{the unique } f \in \mathcal{SA}[\![\tau_{11} \to \tau_{12}]\!]_\nu \text{ such that } \forall\ d \in \mathcal{SA}[\![\tau_{11}]\!]_\nu.\ f(d) = \mathcal{A}\left[\!\left[\Delta_1, \Gamma_1.x{:}\tau_{11} \ \triangledown_\Sigma\ e :: \tau_{12}\right]\!\right]_{\nu,\eta.[x \mapsto d]}$$

  $$\mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta} \overset{\text{Def. 4.2.3}}{=} \text{the unique } f \in \mathcal{SA}[\![\tau_{21} \to \tau_{22}]\!]_{\nu^\sigma} \text{ such that } \forall\ d \in \mathcal{SA}[\![\tau_{21}]\!]_{\nu^\sigma}.\ f(d) = \mathcal{A}\left[\!\left[\Delta_2, \Gamma_2.x{:}\tau_{21} \ \triangledown_\Sigma\ e :: \tau_{22}\right]\!\right]_{\nu^\sigma,\eta.[x \mapsto d]}$$

It remains to show:

i) $\mathcal{SA}[\![\tau_{11} \to \tau_{12}]\!]_\nu = \mathcal{SA}[\![\tau_{21} \to \tau_{22}]\!]_{\nu^\sigma}$

ii) $\mathcal{SA}[\![\tau_{11}]\!]_\nu = \mathcal{SA}[\![\tau_{21}]\!]_{\nu^\sigma}$

iii) $\forall\, d \in \mathcal{SA}[\![\tau_{11}]\!]_\nu.\quad \mathcal{A}\left[\!\left[\Delta_1,\, \Gamma_1.x{:}\tau_{11} \bigtriangledown_\Sigma e :: \tau_{12}\right]\!\right]_{\nu,\eta.[x\mapsto d]} = \mathcal{A}\left[\!\left[\Delta_2,\, \Gamma_2.x{:}\tau_{21} \bigtriangledown_\Sigma e :: \tau_{22}\right]\!\right]_{\nu^\sigma,\eta.[x\mapsto d]}$

ad i) $\mathcal{SA}[\![\tau_{11} \to \tau_{12}]\!]_\nu \overset{*)}{=} \underset{\text{Prop. 4.1.1}}{=} \mathcal{SA}[\![\sigma(\tau_{21} \to \tau_{22})]\!]_\nu = \mathcal{SA}[\![\tau_{21} \to \tau_{22}]\!]_{\nu^\sigma}$

ad $*$) $D_1 \leq^\sigma D_2 \overset{\text{Prop. 4.3.3}}{\Rightarrow} \tau_{11} \to \tau_{12} = \sigma(\tau_{21} \to \tau_{22})$

ad ii) follows directly from i)

ad iii) $D_1 \leq^\sigma D_2 \overset{\text{Def. 4.3.2}}{\Rightarrow} \Delta_1,\, \Gamma_1.x{:}\tau_{11} \bigtriangledown_\Sigma e :: \tau_{12} \leq^\sigma \Delta_2,\, \Gamma_2.x{:}\tau_{21} \bigtriangledown_\Sigma e :: \tau_{22}$

To apply the induction hypothesis we must further show that the new variable assignments satisfy the new variable assumptions, i.e.

$$\eta.[x \mapsto d] \models_{\mathcal{SA},\nu} \Gamma_1.x{:}\tau_{11} \text{ and } \eta.[x \mapsto d] \models_{\mathcal{SA},\nu^\sigma} \Gamma_2.x{:}\tau_{21}$$

This is valid by Prop. 4.2.1 because $d \in \mathcal{SA}[\![\tau_{11}]\!]_\nu \overset{\text{ii)}}{=} \mathcal{SA}[\![\tau_{21}]\!]_{\nu^\sigma}$. Therefore, by induction hypothesis iii) holds.

- $D_1 = \dfrac{\Delta_1, \Gamma_1.x{:}\tau_1 \bigtriangledown_\Sigma e :: \tau_{12}}{\Delta_1, \Gamma_1 \rhd_\Sigma \lambda x{:}\tau_1.\, e :: \tau_1 \to \tau_{12}}\ (\to I_s) \overset{\text{Prop. 4.3.1}}{\Rightarrow}$

  $D_2 = \dfrac{\Delta_2, \Gamma_2.x{:}\tau_1 \bigtriangledown_\Sigma e :: \tau_{22}}{\Delta_2, \Gamma_2 \rhd_\Sigma \lambda x{:}\tau_1.\, e :: \tau_1 \to \tau_{22}}\ (\to I_s)$

  $\mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.2.3}}{=}$ the unique $f \in \mathcal{SA}[\![\tau_1 \to \tau_{12}]\!]_\nu$ such that $\forall\, d \in \mathcal{SA}[\![\tau_1]\!]_\nu.\ f(d) = \mathcal{A}\left[\!\left[\Delta_1,\, \Gamma_1.x{:}\tau_1 \bigtriangledown_\Sigma e :: \tau_{12}\right]\!\right]_{\nu,\eta.[x\mapsto d]}$

  $\mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta} \overset{\text{Def. 4.2.3}}{=}$ the unique $f \in \mathcal{SA}[\![\tau_1 \to \tau_{22}]\!]_{\nu^\sigma}$ such that $\forall\, d \in \mathcal{SA}[\![\tau_1]\!]_{\nu^\sigma}.\ f(d) = \mathcal{A}\left[\!\left[\Delta_2,\, \Gamma_2.x{:}\tau_1 \bigtriangledown_\Sigma e :: \tau_{22}\right]\!\right]_{\nu^\sigma,\eta.[x\mapsto d]}$

  It remains to show:

  i) $\mathcal{SA}[\![\tau_1 \to \tau_{12}]\!]_\nu = \mathcal{SA}[\![\tau_1 \to \tau_{22}]\!]_{\nu^\sigma}$

  ii) $\mathcal{SA}[\![\tau_1]\!]_\nu = \mathcal{SA}[\![\tau_1]\!]_{\nu^\sigma}$

  iii) $\forall\, d \in \mathcal{SA}[\![\tau_1]\!]_\nu.\quad \mathcal{A}\left[\!\left[\Delta_1,\, \Gamma_1.x{:}\tau_1 \bigtriangledown_\Sigma e :: \tau_{12}\right]\!\right]_{\nu,\eta.[x\mapsto d]} = \mathcal{A}\left[\!\left[\Delta_2,\, \Gamma_2.x{:}\tau_1 \bigtriangledown_\Sigma e :: \tau_{22}\right]\!\right]_{\nu^\sigma,\eta.[x\mapsto d]}$

ad i) $\mathcal{SA}[\![\tau_1 \to \tau_{12}]\!]_\nu \overset{*)}{=} \mathcal{SA}[\![\sigma(\tau_1 \to \tau_{22})]\!]_\nu$
$\overset{\text{Prop. 4.1.1}}{=} \mathcal{SA}[\![\tau_1 \to \tau_{22}]\!]_{\nu^\sigma}$

ad $*$) $D_1 \leq^\sigma D_2 \overset{\text{Prop. 4.3.3}}{\Rightarrow} \tau_1 \to \tau_{12} = \sigma(\tau_1 \to \tau_{22})$

ad ii) follows directly from i)

ad iii) $D_1 \leq^\sigma D_2 \overset{\text{Def. 4.3.2}}{\Rightarrow} \Delta_1, \Gamma_1.x{:}\tau_1 \bigtriangledown_\Sigma e :: \tau_{12} \leq^\sigma \Delta_2, \Gamma_2.x{:}\tau_1 \bigtriangledown_\Sigma e :: \tau_{22}$

To apply the induction hypothesis we must further show that the new variable assignments satisfy the new variable assumptions, i.e.

$$\eta.[x \mapsto d] \models_{\mathcal{SA},\nu} \Gamma_1.x{:}\tau_1 \text{ and } \eta.[x \mapsto d] \models_{\mathcal{SA},\nu^\sigma} \Gamma_2.x{:}\tau_1$$

This is valid by Prop. 4.2.1 because $d \in \mathcal{SA}[\![\tau_1]\!]_\nu \overset{\text{ii)}}{=} \mathcal{SA}[\![\tau_1]\!]_{\nu^\sigma}$. Therefore, by induction hypothesis iii) holds.

- Case (*constrained*) is similar to $(\to E)$.

□

Proposition 4.3.9 *Equivalent sort derivations are satisfaction equivalent*

Let $\mathcal{A}$ be a polymorphic Σ algebra. Let D_1 and D_2 be two sort derivations for a closed formula. If D_1 and D_2 are equivalent sort derivations then D_1 and D_2 are satisfaction equivalent, i.e.

$$\text{if} \quad D_1 \leq D_2 \text{ and } D_2 \leq D_1 \quad \text{then} \quad \mathcal{A} \models D_1 \Leftrightarrow \mathcal{A} \models D_2$$

Proof:

"$\Rightarrow$" $D_2 \leq D_1 \overset{\text{Prop. 4.3.5}}{\Rightarrow} D_1 \models D_2 \overset{\text{Def. 4.3.1}}{\Leftrightarrow} \mathcal{A} \models D_1 \Rightarrow \mathcal{A} \models D_2$

"$\Leftarrow$" $D_1 \leq D_2 \overset{\text{Prop. 4.3.5}}{\Rightarrow} D_2 \models D_1 \overset{\text{Def. 4.3.1}}{\Leftrightarrow} \mathcal{A} \models D_2 \Rightarrow \mathcal{A} \models D_1$

□

Proposition 4.3.10 *Satisfaction is equivalent to p-satisfaction*

Let $\mathcal{A}$ be a polymorphic Σ algebra. Let $f \in \mathrm{PF}_\Sigma(\Phi)$ be a closed well-formed Σ formula. Formula f is satisfied in $\mathcal{A}$ iff f is p-satisfied in $\mathcal{A}$, i.e.

$$\mathcal{A} \models_p f \quad \Leftrightarrow \quad \mathcal{A} \models f$$

Proof:

$$\begin{array}{lll} \mathcal{A} \models_p f & \overset{\text{Def. 4.3.5}}{\Leftrightarrow} & \mathcal{A} \models D \text{ for some principal sort derivation } D \text{ for } f \\ & \overset{*)}{\Leftrightarrow} & \mathcal{A} \models D \quad \text{for every sort derivation } D \text{ that ends with} \\ & & \Delta, \emptyset \rhd_\Sigma f \text{ for some sort predicate assumption } \Delta \\ & \overset{\text{Def. 4.2.7}}{\Leftrightarrow} & \mathcal{A} \models f \end{array}$$

ad $*$)

"$\Rightarrow$" Because D is a principal sort derivation for f, by Def. 4.3.4 for every other sort derivation $D' = \Delta', \emptyset \bigtriangledown_\Sigma f$ holds that $D' \leq D$. Therefore, by Prop. 4.3.5 $D \models D' \overset{\text{Def. 4.3.1}}{\Leftrightarrow} \mathcal{A} \models D \Rightarrow \mathcal{A} \models D'$

"$\Leftarrow$" This direction trivially holds, because the principal sort derivation also ends with $\Delta, \emptyset \rhd_\Sigma f$ for some sort predicate assumption Δ.

□

Proposition 4.3.11 *Equivalence of well-formedness*

Let $qf \in \mathrm{QF}_\Sigma$ be a qualified Σ-formula. There exists an extended sort derivation for qf iff there exists a sort derivation for qf.

Proof:

"$\Rightarrow$" Let $\dfrac{\Delta, \Gamma \bigtriangledown_\Sigma f}{\Delta, \Gamma \rhd_\Sigma qf}$ (*ext. qual.*) be an extended sort derivation for $qf = \forall\overline{\alpha_n}.\overline{A_m} \Rightarrow f$. By the side condition of rule (*ext. qual.*) there exists a sort variable substitution σ such that $\overline{A_m} \vdash_{SA} \sigma(\Delta)$. We apply this substitution to the derivation tree $\Delta, \Gamma \bigtriangledown_\Sigma f$. By Prop. 4.3.13 $\sigma_\emptyset(\Delta, \Gamma \bigtriangledown_\Sigma f) \overset{\text{Def. 4.3.8}}{=} \sigma(\Delta), \sigma(\Gamma) \bigtriangledown_\Sigma f$ is a sort derivation for f. Thus, $\dfrac{\sigma(\Delta), \sigma(\Gamma) \bigtriangledown_\Sigma f}{\sigma(\Delta), \sigma(\Gamma) \rhd_\Sigma qf}$ (*qualification*) is a sort derivation for qf because $\overline{A_m} \vdash_{SA} \sigma(\Delta)$.

"$\Leftarrow$" Let $\dfrac{\Delta, \Gamma \bigtriangledown_\Sigma f}{\Delta, \Gamma \rhd_\Sigma qf}$ (*qualification*) be a sort derivation for $qf = \forall\overline{\alpha_n}.\overline{A_m} \Rightarrow f$. By the side condition of rule (*qualification*) we know that $\overline{A_m} \vdash_{SA} \Delta$. Thus, $\dfrac{\Delta, \Gamma \bigtriangledown_\Sigma f}{\Delta, \Gamma \rhd_\Sigma qf}$ (*ext. qual.*) is trivially an extended sort derivation for qf because $\overline{A_m} \vdash_{SA} \varepsilon(\Delta)$.

□

Proposition 4.3.12 *Substitution preserves sort derivation*

Let $D = \Delta, \Gamma \bigtriangledown_\Sigma t :: \tau$ be a sort derivation for a pre-term $t \in \mathrm{PT}_\Sigma(\Phi)$. Let σ be a sort variable substitution, and let $\Delta' \in \mathcal{P}(\mathrm{AF}_\Omega(\Phi)$ be a set of atomic sort formulae.

If $\forall \alpha \in FSV(t).\ \sigma(\alpha)=\alpha$ then $\sigma_{\Delta'}(D)$ is a sort derivation for t.

Proof:

- $\sigma_{\Delta'}(\dfrac{}{\Delta, \Gamma.x{:}\tau \rhd_\Sigma x :: \tau}\ (var)) \overset{\text{Def. 4.3.7}}{=}$
  $\dfrac{}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma.x{:}\tau) \rhd_\Sigma x :: \sigma(\tau)}\ (var) \overset{\text{Def. 3.2.6}}{=}$
  $\dfrac{}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma).x{:}\sigma(\tau) \rhd_\Sigma x :: \sigma(\tau)}\ (var)$

  This is a valid application of rule (var), and therefore the substitution yields a sort derivation for x.

- $\sigma_{\Delta'}(\dfrac{}{\Delta, \Gamma \rhd_\Sigma c :: \tau}\ (id)) \overset{\text{Def. 4.3.7}}{=} \dfrac{}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma c :: \sigma(\tau)}\ (id)$
  $\overset{\tau \in T_\Omega(\emptyset)}{=} \dfrac{}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma c :: \tau}\ (id)$

  This is a valid application of rule (id) because $c : \tau$ is a function identifier from the signature which is not affected by the substitution. Therefore, the substitution yields a sort derivation for c.

- $\sigma_{\Delta'}(\dfrac{}{\Delta, \Gamma \rhd_\Sigma c :: \tau'}\ (polyid)) \overset{\text{Def. 4.3.7}}{=} \dfrac{}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma c :: \sigma(\tau')}\ (polyid)$

  We have to prove the side conditions of the rule $(polyid)$. Let $c{:}\Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau \in F$. Firstly, we have to prove that $\sigma(\tau')$ is an instance of τ, i.e. there exists a substitution σ_2 such that $\sigma(\tau') = \sigma_2(\tau)$. This is valid because there exists a substitution σ_1 with $\sigma_1(\tau) = \tau'$. This substitution is uniquely determined for all Π-bound α_i, $1 \leq i \leq n$, because all α_i must occur in τ. Thus, $\sigma_2 := \sigma \circ \sigma_1$ is also uniquely determined for all Π-bound α_i and $\sigma(\tau') = \sigma(\sigma_1(\tau)) = \sigma_2(\tau)$.

  Secondly, we must prove that $\sigma_2(A_i) \in \sigma(\Delta) \cup \Delta' \Leftarrow \sigma(\sigma_1(A_i)) \in \sigma(\Delta)$, $1 \leq i \leq m$. This follows immediately by the side condition $\sigma_1(A_i) \in \Delta$ of the given rule application.

Inductive cases:

- $\sigma_{\Delta'}(\dfrac{\Delta, \Gamma \bigtriangledown_\Sigma e_1 :: \tau_2 \to \tau_1 \quad \Delta, \Gamma \bigtriangledown_\Sigma e_2 :: \tau_2}{\Delta, \Gamma \rhd_\Sigma e_1 e_2 :: \tau_1}\ (\to E)) \overset{\text{Def. 4.3.7}}{=}$
  $\dfrac{\sigma_{\Delta'}(\Delta, \Gamma \bigtriangledown_\Sigma e_1 :: \tau_2 \to \tau_1) \quad \sigma_{\Delta'}(\Delta, \Gamma \bigtriangledown_\Sigma e_2 :: \tau_2)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma e_1 e_2 :: \sigma(\tau_1)}\ (\to E) \overset{\text{Def. 4.3.7, 3.1.4}}{=}$
  $\dfrac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \bigtriangledown_\Sigma e_1 :: \sigma(\tau_2) \to \sigma(\tau_1) \quad \sigma(\Delta) \cup \Delta', \sigma(\Gamma) \bigtriangledown_\Sigma e_2 :: \sigma(\tau_2)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma e_1 e_2 :: \sigma(\tau_1)}\ (\to E)$

  This is a valid application of rule $(\to E)$ and yields a sort derivation for $e_1 e_2$ because by induction both $\sigma_{\Delta'}(\Delta, \Gamma \bigtriangledown_\Sigma e_1 :: \tau_2 \to \tau_1)$ and $\sigma_{\Delta'}(\Delta, \Gamma \bigtriangledown_\Sigma e_2 :: \tau_2)$ are sort derivations for e_1 and e_2, respectively.

- $\sigma_{\Delta'}(\dfrac{\Delta, \Gamma.x{:}\tau_1 \ \nabla_\Sigma \ e :: \tau_2}{\Delta, \Gamma \ \rhd_\Sigma \ \lambda x.e :: \tau_1 \to \tau_2} \ (\to I_u))$ $\overset{\text{Def. 4.3.7}}{=}$
$\dfrac{\sigma_{\Delta'}(\Delta, \Gamma.x{:}\tau_1 \ \nabla_\Sigma \ e :: \tau_2)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ \lambda x.e :: \sigma(\tau_1 \to \tau_2)} \ (\to I_u)$ $\overset{\text{Def. 4.3.7, 3.1.4, 3.2.6}}{=}$
$\dfrac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma).x{:}\sigma(\tau_1) \ \nabla_\Sigma \ e :: \sigma(\tau_2)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ \lambda x.e :: \sigma(\tau_1) \to \sigma(\tau_2)} \ (\to I_u)$

  This is a valid application of rule $(\to I_u)$ and yields a sort derivation for $\lambda x.e$ because, by induction, $\sigma_{\Delta'}(\Delta, \Gamma.x{:}\tau_1 \ \nabla_\Sigma \ e :: \tau_2)$ is a sort derivation for e.

- $\sigma_{\Delta'}(\dfrac{\Delta, \Gamma.x{:}\tau_1 \ \nabla_\Sigma \ e :: \tau_2}{\Delta, \Gamma \ \rhd_\Sigma \ \lambda x{:}\tau_1.e :: \tau_1 \to \tau_2} \ (\to I_s))$ $\overset{\text{Def. 4.3.7}}{=}$
$\dfrac{\sigma_{\Delta'}(\Delta, \Gamma.x{:}\tau_1 \ \nabla_\Sigma \ e :: \tau_2)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ \lambda x{:}\tau_1.e :: \sigma(\tau_1 \to \tau_2)} \ (\to I_s)$ $\overset{\text{Def. 4.3.7, 3.1.4, 3.2.6}}{=}$
$\dfrac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma).x{:}\sigma(\tau_1) \ \nabla_\Sigma \ e :: \sigma(\tau_2)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ \lambda x{:}\tau_1.e :: \sigma(\tau_1) \to \sigma(\tau_2)} \ (\to I_s)$ $\overset{FSV(\tau_1) \subseteq FSV(x:\tau_1.e)}{=}$
$\dfrac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma).x{:}\tau_1 \ \nabla_\Sigma \ e :: \sigma(\tau_2)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ \lambda x : \tau_1.e :: \tau_1 \to \sigma(\tau_2)} \ (\to I_s)$

  This is a valid application of rule $(\to I_s)$ and yields a sort derivation for $\lambda x{:}\tau_1.e$ because by induction $\sigma_{\Delta'}(\Delta, \Gamma.x{:}\tau_1 \ \nabla_\Sigma \ e :: \tau_2)$ is a sort derivation for e.

- $\sigma_{\Delta'}(\dfrac{\Delta, \Gamma \ \nabla_\Sigma \ e :: \tau}{\Delta, \Gamma \ \rhd_\Sigma \ e{:}\tau :: \tau} \ (\mathit{constrained}))$ $\overset{\text{Def. 4.3.7}}{=}$
$\dfrac{\sigma_{\Delta'}(\Delta, \Gamma \ \nabla_\Sigma \ e :: \tau)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ e{:}\tau :: \sigma(\tau)} \ (\mathit{constrained})$ $\overset{\text{Def. 4.3.7}}{=}$
$\dfrac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \nabla_\Sigma \ e :: \sigma(\tau)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \rhd_\Sigma \ e{:}\tau :: \sigma(\tau)} \ (\mathit{constrained})$ $\overset{FSV(\tau) \subseteq FSV(e:\tau)}{=}$
$\dfrac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \nabla_\Sigma \ e :: \tau}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma e{:}\tau :: \tau} \ (\mathit{constrained})$

  This is a valid application of rule (*constrained*) and yields a sort derivation for $e{:}\tau$ because by induction $\sigma_{\Delta'}(\Delta, \Gamma \ \nabla_\Sigma \ e :: \tau)$ is a sort derivation for e.

□

Proposition 4.3.13 *Substitution preserves sort derivation*

Let $D = \Delta, \Gamma \ \nabla_\Sigma \ f$ be a sort derivation for a non-qualified formulae $f \in \mathrm{PF}_\Sigma(\Phi)$. Let σ be a sort variable substitution, and let $\Delta' \in \mathcal{P}(\mathrm{AF}_\Omega(\Phi)$ be a set of atomic sort formulae.

$$\text{If } \ \forall \alpha \in FSV(f).\ \sigma(\alpha){=}\alpha \ \text{ then } \ \sigma_{\Delta'}(D) \text{ is a sort derivation for } f.$$

Proof: By induction on the structure of D.

Base Cases:

- $\sigma_{\Delta'}(\dfrac{}{\Delta, \Gamma \rhd_\Sigma p}\ (const)) \overset{\text{Def. 4.3.8}}{=} \dfrac{}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma p}\ (const)$

  This is a valid application of rule $(const)$ because p is a predicate identifier from the signature which is not affected by the substitution. Therefore, the substitution yields a sort derivation for p.

- $$\begin{aligned}\sigma_{\Delta'}(\frac{\Delta, \Gamma \bigtriangledown_\Sigma t :: \tau}{\Delta, \Gamma \rhd_\Sigma p\, t}\ (appl)) &\overset{\text{Def. 4.3.8}}{=} \frac{\sigma_{\Delta'}(\Delta, \Gamma \bigtriangledown_\Sigma t :: \tau)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma p\, t}\ (appl)\\ &\overset{\text{Def. 4.3.7}}{=} \frac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \bigtriangledown_\Sigma t :: \sigma(\tau)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma p\, t}\ (appl)\\ &\overset{*)}{=} \frac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \bigtriangledown_\Sigma t :: \tau}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma p\, t}\ (appl)\end{aligned}$$

  ad $*$) Because $p{:}\tau \in P$ and $\tau \in \mathrm{T}_\Omega(\emptyset)$.

  This is a valid application of rule $(appl)$ because p is a predicate identifier from the signature which is not affected by the substitution. By Prop. 4.3.12 we know that, if $\Delta, \Gamma \bigtriangledown_\Sigma t :: \tau$ is a sort derivation for t, then $\sigma_{\Delta'}(\Delta, \Gamma \bigtriangledown_\Sigma t :: \tau)$ is also a sort derivation for t under the assumption made above. Therefore, the application yields a sort derivation for $p\, t$.

- $$\begin{aligned}\sigma_{\Delta'}(\frac{\Delta, \Gamma \bigtriangledown_\Sigma t :: \tau'}{\Delta, \Gamma \rhd_\Sigma p\, t}\ (pappl)) &\overset{\text{Def. 4.3.8}}{=} \frac{\sigma_{\Delta'}(\Delta, \Gamma \bigtriangledown_\Sigma t :: \tau')}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma p\, t}\ (pappl)\\ &\overset{\text{Def. 4.3.7}}{=} \frac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \bigtriangledown_\Sigma t :: \sigma(\tau')}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma p\, t}\ (appl)\end{aligned}$$

  Like in case $(polyid)$ of the proof of Prop. 4.3.12, we have to prove the side conditions of the rule $(pappl)$. Let $p{:}\Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau \in P$. We have to prove that $\sigma(\tau')$ is an instance of τ, i.e. there exists a substitution σ_2 such that $\sigma(\tau') = \sigma_2(\tau)$ and $\sigma_2(A_i) \in \sigma(\Delta) \cup \Delta'$, $1 \leq i \leq m$. The proof is the same as in Prop. 4.3.12.

  By Prop. 4.3.12 we know that, if $\Delta, \Gamma \bigtriangledown_\Sigma t :: \tau'$ is a sort derivation for t, then $\sigma_{\Delta'}(\Delta, \Gamma \bigtriangledown_\Sigma t :: \tau')$ is also a sort derivation for t under the assumption made above. Therefore, the application yields a sort derivation for $p\, t$.

Inductive Cases:

- $$\begin{aligned}\sigma_{\Delta'}(\frac{\Delta, \Gamma.x{:}\tau \bigtriangledown_\Sigma f}{\Delta, \Gamma \rhd_\Sigma \forall x.f}\ (univ_u)) &\overset{\text{Def. 4.3.8}}{=} \frac{\sigma_{\Delta'}(\Delta, \Gamma.x{:}\tau \bigtriangledown_\Sigma f)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma \forall x.f}\ (univ_u)\\ &\overset{\text{Def. 4.3.8}\ \text{Def. 3.2.6}}{=} \frac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma).x{:}\sigma(\tau) \bigtriangledown_\Sigma f}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \rhd_\Sigma \forall x.f}\ (univ_u)\end{aligned}$$

  This is a valid application of rule $(univ_u)$ and yields a sort derivation for $\forall x.f$, because, by induction, $\sigma_{\Delta'}(\Delta, \Gamma.x{:}\tau \bigtriangledown_\Sigma f)$ is a sort derivation for f.

- $\sigma_{\Delta'}(\dfrac{\Delta, \Gamma.x{:}\tau \ \triangledown_\Sigma \ f}{\Delta, \Gamma \ \triangleright_\Sigma \ \forall x{:}\tau.f} \ (univ_s))$ $\overset{\text{Def. 4.3.8}}{=}$
  $\dfrac{\sigma_{\Delta'}(\Delta, \Gamma.x{:}\tau \ \triangledown_\Sigma \ f)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \triangleright_\Sigma \ \forall x{:}\tau.f} \ (univ_s)$ $\overset{\text{Def. 4.3.8, 3.2.6}}{=}$
  $\dfrac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma).x{:}\sigma(\tau) \ \triangledown_\Sigma \ f}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \triangleright_\Sigma \ \forall x{:}\tau.f} \ (univ_s)$ $\overset{FSV(\tau) \subseteq FSV(\forall x:\tau.f)}{=}$
  $\dfrac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma).x{:}\tau \ \triangledown_\Sigma \ f}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \triangleright_\Sigma \ \forall x{:}\tau.f} \ (univ_s)$

  Thus, this is a valid application of rule $(univ_s)$ and yields a sort derivation for $\forall x{:}\tau.f$, because, by induction, $\sigma_{\Delta'}(\Delta, \Gamma.x{:}\tau \ \triangledown_\Sigma \ f)$ is a sort derivation for f.

- $\sigma_{\Delta'}(\dfrac{\Delta, \Gamma \ \triangledown_\Sigma \ f}{\Delta, \Gamma \ \triangleright_\Sigma \ \neg f} \ (not))$ $\overset{\text{Def. 4.3.8}}{=}$ $\dfrac{\sigma_{\Delta'}(\Delta, \Gamma \ \triangledown_\Sigma \ f)}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \triangleright_\Sigma \ \neg f} \ (not)$
  $\overset{\text{Def. 4.3.8}}{=}$ $\dfrac{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \triangledown_\Sigma \ f}{\sigma(\Delta) \cup \Delta', \sigma(\Gamma) \ \triangleright_\Sigma \ \neg f} \ (not)$

  This is a valid application of rule (not) and yields a sort derivation for f, because, by induction, $\sigma_{\Delta'}(\Delta, \Gamma \ \triangledown_\Sigma \ f)$ is a sort derivation for f.

- Case (or) is similar to (not).

□

Proposition 4.3.14 *The interpretation function* $\mathcal{A}\left[\!\left[.\right]\!\right]_{\nu,\eta}$ *is well-defined*

Let $D = \dfrac{\Delta, \Gamma \ \triangledown_\Sigma \ f}{\Delta, \Gamma \ \triangleright_\Sigma \ \forall \alpha_1, \ldots, \alpha_n.A_1, \ldots, A_m \Rightarrow f} \ (ext.qual.)$ be an extended sort derivation for a qualified formula. Let ν be a sort variable assignment and η be a variable assignment.

$$\text{If } \models_{\mathcal{SA},\nu} \Delta \text{ and } \eta \models_{\mathcal{SA},\nu} \Gamma$$
$$\text{then } \mathcal{A}\left[\!\left[D\right]\!\right]_{\nu,\eta} \text{ yields a uniquely determined result in } \{\mathit{true}, \mathit{false}\}.$$

Proof: This proof is very similar to the proof of Prop. 4.2.4. Like in that proof we must show that $\mathcal{A}\left[\!\left[\Delta, \Gamma \ \triangledown_\Sigma \ f\right]\!\right]_{\nu,\eta}$ yields a uniquely determined truth value. Again, this holds by Prop. 4.2.3 because $\models_{\mathcal{SA},\nu} \Delta$ holds. However, in contrast to Prop. 4.2.4, this proposition only holds because we explicitly assumed $\models_{\mathcal{SA},\nu} \Delta$. □

Proposition 4.3.15 *If an extended sort derivation is more general then it is more restricting*

Let $D_1{=}\Delta_1, \emptyset \ \triangledown'_\Sigma \ qf$ and $D_2{=}\Delta_2, \emptyset \ \triangledown'_\Sigma \ qf$ be two extended sort derivations for a closed qualified formula $qf = \forall \overline{\alpha_n}.\overline{A_m} \Rightarrow f$. If D_2 is more general than D_1 then D_2 is more restricting than D_1, i.e.

$$D_1 \leq D_2 \quad \Rightarrow \quad D_2 \models D_1$$

Proof: The proceeding is the same as in Prop. 4.3.5. Assuming $D_1 \leq D_2$ we have to show that $D_2 \models D_1$. By Def. 4.3.11 we must prove that for every Σ-algebra $\mathcal{A}$ the following holds:

$$\begin{aligned}
&\mathcal{A} \models D_2 \quad \Rightarrow \quad \mathcal{A} \models D_1 \qquad \overset{\text{Def. 4.3.10}}{\Leftrightarrow} \\
&\quad \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu,\eta_0} = \mathit{true} \quad \text{for every } \nu : \Phi \to \mathcal{U}.\ \models_{\mathcal{SA},\nu} \Delta_2 \\
&\quad \Rightarrow \\
&\quad \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta_0} = \mathit{true} \quad \text{for every } \nu : \Phi \to \mathcal{U}.\ \models_{\mathcal{SA},\nu} \Delta_1
\end{aligned}$$

where η_0 is an arbitrary variable assumption.

Let ν be an arbitrary but fixed sort variable assignment with $\models_{\mathcal{SA},\nu} \Delta_1$. If we can show that there exists a sort variable assignment ν^σ such that $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$ and $\mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta_0} = \mathit{true} \Rightarrow \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta_0} = \mathit{true}$, we have proven the proposition. We prove this by constructing a sort variable assignment ν^σ. Because of $D_1 \leq D_2$, by Def. 4.3.13, there exists a sort variable substitution σ such that $\sigma(\alpha_i) = \alpha_i$, $1 \leq i \leq n$ and $\Delta_1 \vdash_{SA} \sigma(\Delta_2)$ and $D_1 \leq^\sigma D_2$. We use this substitution to define ν^σ in the following way:

$$\nu^\sigma(\alpha) := \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu$$

By Prop. 4.3.16 we know that $\mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} = \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta}$ if $\sigma(\alpha_i) = \alpha_i$, $1 \leq i \leq n$ and $D_1 \leq^\sigma D_2$ and $\nu^\sigma(\alpha) = \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu$ and $\models_{\mathcal{SA},\nu} \Delta_1$ and $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$. Thus, it remains to show $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$. This follows by Prop. 4.1.4 because $\Delta_1 \vdash_{SA} \sigma(\Delta_2)$. □

Proposition 4.3.16 *Interpretation equivalent extended sort derivations*

Let

$$D_1 = \frac{\Delta_1, \emptyset\ \triangledown_\Sigma\ f}{\Delta_1, \emptyset\ \triangleright_\Sigma\ qf}\ (\mathit{ext.qual.}) \quad \text{and} \quad D_2 = \frac{\Delta_2, \emptyset\ \triangledown_\Sigma\ f}{\Delta_2, \emptyset\ \triangleright_\Sigma\ qf}\ (\mathit{ext.qual.})$$

be two extended sort derivations for a closed qualified formula $qf = \forall \alpha_1, \ldots, \alpha_n.\ A_1, \ldots, A_m \Rightarrow f$. Let σ be a sort variable substitution such that $\sigma(\alpha_i) = \alpha_i$, $1 \leq i \leq n$ and $D_1 \leq^\sigma D_2$. Let ν and ν^σ be sort variable assignments such that $\models_{\mathcal{SA},\nu} \Delta_1$ and $\models_{\mathcal{SA},\nu^\sigma} \Delta_2$. Let η be an arbitrary variable assignment.

$$\text{If} \quad \nu^\sigma(\alpha) = \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu \text{ for each } \alpha \in \Phi \quad \text{then} \quad \mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} = \mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta}$$

Proof:

$$\mathcal{A}\left[\!\left[D_1\right]\!\right]_{\nu,\eta} \overset{\text{Def. 4.3.9}}{=} \begin{cases} \mathit{true} & \text{if not } \models_{\mathcal{SA},\nu} \{\overline{A_m}\} \quad \text{or} \quad \mathcal{A}\left[\!\left[\Delta_1, \emptyset\ \triangledown_\Sigma\ f\right]\!\right]_{\nu,\eta} = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\mathcal{A}\left[\!\left[D_2\right]\!\right]_{\nu^\sigma,\eta} \overset{\text{Def. 4.3.9}}{=} \begin{cases} \mathit{true} & \text{if not } \models_{\mathcal{SA},\nu^\sigma} \{\overline{A_m}\} \quad \text{or} \quad \mathcal{A}\left[\!\left[\Delta_2, \emptyset\ \triangledown_\Sigma\ f\right]\!\right]_{\nu^\sigma,\eta} = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}$$

By Prop. 4.3.6 we know that

$$\mathcal{A}\left[\!\left[\Delta_1, \emptyset \bigtriangledown_\Sigma f\right]\!\right]_{\nu,\eta} = \mathcal{A}\left[\!\left[\Delta_2, \emptyset \bigtriangledown_\Sigma f\right]\!\right]_{\nu^\sigma,\eta}$$

if

$$\begin{array}{ll}
\Delta_1, \emptyset \bigtriangledown_\Sigma f \leq^\sigma \Delta_2, \emptyset \bigtriangledown_\Sigma f & \text{and} \\
\models_{\mathcal{SA},\nu} \Delta_1 \text{ and } \models_{\mathcal{SA},\nu^\sigma} \Delta_2 & \text{and} \\
\eta \models_{\mathcal{SA},\nu} \emptyset & \text{and} \\
\eta \models_{\mathcal{SA},\nu^\sigma} \emptyset & \text{and} \\
\nu^\sigma(\alpha) = \mathcal{SA}[\![\sigma(\alpha)]\!]_\nu. &
\end{array}$$

All these conditions are trivially true. Therefore, it remains to show:

$$\models_{\mathcal{SA},\nu} \{\overline{A_m}\} \quad \overset{*)}{\Leftrightarrow} \quad \models_{\mathcal{SA},\nu} \sigma(\{\overline{A_m}\})$$
$$\overset{\text{Prop. 4.1.2}}{\Leftrightarrow} \quad \models_{\mathcal{SA},\nu^\sigma} \{\overline{A_m}\}$$

ad $*$) $A_j = \sigma(A_j)$ because by Def. 3.2.4 $A_j \in \mathrm{AF}_\Omega(\{\alpha_1, \ldots, \alpha_n\})$ and $\sigma(\alpha_i) = \alpha_i$, $1 \leq i \leq n$.
This completes the proof. □

Proposition 4.3.18 *Equivalent extended sort derivations are satisfaction equivalent*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let D_1 and D_2 be two extended sort derivations for a closed qualified formula.

$$\text{if} \quad D_1 \leq D_2 \text{ and } D_2 \leq D_1 \quad \text{then} \quad \mathcal{A} \models D_1 \Leftrightarrow \mathcal{A} \models D_2$$

Proof:

"$\Rightarrow$" $D_1 \leq D_2 \overset{\text{Prop. 4.3.15}}{\Rightarrow} D_2 \models D_1 \overset{\text{Def. 4.3.11}}{\Leftrightarrow} \mathcal{A} \models D_2 \Rightarrow \mathcal{A} \models D_1$

"$\Leftarrow$" $D_2 \leq D_1 \overset{\text{Prop. 4.3.15}}{\Rightarrow} D_1 \models D_2 \overset{\text{Def. 4.3.11}}{\Leftrightarrow} \mathcal{A} \models D_1 \Rightarrow \mathcal{A} \models D_2$

□

Proposition 4.3.19 *P-Satisfaction implies satisfaction*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let $qf \in \mathrm{QF}_\Sigma$ be a closed well-formed qualified Σ-formula. Formula qf is satisfied in $\mathcal{A}$ if qf is p-satisfied in $\mathcal{A}$, i.e.

$$\mathcal{A} \models_p qf \quad \Rightarrow \quad \mathcal{A} \models qf$$

Proof:

$$\begin{array}{ll}
\mathcal{A} \models_p qf & \overset{\text{Def. 4.3.15}}{\Leftrightarrow} \quad \mathcal{A} \models D \text{ for some principal extended sort derivation } D \text{ for } qf \\
& \overset{*)}{\Leftrightarrow} \quad \mathcal{A} \models D \text{ for every extended sort derivation } D \text{ that ends with} \\
& \qquad \Delta, \emptyset \bigtriangledown'_\Sigma qf \text{ for some sort predicate assumption } \Delta \\
& \overset{\text{Prop. 4.3.20}}{\Rightarrow} \quad \mathcal{A} \models D \text{ for every sort derivation } D \text{ that ends with} \\
& \qquad \Delta, \emptyset \bigtriangledown_\Sigma qf \text{ for some sort predicate assumption } \Delta \\
& \overset{\text{Def. 4.2.7}}{\Leftrightarrow} \quad \mathcal{A} \models qf
\end{array}$$

ad $*$)

"$\Rightarrow$" Because D is a principal extended sort derivation for qf, by Def. 4.3.14, $D' \leq D$ holds for each extended sort derivation D' for qf. Therefore, by Prop. 4.3.15 $D \models D' \overset{\text{Def. 4.3.11}}{\Leftrightarrow} \mathcal{A} \models D \Rightarrow \mathcal{A} \models D'$ for each D'.

"$\Leftarrow$" This direction trivially holds because the principal extended sort derivation also ends with $\Delta, \emptyset \rhd'_\Sigma qf$ for some sort predicate assumption Δ.

□

Proposition 4.3.20 *Existence of satisfaction implying extended sort derivations*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let $qf = \forall \overline{\alpha_n}.\overline{A_m} \Rightarrow f$ be a qualified formula. For each sort derivation $D = \Delta_1, \emptyset \bigtriangledown_\Sigma qf$ there exists an extended sort derivation D'=$\Delta_1, \emptyset \bigtriangledown'_\Sigma qf$ such that

$$\mathcal{A} \models D' \quad \Rightarrow \quad \mathcal{A} \models D$$

Proof: In Prop. 4.3.11 we showed that, if there exists a sort derivation D for a qualified formula qf, there also exists an extended sort derivation D' for qf. We proved this by constructing an extended sort derivation D' from an existing sort derivation D for qf. In addition, we now show that $\mathcal{A} \models D' \Rightarrow \mathcal{A} \models D$:

Let $D = \dfrac{\Delta, \emptyset \bigtriangledown_\Sigma f}{\Delta, \emptyset \rhd_\Sigma qf}\ (qual.)$ be an arbitrary sort derivation for qf. By Prop. 4.3.11 we know that D'= $\dfrac{\Delta, \emptyset \bigtriangledown_\Sigma f}{\Delta, \emptyset \rhd'_\Sigma qf}\ (ext.\ qual.)$ is an extended sort derivation for qf.

$$
\begin{array}{ll}
(\mathcal{A} \models D' \ \Rightarrow \ \mathcal{A} \models D) & \overset{\text{Def. 4.3.10, 4.2.6}}{\Leftrightarrow} \\
(\mathcal{A}\left[\!\left[D' \right]\!\right]_{\nu,\eta_0} = \mathit{true} \quad \text{for every } \nu : \Phi \to \mathcal{U}.\ \models_{\mathcal{SA},\nu} \Delta & \\
\Rightarrow & \\
\mathcal{A}\left[\!\left[D \right]\!\right]_{\nu,\eta_0} = \mathit{true} \quad \text{for every } \nu : \Phi \to \mathcal{U}.\ \models_{\mathcal{SA},\nu} \Delta) & \overset{\text{Def. 4.3.9, 4.2.5}}{\Leftrightarrow} \\
(\models_{\mathcal{SA},\nu} \Delta \ \Rightarrow \ \mathcal{A}\left[\!\left[\{A_1, \ldots, A_m\}, \emptyset \bigtriangledown_\Sigma f \right]\!\right]_{\nu,\eta_0} & \\
\text{for every } \nu : \Phi \to \mathcal{U}.\ \models_{\mathcal{SA},\nu} \{A_1, \ldots, A_m\} & \\
\Rightarrow & \\
\models_{\mathcal{SA},\nu} \{A_1, \ldots, A_m\} \ \Rightarrow \ \mathcal{A}\left[\!\left[\{A_1, \ldots, A_m\}, \emptyset \bigtriangledown_\Sigma f \right]\!\right]_{\nu,\eta_0} & \\
\text{for every } \nu : \Phi \to \mathcal{U}.\ \models_{\mathcal{SA},\nu} \Delta) &
\end{array}
$$

This completes the proof. □

Proposition 4.3.21 *X-Satisfaction is equivalent to p-satisfaction*

Let $\mathcal{A}$ be a polymorphic Σ-algebra. Let $qf \in \mathrm{QF}_\Sigma$ be a closed well-formed qualified Σ-formula. Formula qf is x-satisfied in $\mathcal{A}$ iff qf is p-satisfied in $\mathcal{A}$, i.e.

$$\mathcal{A} \models_x qf \quad \Leftrightarrow \quad \mathcal{A} \models_p qf$$

Proof: See first part of the proof of Prop. 4.3.19 □

Proposition 4.3.22 *The p-model concept is more rigorous than the model concept*

Let $PS = (\Sigma, A)$ be a polymorphic specification. Let $\mathcal{A}$ be a polymorphic Σ-algebra.

$$\text{If} \quad \mathcal{A} \text{ is a p-model of } PS \quad \text{then} \quad \mathcal{A} \text{ is a model of } PS$$

Proof: By Def. 4.3.17 and 4.2.8 we have to prove:

$$\forall f \in A.\ \mathcal{A} \models_p f \Rightarrow \forall f \in A.\ \mathcal{A} \models f$$

If $f \in \mathrm{PF}_\Sigma(\Phi)$ is a non-qualified well-formed formula, then, by Prop. 4.3.10, we know that $\mathcal{A} \models_p f \ \Rightarrow\ \mathcal{A} \models f$. If $f \in \mathrm{QF}_\Sigma$ is a qualified well-formed formula, then, by Prop. 4.3.19, we know that $\mathcal{A} \models_p f \ \Rightarrow\ \mathcal{A} \models f$. Thus, the proposition holds. □

Proposition 4.3.23 *The p-model concept is equivalent to the x-model concept*

Let $PS = (\Sigma, A)$ be a polymorphic specification. Let $\mathcal{A}$ be a polymorphic Σ-algebra.

$$\mathcal{A} \text{ is a p-model of } PS \quad \text{iff} \quad \mathcal{A} \text{ is an x-model of } PS$$

Proof: Follows immediately from Prop. 4.3.10 and Prop. 4.3.21. □

Proposition 5.2.1 *Sort inference algorithm W is sound w.r.t.* $\rhd$

Let $\Sigma = (S, P, F)$ be a polymorphic signature. Let Γ be a variable assumption, and let e be a pre-term. If

$$W(F, \Gamma, e) = (\Delta, \sigma, \tau, D)$$

then

1. $\forall \alpha \notin UV.\ \sigma(\alpha) = \alpha$
2. $\Delta, \sigma(\Gamma) \rhd_\Sigma e :: \tau$
3. D is a sort derivation ending with $\Delta, \sigma(\Gamma) \rhd_\Sigma e :: \tau$

Proof: The first proposition is obvious because σ results from compositions of substitutions computed by mgu. Thus, we omit the formal proof. The second proposition follows immediately from the third proposition. We prove the third proposition by induction on the structure of e.

Base Cases:

$e = x$**:** We have

$$W(F, \Gamma, x) \quad \overset{\text{Def. } 5.2.1}{=} \quad (\emptyset, \varepsilon, \Gamma(x),\ \frac{}{\emptyset, \Gamma \rhd_\Sigma x :: \Gamma(x)}\ (var))$$

By Def. 3.2.7 $\frac{}{\emptyset, \Gamma \rhd_\Sigma x :: \Gamma(x)}\ (var)$ is a proof tree for $\emptyset, \varepsilon(\Gamma) \rhd_\Sigma x :: \Gamma(x)$.

$e = c$**:** $F(c) \in \mathrm{T}_\Omega(\emptyset)$. We have

$$W(F,\Gamma,c) \quad \stackrel{\text{Def. 5.2.1}}{=} \quad (\emptyset, \varepsilon, F(c), \ \frac{}{\emptyset, \Gamma \rhd_\Sigma c :: F(c)}\ (id))$$

By Def. 3.2.7 $\frac{}{\emptyset, \Gamma \rhd_\Sigma c :: F(c)}\ (id)$ is a proof tree for $\emptyset, \varepsilon(\Gamma) \rhd_\Sigma c :: F(c)$.

$e = c$**:** $F(c) \in \Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau$. We have

$$W(F,\Gamma,c) \quad \stackrel{\text{Def. 5.2.1}}{=} \quad ([\overline{\beta_n/\alpha_n}]\overline{A_m}, \varepsilon, [\overline{\beta_n/\alpha_n}]\tau, D)$$

$$\textbf{where} \quad D = \frac{}{[\overline{\beta_n/\alpha_n}]\overline{A_m}, \Gamma \rhd_\Sigma c :: [\overline{\beta_n/\alpha_n}]\tau}\ (polyid)$$

By Def. 3.2.7 D is a proof tree for $[\overline{\beta_n/\alpha_n}]\overline{A_m}, \varepsilon(\Gamma) \rhd_\Sigma c :: [\overline{\beta_n/\alpha_n}]\tau$ because the side condition of rule (*polyid*) is trivially fulfilled.

Inductive Cases:

$e = e_1 e_2$**:** We have

$$\begin{aligned} W(F,\Gamma,e_1\,e_2) \quad = \quad & (\Delta, u\sigma_2\sigma_1, u(\alpha), D) \\ & \textbf{where} \quad (\Delta_1, \sigma_1, \tau_1, D_1) = W(F,\Gamma,e_1) \\ & \qquad\quad (\Delta_2, \sigma_2, \tau_2, D_2) = W(F,\sigma_1(\Gamma),e_2) \\ & \qquad\quad u = mgu(\sigma_2(\tau_1), \tau_2 \to \alpha) \\ & \qquad\quad \Delta = u(\sigma_2(\Delta_1) \cup \Delta_2) \\ & \qquad\quad D = \frac{(u\sigma_2)_{u(\Delta_2)}(D_1)\ \ u_{u\sigma_2(\Delta_1)}(D_2)}{\Delta, u\sigma_2\sigma_1(\Gamma) \rhd_\Sigma e_1 e_2 :: u(\alpha)}\ (\to E) \end{aligned}$$

By applying induction hypothesis we know that D_1 is a proof tree for $\Delta_1, \sigma_1(\Gamma) \rhd_\Sigma e_1 :: \tau_1$ and that D_2 is a proof tree for $\Delta_2, \sigma_2\sigma_1(\Gamma) \rhd_\Sigma e_2 :: \tau_2$. By Def. 3.2.7 it follows immediately that D is a proof tree for $\Delta, u\sigma_2\sigma_1(\Gamma) \rhd_\Sigma e_1 e_2 :: u(\alpha)$ if

$$(u\sigma_2)_{u(\Delta_2)}(D_1) \quad \stackrel{\text{Def. 4.3.7}}{=} \quad \Delta, u\sigma_2\sigma_1(\Gamma) \bigtriangledown_\Sigma e_1 :: u\sigma_2(\tau_1)$$

is a proof tree for e_1 and if

$$u_{u\sigma_2(\Delta_1)}(D_2) \quad \stackrel{\text{Def. 4.3.7}}{=} \quad \Delta, u\sigma_2\sigma_1(\Gamma) \bigtriangledown_\Sigma e_2 :: u(\tau_2)$$

is a proof tree for e_2 because by Def. 5.1 $u\sigma_2(\tau_1) = u(\tau_2 \to \alpha) = u(\tau_2) \to u(\alpha)$.

By Prop. 4.3.12 $(u\sigma_2)_{u(\Delta_2)}(D_1)$ is a sort derivation for e_1 if:

i) D_1 is a sort derivation for e_1

ii) $\forall \alpha \in FSV(e_1).u\sigma_2(\alpha) = \alpha$

ad i) by induction hypothesis

ad ii) By Def. 5.1 mgu yields a substitution u with $\forall \alpha \notin UV.u(\alpha) = \alpha$. Furthermore, by 1. $\forall \alpha \notin UV.\sigma_1(\alpha) = \alpha$. Thus, ii) follows immediately because $FSV(e_1) \cap UV = \emptyset$.

The proof for $u_{u\sigma_2(\Delta_1)}(D_2)$ is similar to $(u\sigma_2)_{u(\Delta_2)}(D_1)$.

$e = \lambda x.e_1$**:** We have

$$W(F,\Gamma,\lambda x.e_1) \overset{\text{Def. 5.2.1}}{=} (\Delta_1, \sigma_1, \sigma_1(\alpha) \to \tau_1, D)$$
$$\textbf{where} \quad (\Delta_1, \sigma_1, \tau_1, D_1) = W(F, \Gamma.x{:}\alpha, e_1)$$
$$D = \frac{D_1}{\Delta_1, \sigma_1(\Gamma) \rhd_\Sigma \lambda x.e_1 :: \sigma_1(\alpha) \to \tau_1} \; (\to I_u)$$

By induction hypothesis we know that D_1 is a proof tree for $\Delta_1, \sigma_1(\Gamma.x{:}\alpha) \rhd_\Sigma e_1 :: \tau_1$. Thus, by Def. 3.2.7 D is a proof tree for $\Delta_1, \sigma_1(\Gamma) \rhd_\Sigma \lambda x.e_1 :: \sigma_1(\alpha) \to \tau_1$ because $\sigma_1(\Gamma.x{:}\alpha) = \sigma_1(\Gamma).x{:}\sigma_1(\alpha)$.

$e = \lambda x{:}\tau_2.e_1$**:** We have

$$W(F,\Gamma,\lambda x{:}\tau_2.e_1) \overset{\text{Def. 5.2.1}}{=} (\Delta_1, \sigma_1, \tau_2 \to \tau_1, D)$$
$$\textbf{where} \quad (\Delta_1, \sigma_1, \tau_1, D_1) = W(F, \Gamma.x{:}\tau_2, e_1)$$
$$D = \frac{D_1}{\Delta_1, \sigma_1(\Gamma) \rhd_\Sigma \lambda x{:}\tau_2.e_1 :: \tau_2 \to \tau_1} \; (\to I_s)$$

By induction hypothesis we know that D_1 is a proof tree for $\Delta_1, \sigma_1(\Gamma.x{:}\tau_2) \rhd_\Sigma e_1 :: \tau_1$. Thus, by Def. 3.2.7 D is a proof tree for $\Delta_1, \sigma_1(\Gamma) \rhd_\Sigma \lambda x{:}\tau_2.e_1 :: \tau_2 \to \tau_1$ because:

$$\begin{aligned} \sigma_1(\Gamma.x{:}\tau_2) &= \sigma_1(\Gamma).x{:}\sigma_1(\tau_2) \\ &\overset{*)}{=} \sigma_1(\Gamma).x{:}\tau_2 \end{aligned}$$

ad $*$) $FSV(\tau) \cap UV = \emptyset$ and by 1. $\forall \alpha \notin UV.\sigma_1(\alpha) = \alpha$

$e = e_1{:}\tau$**:** We have

$$W(F,\Gamma,e_1{:}\tau) \overset{\text{Def. 5.2.1}}{=} (u(\Delta_1), u\sigma_1, \tau, D)$$
$$\textbf{where} \quad (\Delta_1, \sigma_1, \tau_1, D_1) = W(F, \Gamma, e_1)$$
$$u = mgu(\tau_1, \tau)$$
$$D = \frac{u_\emptyset(D_1)}{u(\Delta_1), u\sigma_1(\Gamma) \rhd_\Sigma e_1{:}\tau :: \tau} \; (constrained)$$

By induction hypothesis we know that D_1 is a proof tree for $\Delta_1, \sigma_1(\Gamma) \rhd_\Sigma e_1 :: \tau$. By Def. 3.2.7 it follows immediately that D is a proof tree for $u(\Delta_1), u\sigma_1(\Gamma) \rhd_\Sigma e_1{:}\tau :: \tau$ if

$$u_\emptyset(D_1) \overset{\text{Def. 4.3.7}}{=} u(\Delta_1), u\sigma_1(\Gamma) \bigtriangledown_\Sigma e_1 :: u(\tau) \overset{u(\tau)=\tau}{=} u(\Delta_1), u\sigma_1(\Gamma) \bigtriangledown_\Sigma e_1 :: \tau$$

is a proof tree for $u(\Delta_1), u\sigma_1(\Gamma) \rhd_\Sigma e_1 :: \tau$.

By Prop. 4.3.12 $u_\emptyset(D_1)$ is a sort derivation for e_1 if:

i) D_1 is a sort derivation for e_1

ii) $\forall \alpha \in FSV(e_1).u(\alpha) = \alpha$

ad i) by induction hypothesis

ad ii) by Def. 5.1 because $FSV(e_1) \cap UV = \emptyset$

□

Proposition 5.2.2 *Sort inference algorithm W is complete w.r.t.* $\rhd$

Let $\Sigma = ((\Omega, SA), P, F)$ be a polymorphic signature. Let Γ be a variable assumption, and let e be a pre-term. If there exists a sort derivation $\Delta', \sigma'(\Gamma) \bigtriangledown_\Sigma e :: \tau'$ such that $\forall \alpha \notin UV : \sigma'(\alpha) = \alpha$ then

$$W(F, \Gamma, e) = (\Delta, \sigma, \tau, (\Delta, \sigma(\Gamma) \rhd_\Sigma e :: \tau))$$

and there exists a substitution ρ such that

$$\begin{gathered} \forall \alpha \notin UV.\rho(\alpha) = \alpha \\ \rho(\Delta) \subseteq \Delta' \\ \sigma'(\Gamma) = \rho\sigma(\Gamma) \\ \Delta', \sigma'(\Gamma) \bigtriangledown_\Sigma e :: \tau' \leq^\rho \Delta, \sigma(\Gamma) \bigtriangledown_\Sigma e :: \tau \end{gathered}$$

Proof: By induction on the structure of e. To avoid unnecessary complications we assume that all λ-bound variables in e are different and that all variables in Γ are different from all λ-bound variables in e. This can be achieved by performing a suitable α-conversion on e. We do not lose generality by this assumption because we can easily show that sort derivations are invariant under α-conversion.

Base Cases:

$e = x$: We have a derivation of the form

$$\frac{}{\Delta', \sigma'(\Gamma) \rhd_\Sigma x :: \tau'} \ (var)$$

where $\sigma'(\Gamma)(x) = \tau'$. By Def. 5.2.1:

$$W(F, \Gamma, x) \quad = \quad (\emptyset, \varepsilon, \Gamma(x), \ \frac{}{\emptyset, \Gamma \rhd_\Sigma x :: \Gamma(x)} \ (var))$$

The algorithm does not fail in the call of Γ because $\sigma'(\Gamma)(x) = \tau'$. Thus, W results in the tuple above. It remains to show that there exists a substitution ρ such that:

i) $\forall \alpha \notin UV.\rho(\alpha) = \alpha$

ii) $\rho(\emptyset) \subseteq \Delta'$

iii) $\sigma'(\Gamma) = \rho\varepsilon(\Gamma)$

iv) $\frac{}{\Delta', \sigma'(\Gamma) \rhd_\Sigma x :: \tau'} \ (var) \leq^\rho \frac{}{\emptyset, \Gamma \rhd_\Sigma x :: \Gamma(x)} \ (var)$

Let $\rho = \sigma'$:

ad i) Trivially valid because we assumed $\forall \alpha \notin UV.\sigma'(\alpha) = \alpha$.

ad ii) Trivially valid.

ad iii) Trivially valid.

ad iv) Immediately by Def. 4.3.2.

$e = c$: $\mathrm{F(c)} \in \mathrm{T}_\Omega(\emptyset)$. Similar to $e = x$.

$e = c$: $\mathrm{F(c)} \in \Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau$. We have a derivation D' of the form

$$\frac{}{\Delta', \sigma'(\Gamma) \rhd_\Sigma x :: \sigma(\tau)} \ (polyid)$$

where σ is an arbitrary substitution and $\sigma(\{\overline{A_m}\}) \subseteq \Delta'$. By Def. 5.2.1:

$$\begin{aligned} W(F, \Gamma, c) \quad = \quad & ([\overline{\beta_n/\alpha_n}]\overline{A_m}, \varepsilon, [\overline{\beta_n/\alpha_n}]\tau, D) \\ & \textbf{where} \quad D = \frac{}{[\overline{\beta_n/\alpha_n}]\overline{A_m}, \Gamma \rhd_\Sigma c :: [\overline{\beta_n/\alpha_n}]\tau} \ (polyid) \\ & \overline{\beta_n} \text{ are new variables from } UV \end{aligned}$$

The algorithm does not fail in the call of F because $F(c) \in \Pi\overline{\alpha_n}.\overline{A_m} \Rightarrow \tau$. Thus, W results in the tuple above. It remains to show that there exists a substitution ρ such that:

i) $\forall \alpha \notin UV.\rho(\alpha) = \alpha$

ii) $\rho[\overline{\beta_n/\alpha_n}](\overline{A_m}) \subseteq \Delta'$

iii) $\sigma'(\Gamma) = \rho\varepsilon(\Gamma)$

iv) $D' \leq^{\rho} D$

Let $\rho = \sigma'.[\overline{\sigma(\alpha_n)/\beta_n}]$:

ad i) Because $\beta_i \in UV$ and we assumed $\forall \alpha \notin UV.\sigma'(\alpha) = \alpha$.
ad ii)

$$\begin{array}{lll} (\sigma'.[\overline{\sigma(\alpha_n)/\beta_n}])[\overline{\beta_n/\alpha_n}](\overline{A_m}) \subseteq \Delta' & \Leftrightarrow & \sigma'.[\overline{\sigma(\alpha_n)/\alpha_n}](\overline{A_m}) \subseteq \Delta' \\ & \underset{FSV(\overline{A_m}) \subseteq \{\overline{\alpha_n}\}}{\Leftrightarrow} & [\overline{\sigma(\alpha_n)/\alpha_n}](\overline{A_m}) \subseteq \Delta' \\ & \Leftrightarrow & \sigma(\overline{A_m}) \subseteq \Delta' \end{array}$$

This is the side condition of the derivation D_1.

ad iii) $$\begin{array}{lll} \sigma'(\Gamma) & = & \sigma'.[\overline{\sigma(\alpha_n)/\beta_n}](\Gamma) \\ & \underset{\overline{\beta_n} \notin FSV(\Gamma)}{=} & \sigma'(\Gamma) \end{array}$$

ad iv) By Def. 4.3.2 we have to show:

$$\begin{array}{lll} \sigma(\tau) = (\sigma'.[\overline{\sigma(\alpha_n)/\beta_n}])[\overline{\beta_n/\alpha_n}](\tau) & \Leftrightarrow & \sigma(\tau) = \sigma'.[\overline{\sigma(\alpha_n)/\alpha_n}](\tau) \\ & \underset{FSV(\tau) = \{\overline{\alpha_n}\}}{\Leftrightarrow} & \sigma(\tau) = [\overline{\sigma(\alpha_n)/\alpha_n}](\tau) \\ & \Leftrightarrow & \sigma(\tau) = \sigma(\tau) \end{array}$$

Inductive Cases:

$e = e_1 e_2$**:** We have a derivation D' of the form

$$\frac{D_1' \qquad D_2'}{\Delta', \sigma'(\Gamma) \rhd_\Sigma e_1 e_2 :: \tau_1'} \ (\rightarrow E)$$

where $D_1 = \Delta', \sigma'(\Gamma) \bigtriangledown_\Sigma e_1 :: \tau_2' \rightarrow \tau_1'$ and
$D_2 = \Delta', \sigma'(\Gamma) \bigtriangledown_\Sigma e_2 :: \tau_2'$

By applying induction hypothesis to e_1 we know:

1. $W(F, \Gamma, e_1) = (\Delta_1, \sigma_1, \tau_1, D_1)$
   where $D_1 = \Delta_1, \sigma_1(\Gamma) \bigtriangledown_\Sigma e_1 :: \tau_1$
2. There exists a substitution ρ_1 such that:

(a) $\forall \alpha \notin UV.\rho_1(\alpha) = \alpha$
(b) $\rho_1(\Delta_1) \subseteq \Delta'$
(c) $\sigma'(\Gamma) = \rho_1\sigma_1(\Gamma)$
(d) $D_1' \leq^{\rho_1} D_1$

Because $\sigma'(\Gamma) = \rho_1\sigma_1(\Gamma)$ and $\forall \alpha \notin UV.\rho_1(\alpha) = \alpha$ we can apply induction hypothesis to e_2 with $\sigma_1(\Gamma)$ yielding:

3. $W(F, \sigma_1(\Gamma), e_2) = (\Delta_2, \sigma_2, \tau_2, D_2)$
   where $D_2 = \Delta_2, \sigma_2\sigma_1(\Gamma) \bigtriangledown_\Sigma e_2 :: \tau_2$
4. There exists a substitution ρ_2 such that:
   (a) $\forall \alpha \notin UV.\rho_2(\alpha) = \alpha$
   (b) $\rho_2(\Delta_2) \subseteq \Delta'$
   (c) $\sigma'(\Gamma) = \rho_2\sigma_2\sigma_1(\Gamma)$
   (d) $D_2' \leq^{\rho_2} D_2$

By Def. 5.2.1:

$$W(F, \Gamma, e_1\, e_2) = (\Delta, u\sigma_2\sigma_1, u(\alpha), D)$$
$$\textbf{where} \quad u = mgu(\sigma_2(\tau_1), \tau_2 \to \alpha)$$
$$\Delta = u(\sigma_2(\Delta_1) \cup \Delta_2)$$
$$D = \frac{(u\sigma_2)_{u(\Delta_2)}(D_1)\ u_{u\sigma_2(\Delta_1)}(D_2)}{\Delta, u\sigma_2\sigma_1(\Gamma) \rhd_\Sigma e_1 e_2 :: u(\alpha)}\ (\to E)$$

By 1. and 3. the recursive calls of W terminate. The algorithm may, however, fail during unification of $\sigma_2(\tau_1)$ and $\tau_2 \to \alpha$. To prove that W results in the tuple above we must show that $\sigma_2(\tau_1)$ and $\tau_2 \to \alpha$ are unifiable w.r.t. UV, i.e. that there exists a substitution u' such that $u'(\sigma_2(\tau_1)) = u'(\tau_2 \to \alpha)$. Let u' be defined as:

$$u'(\beta) = \begin{cases} \rho_1(\beta) & \text{if } \beta \in FSV(D_1) \setminus FSV(\sigma_2) \\ \tau_1' & \text{if } \beta = \alpha \\ \rho_2(\beta) & \text{otherwise} \end{cases}$$

In the following case analyses we will always omit case $\beta = \alpha$ because α is a new sort variable and cannot occur in the previous derivations. We show:

$$u'(\sigma_2(\tau_1)) \overset{\text{i)}}{=} \tau_2' \to \tau_1' \overset{\text{ii)}}{=} u'(\tau_2 \to \alpha)$$

ad i) $u'(\sigma_2(\tau_1)) = \tau_2' \to \tau_1' \overset{\text{2.(d), Prop. 4.3.3}}{\Leftrightarrow} u'(\sigma_2(\tau_1)) = \rho_1(\tau_1)$
$\Leftrightarrow \forall \beta \in FSV(\tau_1).u'(\sigma_2(\beta)) = \rho_1(\beta)$
$\overset{\text{Prop. A.0.1}}{\Leftrightarrow} true$

ad ii) $u'(\tau_2 \to \alpha) = \tau_2' \to \tau_1' \Leftrightarrow$ a) $u'(\tau_2) = \tau_2'$ and
b) $u'(\alpha) = \tau_1$

ad a) $u'(\tau_2) = \tau_2' \overset{\text{4.(d), Prop. 4.3.3}}{\Leftrightarrow} u'(\tau_2) = \rho_2(\tau_2)$
$\Leftrightarrow \forall \beta \in FSV(\tau_2).u'(\beta) = \rho_2(\beta)$
$\overset{\text{Prop. A.0.2}}{\Leftrightarrow} true$

ad b) By definition of u'.

Now it remains to show that there exists a substitution ρ such that:

i) $\forall \alpha \notin UV.\rho(\alpha) = \alpha$

ii) $\rho(\Delta) \subseteq \Delta'$

iii) $\sigma'(\Gamma) = \rho u \sigma_2 \sigma_1(\Gamma)$

iv) $D' \leq^{\rho} D$

By Def. 5.1 mgu computes a most general unifier u for $\sigma_1(\tau)$ and $\tau_2 \to \alpha$, i.e. there exists a substitution ρ' such that $u' = \rho' u$. Let $\rho = \rho'$:

ad i) By Def. 5.1 the result of mgu substitutes only sort variables from UV. By definition of u': $\forall \alpha \notin UV.u'(\alpha) = \alpha$. Because $u' = \rho' u$ it holds that $\forall \alpha \notin UV.\rho'(\alpha) = \alpha$.

ad ii) $\rho'(u\sigma_2(\Delta_1) \cup \Delta_2) \subseteq \Delta' \Leftrightarrow$ a) $\rho' u \sigma_2(\Delta_1) \subseteq \Delta'$ and
b) $\rho' u(\Delta_2) \subseteq \Delta'$

ad a) $\rho'(u(\sigma_2(\Delta_1)) \subseteq \Delta' \overset{u' = \rho' u}{\Leftrightarrow} u'(\sigma_2(\Delta_1) \subseteq \Delta'$
$\overset{\text{Prop. A.0.1, } *)}{\Leftrightarrow} \rho_1(\Delta_1) \subseteq \Delta'$
$\overset{\text{2.(b)}}{\Leftrightarrow} true$

ad b) $\rho' u(\Delta_2) \subseteq \Delta' \overset{u' = \rho' u}{\Leftrightarrow} u'(\Delta_2) \subseteq \Delta'$
$\overset{\text{Prop. A.0.2, } *)}{\Leftrightarrow} \rho_2(\Delta_2) \subseteq \Delta'$
$\overset{\text{4.(b)}}{\Leftrightarrow} true$

ad $*$) Δ_1, respectively Δ_2, contains all instantiated atomic sort formulae of the applied polymorphic functions. By Def. 3.1.10 all sort variables occurring in the qualifying sort predicates also occur in the body of the polymorphic sort term. Thus, it follows immediately that $FSV(\Delta_1) \subseteq FSV(D_1)$, respectively $FSV(\Delta_2) \subseteq FSV(D_2)$.

ad iii) $\sigma'(\Gamma) = \rho' u \sigma_2 \sigma_1(\Gamma) \quad \overset{u' \overset{=}{\Leftrightarrow} \rho' u}{\Leftrightarrow} \quad \sigma'(\Gamma) = u' \sigma_2 \sigma_1(\Gamma)$

$\overset{4.(c)}{\Leftrightarrow} \quad \rho_2 \sigma_2 \sigma_1(\Gamma) = u' \sigma_2 \sigma_1(\Gamma)$

$\Leftrightarrow \quad \forall \beta \in FSV(\sigma_2 \sigma_1(\Gamma)).\rho_2(\beta) = u'(\beta)$

By case analysis:

$\beta \in FSV(D_1) \setminus FSV(\sigma_2)$**:**

$$\rho_2(\beta) = \rho_1(\beta) \quad \overset{*)}{\underset{\beta \notin DOM(\sigma_2)}{\Leftrightarrow}} \quad \rho_2(\beta) = \rho_2 \sigma_2(\beta) \quad \overset{\beta \notin DOM(\sigma_2)}{\Leftrightarrow} \quad \rho_2(\beta) = \rho_2(\beta)$$

ad $*$)

$$\left.\begin{array}{l} \beta \in FSV(\sigma_2 \sigma_1(\Gamma)) \\ \text{and } \beta \notin FSV(\sigma_2) \end{array}\right\} \quad \Rightarrow \quad \beta \notin COD(\sigma_2)$$

$$\Rightarrow \quad \beta \in \sigma_1(\Gamma)$$

$$\overset{2.(c),\ 4.(c)}{\Rightarrow} \quad \forall \beta \in FSV(\sigma_1(\Gamma)).\rho_1(\beta) = \rho_2 \sigma_2(\beta)$$

$\beta \notin FSV(D_1) \setminus FSV(\sigma_2)$**:** $\rho_2(\beta) = \rho_2(\beta)$

ad iv)

$$\frac{D_1' \qquad D_2'}{\Delta', \sigma'(\Gamma) \rhd_\Sigma e_1 e_2 :: \tau_1'} (\to E) \quad \overset{\rho'}{=} \quad \frac{(u\sigma_2)_{u(\Delta_2)}(D_1) \;\; u_{u\sigma_2(\Delta_1)}(D_2)}{\Delta, u\sigma_2\sigma_1(\Gamma) \rhd_\Sigma e_1 e_2 :: u(\alpha)} (\to E)$$

Therefore, by Def. 4.3.2, 4.3.7, we have to show:

a) For each derivation node $\dfrac{}{\Delta', \Gamma' \rhd_\Sigma c :: \tau'}$ $(polyid)$ occurring in D_1' and its corresponding node $\dfrac{}{\Delta, \Gamma'' \rhd_\Sigma c :: u\sigma_2(\tau'')}$ $(polyid)$ in $(u\sigma_2)_{u(\Delta_2)}(D_1)$ holds:

$$(1) \quad \tau' = \rho' u \sigma_2(\tau'')$$

and for each derivat. node $\dfrac{D'}{\Delta', \Gamma' \rhd_\Sigma \lambda x.e_1' :: \tau_{11} \to \tau_{12}} (\to I_u)$ occurring in D_1' and its corresp. node $\dfrac{D''}{\Delta, \Gamma'' \rhd_\Sigma \lambda x.e_1' :: u\sigma_2(\tau_{21} \to \tau_{22})} (\to I_u)$ in $(u\sigma_2)_{u(\Delta_2)}(D_1)$ holds:

$$(2) \quad \tau_{11} = \rho' u \sigma_2(\tau_{21})$$

b) Like a) only for D_2' and $u_{u\sigma_2(\Delta_1)}(D_2)$.

ad a) ad (1)

$$\tau' = \rho' u \sigma_2(\tau'') \quad \overset{u' \overset{=}{\Leftrightarrow} \rho' u}{\Leftrightarrow} \quad \tau' = u' \sigma_2(\tau'')$$

$$\overset{2.(d)}{\Leftrightarrow} \quad \rho_1(\tau'') = u' \sigma_2(\tau'')$$

$$\Leftrightarrow \quad \forall \beta \in FSV(\tau'').\rho_1(\beta) = u' \sigma_2(\beta)$$

$$\overset{\text{Prop. A.0.1}}{\Leftrightarrow} \quad true$$

ad (2) similar to (1)

ad b) ad (1) $\tau' = \rho' u(\tau'')$ $\overset{u' = \rho' u}{\Leftrightarrow}$ $\tau' = u'(\tau'')$

$\overset{4.(d)}{\Leftrightarrow}$ $\rho_2(\tau'') = u'(\tau'')$

$\Leftrightarrow$ $\forall \beta \in FSV(\tau'').\rho_2(\beta) = u'(\beta)$

$\overset{\text{Prop. A.0.2}}{\Leftrightarrow}$ $true$

ad (2) similar to (1)

Proposition A.0.1 *Auxiliary Lemma*

We prove the following auxiliary lemma:

$$\forall \beta \in FSV(D_1).u'\sigma_2(\beta) = \rho_1(\beta)$$

Proof: By case analysis:

$\beta \notin FSV(\sigma_2)$: $u'\sigma_2(\beta) = \rho_1(\beta)$ $\overset{\beta \notin DOM(\sigma_2)}{\Leftrightarrow}$ $u'(\beta) = \rho_1(\beta)$

$\overset{\beta \notin FSV(\sigma_2), \beta \in FSV(D_1)}{\Leftrightarrow}$ $\rho_1(\beta) = \rho_1(\beta)$

$\beta \in FSV(\sigma_2)$: By case analysis:

$\beta \notin UV$: $u'\sigma_2(\beta) = \rho_1(\beta)$ $\overset{\text{Prop. 5.2.1, 2.(a)}}{\Leftrightarrow}$ $u'(\beta) = \beta$

$\overset{2.(a),4.(a)}{\Leftrightarrow}$ $\beta = \beta$

$\beta \in UV$: All unification variables occurring in $FSV(\sigma_2)$ are introduced by choosing somewhere a "new" sort variable. Because β already occurs in $FSV(D_1)$ it cannot be introduced in the call of $W(F, \sigma_1(\Gamma), e_2)$. Thus, β must have been passed by an argument. Because $\sigma_1(\Gamma)$ is the only argument containing unification variables β must be an element of $FSV(\sigma_1(\Gamma))$. By 2.(c):

$\sigma'(\Gamma) = \rho_1\sigma_1(\Gamma)$ $\overset{4.(c)}{\Leftrightarrow}$ $\rho_2\sigma_2\sigma_1(\Gamma) = \rho_1\sigma_1(\Gamma)$

$\Leftrightarrow$ $\forall \alpha \in FSV(\sigma_1(\Gamma)).\rho_2\sigma_2(\alpha) = \rho_1(\alpha)$

Thus: $u'\sigma_2(\beta) = \rho_1(\beta)$ $\Leftrightarrow$ $u'\sigma_2(\beta) = \rho_2\sigma_2(\beta)$

By case analysis:

$\beta \notin DOM(\sigma_2)$: $u'\sigma_2(\beta) = \rho_2\sigma_2(\beta)$ $\Leftrightarrow$ $u'(\beta) = \rho_2(\beta)$

$\overset{\beta \in FSV(\sigma_2)}{\Leftrightarrow}$ $\rho_2(\beta) = \rho_2(\beta)$

$\beta \in DOM(\sigma_2)$: $u'\sigma_2(\beta) = \rho_2\sigma_2(\beta)$ $\Leftrightarrow$ $\forall \gamma \in \sigma_2(\beta).u'(\gamma) = \rho_2(\gamma)$

$\overset{*)}{\Leftrightarrow}$ $\forall \gamma \in \sigma_2(\beta).\rho_2(\gamma) = \rho_2(\gamma)$

ad $*$) $\beta \in DOM(\sigma_2) \Rightarrow \gamma \in COD(\sigma_2) \Rightarrow \gamma \in FSV(\sigma_2)$

□

Proposition A.0.2 *Auxiliary Lemma*

We prove the following auxiliary lemma:

$$\forall \beta \in FSV(D_2).u'(\beta) = \rho_2(\beta)$$

Proof: By case analysis:

$\beta \in FSV(D_1) \setminus FSV(\sigma_2)$: $u'(\beta) = \rho_2(\beta) \overset{\text{Def. of } u'}{\Leftrightarrow} \rho_1(\beta) = \rho_2(\beta)$

By case analysis:

$\beta \notin UV$: $\rho_1(\beta) = \rho_2(\beta) \overset{2.(a),4.(a)}{\Leftrightarrow} \beta = \beta$

$\beta \in UV$**:** For the same reason as in Prop. A.0.1 β must be an element of $FSV(\sigma_1(\Gamma))$. By 2.(c):

$$\begin{aligned} \sigma'(\Gamma) = \rho_1\sigma_1(\Gamma) &\overset{4.(c)}{\Leftrightarrow} \rho_2\sigma_2\sigma_1(\Gamma) = \rho_1\sigma_1(\Gamma) \\ &\Leftrightarrow \forall \alpha \in FSV(\sigma_1(\Gamma)).\rho_2\sigma_2(\alpha) = \rho_1(\alpha) \end{aligned}$$

$$\text{Thus: } \rho_1(\beta) = \rho_2(\beta) \overset{\beta \notin DOM(\sigma_2)}{\Leftrightarrow} \begin{aligned} &\rho_2\sigma_2(\beta) = \rho_2(\beta) \\ &\rho_2(\beta) = \rho_2(\beta) \end{aligned}$$

$\beta \notin FSV(D_1) \setminus FSV(\sigma_2)$: $u'(\beta) = \rho_2(\beta) \overset{\text{Def. of } u'}{\Leftrightarrow} \rho_2(\beta) = \rho_2(\beta)$

□

$e = \lambda x.e_1$**:** We have a derivation D' of the form

$$\frac{D_1}{\Delta', \sigma'(\Gamma) \rhd_\Sigma \lambda x.e_1 :: \tau_2' \to \tau_1'} (\to I_u)$$

where $D_1 = \Delta', \sigma'(\Gamma).x{:}\tau_2' \bigtriangledown_\Sigma e_1 :: \tau_1'$. Let α be a new sort variable, and let $\sigma_1' = \sigma'.[\tau_2'/\alpha]$. Then $D_1 = \Delta', \sigma_1'(\Gamma.x{:}\alpha) \bigtriangledown_\Sigma e_1 :: \tau_1'$ and we can apply induction hypothesis yielding:

1. $W(F, \Gamma.x{:}\alpha, e_1) = (\Delta_1, \sigma_1, \tau_1, D_1)$
   where $D_1 = \Delta_1, \sigma_1(\Gamma.x{:}\alpha) \bigtriangledown_\Sigma e_1 :: \tau_1$
2. There exists a substitution ρ_1 such that:
   (a) $\forall \alpha \notin UV.\rho_1(\alpha) = \alpha$
   (b) $\rho_1(\Delta_1) \subseteq \Delta'$
   (c) $\sigma_1'(\Gamma.x{:}\alpha) = \rho_1\sigma_1(\Gamma.x{:}\alpha)$
   (d) $D_1' \leq^{\rho_1} D_1$

By Def. 5.2.1:

$$W(F,\Gamma,\lambda x.e_1) \;=\; (\Delta_1,\sigma_1,\sigma_1(\alpha)\to\tau_1,\ D)$$
$$\textbf{where}\quad D=\frac{D_1}{\Delta_1,\sigma_1(\Gamma)\rhd_\Sigma \lambda x.e_1::\sigma_1(\alpha)\to\tau_1}\ (\to I_u)$$

By 1. the recursive call of W terminates. Thus, W cannot fail in this case. It remains to show that there exists a substitution ρ such that:

i) $\forall\alpha\notin UV.\rho(\alpha)=\alpha$

ii) $\rho(\Delta_1)\subseteq\Delta'$

iii) $\sigma'(\Gamma)=\rho\sigma_1(\Gamma)$

iv) $D'\leq^\rho D$

Let $\rho=\rho_1$:

ad i) by 2.(a)

ad ii) by 2.(b)

ad iii) $\sigma'(\Gamma)=\rho_1\sigma_1(\Gamma)$

$$\begin{array}{lll} & \overset{\alpha\notin FSV(\Gamma)}{\Leftrightarrow} & \sigma'.[\tau_2'/\alpha](\Gamma)=\rho_1\sigma_1(\Gamma)\\ & \Leftrightarrow & \sigma_1'(\Gamma)=\rho_1\sigma_1(\Gamma)\\ & \overset{*)}{\Leftarrow} & \sigma_1'(\Gamma.x{:}\alpha)=\rho_1\sigma_1(\Gamma.x{:}\alpha)\\ & \overset{2.(c)}{\Leftrightarrow} & \mathit{true}\end{array}$$

ad $*$) $x\notin DOM(\Gamma)$ because we assumed that all variables in Γ are different from all λ-bound variables in e.

ad iv) By 2.(d) $D'\leq^{\rho_1} D$. Thus, by Def. 4.3.2 it remains to show:

$$\begin{array}{lll}\tau_2'=\rho_1\sigma_1(\alpha) & \overset{2.(c)}{\Leftrightarrow} & \tau_2'=\sigma_1'(\alpha)\\ & \Leftrightarrow & \tau_2'=\sigma'.[\tau_2'/\alpha](\alpha)\\ & \Leftrightarrow & \tau_2'=\tau_2'\end{array}$$

$e=\lambda x{:}\tau_2.e_1$**:** We have a derivation D' of the form

$$\frac{D_1'}{\Delta',\sigma'(\Gamma)\rhd_\Sigma \lambda x{:}\tau_2.e_1::\tau_2\to\tau_1'}\ (\to I_s)$$

where $D_1=\Delta',\sigma'(\Gamma).x{:}\tau_2\ \triangledown_\Sigma\ e_1::\tau_1'$. As we assumed that $\forall\alpha\notin UV.\sigma'(\alpha)=\alpha$ and because $FSV(\tau)\cap UV=\emptyset$ we know that $\sigma'(\Gamma).x{:}\tau_2=\sigma'(\Gamma.x{:}\tau_2)$. Thus, we can apply induction hypothesis yielding:

1. $W(F, \Gamma.x{:}\tau_2, e_1) \quad = \quad (\Delta_1, \sigma_1, \tau_1, D_1)$
   where $D_1 = \Delta_1, \sigma_1(\Gamma.x{:}\tau_2) \bigtriangledown_\Sigma e_1 :: \tau_1$
2. There exists a substitution ρ_1 such that:
   (a) $\forall \alpha \notin UV.\rho_1(\alpha) = \alpha$
   (b) $\rho_1(\Delta_1) \subseteq \Delta'$
   (c) $\sigma'(\Gamma.x{:}\tau_2) = \rho_1\sigma_1(\Gamma.x{:}\tau_2)$
   (d) $D_1' \leq^{\rho_1} D_1$

By Def. 5.2.1:

$$W(F, \Gamma, \lambda x{:}\tau_2.e_1) \quad = \quad (\Delta_1, \sigma_1, \tau_2 \to \tau_1,\ D)$$
$$\textbf{where} \quad D = \frac{D_1}{\Delta_1, \sigma_1(\Gamma) \rhd_\Sigma \lambda x{:}\tau_2.e_1 :: \tau_2 \to \tau_1}\ (\to I_s)$$

By 1. the recursive call of W terminates. Thus, W cannot fail in this case. It remains to show that there exists a substitution ρ such that:

i) $\forall \alpha \notin UV.\rho(\alpha) = \alpha$

ii) $\rho(\Delta_1) \subseteq \Delta'$

iii) $\sigma'(\Gamma) = \rho\sigma_1(\Gamma)$

iv) $D' \leq^\rho D$

Let $\rho = \rho_1$:

ad i) by 2.(a)

ad ii) by 2.(b)

ad iii) $\sigma'(\Gamma) = \rho_1\sigma_1(\Gamma) \overset{x \notin DOM(\Gamma)}{\Leftrightarrow} \sigma'(\Gamma.x{:}\tau_2) = \rho_1\sigma_1(\Gamma.x{:}\tau_2)$
$\overset{2.(c)}{\Leftrightarrow} \quad true$

ad iv) Follows directly from Def. 4.3.2 and 2.(d).

$e = e_1{:}\tau$**:** We have a derivation D' of the form

$$\frac{D_1'}{\Delta', \sigma'(\Gamma) \rhd_\Sigma e_1 : \tau :: \tau}\ (\textit{constrained})$$

where $D_1' = \Delta', \sigma'(\Gamma) \bigtriangledown_\Sigma e_1 :: \tau$. By applying induction hypothesis we know:

1. $W(F, \Gamma, e_1) \quad = \quad (\Delta_1, \sigma_1, \tau_1, D_1)$
   where $D_1 = \Delta_1, \sigma_1(\Gamma) \bigtriangledown_\Sigma e_1 :: \tau_1$

2. There exists a substitution ρ_1 such that:
   (a) $\forall \alpha \notin UV.\rho_1(\alpha) = \alpha$
   (b) $\rho_1(\Delta_1) \subseteq \Delta'$
   (c) $\sigma'(\Gamma) = \rho_1\sigma_1(\Gamma)$
   (d) $D'_1 \leq^{\rho_1} D_1$

By Def. 5.2.1:

$$W(F, \Gamma, e_1{:}\tau) \quad = \quad (u(\Delta_1), u\sigma_1, \tau, D)$$

$$\textbf{where} \quad u = mgu(\tau_1, \tau)$$

$$D = \frac{u_\emptyset(D_1)}{u(\Delta_1), u\sigma_1(\Gamma) \rhd_\Sigma e_1{:}\tau :: \tau} \; (constrained)$$

By 1. the recursive call of W terminates. The algorithm may, however, fail during unification of τ_1 and τ. To prove that W results in the tuple above we must show that τ_1 and τ are unifiable w.r.t. UV, i.e. that there exists a substitution u' such that $u'(\tau_1) = u'(\tau)$. Let $u' = \rho_1$:

$$\rho_1(\tau_1) \overset{2.(d),\text{Prop. } 4.3.3}{=} \tau \overset{*)}{=} \rho_1(\tau)$$

ad $*$) Because by 2.(a) $\forall \alpha \notin UV.\rho_1(\alpha) = \alpha$ and $FSV(\tau) \cap UV = \emptyset$.

Now it remains to show that there exists a substitution ρ such that:

i) $\forall \alpha \notin UV.\rho(\alpha) = \alpha$

ii) $\rho u(\Delta_1) \subseteq \Delta'$

iii) $\sigma'(\Gamma) = \rho u \sigma_1(\Gamma)$

iv) $D' \leq^{\rho} D$

By Def. 5.1 mgu computes a most general unifier u for τ_1 and τ, i.e. there exists a substitution ρ' such that $u' = \rho' u$. Let $\rho = \rho'$:

ad i) see case $e = e_1 e_2$

ad ii) $\rho' u(\Delta_1) \subseteq \Delta' \overset{u' = \rho' u,\, u' = \rho_1}{\Leftrightarrow} \rho_1(\Delta_1) \subseteq \Delta' \overset{2.(b)}{\Leftrightarrow} true$

ad iii) $\sigma'(\Gamma) = \rho' u \sigma_1(\Gamma) \overset{u' = \rho' u,\, u' = \rho_1}{\Leftrightarrow} \sigma'(\Gamma) = \rho_1\sigma_1(\Gamma) \overset{2.(c)}{\Leftrightarrow} true$

ad iv)

$$\frac{D_1'}{\Delta', \sigma'(\Gamma) \rhd_\Sigma e_1 : \tau :: \tau}\ (constr.) \quad \leq^{\rho'} \quad \frac{u_\emptyset(D_1)}{u(\Delta_1), u\sigma_1(\Gamma) \rhd_\Sigma e_1{:}\tau :: \tau}\ (constr.)$$

Therefore, by Def. 4.3.2 and 4.3.7, we must show:

a) For each derivation node $\frac{}{\Delta', \Gamma' \rhd_\Sigma c :: \tau'}\ (polyid)$ occurring in D_1' and its corresponding node $\frac{}{u(\Delta_1), \Gamma'' \rhd_\Sigma c :: u(\tau'')}\ (polyid)$ in $u_\emptyset(D_1)$ holds:

$$\tau' = \rho' u(\tau'') \quad \overset{u' = \rho' u,\, u' = \rho_1}{\Leftrightarrow} \quad \tau' = \rho_1(\tau'') \quad \overset{2.(d)}{\Leftrightarrow} \quad true$$

b) For each derivation node $\frac{D'}{\Delta', \Gamma' \rhd_\Sigma \lambda x.e_1' :: \tau_{11} \to \tau_{12}}\ (\to I_u)$ occurring in D_1' and its corresponding node $\frac{D''}{u(\Delta_1), \Gamma'' \rhd_\Sigma \lambda x.e_1' :: u(\tau_{21} \to \tau_{22})}\ (\to I_u)$ in $u_\emptyset(D_1)$ holds:

$$\tau_{11} = \rho' u(\tau_{21}) \quad \overset{u' = \rho' u,\, u' = \rho_1}{\Leftrightarrow} \quad \tau_{11} = \rho_1(\tau_{21}) \quad \overset{2.(d)}{\Leftrightarrow} \quad true$$

□

Proposition 5.2.5 *Sort inference algorithm U is sound w.r.t.* $\rhd'$

Let $\Sigma = ((\Omega, SA), P, F)$ be a polymorphic signature, and let qf be a qualified formula.

$$\text{If} \quad U(SA, P, F, qf) = (\Delta, D) \quad \text{then} \quad \Delta, \emptyset \ \rhd'_\Sigma qf$$

and D is a sort derivation ending with $\Delta, \emptyset \rhd'_\Sigma qf$.

Proof: We must show that D is a valid sort derivation for qf. By Prop. 5.2.3 D_1 is a sort derivation ending with $\Delta, \emptyset \rhd_\Sigma f$. Thus, it remains to prove the side condition of rule (*ext. qual.*), namely:

$$\begin{array}{l} \exists \sigma : \Phi \to \mathrm{T}_\Omega(\Phi) \text{ with} \\ \sigma(\alpha_i) = \alpha_i, 1 \leq i \leq n \text{ and} \\ \overline{A_m} \vdash_{SA} \sigma(\Delta) \end{array}$$

By Def. 5.2.3:

$$\sigma' = resolve(SA, \{\overline{A_m}\}, \Delta) \quad \overset{\text{Prop. 5.3.4}}{\Rightarrow} \quad (\Delta, \varepsilon) \vdash^{\mathrm{R}}_{SA, \overline{A_m}} (\emptyset, \sigma') \quad \overset{\text{Prop. 5.3.1}}{\Rightarrow} \quad \overline{A_m} \vdash_{SA} \sigma'(\Delta)$$

Clearly $\sigma'(\alpha_i) = \alpha_i, 1 \leq i \leq n$ because $\alpha_i \notin UV$ and σ' results from mgu calls. □

Proposition 5.2.6 *Sort inference algorithm U is complete w.r.t.* $\rhd'_\Sigma$

Let $\Sigma = ((\Omega, SA), P, F)$ be a polymorphic signature, and let qf be a qualified formula. If there exists a sort derivation $\Delta', \emptyset \bigtriangledown'_\Sigma qf$ then

1. $U(SA, P, F, qf) = (\Delta, (\Delta, \emptyset \bigtriangledown'_\Sigma qf))$ and
2. $\Delta', \emptyset \bigtriangledown'_\Sigma qf \leq \Delta, \emptyset \bigtriangledown'_\Sigma qf$

Proof: We have a derivation of the form

$$\frac{\Delta', \emptyset \bigtriangledown_\Sigma f}{\Delta', \emptyset \rhd'_\Sigma \forall\overline{\alpha_n}.\overline{A_m} \Rightarrow f} \; (ext.qual.)$$

ad 1. By Prop. 5.2.4:

$$V(P, F, \emptyset, f) = (\Delta, \sigma, (\Delta, \emptyset \rhd_\Sigma f))$$

and there exists a substitution ρ such that $\rho(\Delta) \subseteq \Delta'$. Therefore, it remains to show that $resolve(SA, \overline{A_m}, \Delta)$ terminates with some substitution σ''. By the side condition of rule (*ext. qual.*) there exists a substitution σ' such that:

$$\begin{array}{lll} \overline{A_m} \vdash_{SA} \sigma'(\Delta') & \overset{\rho(\Delta) \subseteq \Delta'}{\Rightarrow} & \overline{A_m} \vdash_{SA} \sigma'(\rho(\Delta)) \\ & \overset{\text{Prop. 5.3.1}}{\Rightarrow} & (\Delta, \varepsilon) \vdash^{\mathrm{R}}_{SA, \overline{A_m}} (\emptyset, \sigma') \\ & \overset{\text{Prop. 5.3.4}}{\Rightarrow} & resolve(SA, \overline{A_m}, \Delta) = \sigma'' \end{array}$$

ad 2. $\Delta', \emptyset \bigtriangledown'_\Sigma qf \;\leq\; \Delta, \emptyset \bigtriangledown'_\Sigma qf \overset{\text{Def. 4.3.13}}{\Leftrightarrow} \exists \sigma : \Phi \to \mathrm{T}_\Omega(\Phi)$ with

(a) $\sigma(\alpha_i) = \alpha_i, 1 \leq i \leq n$

(b) $\Delta' \vdash_{SA} \sigma(\Delta)$

(c) $\Delta', \emptyset \bigtriangledown'_\Sigma qf \;\leq^\rho\; \Delta, \emptyset \bigtriangledown'_\Sigma qf \overset{\text{Def. 4.3.13}}{\Leftrightarrow} \Delta', \emptyset \bigtriangledown_\Sigma f \;\leq^\sigma\; \Delta, \emptyset \bigtriangledown_\Sigma f$

By Prop. 5.2.4 there exists a substitution ρ such that

(d) $\forall \alpha \notin UV.\rho(\alpha) = \alpha$

(e) $\rho(\Delta) \subseteq \Delta'$

(f) $\Delta', \emptyset \bigtriangledown_\Sigma f \;\leq^\rho\; \Delta, \emptyset \bigtriangledown_\Sigma f$

Let $\sigma = \rho$:

ad (a) By (d)

ad (b) By (e) because of monotonicity of $\vdash$ (Prop. 3.1.1)

ad (c) By (f)

□

Proposition 5.4.1 *Algorithm analyze_ax is sound and complete*

Let $\Sigma = ((\Omega, SA), P, F)$ be a polymorphic signature and $f \in \mathrm{PF}_\Sigma(\Phi) \cup \mathrm{QF}_\Sigma$ be an arbitrary formula.

1. Algorithm *analyze_ax* is sound, i.e.

$$\mathit{analyze\_ax}(SA, P, F, f) = \mathit{well\text{-}formed} \;\Rightarrow\; f \in \mathrm{SEN}_\Sigma$$

2. Algorithm *analyze_ax* is complete, i.e.

$$f \in \mathrm{SEN}_\Sigma \;\Rightarrow\; \mathit{analyze\_ax}(SA, P, F, f) = \mathit{well\text{-}formed}$$

Proof:

1.
   - Let f be a non-qualified formula
     By Def. 5.4.1 *analyze_ax* yields *well-formed* if $V(P, F, \emptyset, f) = (\Delta, \sigma, D)$ and $\mathit{resolve}(SA, \emptyset, \Delta \cap \mathrm{AF}_\Omega(\emptyset)) = \sigma'$. We have to prove:

$$f \in \mathrm{SEN}_\Sigma \overset{\text{Def. 3.2.13}}{\Leftrightarrow} \Delta, \emptyset \rhd_\Sigma f \text{ for some } \Delta \text{ with } \emptyset \vdash_{SA} \Delta \cap \mathrm{AF}_\Omega(\emptyset)$$

     By Prop. 5.2.3 $D = \Delta, \emptyset \bigtriangledown_\Sigma f$ is a sort derivation for f. Thus, it remains to prove:

$$\begin{array}{lll} \emptyset \vdash_{SA} \Delta \cap \mathrm{AF}_\Omega(\emptyset) & \Leftrightarrow & \emptyset \vdash_{SA} \sigma'(\Delta \cap \mathrm{AF}_\Omega(\emptyset)) \\ & \overset{\text{Prop. 5.3.1}}{\Leftarrow} & (\Delta \cap \mathrm{AF}_\Omega(\emptyset), \varepsilon) \vdash^{\mathrm{R}}_{SA,\emptyset} (\emptyset, \sigma') \\ & \overset{\text{Prop. 5.3.4}}{\Leftarrow} & \mathit{resolve}(\mathrm{SA}, \emptyset, \Delta \cap \mathrm{AF}_\Omega(\emptyset)) = \sigma' \end{array}$$

   - Let f be a qualified formula
     Follows directly from Prop. 5.2.5.

2.
   - Let f be a non-qualified formula.
     If $f \in \mathrm{SEN}_\Sigma$ then by Def. 3.2.13 $\Delta', \emptyset \rhd_\Sigma f$ for some Δ' with $\emptyset \vdash_{SA} \Delta' \cap \mathrm{AF}_\Omega(\emptyset)$. From Prop. 5.2.4 follows that $V(P, F, \emptyset, f) = (\Delta, \sigma, D)$ and that there exists a substitution ρ with $\rho(\Delta) \subseteq \Delta'$. Thus, by Def. 5.4.1 it remains to prove that $\mathit{resolve}(SA, \emptyset, \Delta \cap \mathrm{AF}_\Omega(\emptyset))$ terminates with some substitution σ'':

$$
\begin{array}{lll}
\emptyset \vdash_{SA} \Delta' \cap \mathrm{AF}_\Omega(\emptyset) & \overset{\rho(\Delta) \subseteq \Delta'}{\Rightarrow} & \emptyset \vdash_{SA} \rho(\Delta) \cap \mathrm{AF}_\Omega(\emptyset) \\
& \Rightarrow & \emptyset \vdash_{SA} \Delta \cap \mathrm{AF}_\Omega(\emptyset) \\
& \Leftrightarrow & \emptyset \vdash_{SA} \sigma'(\Delta \cap \mathrm{AF}_\Omega(\emptyset)) \\
& \overset{\text{Prop. 5.3.1}}{\Rightarrow} & (\Delta \cap \mathrm{AF}_\Omega(\emptyset), \varepsilon) \vdash^{\mathrm{R}}_{SA,\emptyset} (\emptyset, \sigma') \\
& \overset{\text{Prop. 5.3.4}}{\Rightarrow} & \mathit{resolve}(SA, \emptyset, \Delta \cap \mathrm{AF}_\Omega(\emptyset)) = \sigma''
\end{array}
$$

- Let f be a qualified formula.
  If $f \in \mathrm{SEN}_\Sigma$ then by Def. 3.2.13 $\Delta', \emptyset \rhd_\Sigma f$ for some Δ'. Thus, by Prop. 5.2.6 $U(SA, P, F, f) = (\Delta, D)$ and *analyze_ax* terminates with *well-formed*.

□

Proposition 5.5.1 *V computes a principal sort derivation*

Let $f \in \mathrm{PF}_\Sigma(\Phi)$ be a closed formula. If there exists a sort derivation for f then there exists a principal sort derivation for f and $V(P, F, \emptyset, f) = (\Delta, \sigma, D)$ where D is a principal sort derivation for f.

Proof: We show that

1. $V(P, F, \emptyset, f)$ terminates with (Δ, σ, D) where D is a sort derivation for f and
2. by Def. 4.3.4 that for each sort derivation D' holds $D' \leq D$.

ad 1. If f is a closed well-formed formula then by Def. 3.2.13 there exists a sort derivation $\Delta', \emptyset \bigtriangledown_\Sigma f$ with $\emptyset \vdash_{SA} \Delta' \cap \mathrm{AF}_\Omega(\emptyset)$. Thus, by Prop. 5.2.4 $V(P, F, \emptyset, f)$ terminates with $(\Delta, \sigma, (\Delta, \emptyset \bigtriangledown_\Sigma f))$ and by Prop. 5.2.3 $\Delta, \emptyset \bigtriangledown_\Sigma f$ is a sort derivation for f.

ad 2. We have to show $\Delta', \emptyset \bigtriangledown_\Sigma f \leq \Delta, \emptyset \bigtriangledown_\Sigma f \overset{\text{Def. 4.3.3}}{\Leftrightarrow} \exists \rho : \Phi \to \mathrm{T}_\Omega(\Phi)$ with

   (a) $\forall \alpha \in FSV(f).\ \rho(\alpha) = \alpha$
   (b) $\Delta' \vdash_{SA} \rho(\Delta)$
   (c) $\Delta', \emptyset \bigtriangledown_\Sigma f \leq^\rho \Delta, \emptyset \bigtriangledown_\Sigma f$

   By Prop. 5.2.4 we know that there exists a substitution ρ' such that

   (d) $\forall \alpha \notin UV.\rho'(\alpha) = \alpha$
   (e) $\rho'(\Delta) \subseteq \Delta'$
   (f) $\Delta', \emptyset \bigtriangledown_\Sigma f \leq^{\rho'} \Delta, \emptyset \bigtriangledown_\Sigma f$

Let $\rho = \rho'$:

ad (a) Follows immediately from (d)

ad (b) Follows from (e) by monotonicity of $\vdash$ (Prop. 3.1.1)

ad (c) Follows immediately from (f)

□

Proposition 5.5.2 *U computes a principal sort derivation*

Let $qf \in \mathrm{QF}_\Sigma$ be a closed qualified formula. If there exists an extended sort derivation for qf then there exists a principal extended sort derivation for qf and $U(SA, P, F, qf) = (\Delta, D)$ where D is a principal extended sort derivation for qf.

Proof: We show that

1. $U(SA, P, F, qf)$ terminates with (Δ, D) where D is a sort derivation for qf and
2. by Def. 4.3.14 that for each sort derivation D' holds $D' \leq D$.

ad 1. If qf is a closed well-formed formula then by Def. 3.2.13 and Prop. 4.3.11 there exists an extended sort derivation $\Delta', \emptyset \bigtriangledown'_\Sigma qf$. Thus, by Prop. 5.2.6 $U(SA, P, F, qf)$ terminates with $(\Delta, (\Delta, \emptyset \bigtriangledown'_\Sigma qf))$ and by Prop. 5.2.5 $(\Delta, \emptyset \bigtriangledown'_\Sigma qf)$ is a sort derivation for qf.

ad 2. Follows directly from Prop. 5.2.6.

□

Appendix B

The Concrete Syntax of PolySpec

B.1 EBNF-Notation

The concrete syntax of PolySpec is presented as an EBNF-like grammar. The notations used are summed up below:

$[rhs]$	*rhs* is optional.
$\{rhs\}^*$	zero or more repetitions of *rhs*
$\{rhs\ //,\}^*$	zero or more repetitions of *rhs* separated by ,
$\{rhs\}^+$	one or more repetitions of *rhs*
$\{rhs\ //,\}^+$	one or more repetitions of *rhs* separated by ,
$rhs_1 \mid rhs_2$	choice
$rhs_{\{\overline{rhs}\}}$	difference: elements generated by *rhs* except those generated by $\overline{rhs}$
terminal	terminal symbols are given in boldface
<nonterminal>	nonterminals are enclosed in angle brackets
<*nonterminal*>	emphasized nonterminals are not defined in the grammar but represent a non-printable letter of the ASCII character set or are given informally

B.2 Lexical Syntax

<spectext> ::= {<lexeme> | <whitespace> | <comment>}*

<lexeme> ::= <special> | <reserved> | <alphanumid> | <num>

Whitespace

⟨whitespace⟩ ::= ⟨*space*⟩ | ⟨*tab*⟩ | ⟨line-delim⟩

⟨line-delim⟩ ::= ⟨*newline*⟩ | ⟨*carriage return*⟩ | ⟨*vtab*⟩ | ⟨*form feed*⟩

Comments

⟨comment⟩ ::= % {⟨any⟩}* ⟨line-delim⟩

Syntactic Categories

⟨letter⟩ ::= **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m**
| **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**
| **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M**
| **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**
| $\boldsymbol{\alpha}$ | $\boldsymbol{\beta}$ | $\boldsymbol{\gamma}$ | $\boldsymbol{\delta}$ | $\boldsymbol{\varepsilon}$ | $\boldsymbol{\zeta}$ | $\boldsymbol{\eta}$ | $\boldsymbol{\vartheta}$ | $\boldsymbol{\iota}$ | $\boldsymbol{\kappa}$ | $\boldsymbol{\mu}$
| $\boldsymbol{\nu}$ | $\boldsymbol{\xi}$ | $\boldsymbol{\pi}$ | $\boldsymbol{\rho}$ | $\boldsymbol{\sigma}$ | $\boldsymbol{\tau}$ | $\boldsymbol{\upsilon}$ | $\boldsymbol{\varphi}$ | $\boldsymbol{\chi}$ | $\boldsymbol{\psi}$ | $\boldsymbol{\omega}$

⟨digit⟩ ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

⟨symbol⟩ ::= _ | ’ | + | − | ∗ | / | < | > | ≠
| # | ⊑ | **$** | & | **!** | **?** | **@** | ⊥

⟨special⟩ ::= (|) | ; | , | : | . | **Π** | → | ×
| **λ** | = | ∀ | ∃ | ¬ | ∧ | ∨ | ⇒ | ⇔

⟨alphanum⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨symbol⟩

⟨any⟩ ::= ⟨alphanum⟩ | ⟨special⟩ | ⟨*space*⟩ | ⟨*tab*⟩

Natural Numbers

⟨num⟩ ::= {⟨digit⟩}+

Identifiers

⟨alphanumid⟩ ::= {⟨alphanum⟩}+{⟨reserved⟩ | ⟨num⟩}

Reserved Words

⟨reserved⟩ ::= **cons** | **sortpred** | **pred** | **fun**
| **sortaxioms** | **axioms** | **in** | **endaxioms**

B.3 Context Free Syntax

⟨spec⟩ ::= {⟨sspec⟩ | ⟨ospec⟩}*

Sort Specifications

<sspec>	::=	<conslist>	
	|	<spredlist>	
	|	<saxioms>	
<conslist>	::=	**cons** {<cons> ;}$^+$	
<cons>	::=	<id> [: <num>]	
<spredlist>	::=	**sortpred** {<spred> ;}$^+$	
<spred>	::=	<id> [: <num>]	
<saxioms>	::=	**sortaxioms** [<svarlist>] **in** {<clause> ;}$^+$ **endaxioms**	
<svarlist>	::=	{<id> //,}$^+$	
<clause>	::=	[<atoms> $\Rightarrow$] <atom>	
<atoms>	::=	{<atom> //,}$^+$	
<atom>	::=	<id> ({<sortterm> //,}$^+$)	

Sort Terms

<asortterm>	::=	<id>	*(Basic Sort, Sort Variable)*
	|	<id> ({<sortterm> //,}$^+$)	*(Applied Sort Constructor)*
	|	(<sortterm>)	*(Grouping)*
<sortterm>	::=	<asortterm>	
	|	<sortterm> $\times$ <sortterm>	*(Product Sort)*
	|	<sortterm> $\longrightarrow$ <sortterm>	*(Functional Sort)*

Object Specifications

<ospec>	::=	<predlist>	
	|	<funlist>	
	|	<axioms>	
<predlist>	::=	**pred** {<id> : <functionality> ;}$^+$	
<funlist>	::=	**fun** {<id> : <functionality> ;}$^+$	
<functionality>	::=	<sortterm>	
	|	**Π** <svarlist> . [<atoms> $\Rightarrow$] <sortterm>	
<axioms>	::=	**axioms** <varlist> **in** {<formula> ;}$^+$ **endaxioms**	
<varlist>	::=	[<context>] {<var> //,}*	
<context>	::=	{<atom> //,}* $\Rightarrow$	
<var>	::=	<id> [: <sortterm>]	

Formulae

<formula>	::=	<id>	*(Constant Predicate)*
	|	<id> <term>	*(Applied Predicate)*
	|	(<formula>)	*(Grouping)*
	|	<term> = <term>	*(Strong Equality)*
	|	¬ <formula>	*(Negation)*
	|	<formula> ∧ <formula>	*(Conjunction)*
	|	<formula> ∨ <formula>	*(Disjunction)*
	|	<formula> ⇒ <formula>	*(Implication)*
	|	<formula> ⇔ <formula>	*(Equivalence)*
	|	∀ <var> . <formula>	*(Universal Quantification)*
	|	∃ <var> . <formula>	*(Existential Quantification)*

Terms

<term>	::=	<id>	
	|	(<tuple>)	*(Tuple)*
	|	(<term>)	*(Grouping)*
	|	<term> : <asortterm>	*(Sort Annotation)*
	|	<term> <term>	*(Application)*
	|	**λ** <var> . <term>	*(λ-Abstraction)*
<tuple>	::=	<term> , <term>	
	|	<term> , <tuple>	

Identifiers

<id>	::=	<alphanumid>	
	|	<num>	

B.4 Priority of Operators

Fig. B.1 shows the priority of the operators of PolySpec. Operators with higher priorities bind stronger. In the last column the associativity of operators with the same priority is shown.

Priority	Operator	Description	Associativity
12	$:$	Sort annotation for term	left
11	$\times$	Carthesian product	right
10	$\longrightarrow$	Function space constructor	right
9		Prefix-Application	left
8	$\boldsymbol{\lambda} x.$	λ-Abstraction	
7	$=$	Identity	
6	$\neg$	Negation	
5	$\wedge$	Conjunction	right
4	$\vee$	Disjunction	right
3	$\Rightarrow$	Implication	right
2	$\Leftrightarrow$	Equivalence	right
1	$\forall x.$ $\exists x.$	Quantifiers	

Figure B.1: Priority of Operators

Bibliography

[BFG+92] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, and K. Stølen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 0.3. Technical Report TUM-I9140, Technische Universität München, 1992.

[BFG+93a] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, May 1993.

[BFG+93b] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part II. Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, May 1993.

[BG80] R.M. Burstall and J.A. Goguen. The semantics of CLEAR, a specification language. In *Proceedings of Advanced Course on Software Specification*, volume 86 of *LNCS*, pages 292–332, 1980.

[Blo92] S. Blott. *An approach to overloading with polymorphism.* PhD thesis, Dept. of Computing Science, University of Glasgow, 1992.

[Bre91] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*, volume 562 of *LNCS*. Springer, 1991.

[BTCGS89] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Proc. of the IEEE Symposium on Logic in Computer Science*, pages 112–129, June 1989.

[BW93] M. Broy and M. Wirsing. Korrekte Software: Vom Experiment zur Anwendung. In Horst Reichel, editor, *GI Informatik aktuell. Informatik, Wirtschaft, Gesellschaft*, pages 29–43. Springer-Verlag, 1993.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In *Proc. Semantics of Data Types*, pages 51–68. Springer, 1984.

[Che95] K. Chen. *A Parametric Extension of Haskell's Type Classes.* PhD thesis, Yale University, 1995.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[DM82] L. Damas and R. Milner. Principle Type-Schemes for Functional Programs. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constrains.* Springer, 1990.

[FM88] Y.-C. Fuh and P. Mishra. Type Inference with Subtypes. In *ESOP 88*, pages 94–114. Springer Verlag, 1988.

[FM89] Y.-C. Fuh and P. Mishra. Polymorphic Subtype Inference: Closing the Theory–Practice Gap. In *TAPSOFT–89*, March 1989.

[FM90] Y.-C. Fuh and P. Mishra. Type Inference with Subtypes. *Theoretical Computer Science*, 73:155–175, 1990.

[Gam94] J. Gambihler. Syntaktische Analyse polymorpher Spezifikationen. Master's thesis, Technische Universität München, 1994.

[Gir72] J.-Y. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These D'Etat, Universite Paris VII, 1972.

[GM87] J.A. Goguen and J. Meseguer. Order–Sorted Algebra Solves the Constructor–Selector, Multiple Representation and Coercion Problems. In *Logic in Computer Science*, IEEE, 1987.

[GN94] R. Grosu and D. Nazareth. Towards a New Way of Parameterization. In *Proceedings of the Third Maghrebian Conference on Software Engineering and Artificial Intelligence*, pages 383–392, 1994.

[Gog78] J.A. Goguen. Order Sorted Algebras: Exception and Error Sorts, Coercions and Overloaded Operators. *Semantics and Theory of Computation*, 14, 1978. University of California, Los Angeles.

[Gog84] M. Gogolla. Partially ordered sorts in algebraic specifications. In *Colloq. on Trees in Algebra and Programming*, pages 139–153. Bourdeaux, Cambridge University Press, 1984.

[Gor85] M.C.J. Gordon. HOL — a machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.

[GR94] Radu Grosu and Franz Regensburger. The Logical Framework of SPECTRUM. Technical Report TUM-I9402, Institut für Informatik, Technische-Universität München, 1994.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[GW88] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab., 1988.

[Han89] M. Hanus. Horn clause specifications with polymorphic types. Technical Report 294, FB Informatik, Universität Dortmund, 1989. Dissertation.

[Han91] M. Hanus. Parametric Order-Sorted Types in Logic Programming. Technical Report 377, Universität Dortmund Fachbereich Informatik, January 1991.

[Har86] R. Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, University of Edinburgh,Department of Computer Science, November 1986.

[HB89] K. Hammond and S. Blott. Implementing Haskell type classes. In *Proc. of the 1989 Glasgow Workshop on Functional Programming*. Fraserburgh, Scotland, Springer-Verlag, 1989.

[Hin69] R. Hindley. The Principle Type-Scheme of an Object in Combinatory Logic. *Trans. Am. Math. Soc.*, 146:29–60, December 1969.

[HJW92] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)*. ACM SIGPLAN Notices, May 1992.

[HLR93] Heinrich Hußmann, Jacques Loeckx, and Wolfgang Reif. Korrekte Software - Das Projekt KORSO. Tagungsband GI-Jahrestagung 1993, 1993.

[HT90] P. Hill and R. Topor. A semantics for typed logic programs. Technical Report TR-90-11, Comuter Science Department, University of Bristol, 1990.

[Jon90] M. P. Jones. Computing with lattices: An application of type classes. Technical Report PRG-TR-11-90, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, June 1990.

[Jon91a] M. P. Jones. Towards a theory of qualified types. Technical Report PRG-TR-6-91, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, April 1991.

[Jon91b] M. P. Jones. Type Inference For Qualified Types. Technical Report PRG-TR-10-91, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, 1991.

[Jon92] M. P. Jones. Qualified Types: Theory and Practice. Technical Monograph PRG-106, Oxford University Computing Laboratory, Programming Research Group, July 1992.

[Jon93a] M. P. Jones. *An Introduction to Gofer*, August 1993.

[Jon93b] M.P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proc. of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 43–52. ACM Press, 1993.

[Kae88] S. Kaes. Parametric Overloading in Polymorphic Programming Languages. In *ESOP 88*, pages 131–144, 1988.

[Kae92] S. Kaes. Type Inference in the Presence of Overloading, Subtyping and Recursive Types. In *Proc. ACM Conf. Lisp and Functional Programming*, 1992.

[KT90] A. Kfoury and J. Tiuryn. Type reconstruction in finite-rank fragments of the polymorphic λ-calculus. In *Proc. of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 2–11, June 1990.

[Läu94] K. Läufer. Combining type classes and existential types. In *Latin American Informatics Conference (PANEL), ITESM-CEM, Mexico*, 1994.

[Llo87] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mit84] John C. Mitchell. Coercion and type inference. In *Proc. 11th ACM Symp. Principles of Programming Languages*, pages 175–185, 1984.

[Mit90] J.C. Mitchell. Type Systems for Programming Languages. In *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. Elsevier Science Publisher, 1990.

[Mit91] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1:245–285, 1991.

[Mos91] P.D. Mosses. The Use of Sorts in Algebraic Specifications. In *Proceedings of the 8th Workshop on Abstract Data Types, Dourdan*, December 1991.

[Naz92] D. Nazareth. Subsorting and Polymorphism in the Specification Language SPECTRUM. In W.-M. Lippe and G. Stroot, editors, *Programmiersprachen – Methoden, Semantik, Implementierungen, Landhaus Rothenberge*, pages 187–204. WWU Münster, Angewandte Mathematik und Informatik, 1992.

[Naz93a] D. Nazareth. Modelling Inheritance in an Algebraic Specification Language. In Jianping Wu et al., editor, *Proceedings of the Third International Conference for Young Computer Scientists, Beijing*, pages 9.05–9.08. Tsinghua University Press, July 1993.

[Naz93b] D. Nazareth. A Universally Polymorphic Specification Language – A Brief Informal Introduction. In Rudolf Berghammer and Gunther Schmidt, editors, *Programmiersprachen und Grundlagen der Programmierung*, pages 140–155. Universität der Bundeswehr, Fakultät für Informatik, 1993.

[NP93] T. Nipkow and C. Prehofer. Type Checking Type Classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 409–418. ACM Press, 1993.

[NS91] Tobias Nipkow and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In J. Hughes, editor, *Proc. of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 1–14. Springer Verlag, 1991.

[Pad89] P. Padawitz. Computing in Horn Clause Theories. In *EATCS Monographs in Computer Science*. Springer, Berlin, 1989.

[Pau92] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1992.

[Pau93] L. C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[Pit87] A. Pitts. Polymorphism is set-theoretic. In D. H. Pitt, A. Poigné, and D. Rydeheard, editors, *Category Theory and Computer Science*, volume 283 of *Lecture Notes in Computer Science*, pages 19–39. Springer-Verlag, 1987.

[Poi84] A. Poigné. Another look at parameterization using algebraic specifications with subsorts. In *MFCS'84, Proc. Symp. on Math. Foundations of Computer Science*, volume 176 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[Poi86] A. Poigné. On Specifications, Theories and Models with Higher Order Types. *Information and Control*, 68:1–46, 1986.

[Qia91] Z. Qian. *Extensions of Order-Sorted Algebraic Specifications: Parameterization, Higher-Order Functions and Polymorphism*. PhD thesis, Universität Bremen, 1991.

[Reg94] Franz Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technische Universität München, 1994.

[Rey74] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Fraserburgh, Scotland, Springer-Verlag, 1974.

[Rey84] J. C. Reynolds. Polymorphism is not set-theoretic. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer-Verlag, 1984.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Smo89] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types.* PhD thesis, Universität Kaiserslautern, 1989.

[Sok89] S. Sokolowski. Applicative High-Order Programming or Standard ML in the Battlefield. Technical Report MIP 8908, Universität Passau, Lehrstuhl für Informatik, February 1989.

[Str67] C. Strachey. Fundamental Concepts in Programming Languages. In *Lecture Notes for International Summer School in Computer Programming*, Copenhagen, 1967.

[SW83] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. Technical Report CSR-131-83, University of Edinburgh, Edinburgh EH9 3JZ, September 1983.

[Tha91] S. Thatte. Coercive type isomorphism. In J. Hughes, editor, *5th ACM Conference on Functional Programming Languages and Computer Architektur*, volume 523 of *Lecture Notes in Computer Science*, pages 29–48. Cambridge, MA, Springer-Verlag, 1991.

[Wad89] P. Wadler. Theorems for free! In *Proc. of the 4th ACM Conference on Functional Programming Languages and Computer Architecture, London, England*, pages 347–359. ACM Press, September 1989.

[WB89] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.