Contents

Chapter 1

Introduction

1.1 Motivation

Many programming languages – including the most common languages C++, Pascal, and Ada – use the concept of types. That means that each variable and each argument in a subprogram is associated with a type which determines the form of the possible values.

There are mainly three reasons for the use of types ([PS94]).

- They make programs more readable and provide information on the intentions of the programmer.
- They make programs more reliable, because they make sure which kind of information is passed around or stored in a place. Offenses are discovered. This helps discovering programming errors where otherwise the program would compute a wrong result.
- They make programs more efficient, because the knowledge of types enables optimizations and helps to avoid runtime checks.

In order to make use of the information provided with types and to make programs more reliable, it is necessary to check whether there are any offenses against the provided type information. This might be done at compile time before execution of the program or at runtime during execution. In the compile time approach, a type–checker computes the types occurring in the expressions and checks for illegal assignments. This will increase the compile time, but no runtime checks are necessary anymore. For the runtime approach it is necessary to make sure that the type constraints are actually checked at runtime. This increases the runtime compared to the compile time approach, but type-checking is not necessary at compile time, and therefore our compiler runs faster. According to the approach we choose, there is obviously a trade–off between runtime and compile time.

For "Hoopla" a compile time solution was chosen. The benefits of the approach chosen for "Hoopla" are obvious. Runtime checks to determine whether a value has a correct type are needless anymore, except for a few special cases. Therefore, the program is much faster than without static resolution.

This thesis deals with the development and design of the type–checker for the new object–oriented programming language "Hoopla". A characteristic idea of object–oriented languages is that different types together implement a class and an instance of this class might be an instance of any of the underlying types. If subprograms are called with classes as arguments, it is necessary to determine the actual type of the object at runtime and invoke the suitable subprogram. In general this process, called dispatching, increases the difficulties to determine all types of expressions at compile time. "Hoopla" is designed in such a way that this resolution is still possible at compile time. Therefore, the type–checker is a major part of "Hoopla"s front–end.

The language "Hoopla" is still under development and the design might change again. The type–checker developed in this thesis is a part of the compiler prototype which is designed under Bernd Holzmüller at the University of Stuttgart.

1.2 Description of the task

Task

An existing front–end for a subset of the programming language "Hoopla" – called "Hoopla_Light" – has to be extended for the full version of "Hoopla" using several tools. This includes

1. small modifications concerning the concrete and abstract syntax and the construction of the abstract syntax tree, and
2. semantic checks concerning the type–system (type conformity, overloading, dispatching). The front–end needs to be tested with a series of test programs.

Tools

The compiler construction tools "Cocktail" from GMD Karlsruhe must be used. Implementation language is Ada.

1.3 Outline

Chapter 2 gives a very short and general survey on the requirements for type–checking object–oriented type–systems. The next chapter presents the main concepts of the programming language "Hoopla" with a special emphasis on how the type–checking related

issues are solved in the language design. Chapter 4 contains the main part of the thesis. Here, the necessary data structures, the type–checking algorithm, and the underlying ideas are presented. Finally, Chapter 5 provides the reader with some details on the implementation of the type–checker and the tests done so far before Chapter 6 concludes.

Appendix A gives an overview on the Vienna development method (VDM) used to describe the algorithms and data structures. Appendix B contains the structure of the abstract syntax tree (AST) of "Hoopla" described in VDM. All the algorithms are described on the abstract syntax tree. Appendix C presents a few basic functions that serve as interface between the AST and the internal data structures. And finally, there is an index of the defined functions in appendix D.

1.4 Acknowledgements

I would like to thank Prof. Dr. Erhard Plödereder and Bernd Holzmüller for making this diploma thesis possible during my stay at the University of Massachusetts, Amherst. Furthermore, I thank my American advisors Prof. Eliot Moss and Kathryn McKinley, and my fellow students Brendon Cahoon, Eric Wright, and Glen Weaver for their resonance. And last I want to thank Nicole Weicker for her love and support.

Chapter 2

Checking object–oriented type–systems

This chapter contains a short description on the purpose of type–checkers, where we put a special emphasize on object–oriented concepts and their impact on type checking. The concepts are described, and for each concept we give an overview of the state of the art in a short survey on common object–oriented languages. In addition, we discuss the role of Hoopla. All the examples given in this chapter are written in pseudo code and have nothing in common with the syntax of "Hoopla".

2.1 Type checking

As the motivation in chapter 1 points out, types are considered useful in order to make programs more readable, more reliable, and more efficient. When we associate a variable with a certain type, this association may be considered as a constraint on the values which actually may be assigned to the variable. Therefore, it is necessary to ensure that only those values are assigned at runtime.

```
I : Integer;
R : Real;
...
I <-- R;
```

For example, the assignment above is not safe, because `R` might hold values that are not assignable to `I`.

Similar constraints occur in control structs; e.g. if an expression serves as a condition, it must be a boolean expression. And also the arguments for subprogram calls must have correct types. As already discussed in the introduction, there are two fundamentally

different possibilities to deal with those assignment–like constraints. First, we can choose to determine at compile time the type of each expression as a part of the static analysis, and refuse illegal assignments, conditions, and subprogram calls. Then, we do not have to worry about appropriate types at runtime anymore. A second approach is to ignore it at compile time, and check the actual values at runtime whether they are appropriate or not. If they have the wrong type we raise an error message and finish the execution of the program. Since programs are desired to be reliable, the static type–checking is preferred.

One very important fact related to the type–checking of assignments and expressions is, that with static type–checking there might be programs which are correct in terms of execution at runtime, but which are not accepted by a static type–checker. The reason for this behavior is that some statements might never be executed because they are not reachable in the program. An example is

```
I : Integer;
I <-- 0;
if I=0 then I <-- 1; else I <-- 1.1; end;
```

Optimizations, like non–reachable code elimination, could deal with the example above, but this is not a general rule to avoid rejection of correct programs. Static type–correctness implies correct assignments at runtime. The reverse direction is wrong.

In traditional languages like Pascal, type checking is a fairly simple task since a name is only bound to one subprogram or variable in each scope. Such a type–checker is described for example in [ASU86].

Usually, there is no overloading of subprogram names in traditional, imperative languages, i.e. a name refers to only one subprogram. Therefore, the argument and result types are determined completely by the name of the subprogram. Only operators are overloaded since e.g. "+" is valid for integer as well as real numbers. Then, it is sufficient for a type–checker to ensure that they are used correctly and to determine the semantics for each used operator.

2.2 Object–oriented languages

This section presents object–oriented concepts and their impact on type checking by surveying different programming languages. Hoopla as well is introduced only conceptually by comparing it with the other languages. A detailed introduction into Hoopla follows in chapter 3. The presentation of the concepts is based on [GHK95]. For the languages the following additional references were used: C++ [Str90, Cli], Modula–3 [CDG+92, Wya94, Sch], Eiffel [Mey, eif], Cecil [CL95, Cha95], and Hoopla ([Hol96]).

2.2.1 Classes and objects

An *object* may be viewed as an abstraction of a "real thing". It has an internal state and may be changed using the object's *methods* (i.e. operations). The internal state is characterized by several *attributes*. Methods define upon which *messages* or method invocations an object may react and how the object reacts.

The control flow in object–oriented systems is implemented by the exchange of messages between objects.

The concept of *classes* is very strongly related to objects. A class describes a set of objects in terms of their structure and hierarchy regarding inheritance or behavior. An object is an instance of a class. Therefore, we might understand a class as a template for generating new objects. Like objects, classes have internal data and methods that might be common for all instances of that class.

The languages $C++$, Modula–3, and Eiffel provide the concept of objects as well as the concept of classes. Cecil is a pure object–oriented model without classes. In addition, Cecil supports multi–methods. We will discuss the difference to single–methods later in this chapter.

Hoopla

In Hoopla there is no explicit class concept that is used as a must–be description for objects. All values are understood as objects. A type describes the representation (attributes) of the object. Furthermore, there are also *type sets* and *type classes* that defines a relation between several types. They provide a similar functionality to usual classes. An instance of a type set or type class is an instance of one type in the type set. Type classes may be extended anytime. The interface of type classes/sets may be defined using subprograms, subprogram classes, and subprogram sets. In subprogram classes/sets several subprograms implement a common signature.

New objects are created using Ada–style *aggregates* or in the case of standard types literals. Objects generated by aggregates or literals have no initial type and are bound to a type as soon as they are assigned to a variable or used in a context. At runtime, each object is bound to exactly one type.

A visualized comparison in terms of classes and objects is shown in figure 2.1. Since Hoopla has a completely different concept, it is not drawn like the other languages with class concepts.

2.2.2 Inheritance and subtyping

Inheritance is a mechanism that allows to reuse attributes and methods from one class in another class. In addition it is possible to extend the class by new attributes and methods

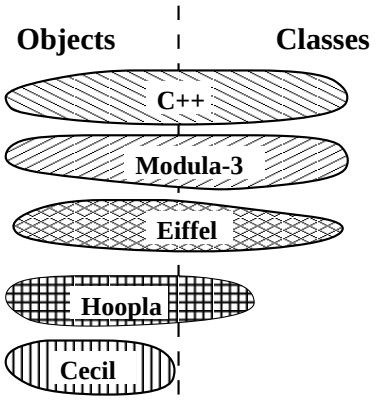

Figure 2.1: Overview: Classes and objects

or to redefine inherited methods. If a class inherits from just one class, we refer to it as *single inheritance*; if the class inherits from several classes directly, we refer to *multiple inheritance*.

Under *subtyping* we understand a hierarchy among classes where a *superclass* is a generalization of one or several more specific *subclasses*. In general, superclasses define some minimal requirements concerning the interface of the class which must be fulfilled by subclasses. Therefore, subtyping could also be viewed as the "inheritance" of interface or of specification.

In C++ and Eiffel inheritance is equivalent with subtyping and both are combined in one concept. That means that by inheriting code or methods from one class to another class, a subtying relation is established. Essentially, Modula–3 combines both concepts too, since inheritance implies subtyping – an additional subtyping relation is provided where the conformance of the functionality of subclass and super class must be ensured by the user. Cecil distinguishes between inheritance and subtyping. Inheritance takes place on the untyped level where the subtyping relation is established among the optional types.

Modula–3 supports only single inheritance. All other discussed languages permit multiple inheritance. Cecil's multiple inheritance mechanism is extended with predicates that define under which conditions an object inherits from another object. Since these conditions might change at runtime, the available methods and fields for an object change. Obviously this concept complicates the task of type checking.

Hoopla

Hoopla distinguishes between inheritance and subtyping. Inheritance is only used to inherit data members from one type's representation to another type's representation. It is possible to retype inherited data members and to inherit from a type only partially. A

type can inherit from several other types (multiple inheritance).

The subtyping relation is merely introduced by type sets and type classes. If a type is included in a type set or type class, there is a "subtyping" relation between the type and the set/class. This subtyping relation is equivalent to the subset relation. All types, type sets, and type classes are in a subtyping relation with the type class `Any`.

Type classes and therefore the subtyping relation may be extended at any point in a program.

A visualized comparison in terms of inheritance and subtyping is shown in figure 2.2.

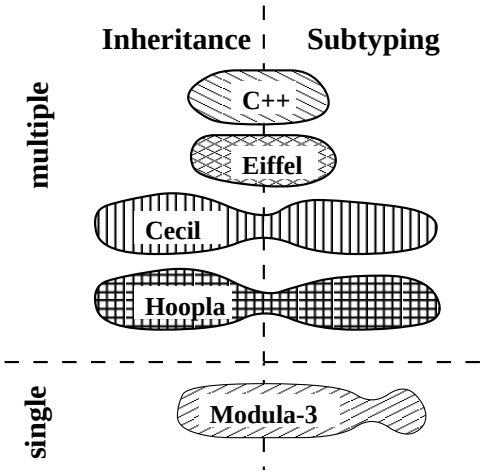

Figure 2.2: Overview: Inheritance and subtyping

2.2.3 Type concepts

In object–oriented languages we distinguish between the *static type* (or *static class*) of an expression, that means the type we can derive at compile time for expressions, and the *dynamic type* (or *dynamic class*) of an object, which is the type of an object at runtime. There are three different concepts how to deal with the static type of an expression and the dynamic type of an object associated with this expression at runtime. We classify the concept as follows.

- *Static typing*: The dynamic type is the same as the static type, i.e. the compiler can determine the type of all expressions in a program. Type errors are recognized at compile time, and therefore there are no such errors at runtime. Usually, it is not possible to combine static typing with inclusion polymorphism.
- *Strong typing*: If there is polymorphism in a language, it is usually not possible to determine the dynamic type of all expressions at compile time, i.e. the dynamic type

differs from the static type. This is the case if objects of a subclass are used in the context of a superclass. The type–checker can determine most type errors, but still additional runtime checks must ensure that there are no type errors at runtime in a program that was compiled without error (*type consistency*).

- *Untyped*: There is no explicit type concept and no type checking takes place. Runtime errors may be caused by type inconsistencies when, e.g. an unknown method is used.

C++, Modula–3, and Eiffel are strongly typed since it is not possible to resolve all method calls in the same way as function calls in traditional, imperative languages because of the fact that the executed method depends on the dynamic type of the argument.

In the case of C++ the type–checker associates a type with each expression using implicit and explicit type conversions. In the other languages implicit type conversions are not used at all or only in a very restricted way.

Type checking Modula–3 tends to be more complicated because *structural equivalence* of two types is used instead of name equivalence. That means that each type must be expanded by replacing constant expressions by their values, and type names by their definitions. The type–checker determines the static type of all expressions.

Cecil supports a hybrid type concept between untyped and strongly typed. The strong type system is layered on top of the untyped language in order to permit rapid prototyping without types and improve the correctness later by adding types. *Types* are abstractions of objects and represent a machine–checkable interface, which is defined using *signatures*. All objects that conform to the type must support the type's interface. Static type–checking takes place when a statically typed expression is assigned to a statically typed variable or formal argument. On the level of types each name is uniquely identified. All signatures are resolvable at compile time, and runtime resolution of methods only takes place inside of signatures.

Hoopla

Hoopla's type system is strongly typed. Like in Cecil all the types, type sets, and type classes are determined at compile time as well as all methods (subprograms, subprogram sets, and subprogram classes) are resolved statically. But late binding still takes place on the basis of type sets and type classes. The compiler does not know the object's dynamic type at compile time. Therefor subprogram sets and subprogram classes are generally bound lately.

Usually the type–checker is only dealing with sets of possible types for expressions and variables (name equivalence for types), and, therefore, checking an assignment is merely a subset check. The checking algorithm is more technical because a list of expressions may be assigned to a list of variables. Since subprograms may return arbitrary numbers

of results, a rather sophisticated algorithm is necessary to check a subprogram call as an argument of another subprogram call.

A visualized comparison in terms of type systems is shown in figure 2.3.

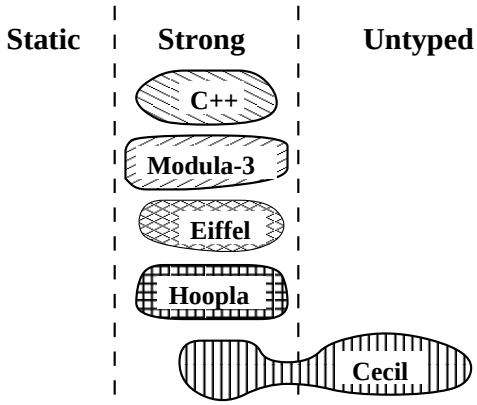

Figure 2.3: Overview: Type concept

2.2.4 Polymorphism and dynamic binding

Polymorphism is the ability of a variable to adopt different shapes at runtime. While, in monomorphic languages, functions and procedures and their formals as well as operators and their operands have a unique type, in polymorphic languages they might have more than one type. Basically, we distinguish between *ad hoc* and *universal* polymorphism.

We refer to *ad hoc* polymorphism if a function can operate on a finite set of types in a way that may depend on the type. There are two cases of ad hoc polymorphism:

- The syntactical concept of *overloading* allows to have one name for different semantic objects like functions or operators. We decide statically, depending on the context which object to use.

- In the semantic concept of *coercion* we do not try to use a suitable function like in overloading. We rather try to transform each argument of a function into a given type (implicit or explicit type conversion at compile time or at runtime.)

Overloading and coercion are provided by C++. Eiffel supports only coercion in a few cases, where Modula–3 provides no ad hoc polymorphism. Cecil is difficult to classify with its two different levels since Cecil provides no ad hoc polymorphism on the untyped language, but the mechanism on the typed level could be regarded as a form of overloading.

If a function acts on a potentially unlimited number of types, we refer to *universal* polymorphism. In contrary to ad hoc polymorphism, the same code will be executed for arguments of different type. Again there are two different cases:

- *Inclusion* polymorphism models the concepts of inheritance and subtyping. An object may be considered to belong to different classes in a subtype hierarchy. Therefore, an object of a subclass might be used in a super class context. It is not possible to determine at compile time whether a more specific object is used in a super class context and therefore method calls cannot be resolved at compile time.

- *Parametric* polymorphism allows to use a function like a template. Explicit or implicit type parameters determine the possible argument types for the usage of a function. Very often parametric polymorphism is implemented by duplicating code at compile time.

All discussed languages support inclusion polymorphism, where this feature is called *pure virtual methods* in C++ in order to distinguish them from overloaded methods. Parametric polymorphism is provided by the following concepts: *templates* (C++), *generic interfaces* (Modula–3), *genericity* (Eiffel), and *parameterized objects* (Cecil).

Polymorphic programming languages allow the usage of constructs in which we do not know the class of an object at compile time or the version of the implementation in a method call.

Dynamic binding means that the name of a method is associated with a method referenced by the name. That means that the binding is established at runtime and not at compile time. This is the case, especially, if inclusion polymorphism is used.

Covariance versus contravariance

There are two different possibilities to redefine methods: following the covariance rule or the contravariance rule. But there are also languages like C++ where no strict policy for redefinition of methods is given.

Usually the contravariance rule is used – this is the case in most programming languages including Modula–3. In the contravariance rule a method m may be redefined by a method m' if and only if the arguments of m are children of those in m' and the result types of m' are children of those in m. This makes type checking very simple, but since there are different rules for arguments and results the contravariance rule is not very natural.

In Eiffel the *covariance* rule is used, i.e. anything declared of type T in a class can be redefined to a child of T in a descendant class. If then an object with such a redefined method is called in a super class context, the type–checker cannot determine statically that the call is legal for modifying the current object.

For Eiffel, different solutions were proposed like determining at compile time all possible dynamic types which is rather infeasible since the entire system needs to be known.

A more restrictive solution was chosen with prohibiting the method calls, called *polymorphic catcalls*, where those problems occur. A call is *polymorphic* if its target is polymorphic, i.e. it can be attached to objects of more than one type. And a call is a *catcall* if some redefinitions of the method would make it invalid because of a change of export status or argument type. Both properties are locally checkable, and therefore, polymorphic catcalls and the problems caused by the covariant rule can be detected at compile time. The catcall property is checked incrementally when calls or redefinitions are made.

This shows that the contravariance as well as the covariance rule has advantages and disadvantages if used as redefinition rule.

Single–methods versus multi–methods

In most object–oriented programming languages single–methods are used. That means that a subprogram is bound to a specific type. If there are different types combined in a class, there might be different subprograms with the same name for each type in the class. When the subprogram name is invoked for an instance of the class, the correct subprogram is invoked, according to the type of the current instance. The algorithm how to choose the correct subprogram is rather straightforward. For example, consider a class `Vehicle` and a type `Car` which is a `Vehicle` too.

```
class Vehicle;
function Vehicle:Drive_1000_miles() return integer is ...
type Car isa Vehicle;
function Car:Drive_1000_miles() return integer is ...
variable Lisa : Vehicle;
variable Peter: Car;
...
Peter.Drive_1000_miles();
Lisa.Drive_1000_miles();
```

We define the function `Drive_1000_miles` for both types. The function defined for `Vehicle` is more general, and the function defined for `Car` is a specialization for this type. If we send a message `Drive_1000_miles` to the variable `Peter`, clearly the more special function will be executed. But if we send the message to `Lisa` we determine which function we want to execute, depending on the current type of `Lisa`'s value. This is determined solely by the single type of the instance we are sending the message to.

In contrast to single–methods, some object–oriented languages use multi–methods. That means that subprograms are not bound to a type or class, but the selection of the right subprogram depends on the current instances of all arguments. Consider the following example.

```
class Vehicle;
type Car isa Vehicle;
function Deal( this_one: Vehicle; for_this: Vehicle ) return
         integer is ...
function Deal( this_one: Car; for_this: Car ) return integer is ...
variable Lisa,Marie: Vehicle;
variable Peter,Tom : Car;
...
Deal(Peter,Tom);
Deal(Lisa,Marie);
```

Now, in the first invocation of `Deal`, obviously the second function will be executed because it is the most special applicable function (considering all arguments). In the second call of `Deal` it depends on the type of `Lisa`'s value and `Marie`'s value at runtime, which function will be executed. If both are instantiations of `Car` it will be the second, more specific function, otherwise the first function. This leads to even more complex algorithms for type–checking. This is especially true since it is possible that there is more than one subprogram applicable for a subprogram call and none of them is more likely applicable than the others. Either the runtime system or the type–checker needs to take care of those situations. In Cecil this is done by the type–checker which verifies the three conditions *conformance*, *completeness*, and *consistency* for multi–methods.

Hoopla

Hoopla does not provide coercion since all type conversions must be done explicitly, but overloading is introduced by type sets and type classes.

Inclusion polymorphism is provided in the form of subprogram classes which provide the possibility to use an object in a superclass context. With *generic types* and *generic subprograms*, Hoopla also supports parametric polymorphism.

In Hoopla the covariance rule is used for the resolution of subprograms. According to the covariance rule, specialization among subprograms is defined. The subtyping relation is defined with subprogram classes and sets where different subprograms implement a subprogram class/set and the applicability is defined by the signature. Therefore, no semantic problems (see [Mey]) with the contra variance rule like in Modula–3 occur. Moreover, there are no type checking problems like in Eiffel. No policy is necessary for the subprograms which implement a subprogram class/set, but full type safety is guaranteed.

The contravariance rule is used for the conformity of subprograms as formal arguments in Hoopla, i.e. a subprogram may be used as a argument in a call if it is contravariant to the subprogram signature given as formal parameter.

Since Hoopla has multi–methods like in Cecil, the methods are defined completely apart from the types. They are not associated with one single type. Since those multi–methods

provide type sets and type classes with an interface, the type sets/classes provide the same functionality as classes. Similarly to Cecil, a type–checker needs to check *completeness*, i.e. all possible argument types are handled, and *uniqueness*, i.e. there are no ambiguities, of subprogram sets and subprogram classes at compile time.

A visualized comparison in terms of polymorphism is shown in figure 2.4.

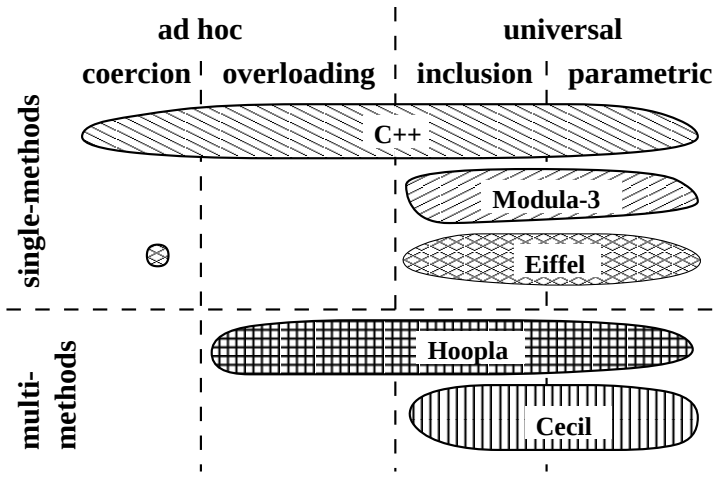

Figure 2.4: Overview: Polymorphism

2.2.5 Encapsulation

In general it is useful to encapsulate a data structure's implementation in order to enable program reuse and maintenance. In object–oriented languages this means especially that we want to hide the internal structure of objects and classes and only present the behavior publically. We refer to this concept as *information hiding*. Very often in singly–dispatched languages like C++ and Eiffel, information hiding may be done by declaring some parts of classes to be public in contrary to other parts. Nevertheless, there are also other approaches, e.g. Modula–3 introduces the additional concept of modules. Also in languages with multi–methods like Cecil a separate concept of information hiding (modules) is necessary because methods do not belong to just one object and therefore they cannot be hidden inside of objects.

If data hiding is supported then the complexity of type checking depends on the world assumption. There are two different assumptions on the world in designing a language:

- the closed–world assumption, and
- the open–world assumption

In the closed–world assumption, the whole program ("world") is known at compile time. In the open–world assumption, the program is divided into independent packages, which can be compiled separately. Type–checking needs to be local for each package. In the age of software engineering and code reuse, the open–world assumption is the more common approach in most languages including the discussed examples.

Hoopla

Data hiding is supported with the separate concept of *packages*. This is again necessary because multi–methods cannot be associated with just one type or class.

In packages types, type sets, and type classes might be declared private as well as subprograms, subprogram sets, and subprogram classes. It is also possible to declare parts of types as private and disable therefore access from other packages.

There are two ways of importing from another package. The normal import assures that only public declarations are visible. A second importing mechanism makes the private parts of declarations visible in the package but those declarations may be used only in a very restricted way.

Existing sets and classes from other packages may be extended in all packages where they are visible. In order to avoid conflicts between different extensions in packages, there are certain conditions for importing from other packages that must hold at link time.

Hoopla follows the open world assumption too.

A visualized comparison in terms of encapsulation is shown in figure 2.5.

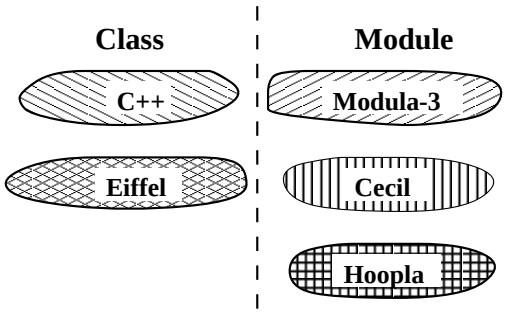

Figure 2.5: Overview: Encapsulation

Chapter 3

The programming language "Hoopla"

This chapter gives a short introduction to the programming language "Hoopla". After a short summary of the general concepts of the language, the basic elements are introduced. Then, this chapter focuses more on the type–checking relevant parts. A more thorough and complete description and the syntax of "Hoopla" may be found in the German document [Hol96]. The algorithms in the following chapters are formulated on the abstract syntax tree which may be found in appendix B.

3.1 General concepts

The programming language "Hoopla" is a new object–oriented language developed at the University of Stuttgart. This description presents version 1.

The basic ideas and concepts that underlie "Hoopla" are the following.

- There are separate concepts for code inheritance and subtype conformance in the language.
- Multi–methods are used instead of single–methods.
- The programmer can decide which subprogram calls will be resolved statically and which will be dispatched at runtime.
- The language description requires complete static resolution and guarantees that dispatching is always possible at runtime.

3.2 Types, Type sets, and Type classes

The flexible type–system in "Hoopla" enables static type–checking of the language. Primarily, this is possible because types and type sets are considered in the type–system.

Types

"Hoopla" offers two different kinds of types to the user:

- enumeration types
- composite types

An *enumeration type* is mainly an enumeration of basic values. The order of those values is arbitrary. Basic values may be numerical literals (`314`), characters (`'c'`), strings (`"string"`), and identifiers. For example,

```
type Colors is { Red, Green, Blue }
```

defines the type `Colors` with three possible values. There are the standard predefined enumeration types `integer` and `boolean`.

It is also possible to inherit the representation of another type. For example,

```
type MyColors is Colors - { Red } + { Yellow }
```

defines the type `MyColors` using the values of `Colors` without the value `Red` but with an additional value `Yellow`.

A *composite type* is a mapping from a set of indices to the possible types. Again the order in which the indices are given is arbitrary. For example, the definitions

```
type X is (1,.., 10 : T)
type Car is ( Color: Colors; Horsepower: integer; Airbag: boolean )
```

may be viewed as an array type `X`, with 10 indices mapped to type `T`, and a record `Car`, with three indices mapped to three different types.

Similar to the enumeration types, it is possible to inherit the structure from another type. It is possible to add new indices mapped to a type, cancel indices, and replace target types by other types. For example,

```
type Truck is Car without Airbag replace Colors by MyColors
           with ( Payload: integer)
```

defines the type `Truck` with three elements `Color`, `Horsepower`, and `Payload` where `Color` is mapped to type `MyColors`.

Where instances of enumeration types are the enumerated literals, instances of composite types must be created using Ada–like *aggregates*. For example,

```
( Color => Red; Horsepower => 120; Airbag => true )
```

creates a valid instance for the type `Car`.

In the same way, a composite type can be reused by another type through inheritance, it is also possible to reuse aggregates for the instantiation of new aggregates. For example,

```
First : Car;
Second : Truck;
...
First  := ( Color => Red; Horsepower => 120; Airbag => true ) ;
Second := First without Color, Airbag
          with ( Color => Blue; Payload => 10);
```

This is called *aggregate reduction* and *aggregate extension.*

Type sets and type classes

As already mentioned "Hoopla" provides *type sets* in addition to single types. Type sets are primarily used to indicate that an expression or a variable may contain a value of one of those types in the set. For example,

```
types Automobile is { Car, Truck }
```

introduces a type set containing two types. As for value sets, union and set difference are possible operations to define type sets. Type sets are non–extendible in contrast to type classes. By using a type set instead of a type class we can ensure that no other types are inserted later.

Type classes are introduced in the same way as type sets, e.g.

```
class Vehicle is { Car, Truck }
```

but they are extendible by an extend declaration, for example,

```
extend Vehicle with {Bike}
```

where `Bike` is another type. This mechanism allows a class to be introduced in the specification of a package and extended in other packages that use the first package.

Moreover, there is a general class `Any` which contains all possible types.

Named types and anonymous types

"Hoopla" distinguishes between named types and anonymous types. Named types were used in the examples above, where we assigned to each type an identifying name. Two types are considered equivalent if they have the same name – not the same structure. For example, in

```
type Point is (x,y: integer);
V,W: Point;
```

we assign the name "`Point`" to the type consisting of two indices (`x` and `y`) mapped to a type named integer. By each use of this name we denote the same type.

In the declaration of variables or subprograms, it is also possible that anonymous types are used, i.e. types which are not associated with a name. For example,

```
V,W : { Yellow, Blue }
X   : ( 1,4,Red: integer )
```

Now, each of the variables `V`, `W`, and `X` are associated with the given type structure. But because we have name equivalence, their types are not equal to any other type – even `V` and `W` are considered to have two different anonymous types. Internally, each anonymous type is associated with a new unique type name. Only literals and aggregates may be assigned to variables of an anonymous type; and instances of anonymous types may only be assigned to variables of the type class `Any`.

Generic types

Generic types are used to describe types in a more abstract way. For example, the element type for a data type `Stack` might be arbitrary. Therefore we could describe the stack as follows.

```
type Stack[T: Any] is ...
```

For each generic parameter, a type set specifies which data types (or type sets) are possible values to instantiate a concrete type. For example,

```
type Int_Stack is Stack[Integer]
```

is a possible generic instantiation.

Type assertions

A *type assertion* is a hint for the compiler how an expression should be interpreted. A type assertion always has the form `type'expression`. It is important that the expression can be interpreted of the stated type at any time. This mechanism might be used to make sure that a constant will be bound to a certain type. For example, in

```
type Digits is { 0,..,9 }
V : { Integer, Digits }
...
V := Digits'(0)
```

the assertion is necessary because otherwise the compiler could not determine the unique current type of the variable.

Type assertions are also helpful and necessary to resolve overloaded subprograms, if there are two applicable subprograms that only differ in the return type.

Type conversions

A *type conversion* enables the user to look at a value of one type as if it was an instance of the second type. This is not a type cast, because it only affects the view or treatment of the value and it does not create a new instance. The notation for a type conversion is `type(expression)`.

A value `v` is *convertible* to a value `w` iff

- `v` and `w` are the same basic value, or
- `v` and `w` are instances of composite types, $Indices(\texttt{w}) \subseteq Indices(\texttt{v})$, and for all $i \in Indices(\texttt{w})$: `v`(i) is convertible to `w`(i).

Type conversion is a runtime issue. That means, if we cannot decide at compile time whether a value is convertible to a new type, the check is postponed until runtime.

3.3 Variables

A variable in "Hoopla" is a reference to a value. In a variable declaration each variable is bound to a type, type set, or type class. For example,

```
V : Integer := 5
W : Vehicle
```

In the first example the variable is provided with an initial value.

At runtime, each variable refers to an object which is bound to exactly one type. If the variable is declared using a type set or type class, each type in the class (or set) denotes a possible type for the objects which might be assigned to the variable at runtime.

The different fields of variables bound to composite types are accessible by writing the index in parenthesis behind the variable name, e.g. `V(2)`. It is also possible to access a whole slice of fields by writing a range as the index of a variable. For example,

```
- V(1,..,4,5) := 1, V(1,..,4);
- V(1),V(2),V(3),V(4),V(5) := 1,V(1),V(2),V(3),V(4);
- X(1) := 1; X(2) := V(1); X(3) := V(2); X(4) := V(3); X(5) := V(4);
  V(1) := X(1); V(2) := X(2); V(3) := X(3); V(4) := X(4); V(5) := X(5);
```

are all equivalent. In the last example, the additional variables are necessary because we need to evaluate all the right–hand side expressions before we assign them to the left–hand side variables.

3.4 Subprograms

The possibilities of how to define subprograms in "Hoopla" are quite similar to those how to define types. There are simple subprograms, subprogram sets, and subprogram classes.

Subprograms

There are two different kinds of simple subprograms in "Hoopla":

- procedures with no return value(s), and
- functions with one or more return value(s).

Possible declarations of subprograms are for example

```
procedure Paint(C: Car; P: Color)
function Combine(X:Integer; Y:Integer) return (Integer;Integer)
```

In the declarations, it is possible to omit the names of formal argument parameters. It is also possible to assign names and default values to the return formal parameters.

Because all variables are references to values, it is possible – in case of a composite type – to change the value a formal parameter is pointing to.

For the definition of a subprogram, the body of the subprogram (and maybe some local variables) needs to be added to the signature. For example,

```
function Combine(X:Integer; Y:Integer) return (Integer,Integer) is
  LocalCopy : Integer
begin
  ...
end Combine;
```

Operators with one or two operands are considered as subprograms too. They may be defined according to the examples above where the operator name is quoted, e.g.

```
function "+"(Digits; Digits) return (Digits)
```

Then, those functions may be used in an operation–like manner:

```
A,B : Digits;
...
A := A + B;
```

Generic subprograms

Generic subprograms are defined according to generic types. For example, in

```
function Pop[T: Any](S: Stack[T]) return (T)
```

the generic parameter is `T` and may be instantiated with any type or type set. Then the argument of the subprogram is determined by the generic instantiation of type `Stack` and the return type is the generic argument.

Like in generic types, instantiations may be generated using an arbitrary subset of the given typeset. For example,

```
Pop[ {Integer,Color} ]( ... )
```

If for any reason the user wants to ensure that generic subprograms may only be instantiated with exactly one type as generic argument, the keyword `type` must be used in front of the generic parameter. For example,

```
function Pop[type T: Any](S: Stack[T]) return (T)
```

Then, the instantiation above was not valid in contrast to all instantiations using exactly one type as argument for the affected generic parameter – and not a type set.

Overloading

Subprograms may be overloaded in "Hoopla", i.e. it is possible to have the same subprogram name for different subprograms. But each subprogram must be invokable in a unique way. That means, it is not possible to define two subprograms with the same name and exactly the same argument types and result types in a scope.

Subprogram sets and subprogram classes

Subprogram sets and subprogram classes are introduced in a quite similar way as type sets. The various subprograms in a subprogram set implement the signature of the subprogram set. For example,

```
function C_AverageCosts( Car ) return (Integer) is ...
function T_AverageCosts( Truck ) return (Integer ) is ...
...
function AverageCosts( Automobile ) return (Integer) is
      { C_AverageCosts, T_AverageCosts }
```

`AverageCosts` is a subprogram on the type set `Automobile`. Therefore, there must be an implementation for the subprogram for each possible implementation of the type set. If there is more than one implementation applicable at runtime, the most specific subprogram is selected (dispatching). In object–oriented terminology, we can conceive the subprogram `AverageCosts` as a method for `Automobile`. Yet in "Hoopla" not only single–methods but also multi–methods are possible. For example, in

```
function Compare_1( Car, {Car,Truck} ) return (Integer) is ...
function Compare_2( {Car,Truck}, Car ) return (Integer) is ...
...
function Compare( Automobile, Automobile ) return (Integer) is
      { Compare_1, Compare_2 }
```

the dispatching at runtime has to consider both arguments to find the correct implementation of `Compare`. In the example above, the problem occurs that for the call

```
C1,C2 : Car
...
Compare(C1,C2)
```

none of the subprograms are more specific than the other. Therefore, it is not enough to ensure that there is an implementation for all possible arguments. It is also important to check that there is exactly one most specific implementation. The next chapter will discuss

in detail which subprogram is more specific than another and how we check whether there is always a most specific subprogram.

Again, subprogram classes are subprogram sets that are extensible. For example,

```
procedure class Scrap( Vehicle ) is {}
extend Scrap with Scrap_Bike;
```

Explicit dispatching

Besides the dispatching in subprogram classes there is a second way to enforce a subprogram call to be dispatched instead of being resolved at compile time. If there are different subprograms with a common subprogram name, e.g. `F`. There are two possibilities to call one of those functions:

```
A := F(X);
A := {F}(X);
```

In the first example, the call is resolved statically. In the second call the most special, applicable subprogram is chosen at runtime.

Of course, it is again necessary to ensure at compile time that there is always exactly one most special subprogram available at runtime.

3.5 Expressions

Since all the operators are conceived as subprograms, expressions consist of literals, aggregates, aggregate extensions and reductions, constants, variables, subprogram calls, type assertions, and type conversions. Because of the fact that subprograms might have more than one return value, but also whole tuples of values, the value of an expression might be a tuple too.

In the next chapter, we distinguish between static expressions which may be computed at compile time and non–static expressions.

3.6 Statements

This section introduces the different statements in “Hoopla”.

Assignments

Assignments have reference semantic, i.e. the value of the expression on the right side is not copied but only a reference to the same value is assigned to the variable on the left side. For example, in

```
type T is (1,2: Integer)
...
V,W : T;
...
V := (1 => 0; 2 => 100);
W := V;
```

both variables, `V` and `W`, point to the same value.

Because of slices in variables and tuples as values of expressions, there are multi–assignments, too. That means that several variables (or fields in variables) get the values on the right side. Consider the following example.

```
A(1,..,4) := 4,5,100,A(1);
B(1,..,3),C := f(100);
```

where `f` is a function returning four values with appropriate types. First the complete right–hand side is evaluated and then assigned to the left–hand variables.

If there is only one value on the right side and more than one variable on the left side, the value is assigned to each variable on the left side. For example;

```
A(1,..,4) := 0;
B(1,..3),C := f(100);
```

where `f` is a function returning only one value with appropriate type.

Conditional branches

The conditional branch as an if–then–else statement has the usual semantics. New `if`s in an else branch may be combined to `elsif`s. Examples are

```
if A < 5 then
  A := 0;
elsif A > 9 then
  A := 10;
else
  A := 5;
end if;
```

There is also a case statement. For example,

```
case A in
  when {value1, value2}     => ...
  when {Integer}              => ...
  when {Boolean, Character} => ...
  others                    => ...
end case;
```

It is possible to branch on the current value of a variable – like in the first branch –, or on the current type of the value – like in the second and the third branch.

Loops

There are three different loops: the for–loop, the while–loop, and the repeat–until–loop.

In the for–loop it is possible to indicate the iteration range either with a type or by enumerating the values. For example,

```
for X := Colors loop
  ...
end loop;
```

or

```
for X := Red, Green, Blue loop
  ...
end loop;
```

The other loops have the usual semantics. Examples are

```
while condition loop
  ...
end loop;
```

and

```
until condition loop
  ...
end loop;
```

Subprogram calls

Each subprogram call is a statement as long as the subprogram is declared as a procedure and is called with the correct number of arguments and correct types.

Return statement

The return statement simply specifies the return values in a function.

3.7 Static resolution

In order to develop and design a type checker for "Hoopla" there are two main issues to be considered. The first deals with the fact that only expressions with correct type may be assigned to variables. And the second deals with the static resolution of overloading.

Type conformity

Following the term *type conformity* refers to the fact that one type (or type set) is assignable to another type (or type set). The language description of "Hoopla" requires that type conformity is statically checked in all assignments, arguments of generic instantiations, and arguments of subprogram calls. That means that the type checker needs to determine the type (or type set) of each expression at compile time. Type conformity for "Hoopla" is defined in a more formal way in the next chapter.

Resolution of overloading

In order to determine the types of all expressions it is necessary to resolve all overloaded functions at compile time. Dispatching at runtime only takes place in subprogram sets and subprogram classes where different subprograms are collected in order to implement the subprogram set/class. But we need to guarantee statically that there is always exactly one most special subprogram to be dispatched for each possible combination of arguments.

3.8 Packages

"Hoopla" follows the open–world assumption which means that there are different packages that might be compiled separately. There is a concept for how to include types and subprograms from different packages. Since this does not affect the type checking mechanism, it is not considered further in this document.

The only issue which must be considered is the fact that the representation (structure) of types can be hidden completely or partly in a package. Then, just the type name may be visible in other packages. In those cases, it is not possible to assign literals or aggregates to variables declared of such a type, because it is not known whether the actual value is an element of the type or not.

Chapter 4

Type Checking

This chapter presents the type checking algorithm for "Hoopla". The general ideas are presented as well as the algorithms on the abstract syntax tree written in VDM. An introduction in VDM may be found in Appendix A. Nevertheless, in order to keep the algorithms simple, we first introduce in Section 4.1 auxiliary data structures to store and pass information in the algorithms. Section 4.2 presents the different relations on type sets and subprograms, and Section 4.3 considers the compatibility between different values and types. The results of both sections are used in Section 4.4 where we present the algorithms to check conformity in a program. Then, Section 4.5 discusses the algorithm to resolve overloaded subprograms in subprogram calls. This is a major result of the thesis. Another focal point of type checking "Hoopla" is presented in Section 4.6 where the algorithms are presented to check at compile time whether dispatching is possible at runtime. Section 4.7 discusses under which circumstances type conversions are correct, and Section 4.8 presents algorithms to process expressions and expression lists. Section 4.9 considers how classes are resolved and transformed into simple sets.

Additional functions that are used as interface between the abstract syntax tree and the data structures are summarized and explained in Appendix C.

4.1 Data structures

In order to formulate the algorithms in the following sections in an obvious and easily understandable way, this section introduces some data structures which are used to hold information on types, type sets, subprograms, or variables. I want to emphasize that its only purpose is better comprehension of the algorithms. It is not necessary to implement these data structures as they are stated here. If their information is used several times, it might be useful to store them as attributes in the abstract syntax tree. But implementation is not an issue in this chapter.

Data structures for types

According to the language definition of "Hoopla", two types are equivalent if they have the same name, and with each use of an anonymous type a new unique type name is associated. This concept looks fairly simple, and in fact only generic type definitions make it more complicated. In the case of generic types, the generic arguments are part of the name with which the type is identified. Therefore, a data structure is needed as unique identifier for types, in which the type name as well as the generic arguments are stored. This data structure is defined as follows.

h_type	:: Name: selected_name GenArgs: h_type_set*

In order to define the data structure for type sets, anonymous types are considered first. We don't generate a new name for anonymous types in this description. It is easier to take care of the restrictive use of anonymous types in the algorithm. Therefore, only a tree node is introduced in order to have a uniform notation. We store the representation of the type for checks on the type structure.

h_anonym	:: Decl: h_rep

Now, the data structure for type sets are defined as follows.

h_type_set	= h_a_type–**Set**
h_a_type	= h_type | h_anonym

In addition to "h_type" and "h_anonym", the keyword "Any" is used to identify the type class `Any`, which includes any other type. Eventually, a data structure for all possible types is introduced containing the data structures above.

h_types	= h_type | h_type_set | h_anonym | "Any"

Following, a few examples demonstrate the use of the data structures.

1. `V : Integer` binds `V` to the type
   mk–h_type(Integer, ⟨⟩).
2. `V : Stack[Integer]` binds `V` to the type
   mk–h_type(Stack, ⟨{mk–h_type(Integer, ⟨⟩)}⟩).
3. `V : Stack[{Boolean, Integer}]` binds `V` to the type
   mk–h_type(Stack, ⟨{mk–h_type(Boolean, ⟨⟩), mk–h_type(Integer, ⟨⟩)}⟩).
4. `V : {Boolean, Integer}` binds `V` to the type
   {mk–h_type(Boolean, ⟨⟩), mk–h_type(Integer, ⟨⟩)}.
5. `V : Any` binds `V` to the type "Any".

Data structures for values

In order to simplify the type checking algorithm, data structures were introduced for values and especially for aggregates in a quite similar way as the data structures for type representations.

h_value	= literal | h_aggregate
h_aggregate	:: Values: h_value $\longrightarrow$ h_value
	Types: h_value $\longrightarrow$ h_types

For aggregates, we distinguish between those indices where we can compute the value at compile time, and those indices where we can only determine the type statically.

The following examples explain how the data structures are used.

1. `(Color => Red; Horsepower => 120; Airbag => true)` is stored as mk–h_aggregate([Color $\rightarrow$ Red, Horsepower $\rightarrow$ 120, Airbag $\rightarrow$ true], [ ]).

2. Supposed the type of the variable `First` is `Car`, which was defined in Chapter 3, but its values are not computable at runtime – maybe `First` is an argument of a subprogram. Then, `First with (Payload => 10)` is stored as mk–h_aggregate([Payload $\rightarrow$ 10], [Color $\rightarrow$ mk–h_type($Colors$, $\langle\rangle$), Horsepower $\rightarrow$ mk–h_type(Integer, $\langle\rangle$), Airbag $\rightarrow$ mk–h_type(Boolean, $\langle\rangle$)]).

Data structures for the internal structure of types

As we mention above, it is sometimes necessary to check whether a value is compatible to a certain data type. Therefore, we introduce a data structure to store the internal representation of types. The representation is not always obvious with inheritance. Moreover, this data structure enables us to distinguish between the computation of the representation and the actual checks using the representation.

A representation of type can either be an enumeration of values for enumeration types, or the mappings of some indices to target types for composite types. In addition it is possible that only the name of the type is visible and its representation is hidden in another package. This is indicated by the keyword “private”.

h_rep	= h_value_set | h_composite | “private”
h_value_set	= h_value–**Set**
h_composite	:: Visible: h_value $\longrightarrow$ h_types
	Private: h_value $\longrightarrow$ h_types

In the representation of composite types we distinguish between those mappings which are visible for all packages importing this type, and those mappings declared only for private use in this package. We need to store them separately in order to forbid access to those indices from other packages.

Again, a few examples explain the use of the data structures.

1. `type Colors is {Red, Green, Blue}` binds `Colors` to the representation { Red, Green, Blue }

2. `type Car is (Color:Colors; Horsepower:Integer; Airbag:Boolean)` binds `Car` to the representation
mk–h_rep([Color → Colors, Horsepower → Integer, Airbag → Boolean], [ ])

3. `type Customer is (pin:Integer) with private (balance:Integer)` binds `Customer` to the representation
mk–h_rep([pin → Integer], [balance → Integer])

Data structures for variables

In case of variables we store the internal type information in addition to the variable name as well as references to the abstract syntax nodes for indices or a slice.

h_var :: Name: selected_name
Type: h_types
Indices: expr*
Slice: expr_list

The following examples illustrate the defined data structures.

1. With the declaration `V : Integer`, an use of the variable `V` is associated with the data structure
mk–h_var(V, mk–h_type(Integer, ⟨⟩), ⟨⟩, ⟨⟩)

2. The variable `V` is declared with `V : X` where `X` is an array.

   (a) `V(2)` is associated with
   mk–h_var(V, mk–h_type(X, ⟨⟩), ⟨2⟩, ⟨⟩)

   (b) `V(1,..,4)` is associated with
   mk–h_var(V, mk–h_type(X, ⟨⟩), ⟨⟩, ⟨1, .., 4⟩)

Data structures for subprograms

Analogously to the data structure "h_types" that represents a type with its generic actuals, a data structure "h_subprog" is introduced to represent subprograms. But in case of subprograms, the name of a subprogram is not sufficient to identify a subprogram, because there might be different subprograms with the same name. Therefore, the declaration is stored instead of the name.

h_subprog	:: Decl: h_subprog_decl Genmap: simple_name $\longrightarrow$ h_types*
h_subprog_decl	= subprog_clause | subprog_class | subprog_def | subprog_set

The mapping is necessary since not only generic parameters can cause the replacement of type names with other types. If a signature is used as formal argument in the signature of a generic subprogram then the mapping of the context is still valid in the signature of the argument.

The following examples illustrate the defined data structures.

1. With the declaration `function F(integer) return integer;` the subprogram `F` is associated with
   mk–*h_subprog*(*Decl*, []), where *Decl* points to the signature of F.
2. With the declaration `function G[T:Any](T, function H(T) return T) return T;`
   (a) the subprogram `G[integer]` is associated with
   mk–*h_subprog*(*Decl*, ["T" $\rightarrow$ *integer*]) where *Decl* points to the signature of G.
   (b) the subprogram `H` inside of G is associated with
   mk–*h_subprog*(*Decl*, ["T" $\rightarrow$ *integer*]) where *Decl* points to the signature of H.

For subprogram calls, there is a data structure which is not based on the declaration but on the subprogram name called. This name might include generic arguments. The function arguments are stored in addition.

h_subprog_call	:: Name: inst_name Args: expr_list Dispatch: **Boolean**

Following, examples for subprogram calls are given. The data structures for subprograms are used equivalently to those for types.

1. `F(A,1)` is stored as
   mk–h_subprog_call(F, $\langle A, 1\rangle$, **false**)
2. {`F`}`(A,1)` is stored as
   mk–h_subprog_call(F, $\langle A, 1\rangle$, **true**)

Data structure for conversions

In order to have a unique notation, an additional structure for type conversions is introduced.

```
h_conversion        :: Type: h_type
                       Arg: expr
```

The value of "Arg" is viewed as a value of type "Type".

4.2 Conformity and specialization

This section discusses the issues like conformity of types, i.e. whether an expression of one type or type set is assignable to a variable associated with another type or type set, conformity of subprograms, and specialization of subprograms. Conformity and specialization occur in connection with definition of subprogram sets, resolution of overloading, and dispatching.

4.2.1 Type conformity

In "Hoopla" two types are equal if they have the same name. In addition, there are no implicit type casts. Therefore, a value of one type can only be assigned to a variable associated with another type if both types have the same name. Since type sets are an important concept in "Hoopla", expressions and variables are very often associated with type sets. Because of name equivalence, assignability (or conformity) is defined as a subset relation between sets of type names. An expression of one type set is assignable to a variable of another type set if the set of the expression's possible types is a subset of the variable's possible types.

Single types are associated with type sets including one type only. There are two different kinds of single types: named types and anonymous types. Let $Type$ be the set of all visible named types declared at a certain point in the program, i.e., all declared types in this compilation unit so far and all imported types from other packages. $Type$ contains all named types as well as all possible instantiations of generic types. Let $Anonym$ be the set of all names associated with anonymous types. Anonymous types are treated separately since each use of an anonymous type is associated with a new, unique, implicit type name. That means that instances of anonymous types do not conform other named or anonymous types – including equivalent anonymous type declarations.

Furthermore, we define the set of all visible named and anonymous types in a scope as $Any := Type \cup Anonym$. This set of types is associated with the type class `Any` in "Hoopla".

Each type set in "Hoopla" is a subset of *Any*. It is not necessary to take care of type classes in a special way. Each type class denotes at each program point a unique, closed type set. Therefore, only this type set is computed and then we can consider type classes as normal type sets.

And last, the empty set is a special type set too. It is especially associated with unbound values like literals, named values, and aggregates. That means that they are assignable to all other possible type sets. As we will see later in section 4.3 assignment of unbound values needs additional type checks.

This results in the following definition of type conformity.

Definition 4.1 (Type conformity) *A type set S is* conform *to a type set R iff*

- *S does not contain anonymous types and $S \subseteq R$, or*
- *R = "Any", i.e. the type class including any type*

As we can easily justify, the conformity relation defines a lattice where $\bot = \emptyset$ is the bottom symbol and $\top = Any$ is the top symbol. Then, we know that there is always a greatest lower bound (glb) and a least upper bound (lub) for any two sets in the lattice.

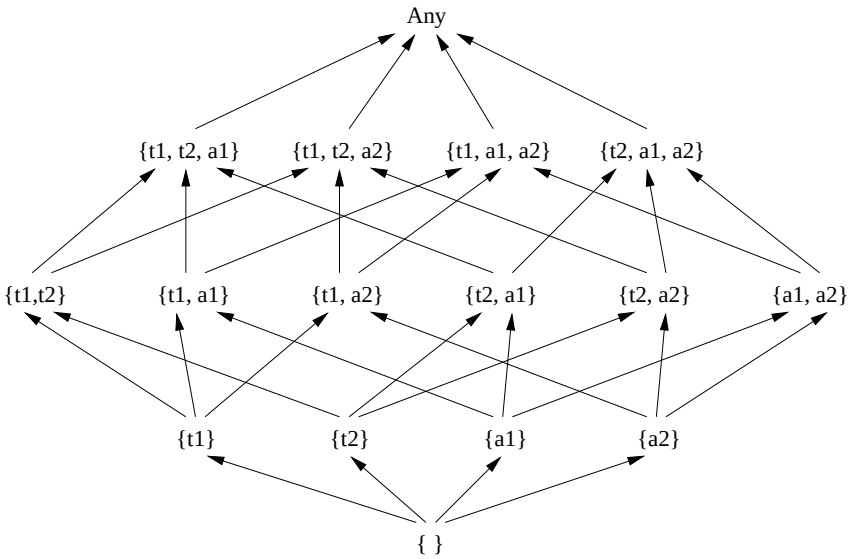

Figure 4.1: Example demonstrating the conformity relation

Figure 4.1 shows an example for the lattice defined by the conformity relation. In this example, `t1`, `t2`, `a1`, and `a2` are named types. Each type (set) in the lattice is assignable to its successors in the lattice. But if `a1` and `a2` are anonymous types, the figure is not correct anymore because each elaboration of its definition is associated with a new name.

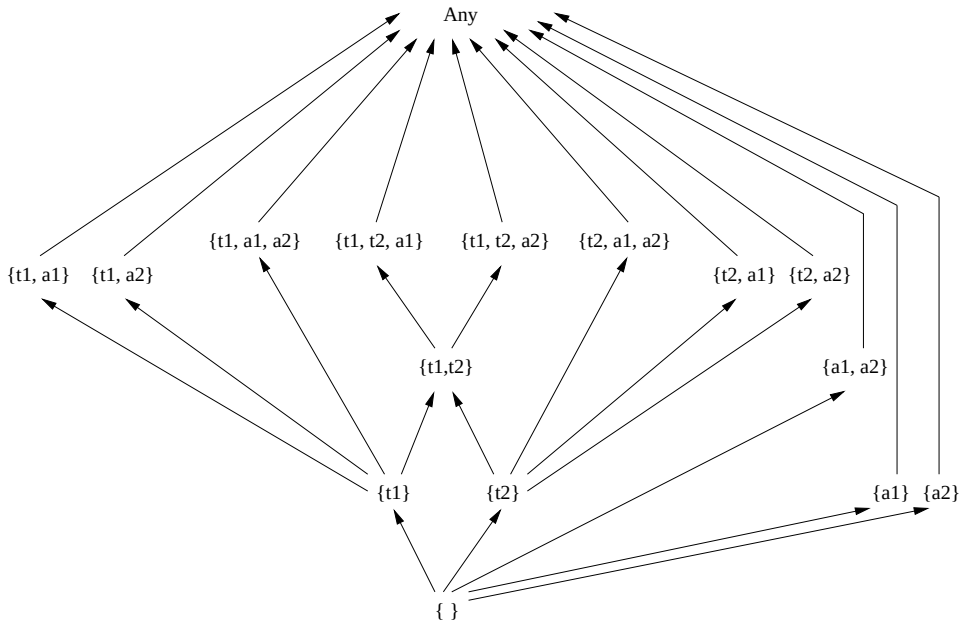

Figure 4.2: Modified example demonstrating the conformity relation

That means one anonymous type used in two different type sets, is associated with two different names. Therefore, the subset relations between `{a1}` and `{t1, a1}` is not possible. Instances of type sets including anonymous types are only assignable to variables of type `Any`. Figure 4.2 shows the resulting lattice for the example above. All the invalid edges are removed and replaced by edges to the type `Any`. In addition, the layout of the nodes is changed.

Now, we can define conformity in a formal way on the abstract syntax tree. Definition 4.1 is in terms of type sets, but as we have already discussed above, single types and values are associated with type sets too. This is done in the formal definition by a function "set" that transforms named single types and values into type sets.

In the previous section, we have presented the conformity relation for type sets and were not talking about single types. In this formal definition we take care of single named types and anonymous types by transforming them into sets containing one type only. Since values, e.g. literals, are associated with the empty set, we allow values as input for this predicate for more convenient use of the predicate. The predicate is defined using an auxiliary predicate "set" to transform named types, anonymous types, and values into type sets.

The function "conform–types" checks whether type (set) t_1 conforms type (set) t_2 .		
INPUT:	t_1	any (but only h_types or h_value make sense)
	t_2	any (but only h_types makes sense)
OUTPUT:	**Boolean**	

$$\begin{aligned}&\text{conform–types}(t_1, t_2) \equiv \\ &\quad \Big(t_2 = \text{“}Any\text{”}\Big) \ \vee \\ &\quad \Big((t_1 \neq \text{“}Any\text{”}) \ \wedge \ (\neg\text{is–h\_anonym}(t_1)) \wedge \\ &\quad \forall t \in \text{set}(t_1) : \ (\text{is–h\_type}(t) \ \wedge \ (t \in \text{set}(t_2))\Big)\end{aligned}$$

It can easily be verified that the predicate implements the condition of the previous section. The first part of the predicate takes care of the class "Any" which is in a conformity relation with all possible sets. For all other sets, we check whether there are only named types in t_1 and each type in t_1 is also in t_2. Note that anonymous types are only assignable to the class "Any" – first condition –, or target in assignments for the empty set – where the second condition is true.

The auxiliary predicate to produce the type sets is defined as follows.

The function "set" transforms types and values into type sets.		
INPUT:	t	h_type, h_type_set, or h_value
OUTPUT:	h_type_set	

$$\begin{aligned}&\text{set}(t) \equiv \\ &\quad \textbf{if } \text{is–h\_type\_set}(t) \textbf{ then } t \\ &\quad \textbf{else if } \text{is–h\_value}(t) \textbf{ then } \emptyset \\ &\quad \textbf{else } \{t\}\end{aligned}$$

4.2.2 Subprogram conformity

If subprograms are used as formal parameter in subprograms then a signature *sig* is given in the subprogram declaration that defines which subprograms are possible arguments in a subprogram invokation, e.g.

```
function G( T, function H( T ) return {T,U} ) return T ...
```

A subprogram with signature sig' is a possible argument if sig' covers all the possible arguments of *sig*, and sig' returns only values that are possible return values of *sig* too. For example, the following subprogram is a possible argument in a call of `G`.

```
function H( {T,U} ) return U ...
```

Now, subprogram conformity is defined more exactly.

Definition 4.2 (Subprogram conformity) *A signature* $F(P_1,\ldots,P_m) \rightarrow (R_1,\ldots,R_n)$ *is* conform *to a signature* $F'(P'_1,\ldots,P'_{m'}) \rightarrow (R'_1,\ldots,R'_{n'})$*, where* P_i *and* P'_i *are the types of the parameters and* R_i *and* R'_i *are the types of the results, iff*

- *the number of arguments is the same* ($m = m'$)*, and*
- *the number of results is the same* ($n = n'$)*, and*
- *each argument* P'_i *of* F' *is conform to the corresponding argument* P_i *of* F ($1 \leq i \leq n$)*, and*
- *each result* R_i *of* F *is conform to the corresponding argument* R'_i *of* F' ($1 \leq i \leq n$)*.*

In the following example

```
function F ( {T1,T2}, T1 ) return ( {T1,T2}, T1 ) ...
function G ( {T1,T2}, {T1,T2} ) return ( T2,T1 ) ...
function H ( T2, T1 ) return ( T2, T1 ) ...
```

signature `G` conforms to signature `F`, because each argument of `F` conforms to `G`'s argument and each result type of `G` conforms to `F`'s result type. `G` is also conform to `H`. But `F` and `H` are in no conformity relation because the arguments of `H` conform to `F`'s arguments but `F`'s result types do not conform to those of `H` and vice versa.

The following function realizes the conformity check for signatures. It uses a second function which extends the check for type conformity on tuples of types.

The function "conform–subprogs" checks whether the subprogram signature t_1 conforms subprogram signature t_2 .		
INPUT:	t_1	any (but only h_subprog makes sense)
	t_2	any (but only h_subprog makes sense)
OUTPUT:	**Boolean**	

conform–subprogs$(t_1, t_2) \equiv$
  is–h_subprog(t_1) $\wedge$ is–h_subprog(t_2) $\wedge$
  conform–tuple(get–subprog–params–types(t_2, env),
    get–subprog–params–types$(t_1, env), env) \wedge$
  conform–tuple(get–return–types(t_1, env),
    get–return–types$(t_2, env), env)$

The function "conform–tuple" checks for two tuples of types, values, and subprogram signatures whether they have the same length and if each component is conform.		
INPUT:	t	(h_types | h_value | h_subprog)*
	t'	(h_types | h_value | h_subprog)*
OUTPUT:	**Boolean**	

conform–tuple$(t, t') \equiv$
len $t =$ **len** $t' \wedge$
$\forall 1 \leq i \leq$ **len** t : $\Big($conform–types$(t(i), t'(i)) \vee$ conform–subprog$(t(i), t'(i))\Big)$

4.2.3 Specialization of subprograms

In order to resolve a conflict between two subprograms which are applicable for a call, the term "specialization" is introduced. That means that the subprogram is chosen in which each argument and each result in the signature is more special than in the other subprogram's signature. Specialization is defined using the conformity relation between types. That means that we check for each argument and each result whether it is conform to the according argument or result. If this holds, the signature is called more special than the other one.

Definition 4.3 (Subprogram specialization) *Let* $F(P_1,\ldots,P_m) \rightarrow (R_1,\ldots,R_n)$ *and* $F'(P'_1,\ldots,P'_{m'}) \rightarrow (R'_1,\ldots,R'_{n'})$ *be signatures.* F *is* more special *than* F'*, where* P_i *and* P'_i *are the types of the parameters and* R_i *and* R'_i *are the types of the results, iff*

- *the number of arguments is the same* ($m = m'$)*, and*
- *the number of results is the same* ($n = n'$)*, and*
- *each argument* P_i *of* F *is conform to the corresponding argument* P'_i *of* F' ($1 \leq i \leq m$)*, and*
- *each result* R_i *of* F *is conform to the corresponding argument* R'_i *of* F' ($1 \leq i \leq n$)
- *not for all* $1 \leq i \leq m$ *and* $1 \leq j \leq n$: $P_i = P'_i$ *and* $R_j = R'_j$ *is true.*

Consider the same example as in the previous section.

```
function F ( {T1,T2}, T1 ) return ( {T1,T2}, T1 ) ...
function G ( {T1,T2}, {T1,T2} ) return ( T2, T1 ) ...
function H ( T2, T1 ) return ( T2, T1 ) ...
```

Now apparently, `H` is more special than `F` as well as `G`. `F` and `G` are in no specialization relation.

The following function realizes the specialization check. It uses the conformity check on tuples from the previous section.

The function "specialized" checks whether the first subprogram is more special than the second.		
INPUT:	$subprog_1$	h_subprog
	$subprog_2$	h_subprog
OUTPUT:	**Boolean**	

specialized$(subprog_1, subprog_2, env) \equiv$
  let $args_1 =$ get–subprog–params–types$(subprog_1, env)$ **in**
  let $args_2 =$ get–subprog–params–types$(subprog_2, env)$ **in**
  let $res_1 =$ get–return–types$(subprog_1, env)$ **in**
  let $res_2 =$ get–return–types$(subprog_2, env)$ **in**
    conform–tuple$(args_1, args_2, env)$ $\wedge$
    conform–tuple(res_1, res_2, env) $\wedge$
    $((args_1 \neq args_2) \vee (res_1 \neq res_2))$

4.3 Structural compatibility between values and types

Above type conformity was defined in order to check whether an instance of one type (or type set) is assignable to an instance of a second type (or type set). In this formalism we identified unbound values like literals, aggregates, and named values with the type $\emptyset$. According to the conformity relation they may be assigned to arbitrary types or type sets. But this causes problems if we try to assign an integer number to a composite type with various indices. Obviously, conformity is not sufficient to guarantee type safety. Therefore, in this section structural compatibility between values and types is introduced. Whenever an instance is associated with the empty set, it is necessary to check for structural compatibility in addition to type conformity.

Definition 4.4 (Structural compatibility) *A value v is* structural compatible *to a type t iff one of the following conditions holds.*

- *v is a basic value and t is an enumeration type where v is an element of the range of t.*
- *v is an aggregate and t is a composite type where the structure of t is completely visible, the indices of v are equal to the indices of t and for each index i: $v(i)$ is structural compatible to $t(i)$.*

In order to specify the formal algorithm for this check, there are a few issues which must be considered in addition to the definition above.

- It might occur that the representation of a type is not visible because it is hidden in another package. If this happens it is not possible to check whether a value belongs to this type and we return the value **false** .
- The above condition for aggregates and composite types is formulated assuming we know the value associated with each index. This is the case at runtime, but at compile time it is possible to determine the type of the value only. Therefore, when implementing the second condition we distinguish between those different kinds of indices and check for compatibility in one case and for conformity in the other.

When testing whether a value is assignable to a type set, the language description of "Hoopla" requires that the value is contained in exactly one type out of this type set since the value can be bound to only one type at runtime. The additional condition was integrated in the structural compatibility because compatibility is only used in this context.

Since all values are assignable to the class "Any" structural compatibility is true for all values and "Any". This happens in the following function if none of the cases discussed above takes place.

The function "compatible" checks whether a value is a possible value of the given type or of exactly one type in a type set.		
INPUT:	*value*	h_value
	types	h_types
OUTPUT:	**Boolean**	

compatible($value, types, env$) $\equiv$
  if is–h_type($types$) $\vee$ is–h_anonym($types$) **then**
    let rep = get–rec–rep($types, env$) **in**
      if rep = "private" **then**
        false
      else if is–h_value_set(rep) **then**
        $value \in rep$
      else if rep = mk–h_composite($visible, private$) **then**
        if ($private \neq []$) $\wedge$ imported–nonprivate(s–Name($types$), env) **then**
          false
        else
          if $value$ = mk–h_aggregate($avalue, atypes$) **then**
            let $map = visible \cup private$ **in**

$$(\mathbf{dom}\ avalues \cup \mathbf{dom}\ atypes = \mathbf{dom}\ map)\ \wedge$$
$$\Big(\forall v \in \mathbf{dom}\ atypes : \text{conform–types}(atypes(v), map(v))\Big)\ \wedge$$
$$\Big(\forall v \in \mathbf{dom}\ avalues : \text{compatible}(avalues(v), map(v), env)\Big)$$

else
false
else
false
else if is–h_type_set($types$) **then**
$\exists!\, t \in type : \text{compatible}(value, t, env)$
else
true — "any"

4.4 Conformity checks

In Section 4.2 conformity is defined. This section has a closer look on how exactly conformity is used in the different checks. Section 4.4.1 discusses the use of conformity in generic instantiations. Then, Section 4.4.2 describes the check of single assignments, where Section 4.4.3 considers arbitrary assignments. As we will see the predicates defined in this part are also applicable for default values and subprogram calls. Initial assignments, return statements, and procedure calls are coverd in Sections 4.4.4, 4.4.5, and 4.4.6. And finally, statements requiring boolean expressions are described in Section 4.4.7.

4.4.1 Checking conformity of generic instantiations

This section presents the algorithm which checks whether a generic instantiation is valid. This is only the case if each generic argument conforms to the type of the formal parameter.

The algorithm is straightforward and distinguishes between the different possible generic declarations. For each formal parameter, the actual argument is checked for conformity to the generic type. This is done by a second function "conform–generic–tuple".

The function "generic–conformity" checks whether the arguments of a generic instantiation are conform to the generic formals.		
INPUT:	*decl*	type_decl, subprog_class, subprog_set, subprog_clause, or subprog_def
	actuals	types*
OUTPUT:	**Boolean**	

generic–conformity($decl, actuals, env$) $\equiv$
cases $decl$:

mk–type_decl(..., *genericformals*, ...) ⟶
  conform–generic–tuple(*actuals*, *genericformals*, *env*)
mk–subprog_class(..., *generics*, ...) ⟶
  conform–generic–tuple(*actuals*, *generics*, *env*)
mk–subprog_clause(..., *generics*, ...) ⟶
  conform–generic–tuple(*actuals*, *generics*, *env*)
mk–subprog_def(..., *clause*, ...) ⟶
  conform–generic–tuple(*actuals*, s–Generics(*clause*), *env*)
mk–subprog_set(..., *clause*, ...) ⟶
  conform–generic–tuple(*actuals*, s–Generics(*clause*), *env*)

The function "conform–generic–tuple" checks whether both tuples have the same length and their elements conform. As chapter 3 pointed out, there are two different kinds of generic parameters. If the keyword `type` is added, only instantiations are allowed where exactly one type is given as an argument. Otherwise, conforming type sets are possible too.

The function "conform–generic–tuple" realizes the actual check whether the generic actuals conform to the parameters.		
INPUT:	*actuals*	types*
	generics	generic_formal*
OUTPUT:	**Boolean**	

conform–generic–tuple(*actuals*, *generics*, *env*) ≡
  if *actuals* = ⟨⟩ ∧ *generics* = ⟨⟩ **then**
    true
  else if *actuals* ≠ ⟨⟩ ∧ *generics* ≠ ⟨⟩ **then**
    let *actual* = **hd** *actuals* **in**
    let *generic* = **hd** *generics* **in**
      if s–*IsType*(*generic*) **then**
        is–h_type(*actual*) ∧
        conform–types(htypes(*actual*, *env*), htypes(s–Types(*generic*), *env*)) ∧
        conform–generic–tuple(**tl** *actuals*, **tl** *generics*, *env*)
      else
        conform–types(htypes(*actual*, *env*), htypes(s–Types(*generic*), *env*)) ∧
        conform–generic–tuple(**tl** *actuals*, **tl** *generics*, *env*)
  else
    false

4.4.2 Checking one assignment

In this section we want to consider conformity of one single assignment, e.g.

```
A := 100 + 2 * f(10)
```

Multiple assignments will be discussed in the next section. In order to check a single assignment, it is necessary to determine the possible types on the right–hand side and check whether they conform to the types of the variable on the left–hand side.

In this section we assume that the types of an expression are already computed – this is described in Section 4.8.

As it was discussed in Section 4.3, a conformity check is not sufficient if the right–hand side is a value which is associated with the empty type set. Then we check for structural compatibility instead of conformity.

The function "conform–assignment" checks whether the assignment of one value or type(set) is conform to another type(set).		
INPUT:	*ltype*	h_types
	rtype	h_types or h_value
OUTPUT:	**Boolean**	

conform–assignment($ltype, rtype, env$) $\equiv$
  if is–h_value($rtype$) **then**
    compatible($rtype, ltype, env$)
  else
    conform–types($rtype, ltype$)

4.4.3 Type checking arbitrary assignments

This chapter considers arbitrary assignments including multiple assignments. Again, all the types of the expressions on the right–hand side must be computed and have to conform to the types of the variables on the left–hand side.

As long as there is only one type set possible for each expression on the right–hand side there are no problems and we have to apply simply the predicate "conform–assignment" for each expression and its associated variable. But consider the following example.

```
function F ( int ) return ( int ) is ...
function F ( int ) return ( int, int ) is ...
...
A,B,C   := 1,A,C;
A,B,C,D := F(A),F(A);
```

The first multiple assignment can be checked immediately because the type is obvious for each expression. In the next assignment, we call a function `F` that can return either one value or two values according to the context in which it is called. Therefore, there are three different possibilities for the types on the right–hand side in this example: two integers, three integers, or four integers. This occurs if there are different functions with the same name. In this case we have to match all the possible return type combinations for the right–hand side to the types on the left–hand side. This matching is shown in figure 4.3.

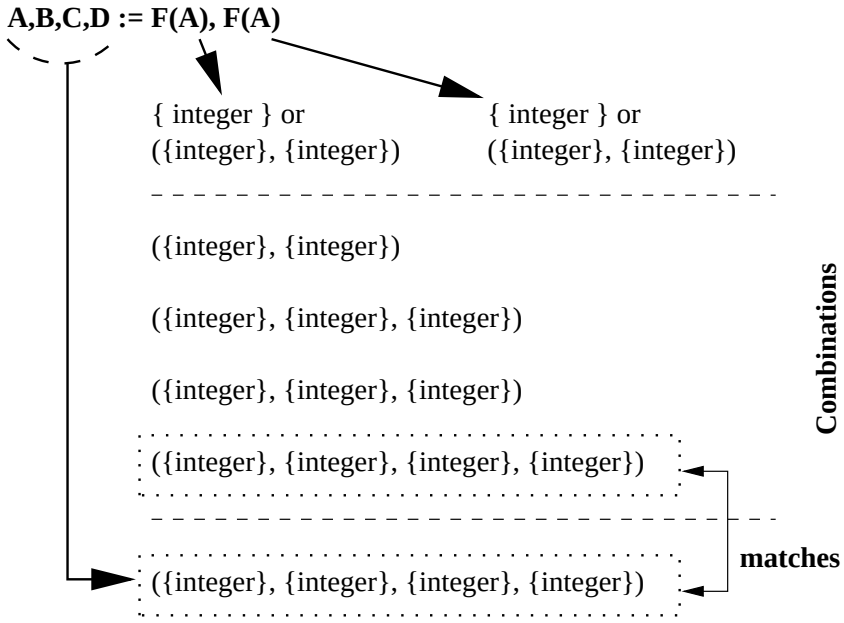

Figure 4.3: Matching types in a multiple assignment

Now, consider that the above expression was assigned to three variables instead of four.

```
A,B,C := F(A),F(A);
```

The according matching is displayed in figure 4.4. Now, the assignment is not unique anymore, since there are two different possibilities how to assign the expression list to three variables. Either the first call returns one integer and the second two, or the other way round. Therefore, we have to make sure that arbitrary assignments are unique.

The matching algorithm is discussed in detail in Section 4.5. The short discussion in this section serves as motivation for the following sections.

The predicate “Check–Conform–Assign–Stmt” which checks the conformity of arbitrary assignments follows. The function “resolve–exprs” settles the computation of the

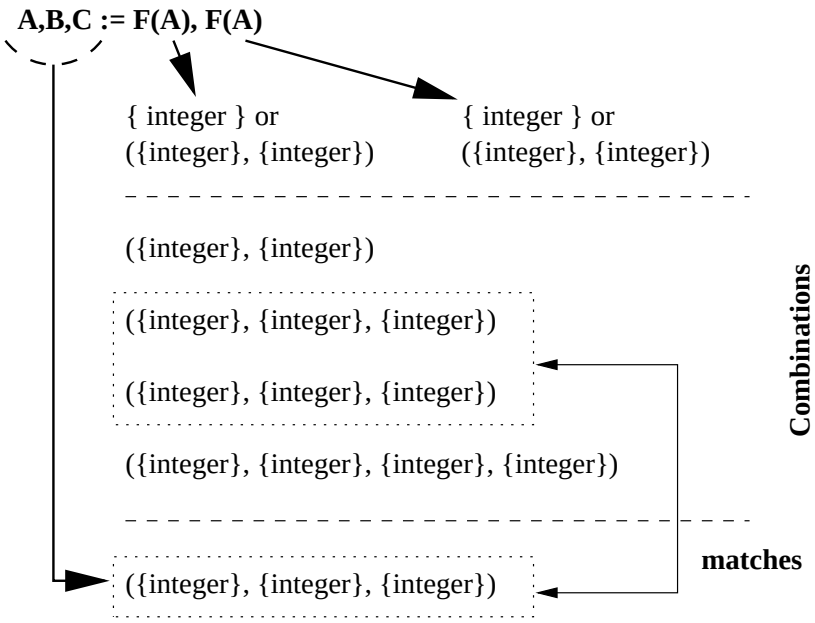

Figure 4.4: Ambiguous matching in an assignment

right hand side and matches the types of both sides. If the matching process fails, the value **nil** is returned. This function is discussed in Section 4.5 in detail, too.

The function "Check–Conform–Assign–Stmt" checks the conformity of an assignment node in the abstract syntax tree.		
INPUT:	stmt	assign_stmt
OUTPUT:	**Boolean**	

Check–Conform–Assign–Stmt($stmt, env$) $\equiv$
  let $stmt$ = mk–assign_stmt($leftside, exprs$) **in**
  let $vartypes$ = get–leftside–types($leftside, env$) **in**
    resolve–exprs($vartypes, exprs, env$) $\neq$ **nil**

4.4.4 Conform initial assignments

In this section, the algorithms for checking conformity in initial assignments are described. Initial assignments occur as initial values for variables and as default values for return parameters in function definitions.

When we declare a variable it is possible to assign the variable an initial value, e.g.

```
A : int := 10
```

In this formal description I only consider literals and aggregates as assignable expressions in a variable's initialization. They are associated with the empty type set. And therefore, it is only necessary to check for structural compatibility in the following function.

The function "Check–Conform–Var–Init" checks whether initial assignments of variable declarations are conform.		
INPUT:	*decl*	var_decl
OUTPUT:	**Boolean**	

Check–Conform–Var–Init($decl, env$) ≡
  let $decl$ = mk–var_decl(..., $binding$) **in**
    if $binding$ = mk–binding_to_type($type, init, \ldots$) **then**
      if $init \neq$ **nil then**
        let $vals$ = **elems** exprs–to–values($init, env$) **in**
          $\forall v \in vals$: compatible(v, htypes($type, [], env$), env)
      else
        true
    else
      error

Similarly, the return values of subprograms may be assigned a default value, e.g.

```
function F ( int ) return ( int := 0 )
```

The type check is exactly the same as with initial values for variables, although a little bit more effort is necessary to extract the used information.

The default values are stored in a node subprog_clause for each subprogram. Therefore, if we call the function first with a subprog_def we call it again with the according subprog_clause. Then for each formal we check conformity with the function "check–conform–formal". This function extracts just the information from the different nodes and checks for structural compatibility.

The function "Check–Conform–Default–Return" checks whether the default values for return values are conform with their types.		
INPUT:	*subprog*	subprog_clause or subprog_def
OUTPUT:	**Boolean**	

Check–Conform–Default–Return($subprog, env$) ≡
  if is–subprog_def($subprog$) **then**
    Check–Conform–Default–Return(s–clause($subprog$), env)
  else

let $formals$ = **elems** s–returns($subprog$) **in**
$\forall f \in formals$: check–conform–formal(f, env)

The function "check–conform–formal" checks whether the default assignment to the formal is conform with its type.		
INPUT:	*formal*	formal_var or subprog_clause
OUTPUT:	**Boolean**	

check–conform–formal($formal, env$) ≡
if is–formal_var($formal$) **then**
if s–default($formal$) ≠ **nil** **then**
let $vals$ = **elems** exprs–to–values(s–default($formal$), env) **in**
$\forall v \in vals$: compatible(v, htypes(s–type($formal$), [], env), env)
else
true
else
error

4.4.5 Conform return statements

Like in normal assignments we also have to check return statements because they are simply assignments to the return parameters of subprograms. Therefore, the context of expressions in return statements is defined in the signature of a subprogram.

The conformity check of return statements is described formally with the following two functions. The first function extracts the return types in a subprogram and invokes the second function which searches in the statement list of the body recursively for return statements and checks them for conformity.

The function "Check–Conform–Return" checks whether the default values for return values are conform with their types.		
INPUT:	*subprog*	subprog_def
OUTPUT:	**Boolean**	

Check–Conform–Return($subprog, env$) ≡
let $context$ = get–return–types(mk–h_subprog($subprog$, []), env) **in**
check–return–stmts(s–StmtList($subprog$), $context, env$)

The function "check–return–stmts" searches recursively all statements for return statements and checks them for conformity to the context.		
INPUT:	*stmts*	stmt*
	context	h_types*
OUTPUT:	**Boolean**	

check–return–stmts($stmts, context, env$) $\equiv$
  if $stmt \neq \langle\,\rangle$ **then**
    let $stmt =$ **hd** $stmts$ **in**
      if is–return_stmt($stmt$) **then**
        resolve–exprs($context$, s–Exprs($stmt$), env) $\neq$ **nil**
      else if is–while_loop_stmt($stmt$) $\vee$ is–until_loop_stmt($stmt$) $\vee$
          is–loop_stmt($stmt$) $\vee$ is–for_loop_stmt($stmt$) **then**
        check–return–stmts(s–Stmts($stmt$), $context$, env) $\wedge$
          check–return–stmts(**tl** $stmts$, $context$, env)
      else if is–if_stmt($stmt$) **then**
        $\Big($$\forall e \in$ **elems** s–Elsifs($stmt$) :
          check–return–stmts(s–Stmts(e), $context$, env)$\Big)$ $\wedge$
        check–return–stmts(s–ThenPart($stmt$), $context$, env) $\wedge$
        check–return–stmts(s–ElsePart($stmt$), $context$, env) $\wedge$
        check–return–stmts(**tl** $stmts$, $context$, env)
      else if is–case_stmt($stmt$) **then**
        $\Big($$\forall b \in$ **elems** s–Branches($stmt$) :
          check–return–stmts(s–Stmts(b), $context$, env)$\Big)$ $\wedge$
        check–return–stmts(s–OthersBranch($stmt$), $context$, env) $\wedge$
        check–return–stmts(**tl** $stmts$, $context$, env)
      else
        check–return–stmts(**tl** $stmts$, $context$, env)
    else
      true

4.4.6 Checking procedure calls

Procedure calls are statements, but since they must be resolved like function calls according to their arguments, we may view them as functions without return values – expressions with an empty context. That means we can apply exactly the same algorithm like for

arbitrary assignments – only the context is an empty list of types. This results in the following algorithm.

The function "Check–Procedure–Calls" checks the conformity of a procedure call.		
INPUT:	stmt	proc_call
OUTPUT:	**Boolean**	

Check–Procedure–Calls$(stmt, env) \equiv$
  resolve–exprs$(\langle\rangle, \langle stmt \rangle, env) \neq$ **nil**

4.4.7 Simple conformity checks

In addition to those more complicated conformity checks discussed in the sections above, there are also type constraints in a few other statements like the constraint that the expression in an if–statement must have the type `boolean`. All those checks are conformity checks too.

Those conditions occur in if–statements, elsif–parts, while–loops, and until–loops. The predicates for the checks follow.

The function "Check–Conform–Bool–Stmt" checks whether the condition is of boolean type.		
INPUT:	*stmt*	if_stmt or elsif or while_loop_stmt or until_loop_stmt
OUTPUT:	**Boolean**	

Check–Conform–Bool–Stmt$(stmt, env) \equiv$
  cases *stmt* :
    mk–if_stmt$(expr, \ldots) \longrightarrow$ check–boolean–conformity$(expr, env)$
    mk–elsif$(expr, \ldots) \longrightarrow$ check–boolean–conformity$(expr, env)$
    mk–while_loop_stmt$(expr, \ldots) \longrightarrow$ check–boolean–conformity$(expr, env)$
    mk–until_loop_stmt$(expr, \ldots) \longrightarrow$ check–boolean–conformity$(expr, env)$

The function "check–boolean–conformity" checks whether an expression is conform or structural compatible to the boolean type.		
INPUT:	*expr*	expr
OUTPUT:	**Boolean**	

check–boolean–conformity$(expr, env) \equiv$
  resolve–expr$(\langle\langle$mk–h_type("boolean", $\langle\rangle)\rangle\rangle, \langle expr \rangle)$

4.5 Static resolution of subprogram calls

This section discusses the computation of types for arbitrary expressions. This is a very simple task as long as the expression is a variable with a declared type or another "basic expression" like type assertions or type conversions. In those cases the type of the expression is the type of the variable, or the type we assert or convert an expression to. Those basic expressions are discussed in more detail in Section 4.8. More effort is necessary if subprogram calls are involved and there are several overloaded subprograms with the same name and different return types. Then for each call the subprogram is determined that is associated with the specific call according to the call's arguments and its context. The algorithm how those subprogram calls are resolved is discussed in the following subsections.

A subprogram call may refer to different kinds of subprograms. Depending on the kind of the call and the kind of the subprogram the type–checker processes the call differently.

Normal subprograms: They are resolved at compile time, i.e. the invoked subprogram is determined statically and the return types of the subprogram are the return types of the subprogram call.

Subprogram sets and classes: Subprogram sets/classes are processed like normal subprograms. The call's return types are determined by the signature of the subprogram set/class. At runtime, dispatching takes place among the subprograms in the set/class. Therefore, each subprogram set/class must be complete and unique, which is described in Section 4.6.

Explicit dispatching calls: They force the compiler that the call is not resolved at compile time but at runtime like in subprogram sets/classes. Since we need to determine the subprogram call's return types, more complicated algorithms are necessary to deal with those calls.

This section is organized as follows. Subsection 4.5.1 explains the different expression types that might occur in the resolution algorithm. Subsection 4.5.2 gives an overview of the algorithm, and 4.5.3, 4.5.4, and 4.5.5 describe the parts of the algorithm in detail. Then, Subsection 4.5.6 discusses the difference between this algorithm and a more common solution. Finally, Subsection 4.5.7 discusses briefly another approach to process explicitly dispatching calls, before the complexity of the algorithm is examined in Subsection 4.5.8.

4.5.1 Types of expressions

The following paragraph briefly discusses the possible types of expressions computed during the first phase of the algorithm. This is important since the type "format" is essential in the algorithm for the resolution of subprogram calls.

The type t of an expression is represented internally as

- h_types$^+$, iff t is a type, a type set, a type class, or a tuple of those.
  This is the case for
  - membership tests which is associated with the boolean type.
  - variables. If the variable is indexed by a slice, a tuple of types, type sets, and type classes will be returned.
  - assertions.
- h_value, iff t is a value, i.e. an aggregate or a literal.
- (h_types$^+$)–**Set**, iff t is a subprogram call. Then, there might be different, applicable subprograms with the same subprogram name. Each tuple in the set describes one possibility of return types. The second phase determines which subprogram will be applied during execution – then only one tuple in the set will be selected.
- h_subprog–**Set**, iff a signature is used as a formal argument for a subprogram call. Then the set of those subprograms is returned that conforms to the signature.

The remainder of this section is concerned with the resolution of subprogram calls and determines the types that are returned. The computation of the types of other expressions are described in "get–basic–expr–types" in Section 4.8.

4.5.2 Resolution algorithm

This section covers the static resolution of subprogram calls. This is one of the crucial parts in the "Hoopla" type checker. The algorithm used for this resolution is a modification of a very common 2–phase algorithm which is described for example in the formal ADA description [Bjø80].

In the original algorithm the context of a call is passed top–down to the leaves of the expression, and by going up again the subprogram calls are resolved according to their arguments and the context.

Because in "Hoopla" a subprogram can return a complete tuple of values instead of just one value, it is necessary to modify the common algorithm in a 2–phase algorithm with "matching". Why the traditional algorithm is not sufficient for the requirements of "Hoopla" is discussed in Section 4.5.6 in more detail.

Phases of the 2–phase algorithm with matching:

Phase 1: Collect recursively – bottom–up – the possible result types for all subexpressions. In order to do this, we match at a subprogram call the already computed types of the arguments against the parameters of the possible subprograms. The return types of all applicable subprograms are the possible return types of the subprogram call.

Matching: The types of the complete expression or expression list are matched against the context of the expression – e.g. the left–hand side of an expression in an assignment.

Phase 2: Determine top–down the actual subprograms invoked at each call site under consideration of the correct context.

How these phases interact is sketched in figure 4.5.

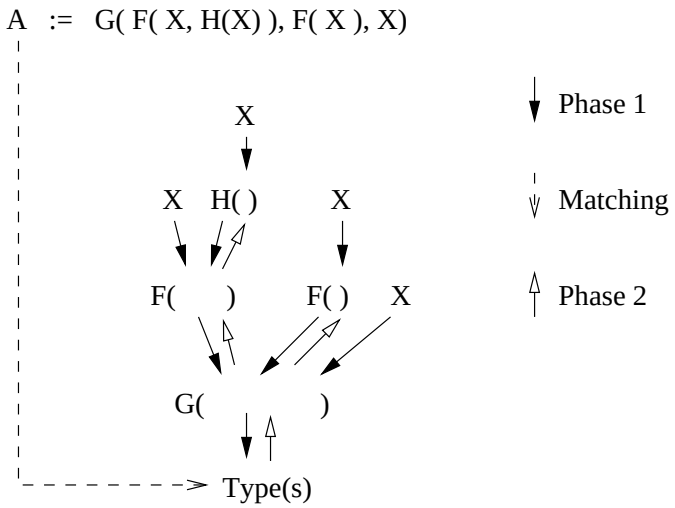

Figure 4.5: Example showing the interaction between the subprogram resolution phases. Phase 1 collects possible return types, matching determines on the highest level which return types are possible, and phase 2 determines the actual subprograms to be called.

Formally the algorithm is written as follows. The result type is either the computed tuple of types or the value **nil** if the expression is ambiguous.

The function "resolve–exprs" computes the types associated with an expression list.		
INPUT:	*context*	(h_types$^+$ | h_subprog)*
	exprs	expr_list
OUTPUT:	((h_types | h_value | h_types$^+$ | h_subprog)*)–**Set** or **nil**	

resolve–exprs($context, exprs, env$) $\equiv$
  let $possible$ = resolve–subprog–phase1($exprs, env$) **in**
  let ok = match–types($context, possible, env$) **in**
    if $ok \neq \emptyset$ **then**
      resolve–subprog–phase2($exprs, ok, env$)
    else
      nil

The two phases and the matching are described in more detail in the following sections.

4.5.3 Phase 1 – Collect result types

This section discusses in detail the first, bottom–up phase of the algorithm, in which we compute the possible return types for each subprogram call depending on the given arguments.

First, we want to look again at a simple example and identify the basic steps in this algorithm. Then, in the following parts of this section, those steps are examined in more detail.

A detailed example

```
function F( T1 ) return ( T1, T3 )                      - F1
function F( {T1,T2} ) return ( {T1,T3} )                - F2
function G( {T1,T3}, T1, {T1,T3} ) return ( {T1,T3} )   - G1
function G( T1, {T1,T3}, {T1,T3} ) return ( T1 )        - G2
function G( T1, {T1,T3}, {T1,T3} ) return ( {T1,T2} )   - G3
function G( T1, T2 ) return ( {T1,T2} )                 - G4
...
X : T1
V : {T1,T2}
...
V := G( F(X), F(X) )
```

Figure 4.6: Example code

Consider the example code shown in figure 4.6. This might be a very pathological example, but it shows all the basic problems that can occur. Furthermore, it might be surprising in the end.

Figure 4.7 shows the computation during the first phase. The algorithm starts bottom–up by determining the type of the variables in the function calls. Then, for the function calls of `F`, we determine the possible return types by looking up which of the functions with name `F` could be invoked by the given arguments. In this case `F1` as well as `F2` are invokable. In the next step, we have to combine the return types of the two function calls `F` in order to get all possible argument combinations for the function call `G`. Then, we look up which subprograms with name `G` could be invoked. And we discover that `G1`, `G2`, as well as `G3` could be invoked by the given arguments. Only `G4` does not find any suitable arguments among the combinations. Therefore, we return the result types of the three applicable functions.

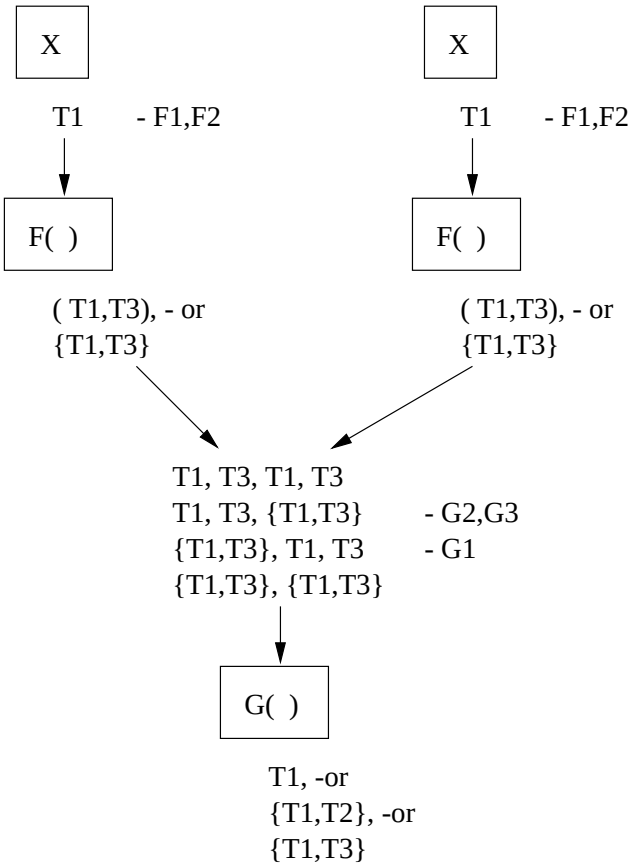

Figure 4.7: Example showing the first phase of the algorithm

Main steps in the first phase

When describing the algorithm, we are looking at how we process one function call in the abstract syntax tree. The steps to do so are actually fairly simple.

1. Get all subprograms, subprogram sets, and subprogram classes which have the correct name

2. Check for each subprogram if the arguments are conform to the subprogram's parameters by using the matching procedure.

3. Return the return types of the possibly correct subprograms.

The recursion will occur in the second step indirectly. The function "resolve–subprog–phase1" computes the possible types of the arguments which are matched against the subprogram's parameters. If those arguments are function calls, the first phase of the resolution algorithm is called for this function call in order to get its possible return types.

Since explicit dispatching requires additional processing in steps 2 and 3, the algorithm for those calls is discussed in the remainder of this section. First, the algorithm according to the description above is presented.

The following function invokes the check for the subprogram calls in an expression list. The actual algorithm of the first phase – the function "possible–return–types" – is described afterwards.

The function "resolve–subprog–phase1" settles the first phase of the resolution algorithm for the subprogram calls in an expression list and returns the possible types of the list.		
INPUT:	*exprs*	expr_list
OUTPUT:	(h_types | h_value | h_types^{+}–**Set** | h_subprog–**Set**)*	

$$
\begin{array}{l}
\text{resolve–subprog–phase1}(exprs, env) \equiv \\
\quad \textbf{let } n = \textbf{len } exprs \textbf{ in} \\
\quad\quad \Big\langle e_1, \ldots, e_n \;\Big|\; \\
\quad\quad\quad \Big((e_i = \text{get–basic–exprs–types}(exprs[i], env)) \wedge \\
\quad\quad\quad\quad \neg \text{is–name\_or\_func\_call}(exprs[i]) \Big) \vee \\
\quad\quad\quad \Big((e_i = \text{possible–return–types}(\text{name–to–hsubprog–call}(exprs[i], env)) \wedge \\
\quad\quad\quad\quad \text{is–name\_or\_func\_call}(exprs[i]) \Big) \Big\rangle
\end{array}
$$

The function "possible–return–types" computes a set of tuples of possible return types of a subprogram call – one for each matching subprogram.		
INPUT:	*call*	h_subprog_call
OUTPUT:	(h_types$^+$)–**Set**	

possible–return–types($call, env$) $\equiv$
  let $call$ = mk–h_subprog_call($name, args, dispatch$) **in**
  let $subprogs$ = name–to–hsubprog–set($name, env$) **in**
  let $argtypes$ = resolve–subprog–phase1($args, env$) **in**
    if $dispatch$ **then**
      dispatching–return–types($subprogs, argtypes, env$)
    else
      let $oksubprogs = \left\{ s \in subprogs \;\middle|\; \text{match–types}(\text{get–subprog–params–types}(s, env), argtypes) \neq \emptyset \right\}$ **in**
      $\left\{ \text{get–return–types}(s, env) \;\middle|\; s \in oksubprogs \right\}$

Problems occurring with explicit dispatching

In the whole strategy presented above it was possible to have either normal subprograms or subprogram sets/classes which would be dispatched. The complete algorithm works because for those sets of subprograms (which will be dispatched at runtime), there is always a unique signature determining what the possible argument types and return types of the whole set/class are. And exactly this is the problem now occurring with explicit dispatching: we need a unique signature covering all the subprograms invokable by the explicit dispatching call. That means that we have to compute a hull for the arguments and the return types for those subprograms. Consider the following example.

```
function F ( {T1,T2}, T1 ) return ( T1 ) is ...     - F1
function F ( T1, T2 ) return ( T2 ) is ...          - F2
function F ( {T1,T2,T3}, T1 ) return ( T3 ) is ...  - F3
A,D : {T1,T2}
B   : {T1,T2,T3}
C   : T1;
...
B := {F}( B, C )
B := {F}( A, D )
```

In the first assignment, the computation of the hull would result in

```
function F ( {T1,T2,T3}, T1 ) return ( {T1,T3} )
```

because only `F1` and `F3` are applicable for the given arguments. At runtime `F1` will be used if the first argument has the type `T1` or `T2`, and otherwise `F3` will be invoked. In this case everything is very smoothly.

In the second example, the hull would be computed as

```
function F ( {T1,T2}, {T1,T2} ) return ( {T1,T2} )
```

because `F1` as well as `F2` could be invoked with a possible combination of the arguments. If `A` is associated with `T2` and `D` is associated with `T1`, `F1` will be invoked. If it is the other way round, `F2` will be called. But what happens, if we call the function with both values associated with the type `T2`? Then, there is no subprogram which is applicable. This example shows that explicit dispatching must be used carefully. It is necessary to check completeness and uniqueness also for explicitly dispatched calls. We will discuss this further in section 4.6.

Given a fixed number of subprograms, the general algorithm for checking an explicit dispatching call is as follows.

Modified steps:

2. (a) Compute the hull of those subprograms
   (b) Verify if the arguments are conform with the hull's parameters
3. Return the hull's return types if the above verification was successful or reject this set of subprograms as a possible dispatchable hull otherwise.

We have to do this computation for each possible subset of the subprograms with this name, because we want to include all possibly dispatchable subprograms and we want to exclude all subprograms which are not invokable with any of the given arguments.

Since there are exponential combinations, this is not a very feasible approach for the implementation. At the end of this section, a different possible approach is presented where some correct expressions are rejected.

This function uses some function to compute the hull of subprograms, which will be explained in the following.

The function "dispatching–return–types" computes the return types of a hull represented by a set of subprograms. Only those subprograms are included which are suitable for the given arguments.		
INPUT:	*subprogs*	h_subprog–**Set**
	args	(h_types | h_value | h_types^{+}–**Set** | h_subprog–**Set**)*
OUTPUT:	(h_types*)–**Set**	

$$\begin{array}{l}\text{dispatching–return–types}(subprogs, args, env) \equiv \\ \quad \textbf{let } combinations = \mathcal{P}(subprogs) \textbf{ in} \\ \quad\quad \Big\{ \text{make–return–hull}(c, env) \;\Big|\; \\ \quad\quad\quad (c \in combinations) \;\wedge\; \text{subprog–hull–is–ok}(c, args, env) \Big\}\end{array}$$

Computing the hull of arguments

If we want to compute the hull of some subprograms, we need to check first, whether all subprograms have the same number of arguments and the same number of return types. If this condition is not fulfilled we are not able to compute a hull of the subprograms. Otherwise this gives us the number of arguments of the subprogram hull.

Then, we simply compute for each argument of the hull the possible types as union of the correspondent arguments in all subprograms in the hull.

The function "make–args–hull" returns the hull of the given subprogram's arguments. If they do not have a hull **nil** is returned.		
INPUT:	*subprogs*	h_subprog–**Set**
OUTPUT:	h_types* or **nil**	

$$\begin{array}{l}\text{make–args–hull}(subprogs, env) \equiv \\ \quad \textbf{if } \exists n, n' \in \textbf{Integer} \;\; \forall sub \in subprogs : \\ \quad\quad \Big(\; (n = \textbf{len } \text{get–subprog–params–types}(sub, env)) \wedge \\ \quad\quad\quad (n' = \textbf{len } \text{get–return–types}(sub, env)) \Big) \textbf{ then} \\ \quad\quad \Big\langle \bigcup_{s \in subprogs} \text{set}(\text{get–subprog–params–types}(s, env)[1]), \ldots, \\ \quad\quad\quad\quad \bigcup_{s \in subprogs} \text{set}(\text{get–subprog–params–types}(s, env)[n]) \Big\rangle \\ \quad \textbf{else} \\ \quad\quad \textbf{nil}\end{array}$$

Verifying which hulls are invokable by arguments

In this step, we check for a given set of subprograms whether the arguments of the call conform the subprograms' hull. This test is done using the matching mechanism introduced in section 4.5.4.

The function "subprog–hull–is–ok" checks whether this subprogram combination is a possible hull.		
INPUT:	*subprogs*	(h_subprog$^+$)–**Set**
	args	(h_types | h_value | h_types$^+$–**Set** | h_subprog–**Set**)*
OUTPUT:	**Boolean**	

subprog–hull–is–ok$(subprogs, args, env) \equiv$
  let $hullargs =$ make–args–hull$(subprogs, env)$ **in**
    $(hullargs \neq$ **nil** $) \wedge$ (match–types$(hullargs, args, env) \neq \emptyset$)

Determine return types of hull

The computation of the hull's return type is analogous to the computation of arguments' hull. The function "make–return–hull" computes the return types for one hull.

The function "make–return–hull" unites the result types of a tuple of subprograms.		
INPUT:	*subprogs*	h_subprog–**Set**
OUTPUT:	h_types*	

make–return–hull$(subprogs, env) \equiv$
  if $\exists n \in$ **Integer** $\forall sub \in subprogs$:
      $(n =$ **len** get–return–types$(sub, env))$ **then**
  $\Big\langle \bigcup_{s \in subprogs}$ set(get–return–types$(s, env)[1]), \ldots,$
      $\bigcup_{s \in subprogs}$ set(get–return–types$(s, env)[n]) \Big\rangle$
  else
    error

4.5.4 Matching

Matching is used for different purposes in the resolution algorithm. First, it is used in assignments to check the possible return types against the context – this is done in "resolve–exprs" in Section 4.5.2. Second, matching is used to check in the first phase of the algorithm which subprogram implementations are invokable with the possible types of the arguments – this happens in "possible–return–types" in Section 4.5.3.

The function "match–types" gets a context and the expression types in the format discussed in Section 4.5.1. First, this representation is transformed into all possible tuples by the function "make–all–tuples". Then, the function "make–one–tuple" transforms each

tuple of tuples into a flat one. Finally, the function "tuple–is–ok" matches each one of the tuples against the context. All correct contexts are returned.

To clarify the different steps, a simple example is given in figure 4.8. The figure shows how the different tuples of types are transformed into the form when they are actually checked for conformity.

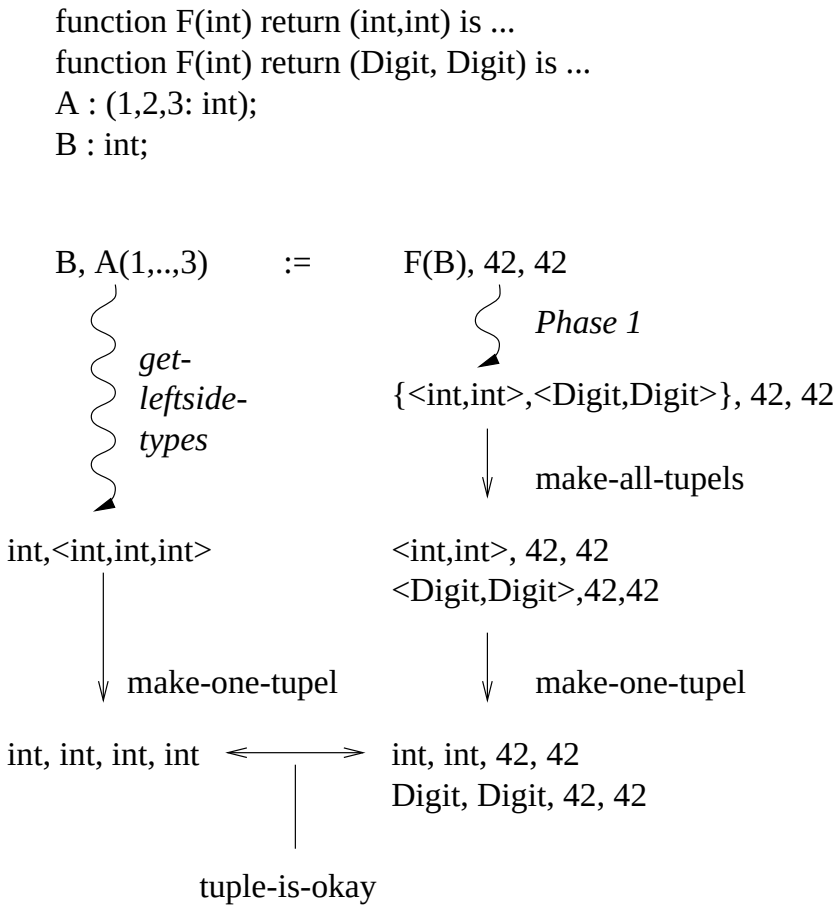

Figure 4.8: Example for the processing of the data tuples in the matching functions

The function "match–types" settles the actual type check of an assignment or a subprogram call. To do so, the types of the left side are associated with the respective expression on the right side in order to resolve function calls.		
INPUT:	*context*	(h_types$^+$ | h_subprog)*
	exprtypes	(h_types | h_value | h_types$^+$–**Set** | h_subprog–**Set**)*
OUTPUT:	((h_types | h_value | h_types$^+$ | h_subprog)*)–**Set**	

match–types($context, exprtypes, env$) $\equiv$
  let $n =$ **len** $exprtypes$ **in**

$$\textbf{let } exprtupels = \text{make–all–tuples}(exprtypes) \textbf{ in}$$
$$\left\{ t \in exprtupels \;\middle|\; \text{tuple–is–ok}(t, context, env) \right\}$$

The function "make–all–tuples" computes all the possible combinations of types given in a tuple of the form described above.

The function "make–all–tuples" gets a tuple where it is possible that at some indices there are more than one possible type. All possible tuples using only one of the types at the indicated indices are returned in a set.		
INPUT:	*exprtypes*	(h_types | h_value | (h_types$^+$)–**Set** | h_subprog–**Set**)*
OUTPUT:	((h_types | h_value | h_types$^+$ | h_subprog)*)–**Set**	

$$\text{make–all–tuples}(exprtypes) \equiv$$
$$\textbf{let } n = \textbf{len } exprtypes \textbf{ in}$$
$$\left\{ \langle t_1, \ldots, t_n \rangle \;\middle|\; \right.$$
$$\Big(\neg \Big(\text{is–}((\text{h\_types}^+)\text{–}\textbf{Set})(exprtypes[i]) \;\vee$$
$$\quad \text{is–}(\text{h\_subprog–}\textbf{Set})(exprtypes(i)) \Big) \;\wedge\; (t_i = exprtypes[i]) \Big) \;\vee$$
$$\Big(\Big(\text{is–}((\text{h\_types}^+)\text{–}\textbf{Set})(exprtypes[i]) \;\vee$$
$$\quad \text{is–}(\text{h\_subprog–}\textbf{Set})(exprtypes[i]) \Big) \wedge\; (t_i \in exprtypes[i]) \Big) \Big\}$$

The actual check whether a tuple is okay is settled by the function "tuple–is–ok". In the tuples generated from the expression list, there are still tuples included that contain the different possible return types for function calls. And in the tuple generated from the variables there might still a tuple be included because of a slice of a variable. In order to compare each single type, we have to transform tuples of tuples into tuples containing only single types and type sets. This is done by using the function "make–one–tuple". Then we check at each index either for subprogram conformity or whether the single assignment is correct.

The function "tuple–is–ok" checks whether the tuple *tupel* is conform in each component to the tuple "vartypes"		
INPUT:	*tupel*	(h_types | h_value | h_types$^+$ | h_subprog)*
	vartypes	(h_types$^+$ | h_subprog)*
OUTPUT:	**Boolean**	

$$\text{tuple–is–ok}(tupel, vartypes, env) \equiv$$
$$\textbf{let } var = \text{make–one–tuple}(vartypes) \textbf{ in}$$
$$\textbf{let } tup = \text{make–one–tuple}(tupel) \textbf{ in}$$
$$\textbf{len } var = \textbf{len } tup \;\wedge$$
$$\forall 1 \leq i \leq \textbf{len } var :$$

$$\Big(\big(\text{is–h\_subprog}(var[i]) \wedge \text{is–h\_subprog}(tup[i]) \wedge \text{conform–subprog}(tup[i], var[i])\big) \vee \text{conform–assignment}(var[i], tup[i], env)\Big)$$

The function "make–one–tuple" concatenates the tuples in the tuple to one big tuple.		
INPUT:	*tupel*	(h_types | h_value | h_types$^+$ | h_subprog)*
OUTPUT:	(h_types | h_value | h_subprog)*	

make–one–tuple($tupel$) $\equiv$
  if $tupel = \langle\rangle$ **then**
    $\langle\rangle$
  else
    let $t =$ **hd** $tupel$ **in**
      if is–htypes(t) $\vee$ is–h_value(t) $\vee$ is–h_subprog(t) **then**
        $\langle t \rangle$^make–one–tuple(**tl** $tupel$)
      else
        t^make–one–tuple(**tl** $tupel$)

4.5.5 Phase 2 – Determine subprogram

This section discusses in detail the second top–down phase of the algorithm, in which we determine at each call, according to the context, which subprogram should be invoked.

A detailed example – continued

In phase 1 we discussed already example 4.6 in figure 4.7. We determined all possible return types in the example. Now, figure 4.9 shows how we determine the subprograms to be invoked at each call in a top–down sweep. For the top level subprogram call, we check which return types are actually conform to the variable type we want to assign them to. This leads to the conclusion that only `G2` and `G3` are invocable given the return type. We decide to associate the subprogram call with subprogram `G2` because `G2` is more special than `G3`. Now, we have exactly one possible return type for each subprogram being an argument for `G2`. We associate the first call of `F` with the return types `(T1,T3)`, and the second call of `F` with the return type {`T1,T3`}. With these return types it is clear that we have to associate the first call with `F1` and the second call with `F2`. And we have resolved all the subprogram calls in this expression.

To get an idea how the two phases interact at one subprogram call, we take a closer look at the function call of G. Figure 4.10 shows the actions of the two phases.

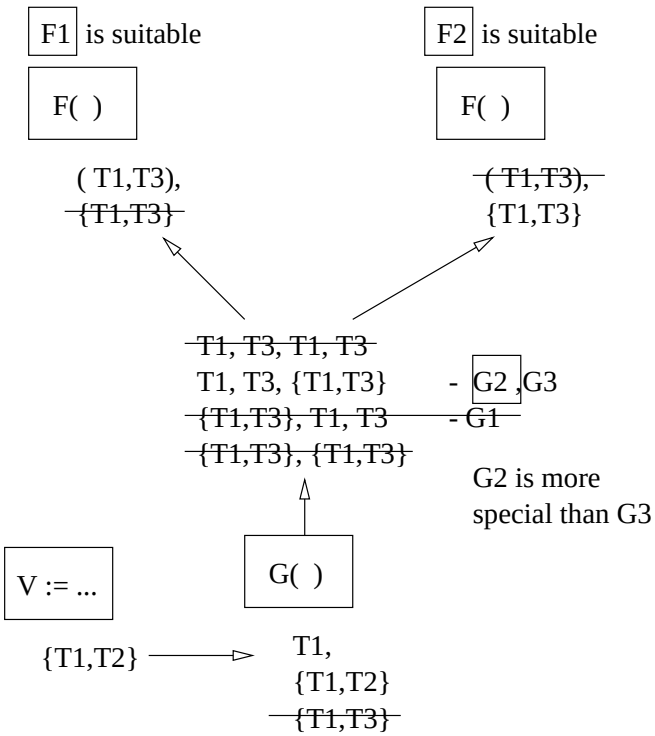

Figure 4.9: Example showing the second phase of the algorithm

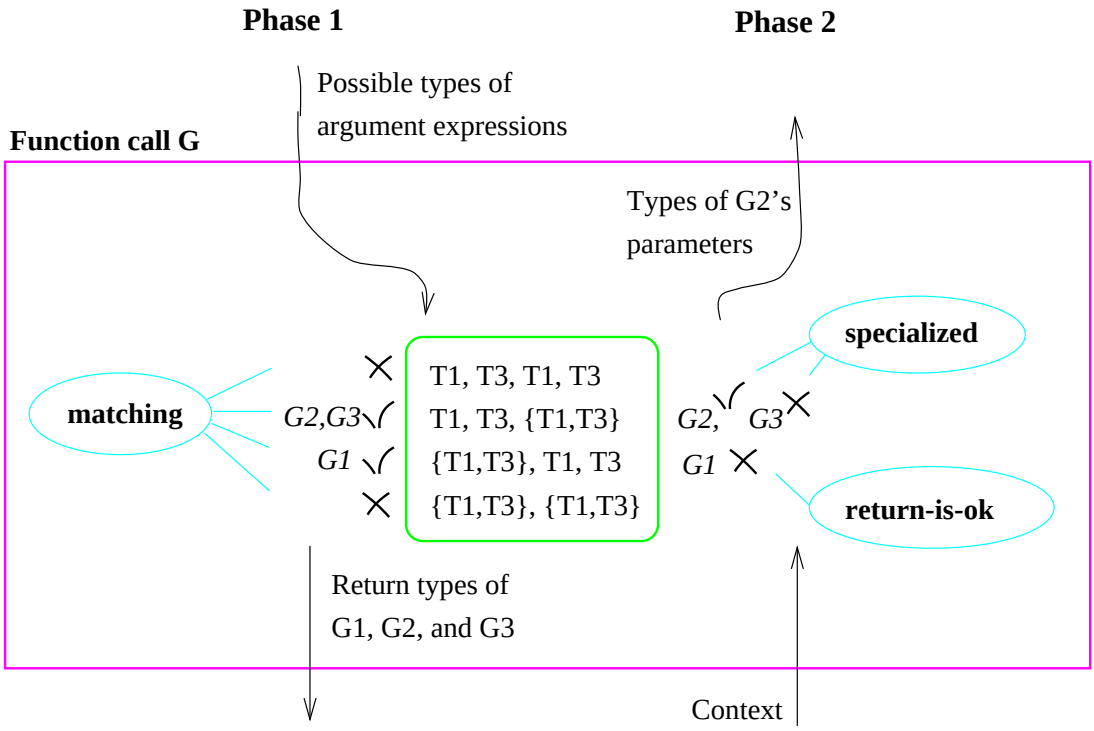

Figure 4.10: Interaction of phase 1 and 2

In phase 1, the possible return types arrive from the call's arguments. There are four possible argument combinations, and by matching those combinations against the parameters of the visible subprograms with name "G" (G_1, G_2, G_3, and G_4) we find out that G_1, G_2, and G_3 are possible for these arguments. Then, we pass the return types of those three subprograms. Note, that only the arguments were considered so far. We can store at the call in the AST which subprograms were okay.

In phase 2, we take the context of the call into consideration. In addition to the arguments from phase 1 we check whether those subprogram conform to the context. We discover that this is not the case with G_1. Finally, we find out that G_2 is more special than G_3, i.e. G_2 is the unique subprogram that will be applied at runtime. The arguments of G_2 are the context for the inner calls and, therefore, passed down in the AST.

Main steps in phase 2

Now, we are given all the possible return types for the arguments of a call – computed in phase 1 –, and we also know the context of the call, i.e. the allowed return types. Using this information we can determine which subprograms are actually invokable for this call and whether there is only one possible invokable subprogram.

The steps to achieve this are as follows:

1. We compute which unique subprogram the regarded call has to be associated with in case of a normal call, or the unique best suiting hull in case of an explicit dispatching call – using the function "choose–sub–implementation".
2. Recursively, we determine the subprograms for the calls in the arguments.
3. In case of calls that will be dispatched, i.e. subprogram sets, subprogram classes, and explicitly dispatching calls, we check for completeness and uniqueness.
4. If the recursion and the additional checks were successful, return **true** , otherwise **false** .

The following function invokes the function "call–is–unique" which settles the second phase of the resolution algorithm for all subprogram calls in an expression list.

The function "resolve–subprog–phase2" completes the resolution algorithm with the second phase for all subprogram calls.		
INPUT:	*exprs*	expr_list
	okcontext	((h_types | h_value | h_types$^+$ | h_subprog)*)–**Set**
OUTPUT:	(h_types | h_value | h_types$^+$ | h_subprog)* or **nil**	

resolve–subprog–phase2($exprs, okcontext, env$) $\equiv$

let $n =$ **len** $exprs$ **then**

$$
\begin{array}{l}
\textbf{let } result = \Big\{ c \in okcontext \;\Big|\; \\
\qquad \forall 1 \le i \le n : \neg \text{is–name\_or\_func\_call}(exprs[i]) \;\vee \\
\qquad\quad (c[i] = \text{call–is–unique}(\text{name–to–subprog–call}(exprs[i], env), \\
\qquad\qquad \{t[i] \mid t \in okcontext\}, env)) \Big\} \textbf{ in} \\
\quad \textbf{if card } result = 1 \textbf{ then} \\
\quad\quad \textbf{let } result = \{c\} \textbf{ in} \\
\quad\quad\quad \langle c \rangle \\
\quad \textbf{else} \\
\quad\quad \textbf{nil}
\end{array}
$$

The following function implements for one subprogram call exactly the steps described above.

The function "call–is–unique" checks whether there is only one possible implementation for a subprogram call and a given context, or – in case of a dispatching call – whether the set of dispatchable subprograms fulfills completeness and uniqueness. If the check is successful the return types of the call are returned and nil otherwise.		
INPUT:	*call*	h_subprog_call
	okreturn	(h_types*)–**Set**
OUTPUT:	h_types* or **nil**	

$$
\begin{array}{l}
\text{call–is–unique}(call, okreturn, env) \equiv \\
\quad \textbf{let } call = \text{mk–h\_subprog\_call}(\ldots, args, dispatch) \textbf{ in} \\
\quad\quad \textbf{if } dispatch \textbf{ then} \\
\quad\quad\quad \textbf{let } subprogs = \text{choose–set–implementations}(call, okreturn, env) \textbf{ in} \\
\quad\quad\quad\quad \textbf{if } subprogs \neq \textbf{nil } \textbf{ then} \\
\quad\quad\quad\quad\quad \textbf{let } hullparams = \text{make–args–hull}(subprogs, env) \textbf{ in} \\
\quad\quad\quad\quad\quad \textbf{let } resolved = \text{resolve–subprog–phase2}(args, \{hullparams\}, env) \textbf{ in} \\
\quad\quad\quad\quad\quad\quad \textbf{if } (resolved \neq \textbf{nil }) \wedge \text{check–complete}(hullparams, subprogs, env) \wedge \\
\quad\quad\quad\quad\quad\quad\quad \text{check–unique}(hullparams, subprogs, env) \textbf{ then} \\
\quad\quad\quad\quad\quad\quad\quad \text{make–return–hull}(subprogs, env) \\
\quad\quad\quad\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{nil} \\
\quad\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad\quad \textbf{nil} \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textbf{let } sub = \text{choose–sub–implementation}(call, okreturn, env) \textbf{ in}
\end{array}
$$

```
if sub = nil then
  nil
else
  let decl = s–Decl(sub) in
  let subparams = get–subprog–params–types(sub, env) in
  let resolved = resolve–subprog–phase2(args, {subparams}, env) in
    if is–subprog_clause(decl) ∨ is–subprog_def(decl) then
      if resolved ≠ nil then
        get–return–types(sub, env)
      else
        nil
    else
      let subs = resolve–subprog–class–set(⟨sub⟩, env) in
        if (resolved ≠ nil ) ∧
             check–complete(subparams, subs, env) ∧
             check–unique(subparams, subs, env) then
          get–return–types(sub, env)
        else
          nil
```

Choosing the subprogram to be invoked

This part considers how to choose the correct, best fitting subprogram for a given call with its arguments and a given context. In phase 1 only the parameters of the call where used to determine the permitted subprograms. Now, in phase 2, in addition the return types must be conform to the context. The algorithm itself is again straightforward, and we describe the different steps preceding the formal description. We distinguish between dispatching calls and normal calls.

In case of normal calls, we proceed as follows:

1. In phase 1, all subprograms were computed that are applicable according to the arguments – now, in addition, the context of the call and the subprograms' return types are considered.

2. If there is more than one subprogram we extract those subprograms that are more special than all other subprograms

3. If this is exactly one subprogram, we return it; otherwise we produce an error

The function "choose–sub–implementation" returns the subprogram implementation which is most suitable for the given subprogram call.		
INPUT:	*call*	h_subprog_call
	context	(h_types*)–**Set**
OUTPUT:	h_subprogram or **nil**	

choose–sub–implementation$(call, context, env) \equiv$
  let $call =$ mk–h_subprog_call$(name, args, dispatch)$ **in**
  let $subprogs =$ name–to–hsubprog–set$(name, env)$ **in**
  let $okphase1 = \Big\{ s \in subprogs \;\Big|$
    match–types(get–subprog–params–types$(s, env), args, env)\Big\}$
  let $oksubprogs = \Big\{ s \in okphase1 \;\Big|$
    return–is–ok(get–return–types$(s, env), context, env)\Big\}$ **in**
  let $special = \Big\{ s \in oksubprogs \Big|$
    $\forall s' \in oksubprogs \setminus \{s\}$: specialized$(s, s', env)\Big\}$ **in**
  if card $special = 1$ **then**
    let $special = \{s\}$ **in**
      s
  else
    nil

In case of dispatching calls, the steps are as follows:

1. Phase 1 computed all those hulls that are applicable to the arguments – now, in addition, the context of the call and the hull of the return types of the subprograms in the hull is computed.
2. The unique hull of those hulls is returned, which contains all other hulls. This most general hull exists according to the construction of the possible hulls.

The function "choose–set–implementations" returns all subprograms that could be dispatched for the given subprogram call.		
INPUT:	*call*	h_subprog_call
	context	(h_types*)–**Set**
OUTPUT:	h_subprogram–**Set**	

choose–set–implementations$(call, context, env) \equiv$
  let $call =$ mk–h_subprog_call$(name, args, dispatch)$ **in**

$$
\begin{array}{l}
\textbf{let } subprogs = \text{name–to–hsubprog–set}(name, env) \textbf{ in} \\
\textbf{let } combinations = \mathcal{P}(subprogs) \textbf{ in} \\
\textbf{let } oksubprogcombs = \Big\{ c \in combinations \;\Big|\; \\
\quad \text{subprog–hull–is–ok}(c, args, env) \;\wedge \\
\quad \text{return–is–ok}(\text{make–return–hull}(c, env), context, env) \Big\} \textbf{ in} \\
\quad (\iota s \in oksubprogcombs)(\forall subs \in oksubprogcombs : subs \subseteq s)
\end{array}
$$

Computing subprograms that conform to the context

Phase 1 used the matching algorithm to consider the types of the arguments and the parameters of the subprogram – this is necessary because of the special structure of the arguments' types. To check the return types, matching is not necessary because all tuples are simple tuples of types. Therefore, we can test directly for conformance.

The function "return–is–ok" checks whether the return types of a subprogram or hull are conform to the possible contexts of the call.		
INPUT:	return	h_types*
	context	(h_types*)–**Set**
OUTPUT:	**Boolean**	

$$
\begin{array}{l}
\text{return–is–ok}(return, context, env) \equiv \\
\quad \exists c \in context : ((\textbf{len } return = \textbf{len } c) \;\wedge \\
\qquad \forall 1 \leq i \leq \textbf{len } c : \text{conform–assignment}(c[i], return[i], env))
\end{array}
$$

4.5.6 New algorithm versus traditional approach

As mentioned previously, our algorithm is a variant of a simpler two–phase algorithm. In the following, we first examine why the original algorithm is insufficient for "Hoopla". Second, we explain how our algorithm manages this problem

The original algorithm

Usually, the original algorithm starts with the context at the most outer function call and determines the possible contexts for the arguments. If those arguments are again function calls, it proceeds recursively. Then, it determines going from inside out for each function call, according to the actual arguments, which function should be invoked.

The reasons for the more complicated way in this type–checking algorithm are the following. First, different functions with the same name are more likely to differ in the argument types than in the result types. Therefore, starting with the arguments reduces

the complexity of the possibilities to be considered. Second, with functions returning different numbers of values, the original algorithm does not work as it is stated in the following lemma.

Lemma 4.1 *The old algorithm is not able to resolve function calls with overloaded functions and return tuples of different lengths.*

Proof: Consider the following example.

```
procedure P( int, int, int ) is ...
function G( int ) return ( int, int ) is ...
function G( int ) return ( int ) is ...
function H( int ) return ( int ) is ...
function H( Digit ) return ( int, int ) is ...
X : int;
...
P( G(X), H(X) )
```

Figure 4.11 shows the first phase of the original algorithm, when the context is computed starting from the procedure call. Then, figure 4.12 shows what happens when we return from the recursion and try to decide for each subprogram call which subprogram has to be invoked. For the call of `H`, it is obvious that the first function is the correct one. But, in case of the function `G`, we cannot decide whether the first or the second program is correct. We would need information how many results `H` actually returns. Therefore, we had to return to the level of `P` and wait for the number of results of `H`. Then, we had to recurse again in order to resolve the unresolved function calls. With this algorithm it is not possible to resolve the function calls walking the tree only once. □

The new algorithm

Now, we want to examine the new algorithm closer. A complete correctness proof is renounced. But we show that the new version of the algorithm does not run into the same problem as the old algorithm.

Lemma 4.2 *The problem discussed in lemma 4.1 is resolved correctly by the new algorithm.*

Proof: The reason for handling the above problem correctly is that each result that is computed for a function call in the first phase has at least one possible subprogram which returns those types. Obviously, when matching result types which are sure to be returned we will get at least one correct result. Less obvious is the answers to the question whether

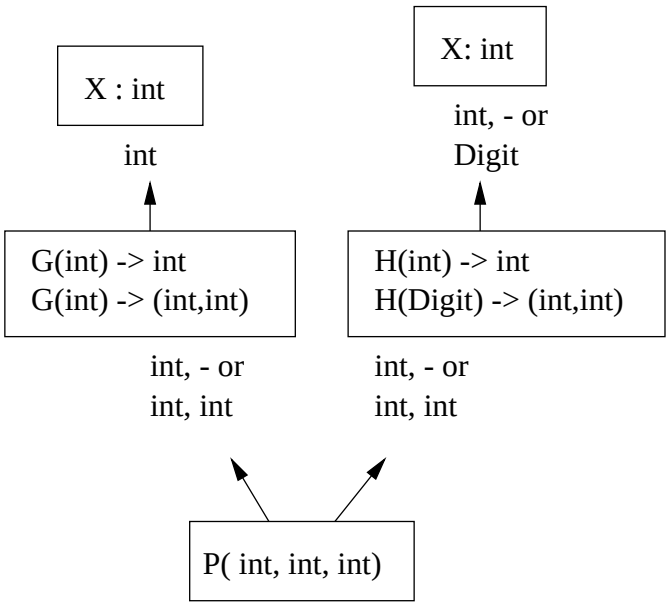

Figure 4.11: First phase of old resolution algorithm.

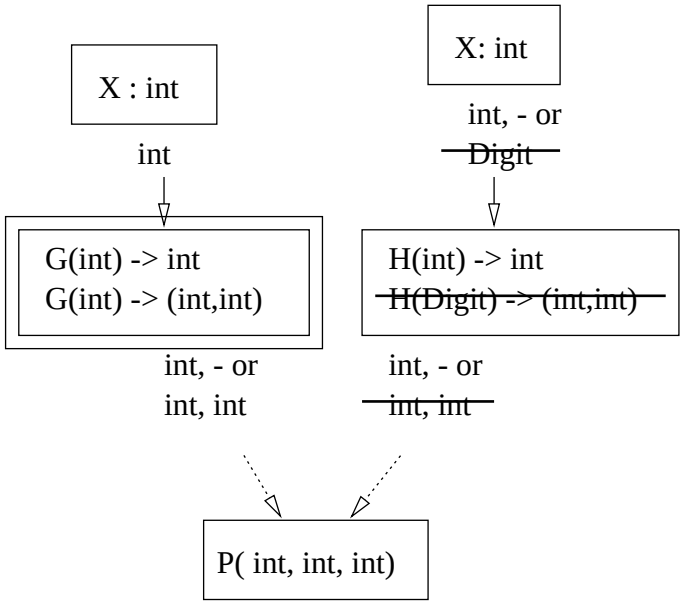

Figure 4.12: Second phase of old resolution algorithm where a problem occurs in resolving `G`.

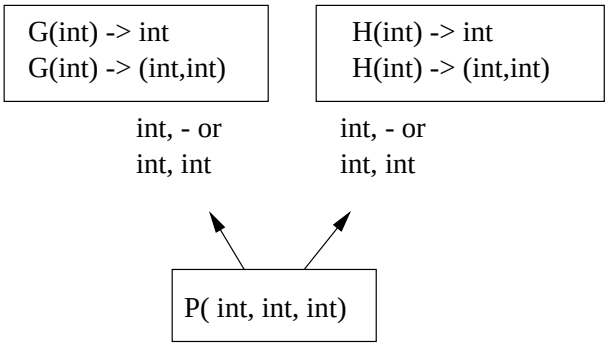

Figure 4.13: An ambiguity when a combination of arguments results from two different result combinations.

the algorithm detects all ambiguities, or what happens if there are more than one possible contexts for a call. We want to address those issues briefly.

We have already seen in the big example in phase 2 what happens if there is more than one context for a function call. Then, all subprograms are computed which are correct with respect to the arguments of the call and have one of those contexts as return types. If – like in phase 2 – there is one most special subprogram among these subprograms, then we choose this subprogram for the call. If, on the other hand, such a subprogram does not exist, we have discovered an ambiguity in the call which cannot be resolved, and we return with an error message.

Now, we take a look at the situation in figure 4.13. In this case, there is one correct tuple of arguments for the call of `P`, but this tuple results from two different combinations of the results of `G` and `H` during the matching. But, in this case, the new algorithm discovers the ambiguity again in the same way as in the first example. Because a subprogram with two results is in no specialization relation to a subprogram with only one result, no unique subprogram can be determined and an error message is returned. □

4.5.7 An alternative approach to process explicitly dispatching calls

The previously presented algorithm to process explicitly dispatching subprogram calls has a high complexity since the power set of the subprograms associated with the call must be considered. Each element of the power set is checked whether it is conform to the arguments. For normal calls we have to consider in the worst case n return tuples in the first phase where n is the number of subprograms. For explicitly dispatching calls, this number may increases up to $\frac{n(n+1)}{2}$ in the worst case. This may slow down the algorithm considerably. Therefore, another approach shall be discussed here that is more restrictive than the previously presented algorithm.

The alternative approach is based on the following observation: If a subprogram hull

is conform to the call's context then all subprograms contained in the hull are conform to the context, too.

Using this property, we can modify the processing of explicitly dispatching calls in phase 1 and phase 2 as follows.

Phase 1: An explicitly dispatching call is handled as follows.

1. Get all subprograms that have the correct name.
2. Exclude all those subprograms that are never applicable with the given arguments, i.e. each subprogram associated with the call is checked whether there is at least one argument combination that could invoke it. At this point no conformity is checked.
3. All those subprogram's results are passed bottom–up to the next subexpression.

Phase 2: The subprogram set is determined during phase two as follows.

1. With regard to the correct subprograms from phase 1, all possible hulls are computed.
2. For each hull conformity of the arguments and the results is checked as well as completeness and uniqueness.
3. Extract the biggest accepted hull as correct subprogram set for the explicitly dispatching call.

The previous algorithm computed the power set of the subprograms already during phase 1 and used them in the rest of the algorithm like normal subprograms which might square the complexity. In the algorithm presented in this section the hulls are computed during phase 2 and checked only locally, i.e. the complexity of checking the remaining nodes of the expression is not influenced by explicitly dispatching calls as much as in the previous algorithm. According to the observation above the further checks during phase 1 are not restricted but rather weakened. I omit any proof whether this algorithm works correctly. Nevertheless, it is obvious that this algorithm is more restrictive than the algorithm presented in the formal specification.

Lemma 4.3 *There are explicitly dispatching subprogram calls that are accepted by the algorithm in Subsection 4.5.2 and rejected by the algorithm presented in Subsection 4.5.7.*

Proof: Consider the following example.

```
function F( T1 ) return ( T1,T1 )       - F1
function F( T1 ) return ( T1 )          - F2
function F( T2 ) return ( T1 )          - F3
function G( {T1,T2} ) return ( T1,T1 )  - G1
```

```
function G( {T1,T2} ) return ( T1 )     - G2
...
X,Y,Z : {T1,T2}
...
X,Y,Z := {F}(X),G(Y)
```

The previously presented algorithm discovers already during phase 1 that there are two possible (biggest) hulls:

```
function F: ( T1 ) return ( T1, T1 )       - {F1}
function F: ( {T1,T2} ) return ( T1 )      - {F2,F3}
```

Furthermore, the algorithm rejects the hull {F1} during phase 1 since the arguments are not conform to the hull. Therefore, the other hull is passed to the assignment and during the matching and phase 2 G2 is selected for the second call. This is a correct assignment according to the presented algorithm.

The algorithm presented in this section passes all three subprograms for F to the assignment. But now it is not possible to find a unique matching. F1 and G1 are assignable to the left–hand side as well as F2 (or F3) and G2. During phase 2 we would discover that F1 is not part of an acceptable hull and therefore {F2,F3} and G2 must be the solution. Then again backtracking was necessary. But with just two phases the algorithm rejects the assignment. □

4.5.8 Complexity of the algorithm

This section gives a short worst case analysis of the algorithm's complexity. For simplification, we assume that the expressions' abstract syntax tree is a full k–ary tree where k is the number of arguments of a call. Moreover, let n be the depth of the tree. First, the matching algorithm and then the two phases of the algorithm are examined.

Complexity of matching

The matching is dominated by the function "make–all–tuples" which generates all possible tuples. There are at most

$$\mathcal{O}(M^E)$$

different tuples, where

M = Maximum number of elements in a set
E = Number of expressions in list to be matched.

For each tuple, a check with $\mathcal{O}(E)$ is necessary, which results in a complexity

$$\mathcal{O}(E * M^E).$$

In case of k–ary trees, the complexity is $\mathcal{O}(k * M^k)$ to check whether the arguments are okay with the subprogram's parameters.

Complexity of phase 1

We formulate the complexity of phase 1 recursively. Since the computation of basic expressions' types is simply a constant look–up, we can write for the complexity

$$C_1 = \mathcal{O}(1).$$

For subprogram calls, we apply phase 1 to the arguments of the call, and match the arguments to the parameters of all possible subprograms, i.e. we get the complexity

$$\begin{aligned} C_i &= k * C_{i-1} + \mathcal{O}(k * M^k) * D, \text{ where} \\ D &= \text{ Number of declared subprograms for the call.} \end{aligned}$$

Then, we get the following complexity for phase 1 by substitution.

$$\begin{aligned} C_n &= k \cdot C_{n-1} + \mathcal{O}(k * M^k) * D \\ &= k^n \cdot C_1 + \left(\sum_{i=1}^{n-1} k^{i-1} \right) \cdot \mathcal{O}(k * M^k) * D \\ &= k^n \cdot \mathcal{O}(1) + \frac{k^{n-1} - 1}{k - 1} \cdot \mathcal{O}(k * M^n) * D \\ &= \mathcal{O}(k^n + k^{n-1} * M^k * D). \end{aligned}$$

Complexity of phase 2

Phase 2 traverses the expression in the same way as phase 1. The subprograms with correct arguments are already computed in phase 1 and only the additional check is necessary whether the context is correct. Since we do not have to consider all subprograms but only those where the arguments were already accepted in phase 1, the complexity of phase 2 is less than or equal to the complexity of phase 1.

4.6 Checks to guarantee dynamic dispatching

This section considers subprogram sets and subprogram classes and the problems which occur when their elements are dispatched at runtime. Each set or class is specified by a signature and implemented by various subprograms. At runtime the most specific subprogram is selected for the given arguments and invoked. There are essentially two problems which could occur:

- There could be no suitable subprogram in the subprogram class.
- There could be more than one applicable subprogram but none of them is more special than the others.

Because we want to avoid those problems at runtime, the type–checker has to ensure that there is always exactly one most special subprogram for all possible arguments. We refer to those required properties as *Completeness* and *Uniqueness*.

Another property that is much simpler to verify is the requirement that all subprograms in a set or class do not return any value which is not conform to the return types of the signature of the subprogram set/class. We refer to this property as *conforming results*.

The following Section 4.6.1 discusses the condition for completeness and presents the algorithm for the completeness check. Section 4.6.2 discusses condition and algorithm for the uniqueness check. Both algorithms are based on the algorithms by Craig Chambers and Gary Leavens [CL95] for checking multi–methods in Cecil. Finally, Section 4.6.3 explains briefly the algorithm for conforming results.

4.6.1 Completeness

In order to ensure completeness, we have to check whether for each possible combination of arguments there is an implementation available which might be dispatched at runtime. Therefore, we define completeness as follows.

Definition 4.5 (Completeness) *A subprogram set/class is called* complete *iff for all possible arguments there is at least one suitable subprogram in the set/class.*

Note, that in the above definition only the arguments are considered. In addition, for each subprogram in the subprogram set or subprogram class it is necessary that the return types of the subprogram are conform to the return types of the set/class. This is not an issue in this section but in Section 4.4.5 .

Consider the following example.

```
function class F ( {T1,T2}, {T1,T2} ) return ( {T1,T2} ) is {f,g}
function f ( T1, {T1,T2} ) return ( T1 )
function g ( {T1,T2}, T1 ) return ( T2 )
```

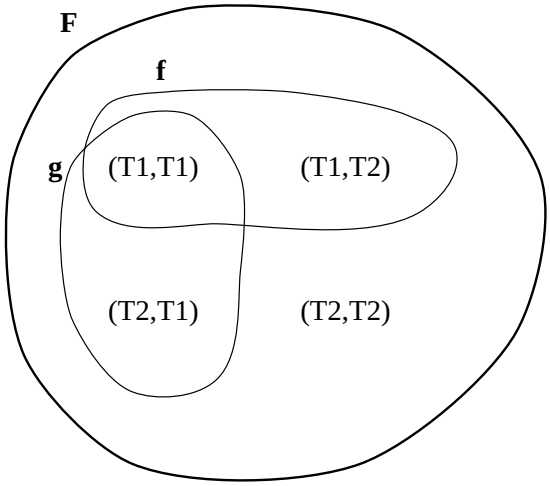

Figure 4.14: Example demonstrating incompleteness

Apparently, `F` might be called with arguments of the types `(T1,T1)`, `(T1,T2)`, `(T2,T1)`, or `(T2,T2)`. Therefore, we need to check for each of these arguments that it is covered by one of the subprograms in the class. Figure 4.14 displays which argument combinations are covered by the subprograms. Obviously, all are covered except `(T2,T2)`. That means the example above is not complete.

This check as well as the uniqueness check discussed below is necessary at the initialization of a subprogram class/set. In case of subprogram classes, they must be checked again if they are extended at some point and called again behind this program point.

The completeness property can be stated formally in VDM as follows.

Condition for completeness. It depends on the class' argument types and the subprograms which implement the class.		
INPUT:	*sigargs*	(h_types | h_subprog)*
	subprogs	h_subprog–**Set**
OUTPUT:	**Boolean**	

$$\begin{aligned}&\text{Complete}(sigargs, subprogs, env) \equiv\\ &\quad \forall args \in \text{possible–types}(sigargs) :\\ &\qquad \text{applicable–methods}(subprogs, args, env) \neq \emptyset\end{aligned}$$

All the possible arguments for invoking the subprogram class are computed by the following function.

The function "possible–types" returns all tuples of types which are included in the argument types of a signature.		
INPUT:	*sigargs*	(h_types | h_subprog)*
OUTPUT:	(h_type | h_subprog)*–**Set**	

$$\begin{aligned}&\text{possible–types}(sigargs) \equiv\\ &\quad \textbf{let } n = \textbf{len } sigargs \textbf{ in}\\ &\quad\quad \Big\{t \in (\text{h\_type} \cup \text{h\_subprog})^n \;\Big|\\ &\quad\quad\quad \forall 1 \le i \le n : \Big(\text{is–h\_types}(sigargs[i]) \;\wedge\; t[i] \in \text{set}(sigargs[i]) \;\vee\\ &\quad\quad\quad\quad (\text{is–h\_subprog}(sigargs[i]) \;\wedge\; t[i] = sigargs[i])\Big)\Big\}\end{aligned}$$

The following function checks for each subprogram in the class whether it is applicable for given argument types and returns the set of all those arguments.

The function "applicable–methods" returns all applicable subprograms for the given arguments.		
INPUT:	*subprogs*	h_subprog–**Set**
	args	(h_type | h_subprog)*
OUTPUT:	h_subprog–**Set**	

$$\begin{aligned}&\text{applicable–methods}(subprogs, args, env) \equiv\\ &\quad \Big\{f \in subprogs \;\Big|\; \text{conform–tuple}(args, \text{get–subprog–params–types}(f, env))\Big\}\end{aligned}$$

Algorithm to check completeness

Now we present the algorithm to check completeness.

The transformation from the completeness condition to the algorithm is straight forward. The only change that was done is that we do not compute the set of all applicable methods and check for the empty set, but rather check for existence of an applicable subprogram and stop the algorithm as soon as we discovered one such subprogram. Then, the following algorithm results.

The function "check–complete" tests for completeness.		
INPUT:	*sigargs*	(h_types | h_subprog)*
	subprogs	h_subprog–**Set**
OUTPUT:	**Boolean**	

$$\begin{aligned}&\text{check–complete}(sigargs, subprogs, env) \equiv\\ &\quad \textbf{let } args = \text{possible–types}(sigargs) \textbf{ in}\\ &\quad\quad \forall a \in args\, \exists sub \in subprogs :\\ &\quad\quad\quad \text{conform–tuple}(a, \text{get–subprog–params–types}(sub, env))\end{aligned}$$

We refer to [CL95] for the proof of correctness.

Complexity

For the worst case complexity of the completeness check we consider the following parameters.

S = Number of subprogram implementations in the set/class
k = Number of arguments
T = Maximal number of types for an argument.

Then, there are $k*T$ argument combinations that must be considered for the completeness check. For each argument combination there are at most S subprograms which may be considered. The resulting complexity is

$$\mathcal{O}(k * T * S).$$

4.6.2 Uniqueness

Uniqueness means that for all subprogram calls there is always just one unambiguous subprogram which will be executed. This condition is defined as follows.

Definition 4.6 (Uniqueness) *A subprogram set/class is called* unique *iff there is always for all possible arguments a most special subprogram in the set/class.*

Consider the following definition of a function class.

```
function class F ( {T1,T2}, {T1,T2} ) return ( {T1,T2} ) is {f,g,h,i}
function f ( T1, {T1,T2} ) return ( T1 )
function g ( {T1,T2}, T1 ) return ( T2 )
function h ( T2, T2 ) return ( T2 )
function i ( T2, T2 ) return ( T1 )
```

Figure 4.15 demonstrates which implementations for the signature are invokable for the possible arguments. To check uniqueness, we have to ensure that for each possible argument there is exactly one most special subprogram. This is the case for `(T1,T2)` and for `(T2,T1)` where there is only one subprogram in the class covering those arguments. But in the case of `(T1,T1)`, `f` as well as `g` are applicable. But none of them is more special than the other. And also in the case of `(T2,T2)`, both `h` and `i` are applicable and equal special. Therefore, subprogram class `F` is not unique.

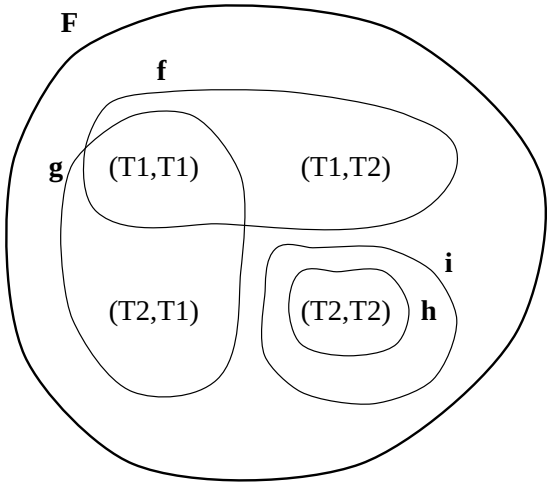

Figure 4.15: Example demonstrating a class being not unique

The following condition for uniqueness follows exactly the same procedure we did in the example above. It considers all possible arguments and checks whether there is exactly one most specialized subprogram among the applicable subprograms.

Condition for uniqueness		
INPUT:	*sigargs*	(h_types | h_subprog)*
	subprogs	h_subprog–**Set**
OUTPUT:	**Boolean**	

$$
\begin{aligned}
&\text{Unique}(sigargs, subprogs, env) \equiv \\
&\quad \forall p \in \text{possible–types}(sigargs) \\
&\quad\quad \exists! sub \in \text{applicable–methods}(subprogs, p, env) \\
&\quad\quad\quad \forall s \in \text{applicable–methods}(subprogs, p, env) \setminus \{sub\} : \\
&\quad\quad\quad\quad \text{specialized}(sub, s, env)
\end{aligned}
$$

Necessary condition

In order to formulate an algorithm for checking uniqueness, we define the following property for subprograms.

Definition 4.7 (Incomparable pair) *Subprograms sub_1 and sub_2 are said to be an* incomparable pair *iff none of them is more special than the other. An incomparable pair is called* non–trivial *iff they cover at least one common argument combination.*

Note that according to the definition of "specialized" (Definition 4.3), two subprograms with equal signatures are an incomparable pair, too.

In figure 4.15 the subprograms `f` and `g` as well as subprograms `h` and `i` are non–trivial incomparable pairs.

We say the condition of incomparable pairs is true for a subprogram class or set if there exist at least two subprograms in the class/set which are incomparable.

In the construction of the algorithm, the following Lemma is used.

Lemma 4.4 (Necessary condition for violated uniqueness) *If the condition for uniqueness is violated then there is a non–trivial incomparable pair in the subprogram set/class.*

Proof: First, we reformulate the uniqueness condition

$$
\begin{array}{l}
\forall p \in \text{possible–types}(sigargs) \\
\quad \exists! sub \in \text{applicable–methods}(subprogs, p, env) \\
\qquad \forall s \in \text{applicable–methods}(subprogs, p, env) \setminus \{sub\} : \\
\qquad\quad \text{specialized}(sub, s, env)
\end{array}
$$

as follows:

$$
\begin{array}{l}
\forall p \in \text{possible–types}(sigargs) \\
\exists sub \in \text{applicable–methods}(subprogs, p, env) \\
\quad \Big[(\forall s \in \text{applicable–methods}(subprogs, p, env) \setminus \{sub\} : \\
\qquad \text{specialized}(sub, s, env)) \ \wedge \\
\quad (\forall sub' \in \text{applicable–methods}(subprogs, p, env) \setminus \{sub\} \\
\qquad \exists s \in \text{applicable–methods}(subprogs, p, env) \setminus \{sub'\} : \\
\qquad\quad \neg\text{specialized}(sub', s, env)) \Big]
\end{array}
$$

Assuming that the condition is violated we negate the formula.

$$
\begin{array}{l}
\exists p \in \text{possible–types}(sigargs) \\
\forall sub \in \text{applicable–methods}(subprogs, p, env) \Big[\ (1) \ \vee \ (2) \ \Big]
\end{array}
$$

where

$$
\begin{array}{lcl}
(1) & \equiv & \exists s \in \text{applicable–methods}(subprogs, p, env) \setminus \{sub\} : \\
 & & \quad \neg\text{specialized}(sub, s, env) \\
(2) & \equiv & \exists sub' \in \text{applicable–methods}(subprogs, p, env) \setminus \{sub\} \\
 & & \quad \forall s \in \text{applicable–methods}(subprogs, p, env) \setminus \{sub'\} : \\
 & & \qquad \text{specialized}(sub', s, env))
\end{array}
$$

Let p be such an argument combination where the uniqueness condition is violated. For the purpose of contradiction we assume that the condition for incomparable pairs is not true. Then,

$$\begin{aligned}&\forall s, s' \in \text{applicable–methods}(subprogs, p, env) :\\ &\quad (\text{specialized}(s, s', env) \ \vee \ \text{specialized}(s', s, env))\end{aligned}$$

Using this property, $\neg\text{specialized}(s, s', env)$ is equivalent to $\text{specialized}(s', s, env)$.

Now, we consider the different cases, where sub is an arbitrary applicable subprogram.

1. If (1) is true, then $\text{specialized}(s, sub, env)$ is true.
2. If (2) is true, then sub' is more special than all other subprograms – including especially sub. That means $\text{specialized}(sub', sub, env)$ is true

Now, we define the following sets of more special subprograms.

$$\begin{aligned}\text{More\_special}(sub) \quad := \quad &\Big\{ s \in \text{applicable–methods}(subprogs, p, env) \ \Big| \\ &\qquad \text{specialized}(s, sub, env) \Big\}\end{aligned}$$

According to the discussion above, $\text{More\_special}(sub) \neq \emptyset$ for all applicable subprograms sub.

Now, we show using an iterative construction, that some subprograms must have equal signatures. This is a contradiction to our assumption and therefore the condition for incomparable pairs must be true.

In the iterative construction we choose arbitrary subprograms that are more special than the previously selected subprograms. Furthermore, we keep track of the subprograms that are still available.

$$\begin{aligned}Next_0 \quad &\in \quad \text{applicable–methods}(subprogs, p, env)\\ Available_0 \quad &:= \quad \text{applicable–methods}(subprogs, p, env) \setminus \{Next_0\}\\ Next_i \quad &\in \quad Available_{i-1} \cap More\_special(Next_{i-1})\\ Available_i \quad &:= \quad Available_{i-1} \setminus \{Next_i\}\end{aligned}$$

for $i \geq 1$. Since in each iteration $Available_i$ is reduced by one element, there exists a $k \geq 1$ such that $Available_{k-1} \cap More\_special(Next_{k-1}) = \emptyset$. Because we have proved above that for all $i \geq 1$ $More\_special(Next_{i-1}) \neq \emptyset$, there could be only subprograms in $More\_special(Next_{k-1})$ that occurred already in $Next_0, \ldots, Next_{k-1}$. That means that some subprograms must have equal signatures. □

Therefore, incomparable pairs is a necessary condition for violated uniqueness. Later in this section we will show a sufficient condition.

Function for the necessary condition

Now, we present the functions to compute incomparable pairs.

The function "incomparable–pairs" returns all subprograms that are not in a specialization relation.		
INPUT:	*subprogs*	h_subprog*
OUTPUT:	(h_subprog × h_subprog)–**Set**	

$$\begin{array}{l}\text{incomparable–pairs}(subprogs, env) \equiv \\ \quad \textbf{let } subprogset = \textbf{elems } subprogs \textbf{ in} \\ \quad\quad \left\{(sub_1, sub_2) \in subprogset \times subprogset \;\middle|\; \begin{array}{l}\neg\text{specialized}\,(sub_1, sub_2, env) \wedge \\ \neg\text{specialized}\,(sub_2, sub_1, env)\end{array}\right\}\end{array}$$

This function returns trivial as well as non–trivial incomparable pairs. As we will see later the trivial ones do not do any harm in the algorithm.

Sufficient condition

Above, we have proved that as a necessary condition for each argument combination where the uniqueness is violated incomparable pairs is true non–trivially. This section provides a sufficient condition.

From the necessary condition we know that as soon as there are no non–trivial incomparable pairs in the subprogram set/class it is unique. For a complete characterization for violated uniqueness we assume from now on that incomparable pairs is true non–trivially, and we give a condition which is true iff the set/class is unique.

First, as a motivation consider the example in Figure 4.16. (f, g) and (h, i) are non–trivial incomparable pairs. Nevertheless, F is unique because for all the argument combinations where there is a conflict, there exists a more special and unique subprogram in F.

Lemma 4.5 (Sufficient condition for violated uniqueness) *Supposed the condition for incomparable pairs is true non–trivially for a subprogram class or set. Then, the subprogram class/set is unique iff the following condition holds for all incomparable pairs (s, s'): the signature that covers exactly those argument combinations, which both s and s' have in common, is complete using only subprograms that are more special than s as well as s'.*

This lemma gives a characterization for all the cases when there are incomparable pairs. As soon as the condition above is not fulfilled for one incomparable pair, the uniqueness property is violated.

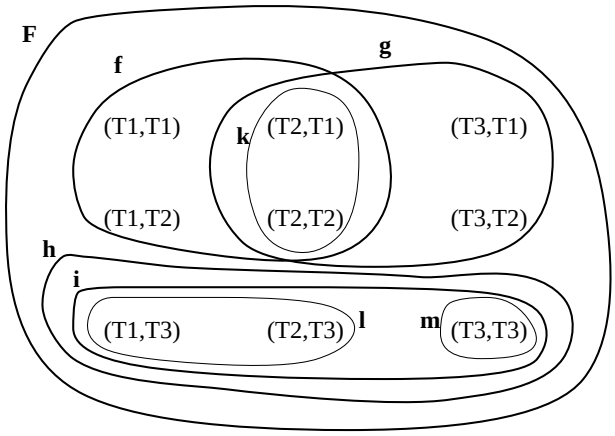

Figure 4.16: Example for the sufficient condition

Proof: Let (s, s') be an incomparable pair. Supposed the condition in the lemma is true, we show that for all argument combinations p for which (s, s') is incomparable, the uniqueness condition is fulfilled. From the condition follows that for each p there is at least one applicable subprogram. If there is more than one, there must be again an incomparable pair according to lemma 4.4, but we have reduced the number of possible subprograms by excluding at least s and s' – in fact we consider only those subprograms which are more special than s and s' (these are exactly those subprograms that are eligible to resolve the conflict between s and s'). Since the number of subprograms in a class is finite, and the condition is fulfilled for all incomparable pairs, a simple induction proves that there is exactly one applicable subprogram for each argument combination.

For the reverse direction we know that the set/class is unique. Then, there is always a most special subprogram for each argument combination. Let (s, s') be an incomparable pair. If (s, s') is trivial, i.e. there are no common arguments, the additional condition in the lemma is true trivially. If (s, s') is non–trivial then according to the uniqueness there is for each argument combination which is covered by both subprograms a subprogram s'' and s'' is more special than s as well as s'. But, then exactly the additional condition in the lemma is true. □

Resulting algorithm

Lemma 4.5 implies the following algorithm. To ensure uniqueness, all incomparable pairs and all subprograms with equal signatures are determined and the sufficient condition is checked.

The function "check–unique" tests for uniqueness.		
INPUT:	*subprogs*	h_subprog–**Set**
OUTPUT:	**Boolean**	

$$\begin{aligned}&\text{check–unique}(subprogs, env) \equiv \\ &\quad \forall (sub_1, sub_2) \in \text{incomparable–pairs}(subprogs, env) : \\ &\qquad \text{check–complete}\Big(\text{glb}(\ \text{get–subprog–params–types}(sub_1, env), \\ &\qquad\qquad\qquad \text{get–subprog–params–types}(sub_2, env)), \\ &\qquad\qquad \text{specialized–subprogs}(sub_1, sub_2, subprogs, env), env\Big)\end{aligned}$$

"glb" is the greatest lower bound. Since the types form a lattice, the greatest lower bound always exists.

The function "specialized–subprogs" returns all subprograms that are more special than both given subprograms.		
INPUT:	sub_1	h_subprog
	sub_2	h_subprog
	subprogs	h_subprog–**Set**
OUTPUT:	h_subprog–**Set**	

$$\begin{aligned}&\text{specialized–subprogs}(sub_1, sub_2, subprogs, env) \equiv \\ &\quad \Big\{ sub \in subprogs \;\Big|\; \text{specialized}\,(sub, sub_1, env) \wedge \text{specialized}\,(sub, sub_2, env) \Big\}\end{aligned}$$

Complexity

For the worst case complexity of the uniqueness check we consider the following parameters – like in the completeness check.

$$\begin{aligned} S &= \text{Number of subprogram implementations in the set/class} \\ k &= \text{Number of arguments} \\ T &= \text{Maximal number of types for an argument.} \end{aligned}$$

There are at most S^2 different incomparable pairs which must be checked. The computation of one incomparable pair requires $\mathcal{O}(k)$. Then for one incomparable pair, the computation of the greatest lower bound requires $\mathcal{O}(k)$ and the computation of the specialized subprograms requires $\mathcal{S} * k$. The completeness check can be at most as costly as the completeness check of the whole subprogram set/class. That means that we get the following worst case complexity:

$$\begin{aligned}&\mathcal{O}(S^2 * (k + k + S * k + S * k * T)) = \\ &\mathcal{O}(S^3 * k * T).\end{aligned}$$

4.6.3 Conforming results

Besides completeness and uniqueness, it is also necessary to ensure that none of the subprograms in a set returns an invalid value which is not conform to the subprogram set's return types.

Definition 4.8 (Conforming results) *A subprogram* $F(P_1, /dots, P_m) \rightarrow (R_1, \ldots, R_n)$ *in a subprogram set/class with signature* $F'(P'_1, /dots, P'_m) \rightarrow (R'_1, \ldots, R'_n)$ *is said to have* conforming results *iff*

- *the number of results is the same* ($n = n'$)*, and*
- *each result* R_i *of* F *is conform to the corresponding argument* R'_i *of* F' $(1 \leq i \leq n)$

Note that subprogram conformity implies conforming results since both terms are defined similarly.

Therefore, conforming results are very easily checkable as follows.

The function "conforming–results" checks for a subprogram whether its results are conform to the results of a second signature.		
INPUT:	sub_1	h_subprog
	sub_2	h_subprog
OUTPUT:	**Boolean**	

conforming–results$(sub_1, sub_2, env) \equiv$
  is–h_subprog(sub_1) $\wedge$ is–h_subprog(sub_2) $\wedge$
  conform–tuple(get–return–types(sub_1, env),
    get–return–types$(sub_2, env), env)$

4.7 Type conversions

This section describes the algorithm for type checking type conversions. In Hoopla, type conversions are the sole part of the type system where runtime checks are necessary. But we can check at compile time whether we know already that the conversion will always fail at runtime. Then, the type checker would not accept the program and exit with an error message.

Check for one type

First, we introduce an algorithm which checks whether one type or value is convertible to the representation of another type. The function is a direct transformation of the conditions for dynamic conversion in Chapter 3.

The function "convert–type–to–type" checks whether a conversion from a value or type to the given type is possible.		
INPUT:	*type*	h_type or h_anonym
	etype	h_type, h_anonym, or h_value
OUTPUT:	**Boolean**	

convert–type–to–type($type, etype, env$) $\equiv$
  let rep = get–rec–rep($type, env$) **in**
    if is–h_types_set(rep) **then**
      convertible–to–type($rep, etype, env$)
    else if is–literal($etype$) **then**
      if is–h_value_set(rep) **then**
        $etype \in rep$
      else
        false
    else if $etype$ = mk–h_aggregate($values, types$) **then**
      if is–h_value_set(rep) **then**
        $etype \in rep$
      else if rep = mk–h_composite($visible, private$) **then**
        if $(private \neq [])$ $\wedge$ imported–nonprivate(s–Name($type$), env) **then**
          false
        else
          let $map = visible \cup private$ **in**
          let $emap = values \cup types$ **in**
            (**dom** $map \subseteq$ **dom** $emap$) $\wedge$
            $\Big(\forall v \in$ **dom** $emap$:
              convert–type–to–type($map(v)$, get–rec–rep($emap(v), env$), env)$\Big)$
      else
        false
    else
      let $erep$ = get–rec–rep($etype, env$) **in**
        if is–h_value_set($erep$) **then**
          if is–h_value_set(rep) **then**
            $erep \subseteq rep$
          else
            false
        else if $erep$ = mk–h_composite($evisible, eprivate$) **then**

```
        if rep = mk–h_composite(visible, private) then
          if ((eprivate ≠ []) ∧ imported–nonprivate(s–Name(etype), env)) ∨
            ((private ≠ []) ∧ imported–nonprivate(s–Name(type), env)) then
            false
          else
            let emap = evisible ∪ eprivate in
            let map = visible ∪ private in
              (dom map ⊆ dom emap) ∧
              (∀t ∈ dom map :
                convertible–to–type(map(t), get–rec–rep(emap(t), env), env))
        else
          false
      else
        false
```

Now, we can extend the function to a more general function for conversion of arbitrary types – including type sets – and values to one type.

The function "convertible–to–type" checks whether an arbitrary type is convertible into a given type.		
INPUT:	*type*	h_types
	etype	h_types or h_value
OUTPUT:	**Boolean**	

```
convertible–to–type(type, etype, env) ≡
  if is–h_type_set(etype) then
    if is–h_type_set(type) then
      ∀t′ ∈ type ∃ t ∈ etype : convert–type–to–type(t′, t, env)
    else
      ∃ t ∈ etype : convert–type–to–type(type, t, env)
  else
    if is–h_type_set(type) then
      ∀t ∈ type convert–type–to–type(t, etype, env)
    else
      convert–type–to–type(type, etype, env)
```

Checking conversions

Previously, we have described conversions from one type or value to another type. Since conversion are only statable for one type, we must ensure, when checking a type conversion, that the expression is only associated with one type and not with a list of several types. This is done by the following algorithm.

Since the conversion defines the context of the subexpression we resolve the subexpression like in an assignment but the check for conversion replaces the matching between the two phases.

The function "check–conversion" checks whether a conversion from an expression to a given type is possible.		
INPUT:	*conv*	h_conversion
OUTPUT:	**Boolean**	

check–conversion$(conv, env) \equiv$
  let $conv =$ mk–h_conversion$(type, arg)$ **in**
  let $etype =$ resolve–subprog–phase1$(\langle arg \rangle, env)$ **in**
    if len $etype \neq 1$ **then**
      false
    else if is–h_value$(etype[1]) \vee$ is–h_types$(etype[1])$ **then**
      convertible–to–type$(type, etype[1], env) \wedge$
      resolve–subprog–phase2$(\langle arg \rangle, \{\langle etype[1] \rangle\}, env)$
    else if is–(h_types$^+$)–**Set**$(etype[1])$ **then**
      let $etypes = \left\{ e \in etype[1] \;\middle|\; (\textbf{len}\ e = 1) \wedge \right.$
        $\left.$ convertible–to–type$(type, e[1], env) \right\}$ **in**
      let $context = \{\langle t \rangle \mid t \in etypes\}$ **in**
        resolve–subprog–phase2$(\langle arg \rangle, \{context\}, env)$
    else
      false

4.8 Functions for expressions

This section presents the various algorithms that deal with types and values of arbitrary expressions.

4.8.1 Static expressions

Static expressions are expressions that may be evaluated at compile time, i.e. in general constants, values (literals and aggregates), and combinations using predefined operators. This description omits the combinations and explains only how we deal with constants and values. The functions may be extended later.

The first algorithms answer the question whether an expression is static. The function is straight forward.

The function "expr–is–static" checks whether an expression is static.		
INPUT:	*expr*	expr
OUTPUT:	**Boolean**	

expr–is–static$(expr, env) \equiv$
  is–literal$(expr) \vee$ value–is–aggregate$(expr) \vee$ name–is–constant$(expr, env)$

The function "exprlist–is–static" checks whether an expression list is static.		
INPUT:	*exprs*	expr_list
OUTPUT:	**Boolean**	

exprlist–is–static$(exprs, env) \equiv$
  if $exprs = \langle\rangle$ **then**
    true
  else
    if hd $exprs =$ mk–range_set$(type, lub, upb) \wedge$
        expr–is–static$(lub, env) \wedge$ expr–is–static(upb, env) **then**
      exprlist–is–static(**tl** $exprs, env$)
    else
      expr–is–static(**hd** $exprs, env) \wedge$ exprlist–is–static(**tl** $exprs, env$)

Next, we discuss the computation of static expressions. Regarding one expression we return the value in case of a literal or an aggregate, and the associated value in the case of a constant.

The function "eval–expr" computes the value of a static expression.		
INPUT:	*expr*	expr
OUTPUT:	h_value*	

eval–expr$(expr, env) \equiv$
  if is–literal$(expr) \vee$ value–is–aggregate$(expr)$ **then**

$\langle$hvalue$(expr, env)\rangle$
else if name–is–constant$(expr, env)$ **then**
$\langle$name–to–hvalue$(expr, env)\rangle$
else
error

If an expression list is static a list of values is returned. Only ranges need special handling.

The function "exprs–to–values" transforms a static expression list (which may include ranges) into a tuple of values.		
INPUT:	*exprs*	expr_list
OUTPUT:	h_value*	

exprs–to–values$(exprs, env) \equiv$
if $exprs = \langle\rangle$ **then**
$\langle\rangle$
else
let $env =$ mk–env$(\ldots, mapdecls)$ **in**
if hd $exprs =$ mk–range_set$(type, lub, upb)$ **then**
let $\langle d\rangle = mapdecls(type)$ **in**
get–values$(d,$ eval–expr$(lub, env),$ eval–expr$(upb, env))$^
exprs–to–values(**tl** $exprs, env$)
else
eval–expr(**hd** $exprs, env$)^exprs–to–values(**tl** $exprs, env$)

4.8.2 Types of basic expressions

In Section 4.5 expressions with function calls were discussed where the resolution algorithm computes the types of subprogram calls. In case of basic expressions the computation of expressions is simple. We distinguish the following cases:

- If the expression is a value like a literal or an aggregation then the value is transformed to the data structure hvalue and returned.
- If the expression is a membership test the boolean type is returned.
- In the case of a type assertion, the assertion declares which type the subexpression should have. If this is a correct context of the subexpression the asserted type is returned, otherwise an error is raised.

- Otherwise the expression is either a variable, a conversion, a subprogram or a subprogram call. Subprogram calls are not processed by this function since they are not basic expressions. They must be resolved by the function resolve–expr. Therefore, an error is raised if this function is called with a subprogram call. We consider the different cases.
  - Is the expression a dispatching call an error is raised (see above).
  - If the expression is a variable the declared type is returned.
  - In the case of a type conversion it has to be checked whether the subexpression is convertible to the conversion type. If this is true the conversion type is returned, otherwise an error is raised.
  - If the expression is a subprogram all subprograms with the given name are returned.

The function "get–basic–expr–types" computes the tuple of types (and values) corresponding to the given expression.		
INPUT:	*expr*	expr
OUTPUT:	h_types$^+$ | h_value | h_subprog–**Set**	

get–basic–expr–types($expr, env$) $\equiv$
  if is–literal($expr$) $\vee$ is–aggregate($expr$) $\vee$
      is–aggregate_extension($expr$) **then**
    hvalue($expr, env$)
  else
    cases $expr$:
      mk–membership_test(...) $\longrightarrow$
        mk–h_type("*boolean*", $\langle\rangle$)
      mk–type_assertion($types, expr'$) $\longrightarrow$
        let $n =$ **len** $types$ **in**
        let $atypes = \langle$ name–to–htype($types(1), [], env$), ...,
                        name–to–htype($types(n), [], env$)$\rangle$ **in**
          if resolve–exprs($atypes, expr', env$) $\neq$ **nil** **then**
            $atypes$
          else
            error
      mk–name_or_func_call($name, dispatch$) $\longrightarrow$
        if $dispatch$ **then**
          error
        else if name–is–var($name, env$) **then**

```
    get–var–type(name–to–hvar(name, env), env)
  else if name–is–conversion(name, env) then
    let conv = name–to–hconversion(name, env)in
      if check–conversion(conv, env) then
        s–Type(conv)
      else
        error
  else if name–is–subprog(name, env) then
    name–to–hsubprog–set(name, env)
  else
    error
```

4.8.3 Functions for right–hand side variables

In this section, we introduce the function to compute the type of an indexed, sliced, or normal variable. In order to do this, we define a more general function to access the partial definition of composite types — get–types–in–composite. This function is reused in the next section for variables that are used on the left side of assignments.

Then, the type of arbitrary right–hand side expressions is computed by the following function.

The function "get–var–type" returns the type of an (indexed) variable.		
INPUT:	*hvar*	h_var
OUTPUT:	h_types^+	

```
get–var–type(hvar, env) ≡
  let hvar = mk–h_var(name, type, indices, slice) in
    get–types–in–composite(type, indices, slice, false , env)
```

In the computation of the type we have to consider the following cases.

- If a variable is neither indexed normally nor indexed as a slice, then the type of the variable is the return value of the function.
- In case of an indexed variable that was declared as a type set/class, we distinguish the different cases for left–hand side and right–hand side expressions. In the left–hand side case we want to make sure that an assignment to all possible instances of the variable is valid. That means that we return the cut of all possible types. In the right–hand side case we want to include all possible runtime types, i.e. we return the union of all types.

- If the type is declared by a type declaration where simply a new name is given to an already existing type, then we recurse with the type used in the declaration.
- If the variable is sliced then it is necessary in this version of the algorithm that the indices in the slice are static. All other cases are refused. Nevertheless, there are also other circumstances under which we might accept an input but then more savvy algorithms are necessary.
- If the variable is indexed normally then we distinguish two different cases.
  - The expression for the index is static, i.e. we can compute the index at compile time and return the according type.
  - The expression is not static. Then, we compute the possible index values and check whether there is one unique type for all those indices in the composite definition.

The function "get–types–in–composite" returns the type at an index in a composite type.		
INPUT:	*type*	h_types or h_rep
	indices	h_value*
	slice	h_value*
	left	**Boolean**
OUTPUT:	h_types+	

get–types–in–composite$(type, indices, slice, left, env) \equiv$
  if $(indices = \langle\rangle) \wedge (slice = \langle\rangle)$ **then**
    $\langle type \rangle$
  else if is–h_type_set$(type)$ **then**
    let $s = \left\{\text{get–types–in–composite}(t, indices, slice, left, env) \;\middle|\; t \in type\right\}$ **in**
      if $\exists t, t' \in s$: **len** $t \neq$ **len** t' **then**
        error
      else
        let $\{n\} = \left\{\textbf{len}\ t \;\middle|\; t \in s\right\}$ **in**
          if $left$ **then**
            $\left\langle \bigcap_{t \in s} \text{set}(t[1]), \ldots, \bigcap_{t \in s} \text{set}(t[n]) \right\rangle$
          else
            $\left\langle \bigcup_{t \in s} \text{set}(t[1]), \ldots, \bigcup_{t \in s} \text{set}(t[n]) \right\rangle$
  else if is–h_a_type$(type)$ **then**
    let $rep = \text{get–rep}(type, env)$ **in**

```
      get–types–in–composite(rep, indices, slice, left, env)
  else if is–h_composite(type) then
    let map = make–visible–map(type, env) in
      if indices = ⟨⟩ then
        if exprlist–is–static(slice, env) then
          let vals = exprs–to–values(slice, env) in
            conc ⟨map(vals[1]), ..., map(vals[n])⟩
        else
          error  – –too restrictive!
      else
        let index = hd indices in
          if expr–is–static(index, env) then
            let t = eval–expr(index, env)in
              if t ∈ dom map then
                get–types–in–composite(map(t), tl indices, slice, left, env)
              else
                error
          else
            let etype = resolve–subprog–phase1(⟨index⟩, env) in
              if is–h_type(etype) then
                let env = mk–env(..., mapdecls) in
                let ⟨d⟩ = mapdecls(etype) in
                vals = elems get–values(d, "min", "max") in
                  if vals ⊆ dom map ∧
                       ∃t ∈ h_type∀v ∈ vals : map(v) = t then
                    get–types–in–composite(t, tl indices, slice, left, env)
                  else
                    error
              else
                error
  else
    error
```

The function "make–visible–map" computes the mapping of all those indices which are visible to their associated types.

The function "make–visible–map" returns the visible mapping of a composite type.		
INPUT:	*type*	h_type or h_anonym
OUTPUT:	h_values $\longrightarrow$ h_types	

make–visible–map($type, env$) $\equiv$
  let mk–h_composite($visible, private$) = get–rep($type, env$) **in**
    if is–h_type( ) $\wedge$ imported–nonprivate(s–Name($type$), env) **then**
      $visible$
    else
      $visible \cup private$

4.8.4 Variables on left–hand side of assignments

If an expression is used on the left–hand side of an assignment then we need to make sure that nothing else but variables are used in the expression. As we have already discussed in the previous section, indexed variables require special handling too.

The function "get–leftside–types" returns the types of the variables on the left side.		
INPUT:	*exprs*	expr_list
OUTPUT:	(h_types$^+$)*	

get–leftside–types($exprs, env$) $\equiv$
  if $exprs = \langle\rangle$ **then**
    $\langle\rangle$
  else
    let $expr$ = **hd** $exprs$ **in**
      if name–is–var($expr, env$) **then**
        let name–to–hvar($expr, env$) = mk–h_var($name, type, indices, slice$) **in**
          $\langle$get–types–in–composite($type, indices, slice$, **true** , env)$\rangle^\frown$
          get–leftside–types(**tl** $exprs, env$)
      else
        error

4.9 Resolution of classes

This section discusses how type classes, type sets, subprogram classes, and subprogram sets are resolved. In this version of Hoopla no sets of classes are used as elements of sets of classes but this may be extended easily.

4.9.1 Types

In order to resolve sets the types in the initialization of the type set are transformed into the internal representation and inserted in the set.

The function "name–to–htypeset" maps the name of a type set to an instance of "h_type_set".		
INPUT:	*name*	selected_name
	genmap	Ident $\longrightarrow$ h_types (an optional mapping for generic arguments in the outer scope)
OUTPUT:	*h_type*–**Set**	

$$
\begin{array}{l}
\text{name–to–htypeset}(name, genmap, env) \equiv \\
\quad \textbf{let } env = \text{mk–env}(\ldots, mapdecls) \textbf{ in} \\
\quad \textbf{let } mapdecls(name) = \text{mk–types\_decl}(\ldots, types, \ldots) \textbf{ in} \\
\qquad \text{htypes}(types, genmap, env)
\end{array}
$$

In the case of type classes it is not enough to consider the types in the initialization. We also have to search all the declarations for extensions of the type class and insert those types in adequate extensions to the type set associated with the class at the current position in the program.

The function "resolve–type–class" computes the set of "h_type" which is at this point of the program equivalent to the class.		
INPUT:	*name*	selected_name
	genmap	Ident $\longrightarrow$ h_types (an optional mapping for generic arguments in the outer scope)
OUTPUT:	h_type–**Set**	

$$
\begin{array}{l}
\text{resolve–type–class}(name, genmap, env) \equiv \\
\quad \textbf{let } env = \text{mk–env}(decls, mapdecls) \textbf{ in} \\
\quad \textbf{let } mapdecls(name) = \langle \text{mk–class\_decl}(\ldots, types) \rangle \textbf{ in} \\
\quad \textbf{let } withset = \left\{ with \;\middle|\; \begin{array}{l} d \in \textbf{elems } decls \wedge d = \text{mk–extend\_decl}(names, with) \wedge \\ name \in \textbf{elems } names \end{array} \right\} \textbf{ in} \\
\qquad \text{htypes}(types, genmap, env) \cup \bigcup_{w \in withset} \bigcup_{t \in \textbf{elems} w} \text{set}(\text{htypes}(t, [], env))
\end{array}
$$

4.9.2 Subprograms

The description of the algorithms for resolution of subprogram sets and subprogram classes are similar to type sets and type classes and straight forward.

The function "resolve–subprog–class–set" computes the declarations in a subprogram class or a subprogram set.		
INPUT:	*subprog*	h_subprog
OUTPUT:	h_subprog–**Set**	

resolve–subprog–class–set($subprog, env$) $\equiv$
  let $decl$ = s–Decl($subprog$) **in**
    if is–subprog_class($subprog$) **then**
      resolve–subprog–class($subprog, env$)
    else if is–subprog_set($subprog$) **then**
      resolve–subprog–set($subprog, env$)
    else
      $\{subprog\}$

The function "resolve–subprog–set" computes the declarations in a subprogram set.		
INPUT:	*subprog*	h_subprog
OUTPUT:	h_subprog–**Set**	

resolve–subprog–set($subprog, env$) $\equiv$
  let $decl$ = s–Decl($subprog$) **in**
  let $init$ = s–Init($decl$) **in**
  let $subprogs = \bigcup_{i \in init}$ name–to–hsubprog–set(i, env) **in**
    if $\exists s \in subprogs$: is–subprog_class(s–Decl(s)) **then**
      error
    else
      $subprogs$

The function "resolve–subprog–class" returns the instances of "h_subprog" that are maybe applicable.		
INPUT:	*subprclass*	h_subprog
OUTPUT:	h_subprog–**Set**	

resolve–subprog–class($subprclass, env$) $\equiv$
  let env = mk–env($decls, \ldots$) **in**
  let $subprclass$ = mk–h_subprog($decl, \ldots$) **in**
  let $decl$ = mk–subprog_class($name, \ldots, init$) **in**

$$\textbf{let } withset = \left\{ with \;\middle|\; \begin{array}{l} d \in \textbf{elems } decls \;\wedge\; d = \text{mk–extend\_decl}(names, with) \wedge \\ name \in \textbf{elems } names \end{array} \right\} \textbf{ in}$$

$$\bigcup_{i \in init} \text{name–to–hsubprog–set}(i, env) \;\cup\; \bigcup_{with \in withset} \bigcup_{w \in \textbf{elems} with} \text{name–to–hsubprog–set}(w, env)$$

Chapter 5

Implementation and testing

Several reasons that are discussed later in more detail prevented a complete implementation of the type checking algorithm. Therefore, I decided to develop a prototype for a reasonable subtask which demonstrates the doability of the type–checker and a possible implementation technique. This chapter is organized as follows. Section 5.1 deals with the general design of the type–checker. Section 5.2 discusses the reasons which led to the prototype and the subtask the prototype is covering. Section 5.3 gives more technical details of the implementation. Finally, Section 5.4 deals with the tests of the prototype and Section 5.5 with a simulation of the completeness and uniqueness test.

5.1 Design

When I started to design the type–checker for the front–end of the "Hoopla" compiler there were basically two different possibilities how the type–checker could be realized and linked with the already existing parts of the front–end. The two approaches are

- an *attribute grammar* where several attributes are defined for the nodes of the abstract syntax tree (AST), and several rules define how those attributes are computed from other attributes at the node itself, its parent, and its children. Here, it is necessary to identify suitable attributes in the algorithms of the formal description and adapt the algorithms for the computation of those attributes.
- a *hand–written solution* where a traversal procedure gets the AST as input, processes it, and sets the new information in the tree as a side–effect. In general, it should be possible to implement the actual type checking algorithm only using the internal data structures and interface functions to extract and set information in the AST. Then, the algorithm would be adaptable to new versions of the AST very easily.

There are different advantages and disadvantages for both methods. Since the existing parts of the front–end are written using the Cocktail tools [GE90] I favoured the solution

with the attribute grammar. The reasons for this decision have been compatibility with the parser, simple extension of the AST with new attributes, and automatic generation of the traversal procedure. A disadvantage is that very often artifices are necessary to pass the needed information to the right place in the tree, where a hand–written solution can be more elegant.

The design of the attribute grammar is implied in a straightforward manner by the algorithms in the main part of this thesis. As attributes variants of the data structures described in Section 4.1 are used. The interface functions are used directly to fill those data structures from the AST.

Figure 5.1 presents a small example which demonstrates how the attributes are used in the resolution algorithm to implement the previously described algorithms. To keep the example easy to survey, the AST was slightly simplified and only the attributes of interest are displayed. Besides the attributes and the evaluation order, the functions of the previous chapter are noted which correspond to the attribute computation. The AST is displayed with thick lines and the attributes are written on the right side besides the AST nodes. The arrows indicate how the different attributes are computed. The nodes labeled "n_o_f_c" refer to nodes name_or_func_call. In attribute computations with two target attributes, the attributes are written in one line. In phase 1 for each node `n_o_f_c` it is determined whether the node refers to a subprogram call and stored in the attribute `is_func`. This information is used to compute the possible types of all expressions which are passed bottom–up using the attributes `types1` and `typesL1`. Then, at the assign statement the types of the right–hand side are matched to the types of the left–hand side, and the flag `unambiguous` is set according to the result of the matching. The types of the left–hand side are copied to the right–hand side into the attribute `context`. This attribute is passed top–down while the correct subprogram is computed in the attribute `oksubs`. The attribute `unambiguous` indicates whether the resolution of the subprogram call was successful.

5.2 Realized prototype

The decision to implement a prototype instead of the complete type–checker was urged by several different reasons.

- Many changes in the language design during the first two months of the thesis caused a lot of fundamental modifications in the design of the type–checker. Moreover, the design of the type–checker used up more time than planned. Especially the merging of the requirements of the different language concepts turned out to be more complex than expected. This led to a very stringent time–table for the remaining parts of the thesis.
- Since the design of the language advanced meanwhile, there are now slightly changed requirements in the new version. Moreover, even fundamental changes in the AST

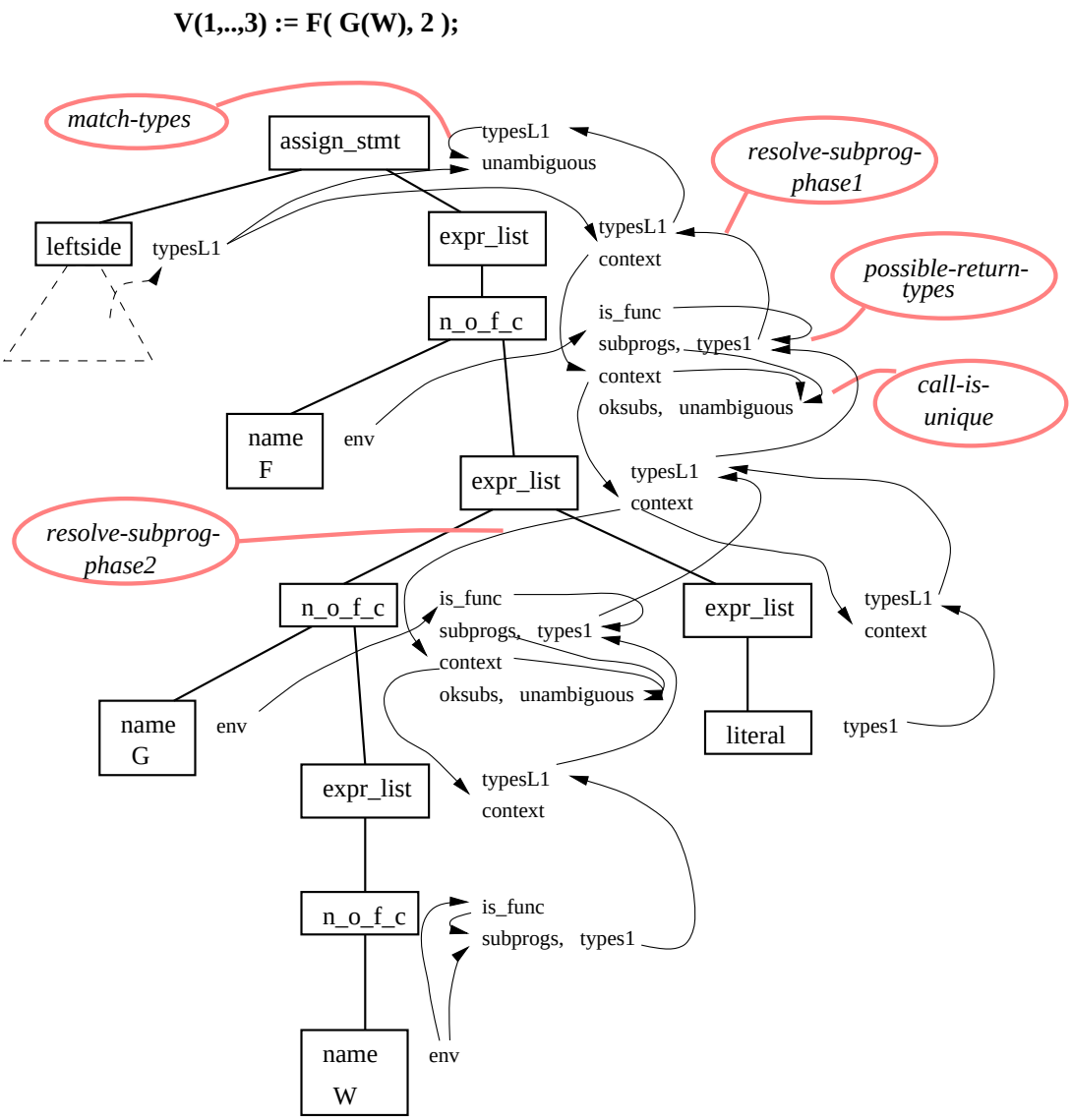

Figure 5.1: Example demonstrating the attribute grammar

were carried through. Therefore, the reuse of the code written for the type–checker for the first Hoopla version was barely possible – it does not make sense to implement the complete algorithms for an obsolete version of the language. Rather a demonstration of the doability and the methods is necessary.

- Finally, several technical difficulties handicapped the implementation. For example, the preliminary version of the AST has too many nodes as well as some properties that complicated the implementation of the type checker. Furthermore, there were several problems with the existing front–end of this version like incorrect writing and reading routines for ASTs.

As a conclusion, rapid prototyping is the only possibility to get a partially working type–checker before the deadline of the thesis.

The implemented prototype realizes the new two–phase algorithm with matching providing full functionality for overloaded functions with arbitrary numbers of return values. As types only normal, non–generic types were used so far. For generic types the conformity checks are not implemented yet. Furthermore, the mechanism to resolve explicitly dispatching calls and the completeness and uniqueness checks have been omitted so far.

The used data structures and the design is general enough that the omitted parts of the algorithm may be filled in.

5.3 Some details

As implementation language Ada was given. Since the parser is written with the parser generator "Lalr" [GV88] and the abstract syntax tree with the generator for abstract syntax trees "Ast" [Gro91a], which are parts of the Cocktail tools, the usage of the attribute evaluator generator "Ag" [Gro89] is implied in order to define additional attributes and evaluate them.

The data structures for the type checking information are defined using generic list and set packages. In addition, a data structure for mappings is defined which is general enough to handle range sets, too.

Approximatly 4,600 lines of code have been written in order to implement the prototype of the resultion algorithm. Nevertheless, most functions to create and access the internal data structures are written. They provide either already now the full functionality or should be easily extendable.

5.4 Tests

The implementation has been tested with several very small test programs in order to verify to a certain degree the functionality of the type–checker. Of course the small number of

tests is not a sufficient criterion on the correctness of the implementation, but the test programs cover the different possible errors.

The different tests are printed and explained in the following. `x` and `y` are types and `xy` is a type set consisting of both types.

```
function put(a: x) return (x)          -- put1
function put(a:x; b:x) return (x)      -- put2
v: x

v := put(put(v),v);
```

Table 5.1: Test program 1

The test program displayed in Table 5.1 is correct with regard to type conformity. The outer subprogram call is resolved to `put2` and the inner to `put1`.

```
function put(a: x) return ( xy )       -- put1
function put(a: xy ) return ( x )      -- put2
v: x
w: xy

w := put(put(v));
```

Table 5.2: Test program 2

The test program displayed in Table 5.2 is ambiguous. Both subprograms are applicable but none of them is more special than the other. The following error messages are produced.

```
20:9: Error       Ambiguous Subprogram call
20:13: Error       Ambiguous Subprogram call
```

The test program displayed in Table 5.3 is correct with regard to type conformity. The two inner calls are resolved to `get2` and the outer call is resolved to `put2`.

The test program displayed in Table 5.4 is ambiguous. The outer call is resolved successful to `put2`, but in the two inner calls there are two different possibilities how they may be resolved to `get1` and `get2`. The following error messages are produced.

```
19:20: Error       Ambiguous Subprogram call
19:13: Error       Ambiguous Subprogram call
```

```
function put(a: x) return ( x )                      -- put1
function put(a: x; b: x; c: x; d: x ) return ( x )   -- put2
function get(a: x) return ( x )                      -- get1
function get(a: x ) return (x;x)                     -- get2
v: x

v := put(get(v),get(v));
```

Table 5.3: Test program 3

```
function put(a: x) return ( x )                -- put1
function put(a: x; b: x; c: x ) return ( x )   -- put2
function get(a: x) return ( x )                -- get1
function get(a: x ) return (x;x)               -- get2
v: x

v := put(get(v),get(v));
```

Table 5.4: Test program 4

```
function K(a: x) return ( y )                    -- K1
function K(a: y) return ( x )                    -- K2
function I(a: xy; b: x) return ( x; xy )         -- I1
function I(a: y; b: x) return ( x )              -- I2
function I(a: xy; b: x) return (x; x )           -- I3
function H(a: xy ) return ( x; y )               -- H1
function H(a: y ) return ( y )                   -- H2
function G(a: xy; b: xy; c: xy ) return ( xy )   -- G1
function G(a: y; b: x; c: x ) return ( x )       -- G2
v: x
w: y
z: xy

z := G(H(w),I(K(v),v));
```

Table 5.5: Test program 5

The test program displayed in Table 5.5 is correct with regard to type conformity. G is resolved to G2 since G2 is more special than G1. H is resolved to H2 and I is resolved to I3 since I3 is more special than I1. K is resolved to K1.

```
function K(a: x) return ( y )                    -- K1
function K(a: y) return ( x )                    -- K2
function I(a: xy; b: x) return ( x; xy )         -- I1
function I(a: y; b: x) return ( x )              -- I2
function I(a: xy; b: x) return (x; x )           -- I3
function H(a: xy ) return ( y; x )               -- H1
function H(a: y ) return ( y )                   -- H2
function G(a: xy; b: xy; c: xy ) return ( xy )   -- G1
function G(a: y; b: x; c: x ) return ( x )       -- G2
v: x
w: y
z: xy

z := G(H(w),I(K(v),v));
```

Table 5.6: Test program 6

The test program displayed in Table 5.6 is ambiguous since there are three different possibilities for the calls H and I that match the parameters of both subprograms with name G: H2 and I1, H2 and I3, and H1 and I2. The following error messages have been produced.

```
28:16: Error       Ambiguous Subprogram call
28:18: Error       Ambiguous Subprogram call
28:11: Error       Ambiguous Subprogram call
```

5.5 Simulation of completeness and uniqueness check

Since it was not possible to implement the complete type–checker with the completeness and uniqueness check, I have implemented a small simulation for those checks in order to get a sense of the runtime.

Several problem sets have been created automatically with a separate program. The problem sets have the following form:

- the signatures and subprograms have three arguments

- the arguments are associated with either a type set of 10 types or a type set of 30 types.
- the number of subprograms implementing the subprogram set varies between 6 and 72

The program checked those problems for completeness and uniqueness.

In the simulation program all the data is stored in bit arrays. Implementation language is C. The experiments were run on one processor of the IBM SP–2. All the measured times are wall–clock times in seconds and not CPU times.

Since unsuccessful checks need less time than successful checks, all following results only refer to complete and unique problem sets. The results are displayed in Figure 5.2 and Figure 5.3.

For fairly small problems (10 types) the checks needed less than 0.5 Seconds. This is in an acceptable range to be used in practice. For larger problem sets all measured time needed more than 3 Seconds. The runtime for the problem set with 30 types and 72 subprograms is almost 25 seconds. This indicates that those checks may become a major part of the compile time in a Hoopla compiler. These are inacceptable runtimes for commercial compilers.

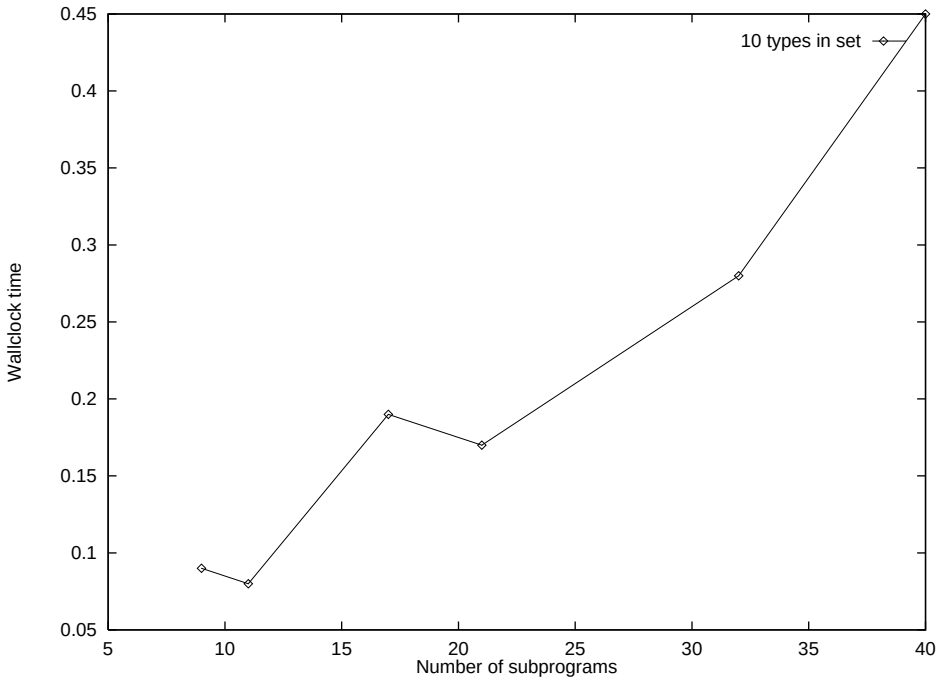

Figure 5.2: Measured time for sets with 10 types

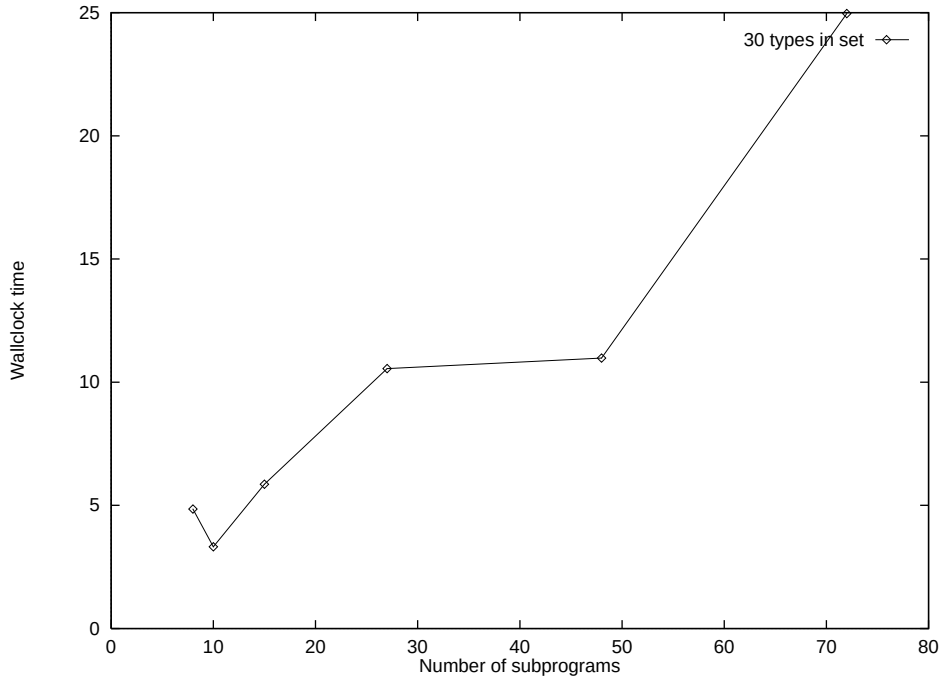

Figure 5.3: Measured time for sets with 30 types

Chapter 6

Conclusion and outlook

Finally, in this chapter, I want to summarize a few conclusive remarks and the contribution of this thesis. Furthermore, an outlook is given on how the type checking algorithm is impaired by the changes of the upcoming new version of Hoopla.

6.1 Contribution

This thesis shows the feasibility of the combination of several concepts in a type–checker. Although this has been proved only partially in practice, the formal specification using VDM shows the practicability pretty well.

The concept of type sets and classes is obviously a worthwhile and useful approach to subtyping. Note, that this concept does not increase the complexity of standard type checking algorithms.

Also functions returning more than one return parameter and multiple assignments are proved to be feasible. But in combination with overloading the complexity increases since all possible combinations must be considered and matched against the context. Without overloading, i.e. if one subprogram name refers to only one subprogram name, type checking may be accomplished by a traditional type checking algorithm. Only the combination of overloading and functions with varying numbers of return values increases the complexity. In my opinion, the benefits of the combination of both concepts are not as big as the costs of checking it. An equivalently powerful language can be reached by omitting one of both concepts. If multiple return values are excluded the same functionality can be reached for example by in/out parameters like in Ada, but they are easier to type check.

Last, the static type checking of subprogram classes and subprogram sets has been proved to be possible. This has been done be adapting the check for signatures in the programming language Cecil. In addition to the completeness and uniqueness check of subprogram sets, a possible solution has been proposed how explicitly dispatching calls may

be checked by the type–checker by computing the hull of those applicable subprograms. Since the proposed approach is very time–costly – the power set of all subprograms is considered – an alternative checking algorithm, which is more restrictive than the proposed one, is discussed briefly.

6.2 Outlook

Finally, I want to give an outlook on the new version of Hoopla and how the changes and new features influence the type–checker.

One of the concepts being responsible for the high complexity of the resolution algorithm is eliminated in the latest version: overloading is not possible anymore and therefore each subprogram name is unique. But as discussed below this does not reduce the complexity or simplifies the type checking algorithm.

The subprogram sets and classes from the first Hoopla version are now called *subprogram sets with a user contract* where the signature defines the interface of the subprogram set. They are checked as usual for completeness and uniqueness. In addition the new feature of *subprogram sets without user contract* are introduced which means that there is no signature given for the set. Then, the same mechanism is necessary like previously in explicitly dispatching calls. Since those sets may contain subprograms with arbitrary numbers of parameters and return values, the problem that is discussed in Section 4.5.6 is still valid – the modified version of Hoopla still needs the modified 2–phase algorithm with matching since the traditional algorithm is not powerful enough to deal with those problems.

Last, a new syntax and semantic has been developed for generic subprograms. But this should not cause any problems since only the binding of generic arguments is done implicitly now instead of explicitly.

Appendix A

Vienna development method

In order to describe the syntax of hoopla and the type–checking algorithms, the Vienna development method (VDM) was selected. VDM is equivalent to the Lambda–calculus but its notation is more readable and much easier to understand. The following short introduction into VDM is based on the introduction in [LS80]. Only functions were omitted because they are not necessary for our purpose. A more detailed reference manual may be found in [Jon78].

- The definition of the meta–language VDM is completely based on the first–order predicate calculus. The following logical notations are used in this paper.

true	truth value "true"
false	truth value "false"
$\wedge$	and
$\vee$	or
$\Rightarrow$	implies
$\Leftrightarrow$	equivalence
$\neg$	not
$\forall$	for all
$\exists$	there exists
$\exists!$	there exists exactly one
ι	the unique objects that fulfills the succeeding predicate

- A–**Set** denotes the power set of a domain A, i.e. all finite subsets with objects of the domain A. There are the standard sets **Boolean** $= \{$**true** , **false** $\}$, **Integer** $= \{\ldots, -1, 0, 1, \ldots\}$, and **Quotation** which contains all finite strings. Moreover, $\emptyset$ denotes the empty set. The following operations are defined on sets.

$S = S'$	comparison for equality of S and S'
$S \cup S'$	union of S and S'
$S \cap S'$	intersection of S and S'
$S \setminus S'$	set difference of S and S'
$S \subseteq S'$	comparison if S is a subset of or equal to S'
$S \subset S'$	comparison if S is a proper subset of S'
card S	returns the number of elements in S
$\mathcal{P}(S)$	returns the power set of S

- A^* or A^+ denote all ordered, finite lists (tuples) with objects of a domain A (with or without the empty list $\langle\rangle$). List instances may be denoted explicitly (e.g. $t = \langle a_1, a_2, \ldots, a_n\rangle, a_i \in A$) or implicitly (e.g. by a function $G : B \longrightarrow A$ in $\langle G(i) | i \in \{1, \ldots, n\}\rangle$). Moreover, the following operations are defined:

len t	returns the length of the tuple
elems t	returns the set of all elements of t
hd t	returns the first element of t
tl t	returns t without the first element
$t\hat{}t'$	concatenates the tuples t and t'
conc tl	concatenates the elements of a tuple of tuples
$t = t'$	comparison for equality of t and t'
$t \neq t'$	comparison for inequality of t and t'
$t[i]$	selects the i-th single instance of the tuple
ind t	returns the index set of the tuple
$t + [i \rightarrow a]$	replaces the i-th single instance of t by a

- In order to provide structures that correspond to nodes of abstract syntax trees the notation of a tree is introduced: $D_1 \ldots D_n$. It combines different, exclusively selectable sub–objects of domains $D_1, \ldots, D_n$. A tree may be provided with a name and labels for the sub–objects (e.g.$D :: L_1 : D_1 \ldots L_n : D_n$). Single tree instances may be constructed with a tree constructor mk–$D(d_1, \ldots, d_n)$ with $d_i \in D_i$. The sub–objects of a tree may be selected with s–$L_i(d) = d_i$ where $d =$ mk–$D(d_1, \ldots, d_n) \in D$ and L_i is the i-th label of D. Furthermore, the key word **nil** denotes an undefined (empty) tree. Then, square brackets are a notation for a tree that may be empty ($[D] = D \cup \{$**nil** $\}$).

- $A \xrightarrow{m} B$ denotes the set of all maps from a finite subset of domain A to domain B. Single instances may be denoted explicitly (e.g. $m = [a_1 \rightarrow b_1, \ldots, a_n \rightarrow b_n]$) or implicitly using a predicate P (e.g. $m = [a \rightarrow b \mid a \in A, b \in B, P(a, b)]$). Moreover, the following operations are defined for maps m and m' and the set s.

dom m	returns the domain set of m
rng m	returns the range set of m
$m(a)$	returns the element belonging to a, if $a \in$ **dom** m
$m \setminus s$	$[a \rightarrow m(a) \mid a \in (\textbf{dom}\ m \setminus s)]$
$m \mid s$	$[a \rightarrow m(a) \mid a \in (\textbf{dom}\ m \cap s)]$
$m + m'$	returns m' extended with $m \setminus (\textbf{dom}\ m \cap \textbf{dom}\ m')$
$m \cup m'$	returns the union of m and m', if **dom** $m \cap$ **dom** $m' = \emptyset$
$m \circ m'$	$[a \rightarrow m(m'(a)) \mid a \in \textbf{dom}\ m']$, if **rng** $m' \subseteq$ **dom** m

For easier readability of a VDM specification, the following notations are introduced.

- Local abbreviations may be defined for a scope. The range of the abbreviation is given by indentation. Example: **let** $cons = v$ **in** $(\ldots)$

- Conditional branches:

  - **if** < predicate > **then** < stmt1 > **else** < stmt2 >
  - **cases** < expression >:
    < expression1 > $\longrightarrow$ < stmt1 >,
    $\vdots$
    < expression(n) > $\longrightarrow$ < stmt(n) >

With the above data structures, the abstract syntax tree of Hoopla is described using syntax rules. As already pointed out above, a tree may be given a name with ::. In addition, it is possible to give rules for alternatives where the name is followed by a = and then the different alternatives are separated be | (like in BNF). For all names D of those rules, there is a predicate is–$D(d)$, which computes whether d is an element of domain D.

The algorithms for the complete type–checker are defined by predicates for the nodes of the abstract syntax tree. In those algorithms the key word **error** indicates that something went wrong and an undescribed error handling takes place.

Appendix B

Abstract syntax tree

This appendix presents the abstract syntax tree (AST) which was used for the implementation of Hoopla. The specification of the AST was written in the Vienna Development Method (VDM). All the type checking algorithms in this thesis are written on the specified AST.

In order to keep the type checking algorithms simple a few slight simplifications were made to the AST – primarily to have a flatter and more readable tree. The different sections point the changes out.

B.1 Compilation units

There are three different compilation units: the specification of a package, its implementation, and free subprograms. The type checking algorithms refer always to just one compilation unit where for imported parts from other packages the ASTs of imported specification units are linked into the AST.

```
package_spec    :: PackageName: Ident
                   Context: context_clause*
                   SpecDecl: spec_declaration*
package_body    :: PackageName: Ident
                   Context: context_clause*
                   BodyDecl: declaration*
```

The units for packages contain the package's name, a context clause with import statements, and the declarations of the package. Declarations in specifications may be declared as private in addition, i.e. they are not visible when the package is imported from another package.

```
spec_declaration    :: Decl: declaration
                       IsPrivate: Boolean
```

Free subprograms may be declared as normal subprogram or as subprogram set consistion of several other subprograms.

```
subprog_def         :: Context: context_clause*
                       Clause: subprog_clause
                       Body: body_declaration*
                       StmtList: stmt*
subprog_set         :: Context: context_clause*
                       Clause: subprog_clause
                       Init: inst_name*
                       IsClass: Boolean
```

Each subprogram may have a context too. The clause is the signature of the subprogram. In case of normal subprograms there are additional declarations for the body and a list of statements that define the functionality of the subprogram. In case of subprogram sets there is an initialization of the set and a flag that specifies whether it is a subprogram set or a class.

B.2 Import of packages

If declarations from other packages are imported there are two possibilities: the "from" declaration where just some declarations are imported from another package, and the "use" declaration where all declarations from other packages are made visible.

```
context_clause      = FromContext_clause | UseContext_clause
FromContext_clause  :: Import: Ident*
                       FromId: Ident
                       IsPrivate: Boolean
UseContext_clause   :: IdList: Ident*
                       IsPrivate: Boolean
```

The attribute "IsPrivate" defines whether private declarations or private parts of declarations are visible in a restricted manner.

B.3 Declarations

This section presents the nodes that represent declarations in an AST. Several different language constructs in Hoopla were united in single nodes in the AST. Therefore, there are values for the different attributes that are prohibited by the grammar in some context.

```
declaration      :: subprog_def | types_decl | class_decl|
                    type_decl | subprog_clause | subprog_class |
                    var_decl | forward_type_decl |extend_decl
```

The real abstract syntax tree of the current version distinguishes among the declarations in specifications and those allowed in bodies.

Declaration of type sets and type classes: Type sets and type classes are declared similarly. They can be defined by the name of another type set/class, a set of types, and additional modification like union or set difference.

```
types_decl       :: Name: Ident
                    Types: types_def
class_decl       :: Name: Ident
                    Types: types_def
types_def        = type_name | type_set | mod_types_def
mod_types_def    :: Right: types_def
                    Left: types_def
                    Sign: Ident
```

Declaration of types: A type declaration may contain besides the name of the type and the representation, generic formals, a flag indicating whether the declared type is constant, and a flag indicating whether the representation is private. There are two different possibilities to declare a type's representation which are described below in more detail.

```
type_decl        :: Name: Ident
                    GenericFormals: generic_formal*
                    IsConstant: Boolean
                    RepClause: rep_clause
                    Ispriv_rep_clause: Boolean
rep_clause       = inherit_rep | mod_rep_clause
```

A generic formal consists of an identifier, the possible types for the formal, and a flag indicating whether just exactly one type is possible for the formal in an instantiation.

```
generic_formal      :: Name: Ident
                       Types: types
                       IsType: Boolean
```

One possibility to declare a type is using a type definition with modifications like union or set difference. In the concrete grammar, this is only possible for types declared by sets of values (enumeration types).

```
inherit_rep         = type_def | rec_inherit_rep
rec_inherit_rep     :: Leftside: inherit_rep
                       Rightside: inherit_rep
                       Sign: Ident
```

A second possibility are type declarations with more sophisticated modifications for composite types. Type modifications may consist of omission or the replacement of certain parts. Different so specified composite types may be merged by with clauses in which a flag expresses whether this part of the type is private.

```
mod_rep_clause      :: Modification: type_modification
                       WithClauses: with_clause*
type_modification   :: Type: type_def
                       Without: expr_list
                       ReplaceList: replace_clause_list
                       – if Type = value-list: without Without and ReplaceList
replace_clause_list :: Replace: types*
                       By: types*
with_clause         :: Modification: type_modification
                       IsPrivate: Boolean
```

A type definition consists either of a set of values (specified in an expression list), a type name, or in case of a composite type where each component is specified by a mapping from an index to a type.

```
type_def            = values | type_name
values              = expr_list | map_def_list
map_def_list        :: list: map_def*
map_def             :: Indices: expr_list
                       TargetType: var_binding
```

Declaration of subprograms: Subprograms are declared by their signatures, and consist of a name, optional generic parameters, normal call parameters, and return parameters.

```
subprog_clause   :: Name: defining_name
                    Generics: generic_formal*
                    Params: formal*
                    Returns: formal*
formal           = formal_var | subprog_clause
formal_var       :: Name: Ident
                    Type: types
                    Default: expr_list
                    IsConstant: Boolean
```

The call and return parameters contain a name, and the possible types. In case of return parameters there might be a default value. In case of call parameters there might be a flag whether the call argument is treated like a constant. Call parameters may be subprogram declarations too.

Declaration of subprogram classes: Subprogram classes are declared like subprograms using a signature but in addition an initialization with subprogram names is given.

```
subprog_class    :: Name: defining_name
                    Generics: generic_formal*
                    Params: formal*
                    Returns: formal*
                    Init: inst_name–Set
```

Declaration of variables: In variable declarations the variable name is associated with a binding where there are two different possible bindings.

```
var_decl         :: Names: Ident*
                    Binding: var_binding
var_binding      = binding_to_type | binding_to_value
```

First, the variable may be bound to a named type or type set.

```
binding_to_type  :: Type: types
                    Init: expr_list
                    IsConstant: Boolean
```

types	= type_def | type_set | mod_types_def
type_set	:: Set: *type*–**Set**
type	= type_name | values

There might be an additional initialization and a flag whether it is a constant. These nodes are slightly different from the real AST because we have introduced a redundant type_set in the declaration of types.

Second, the variable might be a named value which is realized by the following definition.

binding_to_value	:: Values: expr_list

Forward declaration of types: The forward declaration is straight forward.

forward_type_decl	:: Name: Ident GenericFormals: generic_formal*

Extension of classes: Classes may be extended by the following node, where a list of names is added to some classes denoted by their names.

extend_decl	:: Name: inst_name+ With: inst_name+

B.4 Names

There are several different possibilities to refer to names.

name	= selected_name | indexed_name
selected_name	:: [Package: Ident] Name: simple_name
indexed_name	:: Name: name Index: expr
inst_name	= selected_name | generic_instantiation
generic_instantiation	:: Name: selected_name GenericActuals: types*
simple_name	:: Name: Ident
defining_name	= Ident
type_name	= inst_name
Ident	= **Quotation**

"selected_name" provides the possibility to refer to a declaration in another package.
"indexed_name" has additional indices.
And in "inst_name", the generic parameters of a type or subprogram are bound to types.

B.5 Statements

There are the following AST nodes for statements in Hoopla. Most of them are self–explanatory.

```
stmt            = null_stmt | if_stmt | case_stmt | while_loop_stmt |
                  until_loop_stmt | loop_stmt | for_loop_stmt | exit_stmt |
                  proc_call | assign_stmt | return_stmt
if_stmt         :: Expr: expr
                   ThenPart: stmt*
                   [ Elsifs: elsif* ]
                   ElsePart: stmt*
elsif           :: Expr: expr
                   Stmts: stmt*
case_stmt       :: Expr: expr
                   Branches: branch*
                   OthersBranch: stmt*
branch          :: Case: types
                   Stmts: stmt_list
while_loop_stmt :: Expr: expr
                   Stmts: stmt*
until_loop_stmt :: Expr: expr
                   Stmts: stmt*
loop_stmt       :: Stmts: stmt*
for_loop_stmt   :: Ident: T_Ident
                   Values: expr_list
                   IsSet: boolean
                   Stmts: stmt*
assign_stmt     :: LeftSide: parameterized_name*
                   Exprs: expr*
return_stmt     :: Exprs: expr_list
```

The flag "IsSet" in the for–loop shows whether the values refer to a list of expressions or type names.

In case of procedure calls there are normal subprogram calls as well as calls of generic instantiated subprograms.

```
proc_call              = parameterized_name | instantiated_name
parameterized_name     :: Name: name
                          Exprs: expr_list
instantiated_name      :: GenInst: generic_instantiation
                          Exprs: expr_list
```

B.6 Expressions

Normal single expressions are defined by the following nodes. They consist of values like literals and aggregates, variables which are contained in "name_or_func_call", and additional operators like functions, assertions, and membership tests.

```
expr                   = literal | membership_test | type_assertion |
                          aggregate | aggregate_extension | name_or_func_call
literal                = num | char_constant
num                    :: Value: Integer
char_constant          = Ident
membership_test        :: Expr: expr
                          Type: types
type_assertion         :: Types: type_name*
                          Expr: expr
aggregate              :: Associations: association*
association            :: Exprs: expr_list
                          NewVals: expr_list
aggregate_extension    :: Left: aggregate_reduction
                          With: aggregate_reduction*
aggregate_reduction    :: Expr: expr
                          Without: expr_list
name_or_func_call      :: Name: name_in_expr
                          Dispatch: Boolean
name_in_expr           = parameterized_name | instantiated_name
```

"range_set" defines a range.
"membership_test" checks, whether "Expr" is contained in "Type".
"type_assertion" is an assertion for the compiler that "Expr" is of type "Type".
"aggregate" generates instances of composite types.

```
expr_list              :: List: exprEl*
exprEl                 = range_set | expr
range_set              :: Type: type_name
```

Lub: expr
Upb: expr

B.7 Environment

In order to formulate the algorithms, an environment was introduced which is used for locating declarations for names.

env :: Decls: Declaration*
Mapdecls: Ident $\xrightarrow{m}$ Declaration*

This declaration was simplified since the topic of the thesis are the type checking algorithms and not the technical details how declarations are located.

Appendix C

Interface functions

This appendix presents the functions that are used to extract information from the AST and store the information in the data structures used in the algorithms. Furthermore, functions are presented which allow a simpler access of the data structures. Since the focus of this work is on the type checking algorithms and most of the functions are straight forward they are not discussed as thoroughly as the algorithms in the previous chapters.

C.1 Assumptions

The programming language Hoopla allows to split one program into different packages in order to maintain data abstraction and to increase the re–use of code. As a consequence it is necessary to import types, variables, and subprograms from other packages. Because the representation or some parts of the representation of a type may be declared private it is crucial for the type checker to know where a type is declared and if its structure is visible. This may be extracted from the abstract syntax trees of the compilation unit and the imported specifications. But in order to keep the type checking algorithms simple, I renounce the resolution algorithms for names and work with the following assumptions.

All types have unique names. To obtain this, all names declared externally are assumed to be identified with the package name in front of the actual name. This does not help with the problem of the visibility of a type's representation, because it is possible to import a type with private representation in such a way that the representation is restricted visible nevertheless. Therefore, in addition a predicate is assumed which checks for type names whether the representation is visible.

The function "imported–private" checks whether the type was imported in such a way that the private representation is visible.		
INPUT:	*typename*	selected_name or generic_instantiation
OUTPUT:	boolean	

imported–private($typename, env$) $\equiv$

...

The function "imported–nonprivate" checks whether the type was imported and the private parts are hidden.		
INPUT:	*typename*	selected_name or generic_instantiation
OUTPUT:	boolean	

imported–nonprivate($typename, env$) $\equiv$

...

The function "get–values" returns a tuple of the values in the given range of a type.		
INPUT:	*decl*	type_decl
	lower	h_value
	upper	h_value
OUTPUT:	h_value*	

get–values($decl, lower, upper$) $\equiv$

...

C.2 Functions to resolve names

Names may be used to label types, type sets, classes, variables, named constants, subprograms, and subprogram classes. To free the further description of the algorithms from the complicated resolution of names, in this section functions are introduced to deal with name resolution.

Names of types

The function "name–is–type" checks whether a name denotes any visible declared type.		
INPUT:	*name*	any possible name
OUTPUT:	**Boolean**	

name–is–type($name, env$) $\equiv$

let env = mk–env(..., $mapdecls$) **in**
  if is–selected_name($name$) **then**
    if $name$ = "Any" **then**
      true
    else
      let $decls = mapdecls(name)$ **in**
        (**len** $decls = 1$) $\wedge$ is–type_decl($decls[1]$)
  else if $name$ = mk–generic_instantiation($name'$, $genericactuals$) **then**
    let $decls = mapdecls(name')$ **in**
      (**len** $decls = 1$) $\wedge$ is–type_decl($decls[1]$) $\wedge$
      generic–conformity($decls[1]$, $genericactuals$, env)
  else
    false

The function "name–to–htype" maps the type name to an instance of "h_type".		
INPUT:	*name*	generic_instantiation or selected_name
	genmap	Ident $\longrightarrow$ h_types (an optional mapping for generic arguments in the outer scope)
OUTPUT:	h_type or "Any"	

name–to–htype($name$, $genmap$, env) $\equiv$
  cases $name$:
    mk–generic_instantiation($name'$, $genericactuals$) $\Longrightarrow$
      mk–h_type($name'$, gen–tuple($genericactuals$, $genmap$, env))
    mk–selected_name($name'$) $\Longrightarrow$
      if $name'$ = "Any"**then**
        "*Any*"
      else
        mk–h_type($name'$, $\langle\rangle$)

The function "gen–tuple" transforms generic actuals into a tuple of "h_types" under consideration of an outside generic mapping.		
INPUT:	*actuals*	types* (arguments of a generic instantiation)
	genmap	Ident $\longrightarrow$ h_types (an optional mapping for generic arguments in the outer scope)
OUTPUT:	h_types*	

gen–tuple$(actuals, genmap, env) \equiv$
  let $n =$ **len** $actuals$ **in**
    $\langle$htypes(apply–map$(actuals[1], genmap), [], env), \ldots$
      htypes(apply–map$(actuals[n], genmap), [], env)\rangle$

Names of typesets

The function "name–is–typeset" checks whether a name denotes any visible declared type set.		
INPUT:	*name*	any possible name
OUTPUT:	**Boolean**	

name–is–typeset$(name, env) \equiv$
  let $env =$ mk–env$(\ldots, mapdecls)$ **in**
    if is–selected_name$(name)$ **then**
      let $decls = mapdecls(name)$ **in**
        (**len** $decls = 1) \wedge$ is–types_decl$(decls[1])$
    else
      false

Names of type classes

The function "name–is–class" checks whether a name denotes any visible declared class of types.		
INPUT:	*name*	any possible name
OUTPUT:	**Boolean**	

name–is–class$(name, env) \equiv$
  let $env =$ mk–env$(\ldots, mapdecls)$ **in**
    if is–selected_name$(name)$ **then**
      let $decls = mapdecls(name)$ **in**
        (**len** $decls = 1) \wedge$ is–class_decl$(decls[1])$

else
false

Names of variables

The function "name–is–var" checks whether a name denotes any visible declared variable (without named constants). Indices are possible if the variable is an instance of any composite type.		
INPUT:	*name*	any possible name
OUTPUT:	**Boolean**	

name–is–var($name, env$) ≡
let env = mk–env(..., $mapdecls$) **in**
if $name$ = mk–parameterized_name($name', exprs$) **then**
let $decls = mapdecls$(get–basic–name($name'$)) **in**
(**len** $decls = 1$) ∧
$\Big(((decls[1] =$ mk–var_decl(..., $binding$)) ∧ is–binding_to_type($binding$)) ∨
is–formal_var($decls[1]$)$\Big)$
else
false

The function "get–basic–name" returns the base name of indexed names.		
INPUT:	*name*	indexed_name or selected_name
OUTPUT:	selected_name	

get–basic–name($name$) ≡
cases $name$:
mk–indexed_name($name'$, ...) ⟶ get–basic–name($name'$)
mk–selected_name(...) ⟶ $name$

The function "name–to–hvar" maps the name of a variable to an instance of "h_var".		
INPUT:	*name*	parameterized_name
OUTPUT:	h_var	

name–to–hvar($name, env$) ≡
let env = mk–env(..., $mapdecls$) **in**
let $name$ = mk–parameterized_name($name', exprs$) **in**

let $name''$ = get–basic–name($name'$) **in**
let $indices$ = get–indices($name', env$) **in**
if {mk–var_decl(..., $binding$)} = $mapdecls(name)$ **then**
let $binding$ = mk–binding–to–type($type$, ...) **in**
mk–h_var($name''$, htypes($type$, [], env), $indices, expr$)
else
let {mk–formal_var(..., $Type$, ...)} = $mapdecls(name)$ **in**
mk–h_var($name''$, htypes($Type$, [], env), $indices, expr$)

The function "get–indices" returns a tuple of literals and aggregates which are the indices in the name of a variable.		
INPUT:	*name*	indexed_name or selected_name
OUTPUT:	expr*	

get–indices($name, env$) ≡
cases $name$:
mk–indexed_name($name', index$) ⟶
get–indices($name', env$)^⟨$index$⟩
mk–selected_name(...) ⟶
⟨⟩

Names of constants

The function "name–is–constant" checks whether a name denotes any visible declared named constant.		
INPUT:	*name*	any possible name
OUTPUT:	**Boolean**	

name–is–constant($name, env$) ≡
let env = mk–env(..., $mapdecls$) **in**
if $name$ = mk–parameterized_name($name', exprs$) **then**
let $decls$ = $mapdecls$(get–basic–name($name'$)) **in**
(**len** $decls = 1$) ∧ ($decls[1]$ = mk–var_decl(..., $binding$)) ∧
is–binding_to_value($binding$)
else
false

The function "name–to–hvalue" transforms a named constant into the corresponding tuple of "h_value"s.		
INPUT:	*name*	parameterized_name
OUTPUT:	h_value$^+$	

name–to–hvalue($name, env$) $\equiv$
  let env = mk–env($\ldots, mapdecls$) **in**
  let $name$ = mk–parameterized_name($name', exprs$) **in**
  let $name''$ = get–basic–name($name'$) **in**
  let $indices$ = get–indices($name', env$) **in**
  let $slice$ = exprs–to–values($exprs, env$) **in**
  let $mapdecls(name'')$ = $\langle$mk–var_decl($\ldots, binding$)$\rangle$ **in**
  let $binding$ = mk–binding–to–value($value$) **in**
  let val = eval–expr($value, env$) **in**
    if $(indices = \langle\rangle) \wedge (slice = \langle\rangle)$ **then**
      $\langle val \rangle$
    else
      get–agg–at–index($val, indices, slice$)

The function "get–agg–at–index" computes the "h_value"(s) in an aggregate denoted by the indices and/or the indices of a slice. (At the moment only directly given indices in the aggregate are considered.)		
INPUT:	*value*	h_value
	indices	h_value*
	slice	h_value*
OUTPUT:	h_value$^+$	

get–agg–at–index($value, indices, slice$) $\equiv$
  if is–h_aggregate($value$) **then**
    let $value$ = mk–h_aggregate($values, types$) **in**
      if $indices \neq \langle\rangle$ **then**
        if hd $indices \in$ **dom** $values$ **then**
          let $val = values($**hd** $indices)$ **in**
            if tl $indices = \langle\rangle \wedge slice = \langle\rangle$ **then**
              $\langle val \rangle$
            else
              get–agg–at–index($val,$ **tl** $indices, slice$)
        else

error
else
let $n =$ **len** $slice$ **in**
conc $\Big\langle$get–agg–at–index$(value, slice[1], \langle\rangle), \ldots$
get–agg–at–index$(value, slice[n], \langle\rangle)\Big\rangle$
else
error

Names of subprograms

The function "name–is–subprogram" checks whether a name denotes any visible declared subprogram or subprogram class.		
INPUT:	*name*	any possible name
OUTPUT:	**Boolean**	

name–is–subprogram$(name, env) \equiv$
let $env =$ mk–env$(\ldots, mapdecls)$ **in**
cases *name* :
mk–generic_instantiation$(name', genericactuals) \Longrightarrow$
let $decls =$ **elems** $mapdecls(name')$ **in**
$\forall d \in decls :$ $\Big($is–subprog_clause$(d) \vee$ is–subprog_class$(d)) \vee$
is–subprog_def$(d) \vee$ is–subprog_set$(d)\Big)$
mk–selected_name$(\ldots) \longrightarrow$
let $decls =$ **elems** $mapdecls(name)$ **in**
$\forall d \in decls :$ is–subprog_clause$(d) \vee$ is–subprog_class$(d) \vee$
is–subprog_def$(d) \vee$ is–subprog_set(d)

The function "name–to–hsubprog–set" computes the set of subprog and subprog class declarations ("h_subprog") with the given name.		
INPUT:	*name*	generic_instantiation or selected_name
OUTPUT:	h_subprog–**Set**	

name–to–hsubprog–set$(name, env) \equiv$
let $env =$ mk–env$(\ldots, mapdecls)$ **in**
cases *name* :
mk–generic_instantiation$(name', genericactuals) \Longrightarrow$
let $decls = mapdecls(name')$ **in**

get–generic–conform–decls($decls, genericactuals, env$)
mk–selected_name($name'$) $\Longrightarrow$
let $decls = mapdecls(name')$ **in**
let $n =$ **len** $decls$ **in**
$\{$mk–h_subprog($decls[1], \langle\rangle$), ...,
mk–h_subprog($decls[n], \langle\rangle$)$\}$

The function "get–generic–conform–decls" checks for each given declaration whether it is conform to the generic actuals and returns the conform declarations.		
INPUT:	$decls$	(subprog_clause | subprog_def | subprog_set | subprog_class)*
	$gactuals$	types*
OUTPUT:	h_subprog–**Set**	

get–generic–conform–decls($decls, gactuals, env$) $\equiv$
let $n =$ **len** $decls$ **in**
$\{$mk–h_subprog($decls[i]$, make_subprog_gen_map($decls[i], gactuals, env$)) |
$(1 \leq i \leq n) \wedge$ generic–conformity($decls[i], gactuals, env$) $\}$

Subprogram calls

The function "name–is–subprogramcall" checks whether a name denotes a call of any visible declared subprogram or subprogram class ,i.e. with calling arguments.		
INPUT:	$name$	any possible name
OUTPUT:	**Boolean**	

name–is–subprogramcall($name, env$) $\equiv$
if $name =$ mk–parameterized_name($name', exprs$) **then**
if is–selected_name($name'$) **then**
name–is–subprogram($name', env$)
else
if $exprs = \langle\rangle$ **then**
let $name' =$ mk–indexed_name($name'', index$) **in**
if is–selected_name($name''$) **then**
name–is–subprogram($name'', env$)
else

```
            false
        else
          false
    else if name = mk–instantiated_name(name′, exprs) then
      name–is–subprogram(name′, env)
    else
      false
```

The function "name–to–hsubprogcall" transforms a name with its arguments into an instance of "h_subprog_call".		
INPUT:	*name*	parameterized_name or instantiated_name
	dispatch	**Boolean**
OUTPUT:	h_subprog_call	

```
name–to–hsubprogcall(name, dispatch, env) ≡
  cases name :
    mk–parameterized_name(name′, exprs) ⟶
      if is–selected_name(name′) then
        mk–h_subprog_call(name′, exprs, dispatch)
      else
        if exprs = ⟨⟩ then
          let name′ = mk–indexed_name(name″, index) in
            if is–selected_name(name″) then
              mk–h_subprog_call(name″, ⟨index⟩, dispatch)
            else
              error
        else
          error
    mk–instantiated_name(name′, exprs) ⟶
      mk–h_subprog_call(name′, exprs, dispatch)
```

Type conversions

The function "name–is–conversion" checks whether a name denotes a type conversion.		
INPUT:	*name*	any possible name
OUTPUT:	**Boolean**	

```
name–is–conversion(name, env) ≡
```

```
cases name :
  mk–parameterized_name(name′, exprs) ⟶
    if is–selected_name(name′) then
      false
    else
      if exprs = ⟨⟩ then
        let name′ = mk–indexed_name(name″, index) in
          if is–selected_name(name″) then
            name–is–type(name″, env)
          else
            false
      else
        false
  mk–instantiated_name(name′, exprs) ⟶
    let types = resolve–subprog–phase1(exprs, env) in
      (len types = 1) ∧ is–h_type(types[1])
```

The function "name–to–hconversion" transforms a name with its arguments into an instance of "h_conversion".		
INPUT:	*name*	parameterized_name or instantiated_name
OUTPUT:	h_conversion	

```
name–to–hconversion(name, env) ≡
  cases name :
    mk–parameterized_name(name′, exprs) ⟶
      let name′ = mk–indexed_name(name″, index) in
        mk–h_conversion(name″, index)
    mk–instantiated_name(name′, exprs) ⟶
      mk–h_conversion(name′, hd exprs)
```

C.3 Functions to create "h_types"

Generic parameters cause different problems in the resolution of names since in a specific scope each generic parameter must be replaced by the actual type of the instantiation. Therefore, we introduce the mapping of generic formals. Very often composite types or arguments of generic subprograms are defined in terms of some generic formal. To obtain the instantiated type the generic formal has to be replaced by the generic argument. This

is done with a mapping which is passed through a type expression. Such a mapping is used as an argument for the following function which transforms an arbitrary input into the internal data structure for types.

Now, a function "htypes" can be defined which transforms any type or type set of the abstract syntax tree into a "h_types". Then all the checks of conformity are defined using the "h_type" notation. In the function, the argument "genmap" realizes the mapping of generic arguments to the instantiated type as described above.		
INPUT:	t	type_set, values, generic_instantiation, selected_name, or mod_types_def
	genmap	Ident $\longrightarrow$ h_types (an optional mapping for generic arguments in the outer scope)
OUTPUT:	h_types (= h_type_set, h_type, h_anonym, or "Any")	

$$
\begin{array}{l}
\text{htypes}(t, genmap, env) \equiv \\
\quad \textbf{cases } t : \\
\qquad \text{mk–type\_set}(set) \Longrightarrow \\
\qquad\quad \left\{ s' \;\middle|\; (s \in set) \wedge \right. \\
\qquad\qquad \Big(\; (\text{name–is–type}(s, env) \;\wedge\; (s' = \text{name–to–htype}(s, genmap, env))) \vee \\
\qquad\qquad\quad (\neg\text{name–is–type}(s, env) \;\wedge\; (s' = \text{hrep}(s, genmap, env))) \Big) \Big\} \\
\qquad \text{mk–values}(\ldots) \Longrightarrow \\
\qquad\quad \textbf{if } \text{is–expr\_list}(t) \textbf{ then} \\
\qquad\qquad \textbf{if } \forall t' \in \textbf{elems } t : \text{name–is–type}(t', env) \textbf{ then} \\
\qquad\qquad\quad \left\{ \text{name–to–htype}(t', [], env) \;\middle|\; t' \in \textbf{elems } t \right\} \\
\qquad\qquad \textbf{else} \\
\qquad\qquad\quad \text{mk–h\_anonym}(\text{hrep}(t, genmap)) \\
\qquad\quad \textbf{else} \\
\qquad\qquad \textbf{error} \\
\qquad \text{mk–generic\_instantiation}(\ldots) \Longrightarrow \\
\qquad\quad \textbf{if } \text{name–is–type}(t, env) \textbf{ then} \\
\qquad\qquad \text{name–to–htype}(t, genmap, env) \\
\qquad\quad \textbf{else} \\
\qquad\qquad \textbf{error} \\
\qquad \text{mk–selected\_name}(\ldots) \Longrightarrow \\
\qquad\quad \textbf{if } \text{name–is–typeset}(t, env) \textbf{ then} \\
\qquad\qquad \text{name–to–htypeset}(t, genmap, env) \\
\qquad\quad \textbf{else if } \text{name–is–class}(name, env) \textbf{ then} \\
\qquad\qquad \text{resolve–class}(name, genmap, env)
\end{array}
$$

else if name–is–type(t, env) **then**
name–to–htype$(t, genmap, env)$
else
error
mk–mod_types_def$(left, right, sign) \Longrightarrow$
let $l =$ htypes$(left, genmap, env)$ **in**
let $r =$ htypes$(right, genmap, env)$ **in**
if is–h_type_set$(l) \wedge$ is–h_type_set(r) **then**
cases *sign* :
" + " $\longrightarrow l \cup r$
" – " $\longrightarrow l \setminus r$
else if is–h_anonym$(l) \wedge$ is–h_anonym$(r) \wedge$
is–h_valueset(get–rep$(l, env)) \wedge$ is–h_valueset(get–rep(r, env)) **then**
cases *sign* :
" + " $\longrightarrow$ mk–h_anonym(get–rep$(l, env) \cup$ get–rep(r, env))
" – " $\longrightarrow$ mk–h_anonym(get–rep$(l, env) \setminus$ get–rep(r, env))
else
error

C.4 Functions to create and to access "h_rep"

These functions transform an AST node representing the internal structure of a type into the data structure used by the type checking algorithm. Two additional functions the access the representations of arbitrary type names are given.

The function "hrep" transforms the representation of any type into the internal structure "h_rep".		
INPUT:	*rep*	selected_name, generic_instantiation, expr_list, map_def_list, rec_inherit_rep, or mod_rep_clause
	genmap	Ident $\longrightarrow$ h_types (an optional mapping for generic arguments in the outer scope)
OUTPUT:	h_rep	

hrep$(rep, genmap, env) \equiv$
let $env =$ mk–env$(\ldots, mapdecls)$ **in**
cases *rep* :
mk–selected_rep$(\ldots) \longrightarrow$
if $rep \in$ **dom** *genmap* **then**

```
    genmap(rep)
  else
    let decls = mapdecls(rep) in
      if (len decls = 1) ∧ is–type_decl(decls[1]) then
        hrep(s–rep_clause(decls[1]), [ ]
      else
        error
mk–generic_instantiation(...) ⟶
  let decls = mapdecls(rep) in
    if (len decls = 1) ∧ is–type_decl(decls[1]) then
      hrep(s–rep_clause(decls[1]), [ ]
    else
      error
mk–expr_list(list) ⟶ elems exprs–to–values(list, env)
mk–map_def_list(list) ⟶
  mk–h_composite(ind–to–type(list, genmap, env), [])
mk–rec_inherit_rep(left, right, sign) ⟶
  let l = hrep(left, genmap, env) in
  let r = hrep(right, genmap, env) in
    if is–h_value_set(l) ∧ is–h_value_set(r) then
      cases sign :
        " + " ⟶ l ∪ r
        " – " ⟶ l \ r
    else
      error
mk–mod_rep_clause(modification, withclauses) ⟶
  let a = get–visible–map(modification, genmap, env) in
  let b = get–private–map(modification, genmap, env) in
  let b′ = get–private–map(withclauses, genmap, env) in
  let a′ = get–visible–map(withclauses, genmap, env) in
    mk–h_composite(a ∪ a′, b ∪ b′)
```

The function "ind–to–type" transforms the list of map_def's of the definition of a composite type into a mapping from indices to other types. Those types may be anonymous.		
INPUT:	*list*	map_def*
	genmap	Ident $\longrightarrow$ h_types (an optional mapping for generic arguments in the outer scope)
OUTPUT:	h_value $\longrightarrow$ h_types	

ind–to–type($list, genmap, env$) $\equiv$
  if $list \neq \langle\rangle$ **then**
    let hd $list$ = mk–map_def($indices, targettype$) **in**
    let s = **elems** exprs–to–values($indices, env$) **in**
      ind–to–type(**tl** $list, genmap$) $\cup \bigcup_{i \in s}$ [$i \longrightarrow$ htypes($targettype, genmap, env$)]
  else []

The function "get–visible–map" computes the visible part of the declaration of a composite type.		
INPUT:	*part*	type_modification, with_clause, or with_clause*
	genmap	Ident $\longrightarrow$ h_types (an optional mapping for generic arguments in the outer scope)
OUTPUT:	h_value $\longrightarrow$ h_types	

get–visible–map($part, genmap, env$) $\equiv$
  if $part \in$ with_clause* **then**
    if $part = \langle\rangle$ **then**
      []
    else
      get–visible–map(**hd** $part, genmap, env$) $\cup$
      get–visible–map(**tl** $part, genmap, env$)
  else
    cases $part$:
      mk–type_modification($type, without, replacelist$) $\longrightarrow$
        let rep = get–rep–mod($type, genmap, env$) **in**
          if rep = mk–h_composite($visible, private$) **then**
            let $withoutind$ = **elems** exprs–to–values($without, env$) **in**
            let $map = visible \setminus withoutind$ **in**
            let $replacemap$ = make–replacemap($replacelist, genmap, env$) **in**
              $\left[a \rightarrow \text{apply–map}(map(a), replacemap) \;\middle|\; a \in \textbf{dom}\ map\right]$

```
        else
          error
      mk–with_clause(modification, isprivate) ⟶
        if isprivate then
          []
        else
          get–visible–map(modification, genmap, env)
```

The function "get–private–map" computes the private part of the declaration of a composite type.		
INPUT:	*part*	type_modification, with_clause, or with_clause*
	genmap	Ident ⟶ h_types (an optional mapping for generic arguments in the outer scope)
OUTPUT:	h_value ⟶ h_types	

```
get–private–map(part, genmap, env) ≡
  if part ∈ with_clause* then
    if part = ⟨⟩ then
      []
    else
      get–private–map(hd part, genmap, env) ∪
      get–private–map(tl part, genmap, env)
  else
    cases part :
      mk–type_modification(type, without, replacelist) ⟶
        let rep = get–rep–mod(type, genmap, env) in
          if rep = mk–h_composite(visible, private) then
            if (private ≠ []) ∧ is–type_name(type) ∧
                imported–nonprivate(type, env) then
              error
            else
              let withoutind = elems exprs–to–values(without, env) in
              let map = private \ withoutind in
              let replacemap = make–replacemap(replacelist, genmap, env) in
                [a → apply–map(map(a), replacemap) | a ∈ dom map]
          else
            error
```

```
mk–with_clause(modification, isprivate) ⟶
  if isprivate then
    get–visible–map(modification, genmap, env) ∪
    get–private–map(modification, genmap, env)
  else
    get–private–map(modification, genmap, env)
```

The function "get–rep–mod" transforms the type declaration in a type modification to the data structure h_rep.		
INPUT:	*type*	type_def
	genmap	Ident ⟶ h_types (an optional mapping for generic arguments in the outer scope)
OUTPUT:	h_rep	

```
get–rep–mod(type, genmap, env) ≡
  cases type :
    type_name ⟶
      hrep(type, [], env)
    map_def_list ⟶
      mk–ind_to_type(type, [], env), [])
```

The function "make–replacemap" gets a "replace_clause_list", like it is used in modified representation clauses, and transforms it into a mapping to replace target types.		
INPUT:	*replacelist*	replace_clause_list
	genmap	Ident ⟶ h_types (an optional mapping for generic arguments in the outer scope)
OUTPUT:	htypes ⟶ htypes	

```
make–replacemap(replacelist, genmap, env) ≡
  let replacelist = mk–replace_clause_list(replace, by) in
  let n = len replace in
    [ htypes(replace[i], genmap, env) ⟶
      htypes(by[i], genmap, env) | 1 ≤ i ≤ n ]
```

The function "get–rep" gets a type name or an anonymous type and returns the belonging representation of the type.		
INPUT:	*htype*	h_type or h_anonym
OUTPUT:	h_rep (if the representation is visible)	

get–rep($htype, env$) $\equiv$
  if $htype$ = mk–h_type($name, \ldots$) **then**
    let env = mk–env($\ldots, mapdecls$) **in**
    let $mapdecls(name)$ = mk–type_decl($\ldots, repclause, isprivrepclause$) **in**
      if $isprivrepclause \wedge$ imported–nonprivate($name, env$) **then**
        "*private*"
      else
        hrep($repclause, [\,], env$)
  else if $htype$ = mk–h_anonym($decl$) **then**
    $decl$
  else
    error

The function "get–rec–rep" returns the internal structure of a type name or anonymous type. If the type is defined in terms of another type name, the function recurses until it finds the representation.		
INPUT:	*htype*	h_type or h_anonym
OUTPUT:	h_rep	

get–rec–rep($htype, env$) $\equiv$
  let rep = get–rep($htype, env$) **in**
    if is–htype(rep) **then**
      get–rec–rep(rep, env)
    else
      rep

C.5 Functions to access h_subprog

These functions access the different parts of subprogram declarations in a convenient way.

The function "get–subprog–params–types" computes the parameter types of a subprogram.		
INPUT:	*subprog*	h_subprog
OUTPUT:	(h_types | h_subprog)*	

get–subprog–params–types($subprog, env$) ≡
  let $subprog$ = mk–h_subprog($decl, map$) **in**
    cases $decl$:
      mk–subprog_clause(..., $params$, ...) ⟶
        get–formals–types($params, map, env$)
      mk–subprog_def(..., $clause$, ...) ⟶
        let $clause$ = mk–subprog_clause(..., $params$, ...) **in**
          get–formals–types($params, map, env$)
      mk–subprog_set(..., $params$, ...) ⟶
        get–formals–types($params, map, env$)

The function "get–return–types" computes the return types of a subprogram.		
INPUT:	*subprog*	h_subprog
OUTPUT:	(h_types | h_subprog)*	

get–return–types($subprog, env$) ≡
  let $subprog$ = mk–h_subprog($decl, map$) **in**
    cases $decl$:
      mk–subprog_clause(..., $returns$) ⟶
        get–formals–types($returns, map, env$)
      mk–subprog_def(..., $clause$, ...) ⟶
        let $clause$ = mk–subprog_clause(..., $returns$) **in**
          get–formals–types($returns, map, env$)
      mk–subprog_set(..., $returns$) ⟶
        get–formals–types($returns, map, env$)

The function "get–formals–types" returns the types of a list of formals.		
INPUT:	$formals$	formal*
	$genmap$	Ident ⟶ h_types (an optional mapping of generic formals in an outer scope)
OUTPUT:	(h_types | h_subprog)*	

get–formals–types($formals, genmap, env$) ≡

if $formals = \langle\rangle$ **then**
  $\langle\rangle$
else
  if hd $formals$ = mk–formal_var(..., $type$, ...) **then**
    $\langle$htypes($type, genmap, env$)$\rangle$^get–formals–types(**tl** $formals, genmap, env$)
  else
    $\langle$mk–h_subprog(**hd** $formals, genmap$)$\rangle$^
      get–formals–types(**tl** $formals, genmap, env$)

C.6 Functions to generate generic mappings

Generic mapping are created and applied using the following functions.

The function "make–genmap" extracts the generic mapping inherently given in the internal representation of a type name.		
INPUT:	*type*	h_type or h_anonym
OUTPUT:	Ident $\longrightarrow$ h_types	

make–genmap($type, env$) $\equiv$
  cases *type* :
    mk–h_anonym(...) $\longrightarrow$ []
    mk–h_type($name, genargs$) $\longrightarrow$
      let env = mk–env(..., $mapdecls$) **in**
      let $mapdecls(name)$ = mk–type_decl(..., $genericformals$, ...) **in**
        $\left[\text{get–gen–formal–name}(genericformals(i)) \longrightarrow genargs(i) \;\middle|\; 1 \leq i \leq \textbf{len}\ genargs\right]$

The function "make–subprog–gen–map" extracts the generic mapping inherently given in the internal representation of a subprogram.		
INPUT:	*decl*	subprog_clause, subprog_class, subprog_set , subprog_def
	genargs	types*
OUTPUT:	Ident $\longrightarrow$ h_types	

make–subprog–gen–map($decl, genargs, env$) $\equiv$
  cases *decl* :
    mk–subprog_clause(..., $generics$, ...) $\longrightarrow$
      $\left[\text{get–gen–formal–name}(generics[i]) \rightarrow genargs[i]\right]$

$$\left[1 \leq i \leq \textbf{len}\ generics\right]$$

mk–subprog_def$(\ldots, clause, \ldots) \longrightarrow$
  let $clause =$ mk–subprog_clause$(\ldots, generics, \ldots)$ **in**
    $\left[\text{get–gen–formal–name}(generics[i]) \rightarrow genargs[i] \middle| 1 \leq i \leq \textbf{len}\ generics\right]$

mk–subprog_set$(\ldots, generics, \ldots) \longrightarrow$
  $\left[\text{get–gen–formal–name}(generics[i]) \rightarrow genargs[i] \middle| 1 \leq i \leq \textbf{len}\ generics\right]$

The function "get–gen–formal–name" returns the name of a generic formal.		
INPUT:	*formal*	generic_formal
OUTPUT:	Ident	

get–gen–formal–name$(formal) \equiv$
  let $formal =$ mk–generic_formal$(name, \ldots)$ **in**
    name

The function "apply–map" checks whether an element is in the domain of a mapping and returns either the element or the mapped element.		
INPUT:	*elem*	anything
	map	any mapping
OUTPUT:	depends on "elem" and the range of "map"	

apply–map$(elem, map) \equiv$
  if $elem \in$ **dom** map **then**
    $map(elem)$
  else
    elem

C.7 Functions to create "h_values"

Arbitrary values are transformed with this functions into the internal data structure of the type checking algorithm.

The function "hvalue" transforms literals and aggregates into the internal structure "h_value".		
INPUT:	*expr*	literal, aggregate, or aggregate_extension
OUTPUT:	h_value	

hvalue($expr, env$) $\equiv$
  if is–literal($expr$) **then**
    $expr$
  else if value–is–aggregate($expr$) **then**
    let $values$ = make–value–map–agg($expr, env$) **in**
    let $types$ = make–type–map–agg($expr, env$) **in**
      mk–h_aggregate($values, types$)
  else
    error

The function "make–type–map–agg" gets an aggregate and computes the mapping of those parts where only the types but not the values are known at compile time.		
INPUT:	*agg*	aggregate, aggregate_extension, or aggregate_reduction
OUTPUT:	h_value $\longrightarrow$ h_types	

make–type–map–agg(agg, env) $\equiv$
  if is–aggregate(agg) **then**
    $\square$
  else if agg = mk–aggregate_extension($left, with$) **then**
    make–type–map–agg($left, env$) $\cup \bigcup_{a \in with}$ make–type–map–agg(a, env)
  else if agg = mk–aggregate_reduction($expr, without$) **then**
    let $type$ = resolve–subprog–phase1($expr, env$) **in**
    let $withoutind$ = **elems** exprs–to–values($without, env$) **in**
      if $type$ = mk–h_aggregate($\ldots, types, \ldots$) **then**
        $types \setminus withoutind$
      else if is–h_type($type$) $\vee$ is–h_anonym($type$) **then**
        let rep = get–rec–rep($type, env$) **in**
          if rep = mk–h_composite($visible, private$) **then**
            if is–h_type($type$) $\wedge$ ($private \neq \square$) $\wedge$
                imported–nonprivate(s–Name($type$), env) **then**
              error

else
$(visible \cup private) \setminus withoutind$
else
error
else
error

The function "make–value–map–agg" gets an aggregate and returns the mapping of those parts which are already known at compile time.		
INPUT:	*agg*	aggregate, aggregate_extension, aggregate_reduction, association
OUTPUT:	h_value $\longrightarrow$ h_value	

make–value–map–agg$(agg, env) \equiv$
if $agg =$ mk–aggregate$(associations)$ **then**
$\bigcup_{a \in \mathbf{elems} associations}$ make–value–map–agg(a, env)
else if $agg =$ mk–aggregate_extension$(left, with)$ **then**
make–value–map–agg$(left, env) \cup \bigcup_{a \in \mathbf{elems} with}$ make–value–map–agg(a, env)
else if $agg =$ mk–aggregate_reduction$(expr, without)$ **then**
let $type =$ resolve–subprog–phase1$(expr, env)$ **in**
if $type =$ mk–h_aggregate$(values, \ldots)$ **then**
$values \setminus without$
else if is–h_type$(type) \vee$ is–h_anonym$(type)$ **then**
$\square$
else
error
else if $agg =$ mk–association$(exprs, newvals)$ **then**
let indices = exprs–to–values$(exprs)$ **in**
let values = exprs–to–values$(newvals, env)$ **in**
if (**len** $values = 1$) **then**
$\left[i \longrightarrow value \;\middle|\; i \in \mathbf{elems}\ indices \right]$
else
if len $indices =$ **len** $values$ **then**
$\left[indices[i] \longrightarrow values[i] \;\middle|\; 1 \leq i \leq \mathbf{len}\ indices \right]$
else
error

else
error

The function "value–is–aggregate" checks whether the given expression is an aggregate.		
INPUT:	*expr*	expr
OUTPUT:	boolean	

value–is–aggregate($expr$) $\equiv$
is–aggregate($expr$) $\vee$ is–aggregate_extension($expr$)

Appendix D

Function index

Bibliography

[ADL91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static type checking of multi-methods. In *OOPSLA'91*, pages 113–128. ACM, 1991.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bar94] J. G. P. Barnes. *Programming in Ada*. Addison-Wesley, fourth edition edition, 1994.

[Bjø80] Dines Bjørner, editor. *Towards a formal description of ADA*. Lecture notes in computer science ; 98. Springer, Berlin [u.a.], 1980. XII, 630 S.

[Cas] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3).

[CDG+92] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula–3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[Cha95] Craig Chambers. The Cecil language – Specification and rationale. Technical report, University of Washington, 1995.

[CL95] Craig Chambers and Gary T. Leavens. Type checking and modules for multi-methods. Technical Report TR #95-19, Department of Computer Science, Iowa State University, August 1995.

[Cli] Marshall Cline. C++ FAQs Lite.

[DT88] Scott Danforth and Chris Tomlinson. Type theories and object–oriented programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[eif] WWW Eiffel home page. http://www.eiffel.com/doc/eiffel.html.

[GE90] J. Grosch and H. Emmelmann. A tool box for compiler construction. Technical Report Compiler Generation Report No. 20, GMD Forschungsstelle an der Universität Karlsruhe, January 1990.

[GHK95] Harald Gall, Manfred Hauswirth, and Rene Klösch. Objektorientierte Konzepte in Smalltalk, C++, Objective–C, Eiffel und Modula–3. *Informatik Spektrum*, 18:195–202, 1995.

[Gro89] Josef Grosch. Ag – an attribute evaluator generator. Technical Report Compiler Generation Report No. 16, GMD Forschungsstelle an der Universität Karlsruhe, August 1989.

[Gro91a] Josef Grosch. Ast – a generator for abstract syntax trees. Technical Report Compiler Generation Report No. 15, GMD Forschungsstelle an der Universität Karlsruhe, September 1991.

[Gro91b] Josef Grosch. Puma – a generator for the transformation of attributed trees. Technical Report Compiler Generation Report No. 26, GMD Forschungsstelle an der Universität Karlsruhe, July 1991.

[GV88] J. Grosch and B. Vielsack. The parser generators lalr and ell. Technical Report Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, April 1988.

[Hol96] Bernd Holzmüller. Syntax und informelle Semantik der Sprache Hoopla, version 1.0. unpublished, June 1996.

[Joh86] Ralph E. Johnson. Type–checking smalltalk. In *OOPSLA '86*. ACM, September 1986.

[Jon78] Cliff B. Jones. The meta-language: A reference manual. In Dines Bjørner and Cliff B. Jones, editors, The *Vienna development method : the meta-language*, Lecture notes in computer science ; 61, pages 218–277. Springer, Berlin [u.a.], 1978.

[LS80] Winfried Lamersdorf and Joachim W. Schmidt. Specification of Pascal/R. The semantic specification method VDM. Technical Report 73, Universität Hamburg, Fachbereich Informatik, 1980.

[Mey] Bertrand Meyer. Static typing and other mysteries of life.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object–Oriented Type Systems*. John Wiley & Sons, 1994.

[Sch] Stephen Schaub. Modula–3 reference and tutorial.

[Str90] Bjarne Stroustrup. *The C++ Programming Language, Second edition*. Addison–Wesley, 1990.

[VHU92] Jan Vitek, R. Nigel Horspool, and James S. Uhl. Compile–time analysis of object–oriented programs. In *Proc. of the 4th International Conference on Compiler Construction*, pages 236–250. Springer, 1992.

[Wya94] Geoff Wyant. Introducing Modula–3. *Linux Journal*, December 1994.