University of Oslo
Department of Informatics

Compiling Creol Safely

Jørgen Hermanrud Fjeld

Master Thesis

2nd May 2005

Abstract

This thesis provides contributions to the research programming language Creol (Concurrent REflective Object-oriented Language). The first contribution is the EBNF grammar for Creol. The second contribution suggests how to extend the Creol language with functional constructs. The third and major contribution is the design of a type system for the Creol language, as well as some molding of the Creol language, such that static type safety is achieved. The fourth contribution is a prototype implementation of a compiler for Creol.

The Creol language has until now provided static type safety and separation between inheritance and subtyping by assumption only. The creation of the Creol type system investigates this assumption for the Creol language. During the process there has also been a clarification of the Creol language from a type system point of view. The type system designed for Creol is a hybrid between a structural and nominal type system, and is a step towards a novel hybrid type system, that facilitates a separation between inheritance and subtyping, while enforcing nominal constraints, when desireable.

The prototype compiler implemented for Creol is crafted with tools that operate on a higher level than traditional compiler tools. These high level approaches include combinator parsing and attribute grammars.

Acknowledgement

First of all, I would like to thank my supervisors Einar B. Johnsen and Bjarte M. Østvold. Johnsen for accepting me as a master student, greatly raising my textual standards and pulling my mindset in a formal direction. Østvold has relentlessly provided quick and accurate feedback, which as been invaluable. His demand for "horizontal lines" sparked the theoretical aspects of this thesis. Our conversations where of great inspiration, while his focus helped bring the thesis to a closure.

I'm in debt to my family and friends that supported me during the work. My girlfriend Sigrun for her support, patience and push for a laptop, it has proved very convenient and productive. My mother Tove for proofreading my English. My brother Matias for helping me understand my own thoughts. My father Inge for insisting on silly competitions with the sole purpose of my progression as well as providing a non-back-breaking chair. My friend Halvor for providing ego-boosting moral support.

The enthusiastic encouragement of Anders Moen inspired me to pursue my own ideas and focus on research as a joyful activity.

Finally, to all those I have not mentioned here, I'm grateful for all the support I have received, and for the general encouragement.

Contents

List of Figures

Chapter 1

Introduction

Creol is an object-oriented research language with novel approaches to concurrency, program analysis and formal semantics. Other areas of the language have not yet received much attention, but are accounted by intent or assumption, of which the following are of interest to this thesis, the assumed presence of static type checking, the choice of interfaces and classes as central and distinct concepts that facilitate a separation of inheritance and subtyping, the use of interfaces to type all object interaction, and the integration of functional and object-oriented programming in the same language. In light of these ideas, the goal of this thesis, is to contribute to the Creol language by answering the following questions.

- Is it possible to design a formal type system for Creol, which precisely describes type checking, with particular focus on object-orientation.
- Is it feasible to design a compiler for Creol, using high level tools, to easily facilitate language and compiler development?

1.1 The Creol Language

The language Creol [24] is an object-oriented research language with emphasis on distributed concurrent objects. The syntax and semantics of Creol is presented informally in Section 2, and partially in articles [55, 52, 56, 54] produced by the Creol research.

The Creol language research has contributed synchronous and asynchronous method invocation in a uniform manner within the same semantic framework. The choice of synchronous or asynchronous method invocation is entirely decided by the caller through annotation. The Creol language introduces concurrency by giving each object a separate thread. The Creol framework for method invocation where the caller decides synchronicity is a new twist in comparison to languages that offer either synchronous remote procedure calls (RPC), asynchronous events, or a combination of both, which leads to very complicated semantics. Processor release points are places in the code where the object may switch safely between different tasks. Processor release points facilitate reasoning about class invariants, and class invariants form a basis for analysing the behaviour of concurrent programs, hence processor release points are crucial to

the behaviour analysis of concurrent programs. It is safe to switch when class invariants are valid, therefore a processor release point must maintain the invariant. Thus an object may switch to other code in the object at a processor release point, while trusting the invariant. Invariants are not enforced by the type system, but rather by separate analysis of the code. The integration of invariants with the type system by a notion of behavioural subtyping requires theorem proving in the general case. The combined effect of one thread of control for each object with thread switch only on processor release points, is thread-safety in a concurrent and distributed environment. Although the actual implementation details of processor release points are unspecified, an implementation would necessarily resort to low level synchronisation primitives, such as those described by Andrews [5], but notice that given processor release points, it would be possible to model other concurrency models, such as concurrent access to a shared variable.

In order to understand the features of Creol, we place Creol on the language map by listing the major properties.

Object Oriented Objects provide an intuitive abstraction of the real world around us.

Class Based A class is a successful abstraction that allows programmers to comprehend and manage objects, demonstrated by the general success of object-oriented programming with class-based languages.

Functional Creol provides programmers with the right tool for the right job by additionally offering algebraic datatypes and functional constructs, which provide the programmer with a tool that is both object-oriented and functional.

Safe The guaranteed absence of untrapped errors. Untrapped errors are further explained in Section 4.

Statically Typed All type errors are caught at compile time. The definition of type errors is left to Section 4.

Interfaces Interfaces provide behavioural abstraction.

Inheritance Maximised code reuse and behaviour reuse provided through multiple inheritance for both classes and interfaces.

1.2 Creol Virtual Machine and Maude

The operational semantics of the Creol language are defined in rewriting logic, and give a high level specification of how Creol programs execute. The Creol operational semantics is executable by the rewriting logic tool Maude [66], which may be used as a language interpreter. This language interpreter is named the Creol Virtual Machine. The original interpreter was developed in the master thesis of Arnestad [8]. The direct execution of operational semantics, permits semantic changes to be tried out immediately, which gives a very short development cycle. This short development cycle enables rapid research results on the Creol language through experimentation with new ideas and different paradigms

from other languages. Consider for instance different evaluation strategies or scheduling strategies. The high level Creol Virtual Machine has greatly simplified code generation in the compiler. A low level virtual machine would require the generation of machine instructions, which in itself is a large and complex part of compiler construction. Since the Creol Virtual Machine is written as a rewriting logic module, the Creol Virtual Machine can itself be analyzed with the Maude framework. Furthermore Maude itself may be altered through Meta-Maude that allows advanced experiments with the Creol Virtual Machine. An example of such ongoing research is the use of predicates on communication history to guide the order of execution in the Creol Virtual Machine.

1.3 Creol Compiler and Type System Incentive

This section looks at incentives for the development of a Creol compiler. The Creol language has reached a maturity where it was deemed convenient to implement a compiler as a frontend to the Creol Virtual Machine. The reasons for creating a Creol compiler were as follows:

1. Creol machine code is assumed to be *type safe* and does not contain type information, hence the Creol Virtual Machine will not operate correctly if type errors are present in the code. Consequently type checking is needed and a Creol compiler must perform type checking. There is however no formal type system for the Creol language, hence it is unclear how type checking should be done. To successfully implement type checking, it is necessary to create a type system for the Creol language. There are no other languages with all the features of the Creol language, therefore it is unknown exactly how the Creol language can be statically type checked. This requires the careful investigation of how static type checking can be achieved for each of the features in the Creol language. Such investigation requires a formally defined type system.

2. There is presently no developed support for algebraic data types in neither the Creol language nor in the Creol virtual machine. The Creol language should provide algebraic data types.

3. Creol Machine Code was targeted to syntactically resemble Creol within the limits of Maude, and Creol was adapted to Creol Machine Code, making it feasible to hand-translate from Creol to Creol Machine Code. There are some problems with this:

   (a) The Creol Machine Code syntax is not machine generator friendly in the sense that there is a uniform syntax and that information is localised. As a result the Creol compiler is burdened with syntactic concerns in the Creol machine code that actually reflect user-friendliness for programmers using Creol machine code directly.

   (b) The Creol syntax has been adapted towards the Creol Machine Code, which further increases the language barrier for people with experience from other programming languages. A syntax closer to common object-oriented languages is more programmer friendly.

   (c) The Creol Machine Code contains syntactic sugar that complicates the specification, although syntactic sugar isn't a part of the Creol semantics. A compiler can handle syntactic sugar to simplify the Creol Virtual Machine.

4. There is a gap of abstraction between Creol programs and their Creol Machine Code representation. This creates the need for a compiler from Creol to CMC. Due to the active development of both Creol and its interpreter, the compiler should emphasize modularity and flexibility at both language and machine code levels and easily facilitate modifications of both Creol and CMC.

In summary this led to the need of a compiler to further research on Creol. This compiler should do parsing, type checking and generation of CMC. Furthermore it should be written using high level tools. Since Creol is for research, and not for production usage, neither the speed of the compiler nor the speed of compiled programs are essential, and pose little influence on the choice of tools.

1.4 Type Checking

Type checking improves the reliability of programs, by checking for consistency between information and operations on information. A static type checker analyses the program when it is compiled, and rejects the program if there is inconsistency, so the programmer is confronted with potential errors. This places the consequence of errors with responsible party. The exact definition of consistency is difficult. On one hand, one would like to prove that a program does its job and finishes. On the other hand, it is impossible to create a program that checks if any given program finishes, known as the *halting problem* [38, Sect. 8] [61, Sect. 5.3] [34].

Static type checking is concerned with those consistency checks that are decidable and can be solved mechanically, in a reasonable amount of time. Within these limits, static type checking can provide safety from untrapped errors, and a compile time guarantee that no type errors may occur at run-time. An untrapped error is an error that is not noticed by the program. A type error is an error decided just by looking at the types.

A type checked program is no more reliable than the type checker, so errors in the type checker is worse than errors in arbitrary programs. Type systems are used to formally describe type checking, such that the correctness of both the type system and the type checker, can be inspected and verified. A type system consists of a type language and type rules. The type language corresponds to information and operations on information. Type rules describe how to assign types, with the type language, to programs, as well as how to check if the assigned types are valid.

A type system bridges the gap between our understanding of consistency and the type checker, so the type system should be understandable for humans as well as possible to implement faithfully. This gap is narrowed by identifying good abstractions, which allows the type language and the type rules to be understandable as well as implementable.

1.5 Haskell

We chose to develop the prototype compiler, the Functional Creol Compiler, in Haskell [39], a statically typed non-strict functional programming language. The importance of such languages to rapid development is well documented [43]. Statically typed functional languages have a strong track record with respect to symbol manipulation [7], which is important for writing compilers. A non-strict language evaluates expressions only when needed, which avoids unnecessary computations, and facilitates techniques such as combinator parsing in a natural manner. Combinator parsing is heavily used in the Functional Creol Compiler.

Haskell is the default non-strict statically typed functional language, and is used in research environments throughout the world. The popularity of Haskell has fueled language research both on Haskell language itself, and in general with an implementation in Haskell. The research and work has contributed both traditional and new compiler tools written in Haskell, allowing this thesis to use a relatively novel approach. The other non-strict statically typed languages one might consider are Miranda [90] and Clean [21], but these languages have much smaller user communities and much less research associated with them, therefore there are less tools to choose from, less support to get from the community, and less people who may read the code of the Functional Creol Compiler. One might also consider a strict language from the ML family, such as SML [84] or O'Caml [73]; the latter has stable implementation and a decent community, but the documented benefits of non-strictness for rapid prototyping suggests otherwise [43].

Throughout this thesis the reader is assumed to have some basic familiarity with Haskell, including the practical use of monads to structure input and output. A monad is a construct that, from a practical perspective, allows functional code to be structured such that input and output can be expressed in an intuitive manner, which corresponds to how input and output are expressed in imperative languages. A gentle introduction to Haskell, including practical use of monads, may be found in a tutorial [41].

1.6 Combinator Parsers

The Functional Creol Compiler makes use of combinator parsing [45], which is an approach that differs from the traditional parser generator methods found in popular tools such as Yacc [101] and Bison [9]. The general idea of combinator parsing is to build parsers for small subsets of the language, and combine these parsers, forming a parser for a larger subset of the language. This process recursively defines a parser for the whole language. Combinators are different strategies for how parsers can work together. To get an intuition, imagine `h` is a parser for the word `hello` and `w` is a parser for the word `world`. Then we have the sequence combinator `<*>` and the parallel combinator `<|>`. We can now create a parser for `helloworld` by combining `h` and `w` with `<*>` to get `h <*> w`. To create a parser for either `hello` or `world` we use `h <|> w`. Notice how these combinations themselves are new parsers that can be combined further to stepwise create a complete and complex parser. This stepwise combination is the key to the simplicity of combinator parsers.

This thesis uses the specific combinator parsing approach described in the

introductory lectures notes [30] used at the University of Utrecht and in the corresponding library [88], which implements those parser combinators. The choice of combinator parsing was based on the author's experience with Lex and Yacc where the grammar for the compiler quickly becomes cluttered with code, hampering the readability. The error messages from Yacc/Bison when parsing fails does not suggest possibly correct alternatives, and the specification of custom error messages when parsing fails is difficult in Yacc/Bison. The University of Utrecht parser library generates standard error messages with information about possible further parses, thereby helping the programmer and alleviating the need for specifying such messages in detail, although the specification of custom error messages is straightforward. Furthermore the chosen combinator parser library also offers an attribute grammar such as those often found in the compiler literature [63, 97]. These, combined with the curiosity for trying a modern approach to crafting a compiler, were the decisive factors in choosing the University of Utrecht combinator parser and attribute grammar system.

1.7 Attribute Grammars

The Functional Creol Compiler uses attribute grammars to describe calculations on the internal representation of Creol programs. An attribute grammar is a language for specification of how to generate information from a tree structure. The generated information is a set of attributes, and attributes are pieces of information that correspond to a node in the tree. Attributes are either inherited, synthesized or chained. An inherited attribute collects information from the parent node attributes. A synthesized attribute collects information from child node attributes. A chained attribute is both inherited and synthesised. The University of Utrecht (UU) Attribute Grammar (AG), which is written in and integrated with the Haskell programming language, is the concrete attribute grammar used in this thesis. UU-AG is extended with chained attributes, which are both inherited and synthesized. The data structures from which the Functional Creol Compiler generates attributes, are the abstract syntax trees (AST) of Creol programs. An abstract syntax tree is a data structure that contains a program. The information flow between attributes is illustrated in Figure 1.1, that incidentally is the actual information flow to the topmost node of the Creol compiler AST. Notice that information in chained attributes flows directly between siblings. In the actual implementation the information travels unaltered through the parent node, as if it went directly between sibling nodes.

Some attribute grammars allow different attributes to be specified separately, and the attribute grammar weaves together these separate specifications. This separate specification and weaving is similar to that of aspect-oriented programming [59]. The UU-AG system facilitates separate attribute specifications along with a compiler that weaves the specifications together, and produces Haskell code. The Functional Creol Compiler is written as attribute grammar specifications, which are weaved together to form the Haskell code for the Creol compiler, which is further compiled by a Haskell compiler.

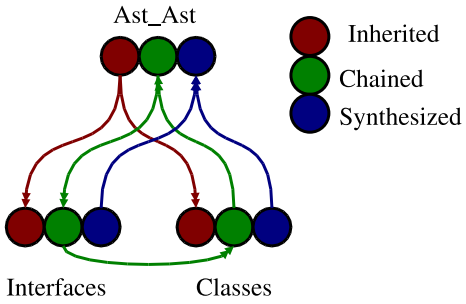

Figure 1.1: Attribute Grammar Flow of Attributes

1.8 Thesis Overview

There are several interwoven threads of information in this thesis. The Creol language is introduced in Section 1.1 with a detailed explanation of language features in Section 2.1 and syntax in Section 2.2. The Functional Creol Compiler is motivated in Section 1.3 and the other subsections in Section 1 provide introductory information. A detailed explanation of the Functional Creol Compiler is in Section 3, and the code in Appendix E. The Creol type system starts with an investigation of type checking issues, such as inheritance and subtyping, in Section 4, then type checking of the object-oriented part of Creol is investigated, in detail and by example, in Section 5, and the resulting type language and type rules are presented in Section 6. The type rules for the functional part of Creol is presented in Appendix B. The viability of both the Functional Creol Compiler and the Creol type system is argued for in Section 7. Suggestions for further work and research is provided in Section 8, and Section 9 provides a conclusion for the thesis.

Chapter 2

The Creol Language

This section is an introduction to the Creol language. The presentation starts with important concepts in the Creol language in Section 2.1. To provide some intuition a sample translation of a small Java program into Creol is given in Figure 2.4. Then the formal grammar is presented piecemeal, interspersed with example syntax and some semantic explanations in Section 2.2. The concept section is an important preliminary to understand both the Creol language semantics and the type analysis in Section 4. The contributions, or suggestions, that have resulted from the work in this thesis are in Section 2.3. The separation of these contributions provides an insight in the Creol syntax development, and makes it easier to evaluate the contributions for integration into the Creol language.

Note that Section 2.3 was entirely decided by this thesis, while Section 2.1 and Section 2.2 where largely decided by prior work on the Creol language.

2.1 Essential Creol Concepts

This section introduces and motivates the concepts that are essential for the semantic and syntactic presentation of the Creol language. The concepts are presented independently from type checking. The type checking of Creol in the Functional Creol Compiler is covered entirely in Section 4.

2.1.1 Processor Release Points

This section investigates how concurrency is expressed in Creol, and specifically the required language constructs. The realisation of concurrency in Creol introduces asynchronous method invocation, along with processor release points and labels.

The Creol language integrates concurrency into the programming language, as opposed to programming languages that provide concurrency constructs as separate libraries. To establish language integration is beneficial, it suffices to examine the motivation for the integration. The current resarch focus for the Creol language is invariant analysis of concurrent programs. Since Creol is a class-based object-oriented language, it is natural to reason about object invariants on a class level. When objects interact concurrently with synchronous com-

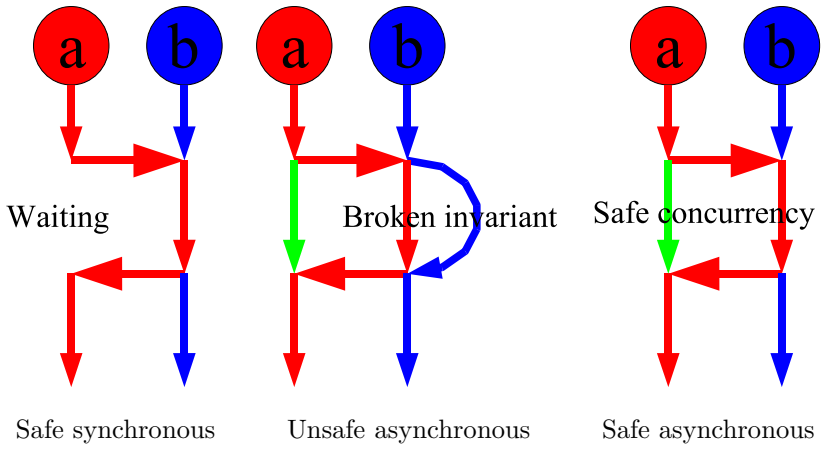

Figure 2.1: Waiting with synchronous and asynchronous communication.

munication, the objects may not perform anything while waiting for an answer. Asynchronous communication is then introduced to eliminate waiting. A naive approach of merely providing a library of asynchronous primitives eliminates the waiting, but requires some sort of synchronisation, otherwise the number of possible code interactions with non-deterministic results make invariant analysis extremely difficult. To remedy this, Creol introduces asynchronous communication into the language with certain restrictions, this combination eliminates waiting while facilitating invariant analysis. The different situations are illustrated in 2.1, where it is shown that with synchronous behaviour, object `a` must wait for object `b` to reply, while with concurrent and asynchronous behaviour, object `a` does not have to wait, but the invariant may be broken because the computations of objects `a` and `b` interleave unpredictably, while with one thread per object and asynchronous behaviour, object `a` may continue while waiting, and the invariant in object `b` is safe. The restriction of one thread per object is crucial to safe concurrency with asynchronous behaviour. There are other techniques that reduce waiting, such as futures [69], that allow an object to continue until the values are first used, and then wait. This allows further related computations, but arbitrary tasks may not be interleaved, because it is not known if the invariant is valid when the waiting occurs, which, in the case of futures, is solved by using a linear type system which imposes relatively strong restrictions on the use of mutability. The Creol language reduces waiting by introduction of explicit processor release points, where the invariant must be valid.

A processor release point is a place in the code of an object where the object can switch between tasks, under the assumption that the invariant is valid. This allows an object to engage in another unrelated computation, safely assuming that the invariant is valid. The Creol framework thus provides asynchronous communication and explicit processor release points, with the restriction that one object may not have more than one thread, at any given time.

It is the programmers responsibility to write code that respects the invariant. The Creol language then guarantees that no broken invariant may occur due to unsafe interaction. This is in contrast to a solution where each code block in

an object maintains an invariant, but when the object switches between code blocks at arbitrary points the resulting interaction can break the invariant. To establish the invariant the code may be analysed by hand or with proof assisting programs [28]. The interaction between asynchronicity and processor release points give rise to active and passive waiting. Active waiting occurs when the object must wait for a result without respecting the invariant, and therefore the object may not switch to another task. Thus active waiting intuitively corresponds to the object being busy while waiting. Passive waiting occurs when the processor is released at a point where the invariant in respected.

The notions of active and passive waiting allow the programmer to decide if waiting should release the processor. However with multiple asynchronous method invocations„ the programmer needs to chose which reply to wait for. The replies of asynchronous method invocations are given a label, therefore every asynchronous method invocation has a distinct label. The label allows the programmer to decide which invocation to wait for, and the waiting can be either active or passive.

The only asynchronous concurrency restriction that is necessary to realise the invariant analysis of Creol programs, is that at most one part of the object is active at any time. The simplest realisation of this is to model each object as a separate process, thus a one-to-one correspondence between objects and processes. Objects are envisioned more as autonomous agents than as data containers, but by that definition each Creol object, regardless of being active or passive, has its own process. In systems with huge quantities of objects, that would be a scalability problem, as each object incurs the overhead of an extra process. Therefore notice that a process in the Creol context is different from that of operating systems, and that the Creol process can be realised by either operating system processes, which are isolated by the operating system, kernel-level threads, which are in the same address space but preempted by the operating system, or user-level fibers, which are light weight with voluntary task-switch. The choice of underlying implementation strategy for object threads can be made freely as the semantics of Creol processes place few restrictions on possible implementation strategies. The Creol language promotes both object-oriented and functional programming. With both types of programming each concept is used where it has merit, hence objects are not abused for data containment where algebraic data types are more appropriate. Hence it is reasonable to expect that the number of objects will be less compared to languages such as Java or C# where almost everything is an object.

2.1.2 Separation of Behaviour and Code Reuse

This section documents that the Creol language research intends to separate behaviour and code reuse by having both interfaces and classes in the Creol language, along with mechanisms for multiple inheritance.

The realisation that behaviour and code reuse are separate aspects and therefore deserve separate treatments [54, 52] is designed to help the Creol language to capture visible object behaviour with interfaces and at the same time facilitate code reuse with classes. This separation allows two different notions of inheritance:

- Inheritance between interfaces.

- Inheritance between classes.

Interfaces are connected to classes by explicit declarations. This separation between inheritance hierarchies is intended to allow different inheritance restrictions on interfaces and classes, so the Creol language can express behaviour inheritance differently than class inheritance.

The notion of behaviour captured by interfaces corresponds to object behaviour through viewpoints as formulated by Johnsen and Owe [51, 53].

The suitability of interfaces and classes for the separation of inheritance and subtyping is discussed in Section 4.3. The suitability of interfaces to capture behavioural restrictions is discussed later in Section 5.7. Multiple inheritance for both interfaces and classes has been added to the Creol language to provide increased behavioural reuse for interfaces and code reuse for classes.

2.1.3 Compositional Program Analysis

This section investigates how compositional invariant analysis necessitates static binding in addition to dynamic binding in a programming language with class inheritance.

The Creol language facilitates compositional program analysis through a language construct for static binding. Static binding is a manner of method invocation on self, where self is the current object. Static binding is a call site annotation that prevents subclass method overrides from affecting the call site. Dynamic binding is the algorithm used in conventional languages such has Java, C++ and C#, where a method invocation binds to the most specialized method body. The discussion of type system implications of static binding is postponed until Section 8.6.

To see why compositional analysis of invariants requires static binding, consider a class `A` with methods `m` and `n` where the class has the invariant I and the method `m` has an additional invariant so $I \wedge P$ is a precondition and $I \wedge Q$ is a postcondition and the method `n` has $I \wedge R$ as precondition and $I \wedge S$ as postcondition. Now suppose that `m` calls `n` so the logical dependencies $P \rightarrow R$ and $S \rightarrow Q$ appear. This is the intuitive situation where somehow the precondition of `n` depends on the precondition of `m` and the postcondition of `m` depends on the postcondition of `n`. Since both `m` and `n` are in the same class, only the current class needs to be analysed to establish that the pre- and post-conditions hold. Then assume that there is a class `B` that extends `A` and overrides `n` with `n'`. Since the postcondition of `m` in the subclass depends on `n'` instead of `n`, the previous analysis about invariants in class `A` is invalidated in the subclass `B`. The analysis of `m` must be done again with `n'`. This situation makes the analysis of method `m` depend on method overrides, which makes compositional analysis impossible. It is then necessary to introduce a mechanism that prevents method override from affecting compositional analysis. Static binding is one such mechanism. Static binding protects the call site from method overrides, so the analysis of `m` is independent of method overrides. The problems with dynamic binding and the solution with static binding are illustrated by pseudo Creol code in Figure 2.2 where invariants, post- and preconditions are specified as comments at the relevant places. The notation $\stackrel{?}{\rightarrow}$ designates a problematic assumption while $\rightarrow$ is a valid assumption.

```
class A // class invariant I
begin
  // precondition I ∧ R
  op n() == ...
  // postcondition I ∧ S
  // precondition  I ∧ P
  op m() == ...
  // postcondition I ∧ Q
          // dynamic binding problem P →? R and S →? Q
          ...n()...
          // static binding solution  P → R and S → Q
          ...n@A()...
end

class B inherits A // class invariant I'
                   // I →? I'
begin
  // n' precondition I' ∧ R'
  op n() == ...
  // postcondition I' ∧ S'

  // dynamic binding problem P →? R' and S' →? Q
  // Problem avoided with static binding!
end
```

Figure 2.2: Example problem and solution with static binding.

Static binding facilitates compositional analysis, so given a superclass with an analysis, the code can be reused and the analysis can be plugged into the analysis of the subclass. Static binding solves the problem because the method may be overridden in a subclass, but the invocation is bound to a specific class and is untouched by the overriding. Compositional analysis is especially important in the case of code reuse where the super-class implementation is hidden. In the example code in the method `m` the invocation `n@A()` would statically bind the call to method `n` to class `A`, with the call `n@A`, efficiently preventing `B` from overriding the call site understanding of `n`. If the subclass `B` would like to override `n` then it must also override `m`, which is exactly what is required, because the analysis of `m` depends on `n`. Consider the inheritance hierarchy of the example code, then the possible search paths for a method `n` has an upper bound in the class `A`, and the static binding `n@A` represents a lower bound, a starting point for the search. In the case of multiple inheritance and static binding these bounds determine a diamond which constrains the search path for a method. Notice that since static binding represents a lower bound, it is not necessary for the target class to implement the statically bound method, as long as the method is available by inheritance. The method search path diamonds example classes are illustrated in Figure 2.3, where static binding is illustrated for `n@B` in `B` and for `n@C` for code in `C`.

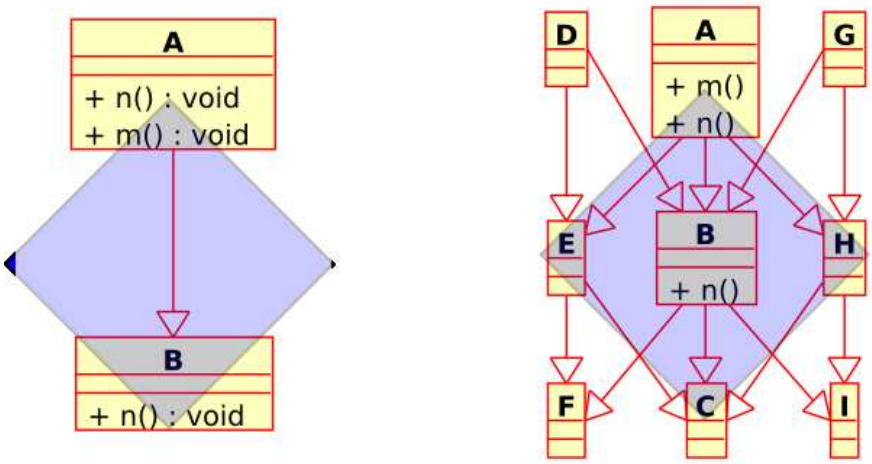

Figure 2.3: Method search path diamond with static binding as lower bound

2.1.4 Cointerface

This section explains the Creol language construct called cointerface.

A cointerface is a caller interface requirement, so the caller of a method must implement the given interface. The notion of cointerface comes from prior Creol research [53,51,75] and is used in formal specifications of external behaviour. A cointerface can be modeled as an input parameter, but a cointerface is a stronger requirement. The caller itself must implement the cointerface. Therefore cointerfaces are both an interface requirement and an object identity requirement. This object requirement of a cointerface is essential for object viewpoints [51], and is a subject for future Creol language research which aims to facilitate security in the sense of guaranteed object identity. By guaranteeing object identity it is envisioned possible to state extra security properties for programs [50]. Cointerfaces in Creol extend the language with a special syntax.

2.1.5 Sample Translation to Creol from Java

This section translates a small Java program into the equivalent Creol program, to give the Java and C# familiar readers an intuitive idea of the Creol syntax.

The sample code in Figure 2.4 declares an interface `AI` and a class `A` that implements the interface.

2.2 Creol Grammar and Syntax Examples

This section presents the grammar for the Creol language.

The grammar for the Creol language is presented gradually through examples and the corresponding EBNF grammar productions. EBNF is shorthand

```
// Java Code
public interface AI {
    public int getset(int newx);
}
public class A implements AI {
     private int x;
     public int getset(int newx) {
              int y = newx;
              x = newx;
              return y;
     }
}

// Creol Code
interface AI
begin
  with Any op getset(in newx:Int out oldx:Int)
end

class A implements AI
begin
  var x:int;
  with Any
    op getset(in newx:Int out oldx:Int) ==
       oldx := x;
       x := newx;
end
```

Figure 2.4: Java to Creol sample translation

for extended Backus-Naur form [35, Sec. 2.1.1] and is a metalanguage for expressing programming language grammars. The complete EBNF is shown in Appendix A. Although EBNF grammars express context free languages, the Creol language is not restricted to being context free. The EBNF presentation is merely used for presentation purposes because it is a common way of describing languages. A more thorough discussion on parsers and the languages they can parse, is given in Section 3.5. There existed no prior syntax oriented EBNF grammars for the Creol language, therefore the EBNF grammar presented is based on example code, discussions and semantic language descriptions found in previous work [51, 55, 53, 52, 56, 54, 8].

A grammar consists of non-terminals and terminals. A grammar has one non-terminal as a start symbol. A production connects a non-terminal to a composition of terminals and non-terminals. Terminals are literate symbols in a program. A program is syntactically valid if it is possible to derive it by starting with the non-terminal start symbol and replace non-terminals by their productions in a manner that generates the given program, which consists only of terminals. The composition of non-terminals and terminals in this grammar are those available in EBNF, but with syntax influenced by GNU extended regular expressions as defined by POSIX 1003.2. The rules for composition are explained when they are first used in the presentation. Verbatim text is presented by `typewriter` and productions are presented as *italics*. Parentheses are used for grouping.

The grammar presentation starts with identifiers and the starting production *program*, then the other productions are explained and exemplified by Creol code.

2.2.1 Identifiers

Case is significant for Creol identifiers. There are variable identifiers and type identifiers. A variable identifier *varid* begins with a lower case letter, while a type identifier *typeid* begins with an upper case letter. Brackets [] are used to denote either of the characters inside the brackets, with the exception of − which is used for character ranges. The productions use the notation [a − zA − Z] to denote any character in lower or upper case ranges. Although these productions only allow letters in the English alphabet, they are sufficient for the Creol grammar, as the treatment of UTF and other encodings are beyond the scope of this thesis. The Functional Creol Compiler uses library provided functions to parse *varid* and *typeid* productions, and these libraries can easily be replaced with UTF aware versions when necessary. As an example consider a variable `foo` and a type `Bar`. These productions are therefore simplified and sufficient.

varid ::= [a − z][a − zA − Z]∗
typeid ::= [A − Z][a − zA − Z]∗

The *typexpr* refers to the name of a type where the type is used while *typedecl* refers to the name of a type where the type is defined. Although *typexpr* and *typedecl* are identical the distinction is necessary for the type parametrisation that is introduced later in Section 2.3.6.

typexpr ::= *typeid*
typdecl ::= *typeid*

2.2.2 Comments

The Creol language has adopted the same syntax for comments as found in C, C++, *C♯*, and Java, as shown in this example:

```
// This is a comment to the end of the line

/* This is a block comment
   that can span several lines
*/
```

The grammar for comments is somewhat simplified, because an accurate EBNF grammar for block comments is quite complicated, and the EBNF presentation for comments is an illustration of the actual comments in Creol, which support nested *comment* blocks. The *comment* production may appear anywhere in the Creol code. The symbol $\backslash n$ denotes the end of the line, and symbol . denotes any character except the end of the line, and the symbol $*$ is a prefix that repeats any number of times, also zero. The part $.*\backslash n$ means any number of characters and an end of line. The bracket [] denotes a range of characters, and the $[.\backslash n]*$ denotes any number of characters or newlines.

comment ::= `//`$.*\backslash n$
| `/*`$[.\backslash n]*$`*/`

2.2.3 Programs

A Creol program is a series of class and interface declarations, and there must at least be one declaration.

program ::= $\overline{(\textit{interface}|\textit{class})}$

Notice that *program* is the starting non-terminal for any Creol program. The | chooses between either *interface* or *class*. The parentheses group together $\overline{\textit{interface}}|\textit{class}$ so the overline affects $(\textit{interface}|\textit{class})$. The content below the overline must have at least one occurrence, but can be repeated any number of times.

2.2.4 Interface Declarations

A Creol interface describes visible object behaviour. The visible object behaviour consists of the method signatures that an object understands. Interface inheritance allows reuse of method signatures. The new EBNF syntax introduced in the *interface* production is the suffix ?, which denotes that the previous construction is optional. Notice that an overline ended with a symbol, such as $\overline{\textit{typexpr}\,\texttt{,}}$ is one or more repetitions of *typexpr*, where the symbol , is used as a separator.

interface ::= `interface` *typedecl*
(`inherits` $\overline{\textit{typexpr}\,\texttt{,}}$)?
`begin`
$\overline{\textit{signature}}$
`end`

The interface contains several method signatures. Each signature describes how a method may be used. A method signature starts with the previously mentioned cointerface, then it denotes the name of the method and then the input and output parameters. Notice how cointerface, `with` *typexpr*, and both intput and output parameters are optional, and that overlines can be nested.

signature ::= (`with` *typexpr*)?
`op` *varid*((`in` $\overline{\overline{varid}:typexpr\,\texttt{,}}$)?(`out` $\overline{\overline{varid}:typexpr\,\texttt{,}}$)?)

The following example code declares an interface `AI` with a method `getset` that has one integer `newx` as in parameter and another integer `oldx` as out parameter. The cointerface `with Any` means that there are no caller interface restrictions, because `Any` is a special interface that all interfaces inherit from.

```
interface AI
begin
  with Any op getset(in newx:Int out oldx:Int)
end
```

2.2.5 Class Declarations

The class is a template for how objects are created. Since interfaces describe object behaviour, classes may implement interfaces, which means that the objects created from the class will behave according to the implemented interfaces. A class may also inherit code from another class, to facilitate code reuse. The class contains both instance variables and methods, which consist of a signature and a body. The class itself optionally accepts parameters, which are used as initialisation values for an object. The instance variables allow an optional expression that describes the initial value of the variable.

class ::= `class` *typedecl*(($\overline{varid\,\texttt{,}}$))?
(`inherits` $\overline{typexpr\,\texttt{,}}$)?
(`implements` $\overline{typexpr\,\texttt{,}}$)?
`begin`
$\overline{var}$?
$\overline{method}$?
`end`

var ::= `var` $\overline{\overline{varid(\texttt{=}expression)?}\,\texttt{,}:typexpr\,\texttt{,}}$`;`
method ::= *signature* `==` $\overline{var}$? `;` *statement* `;`

The $\overline{var}$? declaration in the *class* production optionally declares instance variables for objects, while the $\overline{var}$? declaration in the *method* production optionally declares variables local to the method body, as variables can not be declared inside a statement.

As an example class we create `class A` that implements the interface `AI`. To provide a class we use some simple statements. Although statements are not presented yet the code should intuitively be readable. The body of the `getset` method updates the instance variable `x` and returns the previous value.Notice that methods do not have a return statement, instead all the out parameters are sent back when the method finishes, hence the value of `oldx` is sent back when the method ends, hence a method invocation returns values by implicit assignment, and the method invocation itself doesn't return anything.

```
class A implements AI
begin
  var x:int;
  with Any
    op getset(in newx:Int out oldx:Int) ==
       oldx := x;
       x := newx;
end
```

2.2.6 Statements

Statements express imperative programming with side-effects. Creol is a concurrent language which requires additional statements in comparison to other imperative languages. The production for *statement* is presented stepwise, and each production is commented. Statements can be grouped by parentheses.

statement ::= `(` *statement* `)`

Statements can be composed serially, but since Creol is a concurrent language statements are not just composed serially. The statement operators are sequence, merge and choice. The sequence operator `;` composes statements serially. The merge operator `|||` executes two statements in arbitrary sequence. For instance `x := 1 ||| x := 2` will set `x` to either 1 or 2. The choice operator `[]` executes either one or the other statement. Statements composed with merge can be translated into the more primitive choice, such as the statement `x := 1 ||| x := 2` which is equivalent to `(x := 1 ; x := 2) [] (x := 2 ; x := 1)`. This translation is in fact done by the Creol Virtual Machine. Recent development of the Creol Virtual Machine does not use this translation, but offers a more fine-grained merge. The sequence `;` has higher precedence than merge `|||` and choice `[]`.

| *statement* (`;`|`|||`|`[]`) *statement*

Assignment uses the symbol `:=`, such as in this example `x := 1`, where the variable x is assigned the value 1. The Creol language also supports multiple assignments such as `(x,y,z) := (1,2,3)`, but since the Creol language does not support tuples the multiple assignment is a specific construct where there is syntactically the same number of elements on each side of `:=`. The Creol Virtual Machine executes multiple assignments one variable at the time from left to right, such that `(x,y):=(y,x)` becomes `x:=y;y:=x`, according to the rule `rl [assign]` the Creol Virtual Machine. A more general version of the multiple assignment is possible with tuples, which is outlined in Section 2.3.8.

| *varid* `:=` *expression*
| `(`$\overline{\textit{varid}}$`,)`:=`(`$\overline{\textit{expression}}$`,)`

The conditional `if` statement has a closing `fi` to avoid the dangling `else` problem. An example statement is `if True then x := 1 else x := 0 fi`.

| `if` *expression* `then` *statement* `else` *statement* `fi`

The conditional `while` statement does conditional repetition, as for instance `while x < 10 do x := x + 1 od`.

| `while` *expression* `do` *statement* `od`

The Creol language is imperative to the degree that method invocations are statements and not expressions, which allows method invocations to return multiple values. This applies both to synchronous and asynchronous method invocation. To demonstrate a synchronous method invocation, consider `a.addget(10;r)` where the variable `a` contains an object of interface `AI`, from the running interface example, then the variable `r` will be altered by the method invocation to contain the result. The general form for method invocation has an optional object expression, optional input and output parameters. Notice that both input parameters and output parameters are separated by the symbol `,` while the symbol `;` separates input parameters from output parameters.

| (*expression* `.`)?*varid* `(`$\overline{\textit{expression}\texttt{,}}$`;`$\overline{\textit{varid}\texttt{,}}$`)`

The concurrency in Creol introduces asynchronous method invocations, that are facilitated by labels. A labels identifies an asynchronous method invocation so the reply can be waited for. The caller decides if the method invocation is synchronous or asynchronous. The previous synchronous statement `a.addget(10;r)` can be written asynchronously as `label!a.addget(10) ; label?(r)`. Notice that any number of statements could occur between the asynchronous method invocation and the request for the reply. The production *varid* describes the valid syntax for labels.

| *varid* `!(`*expression* `.`)?*varid* `(`$\overline{\textit{expression}\texttt{,}}$`)`
| *varid* `?(`$\overline{\textit{varid}\texttt{,}}$`)`

As discussed the Creol concurrency also introduces processor release points with the `await` and `wait` statements. The `wait` statement releases control until scheduling returns control at a later time, which is similar to yield in for instance the Java Application Programming Interface thread library [87, Sec. Java.lang.Thread]. The `await` statement releases control until scheduling returns control and the `await` condition is true. The example statement `await x < 42` releases control, which is returned at a later time when the condition `x < 42` is met. The `await` also allows to wait for a method reply, and the previous asynchronous example can be extended to `label!a.addget(10);...; await label?;...; label?(r)` so other computations can take place before the method return is used.

| `await` *expression*
| `await` *varid*`?`
| `wait`

To facilitate invariant analysis the Creol language has static binding, as discussed in Section 2.1.3. Static binding uses a special syntax at method invocation. Let use change the previous examples to use static binding. The synchronous method invocation is written `addget@A(10;r)` while the asynchronous is written `label!addget@A(10);label?(r)`, given that the code is inside a class `A`. Static binding requires special syntax for synchronous and asynchronous method invocation.

| *varid*`@`*typeid* `(`$\overline{\textit{expression}\texttt{,}}$`;`$\overline{\textit{varid}\texttt{,}}$`)`
| *varid* `!` *varid*`@`*typeid* `(`$\overline{\textit{expression}\texttt{,}}$`)`

2.2.7 Predefined Constants

The constants of Creol are numbers, Boolean values, and strings. The productions use some new EBNF syntax, $[1-9]$ denotes either of the characters in the range 1 to 9, while $[0-9]*$ uses the prefix $*$ star to denote zero or more occurrences of characters in the range 0 to 9. The pattern `"[^"]*"` can be read as begin with a double quotation, then repeat zero or more times characters that are not a double quotation, and end with a double quotation. Notice that the Creol language allows double quotations to be embedded into a string by the prefix \, but this is difficult to express with EBNF, therefore a simpler definition of *str* is given. The productions are:

int ::= `[1-9][0-9]*`
bool ::= `True|False`
str ::= `"[^"]*"`

To demonstrate `123` is a number, `True` is a Boolean value, and `"Hello world"` is a string.

2.2.8 Expressions

Expressions become values when evaluated. The simple parts of the expression production contain either identifiers, constants, grouping with parenthesis, along with both unary and binary operators on expressions. These expressions are simple, therefore no examples are provided.

expression ::= *varid* | `int` | *bool* | *str*
| `(` *expression* `)`
| (`-`|`not`) *expression*
| *expression* (`[+-*/=<>]|<=|>=|!=|or|and`) *expression*

The operator precedence for the binary operators is shown in Table 2.1. The expression `1 + 2 * 3 < 4 and True or False` is parsed as `((((1 + (2 * 3)) < 4) and True) or False)`.

Highest precedence first
`*`, `/`
`+`, `-`
`<`, `<=`, `>`, `>=`, `=`, `!=`
`or`, `and`

Table 2.1: Operator precedence for Creol

The `new` keyword creates an object from a class. The expression `new A()` creates an object of the class `A`.

| `new` *typexpr* `(`$\overline{\textit{varid}\texttt{,}}$`)`

The labels used in asynchronous method invocation can be used to test if a method reply has arrived. The label from our running example is used in this example `if label? then label?(r) else x := 5 fi`. where the label check

`label?` evaluates to `False` or `True` depending on whether or not the reply to `label!a.addget(10)` has arrived. The label check is always used in conjunction with asynchronous method invocation.

$$\mid \textit{varid}\texttt{?}$$

2.3 Syntax Proposals for Creol Extensions

This section introduces changes and extensions this thesis proposes for the Creol language.

Recall from Section 1.1 that the Creol language already is targeted to cater for functional programming, however the presence of functional programming in Creol has never been accounted for. This section mainly accounts for the syntax of functional programming in the Creol language by changes to and extensions of the EBNF grammar defined in Section 2.2. The grammar productions for these changes and extensions are given. Functional programming is supported through the declaration and use of *variant*, *tuple*, *list*, *record*, *procedure*, *typedecl* and *typexpr*. The *data* type declaration production introduces *tuple*, *list*, *record* and *variant*. The *program* production is extended with *data* and *procedure*. The *typedecl* and *typexpr* are distinguished to facilitate declaration and use of parametrisation. The new production for *program* permit data type declarations and procedure definitions:

$$\textit{program} \;::=\; \overline{(\textit{interface}|\textit{class}|\textit{data}|\textit{procedure})}$$

2.3.1 Local Declarations

This section suggests changes to allow local declarations in Creol.

The Creol syntax requires all declarations prior to statements. To have a more uniform framework we suggest that the *statement* production is extended with a declaration that scopes over later statements. This allows a simplification of the *method* production. The extended *statement* and the new definition for *method* are:

$$\begin{aligned}\textit{statement} \;&::=\; \ldots \\ &\mid \textit{var}\,\texttt{;}\,\textit{statement}\end{aligned}$$

$$\textit{method} \;::=\; \textit{signature}\ \texttt{==}\ \textit{statement}\,\texttt{;}$$

The introduction of declarations into the *statement* production allows variable declarations to be close to the point where they are needed.

2.3.2 Expressions as Statements

The section motivates and suggests that an expression can be regarded as a statement in the Creol language.

The Creol language considers the conditional `if` as a statement, which only permits imperative programming with `if` statements. To support functional programming the conditional `if` is considered as an expression that evaluates to a value. To pursue the goal of imperative and functional programming in Creol

we wish to consider `if` as an expression where the branches are statements and consider expressions as statements. The functional and imperative usage of `if` is demonstrated by:

```
// Imperative
if True then x := 1 else x := 0 fi

// Functional
x := if True then 1 else 0 fi
```

To remove a production we use the syntax −| and get the necessary changes to the grammar:

statement ::= ...
  −| `if` *expression* `then` *statement* `else` *statement* `fi`
  | *expression*

expression ::= ...
  | `if` *expression* `then` *statement* `else` *statement* `fi`

Intuitively the `if` statement is now an expression, and but since any expression can be a statement, `if` can now be used both as a statement and an expression. Also notice that the branches in the `if` are still statements, but since any expression can be used as a statement, it is possible to use expressions in the branches.

2.3.3 Label Check as Expression

This section motivates and suggests that label check is regarded as an expression.

The label from an asynchronous method invocation can only be used by `await` to wait for a method reply. Since the result of label check is a boolean value, one could let objects react to the absence of a reply, as well as the presence. If no reply has arrived upon reaching the corresponding label check, a false is returned, otherwise true is returned. This would be possible if the label check is regarded as an expression. To do this requires to alter the *statement* and *expression* productions.

statement ::= ...
  −| `await` *varid*`?`

expression ::= ...
  | *varid*`?`

2.3.4 Procedures

This section demonstrates how to extend the Creol language grammar with procedures.

The *procedure* production facilitates function declarations. The production is called procedure rather than function to reflect that functions in Creol can have side-effects, whereas mathematical functions do not have side effects. For instance the procedure `add` that sums two integers and return the result:

```
proc add(left,right:Int):Int == return left + right ;
```

The procedure `add` is used in the expression `add(2,3)`, thus the *expression* production must be extended with procedure calls. Notice that procedures are expressions with only input parameters, which is different from methods that are statements that may contain output parameters. The keyword `return` is introduced as a statement that returns an expression, therefore the *statement* production is extended. To facilitate procedures that do not return anything, the return expression is optional. The grammar for *procedure* and the extensions to *expression* and *statement* are:

$$\begin{aligned}
&\textit{procedure} \;::=\; \texttt{proc}\ \textit{varid}\,\texttt{(}\overline{\overline{\textit{varid}\,\texttt{,}}\,\texttt{:}\,\textit{typexpr}}\texttt{)}\,\texttt{:}\,\textit{typexpr}\ \texttt{==}\ \textit{statement}\\
&\\
&\textit{expression} \;::=\; \ldots\\
&\qquad\quad \mid \textit{varid}\,\texttt{(}\overline{\textit{expression}\,\texttt{,}}\texttt{)}\\
&\textit{statement} \;::=\; \ldots\\
&\qquad\quad \mid \texttt{return}\ \textit{expression}?
\end{aligned}$$

Notice that the *procedure* production provides declarations by the local declaration extension to the *statement* production. Furthermore the type `Void` is introduced to denote the return value when `return` is used without an expression, and the expression `void` that has type `Void`, so a return without an expression is the same as `return void`.

2.3.5 Algebraic Data Types

This section demonstrates how to extend the Creol language with algebraic data types.

Support for algebraic data types are realised by a *data* production that can declare algebraic data types, and by extensions to the *expression* production that can construct and deconstruct an algebraic data type.

Although the Creol language is object-oriented, there is still good reason to support algebraic data types and the functional tools to manipulate them. In contrast to later languages such as SmallTalk, already the developers of the first object-oriented language Simula made a clear distinction between mutable and immutable data values [25], which leads to the concepts of class and abstract data type respectively, where both are equally important tools for different tasks.

The most general notion of algebraic data types is realised by records [77, Sec. 11.8] and variants [77, Sec. 11.10]. Records can further encode tuples, and variants can encode enumerations. Together records and variants can also encode options and lists. As a quick reminder, a record is a collection of values distinguished by a label, a variant is a heterogeneous collection of values, a tuple is a record where the labels are natural numbers, an enumeration is a variant where all values are empty, an option is a variant with two alternatives where one is empty, and a list is a polymorphic recursive data type with a collection of values of the same type. This section will only focus on records and variants, as they constitute a general basis.

Deconstruction for records is facilitated by projection, and deconstruction of variants is facilitated by `case` expressions. Case expressions use a general form of pattern matching, that provides variable binding, guards and value patterns. Variable binding allows a case branch to introduce new bindings. A guard is a condition that must hold in addition to the pattern match. Value patterns are patterns with values, where the pattern matches if the values match.

To provide the reader with an intuition we demonstrate each of the concepts variable binding, guard and value patterns alone and in combination. Both syntax suggestions and examples are provided for tuples, enumerations and lists, along with the encoding into records and variants.

The following code declares a `Person` record that contains two elements labeled `name` and `age`. Next we create a procedure `test` that creates a variable `p` of type `Person`, then we extract the information from the variable with dot-notation. Although it is feasible to extend records such that labels can be used for insertion and not only for extraction, this is not done by suggestion from professor Owe [74].

```
data Person = (name:Str, age:Int) ;

proc test():Void ==
  var p:Person ;
  p := (''John Doe'',37) ;
  print(p.name);
  return ;
```

The symbol `|` is used to distinguish variants. The variant name declares a constructor that creates the variant. The constructors are used in `case` expressions. A case expression inspects a variant type and selects different branches depending on the variant. The symbol `_` is a wildcard pattern that match any variant. Each variant contains a type. Consider a variant type `Angle` with two variants. The variant `Degrees` contains an `Int` and the variant `Radians` contains a `Float`. The value of a variant can be extracted by case pattern with a variable binding, so `Degrees d` binds the variable `d` to the value in the variant `Degrees`. Variables bound by case patterns can be used in the corresponding case branch.

```
data Angle = Degrees Int
           | Radians Float

proc test():Void ==
  var a:Angle ;
  a := Degrees 90 ;
  case a of Degrees d then print(``Degree variant'')
          | _ then print(``Wildcard variant'')
         fo ;
```

The variants can contain any type, and are especially useful and intuitive with records. The next example uses a variant type `Employee` with variants `Manager` and `Staff`, where the variants contain records. Notice that the case branches perform variable binding hence the pattern `Manager m` binds `m` to the record `(name:Str, salary:Int)`. Since `m` is a record *dot-notation* is used to extract information. Notice that the wildcard symbol `_` can be used instead of a variable to avoid variable binding.

```
data Employee = Manager (name:Str, salary:Int)
              | Staff (id:Int) ;

proc test():Void ==
  var e:Employee ;
  e := Manager (``John'',500) ;
```

```
case e of Manager m then print(m.name)
        | Staff _ then print(``A staff member'')
       fo ;
```

Now we extend the previous case expression example with both guards and value patterns. We use a guard to distinguish managers by their salary, and the programmer of the system uses value pattern to give herself special treatment. The guard is prefixed by the keyword `when`.

```
case e of Manager m
          when m.salary > 500
          then print(``Important '' ++ m.name)
        | Staff ((id=123) as s)
          then print(``The divine programmer'') ;
               raise_salary(s)
        | Staff _
          then print(``A staff member'')
      fo ;
```

Notice that value patterns use `(id=123)` to describe a record where the field `id` has value 123, and that `((id=123) as s)` binds the variable `s` to the record that has a field `id` with value 123. The value pattern `Staff ((id=123) as s)` can be rewritten as a guarded pattern `Staff s when s.id = 123` , but this rewrite can be quite lengthy and value patterns provide a compact representation.

The `case` expression does pattern matching on constructors and values, which is a well known technique that is present in many programming languages, and especially in languages of the ML family. Together with dot-notation for record projection the `data` and `case` constructs support construction and deconstruction of algebraic data types in combination with type safety.

The introduction of algebraic data types introduces the *data* production, as well as dot-notation for record projection and `case` to the *expression* production. Notice that the `case` extension to *expression* is defined via the *pattern* production. The *pattern* production facilitates variable binding, the binding with keyword `as`, wildcard patterns, and nested patterns. Since we already consider expressions as statements, the `case` construct features as both an expression and a statement. The *data* production uses the *typedecl* production. The *typedecl* production is further extended to handle records and variants. Also notice that a variant can contain any type, not just records.

data ::= `data` *typedecl* `=` *typedef* `;`
typedef ::= *record*
| $\overline{\textit{variant}\,\texttt{|}}$ `;`
| *typexpr*
record ::= `(` $\overline{\overline{\textit{varid}\,\texttt{,}}\,\texttt{:}\,\textit{typexpr}\,\texttt{,}}$ `)`
variant ::= *typeid* *typedef*

expression ::= ...
| *expression* `.` *label*
| `case` *expression* `of` $\overline{\textit{pattern}\ (\texttt{when}\ \textit{expression})?\ \texttt{then}\ \textit{statement}\,\texttt{|}}$ `fo`
label ::= *varid*
pattern ::= `_`
| *varid*
| *typeid* *pattern*

| *pattern* `as` *varid*
| `(`$\overline{\textit{varid}\texttt{=}\textit{pattern}\texttt{,}}$`)`

2.3.6 Parametrisation

This section proposes a syntax for parametrisation in the Creol language.

When a type definition uses a type that is undefined, the type definition is parametrised by that unknown type. If the unknown type can be replaced by any type, the parametrisation is unconstrained. If there are some restrictions on which types that can be used to replace the unknown type, the parametrisation is constrained. When a parametrised type definition is used, the unknown type must always be replaced with a known type. The production *typedecl* declares type parametrisation, and the production *typexpr* uses parametrized types. The type interpretations of parametrisation is left to Section 5.16. Consider an unconstrained interface `UI` and a constrained interface `CI`. The interface `UI` can be parametrised by any type, and then `var n:UI[Int]` declares the variable `n` to have type `UI` parametrised by an integer. The interface `CI` can only be parametrised by types that are compatible with `B`, where `B` is an interface, therefore `var n:CI[Int]` is illegal. Let `C` be a subinterface of `B`, then `var n:UI[C]` is legal. The definition of compatibility between interfaces is decided by the $\ddot{<}$ relation which is introduced in Section 4. The code for `UI` and `CI` is:

```
interface UI[T]
begin
   ...
end

interface CI[T<B]
begin
   ...
end
```

The parametrisation combines well with algebraic data types and permits to write arbitrary parametrised algebraic data types. The parametrization appears visually as if the type takes an argument, and the argument happens to be a type. To illustrate this the type `Angle` is parametrized by a type called `T` so the `Degree` variant contains a value of type `T`. Then the procedure test is defined for `Angle` parametrised by `Str`, an arbitrary choice that allows us to print the value.

```
data Angle[T] = Degrees T
              | Radians Float

proc test(a:Angle[Str]):Void ==
  case a of Degrees s then print(s)
          | _ then print("Wildcard variant")
         fo ;
```

Now we look at the grammar for parametrisation. The *typedecl* production declare a non-parametrised type, *typeid*, a universally quantified type, *typeid* `[`$\overline{\textit{typeid}\texttt{,}}$`]`, or a bounded type, *typeid* `[`$\overline{\textit{typeid}\texttt{,<}\textit{typexpr}\texttt{,}}$`]`.

typedecl ::= *typeid*
| *typeid* `[` $\overline{\textit{typeid}\ \texttt{,}}$ `]`
| *typeid* `[` $\overline{\textit{typeid}\ \texttt{,}}$ `<` *typexpr* `,` `]`

The *typexpr* production use a non-parametrised type, *typeid*, or a parametrised type, *typeid* `[` $\overline{\textit{typexpr}}$ `]`, where type parameters are applied.

typexpr ::= *typeid*
| *typeid* `[` $\overline{\textit{typexpr}\ \texttt{,}}$ `]`

Both the *typedecl* and the *typexpr* rules could be written more compactly with EBNF, but this verbose form was chosen to intuitively have one production for unconstrained and another production for constrained parametrisation, and to keep the non-parametrised types as a separate production.

2.3.7 Enumeration

This section motivates and provides a customised syntax for enumerations.

An enumeration is a variant that contains no value. This example demonstrates enumeration with the variant type `Color`, and shows how cumbersome the syntax is.

```
data Color = Red Void
           | Green Void
           | Blue Void ;

proc test():Void ==
  var c:Color ;
  c := Red void ;
  case c of Red _ then print(``Red variant'')
          | _ then print(``Not red variant'')
         fo ;
```

It is superfluous to write and repeat that the variants don't contain any information. To achieve this the programmer can omit the type `Void` and the corresponding value `void`. The simplified example becomes:

```
data Color = Red
           | Green
           | Blue

proc test():Void ==
  var c:Color ;
  c := Red;
  case c of Red then print(``Red variant'')
          | _ then print(``Not red variant'')
         fo ;
```

To include simplified variants in the Creol grammar the *variant* and *expression* productions are altered.

variant ::= ...
| *typeid*

expression ::= ...
| *typeid*

2.3.8 Tuples

This section motivates and provides a customised syntax for tuples.

A tuple organises information according to position, just as a record whose labels are natural numbers. Such an interpretation is natural since Creol already facilitates records and variants. The following code creates a tuple and then extracts the information from the tuple.

```
proc test():Int ==
  var t:(String,Int) ;
  t := (``the string'',42) ;
  print t.1 ;
  return t.2 + 6 ;
```

This code is just syntactic sugar for the following

```
proc test():Int ==
  data Tuple2[S,T] = (1:S,2:T) ;
  var t:Tuple2[String,Int] ;
  t := (``the string'',42) ;
  print t.1 ;
  return t.2 + 6 ;
```

Notice that the translation of `(String,Int)` is the parametrized $\texttt{Tuple}_2\texttt{[String,Int]}$. This means that the following is implicitly defined by the compiler:

$$\begin{aligned}
&\texttt{data Tuple}_2\texttt{[}T_1\texttt{] = (1:}T_1\texttt{)}\\
&\texttt{data Tuple}_2\texttt{[}T_1\texttt{,}T_2\texttt{] = (1:}T_1\texttt{,2:}T_2\texttt{)}\\
&\vdots\\
&\texttt{data Tuple}_N\texttt{[}T_1\texttt{,}\ldots\texttt{,}T_N\texttt{] = (1:}T_1\texttt{,}\ldots\texttt{,N:}T_N\texttt{)}
\end{aligned}$$

so for each number of elements in a tuple there is a corresponding data definition, and for any positive N the following translation is valid:

$$\texttt{(}T_1\texttt{,}\ldots\texttt{,}T_N\texttt{)} \rightsquigarrow \texttt{Tuple}_2\texttt{[}T_1\texttt{,}\ldots\texttt{,}T_N\texttt{]}$$

Then it is possible to define parallel assignment, such as `(x,y) := v` as syntactic sugar for `x := v.1 ||| y := v.2`, which in general is the translation

$$\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_N\texttt{):=e} \rightsquigarrow v_1\texttt{:=e.1}\texttt{|||}\ldots\texttt{|||}v_N\texttt{:=e.N}$$

where `|||` is the previously defined arbitrary statement execution order. This interpretation of multiple assignment replaces the previous non-general approach.

To incorporate tuples in this manner requires compiler support for the translation. The necessary changes of the Creol grammar affect the *statement*, *expression* and *typedef* productions:

statement ::= ...
 −| `(`$\overline{varid\texttt{,}}$`):=(`$\overline{expression\texttt{,}}$`)`
 | `(`$\overline{varid\texttt{,}}$`):=`*expression*

expression ::= ...
 | `(`$\overline{expression\texttt{,}}$`)`
 | *expression* `.` *int*

typedef ::= ...
 | `(`$\overline{typexpr\texttt{,}}$`)`

Notice that a valid tuple definition must have at least one type, because overline in the EBNF grammar is at least one element.

2.3.9 Lists

This section motivates and provides a customised syntax for lists.

A list has a fixed type and an arbitrary, but finite, number of values. The algebraic type list is a linked list that allows deconstruction with case patterns. Since a list stores values of only one type, a list is parametrized by the type of values it stores. The next example creates a list of numbers and then extracts the first number:

```
var e:[Int] ;
e := [1,2,3,4] ;
case e of [] then print(''Empty list'')
        | [h::t] then
             print(''First element in list is in h'') ;
             print(``The rest of the list is in t'')
       fo
```

The compiler translates the example into the following

```
data List[T] = Nil | List (head:T tail:List[T]) ;

var e:List[Int] ;
e := [1,2,3,4] ;
case e of Nil then print(''Empty list'')
        | List (head=_ as h,tail=_ as t) then
             print(''First element in list is in h'') ;
             print(``The rest of the list is in t'')
       fo
```

To realise this lightweight support for lists the *expression* production must allow lists as expressions, the *typexpr* production must allow lists as a type specification, and the *pattern* production must allow case expressions with either the empty list with `[]` or a head and tail specification with `[::]` .

expression ::= ...
| `[`$\overline{\textit{expression}\texttt{,}}$`]`

typexpr ::= ...
| `[`*typexpr*`]`

pattern ::= ...
| `[]`
| `[`*pattern*`::`*pattern*`]`

Chapter 3

The Functional Creol Compiler

This section provides an overview of the prototype Functional Creol Compiler found in Appendix E by explaining concepts and providing detailed a walk-through of selected code.

Recall from Section 1 that this thesis provides a prototype implementation of a Creol compiler, the Functional Creol Compiler. The implementation does in general adhere to the guidelines provided in Section 3.1. The job of the Functional Creol Compiler is mainly to read Creol programs, perform type checking and generate Creol Machine Code. All these operations are however concerned with how Creol programs are represented internally in the Functional Creol Compiler. The internal representation of programs in the Functional Creol Compiler is described by the University of Utrecht Attribute Grammar (UUAG) system which is discussed in Section 3.2. The internal representation of code is called an abstract syntax tree and Section 3.3 provides detailed information about the abstract syntax tree in the Functional Creol Compiler. An attribute grammar, introduced in Section 1.7, describes the abstract syntax tree that is built for the Creol program that is being compiled and how to traverse the abstract syntax tree. An abstract syntax tree is a program representation that is suitable for machine processing. The abstract syntax tree is traversed during compilation, for instance during type checking and code generation. The scanning phase takes a Creol program and uses a scanner to group characters into words, called tokens. The Functional Creol Compiler scanner is described in Section 3.4. The parsing phase takes the tokens produced by the scanner and uses a parser to build an abstract syntax tree. The parser in the Functional Creol Compiler is explained in Section 3.5 and is built with combinator parsers. Recall that combinator parsing was introduced in Section 1.6. The type checking phase traverses the abstract syntax tree and analyses the types to check if the program is type safe. The type checking is only commented briefly in Section 3.6 as the type system for Creol is formally presented in Section 4. The formalisation of the type system facilitates static type checking which greatly assists the programmer. The code generation phase traverses the abstract syntax tree and creates Creol Machine Code, which is text that conforms to the input requirements of the Creol Virtual Machine. Code generation for the Functional

Creol Compiler is described in Section 3.7.

3.1 Rationale and Design Goals

This section accounts for high-level design goals for the Functional Creol Compiler.

To make the implementation of the Creol compiler an accomplishable task, it was necessary to limit the scope of this thesis and to have some design guidelines. It is furthermore important that this is recognized as intentional rather than incidental. The following subsections describe the most important design decisions along with their rationale.

3.1.1 Separate Semantics and Syntax

Traditional approaches to creating compilers [63] often push semantic knowledge into the parser, to a degree where it is no longer possible to consider type checking as a separate aspect of the compiler. This weakens the modularity of the code and increases the effort required of others to comprehend and further develop the compiler. To separate between syntactic and semantic domains, the parser in the Functional Creol Compiler does not provide any semantic checking. The following tasks have been deemed part of the semantics and placed in the type checker:

- Existence of the class `Main` .
  The class `Main` is a special class which is instantiated when the program is executed.
- Existence of a method `run` in classes.
  The method `run` is a special method which is responsible for the active behaviour of the object in question. Without a `run` method, the object is strictly reactive, and only responds to external method invocations.

The parser is responsible for syntactical constructs that would otherwise be handled at a later stage. This desugaring is further described Appendix 6.1, and includes transformations such:

- Transformation of syntactical sugar to canonical form, such as expanding `id1,...,idN:AType` into `id1:AType,...,idN:AType`, that is, allow multiple declarations to share the same type, without writing the same type several times.
- Transforming interleaved definitions of classes and interfaces into separate lists.

3.1.2 Reuse of Existing Solutions

Implementation of software components from the ground up often requires considerably more effort to design, implement, learn and maintain, as compared to reusing an existing component with a small amount of modifications.

The Creol compiler uses a customised version of the standard scanner that comes with the University of Utrecht Attribute Grammar library, in contrast

to writing a new scanner especially for Creol. The customisation was needed because the University of Utrecht Attribute Grammar scanner hardcodes the literals denoting comments, and these literals clash with the Creol syntax. The appropriate pattern statements were modified to support Creol syntax. The University of Utrecht Attribute Grammar scanner also contains information about parsing, and this has been worked around through parametrisation of the scanner, although some of the parameters are meaningless from a Creol viewpoint, see Section 3.4.

3.1.3 Modularity Through Expressiveness

The Creol language is actively developed. It is to be expected that language aspects that have not been the focus of development so far, will be subject to change at a later time. It is therefore important to develop a compiler that can handle changes in the language design in a flexible way. Flexibility in the face of uncertainty with respect to expected changes is handled through the deployment of high-level tools. High-level tools with a high degree of expressiveness means that less code needs to change in the face of language evolution.

The work on abstraction of concepts and exploration of the general case has been pursued to avoid code duplication when possible.

3.2 Attribute Grammar System

This section is an introduction to the attribute grammar system that is used in the Functional Creol Compiler. It may be necessary to read this chapter to understand the code found in Appendix E.

The Functional Creol Compiler is programmed in Haskell and uses mainly the University of Utrecht Attribute Grammar system (UUAG). UUAG is first and foremost a support tool for the University of Utrecht Haskell Compiler (UUHC). This section is not an introduction to Haskell, but gives an overview of the UUAG system and how it is integrated with Haskell. It is recommended that this part is read briefly to get an overview of the library, and then consulted as needed when reading the code examples found in this thesis.

A major point in understanding UUAG is the understanding that an attribute denotes information associated with a node in a data structure. Both the node and the attribute are normal Haskell datatypes. To define the values of attributes the programmer must define the semantic functions, each attribute has a semantic function. The separation of concerns is achieved because the attributes and semantic functions of a node can be specified in separate parts. Given a node there can be arbitrary many specifications of attributes and semantic functions for these attributes. It is the piecemeal specification of attributes and semantic functions that realises flexibility and modularity with a separation of concerns. This form for separation of concerns gives flexibility similar to aspect-oriented programming [59].

3.2.1 Overview

An UUAG specification is a file with a special syntax that is translated into valid Haskell code. The special syntax is made up by specification of data,

with the DATA keyword, specification of attributes, with the ATTR keyword, and specification of semantic functions with the SEM keyword. Notice that one ATTR keyword can define attributes for several different DATA specifications. Each combination of DATA and ATTR requires a semantic function, some are given by SEM declarations and others are inferred by UUAG. Each attribute can be either inherited, chained or synthesised, as described in Section 1.7, and according to the attribute type, there is a default semantic function. The default semantic function for an inherited attribute is to copy the same attribute from a parent node. The default semantic function for a synthesised attribute is to copy the same attribute from the first child node that has it. A default semantic function for a chained attribute does both. The SEM declarations can also create local attributes. A local attribute is associated with a node, but is not visible from any other nodes, and as such is more a local definition than an attribute in the attribute grammar sense of the word attribute.

3.2.2 Details

The code for the Function Creol Compiler is written both as regular Haskell and as UUAG modified Haskell. The regular Haskell parts interact with the Haskell translations of UUAG modified Haskell, therefore it is necessary to know exactly how the UUAG modified Haskell is translated to regular Haskell, specifically how DATA, ATTR and SEM declarations are woven together to Haskell code. The DATA declaration is translated into a data where constructors are renamed to meet the Haskell requirement of unique constructor names. All the ATTR declarations that affect one given DATA declaration, are translated into a type declaration that defines how all relevant attributes flow into and out of the given DATA declaration. The SEM declarations that are relevant to a given DATA element are woven together to a semantic function that calculates all relevant attributes for the given DATA element. In pseudo syntax the type of the semantic function is

```
DATA -> Inherited1 -> ... -> InheritedX ->
  Chained1 -> ... -> ChainedY ->
  (Chained1, ..., ChainedN,Synthesized1, ..., SynthesizedY)
```

Intuitively the semantic function takes a DATA element and all the inherited and chained attributes, and produces the chained and synthesised attributes, as illustrated in Figure 3.1.

Translation of UUAG modified Haskell to normal Haskell is illustrated graphically in Figure 3.2. The figure also demonstrates how one DATA declaration may have several ATTR and SEM declarations, allowing an aspect-oriented programming style.

As Figure 3.2 shows, the expressions hexpr1 to hexpr4 define the semantics of the attributes. These expressions actually consist of Haskell code and are woven together as shown on the right side. Some of the code in the figure has been replaced with ... to hide unnecessary details, and the resulting expression of sem_Ast_Ast is a simplified version of the code that is actually generated. The SEM constructs assign arbitrary Haskell expressions to attributes.

The UUAG system has a custom syntax for assigning to and reading from attributes. The syntax used to assign to attributes occurs on the left hand side of = and the syntax to read from attributes occurs in the Haskell code on the

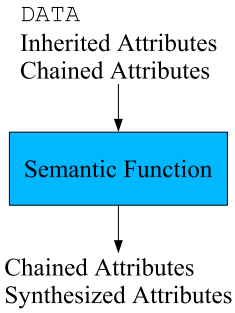

Figure 3.1: Semantic Function

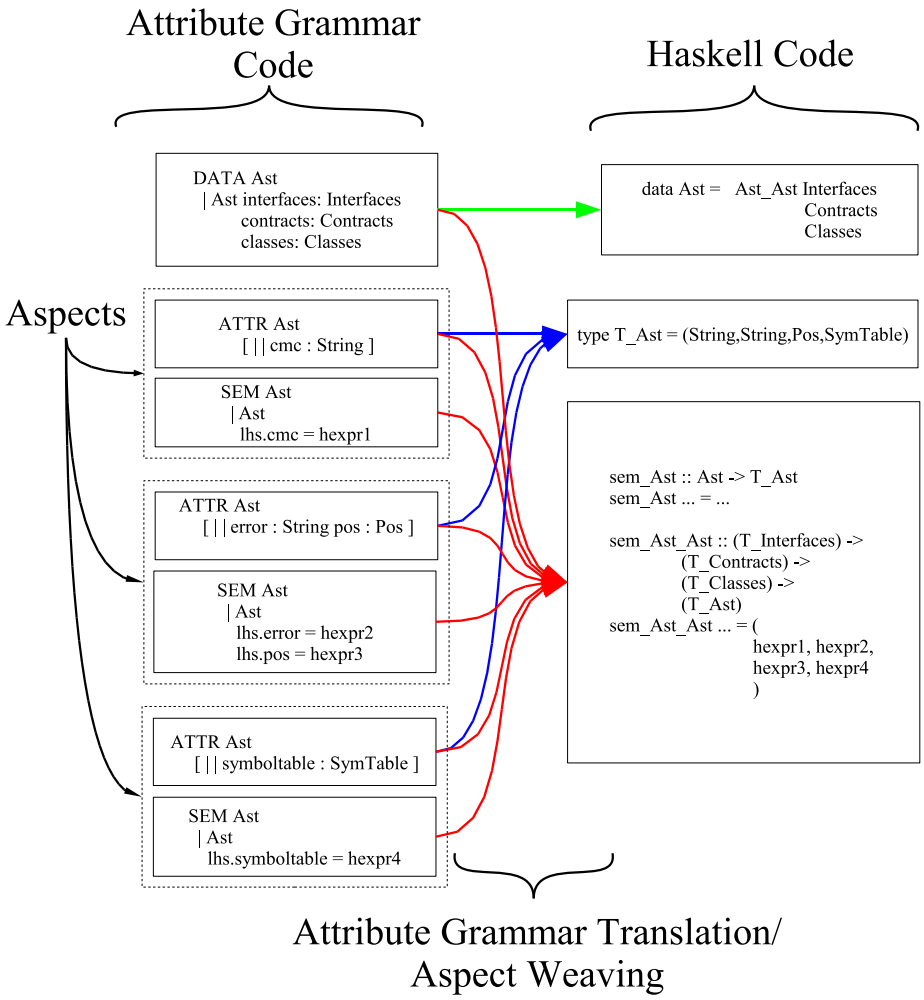

Figure 3.2: Translation from UUAG code to corresponding Haskell code

right hand side of `=`. The syntax uses the name of the attribute together with prefixes. The available prefixes are shown in Table 3.1.

Prefixes on Left Hand Side of =

Prefix	Meaning
`lhs.`	To an inherited or chained attribute produced by this element.
`loc.`	To an attribute local to this element.
`<id>.`	To an inherited or chained attribute in a sub-element named `<id>`.

Prefixes on Right Hand Side of =

Prefix	Meaning
`@lhs.`	From a synthesized or chained attribute given to this element.
`@loc.`	From an attribute local to this element.
`@<id>.`	From a synthesized or chained element from a sub-element named `<id>`.

Table 3.1: Attribute prefixes in `SEM` constructs

More details on the correspondence between UUAG and Haskell is beyond the scope of this thesis, but the interested reader may look in Chapter 3 of the UUAG lecture notes [30].

3.3 Abstract Syntax Trees

This section elaborates on the abstract syntax trees used in the Functional Creol Compiler.

An abstract syntax tree (AST) is a hierarchical representation of source code. The abstract syntax tree has a structure that is suitable for traversals of the tree for the purpose of analysis and transformation is practical. Due to convenience there is a correspondence between types of nodes in the abstract syntax tree and the productions in the EBNF grammar of the source code being represented. Although there are tools [6,91,60] that can generate abstract syntax trees directly from the grammar specification, there are no such available neither for the combinator parsers used in this thesis nor for combinators parsers in general.

This section gives an intuitive connection between Creol code and the corresponding abstract syntax tree. The exact structure of the abstract syntax tree is in the code Appendix E.4. The datatypes constituting the abstract syntax tree are structured similarly to those found in the University of Utrecht Haskell Compiler (UUHC) [92]. The University of Utrecht Attribute Grammar (UUAG) system has been created as part of the work on UUHC. UUHC constitutes the most practical and large scale deployment of UUAG. During the work with the Functional Creol Compiler the code from UUHC has been instructive. In particular the regular Haskell types `Option` and `List` are replaced with corresponding attribute grammar aware equivalents. This facilitates attributes on the equivalent elements, as attributes can only be given for declarations that are in the UUAG code. The `Option` type for node N corresponds to a `Maybe`N element and the `List` type for node N corresponds to the N`s` element. In the example this translation is referred to as "UUAG aware `Option`" and "UUAG aware `List`".

3.3.1 Example Abstract Syntax Tree

This section provides the abstract syntax tree for a sample program.

The following Creol program declares a class with one instance variable and one method that assigns a value to the instance variable.

```
class Main
begin
 var x:Int
 op assigntox() == x := 1
end
```

The abstract syntax tree for this small program is actually quite large. We shall start with an abbreviated version and explain that, and finally show the complete abstract syntax tree, just for demonstration. The abstract syntax tree nodes are described in UUAG but built with Haskell code. The Haskell code interacts with the UUAG provided Haskell translation of the abstract syntax tree. Therefore it is unfortunately necessary to know both the UUAG `DATA` and `TYPE` declarations and the corresponding UUAG translated Haskell `data` and `type` declarations. The principle of translation that UUAG provides to Haskell is explained in Section 1.7 and Section 3.2, and illustrated in Figure 3.2. The actual `DATA` and `TYPE` declarations for the Creol attribute grammar can be found in Appendix E.4. The relevant `DATA` and and `TYPE` declarations are now explained.

`DATA Ast` The `Ast` contains the whole program, which is represented as a list of classes.

`TYPE Classes` An UUAG aware `List` of classes.

`DATA Class` The name of the class, as well as optional instance variables and optional methods.

`DATA MaybeMethods` An UUAG aware `Option` for an UUAG aware `List` of methods.

`TYPE Methods` An UUAG aware `List` of methods.

`DATA Method` A signature and a body statement.

`DATA Signature` A method name and optionally input and output parameters.

`DATA Statement` A piece of code.

`DATA Expression` A piece of code with a value.

`DATA MaybeDeclarations` An UUAG aware `Option` for an AG aware `List` of declarations.

`TYPE Declarations` An UUAG aware `List` of Declarations.

`DATA Declaration` A variable and the type of the variable.

`DATA VarId` An identifier that starts with a lower case letter.

`DATA TypeId` An identifier that starts with an upper case letter.

The abbreviated definitions of these declarations are now given.

```
DATA Ast
     | Ast classes: Classes

TYPE Classes = [ Class ]

DATA Class
     | Class
       name : TypeId
       variables : MaybeDeclarations
       methods : MaybeMethods

DATA MaybeMethods
     | Nothing
     | Just methods : Methods

TYPE Methods = [Method]

DATA Method
     | Method
       signature : Signature
       code : Statement

DATA Signature
     | Signature
       name : VarId
       in : MaybeDeclarations
       out : MaybeDeclarations

DATA Statement
     | Assign
       name : VarId
       expr : Expression

DATA Expression
     | Int
       value : Int
       pos : Pos

DATA MaybeDeclarations
     | Nothing
     | Just declarations : Declarations

TYPE Declarations = [Declaration]

DATA Declaration
     | Var
       name : VarId
       typeid : TypeId

DATA VarId
     | VarId varid : String

DATA TypeId
```

```
        | TypeId typeid : String
```

The translation of these UUAG declarations into Haskell code is done by the UUAG compiler. The most relevant change is the renaming of constructors, and that the constructor arguments are without labels. An abbreviated version of the Haskell translation is now given.

```
data Ast = Ast_Ast Classes

type Classes = [Class]

data Class = Class_Class TypeId MaybeDeclarations MaybeMethods

data MaybeMethods = MaybeMethods_Nothing
                  | MaybeMethods_Just Methods

type Methods = [Method]

data Method = Method_Method Signature Statement

data Signature =
  Signature_Signature VarId MaybeDeclarations MaybeDeclarations

data Statement = Statement_Assign VarId Expression

data Expression = Expression_Int IntPos

data MaybeDeclarations = MaybeDeclarations_Nothing
                       | MaybeDeclarations_Just Declarations

type Declarations = [Declaration]

data Declaration = Declaration_Var VarId TypeId

data VarId = VarId_VarId String

data TypeId = TypeId_TypeId String
```

We are now ready to show what the abstract syntax tree looks like, and we shall progress in a bottom up fashion and start with the innermost and simplest parts, and for each step the whole program and the current development of the abstract syntax tree is shown. Each part of the current development is given the same color in both the code and in the abstract syntax tree. First we consider the constant `1`, which is an Integer expression.

```
class Main
begin
 var x:Int
 op assigntox() == x := 1
end

(Expression_Int 1)
```

The expression is used in an assignment, and the variable name is contained in a `VarId`.

```
class Main
begin
 var x:Int
 op assigntox == x := 1
end

(Statement_Assign (VarId_VarId "x") (Expression_Int 1))
```

This assignment is part of the `assigntox` method, which also has a signature with no out parameters and no in parameters, denoted by the constructor `MaybeDeclarations_Nothing`.

```
class Main
begin
 var x:Int
 op assigntox() == x := 1
end

(Method_Method (Signature_Signature (VarId_VarId "assigntox")
                                    MaybeDeclarations_Nothing
                                    MaybeDeclarations_Nothing
               )
               (Statement_Assign (VarId_VarId "x") (Expression_Int 1))
)
```

Then we look at the declaration of the instance variable `x`, which contains both the name of the variable and the type of the variable.

```
class Main
begin
 var x:Int
 op assigntox == x := 1
end

(Declaration_Var (VarId_VarId "x") (TypeId_TypeId "Int"))
(Method_Method (Signature_Signature (VarId_VarId "assigntox")
                                    MaybeDeclarations_Nothing
                                    MaybeDeclarations_Nothing
               )
               (Statement_Assign (VarId_VarId "x") (Expression_Int 1))
)
```

The class consists of the name of the class along with the instance variable declarations and the methods, however since declarations and methods are both optional they are wrapped in `MaybeDeclarations_Just` and `MaybeMethods_Just` respectively, and these further contain lists of elements, which is why the declaration and the method are placed in brackets.

```
class Main
begin
 var x:Int
 op assigntox == x := 1
end
```

```
(Class_Class (TypeId_TypeId "Main")
(MaybeDeclarations_Just [
(Declaration_Var (VarId_VarId "x") (TypeId_TypeId "Int"))
                        ])
(MaybeMethods_Just [
(Method_Method (Signature_Signature (VarId_VarId "assigntox")
                                    MaybeDeclarations_Nothing
                                    MaybeDeclarations_Nothing
               )
              (Statement_Assign (VarId_VarId "x") (Expression_Int 1))
)                 ])
)
```

The last part of building the abstract syntax tree is putting the structure for the class into a list that is contained by the `Ast` structure.

```
class Main
begin
 var x:Int
 op assigntox == x := 1
end

(Ast_Ast [
(Class_Class (TypeId_TypeId "Main")
(MaybeDeclarations_Just [
(Declaration_Var (VarId_VarId "x") (TypeId_TypeId "Int"))
                        ])
(MaybeMethods_Just [
(Method_Method (Signature_Signature (VarId_VarId "assigntox")
                                    MaybeDeclarations_Nothing
                                    MaybeDeclarations_Nothing
               )
              (Statement_Assign (VarId_VarId "x") (Expression_Int 1))
)                 ])
)
         ])
```

Which completes the abstract syntax tree. Notice that this is a simplified version, where only the relevant parts are presented. To demonstrate both the similarities and the added noise of even more information the actual syntax tree is now shown.

```
Ast_Ast []
[Class_Class "examples/Sample_Ast.creol"(line 1, column 1)
(TypeId_TypeId "Main" "examples/Sample_Ast.creol"(line 1, column 7))
MaybeTypeDeclarations_Nothing
MaybeDeclarations_Nothing
MaybeTypeDeclarations_Nothing
MaybeTypeDeclarations_Nothing
(MaybeDeclarations_Just
[Declaration_Var
(VarId_VarId "x" "examples/Sample_Ast.creol"(line 3, column 7))
(TypeId_TypeId "Int" "examples/Sample_Ast.creol"(line 3, column 9))
MaybeExpression_Nothing
```

```
])
(MaybeMethods_Just
[Method_Method
(Signature_Signature "examples/Sample_Ast.creol"(line 4, column 3)
(VarId_VarId "assigntox" "examples/Sample_Ast.creol"(line 4, column 6))
MaybeDeclarations_Nothing
MaybeDeclarations_Nothing
(TypeId_TypeId "Any" ))
MaybeDeclarations_Nothing
(Statement_Assign
(VarId_VarId "x" "examples/Sample_Ast.creol"(line 4, column 14))
(Expression_Int 1 "examples/Sample_Ast.creol"(line 4, column 19))
)])]
```

3.4 Scanner

This section presents the scanner used in Functional Creol Compiler.

A scanner is responsible for grouping characters into symbols, so for instance the word `class` is returned as a single symbol instead of as 5 distinct characters. The scanner also separates between keywords and identifiers, so `class` is recognised as keyword, and `x` is recognised as an identifier. The scanning and parsing process is illustrated in Section 3.5.2 for a part of the sample program from Section 3.3.1.

Scanners are usually implemented either as a hand written function or as a grammar that uses a program such as Flex [32] to generate a scanner function. During the work on the Functional Creol Compiler, it was planned to use a grammar and generate a scanner, but UUAG provides a hand written scanner function, and it was easy to base the Creol scanner on the UUAG scanner. The UUAG scanner is both patched and parametrized to fit the Creol syntax. This was deemed the fastest and most modular approach to get a working scanner for the Creol language, and in accordance with the goal of modularity and code reuse outlined in Section 3.1.2. The interface to the Creol scanner is also simpler than traditional approaches. Scanner generating tools like Flex assign a special code for each symbol, and emits the code for the scanned symbol. The Creol scanner returns the scanned string. The extra work required to create and maintain the list of scanner codes, makes the approach of the Creol scanner beneficial.

3.4.1 University of Utrecht Scanner

Altough the combinator parser could do lexical analysis, using a scanner increases the parsing speed and provides a modular approach. The modularity occurs because the scanner is less likely to change than the grammar, and the grammar is simpler without the scanner included.

The UUAG system comes with a scanner that is tailored to parse Haskell syntax. In order to apply the scanner to Creol syntax, it has to be adapted. This is done by patching the scanner, and the patch is found in Appendix E.5.

The UUAG scanner defines several functions that transfer information from the scanner to the parser. These functions behave as parsers, and are the

simplest parsers available. The following scanner to parser functions are used by code in this thesis:

`pVarid` Scan and return an identifier that starts with a lower case character.

`pConid` Scan and return an identifier that starts with an upper case character.

`pKey` s Scan and return the identifier or operator s, given that the scanner was parametrised with s.

`pSpec` c Scan and return the character c, given that the scanner was parametrised with c.

`pSucceed` e Return the Haskell expression e without doing any scanning. This is for instance used to insert default information while scanning.

3.5 Parser

This section describes the parser used in the Functional Creol Compiler.

A parser is responsible for validating the sequence of symbols, so for instance the keyword `class` must be followed by an identifier that names the class. When the parser has validated the sequence of symbols, it builds the abstract syntax tree for the parsed symbols, hence the parser automates the building of an example abstract syntax tree as demonstrated in Section 3.3.1. The Creol parser is built using parser combinators [45] as introduced in Section 1.6. Before the presentation of the Creol parser, the combinator parsing approach is compared with traditional parsing methods. The combinators used in the Creol parser are explained in detail. An excerpt of the Creol parser, from code Appendix E.6, is explained in Section 3.5.2 for an extract of the Creol example from Section 3.3.1. The Creol parser was an interesting experience in the creation of the Functional Creol Compiler, because combinator parsing is a novel approach that gives more readable code and better error messages than Yacc.

The reader is assumed to have knowledge about basic parsing techniques such as LL(k), which is a top-down method that scans the input from left to right while conducting a leftmost derivation with at most k symbols of lookahead, and LALR(1) which is a bottom-up method that scans the input from left to right while conducting a rightmost derivation, in reverse, with at most 1 symbol of lookahead. Readers who want a short introduction on the topic of parsing including LL(k) and LALR(1) are referred to an introductory text [78].

LL(k) parsers, such as those crafted by hand or produced by a tools such as ANTLR [6], are normally specified in a manner similar to recursive functions, which is a very intuitive and programmer friendly method of specification. LALR(1) parsers are specified with a grammar and then translated into a function by programs such as Yacc [101] or Bison [9], because it is infeasible to hand write a LALR(1) parser directly. LL(k) parsers are quite popular even though LALR(1) parsers can parse a larger set of languages and handle left recursive grammar rules directly. The popularity of LL(k) parsers is due to the simplicity, as supported by a quotation from the ANTLR [6] documentation:

> ANTLR is popular because it is easy to understand, powerful, flexible, generates human-readable output, and comes with complete source.

The parser combinator approach is superficially similar to the naive method for specifying LL(k) grammars; top-down and similar to recursive function calls. Specifically the University of Utrecht combinator parser library [88] combines decent parsing speed with a relatively simple specification of the grammar and automatic error reporting and correction. The UUAG parser combinators can parse languages that are beyond both LL(k) and LALR(1) parsers [89, Sec. 6]. The UUAG parser combinators can parse LALR(k) languages for any k necessary.

The UUAG parsing library allows the use of greedy and non-greedy functions to implicitly control backtracking. Backtracking occurs when a parse that succeeded must be undone to try out other possible parses. When a parser fails it can either backtrack and try another parse, or it can correct the input, hence the term *error correcting parsers*. The choice of whether to backtrack or correct is determined by greediness. A greedy parser will correct input, while a non-greedy parser will backtracked. The UUAG parser combinators allow the programmer to select combinators with greedy or non-greedy behaviour as appropriate.

The combinator interface to the parser library facilitates transformations so the actual parse algorithm uses tables rather than recursive function calls. The use of tables during parsing is the rule for LALR(1) tools but is also deployed by the LL(k) tool ANTLR, and is the key to achieve usable combinator parser efficiency.

3.5.1 UUAG Combinator Parser Library

The parser in the Functional Creol Compiler is built from the parsing library found in UUAG. The combinators offered by the library are documented by their Haskell source code, we therefore provide an intuition for the combinators used by the Functional Creol Compiler parser. The example parsing in Section 3.5.2 shows an excerpt of how these parser combinators are used in the Functional Creol Compiler parser.

The University of Utrecht library offers a variety of combinators for building parsers. The combinators relevant to this thesis are now described, where p and p' are valid parsers that parse values v and v', e is a valid Haskell expression and m is an error message. Notice that the parser p parses text and returns a value v, and that combinators combine parsers by specifying both how the parsers are combined and how the values of the parsers are combined.

UUAG Combinators

p `<?>` m — Try to parse p and return v, if that fails give the error message m.

p `<|>` p' — Choice, try to parse p and return v, if that fails try to parse p' and return v'.

p `<*>` p' — Sequence, parse p and then p', and return $v\ v'$, which is the value from p applied to the value from p'.

p `<*` p' Parse p and p' in sequence, and return v.

p `*>` p' Parse p and p' in sequence, and return v'.

e `<$>` p Parse p and and return the value $e\ v$, which is the Haskell expression e applied to the value v from the parser p.

e `<$` p Parse p and return e.

p `<**>` p' Parse p and p' in sequence, and return $v'\ v$, which is the value from p' applied to the value from p.

p `<??>` p' Parse p and try to parse p'. If p' is parsed return $v'\ v$ else return v.

e `<$$>` p Parse p and return $\lambda x.e\ x\ v$. Intuitively e is a function that takes two arguments, and the value v is used as the second argument, while the first argument is left open.

p `<+>` p' Parse p and p' in sequence and return a tuple (v, v').

p `` `opt` `` e Return v if p could be parsed, else return the Haskell expression e. If p is parsed it can not be backtracked.

`pAny` e e' Map the Haskell expression e on the list in the Haskell expression e', then fold the resulting function with `<|>` and return this parser. The `pAny` combinator function is typically used with `pKey` such as `pAny pKey ['a','b','c']` which is equivalent to `(pKey 'a') <|> (pKey 'b') <|> (pKey 'c')`, thus trying to parse one of the letters `a`, `b` or `c`, returning the first possible parse.

UUAG Iterative Combinators

`pList` p Repeatedly use the parser p and return a list of the parsed elements.

`pFold` e p Repeatedly use the parser p to parse the list $v_1 \dots v_n$, then fold the list with the Haskell expression e and return the folded list $e\ v_1\ (e\ v_2\ (\dots (e\ v_{n-1}\ v_n) \dots))$.

`pChain` p p' Repeatedly use p and p' to parse the values $v_1 \dots v_n$ and the operators $v'_1 \dots v'_{n-1}$, then return the values chained with the operators, $(v'_{n-1}\ \dots (v'_2\ (v'_1\ v_1\ v_2)\ v_3) \dots v_n)$.

The combinator functions for iteration are provided with several flavours for further customisation, the different flavours are formed by adding a suffix to the combinator function name.

UUAG Iterative Combinator Suffixes

`r` Fold or Chain from left to right.

`l` Fold or Chain from right to left.

`-Sep` p List of Fold with a separator found by parser p.

`1` List with at least one element.

`_ng` Allow backtracking to undo this parse.

`_gr` Never backtrack this parse.

To parse a list of strings, such as

```
"one" "two" "three"
```

the function `pList pString` will produce the correct result. In order to add the additional requirement that there must be at least one string we use a combinator with the suffix `1`, and get `pList1 pString`. Both these parser functions return a list of the parsed strings.

Contributed Combinators

The creation of the Functional Creol Compiler lead to the creation of new combinators. These new combinators make the parser code less verbose. These contributed combinators are found in Appendix E.9.

p `<->` p' Parse p and p' in sequence and return $(\lambda(x, y).v\ x\ y)\ v'$. Intuitively the parser p' returns a tuple with two values, and the value v from the parser p is applied to the values from the tuple. The parser p' is presumably made by the `<+>` combinator.

e `<$->` p Parse p and return $(\lambda(x, y).e\ x\ y)\ v$. Intuitively the parser p returns a tuple with two values, and the Haskell expression e is applied to the values from the tuple. The parser p is presumably made by the `<+>` combinator.

The contributed combinators complement the existing `<+>` combinator. Successive use of the combinator `<+>` combines the result of parsers into nested tuples such as $(v_1, (v_2, \ldots (v_{n-1}, v_n) \ldots))$. Successive use of the combinator `<->` applies the result of a parser to to each value in a nested tuple. The combinator `<$->` applies a function, not a parser, to each value in a nested tuple.

3.5.2 Parsing Example

This section provides a walkthrough of a small parsing example.

A small subset of the Creol program from Section 3.3.1 is parsed, specifically the line

```
x := 1
```

This might seem like an overly simple example, but the corresponding parsing functions rapidly become complex due to the compact handling of operator precedence, although not more complex than with alternative parsing methods.

The body of a method is a `Statement` element, whose parser is defined by `pStatement`. Unfortunately the parser `pStatement` is complex due to how statements can be combined. We restrict ourselves to look at a subset of the more basic `pStatement_Nullary`, that handles the parsing of a single primitive statement, primitive in the sense that it consists only of simpler elements and not combinations of statements. A sketch of `pStatement_Nullary` is as follows:

```
pStatement_Nullary =
      ...
  <|> Statement_Assign
        <$> pVarId
        <* (pKey ":=")
        <*> pExpression
  <|> ...
```

The `<|>` combinator designates alternatives, and this code snippet shows that there are other parsing alternatives to consider as well, which are not shown here. Now to break this down, each of the code-lines are presented along with a detailed explanation of how they work. First the combinator operator precedence is made explicit with bracket annotations:

```
(((Statement_Assign <$> pVarId)<*(pKey ":=")) <*> pExpression)
```

It is assumed that the reader is familiar with partial application of functions and Haskell or ML notation for function types. The constructor `Statement_Assign` is from the Haskell code produced by the UUAG translation, described in Section 3.3, of the `DATA Statement` found in Appendix E.4. The constructor `Statement_Assign` takes two arguments, a `VarId` and an `Expression` and returns a `Statement`, it has type

$$\texttt{VarId} \rightarrow \texttt{Expression} \rightarrow \texttt{Statement}$$

Let us look at the innermost part

```
(((Statement_Assign <$> pVarId)<*(pKey ":=")) <*> pExpression)
```

The combinator `<$>` takes a function, in this case `Statement_Assign`, and applies it to the result from `pVarId`, which again is a parser for a variable identifier. Thus this part has the resulting type

$$\texttt{Expression} \rightarrow \texttt{Statement}$$

The next part of the code is

```
(((Statement_Assign <$> pVarId)<*(pKey ":=")) <*> pExpression)
```

The combinator `<*` requires the parser `pKey ":="` to succeed, but then throws away the result and keeps the existing function, that still is of type

$$\texttt{Expression} \rightarrow \texttt{Statement}$$

The parser `pKey ":="` reads the string `:=` from input, and this must match a corresponding operator keyword definition as found in Appendix E.5. The outermost part of the code

```
(((Statement_Assign <$> pVarId)<*(pKey ":=")) <*> pExpression)
```

uses the `<*>` combinator that applies the parsed result on the left to the result from the `pExpression` parser on the right. This gives the type `Statement` and this is the correct result. Note the distinct difference between `<$>` and `<*>`, in that the left argument of `<$>` is not required to be a parser, but can be any

legal Haskell function, such as the `Statement_Assign` constructor, while the left argument of `<*>` is a parser, which is in accordance with the explanations for combinator parsers found on page 44.

Note that the final result does in reality not have the type `Statement`, rather it has the potential of becoming a `Statement`, whenever the parser is applied to some text. Thus it has the type `Parser Token Statement` which means, that it is a `Parser` that accepts some `Token` and eventually returns a `Statement`.

As such the combinators build computations that may later be realized. In our case the computation is the parsing of tokens. This is an important point in grasping combinator parsing, although from a practical perspective a person should be able to alter the parser using intuition rather than needing a formal understanding. Readers interested in building computations might find a paper on monads interesting [45], and those interested in a generalisation of the monadic paradigm might also find an article on arrows interesting [44].

3.6 Type Checking

This section gives a rough overview of the type checking process, and motivates the type analysis in Section 4.

The Creol language was crafted under an assumption of static type safety, however no such type system was ever devised. The assumption of static type safety is crucial for Creol. Creol provides program analysis as explained in Section 2.1.3. Program analysis improves the dependability of running code. Dependability is only interesting in the absence of those errors prevented by by static type safety. If a program crashes, by lack of static type safety, then the result of a verification is of no interest. Therefore the Creol language depends on static type safety. The work on this thesis started with the initial construction of the Functional Creol Compiler found in Appendix E. During the work with the code it became clear that a formal type system foundation was required to successfully type check the Creol language. The naive algorithmic approaches to type checking, without a type system foundation, that where first deployed in the Functional Creol Compiler failed to type check parts of the Creol language. It became clear that is was unknown if, and how, static type checking could be performed for the Creol language.

The need for both a Creol type system and for an integration of the central concepts of the Creol language with a Creol type system, required investigation of type systems in general. The investigation of type systems, the analysis of Creol concepts with regard to type systems, and the creation of a Creol type system is presented as type analysis in Section 4.

Type analysis provides an important foundation that guides the implementation of static type checking in the Functional Creol Compiler. The Functional Creol Compiler is however in the middle of a rewrite from an earlier version of the type system. The rewrite aims to implement the Creol type system presented in this thesis.

3.7 Code Generator

This section looks at how Creol Machine Code is generated. Some of the required operations are outlined and an actually compiled example is demonstrated. Deficiencies of the Creol Machine Code that affect the code generator are mentioned.

The process of generating Creol Machine Code is a small part of the Functional Creol Compiler because the Creol Virtual Machine is very sophisticated. The generated Creol Machine Code looks very much like Creol code, although with some minor syntactic changes. That is, the classical code generation related tasks are not necessary [7, Sect. 6-10,13,17-21].

The actual Creol Machine Code is just a string that is built up piecemeal by attributes on the nodes in the abstract syntax tree. The most complicated parts of the code generator is the generation of unique labels in a functional setting, the context dependency of the Creol Machine Code syntax, and type dependent function translation.

The generation of unique labels can be found in Appendix E.8, and is greatly simplified by the attribute grammar system, because the attribute grammar system generates default rules, hence unique attributes are only apparent in the code where they are used. This is in contrast to the normal functional approach where the unique attribute would be passed explicitly through every function all the way down to where it was needed. One could of course use a state monad for the generation of unique labels, but a state monad could not interact with the other attributes, that could not be expressed by any state monad. UUAG allows information to flow in several directions, while the state monad directs the flow of information, hence a state monad could not be used for label generation.

The context dependent code generation requires extra attributes for flow of information in the abstract syntax tree. Consider parameters in a method. If there are no parameters the Creol Machine Code keyword `nil` is emitted, if there are parameters they are emitted as a comma separated list. Therefore the code generation for parameters has a special handling for the absence of parameters. The code generator would be simplified if the Creol Machine Code accepted parameters with the same representation regardless of number of elements, such as `()` for an empty list and `(1,2,3)` for a list of three elements, there would be no special case handling of empty lists. Notice that these lists are internal to the Creol Machine Code, and are not related to Creol lists.

There is little overloading in Creol, only the `+` operator is overloaded in the Creol language. The `+` operator is not overloaded in the Creol Machine Code, therefore the code generator uses the type of the arguments to select the correct Creol Machine Code. This special case of overloading is not handled by the Creol type system, and is provided in the code generator as a compliance hack, as there is no support for procedure or operator overloading in the Creol type system.

The Creol Machine Code is defined by the Creol Virtual Machine by principle of source inspection, and there is no standard definition of Creol Machine Code. This is not as bad as it may appear, as the Creol Virtual Machine is the semantic specification of the Creol language. Although the Creol Virtual Machine is high level it requires some knowledge of Maude [66] to be read and understood. The work by Arnestad [8] that defines the first version of the Creol Virtual Machine provided a stepwise high level translation from Creol to Creol Machine

Code, although the translation is now outdated due to further development of the Creol Virtual Machine. The formal translation provided by Arnestad was only partially correct when the work on this thesis began. The code generator for the Functional Creol Compiler was created partially as a transcript of the translation provided by Arnestad, partially by inspection of the Creol Virtual Machine, and partially by inspection of existing Creol Machine Code.

3.7.1 Creol to CMC Example

This section provides the Creol Machine Code for an example class.

The Creol Machine code is generated by the Functional Creol Compiler, and provides an illustration of the close relation between Creol and Creol Machine Code. Recall the example class from Section 3.3.1, now with color highlighting to show the correspondence between Creol and Creol Machine Code. The non colored parts of the Creol Machine Code can be ignored. A detailed explanation of Creol Machine Code is not the subject of this thesis.

```
// Creol Class
class Main
begin
 var x:Int
 op assigntox == x := 1
end
```

As evident from the translation, Creol Machine Code is similar to Creol code.

```
// Creol Machine Code
< 'Main : Cl |
   Inh: nil,
   Att: ('x : null),
   Mtds:  < 'assigntox : Mtdname |
            Latt: no,
            Code: ('x :=  int(1)) ;
                  end( nil)
          >  ,
  Ocnt: 1
>  .
```

For an example comparison between earlier hand written and now compiler generated Creol Machine Code, consult Appendix C.

Chapter 4

Object-Oriented Type Analysis

This section investigates the meaning of static type safety, introduces the terminology used to discuss object-oriented type checking, and investigate the object-oriented challenge in separating inheritance from subtyping.

4.1 Static Type Safety

This section clarifies what static type safety entails in general, including the Creol language.

Creol aims to be a statically type safe language. A static type safe language must provide a typechecked program with safety from untrapped errors, as well as guarantee at compile time that no type errors may occur at run-time. An untrapped error is an error that is not noticed by the program. A type error is an error decided just by looking at the types. This is in accordance with static type safety as defined by Bruce [15, Sec. 13.3]. An example of an untrapped error is accessing past the end of an array. An example of a type error is treating an object reference as an integer, or an integer as a character.

It is important to avoid untrapped errors and type errors because they yield programs that do not operate with predictability as to when the program diverges or aborts. Type checking is used to avoid type errors. Type checking is be static when applied at compile time, and dynamic when applied at run-time. The choice between static and dynamic type checking can not be made freely. Static type checking can always be replaced by dynamic type checking. Dynamic type checking can only be replaced by static type checking withing the limits of decidability. The prevention of untrapped errors is only partially solved by typechecking, because there are untrapped errors that are not type errors. To access beyond the end of an array is not a type error. The untrapped errors that are not type errors require run-time checks to trap them. The insertion of such checks are not a part of the formal type system, rather it is part of the code generator in the compilation process.

Some run-time checks can be avoided by analysis, but due to Richardson's theorem [96], which states that it is statically undecidable if an arithmetic expression of at least a certain complexity will ever become zero, there can be

no general solution to statically check, for instance, division by zero. To prove statically that division by zero could not occur for an expression $\frac{\text{divident}}{\text{divisor}}$ would limit the divisor to Presburger arithmetic [95], which is a very restricted form of arithmetic, containing only the natural numbers and addition, and hence be usable for general programming. Note that a truth decision for Presburger arithmetic, such as the question does the divisor become zero, provably requires more than polynomial run time. The type system for FISh [47] show how an enriched type language can eliminate certain array bounds checks by means of static type checking. This approach is taken even further by Xi's work on dependent types [100]. These solutions are however both partial and subject to ongoing research, therefore such solutions are not considered for the Creol type system, and run-time checks are used to catch errors of, for instance, array bounds checks for array indexing, division by zero, and the occurrence of null-pointers.

In the context of the Creol reseach, the Creol language must provide static type checking in order to realise the benefit of both invariant analysis of concurrent programs and correctness proofs for programs. Concurrency analysis is interesting as it guarantees that a concurrent program will maintain an invariant, without having to run the program to check this property. If the program must be debugged to catch untrapped errors, the whole idea of program analysis instead of debugging is lost, and therefore static type checking leverages the benefit of concurrent program analysis. Static type checking also facilitates run-time efficiency, although run-time efficiency is not at present a research goal of the Creol language development.

Static type checking of the Creol language forms a foundation for concurrent program analysis, and therefore the static type checking should catch as many errors as possible. The ML family of languages, which includes O'Caml [73], has chosen to pursue static type checking as far as feasible, with respect to decidability, although for different reasons than Creol, and the research in relation to these languages is re-usable for the Creol language.

4.2 Essential Type Terminology

This section introduces essential terminology that is used to discuss type checking in general and object-oriented type checking in particular.

4.2.1 Subtyping

Subtyping is a relation $<:$ between a subtype and a super type. Subtyping states that by the *principle of safe substitution* [77, Sect. 15.1] a value v' of type τ' can be used at any place where a program expects a value v of type τ when $\tau' <: \tau$, that is, τ' can subsume τ. The term subsumption denotes the phenomena that τ' can subsume τ, which intuitively means that τ' can masquerade as τ. A small example with subtyping is available in Section 7.1.4.

4.2.2 Object Type

An object type is described by rows. There are different rows for instance variables, inheritance, methods, virtual methods and so on.

Object types are either open or closed. An open object type can be refined by adding more rows at a later point. A closed object type can not be refined. Technically an object type is open when it has a row variable ρ. The row variable ρ can later be replaced with rows that refine the object type. An object type without a row variable ρ is closed.

Intuitively an open object type is a protocol description, as discussed in Section 4.6. A closed object type corresponds to a physical object, which is no longer extensible. A closed object type corresponds to an exact type, as discussed in Section 4.7.

4.2.3 `this`, `self` and `Self`

The type of the object, which the current method executes inside, is denoted `self`. The term `this` is used inside classes to refer to the object instance itself, and `this` has type `self`. Other objects from the same class have type `Self`. The type `Self` is obtained from `self` by removal of information that is private to the instance. Both `self` and `Self` are open object types because they may be refined by inheritance.

4.2.4 Matching

Matching is a relation $<\#$ between a subobject and an open super object, such that the subobject at least can handle all the method invocations the super object can handle. This is a relaxation of subtyping by the removal of subsumption. Matching captures a notion of protocol requirement, and it is possible to verify statically that the subobject is a protocol extension of the protocol described by the open super object. The difference between matching and subtyping, with respect to recursive types, is apparent in the treatment of fixpoints. This is discussed later in Section 5.17.1 and Section 5.17.2.

4.2.5 Record

A record is a collection of values. A record type describes the collection of values where each value is given a name and a type. The name is used to extract values from a record. The type is used to provide static type safety.

4.2.6 Variant

A variant is a tag and a value. A variant type describes a set of variants. Each variant has a tag and the type of the contained value. The tags are used to distinguish variants with respect to the variant type the appear in. Variant types are also referred to as sum types.

4.2.7 Conformance

Conformance is a relation $\ddot{<:}$ between a subtype and a super type. The conformance relation reduces to matching $<\#$ when the super type is an open object, and to subtyping $<:$ for all other super types, therefore the conformance relation $\ddot{<:}$ express both subtyping and matching in the Creol type system. A type τ^b conforms to τ^a when $\tau^b \ddot{<:} \tau^a$ holds.

4.2.8 Nominal Conformance Constraints

Nominal conformance constraints restrict the conformance relation $\ddot{<}$, so conformance is only allowed when declared by the programmer. Nominal conformance constraints correspond to nominal subtyping in other type systems.

4.2.9 Behaviour

Behaviour denotes semantic properties attached to an interface, without regard for the nature of these properties. Conformance preserves behaviour when nominal conformance constraints are respected, which is denoted behavioral conformance, or behavioural subtyping.

4.2.10 Variance

Variance is the nature of a type difference between corresponding parts of an object from a subtype and a super type.

4.2.11 Contravariance

Contravariance is variance where a component in the subtype is a super type of the corresponding component in the super type. One may see contravariance as contraintuitive because the variance is in the opposite direction of the inheritance relation of the surrounding types.

4.2.12 Covariance

Covariance is variance where a component in the subtype is a subtype to the corresponding component in the super type. Intuitively covariance has variance in the same direction as the inheritance relation of the surrounding types.

4.2.13 Invariance

Invariance is variance that requires both co- and contravariance at the same time, which in general leads to type equality. That is $\tau^a \ddot{<} \tau^b$ and $\tau^a \ddot{>} \tau^b$ at the same time which requires $\tau^a \ddot{=} \tau^b$.

4.2.14 Virtual Binding

Virtual binding is an algorithm for lookup of methods or variables in objects. Virtual binding searches for methods or variables at run time, however static type safety guarantees that virtual binding always finds a method at run-time, that is, never throws a *message not understood* error. Since virtual binding happens at run-time the lookup strategy is part of the Creol semantics.

4.2.15 Static Binding

Static binding is an algorithm for lookup of methods and variables in objects, where the lookup is confined to those methods available prior to a given point in the inheritance hierarchy. The type system must record the inheritance hierarchy to statically prevent *message not understood* errors.

4.3 The Problem of Inheritance and Subtyping

This section discusses the problem of inheritance and subtyping, with focus on code reuse with static typing.

Most programming languages with static type checking are such that inheritance implies subtyping, that is, inheritance is restricted by subtyping, which is necessary to statically ensure type safety. Languages with dynamic type checking does not restrict inheritance by subtyping.

When inheritance implies subtyping, there are lost opportunities for code reuse. The importance of this code reuse is demonstrated in a study by William Cook [22] of the libraries in SmallTalk. SmallTalk is a dynamically typed language, with no compile time restrictions on inheritance. The study discovered that the SmallTalk libraries could be statically type checked through a separation of inheritance from subtyping. The need to distinguish between inheritance and subtyping is well known. Both America [3] and Cook, Hill and Canning [23] address the necessity of separating inheritance from subtyping.

We shall call a typing scheme for inheritance simple, whenever inheritance implies subtyping. Surprisingly, even simple typing schemes [23, Sec. 3.1] based on very sophisticated type systems, such as System F [77, Sect. 23.3], causes problems. The problems are prompted by the presence of binary methods. A binary method, is a method, which has an input parameter, with the type of the class, that defines the method. The problems occur in any language that combines subsumption with method override, and subtyping is defined through subsumption.

In contrast a typing scheme for inheritance is called sophisticated, when inheritance does not imply subtyping. Intuitively, a sophisticated typing of inheritance, keeps the type of `self` open, that is adjustable, in the presence of inheritance. Even with this knowledge readily available, the most commonly used object-oriented languages, namely C++ [86, Sec. 12.2], Java [36, Sec. 4.5.6] and C# [19, Sec. 17.2.1], have a simple typing.

The simple typing of inheritance should be abandoned for a sophisticated typing of inheritance, which increases code reuse. However, the exact nature the sophisticated typing is unsettled, so we must investigate which sophisticated typing of inheritance to choose.

4.4 Possible Approaches

This section establishes a list of possible approaches to the problem of inheritance and subtyping , by investigation of literature.

To establish some alternative approaches let us start with an article which compares solutions to the problem of binary methods [16]. The article is inconclusive in the sense that no solution is best, each approach has different strengths and weaknesses. The considered approaches are now briefly mentioned.

The first approaches are not solutions, they avoid binary methods and suggest alternatives that can express something similar, although with code duplication.

Functions This suggests to write binary methods as functions where each combination of object types is defined by a different function.

Pair Class This suggests to create a special class that contains two copies of all instance variables from the class that needs a binary method. In general this requires a pair class for binary methods, a triple class for ternary methods, and so forth.

The next approaches handle allow binary methods through different advanced type systems.

Multi-Methods Use the run-time type of all arguments to do multiple dispatch.

Precise typings Change the type of binary parameters so they describe exactly the minimum requirements of the parameter, so the parameter is simplified and no longer binary.

Matching with MyType Introduce a special ground type, or placeholder type, called MyType and give special type judgements for classes with MyType. These MyType judgements models matching, where matching is a weaker relation than subtyping.

The set of solutions that are considered by the article does not include fixpoint recursive extensible records with polymorphic row variables [80], which is the solution to binary methods that is deployed in the programming language O'Caml [73]. For the sake of brevity the term *rows* as a shorthand for *fixpoint recursive extensible records with polymorphic row variables*. Rows naturally encode objects, such that type equality between object types expresses matching. Since rows variables are have a solid track record, both theoretically [80, 81, 79, 27] and practically [73], we include row variables in our investigation.

Not all of these approaches to binary methods are viable for the Creol language. The gained language expressiveness, by allowing binary methods, is important to the Creol language, as noted in Section 2.1.2, so the the approaches that avoid binary methods are not viable.

Compositional analysis, which is supported by compositional type checking, and static type safety, are important properties in the Creol language. Multimethods either require a complicated type system [20], that makes compositional type checking difficult, or loose static type safety [16, Sec .6].

Precise typings require either extensive code annotations, or partial type inference. Extensive code annotations is very obtrusive, and it is unclear if partial type inference can be combined with behavioural subtyping.

Other approaches encountered are either variations of those identified, or unsuitable for some reason. Dependent types [99] appear as a variation of row variables. Pattern matching on object types [67] appear as a variation of multimethods. Nested inheritance [72] does not address the problem.

The problem of overloading, in functional languages, is closely related to the problem of binary methods, in object-oriented languages, so we consider solutions to overloading. We have found two theoretical approaches to overloading in general, for functional languages, type classes [39, 57] and extensional polymorphism [33].

Type classes, which by definition do not allow subsumption, have successfully been used to model objects [70], however virtual binding is only provided for external dispatch, and not for internal dispatch. It is called internal dispatch

when a class invokes a method on itself or one of its super classes, otherwise it is called external dispatch. This restriction does not sit well, neither with the Creol language, nor with most other object-oriented languages.

Extensional Polymorphism is a framework for general pattern matching on types, in combination with static type checking, and generalises type classes. Both method overloading and multi-methods are limited forms of pattern matching on types, and can be modelled with extensional polymorphism. Extensional polymorphism and algebraic datatypes does at least have the same expressiveness as object-oriented languages, however the notions of classes and objects are lost.

The approaches to overloading may solve the problem of binary methods, at the cost of removing features that are conceived as essential to object-orientation. It is not clear how to express behavioural subtyping, neither with objects encoded by type classes, nor with extensional polymorphism, and the Creol language requires behavioral subtyping. We therefore do not consider these solutions fit for the Creol language.

The approaches to binary methods that are left to consider for the Creol language are matching with MyType and matching with rows. Observe that matching preserves behaviour in Creol [51,53], when nominal restrictions posed by interfaces are satisfied, and this is separates the notion of behavior in Creol from other notions of behaviour [4,62], which we are not concerned with. Matching with MyType does, by definition [15, Def. 10.2.4,], enforce nominal constraints. Matching with rows do not enforce nominal constraints, and is therefore not compatible with behaviour, however, the Creol type system defines, an apparently original, concept of nominal rows, which enforce nominal constraints. This is discussed further in Section 5.6.

4.5 Matching with MyType

Matching with MyType is covered by Bruce's textbook [15]. Matching is realised by the introduction of MyType. MyType is a special ground type, that is interpreted as the current class, by type judgements, so MyType corresponds to the type of `self`, and MyType is reinterpreted on inheritance, by type judgements. Matching is generalised to match-bound polymorphism, which unlike quantified polymorphism in system F [14], provides a sophisticated typing of inheritance. Abadi and Cardelli compare subtyping with matching as well as match-bounded polymorphism with quantified polymorphism in system F [1]. They demonstrate that matching separates inheritance from subtyping, hence a larger set of programs can be statically type checked. The comparison between match-bounded polymorphism and quantified polymorphism in system F, is summarized by a quote from Abadi and Cardelli, where higher-order interpretation refers to match-bound polymorphism:

> Thus, we believe that the higher-order interpretation is preferable; it should be a guiding principle for programming languages with matching.

Later Abadi and Cardelli also investigated how the notion of MyType and match-bounded polymorphism can be encoded into their calculus of objects $\mathbf{Ob}_{\omega<:\mu}$ [2, Sec. 21].

The presentation of MyType and matching [15, Sect. 18] ends with an experimental language $\mathcal{NOOL}$, where subtyping is replaced by hash types. A hash type is prefixed by the symbol #. Hash types allow the programmer to use match-bound polymorphism in a natural manner. This is demonstrated by the interpretation of `a : C` and `a : #C`, where `a : C` declares `a` to have the exact type `C` while `a : #C` declares `a` to have a type that is match-bound by `C`. The need for this distinction becomes apparent with binary methods. A binary method requires an exact MyType, while a hash type requires a matching type. This idea was further pursued by Bruce and Foster [17], by extending Java with matching, MyType and hash types.

4.6 Matching with Rows

The idea of adjusting the type of `self` on inheritance was pursued by the functional language community using extensible records [79] and developed to create an object-oriented extension [80], which was implemented successfully in the O'Caml [73] language, in a manner suitable for industrial use. Objects are encoded as extensible records, and polymorphic row variables capture, in a precise manner, the concept of protocol extension that matching provides. The type of `self` is described as a fixpoint recursive extensible record with a polymorphic row variable, and the type `self` is adjusted on inheritance by instantiation of that polymorphic row variable, with the new information provided by the subclass, and a new polymorphic row variable, to allow further refinement by inheritance. Intuitively the type of `self` is *open*, for adjustment, due to the polymorphic row variable. Fixpoint recursive extensible records with polymorphic row variables allow objects to be compared with respect to the implemented methods, without the restriction of subtyping. Although the type of `self` has a polymorphic row variable, the objects produced by the class do not, inheritance is not possible once an object is created, which provides a `closed` object type, obtained by instantiation of the polymorphic row variable, from the open `self`, to the absent row, as well as precise adjustment of fixpoints.

4.7 MyType versus Rows

Rows and MyType appear somehow equivalent, as they both express matching.

MyType is a ground type, interpreted by type judgement rules, so MyType is the same for all classes, however the type rules judge MyType differently depending on the class MyType occurs in. Note that MyType is, by type rule judgements only, somehow equivalent to a fixpoint for the class. This is in contrast to rows, which provides a precise type for `self`.

Since MyType is a ground type, it is not possible to differentiate between between MyType from different classes, therefore matching with MyType can not express virtual classes and mutual parametrisation, which need to distinguish between MyType from different classes. To solve this problem, Bruce and Vanderwaart introduce a notion of groups of mutually recursive types, where each MyType in the group is given a distinct ground type [18]. This is, again, in contrast to rows, which by definition, due to a precise type of `self`, can distinguish between the `self` of different classes, and therefore naturally express

virtual types [81].

The following quote from Rémy and Vouillon [80, Sec. 11] illustrates the importance of matching, and the close connection to row variables.

> "Open record types are connected to the notion of matching introduced by Kim Bruce. Matching seems to be at least as important as subtyping in object-oriented languages. Row variables in object types express matching in a very natural way."

However, to further decide if rows and MyType have equivalent expressive power requires a more thorough analysis. Bono and Bugliesi [13] pursues a comparison between rows, although with a slightly different encoding [31], that express negative information, through the absence of methods, and MyType, and state that rows seem more expressive than MyType, as illustrated by the following quote:

> "The result does not generalize to arbitrary expressions and judgements, as we have shown giving examples of typing judgements derivable in [31] that cannot be meaningfully encoded into our system. In fact, although we have not proved it, we are convinced that the original system is strictly more expressive than the one we have presented, if we consider arbitrary judgements. The fundamental reason for this is that the kinding system of [31] is strictly more informative than ours, as it conveys information on method absence, a form of negative information that cannot be accounted for with matching."

An even more formal analysis was later done by Bono [12] which supports the first findings.

We have established that it is desireable for the Creol language to separate inheritance and subtyping, which is realised by matching, regardless of the choice between MyType and rows. However, the Creol language requires the Creol type system to combine objects with some variation of a typed λ-calculus, due to the presence of functional programming. Objects encoded by rows are practical to combine with a slightly enriched λ-calculus, because rows are expressed directly, with a precise handling of fixpoints. Objects encoded with MyType are impractical to combine with λ-calculus, even system-F. The practicality is here determined by the gap between abstractions and formal treatment of fixpoints, so there is a large gap between the MyType abstraction and the λ-calculus with fixpoints, while there is a narrow gap between objects as rows and the λ-calculus with fixpoints. Rows do however not express nominal restrictions, which is a weakness we have addressed in the Creol type system.

Intuitively, matching with MyType appear simpler, at first glance, however, rows, with nominal restrictions appear as a much more precise.

Chapter 5

Approaching Creol Typing

This section provides insight into type checking of Creol, by investigation of examples and principles from the object-oriented subset. The goal is to make assumptions, concerning the Creol type system and the Creol language, explicit.

For the sake of simplicity, let us forget about subtyping, matching, MyType, type equality for rows, fixpoints, parametrisation, and so on, and start from scratch.

We wish to type the object-oriented part of the Creol language with as few concepts as possible. The object-oriented subset of the Creol language is centered around classes and interfaces, and how these are related. Classes and interfaces are abstractions for objects. Conformance is a relation that expresses compatibility between objects.

We introduce an object type $\mathcal{O}\{\rho\}$, where ρ is a row that describes the object type. Then we introduce a conformance relation $\ddot{<}$ that can decide compatibility between objects, by looking at rows, so $\mathcal{O}\{\rho^b\} \ddot{<} \mathcal{O}\{\rho^a\}$ is true when $\mathcal{O}\{\rho^b\}$ is compatible with, or conforms to, $\mathcal{O}\{\rho^a\}$, which is determined by comparing rows.

Although classes and interfaces are abstractions at a higher level than object types, they do not facilitate a uniform treatment of type checking issues. Armed with object types $\mathcal{O}\{\rho\}$ and a conformance relation $\ddot{<}$, the Creol type system can express classes and interfaces in a precise and uniform manner, by introducing various rows for Creol language concepts, define conformance for these rows, as well as connect interfaces and classes to objects types, all of which is done throughout this section.

The section may appear as a series of pieces from a puzzle, and this is to be expected, however at the end of the section, the pieces should fall together and form a consistent picture.

5.1 Classes and Interfaces

This section document the connection between the higher level abstractions, class and interface, and the lower level abstraction, object type. Depending on the perspective, a class or interface reveals different information about the objects it describes. Object types reveal, or hide, information about objects, so for each perspective, there is a corresponding object type, and a class or

interface is described by the set of object types corresponding to all possible perspectives. These object types are now presented, although the use is not demonstrated until examples later in the section.

The type of an object, as seen by the instance itself, the type of `self`. The variable `this` has type `self` which exposes instance variables and personal methods. Since inheritance can extend a class, the type of `self` must be open, that is, contain a free row variable ρ. In the type system presentation the type of `self` for an unspecified class is written τ^{self} and the type of self for a class `A` is written $\tau^{\text{A self}}$.

The type of `Self` is the type of other objects from the same class, as seen by the instance. `Self` is written τ^{Self} in the type system presentation. The type τ^{Self} is obtained from τ^{self} by hiding information which is private for each instance, such as instance variables and personal methods. Since inheritance also extends the other objects of the same class the type τ^{Self} must be open, that is, contain a free row variable ρ.

The type of objects of a class, as seen from outside the class, is written τ^{closed}. The object type τ^{closed} is obtained from τ^{Self} by removing information which is private to the class, and by removing the free row variable ρ, because the type of objects produced by this class is the same regardless of later inheritance.

The type of objects from this class when the object type is considered as protocol description, written τ^{open}. The type τ^{open} is obtained from τ^{closed} by adding a free row variable ρ.

An interface is described by the different types of objects that correspond to the interface, similarly to a class. Since an interface does not have any variables, there is no code that requires the perspectives expressed by the types τ^{self} and τ^{Self} found in classes. However, inheritance requires the perspective expressed by τ^{self}. All the methods in τ^{self} are virtual, as no implementation is provided. The methods in τ^{closed} and τ^{open} are not virtual.

5.2 Instance Privacy

This section shows how instance privacy is expressed in the type system. Instance privacy is the phenomena where information in an object instance is hidden from all other object instances, even from the same class, and the Creol language has instance privacy [49]. Note that instance privacy does not correspond to the visibility declaration `private` from Java, which express class privacy. Class privacy implies that information is hidden from object instances from other classes, however, every object instance of a class has access to `private` information inside other object instances of the same class.

Let `this` be an object instance from a class, and let `that` be any other object instance from the same class. Instance privacy, for `this`, means that some information inside `this` is hidden from `that`. Conversely, instance privacy, for `that`, means that some information inside `that` is hidden from `this`. Although all object of a class have the same information, the type of an object reveals, or hides, that information. When inside `this`, we need to provide different types for `this` and `that` in order to have instance privacy. Let `self` be the type of an object instance, as perceived by the object instance itself, hence `this` has type `self`. Let `Self` be the type of any other object instance from the same class, hence, the object instance `this` perceives `that` to have type `Self`. Observe that

the type of `Self` is derived from the type of `self`, by removal of information that is private to the instance, so `self` and `Self` are different types.

We write τ^{self} the type of `self`, and τ^{Self} for the type of `Self`, such that inside a class, the current object instance, `this`, has type τ^{self}, while all other object instances from the same class, `This`, has type τ^{Self}. Observe that τ^{Self} is different from τ^{closed} and τ^{open}, because inheritance adjusts both τ^{self} and τ^{Self}, according to the new information available in the subclass, and in a manner that expresses instance privacy.

By refining the type language and distinguishing between the type of `self`, from the type of `Self`, the Creol type language facilitates instance privacy.

5.3 Namespaces

This section demonstrates how multiple namespaces for identifiers are required.

Intuitively, the type of an identifier depends on the perspective. This is demonstrated with inheritance and initialisation parameters.

```
class A(i:Int)
begin
  var b:Bool
end

class B inherits A(42)
begin
  var s:Str
end

... new A(42); ...
... var o:A; ...
```

The class `A` consists of an initialization parameter `i` and an instance variable `b`. The class `B` inherits from `A` and declares an instance variable `s`. The variable declaration `var o:A` and the code `new A(42)` are taken out of some unknown context.

The object types that describe classes `A` and `B` are in the type environment

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^{\text{A self}} : \ldots \\ \dot{\tau}^{\text{A Self}} : \ldots \\ \dot{\tau}^{\text{A closed}} : \ldots \\ \dot{\tau}^{\text{A open}} : \ldots \\ \dot{\tau}^{\text{A new}} : \ldots \\ \dot{\tau}^{\text{B self}} : \ldots \\ \dot{\tau}^{\text{B Self}} : \ldots \\ \dot{\tau}^{\text{B closed}} : \ldots \\ \dot{\tau}^{\text{B open}} : \ldots \\ \dot{\tau}^{\text{B new}} : \ldots \end{array} \right\}$$

where the details are hidden as they are not important for this example. The creation of a type environment is demonstrated in Section 5.4.

The type of the identifier `A` is different, depending on the use. The different interpretations of `A` are:

`var o:A` The identifier `A` is the closed object type described by class `A`. Hence $\texttt{A} : \dot{\tau}^{\text{A closed}}$.

`new A(42)` The identifier `A` is a function from initialisation parameters to the type of objects produced by class `A`, which are described by the closed object type of class `A`. Therefore $\texttt{A} : \mathit{Int} \to \dot{\tau}^{\text{A closed}}$

`inherit A(42)` The identifier `A` is a function from initialisation parameters to the type of `self` for class `A`. Therefore $\texttt{A} : \mathit{Int} \to \dot{\tau}^{\text{A self}}$. This follows naturally, because inheritance expands the `self` from the super class, and initialisation parameters must be provided.

This example demonstrates that the identifier `A` has a different type, depending on whether it is used for inheritance, object creation or variable declaration. The solution to this is the introduction of a separate namespace for each use. These namespaces are

$$\Gamma \stackrel{\text{def}}{=} \left\{ \ \texttt{A} : \dot{\tau}^{\text{A closed}} \ \right\}$$

$$\Gamma^{\text{new}} \stackrel{\text{def}}{=} \left\{ \texttt{A} : \mathit{Int} \to \dot{\tau}^{\text{A closed}} \right\}$$

$$\Gamma^{\text{inherit}} \stackrel{\text{def}}{=} \left\{ \texttt{A} : \mathit{Int} \to \dot{\tau}^{\text{A self}} \right\}$$

which provides the correct type for each use in a separate namespace.

5.4 Classes and Object Types

This section demonstrates by example the correspondence between a class and the object types that describe that class, including the types for `self` and `Self` with respect to inheritance.

A class is described by several object types τ^{self}, τ^{Self}, τ^{closed}, τ^{open} and τ^{new}. Of these object types, the type τ^{self} reveals all available information about objects from that class, and the other object types are constructed from τ^{self} by transformations.

To provide some substance, let us sketch the object types for the sample class `A`.

```
class A
begin
  var x:Int
  with Any op eq(in o:This) == ...
  op f(in o:A) == ...
  with This op g(out o:#A) == ...
end
```

The `This` in `eq(in o:This)` refers to τ^{Self} for class `A`, written $\tau^{\text{A Self}}$. The `A` in `f(in o:A)` refers to τ^{closed} for class `A`, written $\tau^{\text{A closed}}$. The `#A` in `g(out o:#A)` refers to τ^{open} for class `A`, written $\tau^{\text{A open}}$. The cointerface restriction `Any` refers to the any type $\top$. A class method without a declared cointerface, is given the cointerface `Any`, and is given a personal row, instead of a method row. The personal row is only visible to the object instance itself, and not to other object instances from the same class. Note that the cointerface for a personal

row could be either $\top$ or τ^{self}, it does not matter which, it is the row, not the cointerface, that denotes visibility, so we use $\top$ for simplicity.

The type $\tau^{\text{A self}}$ consists of all available information and becomes

$$\tau^{\text{A self}} \stackrel{\text{def}}{=} \mathcal{O}\left\{\begin{array}{l}\mathsf{variable\ x} : \mathit{Int}\\ \mathsf{method\ eq} : \mathcal{M}(\top, \tau^{\text{A Self}}, ())\\ \mathsf{personal\ f} : \mathcal{M}(\top, \tau^{\text{A closed}}, ())\\ \mathsf{method\ g} : \mathcal{M}(\tau^{\text{A Self}}, \tau^{\text{A open}}, ())\\ \rho^{\text{A self}}\end{array}\right\}$$

where the empty type () is used, when there are no output parameters. Note that $\rho^{\text{A self}}$ is a placeholder for an unknown number of rows. Intuitively this means that more information can be added at a later time by replacing $\rho^{\text{A self}}$. Notice that $\tau^{\text{A self}}$ is defined with $\tau^{\text{A Self}}$, $\tau^{\text{A closed}}$ and $\tau^{\text{A open}}$, and these are defined by $\tau^{\text{A self}}$, so there is mutual recursion between $\tau^{\text{A self}}$ and τ^{Self}, τ^{closed} and τ^{open}. It is generally not possible to write out mutually recursive types, because they are infinite, without fixpoints to break the infinite recursion, which is demonstrated later, along with a formal fixpoint interpretation, in Section 5.14. In this example the types $\tau^{\text{A Self}}$, $\tau^{\text{A closed}}$ and $\tau^{\text{A open}}$ are infinite types. The introduction of a type environment Θ with placeholder types and an implicit fixpoint interpretation, allows placeholder types to be used prior to their definition. Intuitively, $\dot{\tau}$ is a placeholder for a type τ, and Θ can translate from a placeholder to a type. Given a placeholder type $\dot{\tau}$ and a $\Theta \stackrel{\text{def}}{=} \{(\dot{\tau}, \tau)\}$, the type τ can be obtained from $\dot{\tau}$ by doing $\Theta(\dot{\tau})$, which produces τ, therefore, in the presence of Θ, placeholder types can be used freely as types. For the example, the type environment becomes

$$\Theta \stackrel{\text{def}}{=} \left\{\begin{array}{l}\dot{\tau}^{\text{A self}} : \mathcal{O}\left\{\begin{array}{l}\mathsf{variable\ x} : \mathit{Int}\\ \mathsf{method\ eq} : \mathcal{M}(\top, \dot{\tau}^{\text{A Self}}, ())\\ \mathsf{personal\ f} : \mathcal{M}(\top, \dot{\tau}^{\text{A closed}}, ())\\ \mathsf{method\ g} : \mathcal{M}(\dot{\tau}^{\text{A Self}}, \dot{\tau}^{\text{A open}}, ())\\ \rho^{\text{A self}}\end{array}\right\}\\ \dot{\tau}^{\text{A Self}} : \ldots\\ \dot{\tau}^{\text{A closed}} : \ldots\\ \dot{\tau}^{\text{A open}} : \ldots\end{array}\right\}$$

where placeholder types $\dot{\tau}^{\text{A Self}}$, $\dot{\tau}^{\text{A closed}}$ and $\dot{\tau}^{\text{A open}}$ are left unspecified. Given this partial Θ, it is straightforward to derive the unspecified types. The type $\dot{\tau}^{\text{A Self}}$ is derived from $\dot{\tau}^{\text{A self}}$ by removal of information that is private to the instance, such as instance variables and personal methods. This produces an updated type environment

$$\Theta \stackrel{\text{def}}{=} \left\{\begin{array}{l}\vdots\\ \dot{\tau}^{\text{A Self}} : \mathcal{O}\left\{\begin{array}{l}\mathsf{method\ eq} : \mathcal{M}(\top, \dot{\tau}^{\text{A Self}}, ())\\ \mathsf{method\ g} : \mathcal{M}(\dot{\tau}^{\text{A Self}}, \dot{\tau}^{\text{A open}}, ())\\ \rho^{\text{A Self}}\end{array}\right\}\\ \vdots\end{array}\right\}$$

where the placeholder types $\dot{\tau}^{\text{A closed}}$ and $\dot{\tau}^{\text{A open}}$ are left unspecified. Note that $\rho^{\text{A self}}$ is different from $\rho^{\text{A Self}}$, because different rows may replace $\rho^{\text{A self}}$ and

$\rho^{\text{A Self}}$. Intuitively an extension of `self` with instance variables does not extend `Self`.

The type $\dot{\tau}^{\text{A closed}}$ is the type of objects produced from this class, as seen from outside the class. The types $\dot{\tau}^{\text{A self}}$ and $\dot{\tau}^{\text{A Self}}$ are not visible outside the class. The type $\dot{\tau}^{\text{A closed}}$ is derived from $\dot{\tau}^{\text{A self}}$ by removing information that is hidden inside the class, which includes replacing the fixpoints $\dot{\tau}^{\text{A self}}$ and $\dot{\tau}^{\text{A Self}}$ by $\dot{\tau}^{\text{A closed}}$. This hides the information previously exposed by $\dot{\tau}^{\text{A self}}$ and $\dot{\tau}^{\text{A Self}}$. Once an object is produced, it is not possible to add more information, therefore there are no open rows in $\dot{\tau}^{\text{A closed}}$. This produces the type environment

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \vdots \\ \dot{\tau}^{\text{A closed}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{method\ eq} : \mathcal{M}(\top, \dot{\tau}^{\text{A closed}}, ()) \\ \mathsf{method\ g} : \mathcal{M}(\dot{\tau}^{\text{A closed}}, \dot{\tau}^{\text{A open}}, ()) \end{array} \right\} \\ \vdots \end{array} \right\}$$

where the type $\dot{\tau}^{\text{A open}}$ is left unspecified. The type $\dot{\tau}^{\text{A open}}$ is the protocol for objects produced by the class. The separation of inheritance from subtyping, discussed in Section 4, introduces open and closed object types, where, intuitively, an object type describes an object protocol, and a closed object type describes an object. The type $\dot{\tau}^{\text{A open}}$ is derived from $\dot{\tau}^{\text{A closed}}$ by the addition of $\rho^{\text{A open}}$, which opens the type. This produces the final type environment

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^{\text{A self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{variable\ x} : \mathit{Int} \\ \mathsf{method\ eq} : \mathcal{M}(\top, \dot{\tau}^{\text{A Self}}, ()) \\ \mathsf{personal\ f} : \mathcal{M}(\top, \dot{\tau}^{\text{A closed}}, ()) \\ \mathsf{method\ g} : \mathcal{M}(\dot{\tau}^{\text{A Self}}, \dot{\tau}^{\text{A open}}, ()) \\ \rho^{\text{A self}} \end{array} \right\} \\ \dot{\tau}^{\text{A Self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{method\ eq} : \mathcal{M}(\top, \dot{\tau}^{\text{A Self}}, ()) \\ \mathsf{method\ g} : \mathcal{M}(\dot{\tau}^{\text{A Self}}, \dot{\tau}^{\text{A open}}, ()) \\ \rho^{\text{A Self}} \end{array} \right\} \\ \dot{\tau}^{\text{A closed}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{method\ eq} : \mathcal{M}(\top, \dot{\tau}^{\text{A closed}}, ()) \\ \mathsf{method\ g} : \mathcal{M}(\dot{\tau}^{\text{A closed}}, \dot{\tau}^{\text{A open}}, ()) \end{array} \right\} \\ \dot{\tau}^{\text{A open}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{method\ eq} : \mathcal{M}(\top, \dot{\tau}^{\text{A closed}}, ()) \\ \mathsf{method\ g} : \mathcal{M}(\dot{\tau}^{\text{A closed}}, \dot{\tau}^{\text{A open}}, ()) \\ \rho^{\text{A open}} \end{array} \right\} \end{array} \right\}$$

where a definition is provided for all placeholder types.

Let us expand the example with a subclass `B` which extends `A`.

```
class B inherits A
begin
  var y:Bool
  with Any op eq(in o:This) == ...
  with Any op h(in o:B, out p:#B) == ...
end
```

The class `B` adds another instance variable, refines the definition of `eq`, and introduces a new method `h`. The inheritance from class `A` is captured by a row for inheritance from $\dot{\tau}^{\text{A } self}$. Inheritance is always from the type of `self` for the super class, because all information from the super class is available to the

subclass. The type environment Θ is extended with the placeholder types for class `B`. The type environment becomes

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^{\text{A self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{variable}\ \mathtt{x} : \mathit{Int} \\ \mathsf{method}\ \mathtt{eq} : \mathcal{M}(\top, \dot{\tau}^{\text{A Self}}, ()) \\ \mathsf{personal}\ \mathtt{f} : \mathcal{M}(\top, \dot{\tau}^{\text{A closed}}, ()) \\ \mathsf{method}\ \mathtt{g} : \mathcal{M}(\dot{\tau}^{\text{A Self}}, \dot{\tau}^{\text{A open}}, ()) \\ \rho^{\text{A self}} \end{array} \right\} \\ \vdots \\ \dot{\tau}^{\text{B self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{inherit}\ \mathtt{A} : \dot{\tau}^{\text{A self}} \\ \mathsf{variable}\ \mathtt{y} : \mathit{Bool} \\ \mathsf{method}\ \mathtt{eq} : \mathcal{M}(\top, \dot{\tau}^{\text{B Self}}, ()) \\ \mathsf{method}\ \mathtt{h} : \mathcal{M}(\top, \dot{\tau}^{\text{B closed}}, \dot{\tau}^{\text{B open}}) \\ \rho^{\text{B self}} \end{array} \right\} \\ \vdots \end{array} \right\}$$

where the type environment made for the class `A` is left out, and $\dot{\tau}^{\text{B Self}}$, $\dot{\tau}^{\text{B closed}}$ and $\dot{\tau}^{\text{B open}}$ are left unspecified, as they can only be created once the row $\mathsf{inherit}\ \mathtt{A} : \dot{\tau}^{\text{A self}}$ has been expanded. The expansion of inheritance must:

- Replace the types of `self` and `Self` with updated versions. This change of fixpoint is discussed later in Section 5.17.2.
- Copy rows from the super type to the subtype.
- Ensure conformance for those rows that are both in the super type and subtype.
- Add a super row for static binding.

Those rows that can be copied from the `self` of class `A` are

$$\begin{array}{l} \mathsf{variable}\ \mathtt{x} : \mathit{Int} \\ \mathsf{personal}\ \mathtt{f} : \mathcal{M}(\top, \dot{\tau}^{\text{A closed}}, ()) \\ \mathsf{method}\ \mathtt{g} : \mathcal{M}(\dot{\tau}^{\text{A Self}}, \dot{\tau}^{\text{A open}}, ())_{\dot{\tau}^{\text{A Self}} \mapsto \dot{\tau}^{\text{B Self}}} \end{array}$$

where $\dot{\tau}^{\text{A Self}} \mapsto \dot{\tau}^{\text{B Self}}$ means, that the fixpoint $\dot{\tau}^{\text{A Self}}$ is replaced by $\dot{\tau}^{\text{B Self}}$, because the type of `Self` is updated on inheritance. Conformance must be checked for the `eq` rows.

$$\mathsf{method}\ \mathtt{eq} : \mathcal{M}(\top, \dot{\tau}^{\text{B Self}}, ()) \ddot{<} \left(\mathsf{method}\ \mathtt{eq} : \mathcal{M}(\top, \dot{\tau}^{\text{A Self}}, ())\right)_{\dot{\tau}^{\text{A Self}} \mapsto \dot{\tau}^{\text{B Self}}}$$

This reduces to

$$\mathsf{method}\ \mathtt{eq} : \mathcal{M}(\top, \dot{\tau}^{\text{B Self}}, ()) \ddot{<} \mathsf{method}\ \mathtt{eq} : \mathcal{M}(\top, \dot{\tau}^{\text{B Self}}, ())$$

which holds trivially because

$$\dot{\tau}^{\text{B Self}} \ddot{>} \dot{\tau}^{\text{B Self}}$$

where contravariance for input parameters changes the direction of conformance. Conformance must also hold for row variables, hence

$$\rho^{\text{B self}} \ddot{<} \rho^{\text{A self}}$$

which holds trivially, because both are open. With the addition of a super row, the updated type environment becomes

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \vdots \\ \dot{\tau}^{\text{B self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{super}\ \mathtt{A} : \dot{\tau}^{\text{A self}} \\ \mathsf{variable}\ \mathtt{x} : \mathit{Int} \\ \mathsf{personal}\ \mathtt{f} : \mathcal{M}(\top, \dot{\tau}^{\text{A closed}}, ()) \\ \mathsf{method}\ \mathtt{g} : \mathcal{M}(\dot{\tau}^{\text{B Self}}, \dot{\tau}^{\text{A open}}, ()) \\ \mathsf{variable}\ \mathtt{y} : \mathit{Bool} \\ \mathsf{method}\ \mathtt{eq} : \mathcal{M}(\top, \dot{\tau}^{\text{B Self}}, ()) \\ \mathsf{method}\ \mathtt{h} : \mathcal{M}(\top, \dot{\tau}^{\text{B closed}}, \dot{\tau}^{\text{B open}}) \\ \rho^{\text{B self}} \end{array} \right\} \\ \vdots \end{array} \right\}$$

where the type $\dot{\tau}^{\text{B self}}$ is fully described. The types $\dot{\tau}^{\text{B Self}}$, $\dot{\tau}^{\text{B closed}}$ and $\dot{\tau}^{\text{B open}}$ can now be derived, in the same manner as for class `A`. The super row is removed when $\dot{\tau}^{\text{B Self}}$ is derived from $\dot{\tau}^{\text{B self}}$. Static binding is only meaningful for the instance itself, but not for other objects from the same class. The type environment with types for class `B` now becomes

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \vdots \\ \dot{\tau}^{\text{B self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{super}\ \mathtt{A} : \dot{\tau}^{\text{A self}} \\ \mathsf{variable}\ \mathtt{x} : \mathit{Int} \\ \mathsf{personal}\ \mathtt{f} : \mathcal{M}(\top, \dot{\tau}^{\text{A closed}}, ()) \\ \mathsf{method}\ \mathtt{g} : \mathcal{M}(\dot{\tau}^{\text{B Self}}, \dot{\tau}^{\text{A open}}, ()) \\ \mathsf{variable}\ \mathtt{y} : \mathit{Bool} \\ \mathsf{method}\ \mathtt{eq} : \mathcal{M}(\top, \dot{\tau}^{\text{B Self}}, ()) \\ \mathsf{method}\ \mathtt{h} : \mathcal{M}(\top, \dot{\tau}^{\text{B closed}}, \dot{\tau}^{\text{B open}}) \\ \rho^{\text{B self}} \end{array} \right\} \\ \dot{\tau}^{\text{B Self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{method}\ \mathtt{g} : \mathcal{M}(\dot{\tau}^{\text{B Self}}, \dot{\tau}^{\text{A open}}, ()) \\ \mathsf{method}\ \mathtt{eq} : \mathcal{M}(\top, \dot{\tau}^{\text{B Self}}, ()) \\ \mathsf{method}\ \mathtt{h} : \mathcal{M}(\top, \dot{\tau}^{\text{B closed}}, \dot{\tau}^{\text{B open}}) \\ \rho^{\text{B Self}} \end{array} \right\} \\ \dot{\tau}^{\text{B closed}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{method}\ \mathtt{g} : \mathcal{M}(\dot{\tau}^{\text{B closed}}, \dot{\tau}^{\text{A open}}, ()) \\ \mathsf{method}\ \mathtt{eq} : \mathcal{M}(\top, \dot{\tau}^{\text{B closed}}, ()) \\ \mathsf{method}\ \mathtt{h} : \mathcal{M}(\top, \dot{\tau}^{\text{B closed}}, \dot{\tau}^{\text{B open}}) \end{array} \right\} \\ \dot{\tau}^{\text{B open}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{method}\ \mathtt{g} : \mathcal{M}(\dot{\tau}^{\text{B closed}}, \dot{\tau}^{\text{A open}}, ()) \\ \mathsf{method}\ \mathtt{eq} : \mathcal{M}(\top, \dot{\tau}^{\text{B closed}}, ()) \\ \mathsf{method}\ \mathtt{h} : \mathcal{M}(\top, \dot{\tau}^{\text{B closed}}, \dot{\tau}^{\text{B open}}) \\ \rho^{\text{B open}} \end{array} \right\} \end{array} \right\}$$

This example also demonstrates that inheritance is separated from subtyping. Intuitively, this means, that the type of `self` from class `B` matches the type of `self` from class `A`, however objects from class `B` are not subtypes of objects from class `A`. Conformance is satisfied for $\dot{\tau}^{\text{B self}} \ddot{<} \dot{\tau}^{\text{A self}}$ which guarantees that inheritance is type safe, however, we can not show this yet, the exact definition of conformance with respect to open object types and fixpoints is deferred to Section 5.17.2. Conformance does however not hold for $\dot{\tau}^{\text{B closed}} \ddot{<} \dot{\tau}^{\text{A closed}}$, however, we can not show this yet, the exact definition of conformance is de-

ferred to Section 5.17.1, and subsumption, for closed object types, is deferred to Section 7.1.5.

5.5 Information Flow and Conformance

This section demonstrates how information flow influences conformance for types. Although variance restrictions are well known from the literature [77, Sect. 15.2], it is useful to provide some intuition on this topic. A rule with explanation is provided for functions, references and object types.

Intuitively, variance helps to decide when it is safe to override something. Consider a class `A` with a method `m` and a subclass `B` that inherits from `A` and overrides the method `m` with a better version. The cointerface, the in parameter and the out parameter are replaced with types to make reasoning simpler.

```
class A
begin
  with τ^(m^A co) op m(in τ^(m^A in) out τ^(m^A out)) == ...
end

class B inherit A
begin
  //replace foo from A with something better
  with τ^(m^B co) op m(in τ^(m^B in) out τ^(m^B out)) == ...
end
```

The inheritance is safe if the override of method `m` is safe. This can be decided by looking at variance for the types that describe class `A` and class `B`, including those in the two `m` methods. Variance is guided by principles, so we must investigate these principles.

5.5.1 Information Flow, Variance and Conformance

Variance describes the kind of differences that are allowed between types, such that they still can be considered compatible. Compatibility is captured by the conformance relation $\ddot{<}$ and variance describes how the direction of the variance changes between $\ddot{<}$ and $\ddot{>}$. Variance is decided by looking at the flow of information. To reason about variance, we adopt the terms *source* and *sink* [77, Page 199]. A source provides information and a sink receives information.

- Information that flows into a sink requires contravariant $\ddot{>}$ conformance.
- Information that flows out from a source requires covariant $\ddot{<}$ conformance.
- Information flows both into τ and out from τ, when τ is both a sink and a source, and requires both covariant and contravariant conformance at the same time, which is called invariance. Invariance reduces to type equality, denoted $\ddot{=}$.

These restrictions occur naturally for all types, not just for method signatures, instance variables and object types, but also for functions, lists and so on.

The variance restrictions preserve static type safety. A statically type safe language can, by the addition of variance rules, statically type check a larger set of programs. Variances can be illustrated as follows, contravariance changes the direction of the conformance,

$$\frac{\tau^{\text{sub}} \ddot{>} \tau^{\text{sup}}}{\left(\dots \tau^{\text{sub}} \dots\right) \ddot{<} \left(\dots \tau^{\text{sup}} \dots\right)} \text{ contravariance}$$

while covariance keeps the direction of conformance.

$$\frac{\tau^{\text{sub}} \ddot{<} \tau^{\text{sup}}}{\left(\dots \tau^{\text{sub}} \dots\right) \ddot{<} \left(\dots \tau^{\text{sup}} \dots\right)} \text{ covariance}$$

Given a sink or a source, it is perfectly possible, that a program does not actually write to the sink or read from the source, so the variance restrictions are unnecessary rigid for just that program. Such cases always require careful analysis of the code, quickly moving into an area of complex analysis, where theorem proving may be required. The variance restrictions preserve static type safety by a general assumption, that does not require careful case by case analysis. Analysis to determine fine grained variance restrictions on method overloading is still a research topic. Static type safety, with the general requirement of contravariance and covariance, depending on how information flows, is straightforward.

5.5.2 Function Conformance

Let us consider how variance is analysed in view of a type being a sink, a source or both. Consider a function `f` of type $\tau^{\text{f in}} \rightarrow \tau^{\text{f out}}$ and a function `g` of type $\tau^{\text{g in}} \rightarrow \tau^{\text{g out}}$. Can one replace the `f` with a `g`? To answer this question we need to decide if the following conformance is valid.

$$\left(\tau^{\text{g in}} \rightarrow \tau^{\text{g out}}\right) \ddot{<} \left(\tau^{\text{f in}} \rightarrow \tau^{\text{f out}}\right)$$

Both $\tau^{\text{g in}}$ and $\tau^{\text{f in}}$ are sinks because information flows into the function. Both $\tau^{\text{g out}}$ and $\tau^{\text{f out}}$ are sources because information flows out from the function. The resulting variance requirements for functions are illustrated by the following rule.

$$\frac{\tau^{\text{g in}} \ddot{>} \tau^{\text{f in}} \quad \tau^{\text{g out}} \ddot{<} \tau^{\text{f out}}}{\left(\tau^{\text{g in}} \rightarrow \tau^{\text{g out}}\right) \ddot{<} \left(\tau^{\text{f in}} \rightarrow \tau^{\text{f out}}\right)} \begin{array}{l} \text{sink} \Rightarrow \text{contra} \\ \text{source} \Rightarrow \text{co} \end{array}$$

5.5.3 Reference Conformance

Consider a reference `a` of type *ref* τ^{a}, and a reference `b` of type *ref* τ^{b}. Can one replace an `a` with a `b`? Which is possible when the following conformance is valid.

$$\left(\mathit{ref}\ \tau^{\text{b}}\right) \ddot{<} \left(\mathit{ref}\ \tau^{\text{a}}\right)$$

A reference may be both assigned to and read from, therefore both τ^{b} and τ^{a} are both a sink and a source. The resulting variance requirement is

$$\frac{\tau^{b} \ddot{>} \tau^{a} \quad \tau^{b} \ddot{<} \tau^{a}}{\left(\mathit{ref}\ \tau^{b}\right) \ddot{<} \left(\mathit{ref}\ \tau^{a}\right)}$$

which is the same as

$$\frac{\tau^b \ddot{=} \tau^a}{\left(ref\ \tau^b\right) \ddot{<} \left(ref\ \tau^a\right)}$$

In general, it is not statically type safe to refine the type of a reference.

5.5.4 Method Override and Conformance

Consider the example classes `A` and `B` again. The flow of information is decided by how the code is used. A method is used by a method invocation $\mathtt{m}(\tau^{\text{in}};\tau^{\text{out}})$, where the actual input and output parameters are replaced by their types, τ^{in} and τ^{out}. The input parameter of type τ^{in} flows from the point of the invocation, and into the code of the method body. The output parameter of type τ^{out} flows from the code of the method body, and back to where the method invocation was performed.

Now, let us focus on, what type safety requires from inside the method body of method `m` in class `B`. Let $\mathtt{m}^B$ be the method `m` in class `B`, and $\mathtt{m}^A$ be the method `m` in class `A`. It is known that any invocation with input parameter τ^{in} must be compatible with the input parameter $\tau^{\mathrm{m^A\ in}}$, that is $\tau^{\text{in}} \ddot{<} \tau^{\mathrm{m^A\ in}}$. Nothing can go wrong typewise, if the code in $\mathtt{m}^B$ at least is prepared for an argument of type $\tau^{\mathrm{m^A\ in}}$, that is $\tau^{\text{in}} \ddot{<} \tau^{\mathrm{m^A\ in}} \ddot{<} \tau^{\mathrm{m^B\ in}}$. Therefore, method override must respect the requirement $\tau^{\mathrm{m^A\ in}} \ddot{<} \tau^{\mathrm{m^B\ in}}$, which also follows from the flow of information, the input paramater is a sink, therefore contravariance is used.

Now let us look at the output parameter. Let us assume, that the invoked method is $\mathtt{m}^A$, then the information sent back must fit, that is $\tau^{\mathrm{m^A\ out}} \ddot{<} \tau^{\text{out}}$. Now, consider, that we do not know if the invoked method is $\mathtt{m}^A$ or $\mathtt{m}^B$. All is well, as long as the returned parameters from $\mathtt{m}^B$ are such, that the assumption $\tau^{\mathrm{m^A\ out}} \ddot{<} \tau^{\text{out}}$ holds, which is the case when $\tau^{\mathrm{m^B\ out}} \ddot{<} \tau^{\mathrm{m^A\ out}} \ddot{<} \tau^{\text{out}}$. Therefore, method override must respect the requirement $\tau^{\mathrm{m^B\ out}} \ddot{<} \tau^{\mathrm{m^A\ out}}$, which follows from the flow of information, the output paramater is a source, therefore covariance is used.

The cointerface follows the same reasoning as input parameters, and therefore the restriction

$$\tau^{m^{\mathrm{B}}\ \mathrm{co}} \ddot{>} \tau^{\mathrm{m^A\ co}}$$

must be respected, which is the same restriction that follows from the flow of information, a cointerface is a sink, therefore contravariance is used.

Therefore method override is type safe when the following rule holds.

$$\frac{\tau^{\mathrm{m^B\ co}} \ddot{>} \tau^{\mathrm{m^A\ co}} \quad \tau^{\mathrm{m^B\ in}} \ddot{>} \tau^{\mathrm{m^A\ in}} \quad \tau^{\mathrm{m^B\ out}} \ddot{<} \tau^{\mathrm{m^A\ out}}}{\mathcal{M}\left(\tau^{\mathrm{m^B\ co}}, \tau^{\mathrm{m^B\ in}}, \tau^{\mathrm{m^B\ out}}\right) \ddot{<} \mathcal{M}\left(\tau^{\mathrm{m^A\ co}}, \tau^{\mathrm{m^A\ in}}, \tau^{\mathrm{m^A\ out}}\right)}$$

Note that this rule follows directly from type safety for method override, therefore both virtual and static binding, which may return another method body than expected, is type safe.

However if method override is replaced with method overloading, the picture changes. The definition of the Creol language, as defined in this thesis, has method override, but not method overloading, which is mentioned as a possible future enhancement in Section 8.7.2.

5.5.5 From Object to Method Conformance

Let us consider how conformance between objects reduces to conformance between methods, by looking at conformance between objects produced by class `A` and `B`.

Let $\mathcal{O}\left\{\rho^A\right\}$ be the type of an object from class `A`, and $\mathcal{O}\left\{\rho^A\right\}$ be type of an object from class `B`. Now, we want to know if an object of class `B` conforms to an object of class `A`, so when does the following relation hold?

$$\mathcal{O}\left\{\rho^B\right\} \ddot{<} \mathcal{O}\left\{\rho^A\right\}$$

Conformance between object types is determined by the rows that describe the object, which again affects the treatment of fixpoints, which is treated later in Section 5.14. Variance affects those rows in both ρ^A and in ρ^B, that have the same name, and because a row is a source, it provides information, covariance holds for rows. This example is quite simple, as each object type only has one method `m`, which is reflected by a method row. Then the conformance

$$\mathcal{O}\left\{\mathsf{method}\ \mathtt{m} : \mathcal{M}\left(\tau^{m^{\mathrm{B}}\ \mathrm{co}}, \tau^{\mathrm{m^B}\ \mathrm{in}}, \tau^{\mathrm{m^B}\ \mathrm{out}}\right)\right\} \ddot{<} \mathcal{O}\left\{\mathsf{method}\ \mathtt{m} : \mathcal{M}\left(\tau^{\mathrm{m^A}\ \mathrm{co}}, \tau^{\mathrm{m^A}\ \mathrm{in}}, \tau^{\mathrm{m^A}\ \mathrm{out}}\right)\right\}$$

which reduces covariantly to conformance for the method rows with name `m`. So when does

$$\mathcal{M}\left(\tau^{m^{\mathrm{B}}\ \mathrm{co}}, \tau^{\mathrm{m^B}\ \mathrm{in}}, \tau^{\mathrm{m^B}\ \mathrm{out}}\right) \ddot{<} \mathcal{M}\left(\tau^{\mathrm{m^A}\ \mathrm{co}}, \tau^{\mathrm{m^A}\ \mathrm{in}}, \tau^{\mathrm{m^A}\ \mathrm{out}}\right)$$

hold? Both $\tau^{\mathrm{m^B}\ \mathrm{in}}$ and $\tau^{\mathrm{m^A}\ \mathrm{in}}$ are sinks because information flows in. Both $\tau^{\mathrm{m^B}\ \mathrm{out}}$ and $\tau^{\mathrm{m^A}\ \mathrm{out}}$ are sources because information flows out. The cointerfaces $\tau^{m^{\mathrm{B}}\ \mathrm{co}}$ and $\tau^{\mathrm{m^A}\ \mathrm{co}}$ provide information about the caller to the callee, hence they are both sinks because information flows in. Therefore variance for methods is enforced when the following holds.

$$\frac{\tau^{\mathrm{m^B}\ \mathrm{co}} \ddot{>} \tau^{\mathrm{m^A}\ \mathrm{co}} \quad \tau^{\mathrm{m^B}\ \mathrm{in}} \ddot{>} \tau^{\mathrm{m^A}\ \mathrm{in}} \quad \tau^{\mathrm{m^B}\ \mathrm{out}} \ddot{<} \tau^{\mathrm{m^A}\ \mathrm{out}}}{\mathcal{M}\left(\tau^{\mathrm{m^B}\ \mathrm{co}}, \tau^{\mathrm{m^B}\ \mathrm{in}}, \tau^{\mathrm{m^B}\ \mathrm{out}}\right) \ddot{<} \mathcal{M}\left(\tau^{\mathrm{m^A}\ \mathrm{co}}, \tau^{\mathrm{m^A}\ \mathrm{in}}, \tau^{\mathrm{m^A}\ \mathrm{out}}\right)}$$

Note that this rule is the same as variance requirements for method override, not by design, but by definition, due to variance analysis.

5.5.6 Conformance by Source and Sink

The conformance relation $\ddot{<}$ is now extended to arbitrary types, such that the variance requirements determined by source and sink analysis are enforced, such that a sink respects contravariance, and a source respects covariance. This includes functions, references and method signatures. In general, the variance constraints for conformance between any compound types, are described by the following informal rules.

$$\frac{\ldots \quad \tau^{\mathrm{sub}} \ddot{>} \tau^{\mathrm{sup}} \quad \ldots}{\left(\ldots \tau^{\mathrm{sub}} \ldots\right) \ddot{<} \left(\ldots \tau^{\mathrm{sup}} \ldots\right)} \text{ Sink } \tau \Rightarrow \text{contravariance}$$

$$\frac{\ldots \quad \tau^{\mathrm{sub}} \ddot{<} \tau^{\mathrm{sup}} \quad \ldots}{\left(\ldots \tau^{\mathrm{sub}} \ldots\right) \ddot{<} \left(\ldots \tau^{\mathrm{sup}} \ldots\right)} \text{ Source } \tau \Rightarrow \text{covariance}$$

5.6 Structural and Nominal Type Systems

This section briefly motivates why the Creol type system is a hybrid between a structural type system and a nominal type system.

The notion of structural and nominal type system, is that used by Pierce [77, Sec. 19.3]. As noted by Pierce, the distinctions between nominal and structural type systems are still being researched. Structural type systems tend to have explicit fixpoints for recursive types, and decide the subtype relation with respect to type structure. Nominal type systems tend to have implicit fixpoints through a one to one correspondence between names and types, and require subtyping to be declared explicitly. Explicit subtype declarations introduce a *nominal restriction* on possible structural subtyping. Note, that types can have names in both structural and nominal type systems. Structural type systems can also be extended to distinguish types that are structurally equivalent, by the introduction of *name uniqueness* or *occurrence equivalence of types* [7, Page 350], or `branding` [64]. However, structural type systems do not enforce nominal restrictions. Structural type systems are commonly used for functional languages, which usually are based on the λ-calculus. Object-oriented languages tend to choose nominal type systems. It is generally considered easier, in structural type systems rather than in nominal type systems, to reason about and account for advanced uses of type abstraction, such as parametric polymorphism, abstract data types and functors [77, Page 254]. Parametric polymorphism in an object-oriented setting is demonstrated later in Section 7.2.2. Recall from Section 4.7, that it is difficult to encode a nominal type system, with MyType and matching, into the λ-calculus, while it is feasible to encode a structural type system, with objects as extensible records with polymorphic row variables, in a slightly enriched λ-calculus.

5.6.1 Concerning Creol

The presence of procedures, algebraic datatypes and parametrisation, in Creol, suggests that a structural type system is a good approach. The object-oriented part of Creol can be type checked, while separating inheritance from subtyping, with either a nominal or a structural type system. Previously defined structural type systems do not enforce nominal restrictions, which is unacceptable for Creol, however, by the introduction of nominal rows, as in Section 5.7, a structural type system with rows can enforce nominal constraints.

The use of a type environment for sharing of types reduces redundancy, and can be used as implicit fixpoints, to simplify treatment of recursive and mutually recursive types. Nominal type systems naturally produce a type environment, due to a connection between a name and a type. Structural type systems usually treat fixpoints explicitly. It is desireable for the Creol type system to reduce redundancy and simplify treatment of recursive and mutually recursive types, which is obtained by extending a structural type system with a type environment, with an implicit fixpoint interpretation, as in Section 5.14.

Although the Creol type system is a hybrid type system, with properties of both nominal and structural type systems, the approach taken is to start with a structural type system, and add the required properties by extending the type language of the structural type system, as well as the structural type system itself, hence the whole type system can be viewed as structural with a

few extensions.

5.7 Nominal Conformance Constraints

This section explains how the structural Creol type system is altered to express nominal conformance constraints.

To model nominal conformance constraints, we need a mechanism for expressing a nominal constraint, as well as for propagation of nominal constraints. We enrich object types by introduction of new rows. The row nominal `id` expresses that an object type can satisfy or require the nominal constraint named `id`. Note that `id` is not connected to any type, it is merely derived from the name of an interface. The row implement τ expresses the inheritance of nominal constraints from τ. This is different from the row inherit τ which expresses inheritance of everything, except nominal constraints. The conformance relation $\ddot{<}$ is defined for nominal rows. Two rows nominal $\texttt{id}^B$ and nominal $\texttt{id}^A$ conform when $\texttt{id}^B$ is equal to $\texttt{id}^A$.

Consider the following Creol code, that demonstrates interface inheritance.

```
interface A
begin
end

interface B inherits A
begin
end
```

Consider the simplified type of `self` for each of these objects. The interface `A` does not contain any methods, so the type of `self` is

$$\mathcal{O}\{\text{nominal A}\}$$

which only contains a nominal row. The type of `self` for interface `B` is

$$\mathcal{O}\left\{\begin{array}{l}\text{nominal B}\\ \text{implement } \mathcal{O}\{\text{nominal A}\}\end{array}\right\}$$

where there is one nominal row and one row for inheritance of nominal rows. When implement is expanded the type of self for `B` becomes

$$\mathcal{O}\left\{\begin{array}{l}\text{nominal B}\\ \text{nominal A}\end{array}\right\}$$

which contains two nominal rows. The types of `self` for `A` and `B` are conforming

$$\mathcal{O}\left\{\begin{array}{l}\text{nominal B}\\ \text{nominal A}\end{array}\right\} \ddot{<} \mathcal{O}\{\text{nominal A}\}$$

because we have

$$\text{nominal A} \ddot{<} \text{nominal A}$$

which demonstrates how nominal constraints are be handled.

5.8 Interfaces and Virtual Methods

This section investigates the object types corresponding to interfaces.

The type of `self` for any interface can be modelled by object types through the introduction of virtual methods. A method, that does not have a body, is referred to as virtual [81] or abstract [19, 36]. We adopt the term virtual for a method without implementation, and avoid the term abstract, which has a different meaning in type theory. The addition of virtual methods to the Creol type system, does not affect the Creol language, as virtual methods are introduced provide a uniform treatment of objects and classes in the type system only.

All method rows, in the type of `self` for an interface, are virtual. A virtual method may be overridden by either a virtual method or a regular method. A regular method may not be overridden by a virtual method. Intuitively, it is possible to provide the code for a method, but not remove the code for a method. A class successfully implements an interface, when there are no virtual methods in the type of `self` for the class.

To provide some substance consider a simple interface and the corresponding object types.

```
interface A
begin
  with Any op foo()
end
```

Interfaces require a type of `self` and `Self` for the purpose of type checking inheritance in a manner uniform for interfaces and classes. Note, that for interfaces, the types of `self` and `Self` are the same, since interfaces do not provide instance privacy.

Although the types of `self` and `Self` may have virtual methods, the object types described by the interface may not. Virtual methods are only meaningful for the purpose of type checking inheritance. The closed and open object types are used for describing object instances, and no actual object instance can have a virtual method. An object instance provides, by definition, code for all the supported methods. Recall from Section 5.7, that interfaces also describe nominal constraints. The interface `A` is described by the following type environment.

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^{\text{A self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \mathsf{virtual}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \\ \rho^{\text{A self}} \end{array} \right\} \\ \dot{\tau}^{\text{A Self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \mathsf{virtual}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \\ \rho^{\text{A Self}} \end{array} \right\} \\ \dot{\tau}^{\text{A closed}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \quad \mathsf{method}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \end{array} \right\} \\ \dot{\tau}^{\text{A open}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \mathsf{method}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \\ \rho^{\text{A open}} \end{array} \right\} \end{array} \right\}$$

To demonstrate how interface inheritance operates, the interface `B` extends the interface `A` with an additional method `bar`.

```
interface B inherit A
begin
  with Any op bar()
end
```

The type environment is enriched with `self` for interface `B` becomes

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \vdots \\ \dot{\tau}^{\text{B self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{implement}\ \dot{\tau}^{\text{A self}} \\ \mathsf{inherit}\ \texttt{A} : \dot{\tau}^{\text{A self}} \\ \mathsf{nominal}\ \texttt{B} \\ \mathsf{virtual}\ \texttt{bar} : \mathcal{M}(\top, (), ()) \\ \rho^{\text{B self}} \end{array} \right\} \\ \vdots \end{array} \right\}$$

where the other object types for interface `B` are unspecified. Notice, that since interfaces inherit both nominal rows and regular rows, inheritance is represented by both an implement row and an inherit row. The expansion of inheritance is straightforward in this example. There are no fixpoints to replace, and since $\rho^{\text{B self}} \ddot{<} \rho^{\text{A self}}$, the open row $\rho^{\text{A self}}$ is replaced by $\rho^{\text{B self}}$. The type environment after expansion of inheritance becomes

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \vdots \\ \dot{\tau}^{\text{B self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \mathsf{virtual}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \\ \mathsf{nominal}\ \texttt{B} \\ \mathsf{virtual}\ \texttt{bar} : \mathcal{M}(\top, (), ()) \\ \rho^{\text{B self}} \end{array} \right\} \\ \dot{\tau}^{\text{B Self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \mathsf{virtual}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \\ \mathsf{nominal}\ \texttt{B} \\ \mathsf{virtual}\ \texttt{bar} : \mathcal{M}(\top, (), ()) \\ \rho^{\text{B Self}} \end{array} \right\} \\ \dot{\tau}^{\text{B closed}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \mathsf{method}\ \texttt{foo} : \mathcal{M}(,,) \\ \mathsf{nominal}\ \texttt{B} \\ \mathsf{method}\ \texttt{bar} : \mathcal{M}(,,) \end{array} \right\} \\ \dot{\tau}^{\text{B open}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \mathsf{method}\ \texttt{foo} : \mathcal{M}(,,) \\ \mathsf{nominal}\ \texttt{B} \\ \mathsf{method}\ \texttt{bar} : \mathcal{M}(,,) \\ \rho^{\text{B open}} \end{array} \right\} \end{array} \right\}$$

where the open and closed object types for interface `B` are revealed.

To demonstrate the interaction between virtual methods and regular methods, we let a class `C` implements the interface `B`. The class `C` provides bodies for the virtual methods.

```
class C implements B
begin
```

```
  with Any op foo() == ...
  with Any op bar() == ...
end
```

The type of `self` for class `C` before inheritance is

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \vdots \\ \dot{\tau}^{\text{C self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{implement}\ \dot{\tau}^{\text{B self}} \\ \mathsf{method}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \\ \mathsf{method}\ \texttt{bar} : \mathcal{M}(\top, (), ()) \\ \rho^{\text{C self}} \end{array} \right\} \\ \vdots \end{array} \right\}$$

where the other object types for class `C` are unspecified. The expansion of inheritance must provide a conformance check for overlapping rows. Rows are overlapping when they have the same name. Both

$$\mathsf{method}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \ddot{<} \mathsf{virtual}\ \texttt{foo} : \mathcal{M}(\top, (), ())$$

and

$$\mathsf{method}\ \texttt{bar} : \mathcal{M}(\top, (), ()) \ddot{<} \mathsf{virtual}\ \texttt{bar} : \mathcal{M}(\top, (), ())$$

hold, since a virtual method can be replaced by a regular method. The expansion of implement rows on inheritance, transfers nominal rows, but does not produce super rows. The implement row was introduced in addition to the inherit row to capture this difference. The reasons are that super rows are used for static binding, which may not be provided by interfaces, and interfaces propagate nominal rows on inheritance, while classes do not. The type of `self` for class `C` after inheritance becomes

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \vdots \\ \dot{\tau}^{\text{C self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \mathsf{nominal}\ \texttt{B} \\ \mathsf{method}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \\ \mathsf{method}\ \texttt{bar} : \mathcal{M}(\top, (), ()) \\ \rho^{\text{C self}} \end{array} \right\} \\ \dot{\tau}^{\text{C Self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \mathsf{nominal}\ \texttt{B} \\ \mathsf{method}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \\ \mathsf{method}\ \texttt{bar} : \mathcal{M}(\top, (), ()) \\ \rho^{\text{C Self}} \end{array} \right\} \\ \dot{\tau}^{\text{C closed}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \mathsf{nominal}\ \texttt{B} \\ \mathsf{method}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \\ \mathsf{method}\ \texttt{bar} : \mathcal{M}(\top, (), ()) \end{array} \right\} \\ \dot{\tau}^{\text{C open}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{nominal}\ \texttt{A} \\ \mathsf{nominal}\ \texttt{B} \\ \mathsf{method}\ \texttt{foo} : \mathcal{M}(\top, (), ()) \\ \mathsf{method}\ \texttt{bar} : \mathcal{M}(\top, (), ()) \\ \rho^{\text{C open}} \end{array} \right\} \end{array} \right\}$$

where all object types for class `C` are revealed.

Recall from Section 5.3, that there are several identifier namespaces depending on the context the identifier is used in. The namespace for inheritance was populated by functions, from initial parameters to the closed type for the class. It is not possible to create an object from a class with virtual methods, however the closed type for a class has by definition no virtual methods, even if the type of `self` has. To remedy this, another object type is introduced for classes, the type of τ^{new}. The type τ^{new} preserves virtual rows, and is used to build the namespace for inheritance.

This demonstrates how interfaces lead to the addition of virtual rows to object types, so object types for classes and interfaces are treated uniformly.

5.9 Inheritance

This section further clarifies the notions of inheritance. The word inheritance is used with different meanings in the Creol language. To uncover these meanings, we consider an informal description of each form of inheritance, and show how this is reflected in the type language. The different forms of inheritance are:

`interface B inherit A` Inheritance from an interface `A` to an interface `B` allows reuse and refinement of both method signatures and nominal constraints.

`class D inherit C` Inheritance from a class `A` to a class `B` allows code reuse, however nominal constraints are not included in the inheritance.

`class F implement E` A class `F` can be said to satisfy the nominal constraints of interface `E`, when the class `F` implements all methods specified in interface `E`.

To keep the presentation brief, we assume some things, that are not yet introduced, specifically.

- Only interfaces can introduce nominal rows, and inheritance of nominal rows is represented by the implement row, Section 5.7.
- Method signatures in interfaces are virtual, Section 5.8.
- Regular inheritance is represented by inherit Section 5.4.

The inheritance from interface `A` to interface `B`, is represented with both an inherit row and an implement row, which together account for inheritance of both signatures and constraints.

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^A : \mathcal{O} \left\{ \vdots \right\} \\ \dot{\tau}^B : \mathcal{O} \left\{ \begin{array}{l} \mathsf{inherit}\ \mathtt{A} : \dot{\tau}^A \\ \mathsf{implement}\ \dot{\tau}^A \\ \vdots \end{array} \right\} \end{array} \right\}$$

The inheritance from class `C` to class `D` is represented with an inherit row, which disregards nominal constraints with respect to inheritance.

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^C : \mathcal{O} \left\{ \begin{array}{l} \vdots \end{array} \right\} \\ \dot{\tau}^D : \mathcal{O} \left\{ \begin{array}{l} \mathsf{inherit}\ \mathtt{C} : \dot{\tau}^C \\ \vdots \end{array} \right\} \end{array} \right\}$$

The implementation of interface `E` by class `F` is expressed with an implement row.

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^E : \mathcal{O} \left\{ \begin{array}{l} \vdots \end{array} \right\} \\ \dot{\tau}^F : \mathcal{O} \left\{ \begin{array}{l} \mathsf{implement}\ \dot{\tau}^E \\ \vdots \end{array} \right\} \end{array} \right\}$$

5.10 Statically Bound Instance Variables

In Creol, instance variables are statically bound by the compiler. This section show how such static binding affects variance and enriches the type system.

Creol binds instance variables statically, so the lookup of an instance variable always starts in the current class, and is unaffected by later inheritance. Static binding of instance variables is enforced by the compiler. The static binding avoids overriding of instance variables from the super class point of view. Hence code from a super class never gains access to any variables introduced by a subclass. Intuitively, instance variable lookup is never subject to virtual binding. Recall from Section 5.5, that the variance requirement for methods is due to virtual binding, hence the absence of virtual binding places no variance requirements on instance variables.

Static binding for instance variables requires a new type language construct. Let $\mathcal{SB}(\mathtt{Id}, T)$ be a new type language construct that denotes Static Binding. All instance variables τ for a class `Id` must be inserted into the namespace with type $\mathcal{SB}(\mathtt{Id}, \tau)$. The type construct $\mathcal{SB}$ is ignored by all other rules in the type system, and the information provided by $\mathcal{SB}$ is used by the code generator.

A separate type language construct for static binding of instance variables is necessary, because the Creol language is designed so it is not possible to syntactically differentiate instance variables from other variables, hence information about static binding is not available when parsing.

To simplify the treatment of the Creol type system, the type language construct $\mathcal{SB}$ is not introduced in Section 6.3, and is not mentioned in the Creol type system rules.

5.11 Typing Cointerfaces

This section explains how cointerfaces are typed, which is closely related to the type of `self` for classes.

A cointerface is an interface requirement on caller of a method [51, Page 9]. A cointerface provides semantic information than is not captured by a callback. A callback accepts any object, even if not the caller, that has the required interface. From a type system point of view, a cointerface can be treated like

any other input parameter, indistinguishable from a callback, as the type system does not track object identities. The extra information about object identity expressed by cointerfaces, is captured by special treatment of cointerfaces in the type system. The cointerface requirement must be satisfied by the type of `self` for the caller of a method. The type of `self` for the caller is available during type checking. The `this` always has type of `self`.

The Creol language provides a cointerface, if none is specified by the programmer. The default cointerface $\top$ permits any object to invoke the method. However method signatures in an interface are public by default, while methods in a class are private, to the instance, by default. Therefore method signatures, in a class without a cointerface, are given a personal row, which prevents visibility outside the type of `self`, so the cointerface of a personal row is not important, and is given the type $\top$.

Note that a method row with `self` as cointerface is not equivalent to a personal row, because all object instances of a class have type `self`, and therefore trivially satisfy the cointerface `self`, hence the introduction of personal row is necessary to provide instance privacy, as was discussed in Section 5.2.

Although cointerfaces play no special role in the type analysis, cointerfaces are envisioned to support analysis of security properties, that relate to object identities [50]. The nature of these properties is currently a target for research.

5.12 Explicit Type Language References

The Creol type system distinguishes between references and values, although the Creol language does not. This section motivates, that references are represented by a separate type language construct. References provide a type safe model for the use of mutable storage in programs, and was early recognised as a natural model for mutability [25].

Recall from Section 5.5, that the flow of information determines variance, and that conformance between references requires type equality, because the value contained in a reference may be both stored and retrieved. If every type is considered a reference, then static type safety does not allow any variance. The introduction of *ref* τ to denote a reference to a value of type τ, assures that conformance for references satisfy invariance, while contra- and covariance is possible for other types.

The explicit presence of references in the Creol type language, allows the rule for references from Section 5.5 to describe variance for references. This has simplified the other conformance rules, by making the treatment of references is orthogonally to the other type system rules.

The interaction between mutability and unbounded parametrisation is potentially unsafe, and some care is required to ensure type safety. One conservative strategy for achieving type safety is the `value restriction` [77, Page 336]. A more permissive strategy, that can statically type check a larger set of programs, is known as the `relaxed value restriction` [46]. The presence of references in the type system, and in the programming language, is common for these approaches to type safe combination of unbounded parametrisation and mutability. The approach taken by the Creol type system is more conservative than the value restriction, and does not require references in the language, by not permitting references to free types introduced by unbounded parametrisa-

tion [46, Sect. 2.2]. This is ensured by the rule

$$\frac{\bot}{\Theta, \Gamma \vdash \mathit{ref}\ \alpha}$$

where α is a free type defined by unbounded parametrisation. The rule ensures type safety for parametrisation and references, in a very simple manner. When the Creol language matures, it is reasonable to use the relaxed value restriction, that can statically type check a larger set of programs, than the current rule.

The presence of explicit references in the type system, does not mean that the programmer is burdened with references. The parser can insert reference annotations where appropriate. The separation of concerns ensures, that the reference annotations do not affect type safety. The type system ensures type safety for any use of references, independent of annotations. The presence of references in the Creol type language ensures, that unbounded parametrisation and references are handled in a type safe manner in the Creol type system.

5.13 Static and Virtual Method Binding

This section explains how static binding is type checked.

Recall from Section 2.1.3, that the Creol language allows code in a super class to control method override at the call site, where the method invocation is made, by using either virtual or static binding. Virtual binding is the default in Creol. Static binding can be designated internally in a class, with the syntax `m@C`, which denotes that lookup of code for method `m` should start in class `C`, where `C` is a superclass of the class of the caller. This prevents method overrides below class `C` in the inheritance hierarchy from affecting the call site.

The static binding `m@C` requires, that the code for method `m` can be found in class `C`, or in some of the super classes of `C`. The expansion of inheritance, between classes, includes `super` rows with the name of the super class. The class `C` is looked up amongst the `super` rows in the type of `self` for the current class. Recall from Section 2.1.2, that the Creol language permits multiple inheritance for both classes and interfaces, so there can be multiple `super` rows.

The addition of a `super` row to make code from super classes available, is also used by other languages such as O'Caml [73]. The Creol language allows static binding to the current class, not just super classes. This is realised by inserting a `super` row for the type of `self`. Each class inserts its own `super` row, and these are included by inheritance.

Inheritance is always expanded in the Creol type system, because it simplifies the treatment of rows in type judgements, but there are no theoretical reasons that require expansion of inheritance, for the type of `self`, however, inheritance is an implementation detail, so inheritance must be expanded for all object types, with the possible exception of the type of `self`.

Type checking of static binding requires a modest extension to the type system, and integrates naturally.

5.14 Recursive Types

This section investigates the formal foundation for recursive types in the Creol type system. The Creol system can express both types that are recursive with

respect to themselves, recursive types, and types that are recursive with respect each other, mutually recursive types. This is achieved by introduction of a type environment Θ, along with an implicit recursive interpretation of the type environment

$$\Theta \stackrel{\text{def}}{=} \left\{(\dot{\tau}_i, \tau_i)_{i \in I}\right\}$$

where $\dot{\tau}$ is a placeholder type, and fixpoint, for τ, and $\Theta \vdash \dot{\tau} : \tau$ holds when

$$(\dot{\tau}, \tau) \in \left\{(\dot{\tau}_1, \tau_1), \ldots, (\dot{\tau}_n, \tau_n)\right\}$$

5.14.1 Mutual Recursion

Mutual recursion is facilitated by placeholder types. Intuitively, a placeholder type provides indirection, and given a placeholder type $\dot{\tau}$ and a type environment Θ, the type τ can be found in Θ when necessary. Although recursion between types in Creol is mainly between objects, classes and interfaces, we shall look at tuples, to make the presentation simpler. Consider two imaginary tuples `a` and `b` of types τ^a and τ^b , that are mutually recursive, so the type of either tuple is infinite:

$$\tau^a \stackrel{\text{def}}{=} (\mathit{Int}, (\mathit{Str}, (\mathit{Int}, (\mathit{Str}, (\mathit{Int}, (\mathit{Str}, \ldots))))))$$

$$\tau^a \stackrel{\text{def}}{=} (\mathit{Str}, (\mathit{Int}, (\mathit{Str}, (\mathit{Int}, (\mathit{Str}, (\mathit{Int}, \ldots))))))$$

To represent these types in a finite manner we need a type environment that provides indirection for types. Let $\dot{\tau}^a$ be a placeholder for τ^a, and $\dot{\tau}^b$ be a placeholder for τ^b, then the tuples are described in a finite manner by the type environment

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^a : \left(\mathit{Int}, \dot{\tau}^b\right) \\ \dot{\tau}^b : (\mathit{Str}, \dot{\tau}^a) \end{array} \right\}$$

because of the indirection introduced by placeholder types. Now we can write down the types of `a` and `b` in the namespace

$$\Gamma \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathtt{a} : \dot{\tau}^a \\ \mathtt{b} : \dot{\tau}^b \end{array} \right\}$$

and by definition of Θ and Γ in Section 6.2, we have

$$\Theta, \Gamma \vdash \mathtt{a} : \dot{\tau}^a, \mathtt{b} : \dot{\tau}^b$$

but we can neither full expand $\Theta(\dot{\tau}^a)$ nor $\Theta(\dot{\tau}^b)$, that would produce the infinite types we started with. The solution is to do partial expansion, and only when necessary, so from

$$\Theta, \Gamma \vdash \mathtt{a} : \dot{\tau}^a$$

we can partial expand to

$$\Theta, \Gamma \vdash \mathtt{a} : \left(\mathit{Int}, \dot{\tau}^b\right)$$

while still having a finite representation. Intuitively this kind of mutual recursion can occur between a producer interface and a consumer interface. Note that placeholder types also allow sharing, so given $\Theta \stackrel{\text{def}}{=} \{(\dot{\tau}, \tau)\}$, the placeholder type $\dot{\tau}$ can be used by several other types, rather than having several copies of τ, which would be necessary without placeholder types.

5.14.2 Iso-recursion

Before we introduce the implicit recursive interpretation, we investigate how recursion is handled explicitly. The implicit recursive interpretation of placeholder types as fixpoints, is based on the explicit *iso-recursive* model, as described by Pierce [77, Sect. 20.2, 21.11] and Bruce [15, Sect. 9.2].

The iso-recursive model expresses recursion through a fixpoint binder μ, and introduces the term *iso-equivalent*, so a bound fixpoint is iso-equivalent to its fixpoint expression. This is written $\mu\alpha\,(\tau)$, where α is, by definition, iso-equivalent to τ, which means that the free type α can occur in τ, and μ tells us that α is the fixpoint of an infinite type τ^{∞} described by τ. Iso-equivalence states that two entities are iso-equivalent, when the only difference is the level of (un)folding. To provide some intuition, let us consider a simple example such as the recursive tuple `c`, however, since the type τ^c is recursive, it is also infinite, and the infinite type τ^c is approximately

$$\tau^c \stackrel{\text{def}}{=} (Int, (Int, (Int, (Int, (Int, \dots)))))$$

We can describe the structure of the infinite type τ^c as $\mu\alpha\,(Int, \alpha)$, and use the keyword *Fix* to document, that we interpret the finite representation as an infinite type. Therefore

$$\tau^c \stackrel{\text{def}}{=} Fix\ \mu\alpha\,(Int, \alpha)$$

precisely describes τ^c in a finite manner. Any required part of an infinite type is available by the repeated use of *unfold*, since

$$unfold\ Fix\ \mu\alpha.(Int, \alpha) \rightsquigarrow Fix\ \mu\alpha.(Int, Fix\ \mu\alpha.(Int, \alpha))$$

where unfold can be used repeatedly as follows.

$$Fix\ \mu\alpha.(Int, unfold\ Fix\ \mu\alpha.(Int, \alpha)) \rightsquigarrow Fix\ \mu\alpha.(Int, Fix\ \mu\alpha.(Int, Fix\ \mu\alpha.(Int, \alpha)))$$

Notice that to recover the infinite type, unfold must be used an infinite number of times.

The *fold* function can reverse a unfolding, so

$$fold\ Fix\ \mu\alpha.(Int, Fix\ \mu\alpha.(Int, \alpha)) \rightsquigarrow Fix\ \mu\alpha.(Int, \alpha)$$

There is iso-equivalence between $Fix\ \mu\alpha.(Int, \alpha)$ and $Fix\ \mu\alpha.(Int, Fix\ \mu\alpha.(Int, \alpha))$, because *fold* and *unfold* can turn either into the other

$$Fix\ \mu\alpha.(Int, \alpha) \overset{\overset{\text{unfold}}{\longrightarrow}}{\underset{\text{fold}}{\overset{\text{iso-equivalence}}{=}}\underset{\longleftarrow}{}} Fix\ \mu\alpha.(Int, Fix\ \mu\alpha.(Int, \alpha))$$

5.14.3 Implicit Iso-recursion

With a clear definition of iso-recursion, we can show exactly how placeholder types implicitly correspond to explicit iso-recursion.

Intuitively, the tuple `c` with the iso-recursive type

$$\tau^c \stackrel{\text{def}}{=} Fix\ \mu\alpha\,(Int, \alpha)$$

can be expressed, with an implicit iso-recursive interpretation, by the type environment

$$\Theta \stackrel{\text{def}}{=} \{\dot{\tau}^c : (Int, \dot{\tau}^c)\}$$

which is very similar.

All object types are potentially recursive in Creol, thus they are all subject to the iso-recursive interpretation. In the type system presentation, this interpretation will be implicit, but it is explicit here in order to show that the use of a type environment with an implicit recursive interpretation of placeholder types, correctly implements iso-recursive types.

Generally, for a recursive type τ^∞, written with explicit iso-recursion as $Fix\ \mu\alpha.\tau^\mu$, the type environment

$$\Theta \stackrel{\text{def}}{=} \{\dot{\tau} : \tau\}$$

express τ^∞ , where the placeholder type $\dot{\tau}$ corresponds to the fixpoint α, and $\tau \stackrel{\text{def}}{=} \tau^\mu_{\alpha \mapsto \dot{\tau}}$, so τ is derived from τ^μ by replacing the free type α with the placeholder type $\dot{\tau}$. This allows $\dot{\tau}$ and τ to describe the infinite type τ^∞, which can be illustrated as

$$\Theta \stackrel{\text{def}}{=} \left\{ \overbrace{\dot{\tau} : \tau}^{\tau^\infty} \right\}$$

and there is now a correspondence

$$Fix\ \mu\alpha\,(\tau^\mu) \stackrel{\text{correspondence}}{\Leftrightarrow} \Theta \stackrel{\text{def}}{=} \{\dot{\tau} : \tau\}$$

which also holds between unfolding and partial expansion

$$unfold\ \alpha \rightsquigarrow Fix\ \mu\alpha\,(\tau^\mu) \stackrel{\text{correspondence}}{\Leftrightarrow} \frac{(\dot{\tau}, \tau) \in \Theta}{\Theta \vdash \ldots \dot{\tau} \rightsquigarrow \Theta \vdash \ldots \tau}$$

Thus

$$\Theta \stackrel{\text{def}}{=} \{\dot{\tau} : \tau\}$$

is implicitly understood as

$$\Theta \stackrel{\text{def}}{=} \left\{ \overbrace{Fix\ \mu}^{\text{implicit}}\ \dot{\tau}\,(\tau) \right\}$$

where the bound fixpoint $\dot{\tau}$ scopes over all types in the program and $\dot{\tau}$ is iso-equivalent to τ by partial expansion

$$\frac{(\dot{\tau}, \tau) \in \Theta}{\Theta \vdash \ldots \dot{\tau} \rightsquigarrow \Theta \vdash \ldots \tau}$$

which is written

$$\Theta \vdash \ldots \dot{\tau} : \tau$$

for convenience. The infinite type τ^∞ is always available by infinite use of partial expansion, called full expansion, $\Theta\,(\dot{\tau}) = \tau^\infty$.

Neither the Creol language nor the other parts of the Creol type system is concerned with the use of *fold* and *unfold*, the compiler does folding and

unfolding as necessary, and due to iso-equivalence, the level of (un)folding does not matter.

Note that since placeholder types are fixpoints, it is possible to give a precise treatment of fixpoints for the conformance relation, which is done later in Section 5.17.

5.15 Termination of Inheritance Checking

In order to ensure the termination of type checking with inheritance, it is crucial that inheritance is expressed in the type language.

The problem is that without inheritance in the type system, type checking will not terminate when the user has made the type error of circular inheritance. The reason for this is, that expansion of inheritance does not terminate unless the inheritance tree is a directed acyclic graph (DAG). The introduction of the row $\mathsf{inherit}\ \tau$ where τ is not expanded, allows the compiler to check that the inheritance hierarchy is a DAG in a separate step. The use of a type environment Θ with placeholder types $\dot{\tau}$ and types τ, avoids the expansion of τ by introduction of implicit fixpoints. This was described in Section 5.14.

The following example demonstrates a simple case of circular inheritance.

```
class A inherits B
begin
end

class B inherits A
begin
end
```

This inheritance does not make sense, and to discover this, we wish to write down the type of `self` for `A` and `B` prior to inheritance and check if the inheritance makes sense or not. It is possible to perform the inheritance and terminate when the inheritance makes sense. This is achieved in a clean and clear manner with a row for inheritance. The inherit row with a placeholder type $\dot{\tau}$ can write out the cyclic inheritance with finite types, thus inheritance cycles can be found. With a row variable for inheritance, the type of `self` for `A` is given the placeholder type $\dot{\tau}^{\texttt{A self}}$ with the type

$$\mathcal{O}\left\{\mathsf{inherit}\ \texttt{B} : \dot{\tau}^{\texttt{B self}}\right\}$$

and the type of `self` for `B` is given the placeholder type $\dot{\tau}^{\texttt{B self}}$ with the type

$$\mathcal{O}\left\{\mathsf{inherit}\ \texttt{A} : \dot{\tau}^{\texttt{A self}}\right\}$$

which are both finite types. It is now possible to analyse the type environment

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^{\texttt{A self}} : \mathcal{O}\left\{\mathsf{inherit}\ \texttt{B} : \dot{\tau}^{\texttt{B self}}\right\} \\ \dot{\tau}^{\texttt{B self}} : \mathcal{O}\left\{\mathsf{inherit}\ \texttt{A} : \dot{\tau}^{\texttt{A self}}\right\} \end{array} \right\}$$

and give an error message, since the inheritance hierarchy is not a DAG. When the inheritance hierarchy is a DAG, the traversal of the inheritance hierarchy will always terminate, such that inheritance can be safely performed.

5.16 Polymorphism

This section provides some intuition on polymorphism in the Creol type system.

Recall from Section 2.3.6, that we have suggested to extend the Creol language with unconstrained and constrained polymorphism. Unconstrained polymorphism is written $\forall\alpha\,(\tau)$, where the type α is free in τ. Constrained polymorphism is written $\forall\alpha\ddot{<}\tau^{\text{sup}}\,(\tau)$, where the type α must conform to τ^{sup}, and α is free in τ.

Constrained polymorphism is defined with the conformance relation. By definition, the conformance relation reduces to either matching or subtyping, due to the handling of fixpoints and object types, which is treated later in Section 5.17. When conformance reduces to subtyping, the constrained parametrisation is by definition *F-bounded quantification* [77, Page 393]. When conformance reduces to matching, the constrained parametrisation is by definition match-bounded polymorphism [15, Sect. 17.3].

By definition of conformance, the constrained polymorphism $\forall\alpha\ddot{<}\tau^{\text{sup}}\,(\tau)$ reduces to match-bounded polymorphism, when τ^{sup} is an open object type, and otherwise to F-bounded polymorphism.

Polymorphism represents an abstraction with respect to the free type α. When a polymorphic type is used, the free type α must be instantiated, that is given a type. To illustrate how polymorphism and instantiation are represented in the Creol type system, consider the following example which creates a class with unconstrained polymorphism.

```
class A[T] // abstraction with respect to T: α
begin
  var t:T ; // Trivial use of α
  with Any op assign(in x:T) == t := x ;
end
```

The type environment for class `A` becomes

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^{\text{A self}} : \mathcal{O}\left\{ \begin{array}{l} \mathsf{variable}\ \mathtt{t} : \alpha \\ \mathsf{method}\ \mathtt{assign} : \mathcal{M}(\top, \alpha, ()) \\ \rho^{\text{A self}} \end{array} \right\} \\ \dot{\tau}^{\text{A Self}} : \mathcal{O}\left\{ \begin{array}{l} \mathsf{method}\ \mathtt{assign} : \mathcal{M}(\top, \alpha, ()) \\ \rho^{\text{A Self}} \end{array} \right\} \\ \dot{\tau}^{\text{A closed}} : \mathcal{O}\left\{ \mathsf{method}\ \mathtt{assign} : \mathcal{M}(\top, \alpha, ()) \right\} \\ \dot{\tau}^{\text{A open}} : \mathcal{O}\left\{ \begin{array}{l} \mathsf{method}\ \mathtt{assign} : \mathcal{M}(\top, \alpha, ()) \\ \rho^{\text{A open}} \end{array} \right\} \end{array} \right\}$$

where the free type α can be used freely. The parametrisation is not part of the object descriptions for class `A`. Intuitively the parametrisation happens outside the object descriptions. This is important, because of the implicit fixpoint interpretation of Θ, the parametrisation is not part of the fixpoint for an object. The parametrisation is placed outside the object types in the namespaces for identifiers. Recall from Section 5.3, that there are several namespaces, and that the one used, depends on the context where the identifier occurs. The namespaces for class `A` are

$$\Gamma \stackrel{\text{def}}{=} \left\{\ \mathtt{A} : \forall\alpha\left(\dot{\tau}^{\text{A closed}}\right)\ \right\}$$

$$\Gamma^{\text{new}} \stackrel{\text{def}}{=} \left\{ \texttt{A} : \forall\alpha \left(\dot{\tau}^{\text{A closed}} \right) \right\}$$

$$\Gamma^{\text{inherit}} \stackrel{\text{def}}{=} \left\{ \texttt{A} : \forall\alpha \left(\dot{\tau}^{\text{A self}} \right) \right\}$$

where the object types are properly contained by parametrisation.

When the class `A` is used, the parametrised types must be instantiated. Now consider code, using the class `A` in different manners, analogous to the examples from Section 5.3.

`var o:A[Bool]` The type of `A` is $\forall\alpha \left(\dot{\tau}^{\text{A closed}} \right)$. The type of `o` is

$$\dot{\tau}^{a} \stackrel{\text{def}}{=} \left(\dot{\tau}^{\text{A closed}} \right)_{\alpha \mapsto Bool}$$

which is the type $\dot{\tau}^{\text{A closed}}$ where α is substituted by $Bool$.

`new A[Str]` The type of `A` is $\forall\alpha \left(\dot{\tau}^{\text{A closed}} \right)$. The type produced by `new` is

$$\dot{\tau}^{b} \stackrel{\text{def}}{=} \left(\dot{\tau}^{\text{A closed}} \right)_{\alpha \mapsto Str}$$

which is the type $\dot{\tau}^{\text{A closed}}$ where α is substituted by Str.

`inherit A[Int]` The type of `A` is $\forall\alpha \left(\dot{\tau}^{\text{A self}} \right)$. The type seen by `inherit` is

$$\dot{\tau}^{c} \stackrel{\text{def}}{=} \left(\dot{\tau}^{\text{A self}} \right)_{\alpha \mapsto Int}$$

,which is the type $\dot{\tau}^{\text{A self}}$ where α is substituted by Int.

Neither of these uses of parametrisation may modify the original parametrised type, rather they create copies of the original type, and perform the substitution on the copies. The act of doing this is called instantiation. Since object types are recursive, these copies are represented in Θ, which produces the following type environment.

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \vdots \\ \dot{\tau}^{a} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{method}\ \texttt{assign} : \mathcal{M}(\top, Bool, ()) \end{array} \right\} \\ \dot{\tau}^{b} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{method}\ \texttt{assign} : \mathcal{M}(\top, Str, ()) \end{array} \right\} \\ \dot{\tau}^{c} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{variable}\ \texttt{t} : Int \\ \mathsf{method}\ \texttt{assign} : \mathcal{M}(\top, Int, ()) \\ \rho^{a} \end{array} \right\} \end{array} \right\}$$

This model is, however, too simple. Recall from Section 5.15, that type checking does not terminate unless all inheritance hierarchies are directed acyclic graphs (DAG). Although the instantiation of $\dot{\tau}^{c}$ required by `inherit A[Int]` terminates, it would not terminate for a cyclic inheritance hierarchy. To ensure, that inheritance checking terminates, it is necessary to postpone the instantiation. Substitution is made part of the type language, to express future instantiation. A substitution on a placeholder type must always instantiate, that is, first copy the type of the placeholder, and then perform the substitution on the copy. Note that instantiation requires a change of fixpoints as well, which is added

to the substitution. The type and namespace environments in the previous example would therefore go through the intermediary type environment

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \vdots \\ \dot{\tau}^a : \left(\left(\dot{\tau}^{\text{A closed}}\right)_{\alpha \mapsto Bool}\right)_{\dot{\tau}^{\text{A closed}} \mapsto \dot{\tau}^a} \\ \dot{\tau}^b : \left(\left(\dot{\tau}^{\text{A closed}}\right)_{\alpha \mapsto Str}\right)_{\dot{\tau}^{\text{A closed}} \mapsto \dot{\tau}^b} \\ \dot{\tau}^c : \left(\left(\dot{\tau}^{\text{A self}}\right)_{\alpha \mapsto Int}\right)_{\dot{\tau}^{\text{A self}} \mapsto \dot{\tau}^c} \end{array} \right\}$$

which would allow the inheritance hierarchy to be checked for cycles. Then inheritance could be expanded, as well as instantiation and substitution performed, to produce the final type environment.

5.17 Iso-recursion and Conformance

This section investigates the conformance relation with respect to iso-recursion, and shows how conformance expresses both subtyping and matching correctly with respect to fixpoints.

The conformance relation $\dot{\tau}^b \ddot{<} \dot{\tau}^a$ reduces to matching $\dot{\tau}^b < \#\dot{\tau}^a$, when τ^a is an open object type, that is, has a free row ρ, and otherwise to subtyping $\dot{\tau}^b <: \dot{\tau}^a$, given the following type environment.

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^a : \tau^a \\ \dot{\tau}^b : \tau^b \end{array} \right\}$$

5.17.1 Subtyping

This section investigates how to determine subtyping between iso-recursive types. This is done by means of the *Amber rule*, as discussed by Pierce [77, Sec. 21.11]. The same rule is used by Bruce [15, Sec. 9.4], although he does not go into the details of the Amber rule. The Amber rule states, that two iso-recursive types $Fix\ \mu\alpha^b\left(\tau^{\mu b}\right)$ and $Fix\ \mu\alpha^a\left(\tau^{\mu a}\right)$ are subtypes, if $\tau^{\mu b}$ and $\tau^{\mu a}$ are structural subtypes under the assumption, that the fixpoints α^b and α^a are subtypes, where the fixpoints α^b and α^a are renamed if necessary, so they are different, as expressed by the following rule.

$$\frac{\{\alpha^b <: \alpha^a\}, \Gamma \vdash \tau^{\mu b} <: \tau^{\mu a}}{(Fix\ \mu\alpha^b(\tau^{\mu b})) <: (Fix\ \mu\alpha^a(\tau^{\mu a}))}$$

The Amber rule requires fixpoints must to be different. This is realised for the Creol type system by the implicit iso-recursive interpretation of the type environment Θ, as each recursive type has a different placeholder type as its fixpoint. The type environment must be extended to store assumptions about fixpoints, so

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} (\dot{\tau}, \tau)_I \\ (\dot{\tau}, \dot{\tau})_J \end{array} \right\}$$

where $(\dot{\tau}, \dot{\tau})_J$ is a set of fixpoint assumptions, as introduced by the Amber rule. This can also be written

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} (\dot{\tau} : \tau)_I \\ (\dot{\tau} \ddot{<} \dot{\tau})_J \end{array} \right\}$$

which is visually more intuitive. Partial expansion of fixpoints for conformance must be treated specially, to ensure that necessary fixpoint assumptions are introduced into Θ. Note, that subtyping can only consider closed types, because subsumption is only defined for closed types, hence both τ^b and τ^a must be closed. This is expressed by the rule

$$\frac{\Theta \cup \left\{\dot{\tau}^b \ddot{<} \dot{\tau}^a\right\}, \Gamma \vdash \tau^b \ddot{<} \tau^a \quad \text{closed } \tau^a \quad \text{closed } \tau^b}{\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{c} \dot{\tau}^a : \tau^a \\ \dot{\tau}^b : \tau^b \end{array} \right\}, \Gamma \vdash \dot{\tau}^b \ddot{<} \dot{\tau}^a}$$

If the fixpoints already have an associated assumption, then the assumption decides whether the relation holds. When the conformance between fixpoints is supported by an assumption, the following rule holds.

$$\frac{\left(\dot{\tau}^b, \dot{\tau}^a\right) \in \Theta \quad \top}{\Theta, \Gamma \vdash \dot{\tau}^b \ddot{<} \dot{\tau}^a}$$

However, when the conformance between fixpoints is falsified by an assumption, the conformance does not hold, falsified by the following rule.

$$\frac{\left(\dot{\tau}^a, \dot{\tau}^b\right) \in \Theta \quad \bot}{\Theta, \Gamma \vdash \dot{\tau}^b \ddot{<} \dot{\tau}^a}$$

This demonstrates how subtyping between iso-recursive types is decided with the Amber rule in the Creol type system.

5.17.2 Matching

The Creol rule for matching is an adaptation of the definition of type equality for objects, encoded with extensible row polymorphic records from Rémy and Vouillon [80, Sect. 2], [79, App. B], [27, Sect. 3.4].

The essence of matching for iso-recursive types is, that matching disregards fixpoint differences, and, that matching may instantiate a row . Matching allows the type of `self` to be refined, which corresponds to a disregard for fixpoint differences. Matching requires the super type to have a free row variable ρ. When matching is performed, the row ρ can remember the application, to preserve the type of the lower bound. This is the same as an instantiation of the free row variable, where each match introduces an instantiation. The rule for matching is therefore

$$\frac{\left(\Theta, \Gamma \vdash \rho_j^b \ddot{<} \left(\rho_i^a\right)_{\dot{\tau}^a \mapsto \dot{\tau}^b}\right)_{i,j \in I \cap J} \quad \rho_{I \setminus J} = \emptyset}{\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{c} \dot{\tau}^a : \mathcal{O}\left\{ \begin{array}{c} \rho_I^a \\ \rho \end{array} \right\} \\ \dot{\tau}^b : \mathcal{O}\left\{\rho_J^b\right\} \end{array} \right\}, \Gamma \vdash \dot{\tau}^b \ddot{<} \dot{\tau}^a \diamond \Theta \cup \left\{\dot{\tau}^c : \left(\left(\dot{\tau}^a\right)_{\dot{\tau}^a \mapsto \dot{\tau}^c}\right)_{\rho \mapsto \left\{\rho_{J \setminus I}\right\}}\right\}, \Gamma}$$

where
$\rho_{I \setminus J}$ is the set of rows in ρ_I, that are not present in ρ_J. Is such rows exists, the conformance does not hold, because the requirements from the super type is not met by the subtype.

$\rho_{J\setminus I}$ is the set of rows in ρ_J that are not present in ρ_I. These rows are not important for conformance, however matching remembers those unimportant rows, by instantiation of the free row variable ρ.
$(\dots \rho_j \dots \ddot{\leq} \dots \rho_i \dots)_{i,j \in I \cap J}$ is the set of conformance restrictions on those rows that overlap between ρ_I and ρ_J, where $I \cap J$ denotes the set of index pairs for overlapping rows.

An intuition for the different parts of the matching rule is now given.
$(\rho_i^a)_{\dot{\tau}^a \mapsto \dot{\tau}^b}$ Disregard fixpoint differences between the subtype and super type, by replacing the fixpoint of the super type by that of the subtype.
$\left(\Theta, \Gamma \vdash \rho_j^b \ddot{\leq} (\rho_i^a)_{\dot{\tau}^a \mapsto \dot{\tau}^b}\right)_{i,j \in I \cap J}$ Conformance must be satisfied for all rows in the super type.
$\dot{\tau}^c$ The super type instantiated for this particular match. This instantiation allows the application of a procedure to preserve an exact object type, so the type $\dot{\tau}^c$ is made available in the environment.
$(\dot{\tau}^a)_{\dot{\tau}^a \mapsto \dot{\tau}^c}$ An instantiation of the super type that changes the fixpoint, which helps to build the type $\dot{\tau}^c$. Note that instantiation creates a copy, so the super type itself is unaffected.
$((\dot{\tau}^a)_{\dot{\tau}^a \mapsto \dot{\tau}^c})_{\rho \mapsto \{\rho_{J\setminus I}\}}$ Instantiates the row variable ρ in $(\dot{\tau}^a)_{\dot{\tau}^a \mapsto \dot{\tau}^c}$ such that the type $\dot{\tau}^c$ remembers those rows that were matched by the row ρ.

Note that the type $\dot{\tau}^c$ remembers this particular matching, which is most usefull to preserve the type of an object after a match, such that transformation functions can be applied to different objects and preserve the type of the object, which allows the transformation to be performed in place. With subtyping the transformer function would change the type of the object, and require a type cast to recover the original type. Thus matching, with instantiation to remember the lower bound, allows more programs to be statically type checked, as a type cast is a workaround when statical type checking is insufficient.

This demonstrates how matching between iso-recursive types is decided, and how matching instantiates the row variable in the super type.

5.17.3 Decidability

This section identifies the requirements for recursive types and polymorphism, to achieve decidability for the conformance relation.

Conformance is based on subtyping and matching, therefore conformance is decidable, if both subtyping and matching are decidable. Subtyping is decidable for a recursive type $Fix\ \mu\dot{\tau}\,(\tau)$, when τ is contractive [77, Sect. 21.8.2]. Intuitively τ is contractive, when the fixpoint $\dot{\tau}$ in τ is contained in some type language construct. Thus $Fix\ \mu\dot{\tau}\,(\dot{\tau})$ is not contractive, however $Fix\ \mu\dot{\tau}\,(\dot{\tau} \to \dot{\tau})$ is contractive. Matching does not change the fixpoint semantics of the Creol type system, therefore conformance for recursive types is decidable, when all of τ_I from

$$\Theta \stackrel{\text{def}}{=} \left\{(\dot{\tau}, \tau)_{i \in I}\right\}$$

are contractive.

Conformance is decidable, when subtyping is decidable for F-bounded polymorphism, and matching is decidable for match-bounded polymorphism. Sub-

typing for F-bounded polymorphism is undecidable for full $F_{<:}$ with the rule

$$\frac{\Theta, \Gamma \vdash \tau^a \ddot{>} \tau^c \quad \Theta \cup \{\alpha : \tau^c\}, \Gamma \vdash \tau^b \ddot{<} \tau^d}{\Theta, \Gamma \vdash \left(\forall \alpha \ddot{<} \tau^a(\tau^b)\right) \ddot{<} \left(\forall \alpha \ddot{<} \tau^c(\tau^d)\right)}$$

where contravariance is required for the constraint, because a constraint is a sinks, a receiver of information [77, Sect. 28.5].

Subtyping is decidable for kernel $F_{<:}$ [77, Sect. 28.3], where kernel $F_{<:}$ restricts subtyping somewhat with the rule

$$\frac{\Theta, \Gamma \vdash \tau^a \ddot{=} \tau^c \quad \Theta \cup \{\alpha : \tau^c\}, \Gamma \vdash \tau^b \ddot{<} \tau^d}{\Theta, \Gamma \vdash \left(\forall \alpha \ddot{<} \tau^a(\tau^b)\right) \ddot{<} \left(\forall \alpha \ddot{<} \tau^c(\tau^d)\right)}$$

where the constraints must be identical.

Subtyping is also decidable without restrictions, when the type system is limited to the prenex fragment of full $F_{<:}$ [76, Sect. 12]. The prenex fragment only permits quantification as input parameter, so the prenex fragment for $\tau_1 \to \tau_2 \to \ldots$ only permits τ_1 to be quantified. However τ_1 can be a tuple with several quantified types.

The conformance relation $\ddot{<}$ for unconstrained polymorphism restricts the type language to the prenex fragment with the following rule.

$$\frac{\begin{array}{cc} \Theta, \Gamma \vdash \tau^a \ddot{>} \tau^c & \text{prenex } \tau^b \\ \Theta \cup \{\alpha : \tau^c\}, \Gamma \vdash \tau^b \ddot{<} \tau^d & \text{prenex } \tau^d \end{array}}{\Theta, \Gamma \vdash \left(\forall \alpha \ddot{<} \tau^a(\tau^b)\right) \ddot{<} \left(\forall \alpha \ddot{<} \tau^c(\tau^d)\right)}$$

Match-bounded polymorphism does not affect decidability of conformance, because it is realised in a manner orthogonal to the representation of constrained polymorphism. That is, match-bounded polymorphism can be expressed without constrained quantification [15, Sect. 17.3,18].

Chapter 6

Creol Type System

This section presents the Creol type system, with focus on object-orientation. The presentation starts with the assumed desugaring, introduces the meta-syntax, and the type language, which are used in the presentation of type rules. For a gentle start on type rules it is adviceable to start with Appendix B, which presents the type rules for the functional part of Creol.

The Creol type rule presentation starts with the simple and obvious cases, and then progresses towards the advanced. The chapter does not describe how the rules are applied, which is left for Appendix D.1. The type checking process is finished, when all parts of a program have been reduced to axioms by type checking rules. The order of rule application must respect environment update requirements, but are otherwise unimportant.

The type checking process depends on a properly built symbol table. The Creol type system specifies how the symbol table is built through environment updates of Γ. The code for environment handling is found in Appendix E.7.4, Appendix E.7.5 and Appendix E.7.6.

6.1 Desugaring

The type checking rules are given for a kernel language, so syntactic sugar concerns the type checking as little as possible. The rules for removal of syntactic sugar use the symbol $A \rightsquigarrow B$ when A is rewritten to B by desugaring. The rules for removal of syntactic sugar are:

- `var` $\mathtt{id}_1$`=`$\mathtt{e}_1$`,...,`$\mathtt{id}_n$`=`$\mathtt{e}_n$`:T` $\rightsquigarrow$ `var` $\mathtt{id}_1$`,...,`$\mathtt{id}_n$`:T;`$\mathtt{id}_1$`:=`$\mathtt{e}_1$`;...;`$\mathtt{id}_n$`:=`$\mathtt{e}_n$

  Translate the initialisation of variables into separate declaration and assignment sentences.

- `var` $\mathtt{id}_1$`,...,`$\mathtt{id}_n$`:T` $\rightsquigarrow$ `var` $\mathtt{id}_1$`:T...var` $\mathtt{id}_n$`:T`

  Expand shorthand notation for several variables of the same type.

- `var` $\mathtt{id}_1$`:`$\mathtt{T}_1$`,...,`$\mathtt{id}_n$`:`$\mathtt{T}_n$ $\rightsquigarrow$ `var` $\mathtt{id}_1$`:`$\mathtt{T}_1$`...var` $\mathtt{id}_n$`:`$\mathtt{T}_n$

  Expand shorthand notation for several declarations

- `class ... op` $\mathtt{m}_1$ `... op` $\mathtt{m}_n$ $\rightsquigarrow$ `class ... with this op` $\mathtt{m}_1$ `... op` $\mathtt{m}_n$

  Expand missing `with` cointerface to `with this` for classes.

- `interface` ... `op` $\mathtt{m}_1$ ... `op` $\mathtt{m}_n$ $\rightsquigarrow$ `interface` ... `with Any op` $\mathtt{m}_1$ ... `op` $\mathtt{m}_n$

  Expand missing `with` cointerface to `with Any` for interfaces.

- `with Id op` $\mathtt{m}_1$ ... `op` $\mathtt{m}_n$ $\rightsquigarrow$ `with Id op` $\mathtt{m}_1$ ... `with Id op` $\mathtt{m}_n$

  Expand shorthand notation for same `with Id` cointerface to separate cointerface declaration for each method.

- `data` ... `=` ... `|` `Id` ... $\rightsquigarrow$ `data` ... `=` ... `|` `Id Void` ...

  Expand empty variant types to have type `Void`.

- ... `Id` ... $\rightsquigarrow$ ... `Id Void` ...

  Expand constructors for empty variant types to receive type `Void`, when inside an expression or case pattern.

- ... `[]` ... $\rightsquigarrow$ `Nil`

  Expand an empty list to the variant `Nil`

- ... `[`... `::` ... `]` ... $\rightsquigarrow$ ... `List(head=`...`,tail=`...`)` ...

  Expand a list pattern or a list expression into the equivalent `List` variant. The symbol `::` connects a head element to a tail list, as suggested in Section 2.3.9.

- `case`...`v`...`when guard` $\rightsquigarrow$ `case`...$\mathtt{id}^{\mathcal{F}}$...`when guard and` $\mathtt{id}^{\mathcal{F}}$`=v`

  Translate a value pattern `v` into a guard by using a fresh identifier $id^{\mathcal{F}}$. Note that this translation takes place in case patterns, and that `v` is a value, and not a variable.

The usage of comma or semicolon to separate list items is not specified as this is readily available in the grammar in Appendix A.

6.2 Meta Notation for the Creol Type System

The notation and meta-variables used in the presentation of typing rules are presented next.

$(\dots)$ Information of which order is important is presented as tuples, that is, contained in parentheses.

$\{\dots\}$ Information of which order is not important is presented as sets, that is, contained in brackets.

$^{\text{some unique description}}$ Superscripts are used to provide additional description and intuition to aid the reader as well as distinguish different types. For instance τ^{in} and τ^{out} which tells the reader that τ^{in} and τ^{out} are distinct and reminds the reader that τ^{in} is used as an input parameter and τ^{out} is used as an output parameter.

$\stackrel{\text{def}}{=}$ The symbol $\stackrel{\text{def}}{=}$ is used to introduce shorthand notation. For instance $I \stackrel{\text{def}}{=} 1, \dots, n$ which introduces I as a shorthand for $1, \dots, n$.

$_{i\in I}$ The membership subscript is used to describe something about each element in a set. Intuitively $\tau_{i\in I} \stackrel{\text{def}}{=} \tau_1, \ldots, \tau_n$ where $I \stackrel{\text{def}}{=} 1, \ldots, n$. Each distinct upper case letter denotes a separate set $I \stackrel{\text{def}}{=} 1, \ldots, n$, $J \stackrel{\text{def}}{=} 1, \ldots, o$, $K \stackrel{\text{def}}{=} 1, \ldots, p$ and so on. Lower and upper case letters correspond so $i \in I$, $j \in J$, $k \in K$ and so on. The notation τ_I is used when the entire set is in focus, and is defined as $\tau_I \stackrel{\text{def}}{=} \tau_{i\in I}$. The lower case subscript inside a parentheses relates to the upper case subscript outside the parentheses, so

$$(\ldots\tau_i\ldots\tau_i\ldots)_I \stackrel{\text{def}}{=} (\ldots\tau_i\ldots\tau_i\ldots)_{i\in I} \stackrel{\text{def}}{=} (\ldots\tau_1\ldots\tau_1\ldots), \ldots, (\ldots\tau_n\ldots\tau_n\ldots)$$

where only items with subscript inside the parentheses are affected.

τ An arbitrary type τ.

ρ An arbitrary row ρ. Rows are used for describing object types.

α A free type α. Free types are used for parametrisation.

$\dot{\tau}$ A placeholder type $\dot{\tau}$. Placeholder types are introduced by the type environment Θ, and are fixpoints for recursive types.

σ A set of substitutions. That is

$$\sigma \stackrel{\text{def}}{=} \left\{\left(\tau_i^{\text{old}} \mapsto \tau_i^{\text{new}}\right)_{i\in I}\right\}$$

so

$$\tau_\sigma \stackrel{\text{def}}{=} \left(\ldots(\tau)_{\left(\tau_1^{\text{old}}\mapsto\tau_1^{\text{new}}\right)}\cdots\right)_{\left(\tau_n^{\text{old}}\mapsto\tau_n^{\text{new}}\right)}.$$

Θ A type environment Θ is defined by a set of placeholder types $\dot{\tau}$ and types τ, that is

$$\Theta \stackrel{\text{def}}{=} \left\{(\dot{\tau}_i, \tau_i)_{i\in I}\right\}$$

so one can infer that $\Theta \vdash \dot{\tau} : \tau$ when $(\dot{\tau}, \tau) \in \Theta$, where τ may contain placeholder types. The judgement $\Theta \vdash \dot{\tau} : \tau$ is called partial expansion. Placeholder types can be removed entirely by repeating partial expansion until all placeholder types have been removed, which is called full expansion. Full expansion does, however, not terminate, for recursive types. Full expansion of $\dot{\tau}$ to a, potentially infinite, type is written $\Theta(\dot{\tau})$, where Θ is considered a function from placeholder types to types. Intuitively the type environment Θ provides the necessary indirection so recursion can be described. The formal foundation of this intuition is in Section 5.14. The notation $\Theta \stackrel{\text{def}}{=} \left\{(\dot{\tau}_i : \tau_i)_{i\in I}\right\}$ can be used for convenience. When superscripts and subscripts are used, the type $\tau^{\text{superscript}}_{\text{subscript}}$ has the placeholder type $\dot{\tau}^{\text{superscript}}_{\text{subscript}}$ which is obtained by adding the symbol $\dot{}$ to the type. Note that in the presence of Θ, a placeholder type $\dot{\tau}$ may be used wherever a type is expected.

Γ An identifier environment Γ connects identifiers to types or placeholder types, that is

$$\Gamma \stackrel{\text{def}}{=} \left\{(\mathtt{id}_i, \tau_i)_{i\in I}\right\}$$

such that $\Gamma \vdash \texttt{id} : \tau$ when $(\texttt{id}, \tau) \in \Gamma$ and $\Theta, \Gamma \vdash \texttt{id} : \tau$. The notation $\Gamma \stackrel{\text{def}}{=} \left\{(\texttt{id}_i : \tau_i)_{i \in I}\right\}$ can be used for convenience. Note that in the presence of a type environment Θ, the type τ can also be a placeholder type.

$\ddot{=}$ An equivalence relation between types.

$\ddot{<}$ The conformance relation $\tau^{\text{sub}} \ddot{<} \tau^{\text{sup}}$ between a subtype τ^{sub} and a super type τ^{sup}. Conformance expresses both subtyping and matching.

$\diamond$ Denotes updates to Θ and Γ so $\Theta, \Gamma \vdash \cdots \diamond \Theta', \Gamma'$ represents a type judgement with an updated type Θ' and identifier Γ' environment. The updated environment can be used directly so $\Theta, \Gamma \vdash \cdots \diamond \Theta', \Gamma' \vdash \ldots$ is valid.

italics Text written in *italics* is part of the type language

`typewriter` Text written in `typewriter` is part of Creol's syntactic domain.

$\dot{\tau}^{\mathcal{F}}$ A fresh placeholder type $\dot{\tau}$ such that $\Theta(\dot{\tau})$ is undefined.

Type checking rules are written

$$\frac{\Theta', \Gamma' \vdash \ldots \quad \Theta'', \Gamma'' \vdash \ldots \quad \ldots}{\Theta, \Gamma \vdash \ldots}$$

where the judgement below the line holds if all the judgements above the line hold. Whenever there are environment changes the rules must be applied in an order that respects environment updates denoted by $\diamond$. Axioms on the form

$$\Theta, \Gamma \vdash \ldots$$

are rules without conditions.

6.3 Creol Type Language

This section introduces the Creol type language. The type language is used both in the discussion about typing of Creol concepts, and in the presentation of the type checking rules. The letters $\mathcal{M}$,$\mathcal{O}$,$\mathcal{R}$,$\mathcal{V}$ and $\mathcal{L}$ are used to denote a type when the extra information provided by the full type language is not needed.

Void The type of statements with side-effect.

() The absence of any type, the tuple with no types.

Bool The type of a boolean value, `True` or `False`.

Int The type of a signed integer, unspecified precision.

Str The type of a string of arbitrary length.

ref τ The type of a reference to a value of type τ.

$\tau^{\text{in}} \to \tau^{\text{out}}$ The type of a function from argument of type τ^{in} to a result of type τ^{out}.

$\left(\tau_I^{\text{in}}\right) \to \tau^{\text{out}}$ Shorthand for $\left(\tau_1^{\text{in}}, \ldots, \tau_n^{\text{in}}\right) \to \tau^{\text{out}}$ which is the type of a function where the in parameter is a tuple of several arguments.

$\forall\alpha(\tau)$ Unconstrained parametrisation. The type τ has a free type α.

$\forall\alpha_I(\tau)$ Shorthand for $\forall\alpha_1(\ldots(\forall\alpha_n(\tau))\ldots)$.

$\forall\alpha \ddot{<} \tau^{\text{sup}}(\tau)$ Constrained parametrisation. The type τ has a free type α. The free type α is restricted by the conformance relation $\ddot{<}$, so α must conform to τ^{sup}.

$\forall\alpha_I \ddot{<} \tau_I^{\text{sup}}(\tau)$ Shorthand for $\forall\alpha_1 \ddot{<} \tau_1^{\text{sup}}(\ldots(\forall\alpha_n \ddot{<} \tau_n^{\text{sup}}(\tau))\ldots)$.

$\tau_{\tau^{\text{old}} \mapsto \tau^{\text{new}}}$ A substitution where all occurrences of τ^{old} in τ are replaced by τ^{new}.

$\tau_{\tau_I^{\text{old}} \mapsto \tau_I^{\text{new}}}$ Shorthand for $(\ldots(\tau_{\tau_1^{\text{old}} \mapsto \tau_1^{\text{new}}})\ldots)_{\tau_n^{\text{old}} \mapsto \tau_n^{\text{new}}}$.

$\mathcal{R}\left((\mathtt{id}, \tau)_I\right)$ A record type $\mathcal{R}\left((\mathtt{id}_1, \tau_1), \ldots, (\mathtt{id}_n, \tau_n)\right)$ where a record contains values of types $\tau_1, \ldots, \tau_n$ which can be extracted by names $\mathtt{id}_1, \ldots, \mathtt{id}_n$ and the value extracted by $\mathtt{id}_i$ has type τ_i and so on. The order of information in the tuple is important for creation of tuples. The notation $\mathcal{R}\left((\mathtt{id} : \tau)_I\right)$ can be used for convenience.

$\mathcal{V}\left\{(\mathtt{id}, \tau)_I\right\}$ A variant type describes a set of names $\mathtt{id}_{i \in I}$ and types $\tau_{i \in I}$. A value of a variant type contains a name and a value. The name $\mathtt{id}_i$ identifies the type τ_i of the contained value. Each of $\mathtt{id}_{i \in I}$ also corresponds to constructors such as

$$\mathtt{id}_i : \tau_i \to \mathcal{V}\left((\mathtt{id}, \tau)_I\right)$$

where a constructor is the special name for a function that returns a variant type. The order of information in a variant type is not important. The notation $\mathcal{V}\left\{(\mathtt{id} : \tau)_I\right\}$ can be used for convenience.

$\mathcal{L}\left(\tau_I^{\text{out}}\right)$ A label for the asynchronous reply to an asynchronous method invocation. The reply contains out parameters with the types $\tau_1^{\text{out}}, \ldots, \tau_n^{\text{out}}$, where the order of the types matters.

$\bot(s)$ The error type, which may contain an error message s.

$\top$ Any type, used to unify with anything or to replace an error type $\bot$. The error type $\bot(s)$ is replaced by $\top$ when the error s is reported to the user. This prevents error messages from being reported several times.

$\mathcal{M}(\tau^{\text{co}}, \tau_I^{\text{in}}, \tau_J^{\text{out}})$ A method signature with a cointerface requirement τ^{co}, in parameters $\tau_1^{\text{in}}, \ldots, \tau_n^{\text{in}}$ and out parameters $\tau_1^{\text{out}}, \ldots, \tau_m^{\text{out}}$.

$\mathcal{O}\{\rho\}$ An object type with a free row ρ. The order of the rows does not matter. The different rows that can be used for description of an object type are:

tag $\mathtt{id} : \tau$ A row that binds the identifier $\mathtt{id}$ to the type τ. The identifier $\mathtt{id}$ must be unique with respect to the other rows in the same object type. The tag is used to distinguish different bindings. Subscripts for rows are extended to allow selection based on tags, so $\rho_{\text{prefix } i \in I}$ only selects those rows whose tags satisfy the prefix, which is a boolean expression on tags. The possible tags are:

method The name `id` denotes method τ.

personal The name `id` denotes method τ that is only available to the instance itself.

virtual The name `id` denotes a virtual method τ. A virtual method can not be invoked.

variable The name `id` denotes an instance variable of type τ.

inherit Inheritance from class whose type of `self` is τ. Note that nominal rows are not inherited.

super The `self` type τ of the super class is available by the name `id`. This is used to check static binding and to gain access to methods that were otherwise hidden by overriding.

implement τ Nominal inheritance from a type τ, where τ is a valid object type. Note that nominal inheritance only includes nominal rows.

nominal `id` A nominal row. Nominal rows represent the fulfillment or requirement of a nominal restriction `id`.

ρ A free row variable. An object type with any unbound rows is open. An object without any unbound rows is closed. A free row ρ may be replaced by a set ρ_I of rows, as long as, at most one of the rows in ρ_I are free. Sets of rows are always flattened so when ρ in $\{\rho\}$ is replaced by ρ_I, written $\{\rho\}_{\rho \mapsto \rho_I}$, the result is $\{\rho_I\}$ and not $\{\{\rho_I\}\}$.

6.4 Object-Oriented Expressions and Statements

This section investigate how object-orientation appears in expressions and statements, and assumes that the rules in Appendix B are introduced earlier.

The rule for object creation generates objects from classes. Object creation is an expression that creates an object according to the type found in Γ^{new}. Method invocations in Creol are either asynchronous or synchronous, and either use dynamic or static binding, and each case is covered by a separate rule. The synchronous method invocation rule is uses the asynchronous method invocation rules, to simplify the presentation. The unconditional release and the conditional release requires `this` to be defined, to prevent functional code outside objects from using release points. Note that it would be just as natural to regard `wait` and `await` as procedures, which is introduced into the scope by the class declaration, however, that would complicate the class declaration rule further, so for simplicity the presence of `this` is checked.

Object Creation

$$\frac{\begin{array}{c}\Theta, \Gamma^{\text{new}} \vdash \texttt{id} : \tau_I^{\text{in}} \to \dot{\tau} \\ \Theta \vdash \dot{\tau} : \mathcal{O}\{\rho_I\} \\ \rho_{\mathsf{virtual}\ i \in I} = \emptyset \end{array} \qquad \left(\begin{array}{c} \Theta, \Gamma \vdash \texttt{E}_i : \tau_i^E \\ \Theta, \Gamma \vdash \tau_i^E \ddot{<} \tau_i^{\text{in}} \end{array} \right)_{i \in I}}{\Theta, \Gamma \vdash \texttt{new id(E}_I\texttt{)} : \dot{\tau}}$$

Objects can only be created if there are no virtual methods. The object creation requires initial values $\texttt{E}_I$ which must conform to τ_I^{in}.

Asynchronous Dynamic Invocation

$$\frac{\Theta, \Gamma \vdash \mathtt{e} : \mathcal{O}\left\{\rho_K\right\} \quad \begin{array}{c} \Theta, \{\rho_{\mathsf{method} \vee \mathsf{personal} \vee \mathsf{virtual}\ k \in K}\} \vdash \mathtt{m} : \mathcal{M}\left(\tau^{\mathrm{co}}, \tau_I^{\mathrm{in}}, \tau_J^{\mathrm{out}}\right) \\ \Theta, \Gamma \vdash \mathtt{this} : \tau^{\mathrm{self}}, \tau^{\mathrm{self}} \ddot{<} \tau^{\mathrm{co}} \\ \left(\Theta, \Gamma \vdash \mathtt{v}_i : \tau_i^{\mathrm{v}}, \tau_i^{\mathrm{v}} \ddot{<} \tau_i^{\mathrm{in}}\right)_{i \in I} \end{array}}{\Theta, \Gamma \vdash \mathtt{id!e.m(v}_I\mathtt{)} : \mathit{Void} \diamond \Theta, \Gamma \cup \left\{\mathtt{id} : \mathcal{L}\left(\tau_I^{\mathrm{out}}\right)\right\}}$$

where `this` is the object type of the caller, and

$$\left\{\rho_{\mathsf{method} \vee \mathsf{personal} \vee \mathsf{virtual}\ k \in K}\right\}$$

is a namespace built from the method , personal and virtual rows, so the method is looked up amongst the rows in the callee. The subscript prefix method $\vee$ personal $\vee$ virtual selects those rows that are method, personal or virtual. The personal tag is only important when deriving object types and an object can not be created before all virtual methods have an implementation. The cointerface requirement is checked against the type of the caller, and the types of the arguments are checked against the required input parameters of the method. Notice that the label is inserted into the namespace along with information about the out types, such that this can be used both to check if a reply has arrived and to type check the receive statement.

Asynchronous Static Invocation

$$\frac{\begin{array}{c} \Theta, \Gamma \vdash \mathtt{this} : \dot{\tau}^{\mathrm{self}} \\ \Theta, \Gamma \vdash \dot{\tau}^{\mathrm{self}} : \mathcal{O}\left\{\rho_K\right\} \\ \Theta, \{\rho_{\mathsf{super}\ k \in K}\} \vdash \mathtt{C} : \mathcal{O}\left\{\rho_L\right\} \end{array} \quad \begin{array}{c} \Theta, \{\rho_{\mathsf{method} \vee \mathsf{personal}\ l \in L}\} \vdash \mathtt{m} : \mathcal{M}\left(\tau^{\mathrm{co}}, \tau_I^{\mathrm{in}}, \tau_J^{\mathrm{out}}\right) \\ \Theta, \Gamma \vdash \dot{\tau}^{\mathrm{self}} \ddot{<} \tau^{\mathrm{co}} \\ \left(\Theta, \Gamma \vdash \mathtt{v}_i : \tau_i^{\mathrm{v}}, \tau_i^{\mathrm{v}} \ddot{<} \tau_i^{\mathrm{in}}\right)_{i \in I} \end{array}}{\Theta, \Gamma \vdash \mathtt{id!m@C(v}_I\mathtt{)} : \mathit{Void} \diamond \Theta, \Gamma \cup \left\{\mathtt{id} : \mathcal{L}\left(\tau_I^{\mathrm{in}}\right)\right\}}$$

where

$$\left\{\rho_{\mathsf{super}\ k \in K}\right\}$$

is a namespace built by only considering the super rows, which are found in the type of `self` for this class, such that the type of the super class can be identified. Static binding prevents later method override from affecting the call-site. The type checking rule is quite similar to asynchronous method invocation, however the lookup is special, as static binding can only bind to methods in the current class or a super-class of the current class. Note that static invocation is unsafe for virtual methods, because an implementation is not provided at the level in the inheritance hierarchy where the search starts if the method is virtual.

Receive Check
$$\frac{\Theta, \Gamma \vdash \mathtt{id} : \mathcal{L}}{\Theta, \Gamma \vdash \mathtt{id?} : \mathit{Bool}}$$

The receive check decides if a reply is available, thus it extracts a Boolean value from a label $\mathcal{L}$.

Asynchronous Receive

$$\frac{\Theta, \Gamma \vdash \mathtt{id} : \mathcal{L}\left(\tau_I\right) \quad \left(\Theta, \Gamma \vdash \mathtt{out}_i : \tau_i^{\mathrm{out}}, \tau_i \ddot{<} \tau_i^{\mathrm{out}}\right)_{i \in I}}{\Theta, \Gamma \vdash \mathtt{id?(out}_I\mathtt{)} : \mathit{Void}}$$

An asynchronous receive looks up a label in the namespace and checks that the received arguments τ_I fit in the denoted receive variables $\texttt{out}_I$. Each returned argument must be subtype of the variable where it will be stored.

Synchronous Dynamic Invocation

$$\frac{\Theta, \Gamma \vdash \texttt{id!e.m(in}_I\texttt{);id?(out}_J\texttt{)}}{\Theta, \Gamma \vdash \texttt{e.m(in}_I\texttt{;out}_J\texttt{)} : \mathit{Void}}$$

where `id` is a generated unique identifier.

For type checking purposes a synchronous invocation is just translated to an asynchronous invocation, then the type checking rules for asynchronous invocation are used.

Synchronous Static Invocation

$$\frac{\Theta, \Gamma \vdash \texttt{id!m@C(in}_I\texttt{);id?(out}_J\texttt{)}}{\Theta, \Gamma \vdash \texttt{m@C(in}_I\texttt{;out}_J\texttt{)} : \mathit{Void}}$$

where `id` is a generated unique identifier.

For type checking purposes a synchronous invocation is just translated to an asynchronous invocation.

Release Point $\dfrac{\Theta, \Gamma \vdash \texttt{this} : \tau}{\Theta, \Gamma \vdash \texttt{wait} : \mathit{Void}}$

A release point is only a side-effect but can only be used in a class, which checked is by the presence of `this` in the environment.

Conditional Release Point $\dfrac{\Theta, \Gamma \vdash \texttt{e} : \mathit{Bool}, \texttt{this} : \tau}{\Theta, \Gamma \vdash \texttt{await e} : \mathit{Void}}$

A conditional release point requires a boolean expression to decide when execution may be resumed and it requires `this` since it can only be used in a class. Note that no special handling is necessary for labels, since a label check produces a boolean expression.

6.5 Object-Oriented Declarations

This section presents the rules for declaration of interfaces and classes.

Recall from Section 5.1, that a class or interface is fully described by the object types corresponding to all possible perspectives on objects of that class or interface. A class `A` is described by $\dot{\tau}^{\text{A self}}$, $\dot{\tau}^{\text{A Self}}$, $\dot{\tau}^{\text{A closed}}$, $\dot{\tau}^{\text{A open}}$ and $\dot{\tau}^{\text{A new}}$ while an interface `B` is described by $\dot{\tau}^{\text{B self}}$, $\dot{\tau}^{\text{B closed}}$ and $\dot{\tau}^{\text{B open}}$. The type of `self` is reveals the most information, so $\dot{\tau}^{\text{A Self}}$, $\dot{\tau}^{\text{A closed}}$, $\dot{\tau}^{\text{A open}}$ and $\dot{\tau}^{\text{A new}}$ are mechanically built from $\dot{\tau}^{\text{A self}}$, and $\dot{\tau}^{\text{B closed}}$ and $\dot{\tau}^{\text{B open}}$ are built from $\dot{\tau}^{\text{B self}}$. We only need to establish the types of `self` for classes or interfaces, as the others are derived. All these are object, types that are described by rows, that is, $\mathcal{O}\{\rho\}$. Thus, an object oriented declaration is a specification of rows

for the type of `self`. The manner of deriving the other object types is treated later in Section 6.8. The manner of checking and expanding inheritance is treated later in Section 6.7. Recall from Section 5.4, that there are actually three namespaces, the regular Γ used for all identifiers except those used for object creation, which are in Γ^{new} and those used for inheritance which are in Γ^{inherit}. To simplify the rules, the environments Γ^{new} and Γ^{inherit} are implicitly contained in Γ, and only used and updated when explicitly stated. The Γ^{inherit} namespace records both the type of `self` and `Self`, because both are expanded on inheritance.

The rule for interface specifies, how the type of `self` is built, while the other object types, that describe the interface are unspecified. Interfaces introduce nominal rows, which where discussed in Section 5.7. The type of `self` and `Self` for information from the super class must be updated to the subclass through a change of fixpoint. This is represented as a substitution of fixpoints for the implement row. The substitution for change of fixpoint is part of the type language, and is not done before it is verified that the inheritance hierarchy is not cyclic. The method signatures are checked in a separate rule, which checks that all the programmer provided types are defined, and produces a virtual row, because a method signature may not be invoked, it has no body.

Interface

$$\frac{\begin{array}{c}\left(\Theta, \Gamma^{\text{inherit}} \vdash \mathtt{I}_i : \left(\dot{\tau}_i^{\mathrm{I}\ \text{self}}, \dot{\tau}_i^{\mathrm{I}\ \text{Self}}\right)\right)_{i \in I} \\ \left(\Theta', \Gamma' \vdash \mathtt{S}_j : \rho_j\right)_{j \in J}\end{array}}{\Theta, \Gamma \vdash \begin{array}{l}\mathtt{interface\ Id} \\ \left(\mathtt{inherit\ I}_i\right)_{i \in I} \\ \mathtt{begin} \\ \quad \mathtt{S}_J \\ \mathtt{end}\end{array} \diamond \begin{array}{l}\Theta' \stackrel{\text{def}}{=} \Theta \cup \Theta'' \\ \Gamma' \stackrel{\text{def}}{=} \Gamma \cup \left\{\begin{array}{l}\mathtt{This} : \dot{\tau}^{\text{Id Self}} \\ \mathtt{@Id} : \dot{\tau}^{\text{Id closed}} \\ \mathtt{\#Id} : \dot{\tau}^{\text{Id open}} \\ \mathtt{Id} : \dot{\tau}^{\text{Id open}}\end{array}\right\} \\ \quad \Gamma^{\text{inherit}} \cup \left\{\mathtt{Id} : \left(\dot{\tau}^{\text{Id self}}, \dot{\tau}^{\text{Id Self}}\right)\right\}\end{array}}$$

where

$$\Theta'' \stackrel{\text{def}}{=} \left\{\begin{array}{l}\dot{\tau}^{\text{Id self}} : \mathcal{O}\left\{\begin{array}{l}\mathsf{nominal\ Id} \\ \left(\mathsf{inherit}\ \left(\dot{\tau}_i^I\right)_\sigma\right)_{i \in I} \\ \left(\mathsf{implement}\ \left(\dot{\tau}_i^I\right)_\sigma\right)_{i \in I} \\ \rho_J \\ \rho\end{array}\right\} \\ \dot{\tau}^{\text{Id Self}} : \ldots \\ \dot{\tau}^{\text{Id closed}} : \ldots \\ \dot{\tau}^{\text{Id open}} : \ldots\end{array}\right\}$$

and $\sigma \stackrel{\text{def}}{=} \left\{\left(\dot{\tau}_i^{\mathrm{I}\ \text{self}} \mapsto \dot{\tau}^{\text{Id self}}\right), \left(\dot{\tau}_i^{\mathrm{I}\ \text{Self}} \mapsto \dot{\tau}^{\text{Id Self}}\right)\right\}$ are substitutions that update the fixpoint for the types of `self` and `Self` for inherited rows.
The type of `self` for an interface is explained throughout Section 5. The namespace Γ is enriched with a set of standard names so `@Id` is the closed type, and `#Id` is the open type. The name `Id` is by default given the open type, which is the same strategy as chosen by LOOJ [17] for minimal programmer concern for types.

Signature Row

$$\frac{\begin{array}{c}\Theta,\Gamma \vdash \texttt{Co} : \tau^{\mathrm{Co}} \\ \left(\Theta,\Gamma \vdash \texttt{TI}_i : \tau_i^{\mathrm{TI}}\right)_{i\in I} \\ \left(\Theta,\Gamma \vdash \texttt{TO}_j : \tau_j^{\mathrm{TO}}\right)_{j\in J}\end{array}}{\Theta,\Gamma \vdash \begin{array}{l}\texttt{with Co} \\ \texttt{op m (} \\ \texttt{in (}\mathtt{in}_i\texttt{:}\mathtt{TI}_i\texttt{)}_{i\in I}\texttt{ ;} \\ \texttt{out (}\mathtt{out}_j\texttt{:}\mathtt{TO}_j\texttt{)}_{j\in J}\texttt{)}\end{array} : \mathsf{virtual}\ \mathtt{m} : \mathcal{M}\left(\tau^{\mathrm{Co}}, \tau_I^{\mathrm{TI}}, \tau_J^{\mathrm{TO}}\right)}$$

A signature requires valid types for input and output parameters, as well as a valid cointerface. Notice that the signature doesn't change the environment; this is because the signature is included in the type of the interfaces in which it occurs.

The class declaration rule is a bit more complicated, than the interface declaration rule. The complications arise from initialisation parameters and inheritance from both interfaces and classes. Recall from Section 5.7 and Section 5.4, that inheritance from a class is represented by an inherit row, and that a class implements an interface is represented by an implement row.

Each kind of information specified inside a class corresponds to a different row, and these rules are factored out in row rules. The variable row rule checks, that the type of the declared variable is legal, and produces a variable row. The method row rule type checks the body of the method in an environment, that contains both input parameters and output parameters, and produces a method row that can be used for method invocation. The personal row is just like the method row, except that a personal row is not visible outside the type of `self`.

Class

$$\frac{\begin{array}{c}\left(\Theta,\Gamma \vdash \mathtt{in}_i : \tau_i^{\mathrm{TI}}\right)_{i\in I} \\ \left(\Theta,\Gamma^{\mathrm{inherit}} \vdash \texttt{C}_j : \left(\dot\tau_j^{\mathrm{C\ self}}, \dot\tau_j^{\mathrm{C\ Self}}\right)\right)_{j\in J} \\ \left(\Theta,\Gamma^{\mathrm{inherit}} \vdash \texttt{I}_k : \left(\dot\tau_k^{\mathrm{I\ self}}, \dot\tau_k^{\mathrm{I\ Self}}\right)\right)_{k\in K} \\ \left(\Theta',\Gamma' \vdash \texttt{V}_l : \rho_l\right)_{l\in L} \\ \left(\Theta',\Gamma' \cup \left\{\left(\mathtt{in}_i : \tau_i^{\mathrm{TI}}\right)_{i\in I}\right\} \vdash \texttt{R}_m : \rho_m\right)_{m\in M}\end{array}}{\Theta,\Gamma \vdash \begin{array}{l}\texttt{class Id(}\left(\mathtt{in}_i\texttt{:}\mathtt{TI}_i\right)_{i\in I}\texttt{)} \\ \left(\texttt{inherit C}_j\right)_{j\in J} \\ \left(\texttt{implement I}_k\right)_{k\in K} \\ \texttt{begin} \\ \quad \texttt{V}_L \\ \quad \texttt{R}_M \\ \texttt{end}\end{array} \diamond \begin{array}{l}\Theta' \stackrel{\mathrm{def}}{=} \Theta \cup \Theta'' \\ \Gamma' \stackrel{\mathrm{def}}{=} \Gamma \cup \left\{\begin{array}{l}\texttt{this} : \dot\tau^{\mathrm{Id\ self}} \\ \texttt{This} : \dot\tau^{\mathrm{Id\ Self}} \\ \texttt{@Id} : \dot\tau^{\mathrm{Id\ closed}} \\ \texttt{\#Id} : \dot\tau^{\mathrm{Id\ open}} \\ \texttt{Id} : \dot\tau^{\mathrm{Id\ open}}\end{array}\right\} \\ \Gamma^{\mathrm{inherit}} \cup \left\{\texttt{Id} : \left(\dot\tau^{\mathrm{Id\ self}}, \dot\tau^{\mathrm{Id\ Self}}\right)\right\} \\ \Gamma^{\mathrm{new}} \cup \left\{\texttt{Id} : \tau_I^{\mathrm{TI}} \to \dot\tau^{\mathrm{Id\ new}}\right\}\end{array}}$$

where

$$\Theta'' \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^{\text{Id self}} : \mathcal{O} \left\{ \begin{array}{l} \mathsf{super}\ \mathtt{Id} : \dot{\tau}^{\text{Id self}} \\ \left(\mathsf{inherit}\ \mathtt{C}_j : \left(\dot{\tau}_j^C\right)_{\sigma^C}\right)_{j \in J} \\ \left(\mathsf{implement}\ \left(\dot{\tau}_k^I\right)_{\sigma^I}\right)_{k \in I} \\ \rho_L \\ \rho_M \\ \rho \end{array} \right\} \\ \dot{\tau}^{\text{Id Self}} : \ldots \\ \dot{\tau}^{\text{Id closed}} : \ldots \\ \dot{\tau}^{\text{Id open}} : \ldots \\ \dot{\tau}^{\text{Id new}} : \ldots \end{array} \right\}$$

and

$$\sigma^C \stackrel{\text{def}}{=} \left\{ \left(\dot{\tau}_j^{\text{C self}} \mapsto \dot{\tau}^{\text{Id self}}\right), \left(\dot{\tau}_j^{\text{C Self}} \mapsto \dot{\tau}^{\text{Id Self}}\right) \right\}$$

and

$$\sigma^I \stackrel{\text{def}}{=} \left\{ \left(\dot{\tau}_k^{\text{I self}} \mapsto \dot{\tau}^{\text{Id self}}\right), \left(\dot{\tau}_k^{\text{I Self}} \mapsto \dot{\tau}^{\text{Id Self}}\right) \right\}$$

which are substitutions, that update the fixpoint for the types of `self` and `Self` for inherited rows.
The explanations for each condition to the rule are:

- The initialisation parameters are checked to be valid types.
- The placeholder types for the super classes are looked up.
- The placeholder types for the super interfaces are looked up.
- The rows are looked up in an environment where all the object types introduced by the class declaration are available, but the instance variables are not available, to avoid non-deterministic initialisation due to the unknown order of initialisation for instance s.
- The method rows are looked up in an environment with both all object types produced by the class and all instance variables.

The type of `self` for a class is described throughout Section 5.

Notice that the type $\dot{\tau}^{\text{Id new}}$ is different from $\dot{\tau}^{\text{Id closed}}$ due to the presence of virtual rows. Virtual methods are replaced by regular methods in the type of objects that would have been produced by this class, that is $\dot{\tau}^{\text{Id closed}}$. Virtual methods from $\dot{\tau}^{\text{Idself}}$ are kept in $\dot{\tau}^{\text{Id new}}$ such that object creation can be prevented if there are any virtual methods.

Variable Row

$$\frac{\Theta, \Gamma \vdash \mathtt{T} : \tau}{\Theta, \Gamma \vdash \mathtt{var\ id{:}T\ ;} : \mathsf{variable}\ \mathtt{id} : \tau}$$

The variable row rule checks the validity of the type of an instance variable declaration and creates a variable row that is used by the surrounding class.

Method Row

$$\frac{\begin{array}{c}\Theta,\Gamma \vdash \texttt{Co} : \tau^{\mathrm{Co}} \\ \left(\Theta,\Gamma \vdash \texttt{TI}_i : \tau_i^{\mathrm{TI}}\right)_{i\in I} \\ \left(\Theta,\Gamma \vdash \texttt{TO}_j : \tau_j^{\mathrm{TO}}\right)_{j\in J} \\ \Theta,\Gamma \cup \left\{\left(\texttt{in}_i : \tau_i^{\mathrm{TI}}\right)_{i\in I}\right\} \cup \left\{\left(\texttt{in}_j : \tau_j^{\mathrm{TO}}\right)_{j\in J}\right\} \vdash \texttt{B} : \mathit{Void}\end{array}}{\begin{array}{ll}\Theta,\Gamma \vdash & \texttt{with Co} \\ & \texttt{op m (} \\ & \texttt{in (}\texttt{in}_i\texttt{:TI}_i\texttt{)}_{i\in I}\texttt{ ;} \\ & \texttt{out (}\texttt{out}_j\texttt{:TO}_j\texttt{)}_{j\in J}\texttt{)} \\ & \texttt{ == B}\end{array} : \mathsf{method}\ \texttt{m} : \mathcal{M}\left(\tau^{\mathrm{Co}}, \tau_I^{\mathrm{TI}}, \tau_J^{\mathrm{TO}}\right)}$$

The method row is almost identical to the signature row rule, with the additional condition that the body typechecks to *Void* under an environment which includes type information for the in and out parameters of the method.

Personal Row

$$\frac{\begin{array}{c}\left(\Theta,\Gamma \vdash \texttt{TI}_i : \tau_i^{\mathrm{TI}}\right)_{i\in I} \\ \left(\Theta,\Gamma \vdash \texttt{TO}_j : \tau_j^{\mathrm{TO}}\right)_{j\in J} \\ \Theta,\Gamma \cup \left\{\left(\texttt{in}_i : \tau_i^{\mathrm{TI}}\right)_{i\in I}\right\} \cup \left\{\left(\texttt{in}_j : \tau_j^{\mathrm{TO}}\right)_{j\in J}\right\} \vdash \texttt{B} : \mathit{Void}\end{array}}{\begin{array}{ll}\Theta,\Gamma \vdash & \texttt{op m (} \\ & \texttt{in (}\texttt{in}_i\texttt{:TI}_i\texttt{)}_{i\in I}\texttt{ ;} \\ & \texttt{out (}\texttt{out}_j\texttt{:TO}_j\texttt{)}_{j\in J}\texttt{)} \\ & \texttt{ == B}\end{array} : \mathsf{method}\ \texttt{m} : \mathcal{M}\left(\top, \tau_I^{\mathrm{TI}}, \tau_J^{\mathrm{TO}}\right)}$$

The personal row rule is similar to the method row rule, but creates a personal row. The personal row rule is used when no cointerface is specified, and the default cointerface is τ^{self}, however, the type of the cointerface could just as well be $\top$, as is done throughout the presentation of the type system. The cointerface is unimportant because a personal row is not visible outside the type of self, however the type system appears simpler with only one type of method, instead of introducing a special type system construct only for methods without cointerface.

6.6 Conformance

This section provides an overview of where the rules for conformance between different types are found. Most of the rules for conformance were introduced in Section 5, such as the treatment of fixpoints, variance by source and sink analysis, decidability due to parametrisation, instance variables methods, virtual methods, and nominal rows.

The rules for conformance between rows with different tags is only provided as examples, so we provide the rules here.

- A regular row can conform to a virtual row. This happens when a method

is provided for a virtual row.

$$\frac{\Theta, \Gamma \vdash \rho^b \ddot{<} \rho^a}{\Theta, \Gamma \vdash \mathsf{method}\ \mathtt{id} : \rho^b \ddot{<} \mathsf{virtual}\ \mathtt{id} : \rho^a}$$

- A personal row can be made public by inheritance.

$$\frac{\Theta, \Gamma \vdash \rho^b \ddot{<} \rho^a}{\Theta, \Gamma \vdash \mathsf{method}\ \mathtt{id} : \rho^b \ddot{<} \mathsf{personal}\ \mathtt{id} : \rho^a}$$

- Nominal rows require identifier equality to accept conformance.

$$\frac{\mathtt{id}^b = \mathtt{id}^a}{\Theta, \Gamma \vdash \mathsf{nominal}\ \mathtt{id}^b \ddot{<} \mathsf{nominal}\ \mathtt{id}^a}$$

- The method, personal, virtual and super rows conform when there is equality between tag and identifier and conformance holds for the type.

$$\frac{\Theta, \Gamma \vdash \rho^b \ddot{<} \rho^a}{\Theta, \Gamma \vdash \mathsf{tag}\ \mathtt{id} : \rho^b \ddot{<} \mathsf{tag}\ \mathtt{id} : \rho^a}$$

  where $\mathsf{tag} \in \{\mathsf{method}, \mathsf{personal}, \mathsf{virtual}, \mathsf{super}\}$

- Due to static binding of instance variables as described in Section 5.10, variable rows do not participate in conformance.

- The rows for inheritance are always resolved before conformance is checked, hence conformance is not defined for inherit and implement rows.

Conformance is reflexive, a type always conforms to itself.

$$\frac{\Theta, \Gamma \vdash \tau^b \ddot{=} \tau^a}{\Theta, \Gamma \vdash \tau^b \ddot{<} \tau^a}$$

Note that no conformance rules are admissible for labels, records and variants as these are not subject to conformance.

6.6.1 Subsumption

A general rule for the treatment of fixpoints when conformance reduces to subtyping was provided in Section 5.17.1, and by reflexion a closed object type is always equal to itself, however no rule specifies how rows can be forgotten for conformance between closed object types. Conformance between two closed object types is only possible when subsumption is safe, that is rows can be safely forgotten. Recall from Section 4.3 that binary methods and subsumption does not play well together. Therefore subsumption is unsafe when the object type after the subsumption has a binary method. If the type after subsumption does not have any binary methods, then subsumption does not affect binary methods. Note that it is perfectly safe to forget binary methods, by forgetting them they may no longer be used and the problem of binary methods is avoided. The problem of binary methods is more precisely the occurrence of the fixpoint in

contravariant position, that is as an in parameter to some method. However the treatment of fixpoints when conformance reduces to subtyping prevents binary methods from typechecking, therefore no extra checks are necessary. The rule for subsumption, that is conformance between closed object types therefore specifies that conformance must hold for those rows that are not forgotten, and by the treatment of fixpoints conformance does not hold when the fixpoint occurs in contravariant position. The following conformance rule allows safe subsumption between closed object types.

$$\frac{(\Theta, \Gamma \vdash \rho_j \ddot{<} \rho_i)_{\mathsf{method} \vee \mathsf{abstract} \vee \mathsf{personal}\ i,j \in I \cap J} \qquad \begin{array}{c} (\text{not free } \rho_i)_{i \in I} \\ (\text{not free } \rho_i)_{j \in J} \\ \rho_{I \setminus J} = \emptyset \end{array}}{\Theta, \Gamma \vdash \mathcal{O}\{\rho_J\} \ddot{<} \mathcal{O}\{\rho_I\}}$$

The conditions are explained as:

- No rows that appear in ρ_I that do not also appear in ρ_J, expressed by $\rho_{I \setminus J} = \emptyset$.
- No row is free in neither ρ_I nor ρ_J.
- For every method that is found in both objects $\rho_{i,j \in I \cap J}$, regardless if personal or abstract, then conformance must hold.

6.7 Inheritance

This section provides the rules for inheritance. The rules for inherit and implement assumes that fixpoint substitutions have already been performed. Recall from Section 5.16, that substitutions are part of the type language, so the rules that initially create object types are responsible for inserting the proper substitutions. These substitutions are expanded before applying the rules for inherit and implement.

The set notation for rows is now extended. The notation $I \cap J$ is defined as the indexes (i, j) of the rows in the intersection between ρ_I and ρ_J, hence $(\dots \rho_i \dots \rho_j \dots)_{i,j \in I \cap J}$ repeats $(\dots \rho_i \dots \rho_j \dots)$ with the indices of those rows that are found in both ρ_I and ρ_J. The notation $\rho_{\text{prefix } i \in I}$ is used to select only those rows in ρ_I where the tags satisfy the prefix. The prefix is a boolean expression on tags. For instance $\rho_{\mathsf{nominal}\ i \in I}$ selects those rows from ρ_I that have a nominal tag. The prefix can use negation $\neg$, logical and $\wedge$ and logical or $\vee$. For instance $\rho_{\neg \mathsf{nominal}\ i \in I}$ selects all rows from ρ_I that do not have a nominal tag.

The rule for inherit is now given.

$$\frac{\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^A : \mathcal{O}\{\rho_I\} \\ \dot{\tau}^B : \mathcal{O}\left\{ \begin{array}{l} \mathsf{inherit}\ \mathtt{A} : \dot{\tau}^A \\ \rho_J \end{array} \right\} \end{array} \right\} \qquad \begin{array}{c} \rho_{\mathsf{inherit}\ i \in I} = \emptyset \\ \rho_{\mathsf{implement}\ i \in I} = \emptyset \\ (\Theta', \Gamma \vdash \rho_j \ddot{<} \rho_i)_{\neg \mathsf{nominal}\ i,j \in I \cap J} \end{array}}{\Theta' \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^A : \mathcal{O}\{\rho_I\} \\ \dot{\tau}^B : \mathcal{O}\left\{ \begin{array}{l} \rho_{\neg \mathsf{nominal}\ i \in I \setminus J} \\ \rho_J \end{array} \right\} \end{array} \right\}}$$

The rule uses the condition

$$(\Theta', \Gamma \vdash \rho_j \ddot{<} \rho_i)_{\neg \mathsf{nominal}\ i,j \in I \cap J}$$

to check that conformance holds for those rows from ρ_I and ρ_J that are in common. Since each class introduces a super to itself for the purpose of static binding, the super row for $\dot{\tau}^A$ is already in ρ_I and is obtained by inheritance.

The rule for $\mathsf{implement}$ is now given.

$$\frac{\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^A : \mathcal{O}\,\{\rho_I\} \\ \dot{\tau}^B : \mathcal{O} \left\{ \begin{array}{c} \mathsf{implement}\ \dot{\tau}^A \\ \rho_J \end{array} \right\} \end{array} \right\} \qquad \begin{array}{c} \rho_{\mathsf{inherit}\ i \in I} = \emptyset \\ \rho_{\mathsf{implement}\ i \in I} = \emptyset \\ \rho_{\mathsf{inherit}\ j \in J} = \emptyset \\ \Theta', \Gamma \vdash \dot{\tau}^B \ddot{<} \dot{\tau}^A \end{array}}{\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^A : \mathcal{O}\,\{\rho_I\} \\ \dot{\tau}^B : \mathcal{O} \left\{ \begin{array}{c} \rho_{\mathsf{nominal}\ i \in I \setminus J} \\ \rho_J \end{array} \right\} \end{array} \right\}}$$

The condition

$$\Theta', \Gamma \vdash \dot{\tau}^B \ddot{<} \dot{\tau}^A$$

ensures that conformance holds after the transfer of nominal rows.

6.8 Deriving Object Types

This section provides the rule for deriving object types form the type of `self`.

The type of `Self` is derived by removing rows that are private to the instance. The closed type is obtained by replacing the fixpoints for `self` and `Self` with the fixpoint of the closed type. The open type is obtained by taking the rows form the closed type and add an open row variable.

$$\Theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \dot{\tau}^{\mathrm{self}} : \mathcal{O} \left\{ \begin{array}{c} \rho_I \\ \rho^{\mathrm{self}} \end{array} \right\} \\ \dot{\tau}^{\mathrm{Self}} : \mathcal{O} \left\{ \begin{array}{c} \rho_J \stackrel{\text{def}}{=} (\rho_{\neg \mathsf{variable} \wedge \neg \mathsf{personal}\ i \in I}) \\ \rho^{\mathrm{Self}} \end{array} \right\} \\ \dot{\tau}^{\mathrm{new}} : \mathcal{O} \left\{ ((\rho_J)_{\dot{\tau}^{\mathrm{self}} \mapsto \dot{\tau}^{\mathrm{closed}}})_{\dot{\tau}^{\mathrm{Self}} \mapsto \dot{\tau}^{\mathrm{closed}}} \right\} \\ \dot{\tau}^{\mathrm{closed}} : \mathcal{O} \left\{ (((\rho_J)_{\mathsf{virtual} \mapsto \mathsf{method}})_{\dot{\tau}^{\mathrm{self}} \mapsto \dot{\tau}^{\mathrm{closed}}})_{\dot{\tau}^{\mathrm{Self}} \mapsto \dot{\tau}^{\mathrm{closed}}} \right\} \\ \dot{\tau}^{\mathrm{open}} : \mathcal{O} \left\{ \begin{array}{c} (((\rho_J)_{\mathsf{virtual} \mapsto \mathsf{method}})_{\dot{\tau}^{\mathrm{self}} \mapsto \dot{\tau}^{\mathrm{closed}}})_{\dot{\tau}^{\mathrm{Self}} \mapsto \dot{\tau}^{\mathrm{closed}}} \\ \rho^{\mathrm{open}} \end{array} \right\} \end{array} \right\}$$

The special subscript prefix $\neg\mathsf{variable} \wedge \neg\mathsf{personal}$ selects those rows that satisfies the prefix, that is, are neither a $\mathsf{variable}$ nor a $\mathsf{personal}$ row. The substitution $\mathsf{virtual} \mapsto \mathsf{method}$ replaces virtual rows with regular method rows. The virtual rows are not removed from $\dot{\tau}^{\mathrm{new}}$, because the rule for object creation is only type safe when no virtual rows exist.

Chapter 7

Viability

This section provides examples that demonstrate the viability of the Functional Creol Compiler and of the Creol type system. The viability of each is presented in a separate section to reflect that the Functional Creol Compiler used implements an earlier version of the Creol type system.

7.1 Function Creol Compiler

This section provides examples that demonstrate the viability Function Creol Compiler by providing actual output from the compiler. The Functional Creol Compiler used for the compilation uses an earlier version of the Creol type system, and the code in Appendix E is currently in the middle of a rewrite to catch up with the Creol type system described in Section 4. Note that there is a gap between Section 7.1.3 and Section 7.1.4 as no advanced variance errors are demonstrated. This is because the early type system in the Functional Creol Compiler does not detect these variance errors. The present Creol type system does and after a successful rewrite of the Functional Creol Compiler the compiler will catch such variance errors.

7.1.1 Syntax Errors

This section demonstrates how the Functional Creol Compiler responds to syntax errors.

The combinator parser library proposes alternatives when a syntax error occurs. This helps the programmer correct the error, and the parser repairs the error, syntactically. The following erroneous syntax and the FCC error message demonstrates this:

```
// Code with syntax error
interface A
begin
  op m(in x:Int; out y:Int)
end

// FCC error message
?? Error      : at symbol ; at line 3, column 16 of file "syntax_error.creol"
```

```
?? Expecting  : symbol ) or symbol = or symbol out or (symbol , ...)*
?? Repaired by: deleting: symbol ; at line 3, column 16 of file "syntax_error.creol"
```

The error message proposes possible corrections, and fixes the error by removing the erroneous ; in the signature. Without this output, the programmer would have to consult documentation.

The following code also contains an error:

```
// Code with syntax error
class B begin
  op m(in x:Int out y:Int) = y := x
end

// FCC error message
?? Error      : at symbol = at line 2, column 19 of file "syntax_error2.creol"
?? Expecting  : symbol ==
?? Repaired by: deleting: symbol = at line 2, column 19 of file "syntax_error2.creol"

?? Error      : before lower case identifier y at line 2, column 21 of file "syntax_error
?? Expecting  : symbol ==
?? Repaired by: inserting: symbol ==
```

Notice that the syntax error results in two error messages, one for each correction.

These examples demonstrate that the parser combinator library used by the Functional Creol Compiler provides good programmer feedback in the presence of syntax errors as argued for in Section 1.6 and Section 3.5.

7.1.2 Simple Type Errors

This section shows how the Functional Creol Compiler responds to simple type errors.

The following code demonstrates a mismatch, where a Bool is assigned to an Int.

```
// Code with simple type error
class B begin
  op m(in x:Int out y:Int) == y := True
end

// FCC error message
Errors:
"type_error.creol"(line 2, column 30):
Assignment failure because Bool does not fit in Int
```

Another simple type error can occur if one uses new on an interface, because only classes can produce objects.

```
//Code with simple type error
interface A
begin
  op m(in x:Int out y:Int)
end
```

```
class C begin
  op create(out o:A) == o := new A()
end

// FCC error message
Errors:
"type_error2.creol"(line 6, column 34): A is not a class.
```

The definition of static type safety from Section 4.1 ensures that the necessary methods are present, in contrast to this example:

```
//Code with simple type error
interface A
begin
  op m(in x:Int out y:Int)
end
class C begin
  op use(in o:A) == o.n(1)
end

// FCC error message
Errors:
"type_error3.creol"(line 6, column 23): Could not find method n in A object.
```

7.1.3 Inheritance Error

This section demonstrates how type unsafe inheritance is handled by the Functional Creol Compiler.

Consider the following interfaces where the type of an argument is changed, such that it is no longer type safe.

```
// Super interface
interface D
begin
  op foo(in x:Int)
end

// Type unsafe redefinition of foo
interface E inherits D
begin
  op foo(in x:Bool)
end

// FCC error message
Errors:
"unsafe_inheritance.creol"(line 8, column 1):
Inheritance impossible:
interface "E" can not inherit from interface "D":
failed subtyping for method "foo" because Int does not fit in Bool
```

Notice that static type safety requires contravariance for input parameters, and there is no conformance between an *Int* and a *Bool*.

7.1.4 Inheritance with Variances

This section provides an example program with variance that can be statically type checked by the Creol type system which allows contravariant refinement to in parameters and covariant refinement to out parameters. The example is a bit large, because it requires two inheritance hierarchies.

```
// Interface hierarchy
interface F
begin
  op foo(in x:Int)
end
 interface G inherits F
begin
  op bar(in y:Bool)
end

// Classes with contravariant and covariant refinement
class H
begin
  op test(in o:G out p:F) == null
end
class I inherits H
begin
  // Contravariant refinement of o
  // Covariant refinement of p
  op test(in o:F out p:G) == null
end
```

The Functional Creol Compiler successfully type checks the code and provides the following Creol Machine Code for the classes `H` and `I`. Note that the interfaces `F` and `G` only are used in type checking and are not present in the generated code.

```
< 'H : Cl |
 Inh: nil,
 Att: no,
 Mtds:  < 'test : Mtdname |
          Latt: ('o : null), ('p : null),
          Code:  empty ;  end( 'p)
        > ,
 Ocnt: 1
>
< 'I : Cl |
 Inh: 'H,
 Att: no,
 Mtds:  < 'test : Mtdname |
          Latt: ('o : null), ('p : null),
          Code:  empty ;  end( 'p)
        >  ,
 Ocnt: 1
>
```

7.1.5 Subsumption

This section demonstrates that the Functional Creol Compiler handles subsumption. Subsumption can occur for subtypes, such as when an object is treated as one of its declared supertypes. We use the example from Section 7.1.4.

```
// Interface hierarchy
interface F
begin
end

interface G inherits F
begin
end

// Usage of subsumption
class H
begin
  // Requires subsumption to type check
  op test(in i:G out o:F) == o := i
end
```

The statement `o := i` uses subsumption to treat a `G` object as an `F` object. The code produced for class H demonstrates the successful compilation of the program.

```
< 'H : Cl |
 Inh: nil,
 Att: no,
 Mtds:  < 'test : Mtdname |
          Latt: ('i : null), ('o : null),
          Code: ('o := 'i) ;  end( 'o)
        >  ,
 Ocnt: 1
>
```

7.1.6 Recursive Types

This section demonstrates that FCC handles recursive types and mutually recursive types, as discussed in Section 5.14. Consider the two recursive and mutually recursive classes, where type checking terminates, and produces correct types for both classes.

```
// Recursion and Mutual Recursion
class M
begin
  // Mutual recursion
  op foo(in o:N) == null
end

class N
begin
  // Mutual recursion
  op bar(in p:M) == null
```

```
  // Self recursion
  op baz(in q:N) == null
end
```

The code produced for `M` and `N` by the Functional Creol Compiler for this example is now presented to demonstrate that successfull execution of the program.

```
< 'M : Cl |
 Inh: nil,
 Att: no,
 Mtds:  < 'foo : Mtdname |
          Latt: ('o : null),
          Code:  empty ;  end( nil)
        >  ,
 Ocnt: 1
>
< 'N : Cl |
 Inh: nil,
 Att: no,
 Mtds:  < 'bar : Mtdname |
          Latt: ('p : null),
          Code:  empty ;  end( nil)
        > *
        < 'baz : Mtdname |
           Latt: ('q : null),
           Code:  empty ;  end( nil)
        >  ,
 Ocnt: 1
>
```

7.1.7 Code Generator

This section demonstrates the successful compilation of a program that performs a faculty calculation. The example program is from earlier work on Creol [8, Sect. 5.1.1].

```
class Main()
begin
  var f : IFakultet
  op run ==  f := new Fakultet(5)
end
interface IFakultet
begin
        op fac(in n:Int out f:Int)
end
class Fakultet(beregn:Int) implements IFakultet
begin
  var fakultet:Int=1
  op run == fac(beregn; fakultet) .
  op fac(in n: Int out f:Int) ==
     if(n>2)
     then fac(n-1;f); f:= n*f
     else f:=n
     fi ;
end
```

The code produced for the classes is

```
< 'Main : Cl |
 Inh: nil,
 Att: ('f : null),
 Mtds:  < 'run : Mtdname |
          Latt: no,
          Code: ('f :=  new 'Fakultet( int(5))) ;  end( nil)
        >  ,
 Ocnt: 1
>
< 'Fakultet : Cl |
 Inh: nil,
 Att: ('beregn : null), ('fakultet :  int(1)),
 Mtds:  < 'run : Mtdname |
          Latt: no,
          Code: ('this . 'fac('beregn ; 'fakultet)) ;  end( nil)
        > *
        < 'fac : Mtdname |
          Latt: ('n : null), ('f : null),
          Code: (if ('n >  int(2))
                 th (('this . 'fac(('n -  int(1)) ; 'f)) ;
                     ('f := ('n * 'f))) el ('f := 'n)
                 fi) ; end( 'f)
        >  ,
 Ocnt: 1
>
```

7.2 Creol Type System

This section provides examples of code that demonstrate the expressiveness of the Creol type system as defined in Section 4. The Functional Creol Compiler will be able to handle these examples, when the transition to the latest revision of the Creol type system is completed.

7.2.1 Inheritance and Binary Methods

This section demonstrates inheritance that does not imply subtyping, by how binary methods are handled. The example requires a notion of the type of objects from this class, `This`, to be expressed, which is available in the Creol type system.

```
// Class with reusable code
class Point
begin
  var x,y:Int;
  op eq(in other:This out same:Bool) ==
     if x = other.x and y = other.y
     then same := True
     else same := False
     fi
end
```

```
// Inheritance without subtyping
class ColorPoint inherits Point
begin
  var color:Str;
  op eq(in other:This out same:Bool) ==
    if color = other.color
    then eq@Point(other;same) // Safe code reuse
    else same := False
    fi
end
```

Notice that inside `ColorPoint` it is possible and safe to invoke the `eq` method inherited from `Point`, and that is because the type of self, `This`, is kept open during inheritance, as described in Section 4.3 and Section 5.17.2.

7.2.2 Mutual Parametrisation and Refinement

This section demonstrates how the Creol type system can type check an advanced example of mutually parametrised interfaces that are refined on inheritance. The example requires both F-bounded quantification and match-bounded polymorphism, both expressed by polymorphism constrained by conformance. Intuitively the example provides refinement by inheritance that requires both parametrisation and refinement of the type of `self`.

The code is necessarily a bit complicated to demonstrate the powerful expressiveness. Intuitively there is a relationship between a client and a server, however the relationship can be refined by inheritance, such that there is a relationship between the refinements and such that the refinements are not subtypes of the original classes. This example demonstrates how a `Client` and a `Server` class that are mutually recursive can be refined to a `Peer` class that is both a client and a server. The gain of expressiveness in this example allows the `Peer` class to reuse code from both the `Client` and the `Server` class in a type safe manner. Note that both the `Server` and the `Client` class use constrained parametrisation which corresponds to the rule for type abstraction in Appendix B.4. The use of mutually constrained polymorphism between `Server` and `Class` is a relatively standard manner of achieving type safe covariant specialisation in a contravariant position for the abstract types `C` and `S`.

```
class Server[C <: #Client]
begin
  with C op download(in name:Str out file:Str) == ...
end
```

The `Server` provides a method `download` that can be called by any object that at least has the type `C`, where `C` is introduced by bounded polymorphism such that the type `C` conforms to the type `#Client`.

```
class Client[S <: #Server]
begin
  with S op display(message:Str) == ...
end
```

The `Client` provides a method `display` which can be called by any object that at least has the type `S`, where `S` is introduced by bounded polymorphism such that `S` conforms to `#Server`.

These classes can be used to create objects as in the following example code. The example code demonstrates type application as described by the rule for type application in Appendix B.3. Type application performs instantiation and is intuitively like calling a function with a type as an argument, hence `Server[#Client]` replace the `C` in the `Server` class with `#Client`. The type application for constrained polymorphism is safe when conformance holds for the arguments, which is the case since `#Client`$\ddot{<}$`#Client` trivially holds. Note that to simplify the example no code is presented to connect `Server` and `Client` objects.

```
var s:Server[#Client];
var c:Client[#Server];
s := new Server[#Client]();
c := new Client[#Server]();
```

Now consider that one wishes to write a `Peer` class that can serve as both a client and a server.

```
class Peer inherit Server[This] inherit Client[This]
begin
  with This op download(in name:Str out file:Str) == ...
  with This op display(message:Str) == ...
end
```

The class `Peer` reuses code from both `Server` and `Client` in a type safe manner. The inheritance from `Server[This]` is legal because the type of `Self` for `Peer` conforms to `#Client`, written `This`$\ddot{<}$`#Client` and `Client[This]` is legal because `This`$\ddot{<}$`#Client`. The refinement of the cointerface for the methods `download` and `display` appears covariant in a contravariant position, however variance is checked after parametrisation and both `S` and `C` are instantiated to `This` and `This`$\ddot{<}$`This`.

Now suppose one wants to create an advanced peer that can perform searches. Although the search function is only available for `SearchPeer`s the `SearchPeer` should interact with old `Peers` for uploading and downloading.

```
class SearchPeer inherit Peer
begin
  with #Peer op download(in name:Str out file:Str) == ...
  with #Peer op display(message:Str) == ...
  with This op search(in name:Str out result:Str) == ...
end
```

The `SearchPeer` remains compatible with old `Peer`s by performing a contravariant refinement of the `cointerface` for both `download` and `display`, and conformance for methods requires contravariance for cointerfaces as noted in Section 5.5. By inheritance the `This` from `Peer` is extended in `SearchPeer`, such that the conformance check on inheritance for the method download becomes

$$\mathcal{M}\,(\texttt{\#Peer}, \mathit{Str}, \mathit{Str}) \ddot{<} \mathcal{M}\,(\texttt{This}, \mathit{Str}, \mathit{Str})$$

which due to contravariance for cointerfaces requires

$$\texttt{\#Peer} \ddot{>} \texttt{This}$$

and the type of `This` is decided by the current class `SearchPeer` and is, for this particular example, similar to `#SearchPeer`, such that the cointerface requirement reduces to

$$\texttt{\#Peer} \ddot{>} \texttt{\#SearchPeer}$$

which holds because `#SearchPeer` supports the same protocol as `#Peer`, or a little more formally, that without regard for fixpoints the `#SearchPeer` has at least the same methods as `#Peer`.

Note that even though the inheritance is type safe neither `Peer` nor `SearchPeer` conform to either `Server[#Client]` or `Client[#Server]` because of the covariant change in contravariant position for the cointerface of `download` and `display`, hence inheritance is separate from subtyping.

This little example demonstrates that static type checking can allow quite expressive programs, however without a general insight into type theory it can not be expected that ordinary programmers can wield this expressiveness without the help of a compiler.

Chapter 8

Further Work and Research

This section presents suggestions to further work and research on both the Creol language and the Creol type system. The presentation starts with those topics that appear straightforward to formalise and implement, to those that require further research to formalise and where the implementation is unclear until that research is available. Most of these suggestions are concerned either, with an increase in language expressiveness, while retaining static type safety, or, with a strengthening of static type safety, by catching more potential errors at compile-time.

Further work on the prototype implementation of the Functional Creol Compiler is discussed in Appendix D and is not discussed in this section. Note especially Appendix D.1.2 which specifies where the Functional Creol Compiler is lacking with respect to the type system described in Section 4.

8.1 Null Pointers and Type Safety

This section provides a brief discussion on null pointers and suggests a strategy for static type checking that prevents run-time null pointer errors. Type safety is increased by catching potential null pointer errors at compile time.

Null pointer can be considered both a static typing error and a run-time error. The languages Java and `C#` consider null pointers a run-time error, while the languages O'Caml [73] and O'Haskell [70] consider null pointers a compile time error. Creol has followed the object-oriented trend of Java, `C#` and the research language $\mathcal{SOOL}$ [15, Sect. 13.4] that treat null-pointers as run-time errors. This choice is in conflict with the goal of verifiable programs. The usefulness of a program that is verified to uphold an invariant is somewhat lessened when there are run-time errors, especially when those errors could have been prevented. The natural conclusion would be to alter the Creol language and Creol type system such that null pointers never occur, but that may come at the expense of making the language slightly harder to use.

Many simple uses of uninitialized object references can easily be prevented, but in general such prevention requires alias analysis. Aliasing is the phenomenon that several distinct variables point to the same value, thus they are aliases. Alias analysis looks at the information flow in the program and determines possible aliases. Exact alias analysis is undecidable and even *may analy-*

sis is computationally challenging [93]. May analysis of aliases can determine if aliasing between two variables may or may not happen, however if aliasing may happen it is not known if it will happen. May alias analysis requires the source code, or some suitable representation, and is computationally expensive. Let us therefore consider simpler approaches to prevent dereferencing of null pointers.

The Cyclone research language focuses on the creation of a safe C language equivalent [48] and has resulted in a type system that differs between references that can be null and those that may never be null. This null-pointer prevention is realised through the use of type inference and lightweight programmer annotations [37]. The main goal for Cyclone is memory management. Compile time prevention of null pointer errors is merely an added benefit. The approach of annotations and distinguishing between non-null and possibly null references is taken further by the research language Nice [10], where null pointer errors are static type errors [11]. These annotation based approaches avoid dereference of null pointers by providing different types for references that may be null and references that can never be null. A similar scheme could be adopted for Creol and would allow the compiler to guarantee the absence of null pointer exceptions at run-time by requiring the user to explicitly test for null pointers as necessary when the pointer may be null.

The prevention of null-pointer errors at run-time is a significant guarantee, and this guarantee greatly supports the program analysis and verification that the Creol research group is working to achieve. The changes required would be relatively unobtrusive:

- Remove the keyword `null`.
- Declarations require initial values.
- Require `var` declarations to operate as the let construct, such that variables that are declared are always initialized at the same time.
- Introduce non-null references in addition to normal maybe null references.
- Extend conformance such that a non-null type can implicitly be treated as a maybe null reference.

The absence of null-pointer errors is not important for program analysis, but it is clearly a prerequisite towards safe programs in general.

For a somewhat longer discussion on the possible alternatives of encoding null-pointers the reader may consult Bruce [15, Sect. 14.1].

8.2 Compound Object Types

This section motivates compound object types and suggests how these can be added to the Creol type system. Compound object types are unforeseen combinations of object types governed by nominal restrictions and correspond to compound types as described by Weck and Büchi [94]. Compound object types provide increased language expressiveness while retaining static type checking. The increased expressiveness eases unforeseen use of classes, which is especially important for use of third-party code.

Consider that there are two interfaces `A` and `B` that are orthogonal. Consider that one wants a list of objects that supports both interfaces `A` and `B`. This can currently not be expressed in the Creol type system. As a workaround a new interface `AB` is created, and all the classes that implement interfaces `A` and `B` must be changed to also implement interface `AB`. This workaround does not add any new functionality, and the procedure must be repeated for all new combinations of interfaces. The workaround connects interfaces that are otherwise orthogonal. The workaround is necessary in a nominal type system where there is a correspondence between names and types. This is not the case in the Creol type system where nominal restrictions are formalised as nominal rows. The Creol language can be extended with a keyword `and` such that one can declare a compound object type `A and B`. This compound object type is explained by the following rule that combines the two object types, given that type equality holds for the intersection, that is those rows that occur in both object types.

$$\frac{\begin{array}{l}\Theta,\Gamma \vdash \texttt{A} : \mathcal{O}\{\rho_I\} \\ \Theta,\Gamma \vdash \texttt{B} : \mathcal{O}\{\rho_J\}\end{array} \qquad (\rho_i \ddot{=} \rho_j)_{i,j \in I \cap J}}{\Theta,\Gamma \vdash \texttt{A and B} : \mathcal{O}\left\{\begin{array}{l}\rho_{I \setminus J} \\ \rho_J\end{array}\right\}}$$

where $\rho_{I \setminus J}$ denotes the rows in ρ_I that are not in the intersection $\rho_{I \cap J}$.

8.3 Virtual Classes

This section motivates and suggests that the Creol language can be extended with virtual classes. A virtual class can have abstract methods and is therefore a partial implementation of a class. Since virtual classes are incomplete they can not produce objects, however they can be extended by inheritance that provides an implementation of the virtual methods. Virtual classes increase language expressiveness while retaining static type safety. The increase in expressiveness allows structural requirements to be expressed precisely.

Virtual classes provide two separate enhancements to the Creol language. First virtual classes provides a usefull abstraction such that algorithms can be specified modular to their deployment. A virtual class can provide code for the algorithm which uses virtual methods. When the algorithm is used, the programmer must provide bodies for the virtual methods, but can reuse the default algorithm. Second there is currently no manner of expressing object requirements without nominal constraints in Creol and since classes do not produce nominal constraints a virtual class can serve as an object requirement without nominal constraints.

The introduction of virtual classes requires a small addition to the syntax and the type system does not need to be extended as virtual methods are used in describing object types for interfaces.

8.4 Pattern Matching Compilation

The Functional Creol Compiler does currently not generate code for pattern matching. The work by Sestoft describes an algorithm for compiling pattern

matching in a simplistic manner [83] and can serve as a starting point for extending the Functional Creol compiler. Although the implementation of pattern matching does not affect type safety, type checking must perform compile time analysis to ensure that pattern matching always succeeds [83, Sect. 7.4].

8.5 Kind Checking

This section motivates and suggests how the Creol type system can be extended with kind checking. Intuitively kinds are to types what types are to values. Traditionally kind κ checking has been used to ensure type safe use of polymorphism [15, 77]. Kind checking is important for the confidence in the correctness of the Creol type system, and hence that static type checking is performed correctly.

Imagine a polymorphic type $\forall\alpha\,(\mathcal{O}\,\{\mathsf{variable}\ \mathtt{t} : \alpha\})$ where the free type α can be instantiated at a later point. Intuitively there are some restrictions on the instantiation of α. It is reasonable to replace α with *Int*, however it is not reasonable to replace α with another polymorphic type $\forall\alpha\,(\tau)$. This can be solved by introducing kinds. A non polymorphic type has kind $*$. A polymorphic type has kind $* \to *$, as it takes a type and produces a type. Now $\forall\alpha\,(\mathcal{O}\,\{\mathsf{variable}\ \mathtt{t} : \alpha\})$ has kind $* \to *$ and *Int* has kind $*$ and the instantiation of α to *Int* passes kind checking because $* \to *$ applied to $*$ produces $*$ which is a valid kind. However $\forall\alpha\,(\tau)$ has kind $* \to *$ which prevents it from replacing α in $\forall\alpha\,(\mathcal{O}\,\{\mathsf{variable}\ \mathtt{t} : \alpha\})$, because $* \to *$ can not be applied to $* \to *$, it is only possible to apply $* \to *$ to something of kind $*$.

Kinds can also be used ensure other properties of the type system. Recall from Section 6.3 that there are restrictions on how rows can be combined in an object type. No two rows can bind to the same name. There can be at most one free row. These properties are stated informally in the type system, but they can be formalised by kinds and powers [79, App. B], where powers are to kinds what kinds are to types.

Intuitively, given an object

$$\mathcal{O}\left\{\begin{array}{l}\mathsf{method}\ \mathtt{foo} : \tau\\ \rho\end{array}\right\}$$

the free row variable ρ can be replaced by any number of rows as long as the name `foo` is not introduced again. The name `foo` is already taken for this object type. Kinds can help to solve this by keeping track of those names which may not be used. Let $\mathrm{Taken}\,\{\mathtt{id}_I\}$ be a kind that remembers that the names $\mathtt{id}_1, \ldots, \mathtt{id}_n$ are taken. Then the row ρ from the example has kind $\mathrm{Taken}\,\{\mathtt{foo}\}$, and it is possible to device rules that prevent the insertion of a row that uses the name `foo` [68].

A formalisation of kinds along the lines of these intuitive examples can help in further formalisation of the Creol type system by providing correct use of both polymorphism and rows.

8.6 Analysis and Modularity

This section reveals a conflict of interest between compositional analysis and use of inheritance with respect to modularity. The conflict suggests that fur-

ther research on a proper combination of these goals is needed. Modularity in combination with static type checking improves language expressiveness.

Recall from Section 2.1.3 that static binding facilitates compositional invariant analysis through a manner of invoking methods in classes such that subclass overriding does not affect the call site. The benefit of static binding is realised on later inheritance. If a class is known to never be inherited from, that is, never subclassed, there is no need for static binding. The programmer must make a guess at whether the class will be extended to decide if it is necessary to insert annotations for static binding. The point of decision for method override affects modularity. It is not possible to correct the code if the programmer guesses wrong. The programmer of a base class may not know how the class can be extended.

If the programmer uses static binding under the assumption that it is never useful to override the method, and the programmer is wrong, it is not possible to remove static binding annotations. If the programmer does not use static binding where it should have been used, it is not possible to insert static binding annotations.

Static binding is beneficial when future extensions of a class can be predicted correctly. Since inheritance is a specialisation of a base class, it is natural that the choice of what to specialise is done at the time of writing the subclass. The concept of object-orientation is based around inheritance and encapsulation [85]. Inheritance facilitates code reuse and encapsulation increases modularity and a separation of concerns. The choice of which methods that may be overridden is not a concern when writing the base class, rather it is a concern when extending the base class through inheritance. It is first when writing the subclass that the requirements for specialisation, including method override, are known.

Static binding contradicts the purpose of inheritance. The language `C#` has versioning which requires the programmer to explicitly annotate methods in the subclass with either `new` or `override` when the method name is the same as a `virtual` method in the super class [19, Sect. 8.13]. Methods can not be overridden by default, however if they are flagged as `virtual` they can be overridden. Recent research on method refinement and specification of method encapsulation policies [82] also puts the decision of overriding into the subclass, but allows the super class to specify the least restrictive and several default encapsulation policies for different usage scenarios. The least restrictive policy can never be violated, which allows the programmer to protect methods from overriding. The default policy can be overridden, such that the programmer of the subclass can specify what to override and not. Encapsulation policies would allow the programmer to create a least restrictive policy for methods where it is crucial that overriding must never be done, and postpone the question of overloading for other methods to the time of inheritance.

Recall from Section 2.1.2 that the Creol language introduced separate inheritance hierarchies for classes and interfaces. Recall from Section 5.7 that inheritance between interfaces governs nominal constraints that express behavioural constraints while inheritance for classes does is not concerned with nominal constraints, such that nominal constraints are not inherited for classes. Static binding introduces behavioural concerns into class inheritance, which by separation between interfaces and classes should not be a concern for class inheritance. Static binding does this in a manner that contradicts encapsulation in combination with inheritance.

Static binding can be replaced with a notion of versions and still allow the programmer to prevent overloading from affecting code. If more flexibility is needed static binding can be replaced with encapsulation policies.

8.7 Overloading

This section considers two cases where the Creol language may benefit from overloading. Overloading denotes the possibility that one identifier can invoke different code depending on type information. Overloading relieves the programmer from inventing unnecessary names and represents an increase in language expressiveness. The combination of overloading and static type safety is however a theoretically difficult issue, hence the introduction of overloading should not be treated lightly. Further research on the Creol language will hopefully determine a sound theoretical basis for overloading in Creol, and provide overloading for both procedures and methods.

8.7.1 Procedures

Overloading provides a convenient form of programming where the same name changes meaning depending on the type. Programmers use an overloaded + operator that work on both integers and strings, while the machine instructions for adding integers are different from those for concatenation of strings. There are however problems with overloading. From a programmer's perspective the interaction between overloading and subtyping has proved difficult to apprehend [15, Sec. 2.5] and from a type theoretical perspective overloading should have a sound basis.

The Creol language does not allow overloading in general, but provides overloading for certain operators. The + operator has type $Int \to Int \to Int$ or $Str \to Str \to Str$. This overloading is implemented in the Functional Creol Compiler but is not given in the Creol type system. The reason is that there is currently no foundation for expressing overloading in the Creol language. The use of overloading in the Creol language is still a research subject.

Overloading in the Functional Creol Compiler is expressed with a notion similar to intersection types [77, Sect. 15.7], therefore the type of + in FCC can be view formally as

$$+ : (Int \to Int \to Int) \wedge (Str \to Str \to Str)$$

with the result that + can be used in an overloaded manner. The overloading of + can be resolved statically, but the addition of algebraic data types and functional programming with polymorphism of user defined functions requires a more expressive type system to handle overloading.

Overloading in the Creol language can be expressed with type classes as known from functional programming [57]. Type classes are found in Haskell [39, 57]. The work on O'Haskell [70] integrates type classes with object oriented features in a purely functional setting. The operator + can have the type class signature $+ : \tau \to \tau \to \tau$ with two instances $+ : Int \to Int \to Int$ and $+ : Str \to Str \to Str$.

Overloading can also be expressed with extensional polymorphism [33]. The expressive power offered by extensional polymorphism can facilitate almost ar-

bitrary overloading, more so than type classes. The operator + could be given the type

```
generic plus = case
  | Int → Int → Int ⇒ plus_int
  | Str → Str → Str ⇒  plus_str
```

The power of extensional polymorphism may not seem necessary for the Creol language, but since Creol is intended to facilitate both object-oriented and imperative programming, extensional polymorphism may be less confusing than having both classes and type classes in one language. This has yet to be determined.

The main difference between type-classes and extensional polymorphism is the use of an *open world* assumption or a *closed world* but *open recursion* assumption [29]. With an open world assumption, it is not known at compile time how an identifier can possibly be overloaded, that is the set of possible overloadings can be extended. In the closed world assumption, the set of possible overloadings is fixed, and may not be extended. The closed world assumption is not very usefull by itself, however, when combined with the open recursion assumption, it facilitates code optimisation with modular static type safety analysis [33]. The open recursion assumption states, that even in a closed world, extension is possible for recursive invocations.

8.7.2 Methods

Recent work on the Creol language allows methods to be overloaded on class inheritance [50]. The Creol type system does not facilitate overloading for methods, however the suggested form of overloading is expressible with extensional polymorphism [33]. Note however that Bruce strongly discourages to mix overriding and static overloading because it is has proved difficult for programmers to successfully predict which methods are executed [15, Sect. 2.5]. The approach taken with extensional polymorphism, provides static type checking and runtime selection of methods, that is, not static overloading, but rather dynamic overloading with static type checking.

Consider the following inheritance hierarchy where the method m is overloaded with different types.

```
class A
begin
  with Any op m(x:Int) == ...
end

class B inherits A
begin
  with Any op m(x:Bool) == ...
end
```

The generic function that corresponds to m in the class A is

```
generic m = case
  | 𝓜(⊤, Int, ()) ⇒ m@A
```

The generic function m is extended on inheritance in class B such that it becomes

```
generic m = case
| include m
| M (⊤, Bool, ()) ⇒  m@B
```

The invocation of method `m` is now statically type checked and bound to the correct class by virtue of extensional polymorphism.

The generic function `m` is used for type checking but does not occur in object types. This is important because there is no known algorithm for type equality between generic functions. The types of `m@A` and `m@B` are however just normal methods and can therefore be checked for equivalence.

8.8 Modules

There are no stated assumptions about neither compilation units nor modules for the Creol language [51, 55, 53, 52, 56, 54, 8]. Module systems do not directly contribute to type safety, but rather to modularity and encapsulation in a manner orthogonal to object-orientation, as well as expressiveness through higher order modules. Modules often serve as compilation units [65] and further research on the Creol language may consider formalising modules to clarify units of compilation. Modules are also natural for grouping algebraic data types and procedures as well as for hiding implementation details, that is, packing and unpacking of existential types [77, Sect. 24], although module systems usually provide more than existential types through higher ordered modules [26]. The work by Leroy [98] can serve as a good starting point for modules in Creol. Recent work on type systems has also demonstrated that modules and mixins can be described by extensible records with some modifications, although they provide a different abstraction than classes and objects [40]. The Creol type system describes objects with a variant of extensible records. Extensible records appear as a suitable type system abstraction that may also play a role for modules or mixins.

Chapter 9

Conclusion

The main results of this thesis are now reviewed and discussed.

9.1 Contributions

This section emphasises the contributions to the Creol research by this thesis. Note that, of these contributions, the representation of nominal information, as rows, in a structural type system, appears to be original, that is, not done before in any other type systems encountered. The other contributions are new with respect to Creol research. The contributions are ordered by apparent importance. The contributions address the issues raised in the introduction.

- The main contribution is a formal type system for the object-oriented Creol language.

  We have introduced a *conformance* relation that combines matching and subtyping, both with and without nominal restrictions. This is realised by introducing nominal rows for structural object types, which provides a clean and precise type system, which facilitates confidence in a faithful implementation.

- The creation of a prototype compiler for the Creol language by use of high level tools.

  The Functional Creol Compiler is a prototype compiler for the Creol language, and can serve as a platform for further compiler development and static type checking of Creol, especially so when the current Creol type system is implemented.

- An analysis of how the Creol type system can separate inheritance from subtyping.

  The Creol language was made with the intent of separating inheritance from subtyping. We have investigated the actual consequences of this separation, with respect to the type system and the representation of object types.

- The creation of an EBNF grammar for the Creol language.

The previous descriptions of the Creol language has not provided a complete syntax. The creation of a compiler required a formalisation of the syntax, which we have provided.

- Account for the syntax of, and type checking for, functional programming features in the Creol language. This includes procedures, algebraic data types, pattern matching and polymorphism.

  The Creol language has presumed the possibility of functional programming. This thesis has proposed a syntax for functional programming with Creol and provided type checking rules for these functional programming constructs.

- A discussion of possible improvements to the Creol type system.

  The suggestions for further work and research provide pointers to improvements to the Creol language and the Creol type system. The improvements are aimed at static type checking of more programs, and increased confidence in the formal Creol type system.

9.2 Critique

This thesis provides a framework for a Creol compiler and type system, which hopefully will prove to be a valuable contribution to the future development of the Creol language. However, we have not attempted to prove soundness for the Creol type system. The Creol type system is profoundly inspired by an encoding of objects as extensible records [27, 79, 80] and there are readily proofs available [80, App. 4] that could be used as a starting point for work on the Creol type system. We have shown that the Creol type system is expressable as extensible records with nominal rows. The nominal rows only restrict possible subtype/matching relations, that is, preserve safety. Therefore a proof of soundness would most likely, first establish soundness for the structural type system alone, and then establish that nominal restrictions preserve type safety, by only restricting possible relations. The most difficult part of the proof, would be the integration of subtyping, because the proofs only use type equality and matching.

Although the Functional Creol Compiler is made with high level tools, the compiler is still quite complex. The special syntax of the University of Utrecht Attribute Grammar system makes the code less accessible for the common programmer, even if it represents a 65% saving, determined by comparing the count of words, lines of code and size, before and after compilation of the attribute grammar. The increased expressiveness is however beneficial, once the programmer is acquainted with the syntax.

9.3 Experience

All in all, there are a few observations that summarise acquired knowledge from the work with this thesis.

The Functional Creol Compiler is written with high-level tools, and there is a bit to learn in order to extend it, however a corresponding compiler prototype written with traditional low-level tools would, according to experience, surely be

larger and more inconvenient to develop further. The high-level tools appear as an important factor for the relatively fast creation of a prototype, so there was time to pursue the creation of a type system for Creol. However, the chosen tools would benefit from more intuitive documentation. This thesis provides quite elaborate explanations in Section 3, to further aid those who wish to alter the compiler.

Object types appear as an important and very successful abstraction for the Creol type system. The early versions of the type system did not focus on object types, but rather on class and interface types. With class and interface types the type system became very algorithmic, and it was difficult to express properties with the same rigour as with object types. For object types it is clear how to handle fixpoints, while for classes and interfaces, it is not clear what a fixpoint means. Observe from Section 5.4, that classes and interfaces actually describe many different object types, each with a distinct fixpoint, and it is unclear how one could merge these fixpoints in a structural type system. The suitability of objects as an abstraction has been very much appreciated in the work with the Creol type system. By having object types expressed precisely, it is easy to regard and account for interfaces and classes as higher level abstractions.

Extensible records with row variable polymorphism, is a simple model that accounts for objects in a natural manner, where fixpoints are expressed precisely, and one can distinguish between objects and object protocols, which is crucial to the separation of inheritance from subtyping.

With regard to nominal and structural type systems it appears that nominal type systems have lower entry points than structural type systems, however structural type systems extended with nominal constraints provides more expressiveness in a concise manner.

9.4 Related Work

We have not found published work on structural type systems, that respect nominal constraints, by introduction of nominal rows. The work by Bruce combines nominal constraints, with subtyping and matching, in a *nominal* type system [15]. The work on extensible records provides subtyping and matching, without nominal nominal constraints, in a structural type system [80, 79, 27]. Other work that tackles this problem, is usually concerned with extending a nominal type system with structural elements [77, Page 254] [65].

Bibliography

[1] Martin Abadi and Luca Cardelli. On Subtyping and Matching. *Lecture Notes in Computer Science*, 952:145–167, 1995. http://citeseer.ist.psu.edu/abadi96subtyping.html.

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., 1996. ISBN 0387947752.

[3] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In *European conference on object-oriented programming on ECOOP '87*, pages 234–242. Springer-Verlag, 1987. ISBN 0-387-18353-1 http://www.ifs.uni-linz.ac.at/~ecoop/cd/papers/0276/02760234.pdf.

[4] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 161–168. ACM Press, 1990. ISBN 0-201-52430-X http://portal.acm.org/citation.cfm?id=97966\&dl=ACM\&coll=GUIDE.

[5] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley Longman, Inc., 2000. ISBN 0-201-35752-6.

[6] ANTLR - ANother Tool for Language Recognition. http://www.antlr.org/.

[7] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[8] Marte Arnestad. En abstrakt maskin for Creol i Maude. Master's thesis, Department of Informatics, University of Oslo, November 2003. In Norwegian. Available from http://heim.ifi.uio.no/~creol.

[9] Bison - The YACC-compatible Parser Generator. http://www.gnu.org/software/bison/bison.html.

[10] Daniel Bonniot. The Nice programming language. http://nice.sourceforge.net/.

[11] Daniel Bonniot. Why programs written in Nice have less bugs. http://nice.sourceforge.net/safety.html.

[12] Viviana Bono. *Type Systems for the Object Oriented Paradigm*. PhD thesis, Università di Torino, Italy, February 1999. http://citeseer.ist.psu.edu/bono99type.html.

[13] Viviana Bono and Michele Bugliesi. Matching for the lambda calculus of objects. *Theoretical Computer Science*, 212(1–2):101–140, 1999. http://citeseer.ist.psu.edu/bono98matching.html.

[14] Kim B. Bruce. Understanding Object-Oriented Languages: Semantics and Types. Lecture Notes, December 1998. http://www.cs.princeton.edu/courses/archive/fall98/cs441/Lectures/Lec21.ps.

[15] Kim B. Bruce. *Foundations of Object-Oriented Languages*. The MIT Press, Cambridge, Massachusetts, London, England, 2002.

[16] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995. http://citeseer.ist.psu.edu/article/bruce95binary.html.

[17] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 389–413. Springer-Verlag, 2004. ISBN 3-540-22159-X.

[18] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-Driven Language Design: Statically Type-Safe Virtual Types in Object-Oriented Languages. Submitted to MFPS '99, February 1999. http://cs.williams.edu/~kim/ftp/StatVT.ps.gz.

[19] Standard ECMA-334 C# Language Specification, 2002. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[20] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995. http://citeseer.ist.psu.edu/chambers95typechecking.html.

[21] Clean Programming Language. http://www.cs.kun.nl/~clean/.

[22] William R. Cook. Interfaces and Specifications for the Smalltalk-80 collection classes. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 1–15. ACM Press, 1992. ISBN 0-201-53372-3 http://doi.acm.org/10.1145/141936.141938.

[23] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 125–135. ACM Press, 1990. ISBN 0-89791-343-4 http://doi.acm.org/10.1145/96709.96721.

[24] Creol Programming Language Project. http://www.ifi.uio.no/~creol/.

[25] Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare. *Hierarchical Program Structures*, chapter III. Academic Press, 1972. ISBN 0-12-200550-3.

[26] Derek Dreyer, Karl Crary, and Robert Harper. A Type System for Higher-Order Modules, January 2002. Principles of Programming Languages (POPL02) http://citeseer.ist.psu.edu/crary01type.html.

[27] Didier Rémy. Type Inference for Records in a Natural Extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993. ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop1.ps.gz.

[28] Johan Dovland, Einar Broch johnsen, and Olaf Owe. Verification of Concurrent Objects with Asynchronous Method Calls, 2005.

[29] Dominic Duggan and John Ophel. Open and closed scopes for constrained genericity. *Theoretical Computer Science*, 275(1-2):215–258, 2002. http://dx.doi.org/10.1016/S0304-3975(01)00129-3.

[30] Atze Dijkstra (Ed.). Implementation of Programming Languages (Lecture Notes). http://citeseer.nj.nec.com/article/dijkstra02implementation.html.

[31] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, Spring 1994. http://citeseer.ist.psu.edu/fisher94lambda.html.

[32] Flex - A fast scanner generator. http://www.gnu.org/software/flex/.

[33] Jun Furuse. *Extensional Polymorphism: Theory and Application*. PhD thesis, Denis Diderot University, 2002. http://pauillac.inria.fr/~furuse/thesis/thesis_furuse.ps.gz.

[34] Michael R. Garey and David S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Bell Laboratories, 1979. ISBN 0-7167-1045-5.

[35] Carlo Ghezzi and Mehdi Jazayeri. *Programming language concepts (3. ed.)*. John Wiley & Sons, Inc., 1997. ISBN 0-471-10426-4.

[36] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification - 2nd edition, 2000. http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.

[37] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293. ACM Press, 2002. ISBN 1-58113-463-0 http://doi.acm.org/10.1145/512529.512563.

[38] David Harel. *Algorithmics - The Spirit of Computing*. Addison-Wesley, 1992. ISBN 0-201-50401-4.

[39] Haskell Functional Programming Language. http://www.haskell.org/.

[40] Henning Makholm and Joe B. Wells. Type Inference and Principal Typings for Symmetric Record Concatenation and Mixin Modules. Technical report, Heriot-Watt University, March 2005. http://henning.makholm.net/papers/bowtini-long.pdf.

[41] P. Hudak, J. Peterson, and J. H. Fasel. A Gentle Introduction to Haskell 98. http://www.haskell.org/tutorial, June 2000.

[42] Gérard Huet. Constructive Computation Theory, October 2002. Lecture notes in Constructive Computation Theory at INRIA Rocquencourt. http://pauillac.inria.fr/~huet/CCT/.

[43] John Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[44] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000. http://citeseer.ist.psu.edu/hughes98generalising.html.

[45] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[46] Jacques Garrigue. Relaxing the Value Restriction, 2003. http://citeseer.ist.psu.edu/garrigue03relaxing.html.

[47] C. B. Jay. The FISh language definition. Technical report, School of Computing Sciences, University of Technology, Sydney, 1998. http://linus.socs.uts.edu.au/~cbj/Publications/latest_fish.ps.gz.

[48] Trevor Jim, Greg Morrisett, Dan Grossman, and Mike Hicks. Cyclone - A Safe Dialect of C. http://www.research.att.com/projects/cyclone/.

[49] Einar Broch Johnsen. Personal communication, 2004.

[50] Einar Broch Johnsen. Personal communication, April 2005.

[51] Einar Broch Johnsen and Olaf Owe. A Compositional Formalism for Object Viewpoints. In Bart Jacobs and Arend Rensink, editors, *Proceedings of the 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, pages 45–60. Kluwer Academic Publishers, March 2002.

[52] Einar Broch Johnsen and Olaf Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. In *Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE Computer Society Press, September 2004. http://www.ifi.uio.no/~einarj/Papers/johnsen04sefm.pdf.

[53] Einar Broch Johnsen and Olaf Owe. Object-Oriented Specification and Open Distributed Systems. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 137–164. Springer-Verlag, 2004.

[54] Einar Broch Johnsen and Olaf Owe. Inheritance in the Presence of Asynchronous Method Calls. In *Proc. 38th Hawaii Intl. Conf. on System Sciences (HICSS 2005)*. IEEE Computer Society Press, January 2005. http://www.ifi.uio.no/~einarj/Papers/johnsen05hicss.pdf.

[55] Einar Broch Johnsen, Olaf Owe, and Marte Arnestad. Combining Active and Reactive Behavior in Concurrent Objects. In *Proc. of the Norwegian Informatics Conference (NIK'03)*, pages 193–204. Tapir Academic Publisher, November 2003.

[56] Einar Broch Johnsen, Olaf Owe, and Eyvind W. Axelsen. A Run-Time Environment for Concurrent Objects with Asynchronous Method Calls. In *Proc. 5th International Workshop on Rewriting Logic and its Applications (WRLA'04)*, Electronic Notes in Theoretical Computer Science. Elsevier, March 2004. http://www.ifi.uio.no/~einarj/Papers/johnsen04wrla.pdf.

[57] Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming*, pages 97–136, 1995. http://citeseer.ist.psu.edu/jones95functional.html.

[58] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall Press, 1988. ISBN 0-13-110370-9.

[59] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997. http://citeseer.ist.psu.edu/kiczales97aspectoriented.html.

[60] Kimwitu++ - a tool for processing trees, designed to work with Lex and Yacc. http://site.informatik.hu-berlin.de/kimwitu++/.

[61] Harry R. Lewis and Christos H. Papadimitriou. *Elements of The Theory of computation*. Alan Apt, 1998. ISBN 0-13-262478-8.

[62] Barbara H. Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994. http://citeseer.ist.psu.edu/liskov94behavioral.html.

[63] Kenneth C. Louden. *Compiler Construction: Principles and Practice*. PWS Publishing Co., 1997. ISBN 0534939724.

[64] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3: Language definition, November 1989. http://research.compaq.com/SRC/m3defn/html/index.html.

[65] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A Nominal Theory of Objects with Dependent Types. In *Proceedings ECOOP'03*, Springer LNCS, July 2003. http://lampwww.epfl.ch/~odersky/papers/ecoop03.html.

[66] Maude. http://maude.cs.uiuc.edu/.

[67] Todd Millstein, Colin Bleckner, and Craig Chambers. Relaxed MultiJava: Balancing Extensibility and Modular Typechecking. OOPSLA '03: Conference on Object-oriented programming, systems, languages, and applications, October 2003. http://www.cs.washington.edu/research/projects/cecil/www/pubs/icfp02.html.

[68] Francesco Zappa Nardelli. Strong Static Typing and Advanced Functional Programming, March 2005. Lectures given at the Bertinoro International Spring School for graduate students in computer science. http://www.di.ens.fr/~zappa/bertinoro05.html.

[69] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A Concurrent Lambda Calculus with Futures, January 2004. http://www.ps.uni-sb.de/Papers/abstracts/lambdafut.html.

[70] Johan Nordlander, Magnus Carlsson, and Björn von Sydow. O'Haskell - An Object Oriented Extensions to the Language Haskell. http://www.cs.chalmers.se/~nordland/ohaskell/.

[71] Noweb Literate Programming Tool. http://www.eecs.harvard.edu/~nr/noweb/.

[72] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 99–115. ACM Press, 2004. ISBN 1-58113-831-9 http://citeseer.ist.psu.edu/nystrom04scalable.html.

[73] Objective Caml Programming Language. http://www.ocaml.org/.

[74] Olaf Owe. Personal communication, 2004.

[75] Olaf Owe and Isabelle Ryl. A notation for combining formal reasoning, object orientation and openness. Technical Report 278, University of Oslo, November 1999. http://www.ifi.uio.no/~adapt/RS-278-IFI.ps.

[76] Benjamin C. Pierce. Bounded quantification is undecidable. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 427–459. The MIT Press, Cambridge, MA, 1994. http://citeseer.ist.psu.edu/pierce93bounded.html.

[77] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 0-262-16209-1.

[78] James Power. Notes on Formal Language Theory and Parsing. http://www.cs.may.ie/~jpower/Courses/parsing/Index.html.

[79] Didier Rémy. Programming Objects with ML-ART, an Extension to ML with Abstract and Record Types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 321–346. Springer-Verlag, 1994. ISBN 3-540-57887-0 ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/tacs94.dvi.gz.

[80] Didier Rémy and Jerome Vouillon. Objective ML: An Effective Object-Oriented Extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998. ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/objective-ml!tapos98.ps.gz.

[81] Didier Rémy and Jérôme Vouillon. The reality of virtual types for free! Unpublished note. http://pauillac.inria.fr/~remy/work/virtual/virtual.html, October 1998.

[82] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Composable Encapsulation Policies. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 26–50. Springer-Verlag, 2004. ISBN 3-540-22159-X.

[83] Peter Sestoft. ML pattern match compilation and partial evaluation. In Glück Danvy and Thiemann, editors, *Lecture Notes in Computer Science: Partial Evaluation*, volume 1110, pages 446–464. Springer-Verlag, February 1996. http://www.dina.kvl.dk/~sestoft/papers/match.ps.gz.

[84] Programming Language Standard ML of New Jersey. http://www.smlnj.org/.

[85] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 38–45, New York, NY, 1986. ACM Press. http://citeseer.ist.psu.edu/snyder86encapsulation.html.

[86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201700735.

[87] Sun Microsystems. *Java 2 Platform Standard Edition 5.0 API Specification*. http://java.sun.com/j2se/1.5.0/docs/api/.

[88] S. Doaitse Swierstra. UU_Combinator parsing library for Haskell. http://www.cs.uu.nl/groups/ST/Software/UU_Parsing/index.html.

[89] S. Swierstra Doaitse. Combinator Parsers: From Toys to Tools. In Graham Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001. http://math.tulane.edu/~entcs/.

[90] Simon Thompson. *Miranda: The Craft of Functional Programming*. Addison Wesley, July 1995. ISBN 0-201-42279-4 http://www.cs.ukc.ac.uk/pubs/1995/353.

[91] treecc - Tree Compiler Compiler, designed to work with Flex and Bison, used in Portable.NET. http://www.southern-storm.com.au/treecc.html.

[92] University of Utrecht Haskell Compiler. http://www.cs.uu.nl/groups/ST/Center/UtrechtHaskellCompiler.

[93] Vincenzo Martena and Pierluigi San Pietro. Alias Analysis by Means of a Model Checker. *Lecture Notes in Computer Science*, 2027:3–17, 2001. http://www.elet.polimi.it/upload/sanpietr/pubs/CC01submission.pdf or http://citeseer.ist.psu.edu/martena01alias.html.

[94] Wolfgang Weck and Martin Büchi. Compound Types: Strong Typing for Architecture Composition, 1998. http://citeseer.ist.psu.edu/230137.html.

[95] Eric W. Weisstein. Presburger Arithmetic. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PresburgerArithmetic.html.

[96] Eric W. Weisstein. Richardson's Theorem. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/RichardsonsTheorem.html.

[97] Renhard Wilhelm, Dieter Maurer, and R. Wilhelm. *Compiler Design*. Addison Wesley Longman Publishing Co., Inc., 1995. ISBN 0201422905.

[98] Xavier Leroy. A Modular Module System. *Journal of Functional Programming*, 10(3):269–303, 2000. http://pauillac.inria.fr/~xleroy/publi/modular-modules-jfp.ps.gz.

[99] Hongwei Xi. Unifying object-oriented programming with typed functional programming. In *Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*, pages 117–125. ACM Press, 2002. ISBN 1-58113-458-4 http://doi.acm.org/10.1145/568173.568186.

[100] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, 1999. http://citeseer.ist.psu.edu/xi98dependent.html.

[101] Yacc - Yet Another Compiler-Compiler. http://dinosaur.compilertools.net/yacc/index.html.

Appendix A

Creol Grammar

This is a compact version of the Creol grammar. Repetition is written with an $\overline{element}$ while repetition with delimiter is written $\overline{element\ delim}$. Otherwise the standard regular expression operators are used such as | for choice and ? for optional. The grammar is structured to show valid syntax. A more elaborate description of productions along with examples can be found in Section 2.

varid ::= [a − z][a − zA − Z]∗
typeid ::= [A − Z][a − zA − Z]∗
typexpr ::= *typeid*
typdecl ::= *typeid*
comment ::= //.∗\n
| /∗[.\n]∗∗/
program ::= $\overline{(interface|class|data|procedure)}$
interface ::= `interface` *typedecl*
(`inherits` $\overline{typexpr\,\texttt{,}}$)?
`begin`
$\overline{signature}$
`end`
signature ::= (`with` *typexpr*)?
`op` *varid*((`in` $\overline{\overline{varid}\texttt{:}typexpr\,\texttt{,}}$)?(`out` $\overline{\overline{varid}\texttt{:}typexpr\,\texttt{,}}$)?)
class ::= `class` *typedecl*((*varid*,))?
(`inherits` $\overline{typexpr\,\texttt{,}}$)?
(`implements` $\overline{typexpr\,\texttt{,}}$)?
`begin`
$\overline{var}$?
$\overline{method}$?
`end`

var ::= `var` $\overline{\overline{varid(\texttt{=}expression)?}\texttt{,}\texttt{:}typexpr\,\texttt{,}}$;
method ::= *signature* == *statement*;
statement ::= (*statement*)
| *statement* (;||||[]) *statement*
| *varid* := *expression*
| ($\overline{varid\,\texttt{,}}$):=*expression*
| *expression*
| `while` *expression* `do` *statement* `od`
| (*expression*.)?*varid*(*expression*,;*varid*,)
| *varid*!(*expression*.)?*varid*(*expression*,)

| *varid*`?(`$\overline{\textit{varid}\texttt{,}}$`)`
| `await` *expression*
| `wait`
| *varid*`@`*typeid*`(`$\overline{\textit{expression}\texttt{,}}$`;`$\overline{\textit{varid}\texttt{,}}$`)`
| *varid*`!`*varid*`@`*typeid*`(`$\overline{\textit{expression}\texttt{,}}$`)`
| *var*`;`*statement*
| `return` *expression*?

int ::= `[1 − 9][0 − 9]*`
bool ::= `True|False`
str ::= `"[^"]*"`
expression ::= *varid* | `int` | *bool* | *str*
| `(` *expression* `)`
| `(-|not)` *expression*
| *expression* `([+-*/=<>]|<=|>=|!=|or|and)` *expression*
| `new` *typexpr*`(`$\overline{\textit{varid}\texttt{,}}$`)`
| *varid*`?`
| `if` *expression* `then` *statement* `else` *statement* `fi`
| *varid*`?`
| *varid*`(`$\overline{\textit{expression}\texttt{,}}$`)`
| *expression*`.`*label*
| `(`$\overline{\textit{expression}\texttt{,}}$`)`
| *expression*`.`*int*
| `[`$\overline{\textit{expression}\texttt{,}}$`]`
| `case` *expression* `of` $\overline{\textit{pattern}\ (\texttt{when}\ \textit{expression})?\ \texttt{then}\ \textit{statement}\texttt{|}}$ `fo`
| *typeid*

procedure ::= `proc` *varid*`(`$\overline{\overline{\textit{varid}\texttt{,}}\texttt{:}\textit{typexpr}}$`):`*typexpr* `==` *statement*
data ::= `data` *typedecl* `=` *typedef* `;`
typedef ::= *record*
| $\overline{\textit{variant}\texttt{|}}$ `;`
| *typexpr*
| `(`$\overline{\textit{typexpr}\texttt{,}}$`)`
record ::= `(`$\overline{\overline{\textit{varid}\texttt{,}}\texttt{:}\textit{typexpr}\texttt{,}}$`)`
variant ::= *typeid* *typedef*
| *typeid*

label ::= *varid*
pattern ::= `_`
| *varid*
| *typeid* *pattern*
| *pattern* `as` *varid*
| `(`$\overline{\textit{varid}\texttt{=}\textit{pattern}\texttt{,}}$`)`
| `[]`
| `[`*pattern*`::`*pattern*`]`
typedecl ::= *typeid*
| *typeid*`[`$\overline{\textit{typeid}\texttt{,}}$`]`
| *typeid*`[`$\overline{\overline{\textit{typeid}\texttt{,}}\texttt{<}\textit{typexpr}\texttt{,}}$`]`
typexpr ::= *typeid*
| *typeid*`[`$\overline{\textit{typexpr}\texttt{,}}$`]`
| `[`*typexpr*`]`

Appendix B

Functional Subset of Creol Type System

B.1 Expressions

A simple type constructor connects a value with a type. The rules are straightforward, and present Θ and Γ, although, for brevity, not shown when Θ and Γ are inspected.

Numbers $\Theta, \Gamma \vdash \ldots, -\texttt{1} : \mathit{Int}, \texttt{0} : \mathit{Int}, \texttt{1} : \mathit{Int}, \ldots$

Boolean $\Theta, \Gamma \vdash \texttt{True} : \mathit{Bool}, \texttt{False} : \mathit{Bool}$

String $\Theta, \Gamma \vdash \texttt{"..."} : \mathit{Str}$

The simple type constructors map primitive values to their type. Note that the `"..."` is taken to mean any string enclosed in double quotes. The parsing of a string with escaped double quotes is not elaborated, since it is straightforward to deal with.

The next step is to see how expressions are typed. The rules for unary and binary operators are presented in detail, even though the rule for function application in general could cover all the cases, given a prelude with standard definitions, which is not yet developed for Creol. Furthermore the details correspond to the Creol Virtual Machine implementation of these operators. The type rules for operators, function application, identifiers and type application are:

Unary operators $\dfrac{\Theta, \Gamma \vdash \texttt{e} : \mathit{Int}}{\Theta, \Gamma \vdash -\texttt{e} : \mathit{Int}} \quad \dfrac{\Theta, \Gamma \vdash \texttt{e} : \mathit{Bool}}{\Theta, \Gamma \vdash \texttt{not e} : \mathit{Bool}}$

Numeric and Boolean negation preserve type.

Binary operators

$$\frac{\Theta, \Gamma \vdash \texttt{e}^1 : \mathit{Int} \quad \Theta, \Gamma \vdash \texttt{e}^2 : \mathit{Int}}{\Theta, \Gamma \vdash \texttt{e}^1 \ \mathit{op} \ \texttt{e}^2 : \mathit{Int}} \quad \text{where } \mathit{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$$

$$\frac{\Theta, \Gamma \vdash \texttt{e}^1 : \mathit{Int} \quad \Theta, \Gamma \vdash \texttt{e}^2 : \mathit{Int}}{\Theta, \Gamma \vdash \texttt{e}^1 \ \mathit{op} \ \texttt{e}^2 : \mathit{Bool}} \quad \text{where } \mathit{op} \in \{\texttt{=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{!=}\}$$

$$\frac{\Theta, \Gamma \vdash \mathtt{e}^1 : \mathit{Bool} \quad \Theta, \Gamma \vdash \mathtt{e}^2 : \mathit{Bool}}{\Theta, \Gamma \vdash \mathtt{e}^1 \ \mathit{op} \ \mathtt{e}^2 : \mathit{Bool}} \quad \text{where } \mathit{op} \in \{\mathtt{and}, \mathtt{or}\}$$

The binary operators are just the common mathematical operators on integers and booleans, with straightforward rules.

Function Application

$$\frac{\Theta, \Gamma \vdash \mathtt{e} : \tau_I^{\text{in}} \to \tau \quad \left(\Theta, \Gamma \vdash \mathtt{E}_i : \tau_i^E\right)_{i \in I} \quad \left(\Theta, \Gamma \vdash \tau_i^E \ddot{<} \tau_i^{\text{in}}\right)_{i \in I}}{\Theta, \Gamma \vdash \mathtt{e(E}_I\mathtt{)} : \tau}$$

Function application is typed as the return type of the function, given that the actual parameters conform to the required function arguments.

Identifiers

$$\frac{(\mathtt{id}, \tau) \in \Gamma}{\Theta, \Gamma \vdash \mathtt{id} : \tau}$$

An identifier is legal if there exists a mapping for it in Γ.

Placeholder Type Variable

$$\frac{(\dot{\tau}, \tau) \in \Theta}{\Theta \vdash \ldots \dot{\tau} \rightsquigarrow \Theta \vdash \ldots \tau}$$

A placeholder type $\dot{\tau}$ can be replaced by the type τ, if the mapping $(\dot{\tau}, \tau)$ exists in Θ.

The identifier and placeholder type rules demonstrate, how information in the environments, Θ and Γ, are used do determine that an `identifier` or placeholder type has a specific type. Thus the identifier rule and the placeholder type rule show, how elements are moved from the left hand side of $\vdash$ to the right hand side. The identifier rule is used, whenever literal text is looked up in the namespace. All rules on the form $\Theta, \Gamma \vdash \mathtt{id} : \tau$ use the identifier rule. The placeholder type rule is used to transform placeholder types into types. It is used whenever a placeholder type occurs, so placeholder types and types can be used interchangeably, which corresponds to folding and unfolding of iso-recursive types.

B.2 Statements

The next step is to look at how expressions can form statements, and how these statements can be connected. We first consider standard statements such as assignment, condition, loop and statement combinations. Remark that standard imperative features, possibly with side effects, have type *Void*, and their side effects are type checked in the condition of the rule.

Assignment

$$\frac{\Theta, \Gamma \vdash \mathtt{id} : \tau^a \quad \Theta, \Gamma \vdash \mathtt{e} : \tau^b \quad \Theta, \Gamma \vdash \tau^b \ddot{<} \tau^a}{\Theta, \Gamma \vdash \mathtt{id := e} : \mathit{Void}}$$

Assignment is a side effect with type *Void*, and the right hand side must conform to the left hand side.

Condition Expression

$$\frac{\Theta,\Gamma \vdash \texttt{e} : \mathit{Bool} \quad \Theta,\Gamma \vdash \texttt{e}^a : \tau^a, \texttt{e}^b : \tau^b, tau^a \ddot{<} \tau, \tau^b \ddot{<} \tau}{\Theta,\Gamma \vdash \texttt{if e then e}^a \texttt{ else e}^b \texttt{ fi} : \tau}$$

The condition expression rule can be used both as a statement and as an expression. When used as a statement the τ must be *Void*. When used as an expression the two branches must have a common supertype.

Loop

$$\frac{\Theta,\Gamma \vdash \texttt{e} : \mathit{Bool}, \texttt{s} : \mathit{Void}}{\Theta,\Gamma \vdash \texttt{while e do s od} : \mathit{Void}}$$

A loop is type correct when the conditional expression is of type *Bool*, and the body has type *Void*.

Composition

$$\frac{\Theta,\Gamma \vdash \texttt{s}^1 : \mathit{Void} \quad \Theta,\Gamma \vdash \texttt{s}^2 : \mathit{Void}}{\Theta,\Gamma \vdash \texttt{s}^1 \ op \ \texttt{s}^2 : \mathit{Void}} \quad \text{where } op \in \{\texttt{;}, \texttt{|||}, \texttt{[]}\}$$

Any two *Void* statements may be composed, and then the composition also has type *Void*.

The assignment uses the identifier rule to determine the type τ^a of the variable on the left hand side of `:=`, and the necessary rules to determine type τ^b of the value on the right-hand-side of `:=`, then these types are analyzed to determine, if they are compatible $\tau^b \ddot{<} \tau^a$. If these conditions are met, then the assignment is a proper statement, and it has the type *Void*. The type *Void* is a special type with no value, because the assignment doesn't return anything. Since the value is now available through the variable, the assignment either masquerades the type of the value as the type of the variable, or instantiates the open type of the variable, to remember the type of the value. This is determined by the conformance relation. The condition and loop require that the conditional expression has type *Bool*. The statement composition contains the usual `;` that perform actions in sequence, but it also contains two new statement operators, `|||` and `[]`. The `[]` chooses arbitrarily between two statements, and `|||` performs two statements in arbitrary order.

B.3 Algebraic Datatype Expressions

This section shows how to type the algebraic datatype expressions, as proposed in Section 2.3.5. The algebraic data types in Creol are built from records and variants, with record selection and variant case for deconstruction. The other algebraic data types, tuples, enumerations, options and lists can be forged from records and variants. Pattern matching is a generalisation of record selection

and variant case, and facilitates binding, nested patterns, wildcards and guards, which complicate the type checking rule for pattern matching. Since pattern matching is a general mechanism, it naturally allows deconstruction of tuples, enumerations, options and lists. The pattern matching rule is presented piecemeal to accord for each concept.

The record selection rule allows dot-notation to extract an element from a record. The compilation of record selection can safely be translated to indexed lookup, because the label is available at compile time.

The variant case rule would be complicated by empty variants, such as enumerations in C [58, Sect. A4.2]. Therefore the handling of empty variants is considered desugaring and is done at parse time. A `case` statement requires a valid variant type, and the branches in the case expression must correspond to the variants of the variant type. To prevent run-time errors the variant case expression requires a branch for each variant, but such analysis is difficult to express in our formal type system, and hence not shown.

The rule for pattern matching is valid, if the pattern is valid, and the guard is valid, where the guard uses an environment updated with bindings from the pattern. The symbol $\odot$ is used to describe the combination of a pattern with a type, so the rules for different patterns can be specified separately.

Record Selection

$$\frac{\Theta, \Gamma \vdash \mathtt{e} : \mathcal{R}\left((\mathtt{id}_i : \tau_i)_{i \in I}\right) \quad \Theta, \Gamma \stackrel{\text{def}}{=} \left\{(\mathtt{id}_i : \tau_i)_{i \in I}\right\} \vdash \mathtt{label} : \tau}{\Theta, \Gamma \vdash \mathtt{e.label} : \tau}$$

Record selection extracts an element from a record. The type of the selection is that of the selected element. The `label` is looked up with all the record labels $(\mathtt{id}_i : \tau_i)_{i \in I}$ as namespace, and these record labels are found in the record definition.

Variant Case

$$\frac{\Theta, \Gamma \vdash \mathtt{se} : \mathcal{V}\left\{\left(\mathtt{Id}_j : \tau_j^{\text{variant}}\right)_{j \in J}\right\} \quad \left(\begin{array}{c} \Theta, \left\{\left(\mathtt{Id}_j : \tau_j^{\text{variant}}\right)_{j \in J}\right\} \vdash \mathtt{P}_i : \tau^{\text{variant}} \\ \Theta, \Gamma \cup \{\mathtt{id}_i : \tau^{\text{variant}}\} \vdash \mathtt{e}_i : \tau \end{array}\right)_{i \in I}}{\Theta, \Gamma \vdash \mathtt{case\ se}\ (\mathtt{of\ P}_i\ \mathtt{id}_i\ \mathtt{then\ e}_i)_{i \in I}\ \mathtt{end} : \tau}$$

The `case` statement facilitates deconstruction of variant types. Each branch must have the same type, so the `case` expression has the same type, regardless of the chosen branch. The `case` expression is valid, if all the branches are valid. A branch is valid, if the constructor $\mathtt{P}_i$ denotes a valid variant, written

$$\Theta, \left\{\left(\mathtt{Id}_j : \tau_j^{\text{variant}}\right)_{j \in J}\right\} \vdash \mathtt{P}_i : \tau^{\text{variant}}$$

and each branch $\mathtt{e}_i$ can be correctly type checked when $\mathtt{id}_i$ is typed by that valid variant, written

$$\Theta, \Gamma \cup \{\mathtt{id}_i : \tau^{\text{variant}}\} \vdash \mathtt{e}_i : \tau$$

and this is repeated for all the branches.

Notice that $(\mathtt{of} \ldots \mathtt{then} \ldots)_{i \in I}$ is an abbreviation for several `of ... then ...` branches.

Pattern Matching

Lists an value patterns are translated into pattern matching by desugaring.

Case

$$\frac{\Theta, \Gamma \vdash \texttt{se} : \tau^{\text{se}} \quad (\Theta, \Gamma \vdash \texttt{pattern}_i \odot \tau^{\text{se}} \diamond \Theta, \Gamma' \vdash \texttt{guard}_i : \mathit{Bool}, \texttt{e}_i : \tau)_{i \in I}}{\Theta, \Gamma \vdash \texttt{case se } (\texttt{of pattern}_i \texttt{ when guard}_i \texttt{ then e}_i)_{i \in I} \texttt{ end} : \tau}$$

The `case` with pattern matching and guards is simpler, than the variant `case`, because the pattern matching is specified by separate rules for $\odot$, that connect a pattern to a type. The guard of a `case` expression must be a *Bool*, so the branch is chosen if the pattern matches and the guard is `True`. The guard can use the bindings introduced by the pattern through environment updates.

Wildcard $\Theta, \Gamma \vdash \texttt{\_} \odot \tau \diamond \Theta, \Gamma$

The wildcard symbol `_` pattern matches against any type and returns the environment unchanged.

Binding $\Theta, \Gamma \vdash \texttt{id} \odot \tau \diamond \Theta, \Gamma \cup \{\texttt{id} : \tau\}$

A pattern match with only a variable `id`, introduces `id` into the namespace, and binds it to the type τ.

As Binding

$$\frac{\Theta, \Gamma \vdash \texttt{pattern} \odot \tau \diamond \Theta', \Gamma'}{\Theta, \Gamma \vdash \texttt{pattern as id} \odot \tau \diamond \Theta', \Gamma' \cup \{\texttt{id} : \tau\}}$$

The `as` binding is appropriate, when one wishes to introduce a binding for a certain part of a pattern, while still specifying the inner structure of the pattern. Notice, that the resulting environment contains both the introduced binding, as well as other bindings, that are specified in the structure of the pattern.

Record

$$\frac{\begin{pmatrix} \Theta, \left\{(\texttt{id}_j : \tau_j)_{j \in J}\right\} \vdash \texttt{label}_i : \tau \\ \Theta, \Gamma \vdash \texttt{pattern}_i \odot \tau \diamond \Theta_i, \Gamma_i \end{pmatrix}_{i \in I}}{\Theta, \Gamma \vdash \texttt{(}(\texttt{label}_i\texttt{=pattern}_i)_{i \in I}\texttt{)} \odot \mathcal{R}\left((\texttt{id}_j : \tau_j)_{j \in J}\right) \diamond \Theta_I, \Gamma_I}$$

The record pattern matching rule is like record selection, but the pattern syntax is slightly different, as multiple labels can be specified. Each $\texttt{label}_i$ is looked up in the record definition, and its type τ is used to check the $\texttt{pattern}_i$, and the environment Θ_i, Γ_i is returned. Notice that the conclusion uses Θ_I, Γ_I which is the union of all the updated environments from the $\mathit{pattern}_i$ checking.

Variant

$$\frac{\Theta, \left\{(\texttt{Id}_i : \tau_i)_{i \in I}\right\} \vdash \texttt{Id} : \tau \quad \Theta, \Gamma \vdash \texttt{pattern} \odot \tau \diamond \Theta', \Gamma'}{\Theta, \Gamma \vdash \texttt{Id pattern} \odot \mathcal{V}\left\{(\texttt{Id}_i : \tau_i)_{i \in I}\right\} \diamond \Theta', \Gamma'}$$

The variant pattern matching rule is like the variant `case`, although slightly simpler, because the rule only checks one variant, and not several. If the variant `Id` is found in $\mathcal{V}\left\{(\texttt{Id}_i, \tau_i)_{i\in I}\right\}$, so the contained type in `Id` is τ, then the `pattern` is checked against τ.

Tuple

$$\frac{(\Theta, \Gamma \vdash \texttt{pattern}_i \odot \tau_i \diamond \Theta_i, \Gamma_i)_{i\in I}}{\Theta, \Gamma \vdash (\texttt{pattern}_{i\in I}) \odot \mathcal{R}\left((\texttt{id}_i : \tau_i)_{i\in I}\right) \diamond \Theta_I, \Gamma_I}$$

The tuple is valid, if it corresponds to a record of the same number of elements, indicated by the same subscript range, and each pattern can pattern match with the corresponding part of the record.

Unconstrained Type Application

$$\frac{\Theta, \Gamma \vdash \texttt{e} : \forall \alpha_I(\tau) \quad \left(\Theta, \Gamma \vdash \texttt{TE}_i : \tau_i^{\mathrm{TE}}\right)_{i\in I}}{\Theta, \Gamma \vdash \texttt{e}[\texttt{TE}_I] : \tau_{\alpha_I \mapsto \tau_I^{\mathrm{TE}}} \diamond \Theta', \Gamma}$$

Unconstrained type application takes types as parameters, and substitutes free types in a $\forall$ expression. Note that this rule produces an unspecified update of the type environment, due to a possible instantiation of recursive types. Θ is changed as necessary for instantiation to produce Θ'.

Constrained Type Application

$$\frac{\Theta, \Gamma \vdash \texttt{e} : \forall \left(\alpha_i \ddot{<} \tau_i^{\mathrm{sup}}\right)_{i\in I}(\tau) \quad \left(\begin{array}{c} \Theta, \Gamma \vdash \texttt{TE}_i : \tau_i^{\mathrm{TE}} \\ \Theta, \Gamma \vdash \tau_i^{\mathrm{TE}} \ddot{<} \tau_i^{\mathrm{sup}} \end{array}\right)_{i\in I}}{\Theta, \Gamma \vdash \texttt{e}[\texttt{TE}_I] : \tau_{\alpha_I \mapsto \tau_I^{\mathrm{TE}}} \diamond \Theta', \Gamma}$$

Constrained type application is similar to unconstrained, but provides an additional rule that checks conformance.

The type application rules specify how constrained and unconstrained parametrisation is handled. Both rules take a quantified expression, and replace the free types α_I with the type parameters, where the type parameters are looked up in the type environment. The rule for constrained application, additionally checks conformance restrictions. Both rules use substitution to replace the free types. Recall from Section 5.16, that substitution is part of the type language to ensure termination of inheritance checking, and that substitution needs to alter the type environment to perform instantiation on recursive types.

B.4 Declarations

Type checking of declarations is now considered. Declarations interact with the type environment Θ and namespace Γ environment. The simplest declaration is that of a variable. Note the contrast with previous rules, the syntactic part of the conclusion is not given a type. The type system does not give a type to a declaration, rather a declaration inserts a name into the namespace, or a

type into the type environment. The goal of the rules is to only allow valid declarations, not to type a declaration.

The function rule is slightly more complex, than the rule for variables, as there are more conditions. Both parameter types and return type must be valid, and the body of the function must be checked to have the return type, under the assumption that the parameters are in the namespace, and that the function itself is in the namesspace, for recursive function calls.

Variable

$$\frac{\Theta, \Gamma \vdash \texttt{T} : \tau}{\Theta, \Gamma \vdash \texttt{var id} : \texttt{T} \diamond \Theta, \Gamma \cup \{\texttt{id}, \tau\}}$$

If the programmer provided `T` denotes a valid type τ, then the identifier `id` is given type τ in Γ, which extends the namespace, denoted by the $\diamond$ notation.

Function

$$\frac{\left(\Theta, \Gamma \vdash \texttt{TI}_i : \tau_i^{\mathrm{TI}}\right)_{i \in I} \quad \Theta, \Gamma \vdash \texttt{TO} : \tau^{\mathrm{TO}} \quad \Theta, \Gamma' \cup \left\{\left(\texttt{id}_i : \tau_i^{\mathrm{TI}}\right)_{i \in I}\right\} \vdash \texttt{e} : \tau^{\mathrm{TO}}}{\Theta, \Gamma \vdash \texttt{proc f (}\left(\texttt{in}_i\texttt{:TI}_i\right)_{i \in I}\texttt{):TO = e} \diamond \Theta, \Gamma' \stackrel{\mathrm{def}}{=} \Gamma \cup \left\{\texttt{f} : \tau_I^{\mathrm{TI}} \to \tau^{\mathrm{TO}}\right\}}$$

The explanations for each condition are:

- Each declared argument must have a valid type.
- The return value of the function must be a valid type.
- The body is type checked with all the arguments available, and the body must have the declared return type. The body is checked with access to the function itself, which allows recursive calls.

The declaration of functional constructs includes records and variants, and these provide the basis for algebraic datatypes in Creol, as described in 2.3.5. Since tuples and lists are expressible with records and variants, they do not require separate type rules.

The type declarations use $\diamond$ notation to update both the type Θ and namespace Γ environment. The $\dot{\tau}^{\mathcal{F}}$ generates a fresh placeholder type. Thus, the namespace Γ is updated with the name of the type connected to the fresh placeholder type $\dot{\tau}^{\mathcal{F}}$, and $\dot{\tau}^{\mathcal{F}}$ is inserted into the type environment Θ. This indirection is unnecessary for simple examples, but allow recursive types to be expressed naturally where the placeholder type serves as a fixpoint.

The record rule stores both label and type for each contained element, for later label lookup by the selection rule. The variant rule stores label and type for each variant, and also introduces a constructor for each variant into the environment. The approach to variant types is similar to Pierce [77, Sec. 11.10] and Huet [42] for constructive computation theory. The O'Caml [73] language, which is based on constructive computation theory, only permits sum-type constructors with tuples, so the programmer must remember the position of information. To aid the programmer, a variant declaration can include an anonymous record, as suggested in Section 2.3.5, which is similar approach in the language

Haskell [39]. In an effort to keep the type declaration rules simpler, there is no separate rule for anonymous records, as this can be derived from the rules for record and variant.

The type abstraction rules show how constrained and unconstrained polymorphism is declared. The rules are used by other declaration rules when applicable, thus the type abstraction rules do not update the environment, rather they change the result type of a declaration.

Record

$$\frac{(\Theta, \Gamma \vdash \mathtt{T}_i : \tau_i)_{i \in I}}{\Theta, \Gamma \vdash \mathtt{data\ R\ =\ \{}\,(\mathtt{id}_i\mathtt{:T}_i)_{i \in I}\,\mathtt{\}} \diamond \begin{array}{l} \Theta \cup \{\dot{\tau}^{\mathcal{F}} : \mathcal{R}\left((\mathtt{id}_i : \tau_i)_{i \in I}\right)\} \\ \Gamma \cup \{\mathtt{R} : \dot{\tau}^{\mathcal{F}}\} \end{array}}$$

Each programmer provided type in the record is checked to be valid , and the environment is updated with the record type.

Variant Type

$$\frac{(\Theta, \Gamma' \vdash \mathtt{T}_i : \tau_i)_{i \in I}}{\Theta, \Gamma \vdash \mathtt{data\ S=}\,(\mathtt{Id}_i\ \mathtt{T}_i)_{i \in I} \diamond \begin{array}{l} \Theta \cup \left\{\dot{\tau}^{\mathcal{F}} : \mathcal{V}\left\{(\mathtt{Id}_i : \tau_i)_{i \in I}\right\}\right\} \\ \Gamma \cup \left\{ \begin{array}{l} \mathtt{S} : \dot{\tau}^{\mathcal{F}} \\ \left(\mathtt{v}_i : \tau_i \to \dot{\tau}^{\mathcal{F}}\right)_{i \in I} \end{array} \right\} \end{array}}$$

A variant type declaration is valid if each of the variants have a valid type. Variants are checked in an updated environment, that makes the fixpoint of the current declaration available, which allows variants to be recursive. The variant type is put into the type environment, as well as the name and type of each variant, for later typechecking purposes. The name of the variant is put into the environment as a constructor, which is a function that produces the variant type, given an argument with the correct type for that variant. Note that all variants have a type, because enumerations, which are variants without a type, are expanded to have type *Void* as part of desugaring.

Unconstrained Type Abstraction

$$\frac{\begin{array}{c}\text{simplified value restriction} \\ \Theta, \Gamma \cup \left\{(\mathtt{TV}_i : \alpha_i)_{i \in I}\right\} \vdash \ldots \mathtt{id} \ldots : \tau\end{array}}{\Theta, \Gamma \vdash \ldots \mathtt{id[TV}_I\mathtt{]} \ldots : \forall \alpha_I\,(\tau)}$$

Type abstraction can be used for a declaration, by using square brackets to hold free types. To simplify the presentation for syntactic details the rule uses ... to denote any legal surrounding type declaration. The type abstraction is legal, if the declaration is legal. The declaration is checked in an environment, where the type names $\mathtt{TV}_I$ are given free types α_I. The type abstraction poses no restrictions on the free types α_I. Note, that the rule requires a safety condition, with regard to use of references in the body. This was described in Section 5.12 and is here referred to as the simplified value restriction.

Constrained Type Abstraction

$$\frac{\begin{array}{c}\left(\Theta, \Gamma \vdash \mathtt{TS}_i : \tau_i\right)_{i \in I} \\ \Theta, \Gamma \cup \left\{\left(\mathtt{TV}_i : \tau_i\right)_{i \in I}\right\} \vdash \ldots \mathtt{id} \ldots : \tau\end{array}}{\Theta, \Gamma \vdash \ldots \mathtt{id}[\left(\mathtt{TV}_i : \mathtt{TS}_i\right)_{i \in I}] \ldots : \forall \left(\alpha_i \ddot{<} \tau_i\right)_{i \in I} (\tau)}$$

Constrained type abstraction extends unconstrained type abstraction, by checking that the constraints exists, and by checking the body under the, assumption that each abstract type has the type of the corresponding constraint, in contrast to being a free type. Conformance constraints for type application ensures that only parameters which conform to those declared are admissible.

Appendix C

Creol to CMC Comparison

This section provides the translation of the running example from Arnestad [8] and compares it with the Creol Machine Code (CMC) produced by the Functional Creol Compiler.

Consider the following Creol class that implements the faculty function:

```
class Fakultet(beregn:Int)
begin
  var fakultet:Int=1
  op run == fac(beregn; fakultet) ;
  op fac(in n: Int out f:Int) ==
    if(n>2)
        then fac(n-1;f); f:= n*f
        else f:=n
    fi ;
end
```

The encoding of this function into Creol Machine Code provided [8] by Arnestad is

```
< 'Fakultet : Cl |
  Att: ('beregn = null), ('fakultet=null),
  Ocnt: 0.0,
  Init: 'fakultet := int(1), no,
  Run: 'fac('beregn ; 'fakultet ),no,
  Mtds:
  < 'fac : Mtdname |
    Latt: ('label = null), ('caller = null), ('n = null ),
             ('f = null ),
    Code: if ('n > int(2) ) th
                  ('fac('n-int(1) ; 'f));
                  ('f := ('n * 'f));
             el ('f := 'n) fi ;
             end ( 'f )
  >
>
```

The Creol Machine Code generated by the Functional Creol Compiler for this class is:

```
< 'Fakultet : Cl |
 Inh: nil,
 Att: ('beregn : null), ('fakultet :  int(1)),
 Mtds:
 < 'run : Mtdname | Latt: no,
  Code: ('this . 'fac('beregn ; 'fakultet)) ;  end( nil)
 > *
 < 'fac : Mtdname | Latt: ('n : null), ('f : null),
  Code: (if ('n >  int(2))
         th (('this . 'fac(('n -  int(1)) ; 'f)) ;
            ('f := ('n * 'f)))
         el ('f := 'n) fi) ;
         end( 'f)
 >
 ,
 Ocnt: 1
>
```

This demonstrates that the Functional Creol Compiler does indeed generate correct code, modulo changes that where introduced in the Creol Virtual Machine specification since the Arnestad translation was valid.

Appendix D

Implementation Remarks

This section contains remarks on the implementation of the Functional Creol Compiler (FCC).

D.1 Type Checking

This section contains remarks on the implementation of the Creol type system in the Functional Creol Compiler.

D.1.1 Building Environments

The scoping rules for Creol implies that all class declarations are visible to each other. From a practical perspective this requires two passes. The first pass builds the name environments. The second pass uses the name environments for lookup of identifiers. The presence of recursive types and inheritance requires the type environment to be constructed in several passes.

Build Γ The first pass assigns placeholder types to names. All names are given a placeholder type before the type for the name is built. This is similar to the reference trick used by Appel [7].

Build Θ The second pass builds types for each placeholder type. This pass may use Γ for name lookup. Inheritance is not resolved and instantiation is not performed because that requires the complete type of both the super type and the subtype due to possibly mutual type recursion between the super class and the subclass.

Transform Θ The third pass resolves inheritance and performs instantiation while ensuring conformance. This step is performed as a transformation on Θ.

Type checking The final Θ and Γs are used to type check code.

The distinction between passes is somewhat artificial because the FCC is implemented in a non-strict language, however the notion of passes allows code in the FCC to be structured such that dependencies leading to non-termination is avoided.

D.1.2 Current State

Although the FCC implements the Creol type system, there is gap between the representation and details in the code and in the type system presentation. The FCC intends to faithfully implement the Creol type system, however there are differences. The first version of the FCC was written before the work on the type system began, and the FCC was later changed to reflect the type system. Time constraints has lead to a situation where the FCC is still catching up with the Creol type system. The following features are not yet implemented properly in the FCC:

- Abstract data types.
- Polymorphism.
- Separation of nominal constraint layer.
- Static binding.
- Interfaces as abstract classes.
- Row variables.
- Declaration as a statement.

D.1.3 Type Checking Order

The type checking in the Functional Creol Compiler applies rules in the required order. Since the Functional Creol Compiler is implemented in a non-strict language the actual order of the type checking rules depends on the program that is type checked.

D.2 Creol Language Evolution

This section explains changes to the Creol syntax as implemented in the FCC. The contributed changes are made by this thesis. The experienced changes are those resulting from other Creol research.

D.2.1 Contributed

- The keywords `in` and `out` are required for method declaration. This makes method signatures more self descriptive.
- The first letter in a constructor identifier or type is upper case. For instance the type `Bool` is written `Bool`, and the constructor `true` is written `True`.
- The process guard `await` takes an arbitrary boolean expression as parameter.
- Labels are implicitly declared when used in a send statement, and can not be declared otherwise. Labels may not be reused for different method invocations. The label is typed according to the method invocation, therefore label reuse is not type safe. A separation between the label and the return values would improve the situation in the case of repeated invocation.

- Method invocations must have trailing `()` to distinguish syntactically between method invocation and record selection.
- Lines are ended with the character `;` instead of `.` to make Creol syntax more uniform and programmer friendly. The uniformity better supports cut and paste of code.
- The character `;` is optional after variable declarations in classes and at the end of a statement. This makes Creol less picky about syntax.
- Runnable programs must have a `class Main`, similar to the main function found in other programming languages [19, 36, 86, 58]. An object of the `class Main` is created when the program is executed, thus leaving the `run()` method of `class Main` responsible for the active behaviour of the program.
- Throw away objects are admissible. Previously it was not possible to create objects without assigning it to a variable. Since Creol is a language with side effect it makes sense to create an object without keeping a reference to it, as the side effects may be the only purpose of the object.

D.2.2 Experienced

- Removal of the `init()` method in classes. Initialisation is viewed as prior to object existence.
- Introduction of initial expressions for instance variable declarations. Needed due to removal of the `init()` method.
- Overloading for function names. The introduction of overloading for functions, such as the operator `+` that works on both strings and integers.
- The introduction of multiple inheritance.

Appendix E

Code

The program code in this thesis is generated from this document with the help of the literate programming tool noweb [71].

E.1 Literate Programming

Literate programming is a technique that allows code and text to co-exist in the same document. Literate programming has been used during the work with this thesis to integrate the code for the compiler continuously into the text of the thesis, but as the work progressed it became apparent that it is more reader friendly to present the code in Appendix E. The code is still embedded with literate programming.

There exists several tools for literate programming, but due to practical considerations we have deployed Noweb [71]. In particular this means that this thesis contains the prototype Functional Creol Compiler, which it documents. Reading all the source code can however be a daunting task, and it is not expected that the casual reader is able to or interested in reading the source code. To ease the learning curve for the compiler code, a small portion of the code is used in the text to illustrate the underlying principles. A crash course for reading literate code can be found in Appendix E.2.

E.2 Reading Literate Code

The integration of compiler code into the thesis through literate programming annotates the code in a certain predetermined manner, and to efficiently follow the code the reader must understand these annotations. This is some code that is extracted from this document into the separate file named `program.code`, and this separate file can then be used by other tools, such as a compiler.

157 ⟨*program.code* 157⟩≡ 158a ▷

```
...some code...
```

Note how the code is numbered, according to the page number, on the left of the name, and that a letter is appended to distinguish code parts on the same page.

This code can also be split up into several pieces. The next code is added to the previous, and this is illustrated with the + sign next to the name.

158a ⟨*program.code* 157⟩+≡ ◁157
```
...some more code...
```

See how both code parts are annotated to the right of the name with a small directed arrow to show where the code continues. The annotations also allow reference to code in the text, like this ⟨*program.code* 157⟩.

Code can also be presented at different levels, such as in this example, where an outer structure is presented, and the details are found elsewhere.

158b ⟨*highlevel.code* 158b⟩≡
```
⟨part 1 of this code 158c⟩
⟨part 2 of this code 158d⟩
```

Here comes details of part 1.

158c ⟨*part 1 of this code* 158c⟩≡ (158b)
```
...some code...
```

Here comes details of part 2.

158d ⟨*part 2 of this code* 158d⟩≡ (158b)
```
...some other code...
```

Note that the number in brackets to the right of the name shows where this piece of code is included.

Finally the literate programming tool can show such connections for identifiers, such as for instance functions or libraries. This code uses a function.

158e ⟨*some.code* 158e⟩≡
```
x = foo 1 2
```

The reader can then read at the bottom of the code where the identifier foo can be found. The definition of `foo` also states that it contains `foo`, as shown.

158f ⟨*library.code* 158f⟩≡
```
foo x y = ...
```

And that concludes the detailed crash course in reading literate code.

E.3 Main Module and CreolCompiler Library

The compiler is divided in two parts; the main program and the Creol compiler library. Since Haskell is a purely functional language, side effects such as input and output are structured through the IO monad. There are many monads, and the IO monad separates those functions that do input and input from other code, therefore all input and output is typically done in the main function, which is an IO monad. The Functional Creol Compiler does all input and output in the main function, which is an IO monad. The Functional Creol Compiler offers a library with a specialised scanner, a Creol parser, and an attribute grammar that produces synthesized attributes, such as the Creol Machine Code and error information.

E.3.1 The Main Module

The Main module contains the main program that is responsible for opening the source and destination files, as well as applying the scanner to produce tokens and parsing the tokens to produce the abstract syntax tree. Then the semantic function for the abstract syntax tree is used to produce attributes.

159a ⟨*Main.hs* 159a⟩≡ 159b▷

```
module Main(main) where
- System libraries
import IO
import System
import Directory
- Creol Compiler libraries
import CreolCompiler

help :: String
help = "The Fuctional Creol Compiler\n\
\License: GNU GPL Version 2\n\
\Author: Jørgen H. Fjeld <jhf@hex.no>\n\
\Homepage: http://www.kompetent.no/fcc\n\
\Version: 2005.02.01 \n\
\Usage:\n\
\CreolCompiler <sourcefile> <targetfile>\n"
```

The monadic function main is invoked by the operating system when the compiler is executed. This code checks that the user has invoked that compiler correctly.

159b ⟨*Main.hs* 159a⟩+≡ ◁159a 159c▷

```
main :: IO ()
main = do args <- getArgs
          case args of
           [sourcefile,targetfile] -> compile sourcefile targetfile
           _ -> putStr help
```

The compilation itself is broken down into different stages and separated into a compile function. This function uses the scanner to produce tokens, the the parser is applied to the tokens to produce the abstract syntax tree, and finally the semantic function `wrap_Ast` is used to produce the synthesized and chained attributes of the complete abstract syntax tree.

159c ⟨*Main.hs* 159a⟩+≡ ◁159b 160a▷

```
- Inherited attributes specified with field labels
inherited = Inh_Ast {}

- Compile function
compile sourcefile targetfile
  = do token <- scanner sourcefile
       ast <- parser token
       synthesized <- return (wrap_Ast (sem_Ast ast) inherited)
```

The Haskell language does not support named function arguments. This can be awkward for humans, as we have to remember the position of each argument. Moreover each function has only one return value, resulting in tuples as return values, and again we have to remember the position of each synthesized element. To combat this tediousness the wrap_Ast function allows inherited attributes and synthesized attributes to be specified with datatypes that have field labels, and these labels can then be used for insertion and extraction of attributes. This explicitness is beneficial to both readers and code maintainers. The next part opens the output file and writes the virtual machine code, along with eventual error messages, the abstract syntax tree and the symbol table, allowing users to check if the compiler operates correctly. This information is useful due to the evolving nature of the Creol language and hence the Functional Creol Compiler.

160a ⟨*Main.hs* 159a⟩+≡ ◁159c 160b▷

```
file <- openFile targetfile WriteMode
hPutStr file
        ( "***(\n" ++
          {- Extract the error attribute -}
          (preSuffix "- Errors:\n" (error_Syn_Ast synthesized) "\n") ++
          "- Abstract Syntax Tree \n" ++
          (shows ast "\n- Symbol Table \n") ++
          {- Extract the symboltable attribute -}
          ( shows (environments_Syn_Ast synthesized)
                  "\n- Creol Machine Code \n)\n"
          ) ++
          {- Extract the cmc attribute -}
          (cmc_Syn_Ast synthesized) ++ "\n"
        )
```

Finally, error messages are written to the standard destination.

160b ⟨*Main.hs* 159a⟩+≡ ◁160a

```
err <- openFile "/dev/stderr" WriteMode
errors <- return (error_Syn_Ast synthesized)
hPutStr err (preSuffix "Errors:\n" errors "\n")
if length errors > 0
  then exitWith (ExitFailure 1)
  else exitWith ExitSuccess
```

E.3.2 Creol Compiler Library Module

The Creol compiler library is a University of Utrecht Attribute Grammar file that contains the different parts of the compiler. The Attribute Grammar is translated to Haskell with the University of Utrecht Attribute Grammar compiler. Any text within braces {} in an Attribute Grammar file is copied verbatim into the Haskell code, which allows Haskell code in the Attribute Grammar files. There is no explicit module declaration, rather a module is implicitly defined with the same name as the file, thus this defines the CreolCompiler module, as well as the support libraries that are needed.

161a ⟨*CreolCompiler.ag* 161a⟩≡ 161b▷

```
{
- Haskell libraries
import Numeric
- University of Utrech Attribute Grammar libraries
import UU.Parsing
import UU.Parsing.Derived
import UU.Parsing.Perms
import UU.Parsing.Merge
import UU.Scanner
import UU.Scanner.Scanner
import UU.Scanner.Token
import UU.Scanner.TokenParser
import UU.Scanner.Position
}
```

The rest of the library is defined in separate code, and the next part shows how the code in the is assembled together by the literate programming tool.

161b ⟨*CreolCompiler.ag* 161a⟩+≡ ◁161a

⟨*Auxiliary Functions* 216⟩
⟨*Abstract Syntax Tree Structure* 162⟩
⟨*Parametrised Scanner* 166e⟩
⟨*Parser* 167⟩
⟨*Creol Machine Code Attributes* 209⟩
⟨*Creol Machine Code Semantic functions* 210⟩
⟨*Unique Labels* 215⟩
⟨*Type Definitions* 175⟩
⟨*Typechecker Attributes* 179⟩
⟨*Typechecker Semantic Functions* 181⟩
⟨*Symbol Table Definitions* 200⟩
⟨*Symbol Table Attributes* 204⟩
⟨*Symbol Table Semantic Functions* 205⟩

E.4 Abstract Syntax Tree

162 ⟨*Abstract Syntax Tree Structure* 162⟩≡ (161b)

```
DERIVING *     : Show

DATA Ast
     | Ast interfaces: Interfaces
           classes: Classes

TYPE Classes = [ Class ]
DATA Class
     | Class
       pos : Pos
       name : TypeId
       typeparameters : MaybeTypeDeclarations
       parameters : MaybeDeclarations
       implements : MaybeTypeDeclarations
       inherits : MaybeTypeDeclarations
       variables : MaybeDeclarations
       methods : MaybeMethods

DATA MaybeMethods
     | Nothing
     | Just methods : Methods
TYPE Methods = [Method]
DATA Method
     | Method
       signature : Signature
       variables : MaybeDeclarations
       code : Statement

TYPE Interfaces = [ Interface ]
DATA Interface
     | Interface
       pos : Pos
       name : TypeId
       typeparameters : MaybeTypeDeclarations
       inherits : MaybeTypeDeclarations
       signatures: MaybeSignatures

DATA MaybeSignatures
     | Nothing
     | Just signatures : Signatures
TYPE Signatures = [Signature]
DATA Signature
     | Signature
       pos : Pos
       name : VarId
       in : MaybeDeclarations
       out : MaybeDeclarations
       caller : TypeId

DATA Statement
```

```
      | Binary
        operator : String
        pos : Pos
        left : Statement
        right : Statement
      | Unary
        guard : Expression
        statement : Statement
      | Nullary
        statement : Statement
      | Assign
        name : VarId
        expr : Expression
      | While
        pos : Pos
        condition : Expression
        statement : Statement
      | Call
        object : VarId
        method : VarId
        in : MaybeExpressions
        out : MaybeVarIds
      | Send
        label : MaybeLabel
        object : VarId
        method : VarId
        in : MaybeExpressions
      | Receive
        label : Label
        pos : Pos
        vars : MaybeVarIds
      | Await
        pos : Pos
        condition : Expression
      | Wait
        pos : Pos
      | Expression
        expression : Expression

DATA MaybeLabel
      | Nothing
      | Just
        label : Label
DATA Label
      | Label
        label : String
        pos : Pos

DATA MaybeExpressions
         | Nothing
         | Just expressions : Expressions
TYPE Expressions = [Expression]
DATA MaybeExpression
         | Nothing
```

```
          | Just expression : Expression

DATA Expression
     | Binary
       operator : String
       pos : Pos
       left : Expression
       right : Expression
     | Unary
       operator : String
       pos : Pos
       expression : Expression
     | Nullary
       expression : Expression
     | Null
       pos : Pos
     | Variable
       name : VarId
     | Int
       value : Int
       pos : Pos
     | Bool
       value : Bool
       pos : Pos
     | String
       value : String
       pos : Pos
     | New
       constructor : TypeId
       tin : MaybeTypeIds
       in : MaybeExpressions
     | If
       pos : Pos
       condition : Expression
       statement : Statement
       elsepart : Statement

DATA MaybeVarIds
     | Nothing
     | Just varids : VarIds
TYPE VarIds = [VarId]
DATA VarId
     | VarId
       varid : String
       pos : Pos

DATA MaybeTypeIds
     | Nothing
     | Just typeids : TypeIds
TYPE TypeIds = [TypeId]
DATA TypeId
     | TypeId
       typeid : String
       pos : Pos
```

```
DATA MaybeDeclarations
     | Nothing
     | Just declarations : Declarations
TYPE Declarations = [Declaration]
DATA Declaration
     | Var
       name : VarId
       typeid : TypeId
       default : MaybeExpression

DATA MaybeTypeDeclarations
     | Nothing
     | Just declarations : TypeDeclarations
TYPE TypeDeclarations = [TypeDeclaration]
DATA TypeDeclaration
     | Var
       name : TypeId
```

E.5 Scanner

Instead of writing a new scanner, the University of Utrecht Haskell Compiler scanner was patched in a rather non-intrusive way to work with Creol syntax. The patch for the UUAG scanner is presented here for as documentation.

165 ⟨*Patch for Scanner.hs* 165⟩≡

```
Index: Scanner.hs
===================================================================
RCS file: /data/cvs-rep/uust/lib/scanner/UU/Scanner/Scanner.hs,v
retrieving revision 1.3
diff -r1.3 Scanner.hs
77,78c77,78
<    doScan p ('-':'-':s)  = doScan p (dropWhile (/= '\n') s)
<    doScan p ('{':'-':s)  = lexNest doScan (advc 2 p) s
--
>    doScan p ('/':'/':s)  = doScan p (dropWhile (/= '\n') s)
>    doScan p ('/':'*':s)  = lexNest doScan (advc 2 p) s
163,164c163,164
<  where lexNest' c p ('-':'}':s) = c (advc 2 p) s
<        lexNest' c p ('{':'-':s) = lexNest' (lexNest' c) (advc 2 p) s
--
>  where lexNest' c p ('*':'/':s) = c (advc 2 p) s
>        lexNest' c p ('/':'*':s) = lexNest' (lexNest' c) (advc 2 p) s
```

To optimise the Creol scanner, it is given prior knowledge of keywords and operators. This information must correspond with usage of the scanner, and this correspondence is not checked. To illustrate this, there is a correspondence between what the `pSpec` function can parse and the special characters used to parametrise the scanner. Such that each of the ⟨*Scanner Special Characters* 166a⟩ can be parsed by the `pSpec` function, but if one is to use `pSpec` for other characters, the Haskell compiler will not complain, and the parse will always fail. The `pSpec` function and other scanner functions are given an intuitive explanation in Section 3.4.1.

166a ⟨*Scanner Special Characters* 166a⟩≡ (166e)
```
";,()."
```

The operator characters can be combined to form operator keywords. This information is only used to speed up the scanning process.

166b ⟨*Scanner Operator Characters* 166b⟩≡ (166e)
```
"=<>|[]:+-*/?!"
```

The operator keywords are those that are built from one or several operator characters. These may be scanned with the `pKey` function. Note that all the characters used here must be defined in ⟨*Scanner Special Characters* 166a⟩ and that this is by convention and not by design, hence not automatically checked by the Haskell compiler.

166c ⟨*Scanner Operator Keywords* 166c⟩≡ (166e)
```
[">=", "<=", "!=", "=","==", "->", ":=", "[]", "|||"
,"+","-","<",">","!=","*","/",":","?","!"
]
```

The following string keywords that otherwise would scan as identifiers, are now recognised by the `pKey` function as keywords.

166d ⟨*Scanner String Keywords* 166d⟩≡ (166e)
```
["interface", "inherits", "begin", "with", "op", "asm"
,"inv", "where", "func", "end", "contract", "class", "await"
,"implements", "var", "if", "then", "else", "fi", "True"
,"False", "in", "out", "while", "do", "od", "new", "wait"
,"null","and", "or", "not"
-, "this" - Is parsed as a variable reference/varid, not keyword.
]
```

The code that feed the optimisation information to the scanner, and defines the scanner used by Creol is now presented.

166e ⟨*Parametrised Scanner* 166e⟩≡ (161b)
```
{
keywordstxt = ⟨Scanner String Keywords 166d⟩
keywordsop = ⟨Scanner Operator Keywords 166c⟩
specialchars = ⟨Scanner Special Characters 166a⟩
opchars = ⟨Scanner Operator Characters 166b⟩

scanner :: String -> IO [Token]
scanner filename = scanFile keywordstxt
                            keywordsop
                            specialchars
                            opchars
                            filename
}
```

E.6 Parser

167 ⟨*Parser* 167⟩≡ (161b) 171▷

```
{
parser token = parseIO pAst token

pAst :: Parser Token Ast
pAst = pMerged Ast_Ast
               (    list_of pInterface
               <||> list_of pClass
               )

pClass :: Parser Token Class
pClass = Class_Class
  <$> (pKeyPos "class")
  <*> pTypeId
  <*> pMaybeBracksTypeDeclarations
  <*> pMaybeParensDeclarations
  <*> pMaybePrefixTypeDeclarations "implements"
  <*> pMaybePrefixTypeDeclarations "inherits"
  <* (pKey "begin")
  <*> pPrefixMaybeDeclarations "var"
  <* (pKey ";"  <|> pSucceed "") - To remove trailing ";"
  <*> pMaybeMethods
  <* (pKey "end")

pMaybeMethods :: Parser Token MaybeMethods
pMaybeMethods = opt ( MaybeMethods_Just
                      <$> pMethods
                    )
                    MaybeMethods_Nothing

pMethods :: Parser Token Methods
pMethods = pFoldr1_gr ((++),[]) pMethodWith

pMethodWith :: Parser Token Methods
pMethodWith = map
            <$> (   (\x f->f x)
                 <$> pPrefixDefaultTypeId "with" "Any"
                )
            <*> pList1_gr pMethod

pMethod :: Parser Token (TypeId -> Method)
pMethod = (\a b c d -> Method_Method (a d) b c)
      <$> pSignature
      <* (pKey "==")
      <*> pPrefixSuffixMaybeDeclarations "var" ";"
      <*> pStatement
      <* ((pKey ".") <|> (pKey ";") <|> (pSucceed ""))

pInterface :: Parser Token Interface
pInterface = Interface_Interface
  <$> (pKeyPos "interface")
  <*> pTypeId
```

```
  <*> pMaybeBracksTypeDeclarations
  <*> pMaybePrefixTypeDeclarations "inherits"
  <* (pKey "begin")
  <*> pMaybeSignatures
  <* (pKey "end")


pMaybeSignatures :: Parser Token MaybeSignatures
pMaybeSignatures = opt ( MaybeSignatures_Just
                          <$> pSignatures
                        )
                        MaybeSignatures_Nothing

pSignatures :: Parser Token Signatures
pSignatures = pFoldr1_gr ((++),[]) pSignatureWith

pSignatureWith :: Parser Token Signatures
pSignatureWith = map
            <$> (   (\x f ->f x)
                 <$> pPrefixDefaultTypeId "with" "Any"
                )
            <*> pList1_gr pSignature

pSignature :: Parser Token (TypeId -> Signature)
pSignature
  =   (   Signature_Signature
      <$> (pKeyPos "op")
      <*> pVarId
      )
  <-> (   pParens (    pPrefixMaybeDeclarations "in"
                   <+> pPrefixMaybeDeclarations "out"
                  )
      <|> pSucceed (MaybeDeclarations_Nothing,MaybeDeclarations_Nothing)
      )

pStatement :: Parser Token Statement
pStatement = pStatement_Binary

pStatement_Binary :: Parser Token Statement
pStatement_Binary =
 foldr transform
       pStatement_Unary
       precOpers
 where
  transform (assoc,ops) =
    assoc ( Statement_Binary <$-> (pAny pKeyPosTuple ops))
  precOpers =
    [
     (pChainr, ["[]","|||"])
    ,(pChainr_ng, [";"]) - _ng allows ; to end lines as well
     - ,(pChainr, ["->"]) - replaced by await
     -,(pChainr, ["and","or"]) - no meaning without ->
    ]
```

```
pStatement_Unary :: Parser Token Statement
pStatement_Unary = pStatement_Nullary
{-  <|> Statement_Unary
        <$> pExpression
        <* (pKey "->")
        <*> pStatement_Nullary
-}

pStatement_Nullary :: Parser Token Statement
pStatement_Nullary =
      Statement_Nullary
        <$> pParens pStatement
  <|> Statement_Assign
        <$> pVarId
        <* (pKey ":=")
        <*> pExpression
  <|> Statement_While
        <$> (pKeyPos "while")
        <*> pExpression
        <* (pKey "do")
        <*> pStatement
        <* (pKey "od")
  <|> pStatement_Call
  <|> pStatement_Send
  <|> Statement_Receive
        <$> pLabel
        <*> (pKeyPos "?")
        <*> (pParens pMaybeVarIds)
  <|> Statement_Await
        <$> (pKeyPos "await")
        <*> pExpression
  <|> Statement_Wait
        <$> (pKeyPos "wait")
  <|> Statement_Expression
        <$> pExpression

pStatement_Send :: Parser Token Statement
pStatement_Send
  =          Statement_Send
         <$> pMaybeLabel
         <*> (   ( pKey "!" *> pVarId <* pKey ".")
              <|> ( VarId_VarId "this"
                  <$> pKeyPos "!"
                  )
             )
         <*> pVarId
         <*> pParens pMaybeExpressions

pStatement_Call :: Parser Token Statement
pStatement_Call = (Statement_Call
                  <$> ( pVarId <* pKey "."
                      <|> pSucceed (VarId_VarId "this" noPos)
                      )
                  <*> pVarId
```

```
                )
             <-> pInOut

pMaybeLabel :: Parser Token MaybeLabel
pMaybeLabel = opt (MaybeLabel_Just <$> pLabel)
                  MaybeLabel_Nothing

pLabel :: Parser Token Label
pLabel =     Label_Label
         <$-> pVaridPos

pMaybeParensExpressions ::Parser Token MaybeExpressions
pMaybeParensExpressions
  =    (   MaybeExpressions_Just
       <$> pParens pExpressions
       )
   <|> pParens (pSucceed MaybeExpressions_Nothing)
   <|> pSucceed MaybeExpressions_Nothing

pMaybeExpressions ::Parser Token MaybeExpressions
pMaybeExpressions = opt ( MaybeExpressions_Just
                          <$> pExpressions )
                        MaybeExpressions_Nothing

pExpressions :: Parser Token Expressions
pExpressions = pList1Sep_gr pComma pExpression

pMaybePrefixExpression :: String -> Parser Token MaybeExpression
pMaybePrefixExpression p = opt ( MaybeExpression_Just
                                 <$ pKey p
                                 <*> pExpression
                                 )
                                 MaybeExpression_Nothing

pMaybeExpression :: Parser Token MaybeExpression
pMaybeExpression = opt ( MaybeExpression_Just
                         <$> pExpression )
                       MaybeExpression_Nothing

pExpression :: Parser Token Expression
pExpression = pExpression_Binary
}
```

The operator precedence for the binary operators is decided in the Creol parser, where the operators with lowest precedence come first, and a association function determines right or left associativity. The association function `pChainr` is left to right, while `pChainl` is right to left. The expression `1 + 2 * 3 < 4 and True or False` is parsed as `((((1 + (2 * 3)) < 4) and True) or False)`.

171 ⟨*Parser* 167⟩+≡ (161b) ◁167

```
{
pExpression_Binary :: Parser Token Expression
pExpression_Binary =
 foldr transform
       pExpression_Unary
       precOpers
 where
  transform (assoc,ops) =
    assoc (Expression_Binary <$-> (pAny pKeyPosTuple ops))
  precOpers =
    [ (pChainr, ["or", "and"])
    , (pChainr, ["<", "<=", ">", ">=", "!=", "="])
    , (pChainr, ["+","-"])
    , (pChainr, ["*","/"])
    ]

pExpression_Unary :: Parser Token Expression
pExpression_Unary =
             Expression_Unary
         <$-> (pAny pKeyPosTuple ["not","-"])
         <*> pExpression
     <|> pExpression_Nullary

pExpression_Nullary :: Parser Token Expression
pExpression_Nullary =
        Expression_If
        <$> (pKeyPos "if")
        <*> pExpression
        <* (pKey "then")
        <*> pStatement
        <*> ( (   (pKey "else")
                *> pStatement
              )
             <|> (Statement_Expression <$> (Expression_Null <$> pSucceed noPos))
            )
        <* (pKey "fi")
    <|> Expression_Nullary
          <$> ( pParens pExpression )
    <|> Expression_Null
        <$> pKeyPos "null"
    <|> Expression_Variable
          <$> pVarId
    <|> (     Expression_Int
          <$-> ( (\(x,y)->(read x,y)) <$> pIntegerPos )
        )
    <|> (     Expression_Bool
```

```
          <$-> (    (\x->(True,x)) <$> (pKeyPos "True")
               <|> (\x->(False,x)) <$> (pKeyPos "False")
        )      )
    <|> (     Expression_String
          <$-> pStringPos
        )
    <|> Expression_New
          <$> ( (pKey "new")
              *> pTypeId
              )
          <*> pMaybeBracksTypeIds
          <*> pMaybeParensExpressions

pInOut :: Parser Token (MaybeExpressions,MaybeVarIds)
pInOut =  pParens (   pMaybeExpressions
                  <+> ( opt ( pSemi *> pMaybeVarIds )
                            MaybeVarIds_Nothing
                  )   )

pMaybeDeclarations :: Parser Token MaybeDeclarations
pMaybeDeclarations =
  opt (    MaybeDeclarations_Just
       <$> pDeclarations
      )
      MaybeDeclarations_Nothing

pMaybeParensDeclarations :: Parser Token MaybeDeclarations
pMaybeParensDeclarations
  =        MaybeDeclarations_Just
       <$> pParens pDeclarations
    <|> pParens (pSucceed MaybeDeclarations_Nothing)
    <|> pSucceed MaybeDeclarations_Nothing

pPrefixMaybeDeclarations :: String ->
                        Parser Token MaybeDeclarations
pPrefixMaybeDeclarations prefix =
  opt (    MaybeDeclarations_Just
       <$ (pKey prefix)
       <*> pDeclarations
      )
      MaybeDeclarations_Nothing

pPrefixSuffixMaybeDeclarations :: String -> String ->
                        Parser Token MaybeDeclarations
pPrefixSuffixMaybeDeclarations prefix suffix =
  opt (    MaybeDeclarations_Just
       <$ (pKey prefix)
       <*> pDeclarations
       <* (pKey suffix)
      )
      MaybeDeclarations_Nothing

pDeclarations :: Parser Token Declarations
pDeclarations =  pFoldr1Sep_gr ((++),[]) pComma pDeclaration
```

```
pDeclaration :: Parser Token Declarations
pDeclaration =      map
              <$$> (pList1Sep_gr pComma pVarId)
              <*>  (Declaration_Var
                   <$$->  (pPrefixTypeId ":"
                          <+> pMaybePrefixExpression "="
                   )      )

pMaybeBracksTypeDeclarations :: Parser Token MaybeTypeDeclarations
pMaybeBracksTypeDeclarations
  =        MaybeTypeDeclarations_Just
       <$> pBracks pTypeDeclarations
    <|> pBracks (pSucceed MaybeTypeDeclarations_Nothing)
    <|> pSucceed MaybeTypeDeclarations_Nothing

pMaybePrefixTypeDeclarations :: String -> Parser Token MaybeTypeDeclarations
pMaybePrefixTypeDeclarations prefix
  =        MaybeTypeDeclarations_Just
       <$ pKey prefix <*> pTypeDeclarations
    <|> pSucceed MaybeTypeDeclarations_Nothing

pTypeDeclarations :: Parser Token TypeDeclarations
pTypeDeclarations = pList1_gr pTypeDeclaration

pTypeDeclaration :: Parser Token TypeDeclaration
pTypeDeclaration = TypeDeclaration_Var
               <$> pTypeId

pMaybeVarIds :: Parser Token MaybeVarIds
pMaybeVarIds = opt ( MaybeVarIds_Just <$> pVarIds )
                   MaybeVarIds_Nothing
pVarIds :: Parser Token VarIds
pVarIds  = pList1Sep_gr pComma pVarId
pVarId :: Parser Token VarId
pVarId = VarId_VarId <$-> pVaridPos

pMaybeBracksTypeIds :: Parser Token MaybeTypeIds
pMaybeBracksTypeIds
  =        MaybeTypeIds_Just
       <$> pBracks pTypeIds
    <|> pBracks (pSucceed MaybeTypeIds_Nothing)
    <|> pSucceed MaybeTypeIds_Nothing

pTypeIds :: Parser Token TypeIds
pTypeIds = pList1Sep_gr pComma pTypeId
pPrefixDefaultTypeId :: String -> String -> Parser Token TypeId
pPrefixDefaultTypeId p d =     ((pKey p) *> pTypeId)
                           <|> (    (TypeId_TypeId d)
                                <$> (pSucceed noPos)
                               )
pPrefixTypeId :: String -> Parser Token TypeId
pPrefixTypeId p = pKey p *> pTypeId
pTypeId = TypeId_TypeId <$-> pConidPos
```

```
}
```

E.7 Type Analysis

E.7.1 Type Definitions

175 ⟨*Type Definitions* 175⟩≡ (161b) 178▷

```
{
          {- Term Types -}
          {- Primitive Creol Types -}
data Type = BoolType
          | IntType
          | StrType
          {- References -}
          | RefType {ref :: Type}
          {- Algebraic Data Types -}
          | RecordType{ btps::BoundTypes }
          | VariantType {
              name :: String
            , tps :: Types
            }
          {- Procedures -}
          | FunctionType { inparams :: Types
                         , outparam :: Type
                         }
          {- Type from Theta
           This is used to express fixpoints and mutual recursion
           -}
          | VarType { tn :: TypeName }
          {- Parametrisation -}
          | FreeType{ {- A type bound by ForallType -}
              ftn :: TypeName - A free type name
            }
          | ForallType{ -A parametrised type with ftn_count arguments
              ftn_count :: Int - Binds fnt_count free type names
            , tp :: Type
            }
          {- Nominal Constraint(s) -}
          | NominalType{ names :: [Symbols]
                       , tp :: Type
                       }
          {- Object Oriented Types -}
          | ObjectType{ row :: Row - The fields in the object
                      }
          | MethodType { cointerface :: Type
                       , inparams :: Types
                       , outparams :: Types
                       }
          {- Meta types -}
          | VoidType
            {- When Any is used to prevent cascading errors,
                preserve the erroneus type for better error messages.
            -}
          | AnyType { former :: Maybe Type }
          | OverloadedType { tps :: Types }
          | ErrorType {message :: String, former :: Maybe Type}
```

```
         deriving (Eq,Show)

data Row = Row { tag :: Tag
               , name :: Symbol
               , tp :: Type
               , row :: Row
               }
          | Open
          | Closed
         deriving (Eq)

data Tag = Variable
         | Method
         | Abstract
         | Inherit
         deriving (Eq)

instance Show Tag where
  show Variable = "variable "
  show Method = "method "
  show Abstract = "method "
  show Inherit = "inherit "

type Types = [Type]
type BoundType = (Symbol,Type)
type BoundTypes = [BoundType]

rowtpmap :: (Type->Type) -> Row -> Row
rowtpmap f r@Row{} = r{tp=f (tp r),row=rowtpmap f (row r)}
rowtpmap f Open = Open
rowtpmap f Closed = Closed

openrow :: Row -> Bool
openrow r@Row{} = openrow (row r)
openrow Open = True
openrow Closed = False

closedrow :: Row -> Bool;
closedrow row = not (openrow row);

findrow :: [Tag] -> Symbol -> Row -> Maybe Type
findrow ts s r@Row{}
  = if (s == (name r)) && wanted
    then Just (tp r)
    else findrow ts s (row r)
    where wanted = case ts of
                     [] -> True
                     _ ->  (tag r) `elem` ts
findrow ts s r@Open = Nothing
findrow ts s r@Closed = Nothing

{- A valid interface is described by an open object
   with only Abstract and Inherit tags -}
is_interface :: Env -> Type -> Bool
```

```
  is_interface e t@VarType{} = is_interface (expand e t)
  is_interface e t@ObjectType{}
    = onlytags [Abstract,Inherit] (row t)
      where
        onlytags :: [Tag] -> Row -> Bool
        onlytags ts r@Row{} = ((tag r) `elem` ts) && (onlytags ts (row r))
        onlytags _ Open = True
        onlytags _ Closed = True
  is_interface _ _ = False

  instance Show (Env,Row) where
    show (e,r@Row{})
      = (show (row r)) ++ show (e,tp r) ++ show (e,row r)
    show (e,Open) = ".."
    show (e,Closed) = ""

  instance Show (Env,Type) where
    show (_,BoolType) = "Bool"
    show (_,IntType) = "Int"
    show (_,StrType) = "Str"
    show (e,t@RefType{}) = (show (e,ref t))++ " variable"
    show (e,t@RecordType{}) = "{"++(show (e,btps t))++"}"
    show (e,t@VariantType{}) = (name t)++"{"++show (e,tps t)++"}"
    show (e,t@VarType{}) = show (e,expand t)
    show (e,t@FreeType{}) = (name t)++(show (num t))
    show (e,t@ForallType{}) = "Types 0.."++(show (num_offset t))++
                              "=>"++(show (e,tp t))
    show (e,t@NominalType{}) = show (e,tp t) ++ " " ++ (name t)
    show (e,t@ObjectType{}) = "Object{" ++ (show (e,row t)) ++"}"
    show (e,t@MethodType{}) = (show (name t))
    show (e,t@FunctionType{}) = (show (e,inparams t))++
                                "->"++(show (e,outparam t))
    show (_,VoidType) = "Void"
    show (e,AnyType{former=Just t}) = "erroneus " ++ (show (e,t))
    show (_,AnyType{former=Nothing}) = "Any"
    show (e,t@OverloadedType{}) = "Overloaded ["++(show (e,tps t))++"]"
    show (_,t@ErrorType{}) = "error: "++(message t)

  instance Show (Env,Types) where
    show (e,tps) = show (map (\t->show (e,t)) tps)

  instance Show (Env,BoundTypes) where
    show (e,btps) = show (map (\(s,t)->s++":"++(show (e,t))) btps)

  }
```

The MethodType represents a mapping from a set of types to a set of types. FunctionType is different in that application of a functiontype some types will yield the return type as the only type.

178 ⟨*Type Definitions* 175⟩+≡ (161b) ◁175

```
{

stdenv = Env{ symbols =
         [("Bool",(RefType BoolType,[],GammaSymbol))
         ,("Int",(RefType IntType,[],GammaSymbol))
         ,("Str",(RefType StrType,[],GammaSymbol))
         ,("+",(OverloadedType [ii2i,ss2s],[],GammaSymbol))
         ,("-",(OverloadedType [ii2i,ss2s,i2i],[],GammaSymbol))
         ,("*",(ii2i,[],GammaSymbol))
         ,("/",(ii2i,[],GammaSymbol))
         ,("<",(ii2b,[],GammaSymbol))
         ,("<=",(ii2b,[],GammaSymbol))
         ,(">",(ii2b,[],GammaSymbol))
         ,(">=",(ii2b,[],GammaSymbol))
         ,("=",(OverloadedType [ii2b,bb2b],[],GammaSymbol))
         ,("/=",(OverloadedType [ii2b,bb2b],[],GammaSymbol))
         ,("and",(bb2b,[],GammaSymbol))
         ,("or",(bb2b,[],GammaSymbol))
         ,("not",(b2b,[],GammaSymbol))
         ,(";",(vv2v,[],GammaSymbol))
         ,("[]",(vv2v,[],GammaSymbol))
         ,("|||",(vv2v,[],GammaSymbol))
         ,("if",(bvv2v,[],GammaSymbol))
         ,("while",(bv2v,[],GammaSymbol))
         ,("await",(b2v,[],GammaSymbol))
         ]
  , types = []
  , freetype = 0
  , traversed = []
  } where
    b2v = FunctionType [BoolType] VoidType
    bv2v = FunctionType [BoolType,VoidType] VoidType
    bvv2v = FunctionType [BoolType,VoidType,VoidType] VoidType
    vv2v = FunctionType [VoidType,VoidType] VoidType
    ii2b = FunctionType [IntType,IntType] BoolType
    bb2b = FunctionType [BoolType,BoolType] BoolType
    ii2i = FunctionType [IntType,IntType] IntType
    ss2s = FunctionType [StrType,StrType] StrType
    oo2b = FunctionType [ObjectType{row=Open},ObjectType{row=Open}] BoolType
    i2i = FunctionType [IntType] IntType
    b2b = FunctionType [BoolType] BoolType

}
```

E.7.2 Typechecker Attributes

179 ⟨*Typechecker Attributes* 179⟩≡ (161b) 180a▷

```
ATTR * - Ast VarIds TypeIds MaybeTypeIds MaybeVarIds MaybeTypeDeclarations
         TypeDeclarations MaybeDeclarations Declarations MaybeMethods
         Methods MaybeSignatures Signatures MaybeExpressions
         Expressions Classes Interfaces
     [ - Inherited
     | - Chained
     | - Synthesized
       type : Type
     ]

ATTR Expressions
     VarIds TypeIds MaybeTypeIds MaybeVarIds
     MaybeTypeDeclarations TypeDeclarations
     MaybeDeclarations Declarations
     MaybeMethods Methods
     MaybeSignatures Signatures
     MaybeExpressions Expressions
     [ - Inherited
     | - Chained
     | - Synthesized
       types USE { ++ } { [] }: Types
     ]

ATTR MaybeTypeDeclarations TypeDeclarations
     MaybeVarIds VarIds MaybeTypeIds TypeIds
     MaybeSignatures Signatures
     MaybeMethods Methods
     MaybeDeclarations Declarations
     [ - Inherited
     | - Chained
     | - Synthesized
       ids USE { ++ } { [] } : {[String]}
     ]

ATTR TypeDeclaration VarId TypeId Signature Label MaybeLabel
     Signature Method Declaration
     [ - Inherited
     | - Chained
     | - Synthesized
       id : String
     ]

ATTR TypeDeclaration TypeDeclarations MaybeTypeDeclarations
     [ - Inherited
     | - Chained
       type_offset : Int
     | - Synthesized
     ]

ATTR MaybeTypeDeclarations
```

```
[ - Inherited
     | - Chained
       type : Type
     | - Synthesized
     ]
```

The caller must know its own class. The type of the class is then inherited from Class.

180a ⟨*Typechecker Attributes* 179⟩+≡ (161b) ◁179 180b ▷

```
ATTR MaybeMethods Methods Method Statement Expression Expressions
     MaybeExpression MaybeExpressions Declaration Declarations
     MaybeDeclarations Signature Signatures MaybeSignatures
     [ - Inherited
       this : Type - The fix point type of the class
     | - Chained
     | - Synthesized
     ]
```

To print type error messages, any type errors must be transferred to a local error variable, along with the position in the source file where the error occurred. This error must then be synthesized up to the top node, where it is printed to the user. The position must also be synthesized from where it is put during the parsing, such that all elements in the syntax tree may report errors with position.

180b ⟨*Typechecker Attributes* 179⟩+≡ (161b) ◁180a

```
ATTR *
     [ - Inherited
     | - Chained
     | - Synthesized
       error USE { `consNewline` } {""} : String
       pos USE { `first` } {noPos} : Pos
     ]

{
consNewline = consSep "\n"
first = \x y -> x
}
```

E.7.3 Typechecker Semantic Functions

181 ⟨*Typechecker Semantic Functions* 181⟩≡ (161b)

```
{ - Auxiliary Functions

- Expand/Unfold an iso-recursive type.
expand :: Env -> Type -> Type
expand env@Env{} t@VarType{tn=tn}
  = case lookup tn (types env) of
      Just t -> t
      Nothing -> ErrorType{message="Internal error: undefined type variable "++
                                   (show (num t))++"."
                           ,former=Nothing}

- Replace a free type with another type
substfree :: [(TypeName,Type)] -> Type -> Type
substfree _ t@BoolType = t
substfree _ t@IntType = t
substfree _ t@StrType = t
substfree s t@VariantType{} = t{tps=(map (substfree s) (tps t))}
substfree s t@RecordType{} = t{btps=map (\(n,t)->(n,substfree s t)) (btps t)}
substfree _ t@VarType{} = t
substfree s t@FreeType{}
  = case lookup (num t) s of { Just t' -> t' ; Nothing -> t }
substfree s t@ForallType{}
  = t{tp=(substfree (shift_subst (num_offset t) s) (tp t))}
substfree s t@NominalType{} = t{tp = substfree s (tp t)}
substfree s t@ObjectType{} = t{row=rowtpmap (substfree s') (row t)}
substfree s t@MethodType{}
  = t{cointerface=substfree s (cointerface t)
     ,inparams=map (substfree s) (inparams t)
     ,outparams=map (substfree s) (outparams t)
     }
substfree s t@FunctionType{}
  = t{inparams=map (substfree s) (inparams t)
     ,outparam=substfree s (outparam t)
     }
substfree _ t@VoidType = t
substfree _ t@AnyType{} = t
{-
substfree s t@OverloadedType{} = t{tps=map (substfree s) (tps t)}
-}
substfree _ t@ErrorType{} = t

- Shift a substitution with an offset
- Used to support non-flat types
shift_subst :: Int -> [Subst] -> [Subst]
shift_subst offset list = [(n+offset,t)
                           | (n,t) <- list ]

- State variance intents explicitly
contravariance env sub sup = conforms env sup sub
covariance env sub sup = conforms env sub sup
invariance env sub sup = if sub == sup
```

```
                        then Nothing
                        else Just "Invariance not satisfied"

- Check if two types are conforming
- The first type is the lower bound
- The second type is the upper bound
- Return Nothing if conformance, else return error message.
conforms :: Env -> Type -> Type -> Maybe String
- Errors propogate, AnyType is always ok
conforms _ sub@ErrorType{} sup@ErrorType{}
  = Just ((message sub) ++ (message sup))
conforms _ _ e@ErrorType{} = Just (message e)
conforms _ e@ErrorType{} _ = Just (message e)
conforms _ AnyType{} _ = Nothing
conforms _ _ AnyType{} = Nothing
{- Expand type names
   A type name is a fixpoint of a recursive type.
   Subtyping uses the Amber rule for fixpoints.
   Matching disregards fixpoints.
   The special treatment of fixpoints only occurs
   when both sub and sup are type names
   The Amber rule checks conformance under a subtype assumption
   between the fixpoints.
   Matching disregards fixpoints by replacing the fixpoint of sup
   with the fixpoint of sub.
   Subtyping must first check if these type names are already under
   a subtype assumption.
   Matching must check if the sub and sup type names are the same. -}
conforms e sub@VarType{} sup@VarType{}
  = if sametypename
    then Nothing
    else if truebyassumption
         then Nothing
         else if usematching
              then conforms e (expand e sub) supwithsubfixpoint
              else conforms ewithassumption (expand e sub) (expand e sup)
    where
     sametypename = (name sub) == (name sup)
     truebyassumption = or [ ((tn sub) == tn_sub_ass) &&
                             ((tn sup) == tn_sup_ass)
                           | (tn_sub_ass,tn_sup_ass) <- assumptions e ]
     usematching = isopenobject (expand e sup)
     supwithsubfixpoint = substvar [(tn sup),(tn sub)] (expand e sup)
     ewithassumption = e{assumptions=(tn sub,tn sup):(assumptions e)}
     {- An open object can be contained by
          parametrisation (ForallType)
          initial parameters (FunctionType)
          nominal constraints (NominalType)
        and nothing else -}
     isopenobject t@ForallType{} = isopenobject (tp t)
     isopenobject t@FunctionType{} = isopenobject (outparam t)
     isopenobject t@NominalType{} = isopenobject (tp t)
     isopenobject t@ObjectType{} = openrow (row t)
     isopenobject _ = False
```

```
conforms e sub@VarType{} sup = conforms e (expand e sub) sup
conforms e sub sup@VarType{} = conforms e sub (expand e sup)
- Simple types are equal themselves
conforms _ BoolType BoolType = Nothing
conforms _ IntType IntType = Nothing
conforms _ StrType StrType = Nothing
- References are both source and sink, therefore invariance
conforms e sub@RefType{} sup@RefType{} = invariance e (ref sub) (ref sup)
- Variants and records are only equal themselves
conforms e sub@VariantType{} sup@VariantType{} = invariance e sub sup
conforms e sub@RecordType{} sup@RecordType{} = invariance e sub sup
conforms e sub@FreeType{} sup@FreeType{}
  = if (num sub) == (num sup)
    then Nothing
    else Just "Incompatible type parameters"
{- Parametrised types conform when
   * same number of type parameters
   * constraints conforms - not yet implemented
   * contained types conforms -}
conforms e sub@ForallType{} sup@ForallType{}
  = case compare (num_offset sub) (num_offset sup) of
     LT -> Just ("Polymorphic type "++(show (envs,sub))++
                 " has too few parameters"
                )
     GT -> Just ("Polymorphic type "++(show (envs,sub))++
                 " has too many parameters"
                )
     EQ -> conforms e (tp sub) (tp sup)
- Nominal types are restricted by declarations in Theta^Nominal
- and by structural restrictions
conforms e sub@NominalType{} sup@NominalType{}
  = if reachable (name sub) (constraints e)
    then conforms e (tp sub) (tp sup)
    else Just ("No inheritance from " ++ (name sup) ++
               " to " ++ (name sub) ++ ".")
    where {
reachable reachables paths
  = if found
    then True
    else if stuck
         then False
         else searchmore
    where {
found = (name sup) `elem` reachables ;
stuck = (len reachables) == 0;
searchmore = reachable newreachables pathsleft ;
newreachables = [ stop
                | start `elem` reachables
                | (start,stop) <- paths ] ;
pathsleft = [ (start,stop)
            | start `notElem` reachables
            | (start,stop) <- paths ] ;
} ;
}
```

```
{- Object types are checked depending on rows
   An open upper bound reduces to matching
      matching can instantiate a row variable with rows
      - not yet implemented
   A closed upper bound reduces to subtyping
      subtyping can use subsumption to forget rows
   A row variable (open type) may not be forgotten by subsumption
   A row variable can be instantiated to contain another row variable.
     - not yet implemented
   -}
conforms e sub@ObjectType{} sup@ObjectType{}
  = rowconforms e (row sub) (row sup)
    where {
- Make sure each row in sup is present and conforming in sub
rowconforms :: Row -> Row -> Maybe String ;
rowconforms sub@Row{} sup@Row{}
  = if (name sub) == (name sup)
    then case conforms (tp sub) (tp sup) of
           Nothing -> rowconforms sub (row sup)
           errormessage  -> errormessage
    else rowconforms (row sub) sup ;
- A row wasn't found in sub
rowconforms Open r@Row{} = Just ("missing " ++ (name r)) ;
rowconforms Closed r@Row{} = Just ("missing " ++ (name r)) ;
- An open upper bound uses matching to instantiate the open row variable
rowconforms r Open = Nothing ; - Instantiate Open in sup with r
                             - Not yet implemented.
- A closed upper bound can forget methods, but not a row variable!
rowconforms r Closed
  = if openrow r
    then Just "Internal error: matching with closed type"
    else Nothing - The r is forgotten by subsumption ;
}
conforms e sub@MethodType{} sup@MethodType{}
  = case getErrors (c:i++o) of
      Just e -> Just ("incompatible methods because " ++ e)
      Nothing -> Nothing
     where
        c = contravariance e (cointerface sub) (cointerface sup)
        i = zipWith (contravariance e) (inparams sub) (inparams sup)
        o = zipWith (covariance e) (outparams sub) (outparams sup)
conforms e sub@FunctionType{} sup@FunctionType{}
  = case getErrors (o:i) of
       Just e -> Just ("incompatible funcitons because " ++ e)
       Nothing -> Nothing
     where
      i = zipWith (contravariance e) (inparams sub) (inparams sup)
      o = covariance e (outparam sub) (outparam sup)
conforms _ sub@VoidType VoidType = sub
conforms _ sub@AnyType{} _ = sub
conforms _ sub AnyType{} = sub
- Return the error, even if in the supertype
conforms envs sub@ErrorType{} sup = Just (message sub)
conforms envs sub sup@ErrorType{} = Just (message sup)
```

```
conforms envs sub sup = Just ((show (envs,sub))++
                              " does not fit in "++
                              (show (envs,sup)))

- Check that there are no inheritance cycles
- Information lookup in objects may not terminate
- due to inheritance cycles.
checkinheritance :: Env -> Maybe String
checkinheritance e
  = getErrors [findcycle t | (tn,t) <- types e]
    where {
findcycle :: [TypeName] -> Type -> Maybe String;
findcycle visited t@ForallType{} = findcycle visited (tp t);
findcycle visited t@FunctionType{} = findcycle visited (outparam t);
findcycle visited t@NominalType{} = findcycle visited (tp t);
findcycle visited t@VarType{}
  = if (tn t) `member` visited
    then Just ("Inheritance cycle " ++ (show (e,t)) ++
               "inherits from itself.")
    else findcycle ((tn t):visited) (expand e t);
findcycle visited t@ObjectType{} = findrowcycle visited (row t);
findcycle _ _ = Nothing;
findrowcycle :: [TypeName] -> Row -> MaybeString;
findrowcycle visited r@Row{tag=Inherit}
  = findcycle visited (tp r);
findrowcycle visited r@Row{} = findrowcycle visited (row r);
findrowcycle _ _ = Nothing;
}

- Build the nominal constraints according to
- declared inheritance
constraints :: Env -> Env
constraints e = e{constraints=[ (lower,upper)
                              | upper <- findupper lower
                              | lower <- findnominals t
                              | (tn,t) <- types e
                              ]
                 }
                where {
findnominals :: Type -> [Symbol] ;
findnominals t = case t of
  FunctionType{} -> findnominals (outparam t)
  ForallType{} -> findnominals (tp t)
  NominalType{} -> names t
  ObjectType{} -> findrownominals (row t)
  _ -> []
;
findrownominals :: Row -> [Symbol];
findrownominals r = case r of
  Row{tag=Inherit} -> (findnominals (tp r)) ++ (findrownominals (row r))
  Row{} -> findrownominals (row r)
  _ -> []
;
}
```

```
{- Inheritance from A to B is type safe when
   the most revealing types of self are in
   the matching relation.
   This is checked by safeinheritance
-}
safeinheritance
-
- Check if conformance is satisfied on inheritance
- Return possible error message(s)
checkvariance :: Env -> Maybe String
checkvariance e = getErrors [ variance t
                            | (tn,t) <- types e
                            ]
                  where {
variance :: Type -> Maybe String

}

- extend a BoundTypes with another according to variance
oplus :: (Type->Type->Type) -> BoundTypes -> BoundTypes -> BoundTypes
oplus variance subs sups
  = [ case lookup subn sups of
       Nothing -> sub
       Just supt -> (subn,t)
         where
           t = case subt `variance` supt of
             e@ErrorType{} -> e
                 - e{message="There is a problem in "++subn++
                 - " because "++(message e)}
             t -> t
    | sub@(subn,subt) <- subs
    ]

- Extend sub with each of sup in succession
inherits :: Envs -> Type -> [Type] -> Envs
inherits envs varsub [] = envs
inherits envs varsub (h:t) = inherits (inherit envs varsub h) varsub t

- Extend sub with sup, such that sub <: sup
- Update the type environment with the changes
- Notice that the supertype can be expanded
- while the subtype can not be expanded, because that would give
- an infinite loop. The subtype is therefore updated
- in place with replaceType
inherit :: Envs -> Type -> Type -> Envs
inherit envs _ VarType{} = envs
inherit envs _ _ = envs
{-
inherit envs varsub varsup
  = case (varsub,varsup) of
      (VarType{},VarType{})
         -> envs
      (_,_) -> envs
```

```
-}
{-
inherit envs varsub varsup
  = case (varsub,varsup) of
      - Both must be vartypes, to expand fixpoints simultaneously
      (VarType{},VarType{})
         -> envs
             {- envs3
             where
             - Assume that fix points are subtypes
             envs1 = envs{assumptions=(num varsub,num varsup)
                                      :(assumptions envs)}
             (envs2,oldsub) = replaceType envs1 (num varsub) newsub
             (envs3,newsub) = extend envs2 oldsub (partExpand envs varsup)
             -}
      (_,_) -> envs
                 {-envs1
                where
                 envs1 = envs
                 (envs1,t) = replaceType envs (num varsub) ErrorType{
                        message="Internal compiler error in inherit"
                        ,former=Just t}
                 -}
    where {
extend :: Envs -> Type -> Type -> (Envs,Type) ;
extend envs sub@ObjectType{} sup@ObjectType{}
  = (envs,newsub)
    where
      mbts = oplus (subtype envs) (methods sub) (methods sup)
      vbts = oplus (invariance envs) (variables sub) (variables sup)
      error = getError ((map snd mbts) ++ (map snd vbts))
      newsub = case error of
        ErrorType{} -> error{message="Inheritance impossible  because "++
                        (show (envs,sub))++" can not inherit from "++
                        (show (envs,sup)) ++ " because " ++ (message error)
                        ,former=Just sub}
        _ -> sub{variables = vbts
                ,methods = mbts
                }
;
extend envs sub sup
  = case (sub,sup) of
      (ClassType{},ClassType{}) -> extend envs sub sup
      (ClassType{},InterfaceType{}) -> extend envs sub sup
      (InterfaceType{},InterfaceType{}) -> extend envs sub sup
  where {
extend :: Envs -> Type -> Type -> (Envs,Type) ;
extend envs sub sup
  = (envs3,newsub)
    where
      envs1 = inherit envs (self sub) (self sup)
      envs2 = inherit envs1 (open sub) (open sup)
      envs3 = inherit envs2 (closed sub) (closed sup)
      error = getError (map (partExpand envs3)
```

```
                          [self sub,open sub,closed sub])
      newsub = case error of
        ErrorType{} -> error{message="Inheritance impossible  because "++
                        (show (envs,sub))++" can not inherit from "++
                        (show (envs,sup)) ++ " because " ++ (message error)
                        ,former=Just sub}
        _ ->case sup of
             ClassType{}->sub{classes = (name sup,varsup):(classes sub)}
             InterfaceType{}->
                sub{interfaces = (name sup,varsup):(interfaces sub)}

}
;
extend envs sub sup = replaceType envs (num varsub)
                    ErrorType{message="Inheritance impossible for "++
                                      (show (envs,sub))++
                                      " from "++(show (envs,sup))
                             ,former=Just sub}
} - End where
-}

{- Find method from object
   First search the current object, then traverse parents
-}
findrow :: Env -> Type -> String -> Type
findrow e t s = case t of
  VarType{} -> findrow e (expand e t) s
  FunctionType{} -> findrow (outparam t)
  ForallType{} -> findrow (tp t)
  ObjectType{} -> rowfindrow (row t)
  where {
rowfindrow :: Row -> Type
}

- Get method from interface
getMethod :: Envs -> Type -> String -> Type
getMethod envs@Envs{complete=env@Env{}} mu m = getMethod (expand envs mu) m
  where {
getMethod mu@InterfaceType{} m
  = case lookup m (methods mu) of
      Just mt -> mt
      Nothing -> ErrorType{message="The interface " ++ (name mu) ++
                                   " has no method " ++ m
                          ,former=Nothing}
;
getMethod mu@ErrorType{} m = mu
;
getMethod mu m = ErrorType{message=(show mu)++" has no methods.",former=Nothing}
}

- Create the open,closed and self types corresponding to a class/interface
- and put these into the environment and the class/interface
- Return the new type environment.
selves :: Envs -> Type -> Type -> Envs
```

```
selves envs _ _ = envs
{-
selves envs vartype@VarType{} self0@ObjectType{} = envs3
  where
    - If this is a parametrised class/interface
    - then the open and closed types are also parametrised
    - while the type of self is not parametrised.
    - inspect self0 to determine if this and
    - create wrap to parametrize, and get the unparametrised type of self
    (wrap,self1) = case self0 of
      t@ForallType{tp=self} -> (\o->t{tp=o},self)
      self@ObjectType{} -> (id,self)
    - Keep the origin of this object for better error messages
    - Make self an open type
    self = self1{origin = vartype
                 ,openrow = True}
    - Put self into the type environment
    (envs0,varself) = newType envs self
    {- The closed type derived from self is obtained by
       * remove hidden information
       * Close the object
       * Replace fixpoint with this new closed type.
       * use wrap to maintain parametrisation
    -}
    hiddenself = self{variables=[]}
    closedself = hiddenself{openrow=False}
    closed = substvar [(num varself,varclosed)] closedself
    (envs1,varclosed) = newType envs0 (wrap closed)
    {- The open type derived from self is obtained by
       * Remove hidden information (Already done for closed)
       * Open the object (Already open)
       * Replace fixpoint with this new open type
       * use wrap to maintain parametrisation
    -}
    open = substvar [(num varself,varopen)] hiddenself
    (envs2,varopen) = newType envs1 (wrap open)
    - Insert the updated information about selves into class/interface
    - Don't expand, but use replace to exchange type,
    - Expand would cause infinite loop (obviously)
    - since we wan't to update the partial type,
    - not the finished type.
    (envs3,t') = replaceType envs2 (num vartype) t
    t = t'{open = varopen, closed = varclosed, self = varself}
selves envs _ ErrorType{} = envs
selves envs ErrorType{} _ = envs
selves envs _ AnyType{} = envs
selves envs AnyType{} _ = envs
-}
- Get error from list of types
getError :: Types -> Type
getError [] = AnyType{former=Nothing}
getError (h@(ErrorType{}):tl) = h
getError (h:tl) = getError tl
```

```
- Get errors from list of Nothing or Just
getErrors :: [Maybe String] -> Maybe String
getErrors l = foldl combine Nothing l
  where
    combine Nothing Nothing = Nothing
    combine (Just a) Nothing = Just a
    combine Nothing (Just b) = Just b
    combine (Just a) (Just b) = Just (a ++" "++ b)

- Check that type application is possible
- and return result type or error message
apply_tp :: Envs -> Type -> Types -> Type
apply_tp envs@Envs{complete=e@Env{}} tf@VarType{} tps
  = case lookup (num tf) (types e) of
      Just t -> apply_tp envs t tps
      Nothing -> ErrorType{message="Compiler error: undefined type "++
                                   (show (num tf))
                           ,former=Nothing}
apply_tp envs tf@ForallType{} tps
  = case compare (length tps) (num_offset tf) of
      LT -> ErrorType{message="Not enough type parameters",former=Just tf}
      GT -> ErrorType{message="Too many type parameters",former=Just tf}
      EQ -> substfree [(n,tps!!n) | n <- [0..(num_offset tf)]] (tp tf)
apply_tp _ tf [] = tf - When no parameters and not Forall, just keep the type
apply_tp _ tf@ErrorType{} _  = tf
apply_tp _ tf _ = ErrorType{message="Can not parametrise " ++ (show tf)
                            ,former=Just tf}

- Check that parameter application  is possible
- and return result type or error message
apply_fun :: Envs -> Type -> Types -> Type
apply_fun envs f@FunctionType{} ps
  = case compare (length ps) (length (inparams f)) of
     LT -> ErrorType{message="Not enough parameters",former=Just f}
     GT -> ErrorType{message="Too many parameters",former=Just f}
     EQ -> case getError (zipWith (subtype envs) ps (inparams f)) of
            t@ErrorType{} -> ErrorType{message="Incompatible parameter " ++
                                               (message t)
                                       ,former=Just f}
            _ -> (outparam f)
apply_fun envs fs@OverloadedType{} ps
  = case compare (length candidates) 1 of
     LT -> ErrorType{message="Could not apply "++(show ps)++
                             " to any of "++(show (tps fs))
                     ,former=Just fs}
     GT -> ErrorType{message="Too many possible subtypes!",former=Just fs}
     EQ -> candidates!!0
     where candidates = [ t
                        | f <- (tps fs)
                        , let t = apply_fun envs f ps
                        , case t of ErrorType{} -> False; _ -> True
                        ]
apply_fun envs f@ErrorType{} _  = f
apply_fun envs f ps = ErrorType{message="Internal error: apply_fun "++
```

```
                                        (show f)++" "++(show ps)
                                  ,former=Just f}

- Assignment requires a reference or an object type.
apply_assign envs lval rval
  = case (expand envs lval,expand envs rval) of
      (ErrorType{},_) -> lval
      (_,ErrorType{}) -> rval
      (AnyType{},_) -> lval
      (_,AnyType{}) -> rval
      (RefType{ref=lval},_) -> assignable lval
      (ObjectType{},_) -> assignable lval
      (InterfaceType{},_) -> assignable lval
      (ClassType{},_) -> assignable lval
      (_,_) -> ErrorType{former = Nothing
             ,message = "Assignment requires variable left of :="
             }
    where {
assignable lval
  = case subtype envs rval lval of - Substitutability
      t@ErrorType{} -> t{message="Assignment failure because " ++
                                  (message t)
                        ,former=Just lval}
      _ -> VoidType
}

apply_receive :: Envs -> Type -> Types -> Type
apply_receive envs f@MethodType{} ops - Asynchronous Receive
  = case compare (length ops) (length (outparams f)) of
      LT -> ErrorType{message="Not enough out parameters",former=Nothing}
      GT -> ErrorType{message="Too many out parameters",former=Nothing}
      EQ -> case getError (zipWith (subtype envs) (outparams f) ops) of
              e@ErrorType{} -> e{message="Incompatible out " ++
                                         "parameter " ++ (message e)}
              _ -> VoidType
apply_receive _ f@ErrorType{} _  = f

apply_send :: Envs -> Type -> Type -> Types -> Type
apply_send envs f@MethodType{} caller ips - Asynchronous Send
  = case compare (length ips) (length (inparams f)) of
     LT -> ErrorType{message="Not enough in parameters",former=Nothing}
     GT -> ErrorType{message="Too many in parameters",former=Nothing}
     EQ -> case subtype envs caller (cointerface f) of
            t@ErrorType{} -> t{message="Caller has wrong interface"}
            _ -> case getError (zipWith (subtype envs) ips (inparams f)) of
                  e@ErrorType{} -> e{message="Incompatible parameter "
                                                      ++ (message e)}
                  _ -> VoidType
apply_send envs f@ErrorType{} _ _  = f

apply_method :: Envs -> Type -> Type -> Types -> Types -> Type
apply_method envs f@MethodType{} caller ips ops - Synchronous
  = case getError [ apply_send envs f caller ips, apply_receive envs f ops ] of
      e@ErrorType{} -> e
```

```
      _ -> VoidType
apply_method envs f@AnyType{} _ _ _ = f
apply_method envs f@ErrorType{} _ _ _  = f

apply_class :: Envs -> Type -> Types -> Type
{-
apply_class envs f@OverloadedType{} ps
  = case compare (length ts) 1 of
      LT -> ErrorType{message="None of "++(show (tps f))++" is a class."
                     ,former=Nothing}
      GT -> ErrorType{message="More than one of "++(show (tps f))++" is a class."
                     ,former=Nothing}
      EQ -> apply_class (ts!!0) ps
    where ts = [ t | t@ClassType{} <- (tps f) ]
-}
apply_class envs f@ClassType{} ps
  = case compare (length ps) (length (parameters f)) of
     LT -> ErrorType{message="Not enough class parameters",former=Nothing}
     GT -> ErrorType{message="Too many class parameters",former=Nothing}
     EQ -> case getError (zipWith (subtype envs) ps (parameters f)) of
            e@ErrorType{} -> e{message="Incompatibel class parameter "
                                       ++ (message e)}
            _ -> self f
apply_class envs f@ErrorType{} _  = f
apply_class envs f@AnyType{} _ = f
apply_class envs f@InterfaceType{} ps
  = ErrorType{message=(name f)++" is not a class.",former=Nothing}
apply_class envs f ps = ErrorType{message=(show f)++" is not a class."
                                  ,former=Nothing}

}


SEM Ast
  | Ast
    (loc.envs)
       = envs2
         where
          (envs0,vartype) = newType @loc.emptyenvs it
          it = InterfaceType
                {name = "Any"
                ,interfaces = []
                ,open = AnyType Nothing
                ,closed = AnyType Nothing
                ,self = AnyType Nothing
                }
          envs1 = selves envs0 vartype ot
          ot = ObjectType
               {origin = AnyType Nothing
               ,variables=[]
               ,methods=[]
               ,openrow = True
               }
          t = partExpand envs1 vartype
```

```
          ns = @loc.namespace
          (envs2,_) = addSymbols envs1
            [("Any",(open t,ns,GammaSymbol))
            ,("@Any",(closed t,ns,GammaSymbol))
            ,("Any",(vartype,ns,GammaInterface))
            ]
    lhs.error = insertSep "\n"
                  [@classes.error,@interfaces.error]

SEM Class
    | Class
      typeparameters.type_offset = 0
      - Create prelminiary type of objects for this class
      - and let parametrisation change if necessary
      typeparameters.type
        = ObjectType
           {origin = AnyType Nothing
           ,variables=zip @variables.ids @variables.types
           ,methods=zip @methods.ids @methods.types
           ,openrow = True
           }
      - Create the class type ct and use
      - type from parametrisation to build selves,
      - which also updates information in ct about selves.
      - then perform inheritance
      - and find errors
      (loc.envs,loc.vartype,loc.error)
       = (envs3,vartype,error)
         where
          (envs0,vartype) = newType @lhs.environments ct
          ct = ClassType
                 {name = @name.id
                 ,interfaces = []
                 ,classes = []
                 ,parameters = @parameters.types
                 ,open = AnyType Nothing
                 ,closed = AnyType Nothing
                 ,self = AnyType Nothing
                 }
          envs1 = selves envs0 vartype @typeparameters.type
          ns = @lhs.namespace
          getClass = getSymbol envs1 ns GammaClass
          superclasses = map getClass ("Any" : @inherits.ids)
          envs2 = inherits envs1 vartype superclasses
          getInterface = getSymbol envs2 ns GammaInterface
          superinterfaces = map getInterface @implements.ids
          envs3 = inherits envs2 vartype superinterfaces
          (envs4,error) = findError envs3 @pos vartype @loc.error1
      lhs.error = insertSep "\n"
                    [@loc.error, @variables.error, @methods.error]

SEM Interface
    | Interface
      typeparameters.type_offset = 0
```

```
      - Create prelminiary type of objects for this interface
      - and let parametrisation change if necessary
      typeparameters.type
        = ObjectType
           {origin = AnyType Nothing
           ,variables=[]
           ,methods=[] - zip @signatures.ids @signatures.types
           ,openrow = True
           }
      - Create the interface type (it) and use
      - type from parametrisation to build selves
      - then perform inheritance
      - and find errors
      (loc.envs,loc.this,loc.error)
       = (envs3,vartype,error)
         where
          (envs0,vartype) = newType @lhs.environments it
          it = InterfaceType
                {name = @name.id
                ,interfaces = []
                ,open = AnyType Nothing
                ,closed = AnyType Nothing
                ,self = AnyType Nothing
                }
          envs1 = selves envs0 vartype @typeparameters.type
          ns = @lhs.namespace
          getInterface = getSymbol envs1 ns GammaInterface
          superinterfaces = map getInterface ("Any" : @inherits.ids)
          envs2 = inherits envs1 vartype superinterfaces
          (envs3,error) = findError envs2 @pos vartype @loc.error1
      lhs.error = insertSep "\n" [@loc.error, @signatures.error]

SEM MaybeMethods
    | Nothing
      lhs.types = []
    | Just
      lhs.types = @methods.types

SEM Methods
    | Cons
      lhs.types = @hd.type : @tl.types
      lhs.ids = @hd.id : @tl.ids

SEM Method
    | Method
      loc.type1 = @signature.type
      lhs.error = insertSep "\n" [ @loc.error, @signature.error, @code.error]
      (loc.type,loc.error)
        = catchError @signature.pos @loc.type1 @loc.environments ""

SEM MaybeSignatures
    | Just
      lhs.types = @signatures.types
```

```
SEM Signatures
    | Cons
      lhs.types = @hd.type : @tl.types
      lhs.ids = @hd.id : @tl.ids

SEM Signature
    | Signature
      loc.type1 = MethodType{name = @name.id
                            ,cointerface = @loc.caller
                            ,inparams = @in.types
                            ,outparams = @out.types
                            }
      loc.caller = getSymbol @lhs.environments @lhs.namespace
                             GammaSymbol
                             @caller.id
      lhs.error = insertSep "\n" [ @loc.callererror
                                 , @loc.error,@in.error,@out.error]
      (loc.type,loc.error)
        = catchError @pos @loc.type1 @loc.environments @loc.error1
      (_,loc.callererror)
        = catchError @pos @loc.caller @loc.environments ""
      lhs.id = @name.id

SEM Statement
    | Binary
      loc.type
        = apply_fun envs operator [ @left.type , @right.type ]
           where operator = getSymbol envs @lhs.namespace
                                       GammaSymbol
                                       @operator
                 envs = @lhs.environments
      lhs.error = insertSep "\n" [@loc.error, @left.error, @right.error]
      (lhs.type,loc.error)
        = catchError @pos @loc.type @lhs.environments ""
    | Assign
      loc.type = apply_assign envs lval @expr.type
                  where
                   lval = getSymbol envs @lhs.namespace
                                    GammaSymbol
                                    @name.id
                   envs = @lhs.environments
      lhs.error = insertSep "\n" [@loc.error, @expr.error]
      (lhs.type,loc.error)
        = catchError @name.pos @loc.type @lhs.environments ""
    | While
      loc.type
        = apply_fun envs whilestmt [@condition.type,@statement.type]
           where whilestmt = getSymbol envs [] GammaSymbol "while"
                 envs = @lhs.environments
      lhs.error = insertSep "\n" [@loc.error, @condition.error, @statement.error]
      (lhs.type,loc.error)
        = catchError @pos @loc.type @lhs.environments ""
     | Call
      loc.type
```

```
        = apply_method envs method @lhs.this @in.types outs
           where outs = map (getSymbol envs @lhs.namespace GammaSymbol)
                            @out.ids
                 method = getSymbols envs @lhs.namespace GammaSymbol
                                        [@object.id,@method.id]
                 envs = @lhs.environments
      lhs.error = insertSep "\n" [@loc.error]
      (lhs.type,loc.error) = catchError @method.pos @loc.type @lhs.environments ""
     | Send
      loc.type = apply_send envs @loc.methodtype @lhs.this @in.types
                  where envs = @lhs.environments
      loc.methodtype
        = getMethod envs interface @method.id
           where interface = getSymbol envs @lhs.namespace
                                       GammaSymbol
                                       @object.id
                 envs = @lhs.environments
      lhs.error = insertSep "\n" [@loc.error]
      (lhs.type,loc.error)
        = catchError @object.pos @loc.type @loc.environments @loc.error1
    | Receive
      loc.type
        = apply_receive envs method @vars.types
           where method = getSymbol envs @lhs.namespace
                                    GammaSymbol
                                    @label.id
                 envs = @lhs.environments
      lhs.error = insertSep "\n" [@loc.error]
      (lhs.type,loc.error)
        = catchError @label.pos @loc.type @lhs.environments ""
    | Await
      loc.type
        = apply_fun envs await [@condition.type]
           where await = getSymbol envs [] GammaSymbol "await"
                 envs = @lhs.environments
      lhs.error = insertSep "\n" [@loc.error, @condition.error]
      (lhs.type,loc.error)
        = catchError @pos @loc.type @lhs.environments ""
    | Wait
      lhs.type = VoidType
     | Expression
       lhs.type = @expression.type

SEM MaybeLabel
    | Nothing
      lhs.type = VoidType

SEM Label
    | Label
      lhs.id = @label
      lhs.pos = @pos
      lhs.type = VoidType

SEM MaybeExpressions
```

```
    | Just
      lhs.types = @expressions.types

SEM Expressions
    | Nil
      lhs.types = []
    | Cons
      lhs.types = @hd.type : @tl.types

SEM MaybeExpression
    | Nothing
      lhs.type = VoidType

SEM Expression
  | Binary
    loc.type
      = apply_fun envs operator [ @left.type , @right.type ]
        where operator = getSymbol envs [] GammaSymbol @operator
              envs = @lhs.environments
    lhs.error = insertSep "\n" [@loc.error, @left.error, @right.error]
    (lhs.type,loc.error) = catchError @left.pos @loc.type @lhs.environments ""
  | Unary
    loc.type
      = apply_fun envs operator [ @expression.type ]
        where operator = getSymbol envs [] GammaSymbol @operator
              envs = @lhs.environments
    lhs.error = insertSep "\n" [@loc.error, @expression.error]
    (lhs.type,loc.error)=catchError @expression.pos @loc.type @lhs.environments ""
  | Null
    loc.type = VoidType
  | Variable
    loc.type = getSymbol @lhs.environments @lhs.namespace
                         GammaSymbol
                         @name.id
    lhs.error = insertSep "\n" [@loc.error]
    (lhs.type,loc.error) = catchError @name.pos @loc.type @lhs.environments ""
  | Int
    loc.type = IntType
  | Bool
    loc.type = BoolType
  | String
    loc.type = StrType
  | New
    loc.type
      = apply_class envs constr @in.types
        where
          constr = apply_tp envs pconstr @tin.types
          pconstr = getSymbol envs @lhs.namespace
                              GammaClass
                              @constructor.id
          envs = @lhs.environments
    lhs.error = insertSep "\n" [@loc.error,@tin.error,@in.error]
    (lhs.type,loc.error)
      = catchError @constructor.pos @loc.type @lhs.environments ""
```

```
  | If
    loc.type
      = apply_fun envs ifstmt [@condition.type,@statement.type,@elsepart.type]
         where ifstmt = getSymbol envs [] GammaSymbol "if"
               envs = @lhs.environments
    lhs.error = insertSep "\n" [@loc.error, @condition.error,
                @statement.error, @elsepart.error]
    (lhs.type,loc.error)
      = catchError @pos @loc.type @loc.environments ""

SEM MaybeDeclarations
    | Just
      lhs.types = @declarations.types
      lhs.error = @declarations.error

SEM Declarations
    | Cons
      lhs.types = @hd.type : @tl.types
      lhs.ids = @hd.id : @tl.ids
      lhs.error = consNewline @hd.error @tl.error

SEM Declaration
    | Var
      loc.type1 = getSymbol @lhs.environments @lhs.namespace
                            GammaSymbol @typeid.id
      lhs.error = insertSep "\n" [@loc.error]
      (loc.type,loc.error)
        = catchError @name.pos @loc.type1 @lhs.environments @loc.error1
      lhs.type = @loc.type
      lhs.id = @name.id

SEM MaybeTypeDeclarations
    | Nothing
      lhs.types = []
      lhs.type = @lhs.type
    | Just
      lhs.types = @declarations.types
      lhs.type = ForallType @declarations.type_offset @lhs.type

SEM TypeDeclarations
    | Cons
      tl.type_offset = @lhs.type_offset
      hd.type_offset = @tl.type_offset
      lhs.type_offset = @hd.type_offset
      lhs.types = @hd.type : @tl.types
      lhs.ids = @hd.id : @tl.ids
      lhs.error = consNewline @hd.error @tl.error

SEM TypeDeclaration
    | Var
      lhs.id = @name.id
      loc.type1 = FreeType @lhs.type_offset @name.id
      lhs.type_offset = @lhs.type_offset + 1
      lhs.error = insertSep "\n" [@loc.error]
```

```
        (loc.type,loc.error)
          = catchError @name.pos @loc.type1 @lhs.environments @loc.error1

SEM VarIds
    | Cons
      lhs.types = @hd.type : @tl.types
      lhs.ids = @hd.id : @tl.ids

SEM VarId
    | VarId
      lhs.id = @varid
      lhs.pos = @pos
      lhs.type = getSymbol @lhs.environments @lhs.namespace
                           GammaSymbol @varid

SEM TypeIds
    | Cons
      lhs.types = @hd.type : @tl.types
      lhs.ids = @hd.id : @tl.ids

SEM TypeId
    | TypeId
      lhs.id = @typeid
      lhs.pos = @pos
      lhs.type = getSymbol @lhs.environments @lhs.namespace
                           GammaSymbol @typeid
```

E.7.4 Symbol Table Definitions

200 ⟨*Symbol Table Definitions* 200⟩≡ (161b)

```
{
type Namespace = [String]
type Symbol = String
data Gamma = GammaSymbol | GammaClass | GammaInterface deriving (Show,Eq)
type SymbolInfo = (Type,Namespace,Gamma)
type TypeName = Int
type TypeNames = [TypeName]
data Env = Env
  { symbols :: [(Symbol,SymbolInfo)] - A list of symbol mappings
  , types::[(TypeName,Type)] - Environment of types
  , constraints::[(Symbol,Symbol)] - Nominal restrictions
  , assumptions :: [(TypeName,TypeName)] - fixpoint subtype assumptions
  , freetype::TypeName - The next free type variable
  , traversed :: [TypeName] - To detect cycles from mutual recursion
  }

emptyEnv = Env [] [] [] 0 []

type Subst = (TypeName,Type)

instance Show Env where
  show e@Env{} = "Symbols:\n" ++
                 (insertSep "\n" (map show (symbols e))) ++
                 "\nTypes: \n" ++
                 (insertSep "\n" (map show (types e))) ++
                 "\n" ++
                 "\nAssumptions:\n" ++
                 (insertSep "\n" [(show a)++"<:"++(show b)
                                 | (a,b) <- assumptions es]) ++
                 "\n\n"

- Insert symbol into environment
addSymbol :: Env -> (Symbol,SymbolInfo) -> (Env,String)
addSymbol e@Env{} r@(name,(tn,ns,g))
  = case (lookup name (symbols e)) of
      (Just (_,ns',g')) | (ns == ns') && (g == g')
         -> (e,"Redeclaration error of " ++ name)
      (_) -> (e',"")
    where e' = e{symbols=r:(symbols e)}

- Add several symbols
addSymbols :: Env -> [(Symbol,SymbolInfo)] -> (Env,String)
addSymbols e [] = (e,"")
addSymbols e (h:tl) = (e'',err'')
    where
      (e',err') = addSymbol e h
      (e'',err'') = case err' of
                      [] -> addSymbols e' tl
                      _ -> (e',err')

- Find symbol
```

```
getSymbol :: Env -> Namespace -> Gamma -> Symbol -> TypeName
getSymbol es ns g s = getSymbols es ns g [s]

- Use a path of symbols to find a type.
- Traverses internal namespaces in types.
getSymbols :: Env -> Namespace -> Gamma -> [Symbol] -> TypeName
getSymbols e@Env{} ns g (s:st)
  = foldl (\tn->findin (expand e tn)) (find (symbols e) ns g s) st
    where {
find::[(Symbol,SymbolInfo)]->Namespace->Gamma->Symbol->TypeName;
find [] ns g s = ErrorType{message="Undeclared symbol "++(show s)++ "."
                          ,former=Nothing} ;
find ((n',(t,ns',g')):tl) ns g n
  = if (n == n') && (contains ns' ns) && (g == g')
    then t
    else find tl ns g n ;
contains :: Namespace -> Namespace -> Bool ;
contains (h:t) (h':t') = if h == h' then contains t t' else False ;
contains [] _  = True ;
contains _ [] = False ;
findin :: Type -> Symbol -> Type ;
findin t@ErrorType{} _ = t ;
findin t@ObjectType{} s
  = case findrow [Method,Variable] s t of
     Just t' -> t'
     Nothing -> ErrorType{former=Nothing
         ,message="Could not find " ++ s ++ " in " ++
                  (show (e,t))++" object."
         }
;
findin t@AnyType{} s = t - AnyType prevent cascading errors
;
findin t s = ErrorType{message="Internal error: undefined findin for " ++
                                (show t)
                      ,former=Nothing}
}

- Add a new type to the environment
- The TypeName of the type is returned
newType :: Env -> Type -> (Env,TypeName)
newType e@Env{} t
  = (e',tn)
    where
     tn = freetype e
     e' = e{types=(tn,t):(types e),freetype=tn + 1}

- Replace a type in the environment
- Return the existing version for use in the update
replaceType :: Env -> TypeName -> Type -> (Env,Type)
replaceType e@Env{} tn t
  = case lookup tn (types e) of
      Nothing -> (e
                 ,ErrorType{message="Internal error: "++
                            "undefined type variable " ++ (show n)++"."
```

```
                          ,former=Nothing}
                )
      Just t'' -> (e{types=updated},t'')
                where
                 updated = [ (n',if n == n' then t else t')
                           | (n',t') <- types e
                           ]

- Update a type in the type environment
updateType :: Env -> TypeName -> Type -> Env
updateType e@Env{} tn t
  = case lookup tn (types e) of
      Nothing -> e{types=(n,t):(types e)}
      Just t' -> substvars e' substs
                 where
                  (t'',e',substs) = unify e t t'

- Apply each substitution to every element in the environment
- Use traversed to avoid infinite looops
substvars :: Env -> [Subst] -> Env
substvars env@Env{} substs
  = env{types=[ (n,substvar substs t) | (n,t) <- (types env)]}

- Apply substitution to type
substvar :: [Subst] -> Type -> Type
substvar _ t@BoolType = t
substvar _ t@IntType = t
substvar _ t@StrType = t
substvar s t@RefType{} = t{ref = substvar s (ref t)}
substvar s t@VariantType{} = t{tps=(map (substvar s) (tps t))}
substvar s t@RecordType{} = t{btps=map (\(n,t)->(n,substvar s t)) (btps t)}
substvar s t@VarType{}
  = case lookup (num t) s of { Just t' -> t' ; Nothing -> t }
substvar _ t@FreeType{} = t
substvar s t@ForallType{} = t{tp=substvar s (tp t)}
substvar s t@ObjectType{}
  = t{variables=map replace (variables t)
     ,methods=map replace (methods t)
     } where replace = \(n,t)->(n,substvar s t)
substvar s t@InterfaceType{}
  = t{self = substvar s (self t)
     ,open = substvar s (open t)
     ,closed = substvar s (closed t)
     ,interfaces=map replace (interfaces t)
     } where replace = \(n,t)->(n,substvar s t)
substvar s t@ClassType{}
  = t{parameters=map (substvar s) (parameters t)
     ,interfaces=map replace (interfaces t)
     ,classes=map replace (classes t)
     ,self = substvar s (self t)
     ,open = substvar s (open t)
     ,closed = substvar s (closed t)
     } where replace = \(n,t)->(n,substvar s t)
substvar s t@MethodType{}
```

```
  = t{cointerface=substvar s (cointerface t)
     ,inparams=map (substvar s) (inparams t)
     ,outparams=map (substvar s) (outparams t)
     }
substvar s t@FunctionType{}
  = t{inparams=map (substvar s) (inparams t)
     ,outparam=substvar s (outparam t)
     }
substvar _ t@VoidType = t
substvar _ t@AnyType{} = t
{-
substvar s t@OverloadedType{} = t{tps=map (substfree s) (tps t)}
-}
substvar _ t@ErrorType{} = t
{-
substvar s t = ErrorType{message="Compiler error: substvar not defined for "++
                                  (show t)
                        ,former=Nothing}
-}


- Unification on two types,
- returning the unified type and substitutions
unify :: Envs -> Type -> Type -> (Type,Envs,[Subst])
unify envs l@VarType{} r@VarType{}
  = if (num l) == (num r)
    then (l,envs,[])
    else (error,envs,[(num l,error),(num r,error)])
    where
      error = ErrorType{message="Incompatible types: "++ (show (num l)) ++
                                " and "++(show (num r))
                        ,former=Nothing}
unify envs l r@VarType{} = unify envs r l
unify envs@Envs{partial=env@Env{}} l@VarType{} r
  = if elem (num l) (traversed env) - Detect cycles
    then (error,envs,[(num l,error),(num r,error)])
    else (r,envs,[(num l,r)])
    where
      error = ErrorType{message="Illegal type cycle detected",former=Nothing}
unify envs t@ErrorType{} _ = (t,envs,[])
unify envs _ t@ErrorType{} = (t,envs,[])
unify envs AnyType{} t = (t,envs,[])
unify envs t AnyType{} = (t,envs,[])
unify envs _ _ = (ErrorType{message="Compiler error: could not unify."
                           ,former=Nothing}
                 ,envs,[])

}
```

E.7.5 Symbol Table Attributes

204 ⟨*Symbol Table Attributes* 204⟩≡ (161b)

```
- Chained environment
ATTR * - Ast
     [ - Inherited
      namespace : { [String] } - Contains starting point for symbol lookup
     | - Chained
      environments : Envs - Contains all symbols
     | - Synthesized
     ]

ATTR Signature
     [ - Inherited
     | - Chained
     | - Synthesized
      namespace : { [String] } - Contains starting point for symbol lookup
     ]

ATTR Ast
     [ - Inherited
     | - Chained
     | - Synthesized
      environments : Envs
     ]
```

E.7.6 Symbol Table Semantic Functions

205 ⟨*Symbol Table Semantic Functions* 205⟩≡ (161b) 208▷

```
SEM Ast
    | Ast
      loc.namespace = []
      classes.namespace = []
      interfaces.environments = @loc.envs
      loc.emptyenvs =  Envs{partial = stdenv
                           ,complete = partial @classes.environments
                           ,assumptions = []
                           }
      classes.environments
        = Envs{partial = partial @interfaces.environments
               ,complete = complete @interfaces.environments
               ,assumptions = assumptions @interfaces.environments
               }

SEM Interfaces
    | Nil
      lhs.environments = @lhs.environments
    | Cons
      tl.environments = @lhs.environments
      hd.environments = @tl.environments
      lhs.environments = @hd.environments

SEM Interface
    | Interface
      (signatures.environments,loc.error1)
        = addSymbols @loc.envs
            [(@name.id,(open it,@lhs.namespace,GammaSymbol))
            ,("@" ++ @name.id,(closed it,@lhs.namespace,GammaSymbol))
            ,(@name.id,(@loc.this,@lhs.namespace,GammaInterface))
            ,("This",(self it,@loc.namespace,GammaSymbol))
            ] where it = partExpand @loc.envs @loc.this
      loc.environments = @signatures.environments
      lhs.environments = @loc.environments
      loc.namespace = @lhs.namespace ++ [@name.id]

SEM Classes
    | Nil
      lhs.environments = @lhs.environments
    | Cons
      tl.environments = @lhs.environments
      hd.environments = @tl.environments
      lhs.environments = @hd.environments

SEM Class
    | Class
      (typeparameters.environments,loc.error1)
        = addSymbols @loc.envs
            [(@name.id,(open ct,@lhs.namespace,GammaSymbol))
            ,("@" ++ @name.id,(closed ct,@lhs.namespace,GammaSymbol))
```

```
            ,(@name.id,(@loc.vartype,@lhs.namespace,GammaClass))
            ,("This",(self ct,@loc.namespace,GammaSymbol))
            ,("this",(self ct,@loc.namespace,GammaSymbol))
            ] where ct = partExpand @loc.envs @loc.vartype
      loc.this = self (partExpand @loc.envs @loc.vartype)
      parameters.environments = @typeparameters.environments
      variables.environments = @parameters.environments
      methods.environments = @variables.environments
      loc.environments = @methods.environments
      lhs.environments = @loc.environments
      loc.namespace = @lhs.namespace ++ [@name.id]

SEM MaybeMethods
    | Nothing
      lhs.environments = @lhs.environments

SEM Methods
    | Nil
      lhs.environments = @lhs.environments
    | Cons
      tl.environments = @lhs.environments
      hd.environments = @tl.environments
      lhs.environments = @hd.environments

SEM Method
    | Method
      signature.environments = @lhs.environments
      variables.environments = @signature.environments
      code.environments = @variables.environments
      loc.environments = @code.environments
      lhs.environments = @loc.environments
      signature.namespace = @lhs.namespace
      variables.namespace = @signature.namespace
      code.namespace = @signature.namespace

SEM MaybeSignatures
    | Nothing
      lhs.environments = @lhs.environments

SEM Signatures
    | Nil
      lhs.environments = @lhs.environments
    | Cons
      hd.namespace = @lhs.namespace
      tl.namespace = @lhs.namespace
      tl.environments = @lhs.environments
      hd.environments = @tl.environments
      lhs.environments = @hd.environments

SEM Signature
    | Signature
      (loc.environments,loc.error1)
        = addSymbols @lhs.environments
                     [(@name.id, (@loc.type ,@lhs.namespace,GammaSymbol))
```

```
                    ,("caller", (@loc.caller,@loc.namespace,GammaSymbol))
                    ]
      in.environments = if elem @name.id ["run"] - Don't put "run" in namespace
                        then @lhs.environments
                        else @loc.environments
      out.environments = @in.environments
      lhs.environments = @out.environments
      loc.namespace = @lhs.namespace ++ [@name.id]
      in.namespace = @loc.namespace
      out.namespace = @loc.namespace
      lhs.namespace = @loc.namespace

SEM MaybeDeclarations
    | Nothing
      lhs.environments = @lhs.environments

SEM Declarations
    | Nil
      lhs.types = []
      lhs.environments = @lhs.environments
    | Cons
      tl.environments = @lhs.environments
      hd.environments = @tl.environments
      lhs.environments = @hd.environments
```

208 ⟨*Symbol Table Semantic Functions* 205⟩+≡ (161b) ◁205

```
SEM Declaration
    | Var
      (loc.environments,loc.error1)
        = addSymbol @lhs.environments
                    (@name.id,(@loc.type,@lhs.namespace,GammaSymbol))
      lhs.environments = @loc.environments

SEM MaybeTypeDeclarations
    | Nothing
      lhs.environments = @lhs.environments

SEM TypeDeclarations
    | Nil
      lhs.types = []
      lhs.environments = @lhs.environments
    | Cons
      tl.environments = @lhs.environments
      hd.environments = @tl.environments
      lhs.environments = @hd.environments

SEM TypeDeclaration
    | Var
      (loc.environments,loc.error1)
        = addSymbol @lhs.environments
                    (@name.id
                    ,(@loc.type,@lhs.namespace,GammaSymbol)
                    )
      lhs.environments = @loc.environments

SEM Statement
    | Binary
      left.environments = @lhs.environments
      right.environments = @left.environments
      lhs.environments = @right.environments
    | Unary
      guard.environments = @lhs.environments
      statement.environments = @lhs.environments
      lhs.environments = @statement.environments
    | Nullary
      statement.environments = @lhs.environments
      lhs.environments = @statement.environments
    | Assign
      lhs.environments = @lhs.environments
    | While
      condition.environments = @lhs.environments
      statement.environments = @lhs.environments
      lhs.environments = @statement.environments
    | Call
      lhs.environments = @lhs.environments
    | Send
      (loc.environments,loc.error1)
        = addSymbol @lhs.environments
                    (@label.id
```

```
                    ,(@loc.methodtype,@lhs.namespace,GammaSymbol)
                    )
      lhs.environments = @loc.environments
    | Receive
      lhs.environments = @lhs.environments
    | Await
      lhs.environments = @lhs.environments
    | Wait
      lhs.environments = @lhs.environments
    | Expression
      lhs.environments = @lhs.environments

SEM Expression
    | If
      condition.environments = @lhs.environments
      statement.environments = @lhs.environments
      elsepart.environments = @statement.environments
      loc.environments = @elsepart.environments
      lhs.environments = @loc.environments
```

E.8 Code Machine Code Generation

E.8.1 Creol Machine Code Attributes

The following is a first attempt at an Attribute Grammar for these steps

209 ⟨*Creol Machine Code Attributes* 209⟩≡ (161b)

```
ATTR *
     [ - Inherited
     | - Chained
     | - Synthesized
       cmc USE {++} {""} : String
     ]

ATTR MaybeDeclarations Declarations Declaration
     Signature
     [ - Inherited
     | - Chained
     | - Synthesized
       init USE {++} {""} : String
     ]
```

E.8.2 Creol Machine Code Semantic functions

210 ⟨*Creol Machine Code Semantic functions* 210⟩≡ (161b)

```
SEM Ast
    | Ast lhs.cmc = "in CreolVirtualMachine" ++
                    "\nfmod CreolMachineCode is" ++
                    "\n pr CreolVirtualMachine ." ++
                    "\n op cmc : -> Configuration . " ++
                    "\n eq cmc = " ++
                    "(new 'Main (nil)) " ++
                    @classes.cmc ++
                    "\n . " ++
                    "\nendfm" ++
                    "\nrew cmc .\n"

SEM Classes
    | Cons lhs.cmc = consSep "\n" @hd.cmc @tl.cmc

SEM Class
    | Class lhs.cmc = "\n< " ++ @name.cmc ++
                      (cmcc @loc.error) ++ " : Cl | " ++
                      "\n Inh: " ++
                       (xorPrint
                        (insertSep ", "
                          [ @inherits.cmc
                          ]
                        )
                        "nil"
                       ) ++ ", " ++
                       "\n Att: " ++
                        (xorPrint
                          (insertSep ", "
                            [  @parameters.init,
                               @variables.init
                            ]
                          )
                          "no"
                        ) ++ ", " ++
                        @methods.cmc ++
                        "\n Ocnt: 1 " ++
                        "\n>"

SEM MaybeMethods
    | Nothing
      lhs.cmc = "\n Mtds: none,"
    | Just lhs.cmc = "\n Mtds: " ++
                     @methods.cmc ++
                     "\n , "

SEM Methods
    | Cons
      lhs.cmc = consSep " * " @hd.cmc @tl.cmc
```

```
SEM Method
    | Method
      lhs.cmc =      "\n < '" ++ @signature.id ++
                     (cmcc @loc.error) ++
                     " : Mtdname | " ++
                     "Latt: " ++
                      (xorPrint
                        (insertSep " , "
                          [ @signature.init,
                            @variables.init
                          ]
                        )
                        "no"++
                      )
                     ", " ++
                     "\n  Code: " ++
                     (combineSep " ; "
                       @code.cmc
                       " end( "
                     ) ++
                     (xorPrint @signature.cmc "nil") ++
                     ")\n >"

SEM Signature
    | Signature
      lhs.cmc = @out.cmc
      lhs.init = insertSep ", " [@in.init, @out.init]

SEM Statement
    | Binary
      lhs.cmc
       =transform @left.cmc @operator @right.cmc
         where
          transform l o r
            = "(" ++ l ++ " " ++ o ++ " " ++ r ++ ")" ++
              (cmcc @loc.error)
{-    | Unary
      lhs.cmc = "(" ++
                @guard.cmc ++ "->" ++
                @statement.cmc ++
                ")" ++ (cmcc @loc.error)
-}
    | Nullary
      lhs.cmc = "(" ++ @statement.cmc ++ ")"
    |  Assign
      lhs.cmc = "(" ++ @name.cmc ++
                " := " ++ @expr.cmc ++ ")" ++ (cmcc @loc.error)
    | While
      lhs.cmc = "(while " ++
                @condition.cmc ++
                " do " ++
                @statement.cmc ++
                " od)" ++ (cmcc @loc.error)
    | Call
```

```
      lhs.cmc = "(" ++ @object.cmc ++
                " . " ++ @method.cmc ++
                "(" ++ @in.cmc ++ " ; " ++ @out.cmc ++ ")" ++
                ")" ++ (cmcc @loc.error)
    | Send
      lhs.cmc = "(" ++ @label.cmc ++ " ! " ++
                (transform @object.cmc) ++
                @method.cmc ++
                "(" ++ @in.cmc ++ ")" ++
                ")" ++ (cmcc @loc.error)
                 where
                  transform "'this" = ""
                  transform o = o ++ "."
    | Receive
      lhs.cmc = "(" ++ @label.cmc ++
                " ?(" ++ @vars.cmc ++ ")" ++
                ")" ++ (cmcc @loc.error)
    | Await
      lhs.cmc = "(await " ++ @condition.cmc ++ ")" ++
                (cmcc @loc.error)
    | Wait
      lhs.cmc = "(await wait)"

SEM MaybeLabel
    | Nothing
      lhs.cmc = " '" ++ @loc.id

SEM Label
    | Label
      lhs.cmc = " '" ++ @label

SEM MaybeExpressions
    | Nothing
      lhs.cmc = "nil"
    | Just
      lhs.cmc = (xorPrint @expressions.cmc "nil")

SEM Expressions
    | Cons
      lhs.cmc = consSep ", " @hd.cmc @tl.cmc

SEM Expression
    | Binary
      lhs.cmc = "(" ++ @left.cmc ++
                 @loc.cmc ++
                 @right.cmc ++ ")" ++ (cmcc @loc.error)
      loc.cmc = case (@left.type,
                      @operator ,
                      @right.type
                     ) of
                      (IntType, "+", IntType) -> " + "
                      (StrType, "+", StrType) -> " cat "
                      (_, "+", _) -> " + "
                      (_, "-", _) -> " - "
```

```
                    (_, "*", _) -> " * "
                    (_, "/", _) -> " / "
                    (_, "^", _) -> " ^ "
                    (_, "<", _) -> " < "
                    (_, "<=", _) -> " <= "
                    (_, ">", _) -> " > "
                    (_, ">=", _) -> " >= "
                    (_, "=", _) -> " = "
                    (_, "/=", _) -> " /= "
                    (_, "!=", _) -> " /= "
                    (_, "and", _) -> " and "
                    (_, "or", _) -> " or "
    | Unary
      lhs.cmc = (transform @operator) ++ @expression.cmc
                where
                 transform operator
                   = case (operator)
                     of ("-") -> " neg "
                        ("not") -> " not "
    | Null
      lhs.cmc = " empty"
    | Variable
      lhs.cmc = @name.cmc ++
                (cmcc @loc.error)
    | Int
      lhs.cmc = " int(" ++ (show @value) ++ ")"
    | Bool
      lhs.cmc = " bool(" ++
                @loc.cmc ++
                ")"
      loc.cmc = case @value
                of True -> "true"
                   False -> "false"
    | String
      lhs.cmc = " str(\"" ++ @value ++ "\")"
    | New
      lhs.cmc = " new " ++ @constructor.cmc ++
                "(" ++ @in.cmc ++ ")" ++
                (cmcc @loc.error)
    | If
      lhs.cmc = "(if " ++
                @condition.cmc ++
                " th " ++
                @statement.cmc ++
                " el " ++
                @elsepart.cmc ++
                " fi)" ++ (cmcc @loc.error)

SEM MaybeVarIds
    | Nothing
      lhs.cmc = "nil"
    | Just
      lhs.cmc = (xorPrint @varids.cmc "nil")
```

```
SEM VarIds
    | Cons
      lhs.cmc = consSep ", " @hd.cmc @tl.cmc

SEM VarId
    | VarId
      lhs.cmc = "'" ++ @varid

SEM MaybeTypeIds
    | Nothing
      lhs.cmc = ""
    | Just
      lhs.cmc = ""

SEM TypeIds
    | Cons
      lhs.cmc = consSep ", " @hd.cmc @tl.cmc

SEM TypeId
    | TypeId
      lhs.cmc = "'" ++ @typeid

{-
SEM TypeDeclarations
    | Cons


SEM TypeDeclaration
    | Var
      lhs.cmc = @name.cmc
-}

SEM Declarations
    | Cons
      lhs.cmc = consSep ", " @hd.cmc @tl.cmc
      lhs.init = consSep ", " @hd.init @tl.init

SEM Declaration
    | Var
      lhs.cmc = @name.cmc
      lhs.init = "(" ++ @name.cmc ++ " : " ++
                 (xorPrint @default.cmc "null") ++
                 ")" ++
                 (cmcc @loc.error)
```

E.8.3 Unique Labels

215 ⟨*Unique Labels* 215⟩≡ (161b)

```
- Globally Unique Label
ATTR Classes MaybeMethods Methods Method Statement Expression
     Class Interface Interfaces Expressions MaybeExpression MaybeExpressions
     Declaration Declarations MaybeDeclarations Signature Signatures
     MaybeSignatures MaybeLabel
     [ - Inherited
     | - Chained
       unique : Int
     | - Synthesized
     ]

SEM Ast
    | Ast
      loc.unique = 1
      interfaces.unique = @loc.unique
      classes.unique = @interfaces.unique

SEM MaybeLabel
    | Nothing
      loc.unique = @lhs.unique + 1
      lhs.unique = @loc.unique
      loc.id = "UniqueLabel" ++ (show @loc.unique)
      lhs.id = @loc.id
```

E.9 Auxiliary Functions

216 ⟨*Auxiliary Functions* 216⟩≡ (161b)

```
{
- Combinator to apply functions to recursive tuples
- Such as built by successive use of <+>
- (<->) :: (IsParser p s) => p f -> p a -> p (f a b)
pf <-> pt = tupply <$> pf <*> pt
f <$-> pt = (tupply f) <$> pt
f <$$-> pt = (tupplysec f) <$> pt

- Apply function to elements of tuple
tupply f (a,b) = (f a) b
tupplysec f (a,b) = \x -> ((f x) a) b
tripply f (a,b,c) = ((f a) b) c

- Combine with separator
combineSep _ [] [] = []
combineSep _ [] l = l
combineSep _ l [] = l
combineSep sep l1 l2 = l1 ++ sep ++ l2

- Insert separator between list elements
- Discard empty items
insertSep sep list = foldl (combineSep sep) "" list

- : with separator between
consSep _ h [] = h
consSep sep [] t = t
consSep sep h t = h ++ sep ++ t

- : with separator and item prefix
consPrefixSep prefix _ [] [] = []
consPrefixSep prefix _ h [] = prefix ++ h
consPrefixSep prefix sep h t = prefix ++ h ++ sep ++ t

- Print with prefix and suffix, if present
preSuffix _ [] _ = []
preSuffix prefix item suffix = prefix ++ item ++ suffix

- Print one of the alternatives, preferrably the first.
xorPrint [] b = b
xorPrint a _ = a

- Parse key and return tuple with key and pos
pKeyPosTuple key = (\x->(key,x)) <$> pKeyPos key

- Print text as CMC Comment
cmcc :: String -> String
cmcc "" = ""
cmcc s =  preSuffix " ***(" s ")\n"

findError :: Envs -> Pos -> Type -> String -> (Envs,String)
findError envs pos vartype otherError
```

```
  = (envs',error)
    where (envs',t') = replaceType envs (num vartype) t
          (t,error) = case t' of
            ErrorType{}
              -> (AnyType{former=(former t')}
                  , (show pos) ++ ": " ++ (message t') ++
                   - " from symboltable "  ++ (show (complete e)) ++
                   (preSuffix ", " otherError "")
                  )
            t -> (t, (preSuffix ((show pos) ++ ": ") otherError ""))

catchError :: Pos -> Type -> Envs -> String -> (Type,String)
catchError pos ErrorType{message=msg,former=t} e otherError
  = (AnyType{former=t}, (show pos) ++ ": " ++ msg ++
                - " from symboltable "  ++ (show (complete e)) ++
                (preSuffix ", " otherError "")
             )
catchError pos t _ otherError
  = (t, (preSuffix ((show pos) ++ ": ") otherError ""))

}
```

Appendix F

Code Macros

⟨*Abstract Syntax Tree Structure* 162⟩ 161b, 162
⟨*Auxiliary Functions* 216⟩ 161b, 216
⟨*Creol Machine Code Attributes* 209⟩ 161b, 209
⟨*Creol Machine Code Semantic functions* 210⟩ 161b, 210
⟨*CreolCompiler.ag* 161a⟩ 161a, 161b
⟨*highlevel.code* 158b⟩ 158b
⟨*library.code* 158f⟩ 158f
⟨*Main.hs* 159a⟩ 159a, 159b, 159c, 160a, 160b
⟨*Parametrised Scanner* 166e⟩ 161b, 166e
⟨*Parser* 167⟩ 161b, 167, 171
⟨*part 1 of this code* 158c⟩ 158b, 158c
⟨*part 2 of this code* 158d⟩ 158b, 158d
⟨*Patch for Scanner.hs* 165⟩ 165
⟨*program.code* 157⟩ 157, 158a
⟨*Scanner Operator Characters* 166b⟩ 166b, 166e
⟨*Scanner Operator Keywords* 166c⟩ 166c, 166e
⟨*Scanner Special Characters* 166a⟩ 166a, 166e
⟨*Scanner String Keywords* 166d⟩ 166d, 166e
⟨*some.code* 158e⟩ 158e
⟨*Symbol Table Attributes* 204⟩ 161b, 204
⟨*Symbol Table Definitions* 200⟩ 161b, 200
⟨*Symbol Table Semantic Functions* 205⟩ 161b, 205, 208
⟨*Type Definitions* 175⟩ 161b, 175, 178
⟨*Typechecker Attributes* 179⟩ 161b, 179, 180a, 180b
⟨*Typechecker Semantic Functions* 181⟩ 161b, 181
⟨*Unique Labels* 215⟩ 161b, 215