Enhancing a Modular Effectful Programming Language

Lukas Convent

Master of Science
School of Informatics
University of Edinburgh
2017

Abstract

The focus of this thesis lies on several enhancements to Frank, a programming language that supports algebraic effects and effect handlers as core features. Effects serve as interfaces between command-requesting and command-handling components, thereby offering the programmer a modular and powerful means of abstraction.

We improve the expressiveness of commands by making them polymorphic. Together with an extension of ML-style references, we can demonstrate the gained potential by implementing two concurrency models (actors, promises) within Frank.

A prominent feature of Frank is the *ambient ability* which allows convenient composition of effect generators and effect handlers. We discuss different scenarios of composition and identify a problem with intermediate effects that are unintentionally exposed. We solve part of the problem and discuss how to solve the more general scenario.

Another prominent feature of Frank are multihandlers that allow handling of multiple effects simultaneously. A dynamic semantics of this setting has been given so far via a translation to a setting of unary handlers. We give an operational small-step semantics that is more direct and closer to the actual implementation. Furthermore, we prove type soundness of the enhanced Frank version.

Acknowledgements

Firstly, I would like to greatly thank Sam for giving me the possibility to write this thesis, discussing ideas with me and supporting me throughout the whole process. A thank-you goes also to Craig for answering my questions about the Frank compiler and to Daniel for giving me a jump start on Links. Finally, I would like to thank my brother Simon and my friend Pinak for proof-reading an earlier draft of this thesis.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Lukas Convent*)

Table of Contents

Chapter 1

Introduction

In this thesis, we focus on several enhancements to Frank [20], a functional programming language that supports algebraic effects and effect handlers as core features. We discuss composition of effectful components, examine connected problems and solve a part of it. Furthermore, we add polymorphic commands to Frank as well as ML-style references and demonstrate the gained expressiveness by implementing two concurrency models (actors, promises) *within* Frank. The dynamic semantics of Frank has so far been given via a translation to a core calculus; by giving an operational small-step semantics we provide an additional, more direct formalisation that is closer to the actual implementation. Furthermore, we prove type soundness for the enhanced version of Frank.

We base our enhancements directly on work by Lindley, McBride and McLaughlin in two ways, firstly by extending the formal syntax and semantics of Frank [20], and secondly by extending the existing Frank compiler [21] which is written in Haskell. In the following we give a short primer on effect handling, then mention some of Frank's characteristics and how they connect to our work. After reviewing related work done in this field, we summarise our main contributions.

From Exceptions in Java to Effects in Frank. Algebraic effects serve as an interface between effect generators and effect handlers. Examples include state, I/O and non-deterministic choice. A widely used example of effects are exceptions (as in Java) where they are generated via `throw` and handled via `try...catch`. The handler gears into a potentially exception-generating computation (`try` block) by either encountering the end of the computation (the end of the `try` block) or receiving an `Exception` object and dealing with it in a corresponding `catch` block. Below is an example that demonstrates how exception handling can occur in Frank, as a particular instance of effect handling.

```
1  parseContact: {String -> [Malformed]Contact}
2  parseContact s = ... if (hasEmailFormat s')
3                       { stringToEmail s'  }
4                       { malformedEmail s' } ...
5
6  handleBlank: {<Malformed>Contact -> Contact}
7  handleBlank c = c
8  handleBlank <malformedEmail s -> k> = blankContact
9
10 getContact: {String -> Contact}
11 getContact s = handleBlank (parseContact s)
```

The example gives the definitions of three operators. The `parseContact` operator parses a `Contact` from a `String` which involves the parsing of emails. If an email string is of the correct format, it is cast to an `Email` (*line 3*). Else, a `malformedEmail` effect is exhibited (*line 4*). The `handleBlank` operator behaves like `try...catch` in Java, with its argument being the computation to try.

Comparing the argument's type (`<Malformed>Contact`) with the return type of `parseContact` (`[Malformed]Contact`) reveals their match, allowing `handleBlank` to be applied to `parseContact s` (*line 11*). When applied, `handleBlank` behaves like `try...catch` in the following sense: If no exception is thrown, the computed contact is returned ("`try` block successfully executed", *line 7*). If an exception is thrown, a `blankContact` is returned ("`catch` block executed", *line 8*). The parameter `s` (the string containing the malformed email) is ignored as well as the continuation parameter `k`, which we come back to shortly.

How is effect handling more general than exception handling? In Java, imagine that upon receipt of an `Exception` object from within the `try` block, the `try...catch` handler additionally has access to a continuation that allows resuming execution of the `try` block as part of its handling behaviour. This is the essence of effect handling: The handler gears into the computation by handling a request (here: exceptions) and then giving back control to the suspended continuation as it "wants to". To demonstrate the use of continuations in Frank, we first reveal how the `Malformed` effect is defined and then focus on a second handler.

```
1  interface Malformed = malformedEmail: String -> Email
2
3  handleDefault: {<Malformed>Contact -> Contact}
4  handleDefault c = c
5  handleDefault <malformedEmail s -> k> = handleDefault
6                                          (k (toEmail "m@lform.ed"))
7
8  getContact': {String -> Contact}
9  getContact' s = handleDefault (parseContact s)
```

The `Malformed` effect allows one command `malformedEmail`, in accordance with the first example. Similar to the `handleBlank` handler, `handleDefault` is also able to "catch" the `Malformed` effect, but in this case it passes a default `Email` (representing `m@lform.ed`) to `k`. The continuation `k` takes exactly one argument whose type corresponds to the return type of the command `malformedEmail`. The `Contact`-parsing computation can thus go on with an alternative `Email` and is not discarded in favour of a `blankContact` as in the last example. Applying the alternative `Email` to the continuation results in a computation, which may exhibit further effects and therefore is passed to the same handler again (*lines 5, 6*). Only when the computation is finished, the final `Contact` is returned (*line 4*).

Another feature of Java's exception handling is desirable for general effect handling, namely a strong support by the type system. In Java, exception-throwing methods must be annotated with a `throws` clause that lists the potentially generated exceptions. When calling such a method, handling of the listed exceptions is statically enforced. In a similar way, Frank enforces *effect safety* (no effect remains unhandled) by matching the exhibitable effects (`[Malformed]`) with the handled effects (`<Malformed>`). Let us take this aspect of effect safety along to the next section where we focus on the aspects of Frank that we base our main contributions on.

Building upon Effects in Frank. A prominent feature of Frank is the ambient ability. An ability is a collection of effects with an optional effect variable, and the ambient ability describes the effects that are handled at a particular point in the program. By composing handlers, the ability increases as each handler adds further effects as "offers" to it. A simple example of how the ambient ability comes into play is found in `getContact'` (and `getContact`) above. At the point where the sub-term `parseContact s` is type-checked, the ambient ability includes `Malformed` as offered by `handleDefault` (*line 9*). Type-checking succeeds because it matches the effects exhibited by `parseContact s`. As part of our work, we examine different scenarios for composing handlers and point out the shortcoming that intermediate effects can unintentionally be exposed. We discuss remedies and present a solution for a part of the problem.

Frank supports polymorphic effect interfaces. In the following example, `BinChoice` is polymorphic in `X` and `BinChoice Int` defines the command `binChoose` that requests a choice between two integers. Once the interface is parameterised, the command `binChoose` itself is not polymorphic. We extend Frank by polymorphic commands which allows the implicit parameterisation *on each command request*. In the following example, `BinChoice'` is not polymorphic but its

command `binChoose'` is. For contrast, we prepend the definition of an interface `BoolChoice` with no polymorphism at all.

```
interface BoolChoice = boolChoose: Bool
interface BinChoice X = binChoose: X -> X -> X
interface BinChoice'  = binChoose' X: X -> X -> X
```

Shifting polymorphism down to the commands is of course not always possible, but in this case it adds more flexibility at no cost. As shown before [14], polymorphic commands integrate naturally and are rather easy to implement, which we can confirm for Frank.

Another prominent feature of Frank is that functions and handlers are conjoined in the notion of a multihandler. Multihandlers are simultaneously applied to multiple arguments (like n-ary functions) and simultaneously handle their effects. When specifying Frank [20], Lindley et al. give Frank's dynamic semantics via a translation to Core Frank for which they specify an operational small-step semantics. Core Frank is a standard calculus that supports functions and unary handlers but no multihandlers. The translational approach of giving a dynamic semantics has the advantage of revealing the relation between the two settings but is not as direct as describing reduction steps of Frank terms themselves. We give a direct operational small-step semantics together with proofs of type preservation and progress. The two properties are given for the enhanced version of Frank, thus guaranteeing their soundness.

Related Work. Frank has been plotted since 2007 by McBride [29], implemented as a prototype for a previous version of Frank in 2012 by McBride [28, 30], reimplemented since 2016 and presented in 2017 by Lindley, McBride and McLaughlin [20, 21]. We give an overview of further research done on effect handling, relying on a short review we compiled previously [7].

Although effect systems have been around for a while (e.g. Lucassen and Gifford's polymorphic effect systems [26] in 1988), Plotkin and Power's algebraic effects [37, 34, 36, 35] only recently gave rise to Plotkin and Pretnar's effect handlers [33] in 2009. Plotkin, Power and Pretnar give a category-theoretic account of how the notion of algebraic effects can be integrated into the monadic semantics of the λ-calculus. Since then, multiple language extensions and languages have been developed.

Kammar at al. [14] build on Plotkin and Pretnar's work by giving language extensions for Haskell, OCaml, SML and Racket that introduce effect handling. Furthermore, these libraries are formally underpinned by a core calculus λ_{eff} with operational small-step semantics and are accompanied by examples which demonstrate the power of effectful programming in different settings.

Brady [3] shows how algebraic effects can be integrated in the context of the programming language Idris which supports dependent types. A key feature of his work is that the type of effect resources (e.g. the resource of a state) can be transformed by a computation. This way, computations can be confined to transition systems induced by type signatures. Brady further builds upon his work by integrating dependent types into the effect system [4]. The key idea here is that the type of an effectful computation is *dependent* on its outcome, allowing the specification of even finer-grained transition systems.

Kiselyov et al. [15] present a library for extensible effects in Haskell. They tackle the problem of the linearity that is induced by expressing effects in terms of monad transformers. Instead of stacking up monads through monad transformers, their extensible effects provide open unions, which represent unordered coproducts of monads. These open unions can be added to and removed from, depending on where handling occurs.

Leijen [17] introduces the effectful programming language Koka and later extends the language by effect handlers [18]. Koka uses row polymorphism to combine effects and provides a Hindley-Milner-style automatic type inference, also for effect types. Similarly to Frank, Koka makes a clear distinction between values and (potentially effectful) computations. Koka employs deep handlers (unlike Frank's shallow handlers) but allows multi-shot continuations (like Frank).

Dolan et al. [8] introduce algebraic effects in OCaml as a new language feature and demonstrate its usefulness by an interface to Multicore OCaml, allowing the encapsulation of concurrency primitives via effects. OCaml employs deep handlers (unlike Frank's shallow handlers)

and only allows one-shot continuations (unlike Frank's multi-shot continuations). Due to being embedded into an already-existing OCaml type system, the type system does not enforce type safety to the degree that Frank does.

Bauer and Pretnar [1, 2] present the programming language Eff which supports algebraic effects, along with denotational semantics, a prototype implementation and a presentation of useful programming techniques. Effects are defined and then *instantiated*, e.g. a state effect can be instantiated multiple times to model distinct states. Furthermore, their effect system supports local referentiable state.

Hillerström [12] and Hillerström and Lindley [13] give an implementation of effect handlers using row polymorphism for the programming language Links. They argue that row polymorphism is exactly the typing discipline needed to compose effects, solving the problem of rigid orderings of effects, which occurs for example when expressing effects in terms of monad transformers.

Dolan et al. [9] explore further applications of effect handling to concurrency (e.g. by providing an implementation of promises, a concurrency model we implement, too). They carry their work out in Multicore OCaml.

Contributions. We make the following contributions.

- **Operator Composition.** We discuss scenarios of handler composition in Frank and come to the conclusion that intermediate effects can violate encapsulation of components (Ch. 4.2). We define *equivalence-up-to-inactive-instantiations* on abilities and integrate it in the typing rules, which solves part of the problem (Ch. 4.3). We discuss possible approaches for solving the rest of the problem, including an *effect-hiding composition operator* (Ch. 4.4).

- **Polymorphic Commands and Generative Effects.** We have added polymorphic commands formally and as an extension to the compiler (Ch. 5). Among the gained potential, this allows the extension by ML-style referentiable state (Ch. 7.1). We examine the potential of the new features and give two concurrency models (actors, promises) as examples, implemented within Frank (Ch. 7.3, 7.4).

- **Operational Semantics.** We provide an operational small-step semantics of Frank (Ch. 3.3) and prove type preservation and progress for the enhanced Frank version (Ch. 6). While the dynamic semantics via a translation to Core Frank which was given before [20] focuses on the relationship between Frank and a standard unary-handler language, the new operational semantics is more direct and closer to the actual implementation.

- **Technical Enhancements and Fixes.** We implement interface aliases as a way of grouping together interfaces, similar to the way in which type aliases can abbreviate composed types (Ch. 8.1). We also implement a procedure that resolves implicit effect variables while respecting interdependencies, a procedure that has not been present in the compiler before and therefore used to cause a bug (Ch. 8.2). Error-reporting has so far been only rudimentary; we discuss how nodes of the abstract syntax tree can be enriched by meta-information like source code locations and present our implementation based on fixed-points of type constructors (Ch. 8.3).

Chapter 2

The Frank Programming Language

Frank is a functional programming language which focuses on the definition, composition and use of algebraic effects. To this end, it uses the *ambient ability* to incrementally compose effects, *multihandlers* which control evaluation in a fine-grained way, and a clear distinction between computation and value as well as the instruments to switch between these levels.

In the following, we first give an overview of Frank from the perspective of a functional programmer to present its key features. Although it covers about the same content as the tutorial of Lindley et al.'s paper[20], it is entirely written from scratch and focuses more concisely on core elements of Frank[1]. In the next chapter then, we formally introduce Frank's syntax, its type system and its operational semantics. In both chapters, we present the original version of Frank (as specified [20] and implemented [21]) and only briefly point out differences to the current version (as specified in this thesis and implemented [23]).

2.1 Programming Without Effects

Although effect handling is a core feature of Frank, it can also be seen as an extension to a language that does not deal with effects in the first place. We thus start introducing Frank by a brief walk-through from the perspective of a functional programmer and then go into more detail when presenting effect handling.

Algebraic and Primitive Data Types. Let us start by looking at data types. Algebraic data types can be defined in a standard way.

```
data Bool = false | true
data Maybe X = nothing | just x
data Unit = unit
data List X = nil | cons X
```

The algebraic data types `Unit` and `List` (with syntactic sugar enabling expressions like `a::[b,c]`) are built-in, among primitive data types are `Int` and `Char`, and `String` is a type synonym for `List Char`.

Operators. Frank generalises the notion of a function to the notion of an operator in three ways: Operators 1) can handle effects, 2) are n-ary, i.e., n arguments are applied simultaneously which plays a key role in effect handling, 3) comprise 0-ary computations, so-called thunks.

For now we focus on the aspects of operators that are not directly related to effect handling by explaining how pure (i.e., effect-free) functions integrate into Frank. Let us consider the definition of `map`.

```
map: {{X -> Y} -> List X -> List Y}
map f [] = []
map f (x::xr) = f x :: map f xr
```

[1] Being more elaborate e.g. on multihandling and on comparison to Haskell, it can be insightful in addition to this overview.

Operators are implicitly parameterised by any occurring type variables (here: `X`, `Y`) and operator application is written as usual (here e.g.: `f x`). The only non-standard aspect of these definitions are the curly braces in the type signature. They denote operator types which we classify in the following.

Value Types. There are two different categories of types, namely *value types* which correspond to the standard notion of types and *effect types* which are entirely different and which we come back to later. We distinguish value types between:

1. *Data types*, like `Bool`
2. *Operator types*, which we distinguish between
   (a) *Multihandler types*, like `{<Effect>X -> <Effect>Y -> X}`
   (b) *n-ary function types*, like `{X -> Y}` or `{{X -> Y} -> List X -> List Y}`
   (c) *Thunk types*, like `{Bool}`

The notions of operator types, multihandler types, n-ary function types and thunk types are listed in a subsuming order in the sense that e.g. an n-ary function type `{X -> Y}` is also a multihandler type (without making use of the multihandlers' handling potential). We mostly use "operator" as a term when we generally speak of multihandlers, functions or thunks. For now we leave aside the handling potential of multihandlers and focus on functions and thunks.

Functions. Functions of arrow types in functional programming languages such as Haskell are present as n-ary functions in Frank, e.g. `map`. While functions in Haskell can be partially applied (if they are curried), n-ary functions in Frank allow only "full" application in order to trigger reduction. Although embedded into the more general setting of effect handling, reduction follows a left-to-right call-by-value scheme. We will come back to operator application when explaining the handling mechanism.

Thunks. Thunks are n-ary function for the special case of $n = 0$, i.e., they don't expect arguments. Thunks are suspended computations and need to be triggered in order to evaluate. We call this triggering *forcing* which can be viewed as the 0-ary application of a thunk and is denoted by the `!` suffix. The following example shows how Frank's clear separation between value and computation allows the elegant definition of control structures such as `if`.

```
if: {Bool -> {X} -> {X} -> X}
if true  t _ = t!
if false _ f = f!
```

Here, `if` takes a consequence `t` and an alternative `f` as thunks and only after examining the boolean condition, forces *either* `t` or `f`. An example of Lindley et al. [20] where premature evaluation would be fatal demonstrates a use of `if`.

```
if fire! {launch missiles} {unit}
```

Note that premature evaluation would only be fatal if `launch` actually causes effects. A consequence of the curly braces is that the use of such control structures bears similarity to C syntax.

Anonymous Operators. Similar to anonymous functions (i.e., lambda expressions) in standard functional programming languages, Frank provides anonymous operators. We start with the anonymous thunk, because it is so simple: Given a term `t` of type `T`, an anonymous thunk is given by `{t}` of type `{T}`.

Anonymous multihandlers are given the same pattern-matching construct as present in multihandler definitions. Consider the following anonymous multihandler of type `{Bool -> Bool}` that computes boolean negation.

```
{ false -> true
| true  -> false }
```

It could be used in combination with `map` like this:

```
map {false -> true | true -> false} [true, false]   ⇝   [false, true]
```

Anonymous multihandlers are very expressive in that they comprise ad-hoc-ness (by being anonymous) and pattern-matching at the same time. This allows us to have the standard case-expression construct (as present e.g. in Haskell) available in terms of anonymous multihandlers.

```
case: {X -> {X -> Y} -> Y}
case x f = f x
```

For an example of how `case` can be used, consider the following operator `first` which takes a list of thunks that `Maybe` contain a value and forces them until one of them computes to `just` a value. If none of them does so, `nothing` is returned.

```
first: {List {Maybe X} -> Maybe X}
first []      = nothing
first (x::xr) = case x! { (just x) -> just x
                        | nothing  -> first xr }
```

Let-expressions and Sequences. Using the `case` operator, we can enable (monomorphic) **let-expressions** as syntactic sugar. Let-expressions are desugared as follows: `let x = t in t'` := `case t in {x -> t'}`. Another standard syntactic construct that can be defined in terms of `case` is sequencing. A *sequence* `t; t'` has semantics as expected: First `t` is evaluated and discarded, then `t'` is evaluated and returned. Sequences can thus be desugared as follows: `t; t'` := `case t {_ -> t'}`.

2.2 Programming With Effects

Effects in Frank are interfaces for both requesting and handling certain operations. They are a means of abstraction in that effect-generating and effect-handling behaviour can be separated into components that are easy to compose. In particular, effects can deliberately be leaking outside of pure Frank and be handled e.g. via user input on the console. Let us stay in our pure Frank world for now and make effect handling more concrete by considering our first effect interface.

Defining Effects. A typical example of effects is state. Getting and putting a state are operations that are requested by a stateful program (that uses a state) and handled by a state-handling program (capable of managing state). Let us take a look at how the state effect interface is defined.

```
interface State X = get: X
                  | put: X -> Unit

interface Choice = choose: Bool
```

An *interface* comprises multiple *commands* (here: `get`, `put`), of which each contains a list of argument types and a result type (the right-most type). Interfaces can be parameterised by type variables (here: `X`). We call an instantiated interface like `State Bool` an *interface instantiation*. We also mention here the ambiguity of the term "effect", which may refer to "interface", "command" or other related terms; for this reason we aim to use the defined terminology whenever the context is not clear.

Generating Effects. Let us stick with the state example and have a look at how interfaces are used by considering the following example of an operator that takes a list and returns the same list but with every second element dropped.

```
1 dropEverySecond: {List X -> [State Bool](List X)}
2 dropEverySecond [] = []
3 dropEverySecond (x::xr) = put (not get!);
4                           if get! { x::(dropEverySecond xr) }
5                                   { dropEverySecond xr }
```

The operator reads the state and flips it (*line 3*) and then — depending on the state — drops or does not drop the current element before going on with the rest (*lines 4, 5*). Notice that although this operator may seem at first glance like an executable program, this is not the case. The `State Bool` interface instantiation (and therefore the commands `get`, `put`) are as yet uninterpreted.

Abilities and Pegs. Let us first focus on the operator's signature. Its result type is `List X`, and the prefix `[State Bool]` denotes that `dropEverySecond` requires the `State Bool` interface instantiation. We call a bracket-enclosed collection of such required instantiations an *ability* or an *effect type*[2]. Every operator has exactly one ability which — if it is explicit as in `dropEverySecond` (and unlike in `map`) — is always prefixed to the result type. We call the ability and the result type together the *peg* of an operator (here: `[State Bool](List X)`). The counterpart of a peg is a port, giving rise to the image of a peg slotting into a port if their value types and the exhibited effects (on the peg's side) and handled effects (on the port's side) match. Before we describe ports in more detail, let us move our focus to the operator's body. All commands of the interface instantiations requested in the peg's ability are available as operators. In our example, this means we have `get: {Bool}` and `put: {Bool -> Unit}`, which are both used.

Handling Effects. We have now an operator that exhibits `State Bool` effects and in order to use it, we need to provide a handler for it. There is a natural handler for state: It holds the value of the current state, returns it on every `get` and updates it on every `put`. Let us see what such a definition looks like.

```
evalState: {<State Y>X -> Y -> X}
evalState <get -> k>   s = evalState (k s) s
evalState <put s -> k> _ = evalState (k unit) s
evalState x            _ = x
```

The first two cases belong exactly to the described behaviour and in the third case, when no more effects are exhibited by the computation, the operator returns the final value. Let us go into more detail now, beginning with the type signature of `evalState`.

Adjustments and Ports. Similar to how we have to declare the required effects in the peg, we prefix the argument types by the effects (i.e., interface instantiations) that can be handled. We call such a list of interface instantiations an *adjustment* (here e.g.: `<State Y>`) and the adjustment (if explicit) and argument type together a *port* (here e.g.: `<State Y>X`). The term "adjustment" signifies an addition to the current ambient ability, which describes the effects that are handled at a particular point and which we will explain in detail later.

Ports specify not only that an operator takes values of a particular type as argument. Additionally, they also specify that an operator participates in the computation of such values by handling certain commands that arise during their computation. Only when the computation is finished a value binding will actually occur, in `evalState` this is covered by the last case.

Computation patterns. To handle the different commands, a new kind of pattern is introduced which comprises the command name, patterns for the command's arguments and a name for the continuation (here e.g.: `<put s -> k>`). The command's arguments are computed values and are provided during the handling of the case. The continuation is an operator that awaits the result type of the command and can be given to the handler at the same argument position again. Let us make this more concrete: In the first case of the `evalState` handler, `k: {Y -> [State Y]X}` and in the second case, `k: {Unit -> [State Y]X}`. The continuations potentially exhibit further

[2] "Ability" and "effect type" are two further terms that "effect" may refer to.

commands of `State Y`. This is why we can simply give the applied continuations again as arguments to `evalState` at the port that can handle commands of `State Y`.

Combining Effect Generation and Effect Handling. We now have all the ingredients we need to let the effects flow. Consider the following example of a `main` operator that applies the `dropEverySecond` operator to the list `[1, 2, 3, 4]`, resulting in `[1, 3]`.

```
main: {List Int}
main! = evalState (dropEverySecond [1, 2, 3, 4]) false
```

Because the interface instantiation `State Bool` which is required by `dropEverySecond` is handled by `evalState`, all effects are handled which leaves `main` with an implicit *empty* ability. However, this is only the short explanation and after the next paragraph we demonstrate how effect generation and effect handling interact.

Escaping Pure Frank. Let us briefly take a look at how Frank programs interact with the outside world. Generally, the `main` operator is the default entry point to a program and the returned value is displayed on standard output. What about any effects exhibited by `main`? Effects appearing in the peg of `main` must be handled from outside of the pure Frank, so these interfaces must be built-in and handled (translated) by the compiler. An example of such an interface is `Console` which allows commands for console input and output. We will see an example of this later and will now focus back on the handling mechanism.

Effect Variables. First of all, we extend our definition of ability. So far, we have defined an ability as a collection of instantiated interfaces, e.g. `[State Bool]` or `[State Bool, Choice]`. In addition to this collection, an ability can contain up to one *effect variable* E, explicitly written like this: `[E | State Bool]` or `[E | State Bool, Choice]`. An effect variable can be bound to an ability and such bindings occur every time effects are handled.

Implicitness. Before we go into detail about how effect variables come into play, let us take a look at why we did not come across them so far, although having seen a complete example of effect handling. The reason for this is implicitness, which acts on three levels: If not written explicitly, there are 1) implicit abilities, 2) implicit effect variables and 3) implicit adjustments. To illustrate these three points, we give the type signatures of `dropEverySecond`, `evalState` and `main` with the mentioned elements explicitly included. As the inclusion of implicit variables abandons the level of concrete syntax, we use formal notation from now on if appropriate.

$$\begin{aligned} \mathit{dropEverySecond} &: \{\langle\iota\rangle \mathit{List}\ X \to [\epsilon \mid \mathit{State}\ \mathit{Bool}](\mathit{List}\ X)\} \\ \mathit{evalState} &: \{\langle \mathit{State}\ Y\rangle X \to \langle\iota\rangle Y \to [\epsilon|]X\} \\ \mathit{main} &: \{[\epsilon|]\mathit{List}\ \mathit{Int}\} \end{aligned}$$

First, if a peg is given without explicit ability, it always has an *implicit ability* as in `evalState` and `main`. Second, if an ability is given without explicit effect variable, it is always assigned an *implicit effect variable* ϵ. In the case of both implicit ability and effect variable, the notation $[\epsilon|]$ is not particularly nice but unambiguous. Finally, if a port is given without explicit adjustment, it always has the *implicit empty adjustment* $\langle\iota\rangle$. Notice that within a particular signature, several occurrences of ϵ refer to the same variable, as is the case for `map` where the function passed as argument has the same ability as `map` itself.

$$\mathit{map}: \{\langle\iota\rangle\{\langle\iota\rangle X \to [\epsilon|]Y\} \to \langle\iota\rangle \mathit{List}\ X \to [\epsilon|]\mathit{List}\ Y\}$$

Globally though, every signature generally has fresh type variables, e.g. X and ϵ which are both referred to from `dropEverySecond` and `evalState` are distinct.

Type-Checking and Ambient Ability. Let us now examine how the type-checker makes sure that effect generators and effect handlers are tied together in a sensible way. Type-checking consists of two parts: First, value types have to be checked (e.g., `if` may only take `Bool` values as its first argument), which is standard. Second, effect types have to be checked. Frank introduces to this means the notion of an ambient ability. The *ambient ability* is the ability (i.e., available interface instantiations) that is available *at a specific point* in the term. By "*point*" we mean the sub-term that is currently checked. For each operator that a sub-term is nested within by operator application (possibly indirectly), the operator's ability is added to the ambient ability.

We demonstrate the evolution of the ambient ability by walking through a type-check of `main` and consider all sub-terms that need to be type-checked. The aim is to informally provide intuition for how checking effect types via the ambient ability works. We underline the currently-typed sub-term (the current *point* of type-checking) in the following.

1. `evalState (dropEverySecond [1, 2, 3, 4]) false` $\quad \Sigma := [\epsilon_1|]$

   The term's type is determined by the result type of `evalState`. So in order to type-check the term, we need to make sure that the peg of `main`, $[\epsilon_1|](\mathit{List}\ X_1)$, matches the peg of `evalState`, $[\epsilon_2|]X_2$. We index the ϵ and X type variables, as they belong to different definitions and are therefore distinct.

   We call the value and effect variables (here: ϵ_1, X_1) occurring in the embracing peg $[\epsilon_1|](\mathit{List}\ X)$ *rigid*, because they are treated as constants as type-checking must succeed for any instantiation. On the other hand, we call the value and effect type variables of $[\epsilon_2|]X_2$ (here: ϵ_2, X_2) *flexible* as they can be implicitly instantiated for every single occurrence of `evalState`.

   Matching the two pegs means to unify them. Not only need the value types be unified (resulting in $X_2 := \mathit{List}\ X_1$) but also the abilities. This is where the notion of ambient ability Σ comes in, and initially Σ is just $[\epsilon_1|]$. We give it this special name because it will accumulate as we type-check further sub-terms. Unifying the ambient ability with $[\epsilon_2|]$ succeeds and results in $\epsilon_2 := \epsilon_1$.

2. `evalState (dropEverySecond [1, 2, 3, 4]) false` $\quad \Sigma := [\epsilon_1 \mid \mathit{State}\ Y]$

   Again, the term's type is determined by the result type of `dropEverySecond`. Also, the term is given as argument to `evalState` at its port $\langle \mathit{State}\ Y\rangle X_2$. So in order to type-check this term, again we need to unify the value types (which succeeds as the term has type $X_2 = \mathit{List}\ X_1$).

   Let us now focus on the abilities. The ambient ability at this point gets augmented by the adjustment $\langle \mathit{State}\ Y\rangle$, evolving from $[\epsilon_1|]$ to $[\epsilon_1 \mid \mathit{State}\ Y]$. As in 1), we need to unify this ambient ability with `negAltList`'s peg's ability $[\epsilon_3 \mid \mathit{State}\ \mathit{Bool}]$. This succeeds as we can instantiate the flexible variables $Y := \mathit{Bool}$ and $\epsilon_3 := \epsilon$.

3. `evalState (dropEverySecond [1, 2, 3, 4]) false` $\quad \Sigma := [\epsilon_1 \mid \mathit{State}\ Y]$ and
   `evalState (dropEverySecond [1, 2, 3, 4]) false` $\quad \Sigma := [\epsilon_1 \mid \mathit{State}\ Y]$

   Both terms are values and thus only need to match their expected value types $\mathit{List}\ X_1$ and Bool, which is fulfilled for $X_1 := \mathit{Int}$.

The formal typing rules are explained later in Chapter 3.2. Especially relevant with respect to the ambient ability's evolution (i.e., its accumulation) is the Ty-App rule.

User Interaction. Before moving on the formalisation of Frank, we come back to the question of how to communicate with the user. So far, we have only seen a `main` operator that only returned a value but did not exhibit any effects to be handled externally. The only means of user interaction currently is the built-in `Console` interface, defined as follows.

```
interface Console = inch: Char
                  | ouch: Char -> Unit
```

Both its usage and its semantics with respect to the console when exhibited by `main` are straightforward. As an example, after defining a `print` operator that prints whole `String`s, we give a program that asks the user whether the program output should be *true* or *false*.

```
print : {String -> [Console]Unit}
print s = map ouch s; unit

main: {[Console]Bool}
main! = print "True␣(t)␣or␣false␣(_)?";
        case inch! {'t' -> true
                   | _  -> false}
```

This concludes the practical introduction to Frank, allowing us to proceed to its counterpart, the formal underpinnings. We hope these will be much more understandable after having gained some intuition in this chapter.

Chapter 3

Formal Specification of Frank

After giving an practical introduction to Frank we now give its formal definition. We begin with the syntax in Section 3.1 by describing the abstract syntax of types and terms. The idea is that every Frank program (written in concrete syntax) can be expressed and type-checked in abstract syntax terms which we use as the base for further formal reasoning. To reason about typing of terms, we present Frank's type system in Section 3.2 and finally the operational semantics in 3.3. We mention here that the syntax and typing rules have been defined by Lindley et al. [20] while we provide the operational small-step semantics.

3.1 Syntax

We go in the following through the different syntactic categories (Figure 3.1) which relate tightly to the concrete syntax seen so far in chapter 2.

Types

(value types)	$A, B ::= D\ \overline{R} \mid \{C\} \mid X$
(computation types)	$C ::= \overline{T \to} G$
(ports)	$T ::= \langle \Delta \rangle A$
(pegs)	$G ::= [\Sigma] A$
(type variables)	$Z ::= X \mid [E]$
(type arguments)	$R ::= A \mid [\Sigma]$
(polytype)	$P ::= \forall \overline{Z}.A$
(abilities)	$\Sigma ::= \varnothing \mid E \mid \Sigma, I\ \overline{R}$
(adjustments)	$\Delta ::= \iota \mid \Delta + I\ \overline{R}$
(type environments)	$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, f : P$

with data types D
interfaces I
value type var.s X
effect type var.s E

Terms

(uses)	$m ::= x \mid f\ \overline{R} \mid m\ \overline{n} \mid (n : A)$
(constructions)	$n ::= m \mid k\ \overline{n} \mid c\ \overline{n} \mid \{e\}$
	$\mid$ **let** $f : P = n$ **in** n'
	$\mid$ **letrec** $\overline{f : P = e}$ **in** n
(computations)	$e ::= \overline{\overline{r} \to n}$
(comp. patterns)	$r ::= p \mid \langle c\ \overline{p} \to z \rangle \mid \langle x \rangle$
(value patterns)	$p ::= k\ \overline{p} \mid x$

with term var.s x, y, z
polyterm var.s f
constructors k
commands c

Figure 3.1: Abstract Syntax of Types and Terms [20]

Types. We begin by describing the types. Value types A, B are either instantiated data types $D\ \overline{R}$, operator types $\{C\}$ or value type variables X. Type variables are instantiated by applying type arguments R. We denote a list of syntactic objects by overlining a representative (e.g. $\overline{R}$

with a general representative R) or using index notation (e.g. (R_i) with a particular element $(R_i)_i$).

Computation types C consist of a list of ports T and a peg G, with a port containing an adjustment Δ and a peg containing an ability Σ. An adjustment is a list of interface instantiations $I\ \overline{R}$ that may be empty (ι). Similarly, an ability contains also a list of interface instantiations $I\ \overline{R}$ and in addition an effect variable (E) or not ($\varnothing$).

Type arguments R are to be bound by type variables Z. With the exception of abilities (which may contain at most a single effect variable E), value and effect type variables can be declared side by side, e.g. in a data type definition. Because value and effect type variables belong to separate namespaces, we here explicitly write effect type variables in brackets, $[E]$, and value type variables without, X. There are three kind of constructs that are parameterised by type arguments: 1) polymorphic data types, 2) polymorphic interfaces and 3) polymorphic operators. Notice the significant difference here: 1) and 2) are type-constructs where as 3) polymorphic operators are terms. By introducing the syntactic category of a polytype P, we can bestow a notion of typing on polymorphic operators.

When instantiating a polymorphic operator of polytype $\forall\overline{Z}.A$ with type arguments $\overline{R}$, the instantiated type is $A[\overline{R}/\overline{Z}]$ which denotes the substitution of $(R_i)_i$ for free type variables $(Z_i)_i$ in A.

Terms. Let us now focus on the terms. They are given in two categories, uses m and constructions n, with the latter including the former. The reason for introducing two categories is that Frank uses a bidirectional type system [32] that distinguishes between terms whose type is inferred (uses m) and checked (constructions n). This distinction will appear straightforward when looking at the typing rules and should not matter to us for now.

Uses m may be monomorphic term variables x, instantiated polymorphic term variables $f\ \overline{R}$, applications $m\ \overline{n}$ or coercions $(n : A)$. Polymorphic term variables f that are instantiated by $\overline{R}$ are used in a more explicit manner in the abstract syntax than in the concrete syntax. E.g., for an application `map not [false, true]` we implicitly instantiate the two value type variables of `map` by `Bool` and the effect type variable by the ambient ability. Implicit instantiation is a service provided by the unification algorithm during type-checking and while we rely on this in practice, we facilitate formal reasoning by making these instantiations explicit in abstract syntax.

Another syntactic object that is solely there to facilitate formal reasoning is the coercion $(n : A)$. It is used to allow type inference for terms whose type has been determined before. Of course, annotating n by a type A is only useful if the annotation has been backed by a typing judgement.

Constructions n may be either uses m, constructor applications $k\ \overline{n}$, command requests $c\ \overline{n}$, operators $\{e\}$, **let**-expressions or **letrec**-expressions. Frank allows constructors k only when directly applied (to $\overline{n}$). We mention here a difference to the original Frank version [20] with respect to commands and command requests. While originally commands c were allowed as stand-alone use terms, they now behave similarly to constructors k in that only applied commands are legal[1]. The **let**- and **letrec**-expressions can be essentially used to represent top-level (recursive) definitions from concrete syntax as abstract syntax terms. There are two important points to mention here. First, **let**-expressions in abstract syntax are different from the concrete syntactic sugar introduced before, which only provides *monomorphic* `let`-expressions. Second, we mention that while abstract **letrec**-expressions form the counterpart to concrete top-level operator definitions, abstract **let**-expressions do not have their counterpart implemented yet in our compiler. The counterpart would be top-level polymorphic definitions that are *not necessarily operators* (e.g. an definition that binds the polymorphic empty list `[]`).

Computations e consist of patterns r which make binding available for terms n. Each computation e involves a matrix of patterns r because for *each case* there must be a pattern for *each port*. A pattern r can be either a value pattern p, a command-request pattern $\langle c\ \overline{p} \to z\rangle$ or a catch-all pattern $\langle x\rangle$. First, a value pattern p corresponds to classical pattern matching that only can match when a computation has terminated with a value. Second, a command-request

[1] The motivation for this is that commands c cannot considered as values anymore. They used to be in Core Frank, but since their occurrence is dependent on the ambient ability, this seemed not sensible to us.

pattern $\langle c\ \overline{p} \to z \rangle$ matches a particular command c, but only if the command's arguments match the value patterns $\overline{p}$. Resulting bindings include next to the bindings of $\overline{p}$ a continuation z. Finally, a catch-all pattern $\langle x \rangle$ matches in any case, no matter whether a value is yielded or a command request is made, and results in the binding of a thunk.

$$
\begin{array}{rcl}
\mathit{Definition} & ::= & \mathit{DatatypeDef} \mid \mathit{InterfaceDef} \mid \mathit{TermDef} \mid \mathit{OperatorDef} \\
\mathit{DatatypeDef} & ::= & \mathbf{data}\ D\ \overline{Z}\ =\ \overline{\mathit{Constructor}} \\
\mathit{Constructor} & ::= & k\ \overline{A} \\
\mathit{InterfaceDef} & ::= & \mathbf{interface}\ I\ \overline{Z}\ =\ \overline{\mathit{Command}} \\
\mathit{Command} & ::= & c\ :\ \overline{A \to}\ B \\
\mathit{TermDef} & ::= & f\ :\ P \\
 & & f = n \\
\mathit{OperatorDef} & ::= & f\ :\ \forall \overline{Z}.\{C\} \\
 & & \overline{f\ \overline{r}\ =\ n}
\end{array}
$$

Figure 3.2: Abstract Syntax of Top-Level Definitions

Top-Level Definitions. At last, we give a formal account of the top-level definitions available in Frank (Figure 3.2). Although we will not reason about top-level definitions (we argued already that terms and types capture all Frank programs), we see their formal specification useful as 1) it closely connected to the concrete syntax (which we have only given by example) and 2) it is closely connected to the abstract syntax of terms and types. The specification should be fairly familiar, but we make a few comments.

Because defined operators f, constructors k and commands k have the same syntactical form, they share the same namespace. As mentioned before, term definitions *TermDef* are not implemented in Frank yet. We can help ourselves though for now by defining thunks instead of values and forcing them when needed. The following example is given in concrete syntax.

```
justNil: Maybe (List X)
justNil = just []
```

can be simulated by

```
justNil: {Maybe (List X)}
justNil! = just (nil)
```

Finally, we mention that although in concrete syntax operator definitions are implicitly polymorphic, both term definitions *TermDef* and operator definitions *OperatorDef* explicitly quantify over type variables.

3.2 Typing Rules

We now proceed to the typing rules of Frank (Figure 3.3) which establish a bidirectional effect type system [32]. It is bidirectional in the sense that from an algorithmic perspective, there are two modes. The first mode is an *inferring* one: Given a term m, a type annotation A is inferred. The second mode is a *checking* one: Given both a term n and a type annotation A, it is checked whether this annotation fulfils the rules or not. The interplay between the two modes becomes clear in the following.

Besides the first two kinds of typing judgements representing the *check* and *infer* modes, there are three auxiliary kinds of typing judgements concerned with computation types and pattern types. Two central elements of most kinds are the type environment Γ and the ambient ability Σ, keeping track of the bindings and effects that are currently accessible to a term.

$\boxed{\Gamma[\Sigma]\vdash m \Rightarrow A}$ *For environment* Γ *and ambient* Σ*, use* m *is inferred to have type* A*.*

$$\textsc{Ty-Var}\ \frac{x : A \in \Gamma}{\Gamma[\Sigma]\vdash x \Rightarrow A} \qquad \textsc{Ty-PolyVar}\ \frac{f : \forall \overline{Z}.A \in \Gamma}{\Gamma[\Sigma]\vdash f\ \overline{R} \Rightarrow A[\overline{R}/\overline{Z}]} \qquad \textsc{Ty-Coerce}\ \frac{\Gamma[\Sigma]\vdash n : A}{\Gamma[\Sigma]\vdash (n : A) \Rightarrow A}$$

$$\textsc{Ty-App}\ \frac{\Gamma[\Sigma]\vdash m \Rightarrow \{\overline{\langle\Delta\rangle A \to [\Sigma']B}\} \qquad \Sigma' = \Sigma \qquad \overline{\Gamma[\Sigma \oplus \Delta]\vdash n : A}}{\Gamma[\Sigma]\vdash m\ \overline{n} \Rightarrow B}$$

$\boxed{\Gamma[\Sigma]\vdash n : A}$ *For environment* Γ *and ambient* Σ*, construction* n *is checked to have* type A*.*

$$\textsc{Ty-Switch}\ \frac{\Gamma[\Sigma]\vdash m \Rightarrow A \qquad A = B}{\Gamma[\Sigma]\vdash m : B} \qquad \textsc{Ty-Data}\ \frac{k\ \overline{A} \in D\ \overline{R} \qquad \overline{\Gamma[\Sigma]\vdash n : A}}{\Gamma[\Sigma]\vdash k\ \overline{n} : D\ \overline{R}}$$

$$\textsc{Ty-Command}\ \frac{c : \overline{A \to} B \in \Sigma \qquad \overline{\Gamma[\Sigma]\vdash n : A}}{\Gamma[\Sigma]\vdash c\ \overline{n} : B} \qquad \textsc{Ty-Let}\ \frac{P = \forall \overline{Z}.A \qquad \Gamma[\Sigma]\vdash n : A \qquad \Gamma, f : P\ [\Sigma]\vdash n' : B}{\Gamma[\Sigma]\vdash \mathbf{let}\ f : P = n\ \mathbf{in}\ n' : B}$$

$$\textsc{Ty-Operator}\ \frac{\Gamma \vdash e : C}{\Gamma[\Sigma]\vdash \{e\} : \{C\}} \qquad \textsc{Ty-LetRec}\ \frac{\overline{P = \forall \overline{Z}.\{C\}} \qquad \overline{\Gamma, \overline{f : P} \vdash e : C} \qquad \Gamma, \overline{f : P}\ [\Sigma]\vdash n : B}{\Gamma[\Sigma]\vdash \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n : B}$$

$\boxed{\Gamma \vdash e : C}$ *For environment* Γ *and ambient* Σ*, comp.* e *is checked to have comp. type* C*.*

$$\textsc{Ty-Comp}\ \frac{(r_{i,j} : T_j \dashv[\Sigma]\ \Gamma'_{i,j})_{i,j} \qquad (\Gamma, (\Gamma'_{i,j})_j\ [\Sigma]\vdash n_i : B)_i \qquad ((r_{i,j})_{i,j})\ \text{covers}\ (T_j)_j}{\Gamma \vdash ((r_{i,j})_j \to n_i)_i : (T_j \to)_j\ [\Sigma]B}$$

$\boxed{p : A \dashv \Gamma}$ *Value pattern* p *matches values of type* A *and provides bindings* Γ*.*

$$\textsc{Ty-P-Var}\ \frac{}{x : A \dashv x : A} \qquad \textsc{Ty-P-Data}\ \frac{k\ \overline{A} \in D\ \overline{R} \qquad \overline{p : A \dashv \Gamma}}{k\ \overline{p} : D\ \overline{R} \dashv \overline{\Gamma}}$$

$\boxed{r : T \dashv[\Sigma]\ \Gamma}$ *Comp. pat.* r *matches comp.s of port* T *under amb.* Σ *and provides bind.s* Γ*.*

$$\textsc{Ty-P-Value}\ \frac{p : A \dashv \Gamma}{p : \langle\Delta\rangle A \dashv[\Sigma]\ \Gamma} \qquad \textsc{Ty-P-Request}\ \frac{c : \overline{A \to} B \in \varnothing \oplus \Delta \qquad (p_i : A_i \dashv \Gamma_i)_i}{\langle c\ \overline{p} \to z\rangle : \langle\Delta\rangle B' \dashv[\Sigma]\ \overline{\Gamma}, z : \langle\iota\rangle B \to [\Sigma \oplus \Delta]B'}$$

$$\textsc{Ty-P-CatchAll}\ \frac{}{\langle x\rangle : \langle\Delta\rangle A \dashv[\Sigma]\ x : \{[\Sigma \oplus \Delta]A\}}$$

Figure 3.3: Typing Rules [20]

Inferring Types. We begin by explaining the rules belonging to the *infer* mode of the form $\Gamma[\Sigma]\vdash m \Rightarrow A$. A monomorphic term variable x is inferred to have the type A that Γ assigns to it (Ty-Var). A polymorphic term variable f that is assigned the polytype $\forall\overline{Z}.A$ by Γ can be instantiated to $f\ \overline{R}$ and consequently is inferred to have the type $A[\overline{R/Z}]$ (Ty-PolyVar).

We come now to the Ty-App rule which captures how the ambient ability evolves and how type inference and type checking interact. The type of an application $m\ \overline{n}$ under ambient ability Σ is inferred by first *inferring* the type of m under Σ and then *checking* that each argument $(n_i)_i$ is of the type expected by m, under an augmented ambient ability $\Sigma \oplus (\Delta_i)_i$. The ability

is augmented by exactly the adjustment that is handled at the respective port. Finally, the ability required by m must match exactly the current ambient ability ($\Sigma' = \Sigma$).

The last type inference rule Ty-Coerce establishes a switch of mode (from inference to checking) based on a very straightforward consequence: If a term n has an annotated type A and the annotation is checked to be true, then A can be inferred to be indeed the type of n.

Checking Types. We proceed by explaining the rules belonging to the *checking* mode of the form $\Gamma[\Sigma]\vdash n : A$. First, checking that a use m is of type B is achieved by inferring m's type and then making sure that it is equal to B (Ty-Switch). Type-checking applied constructors $k\ \overline{n}$ via Ty-Data, requested commands $c\ \overline{n}$ via Ty-Command, operators $\{e\}$ via Ty-Operator, **let**- and **letrec**-expressions (Ty-Let and Ty-LetRec) is rather standard, relying on recursive type-checks of sub-terms.

Checking Computation Types. The rule Ty-Comp checks the sub-terms of a computation e: Patterns must respect the expected argument types of e and for each clause, when provided with bindings of the clause's patterns, the handling term must respect the result type of e. Furthermore, the clauses must be exhaustive[2].

Checking Pattern Bindings. The remaining rules connect patterns with the bindings they yield. There are two sets of rules, one for value patterns and one for computation patterns. Value patterns represent classical pattern matching by decomposing constructors and yielding according bindings (Ty-P-Var, Ty-P-Data). Computation patterns subsume value patterns (Ty-P-Value) and catch command requests. Let us take a closer look at the Ty-P-Request rule. When matching a particular command request via $\langle c\ \overline{p} \to z\rangle$, the command c needs to be available in the port's adjustment Δ and the patterns $\overline{p}$ need to respect the argument types of c. The bindings include not only the bindings yielded by $\overline{p}$ but also the continuation z which takes a value of the argument type of c and results in the port's type. The ability of z is the augmented ambient ability $\Sigma \oplus \Delta$ making z suitable to be handed to the port again. Finally, by the Ty-P-CatchAll rule, any value or command request that is matched by $\langle x\rangle$ will be bound to x as a thunk.

3.3 Operational Small-Step Semantics

We specify the operational semantics of Frank by providing an operational small-step reduction relation. Lindley et al. [20] give a small-step reduction relation for Core Frank which inspires our definition, but because Frank differs in important aspects like effect-handling from Core Frank, so does its operational semantics. Before finally giving the small-step reduction relation, we begin by defining additional syntactic categories like values and contexts (Figure 3.4) and later define how pattern matching results in mono-value substitutions.

$$
\begin{array}{ll}
\text{(use values)} & v ::= (w : A) \\
\text{(construction values)} & w ::= v \mid k\ \overline{w} \mid \{e\} \\
\text{(normal forms)} & u ::= w \mid \mathcal{E}[c\ \overline{w}] \\
\text{(evaluation contexts)} & \mathcal{E} ::= \bullet \mid \mathcal{E}\ \overline{n} \mid \{e\}\ (\overline{u}, \mathcal{E}, \overline{n}) \mid k\ (\overline{w}, \mathcal{E}, \overline{n}) \mid c\ (\overline{w}, \mathcal{E}, \overline{n}) \mid \\
 & \qquad \textbf{let}\ f : P = \mathcal{E}\ \textbf{in}\ n \mid (\mathcal{E} : A) \\
\text{(mono-value substitutions)} & \sigma ::= \varnothing \mid \sigma, [x \mapsto (w : A)]
\end{array}
$$

$$
\begin{array}{ll}
TyEnv : & \sigma \mapsto \Gamma \\
TyEnv\ \ \overline{[x \mapsto (w : A)]} & = \overline{x : A}
\end{array}
$$

Figure 3.4: Further Abstract Syntax w.r.t. Operational Semantics

[2]Coverage checking is not yet implemented in the Frank compiler [23]

Values. Values are terms that are irreducible and considered as desired output of a computation. We prove later that every well-typed and terminating Frank program reduces to a value (Chapter 6. We distinguish again between *use values* v and *construction values* w which together are made up of operators (anonymous operators $\{e\}$, applied constructors $k\ \overline{w}$ and their coerced variants $(w : A)$.

Normal Forms and Contexts. Let us take a look at contexts which are central to the reduction relation. Next to the atomic context hole $\bullet$, contexts fix the evaluation position within different syntactic constructs. In an application, a context can take up two different positions. First, it can take up the position of the operator, $\mathcal{E}\ \overline{n}$. Second, if the operator is already reduced to an operator $\{e\}$, it can focus the position of an argument whose left neighbour arguments are all in normal form, $\{e\}\ (\overline{u}, \mathcal{E}, \overline{n})$.

A term is in *normal form* if it is either a value w or in a form where the next reduction step requires the handling of a command request, $\mathcal{E}[c\ \overline{w}]$. Taking a second glance at the context $\{e\}\ (\overline{u}, \mathcal{E}, \overline{n})$, we can observe that this is a fitting choice, as arguments are allowed to reduce from left to right until they are all either values or exhibit a command — at which point handling can take place.

The other contexts are rather standard. Arguments in constructor applications and command requests are reduced from left to right, $k\ (\overline{w}, \mathcal{E}, \overline{n})$ and $c\ (\overline{w}, \mathcal{E}, \overline{n})$. Finally, **let**-expressions allow reduction of their bound term, $\mathbf{let}\ f : P = \mathcal{E}\ \mathbf{in}\ n$, and reduction can happen under coercion, $(\mathcal{E} : A)$.

Term Substitutions. Reducing terms involves the substitution of values for variables. In our setting, we deal with two kinds of term substitutions, namely *polymorphic substitutions* and *monomorphic substitutions* σ which are used for different constructs. On one hand, because **let**- and **letrec**-expressions bind polymorphic terms, we need *polymorphic* substitutions to resolve them. On the other hand, pattern matching generates monomorphic bindings, thus requiring *monomorphic substitutions*. A particularity common to both substitutions is that due to further formal reasoning we require the substituted values to be annotated by their type (coercions) and to keep this annotation. Instead of choosing the most general form of substitution (polymorphic with annotated polytypes), we distinguish between the two kinds and define them here. Because monomorphic substitutions are used extensively during reasoning, we assign to them the metavariable σ.

Applying a single-entry *polyvalue substitution* is defined as follows: $n[(w : \forall \overline{Z}.A)/f]$ is the term n with $(w : A[\overline{R}/\overline{Z}])$ substituted for every free variable occurrence $f\ \overline{R}$. This definition extends to general substitutions of multiple entries.

Applying a single-entry *monovalue substitution* $\sigma = [x \mapsto (w : A)]$ is defined as follows: $n[\sigma]$ is the term n with $(w : A)$ substituted for every free variable occurrence x. This definition extends to general substitutions of multiple entries.

Given a mono-value substitution σ, we can strip off the actual values and obtain a type environment Γ, a transformation that is defined as the function $TyEnv$. This is of course only possible due to the annotations in σ and will enable us later to reason about pattern bindings.

Pattern Binding. Before we finally come to the reduction relation, we need to define the relationship between patterns and normal forms to be matched by them (Figure 3.5). If there is a match, this will result in a set of bindings — which is just a monovalue substitution σ. In this case in particular, σ will only contain monomorphic entries. Notice that the Bind-P rules we describe in the following constitute the dynamic counterparts to the static Ty-P typing rules.

We distinguish between two sets of rules, one for value patterns and one for computation patterns. Value patterns either bind whole values (Bind-P-Var) or first decompose them into sub-values (Bind-P-Data). Computation patterns subsume value patterns (Bind-P-Value) and catch command requests. Let us take a closer look at the Bind-P-Request rule. When matching a particular command request via $\langle c\ \overline{p} \to z\rangle$, the command c needs to be available in the port's adjustment Δ. Furthermore, patterns p_i need to match the request's arguments

$\boxed{p : A \leftarrow w \dashv \sigma}$ *Value pattern p for type A matches w and binds σ.*

$$\textsc{Bind-P-Var}\ \frac{}{x : A \leftarrow w \dashv \{x \mapsto (w : A)\}} \qquad \textsc{Bind-P-Data}\ \frac{k\ \overline{A} \in D\ \overline{R} \qquad p_i : A_i \leftarrow w_i \dashv \sigma_i}{k\ \overline{p} : D\ \overline{R} \leftarrow k\ \overline{w} \dashv \sigma_1, ..., \sigma_n}$$

$\boxed{r : T \leftarrow u \dashv[\Sigma]\ \sigma}$ *Comp. pattern r for peg T matches w under ambient Σ and binds σ.*

$$\textsc{Bind-P-Value}\ \frac{p : A \leftarrow w \dashv \sigma}{p : \langle\Delta\rangle A \leftarrow w \dashv[\Sigma]\ \sigma}$$

$$\textsc{Bind-P-Request}\ \frac{c : \overline{B \to} C \in \Delta \qquad p_i : B_i \leftarrow w_i \dashv \sigma_i}{\langle c\ \overline{p}\ \to\ z\rangle : \langle\Delta\rangle A \leftarrow \mathcal{E}[c\ \overline{w}] \dashv[\Sigma]\ \sigma_1, ..., \sigma_n, \{z \mapsto (\{x \to \mathcal{E}[x]\} : \{C \to [\Sigma \oplus \Delta]A\})\}}$$

$$\textsc{Bind-P-CatchAll}\ \frac{}{\langle x\rangle : \langle\Delta\rangle A \leftarrow u \dashv[\Sigma]\ \{x \mapsto (\{u\} : \{[\Sigma \oplus \Delta]A\})\}}$$

Figure 3.5: Pattern Binding

w_i, resulting in bindings (i.e., mono-value substitutions) σ_i. The overall generated mono-value substitution contains next to the entries of all σ_i the continuation z that is bound to an operator $\{x \to \mathcal{E}[x]\}$ of the according continuation type. When fed with the handler's provided result value, that value is substituted in and the handled computation can continue to reduce. Finally, by the Bind-P-CatchAll rule, any value or command request that is matched by $\langle x\rangle$ will be bound to x as a thunk.

Small-Step Reduction. We have now introduced all preliminaries to proceed to the core of the description of the dynamic semantics, the small-step reduction relation (Figure 3.6). We begin with the rule that describes handling (including the degenerate case of ordinary beta-reduction), Step-Handle. More specifically, it describes the reduction of an operator that provides a pattern matrix $r_{i,j}$ and handling terms n_i, applied to a vector of normal-form arguments u_j. We require that at least one case i matches the arguments, resulting in mono-value substitutions $\sigma_{i,j}$ (one for each argument). If multiple cases i match, we pick the minimal i, as is standard in pattern matching. Finally, the application reduces to n_i with all mono-value substitutions $\sigma_{i,j}$ provided as bindings.

Reduction via Step-Let and Step-LetRec are standard. While a **let**-expression binding an value w is simply dissolved by substituting in w, the **letrec**-expression reproduces itself for further recursive calls, in particular by substituting itself into the reduced expression.

A use value coerced to a type, $(v : A)$, can strip off its coercion as coercions are only important for construction values (Step-Coerce). Last, reduction can happen under context, relying on contexts to define the desired evaluation order (Let-Ctx).

$$\textsc{Step-Handle}\quad \frac{(r_{i,j} : T_j \leftarrow u_j \dashv[\Sigma]\ \sigma_{i,j})_{i,j} \qquad i \text{ is minimal}}{\left(\begin{array}{ll}\{r_{1,1}\ ...\ r_{1,l} \to n_1 & \\ \mid ... & : \{\overline{T \to}\ [\Sigma]B\} \\ \mid r_{k,1}\ ...\ r_{k,l} \to n_k\} & \end{array}\right) u_1\ ...\ u_l\ \to\ n_i[\sigma_{i,1}, ..., \sigma_{i,l}]}$$

$$\textsc{Step-Let}\quad \frac{}{\mathbf{let}\ f : P = w\ \mathbf{in}\ n \to n[(w : P)/f]}$$

$$\textsc{Step-LetRec}\quad \frac{}{\mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n' \to n'[(\{\overline{\overline{r} \to \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n}\} : P)/f]}$$

$$\textsc{Step-Coerce}\quad \frac{}{(v : A) \to v} \qquad \textsc{Step-Ctx}\quad \frac{n \to n'}{\mathcal{E}[n] \to \mathcal{E}[n']}$$

Figure 3.6: Small-Step Reduction Relation

Chapter 4

Ability Representation and Operator Composition

An ability consists of an optional effect variable and a set of instantiated interfaces, e.g. $[\epsilon \mid \mathit{State\ Int},\ \mathit{Console}]$. When handlers are composed, the handlers' adjustments are accumulated by the ambient ability and thereby offer an execution environment for the handlers' arguments. What happens if two handlers offer to handle the same interface? The mechanism to resolve this issue is shadowing, i.e., the inner handling offer shadows the outer handling offer. We make ability shadowing precise as follows.

- Shadowing can occur between instantiations of the same interface.
  For example, *State Int* can shadow *State Bool* and vice versa (despite different instantiations).
- For the notation of interface instantiations, left-to-right precedence denotes shadowing order.
  For example, in $[\epsilon \mid \mathit{State\ Bool},\ \mathit{State\ Int}]$, *State Bool* is shadowed by *State Int*.
- Abilities bound to the effect variable have lower precedence than listed interface instantiations.
  For example, if $\epsilon := [0 \mid \mathit{State\ Bool}]$, then $[\epsilon \mid \mathit{State\ Int}]$ expands to $[0 \mid \mathit{State\ Bool},\ \mathit{State\ Int}]$.
- Adjustments shadow previous interface instantiations.
  For example, $[\epsilon \mid \mathit{State\ Bool}]$ adjusted by $\langle \mathit{State\ Int} \rangle$ results in $[\epsilon \mid \mathit{State\ Bool},\ \mathit{State\ Int}]$.

We say that the right-most instantiation of an interface is **active**. Given an operator of a computation type $\{\overline{A \ \rightarrow} \ [\Sigma]B\}$, any command issued on the operator's application belongs necessarily to an active instantiation of Σ.

Besides the precedence which determines shadowing for each interface, the order of interface instantiations in the notation of an ability is irrelevant. For example, we have:

$$[\epsilon \mid \mathit{State\ Int},\ \mathit{Console}] = [\epsilon \mid \mathit{Console},\ \mathit{State\ Int}]$$

4.1 Internal Representation of Abilities

How shall abilities be represented internally, i.e., during the unification process? Because shadowed interface instantiations might seem to have no role any more within an ability, one could think of discarding them and represent abilities merely in terms of their active instantiations. This would for example identify $[\epsilon \mid \mathit{State\ Bool},\ \mathit{State\ Int}] = [\epsilon \mid \mathit{State\ Int}]$ as both abilities would have the same representation. Even though the Frank specification [20] does not state precisely whether identification should be allowed or not, the original Frank compiler [21] does implement this identification.

However, there is a problem with this identification[1]. When resolving two unification constraints, it should not matter in which order these are resolved, i.e., for any order, unification should either succeed or fail. Choosing a canonical representation of interface instantiations that discards inactive instantiations breaks this property. To see this, consider the following example.

1. Unify $[\epsilon|]$ with $[0 \mid \mathit{Console}]$
2. Unify $[\epsilon \mid \mathit{Console}]$ with $[0 \mid \mathit{Console}]$

First unifying 1) results in the solution $\epsilon := [0 \mid \mathit{Console}]$, then solving 2) is successful as we have $[0 \mid \mathit{Console},\ \mathit{Console}] = [0 \mid \mathit{Console}]$. On the other hand, first unifying 2) results in the solution $\epsilon := [0 \mid]$, then solving 2) fails as there is no solution to $[0 \mid] = [0 \mid \mathit{Console}]$. This kind of situation can indeed occur in Frank [25].

This phenomenon rules out discarding inactive interface instantiations during unification and lets us stick with a representation comprising both active and inactive instantiations. Note that unification order is irrelevant now since in situations as given above neither unification order succeeds.

4.2 Composing Handlers

The central aspect of the ambient ability is that it makes handlers compositional: When composing handlers, their handling capabilities expressed as adjustments are implicitly accumulated. When a term is executed in a particular ambient ability, the ambient ability can be seen both 1) as an *offer* to handle certain effects but at the same time also 2) as a *demand* to handle certain effects. The first point of an offer is quite clear as this is the whole point of using handlers. The second point, we think, is more subtle but can come in our way when composing handlers as shown during our discussion of the following four scenarios. We believe these scenarios represent the common ways in which handlers are composed in Frank and are therefore worth examining.

4.2.1 Scenario 1: Accumulating Effects

Consider the following example in which two handlers are presented. First, `boolFix` handles binary choice requests by always returning `false`. Second, `intFix` handles integer choice by always returning `42`.

```
interface BChoice = bchoose: Bool
interface IChoice = ichoose: Int

boolFix: {<BChoice>X -> X}
boolFix x = x
boolFix <bchoose -> k> = boolFix (k false)

intFix: {<IChoice>X -> X}
intFix x = x
intFix <ichoose -> k> = intFix (k 42)
```

Let us now compose the two handlers to arrive at a handler `fixBoolInt` that handles both `BChoice` and `IChoice`. By composing `fixBool` and `fixInt`, we arrive at the following definition.

```
boolIntFix: {<BChoice,IChoice>X -> X}
boolIntFix <m> = boolFix (intFix m!)
```

Understanding why the definition matches our expectation and type-checks is straightforward: The ambient ability at the point where `m` is executed is $[\epsilon \mid \mathit{BChoice},\ \mathit{IChoice}]$. The argument `m` is thus expected to match the ambient ability which it does by specification of the adjustment $\langle \mathit{BChoice},\ \mathit{IChoice} \rangle$.

[1] This was pointed out by Conor McBride, Sam Lindley and Craig McLaughlin via private communication.

We can generalise this scenario by allowing the two composed handlers to exhibit effects themselves, too. As an example, let us swap the first handler `boolFix: {<BChoice>X -> X}` by a handler `boolByCons: {<BChoice>X -> [Console]X}` that asks the user to determine the choice, resulting in the handler `boolByConsIntFix`.

```
boolByCons: {<BChoice>X -> [Console]X}
boolByCons x = x
boolByCons <bchoose -> k> = print "true␣(t)␣or␣false␣(_)?␣"; case inch!
                            { 't' -> print "t\n"; boolByCons (k true)
                            | _   -> print "_\n"; boolByCons (k false) }

boolByConsIntFix: {<BChoice,IChoice>X -> [Console]X}
boolByConsIntFix <m> = boolByCons (intFix m!)
```

Any effects required by the composed handlers have to occur in the peg of the composition `boolByConsIntFix`, which ensures that the required effects are indeed available, thus making unification during type-checking succeed. The pattern behind the given example is very common and demonstrates how composition can be implemented in a straightforward manner.

4.2.2 Scenario 2: Unintentionally Exposed Intermediate Effects

Consider now an extension to the previous example. We introduce the handler `intByBool` which handles integer choice requests by making multiple binary choice requests (for conciseness, it only chooses from 0 to 3).

```
intByBool: {<IChoice>X -> [BChoice]X}
intByBool x = x
intByBool <ichoose -> k> = let n = if bchoose! { if bchoose! {3} {2} }
                                               { if bchoose! {1} {0} }
                           in intByBool (k n)
```

Let us now compose the two handlers `boolByCons` and `intByBool` to arrive at a handler `intByCons` that handles `IChoice` and makes use of `Console`. At first thought, we would expect the type to be `{<IChoice>X -> [Console]X}`. However, by composing `boolByCons` and `intByBool` in the same way as in scenario 1), we arrive at the following definition.

```
intByCons: {<BChoice,IChoice>X -> [Console]X}
intByCons <m> = boolByCons (intByBool m!)
```

The reason for having the adjustment $\langle BChoice,\ IChoice\rangle$ instead of $\langle IChoice\rangle$ is that at the point where `m` is executed the ambient ability is $[\epsilon \mid Console,\ BChoice,\ IChoice]$. The ambient ability thus *demands* `m` to have the same ability which it only does by the specified adjustment.

Violation of Encapsulation We observe that this causes a major shortcoming, namely violation of encapsulation. The internal mechanism of `intByCons` (the fact that it uses $BChoice$) is exposed to the outside as part of the interface. This violation bears two undesired consequences:

First, there is potential for undesired interference with internals. If `intByCons` is applied to an argument that requires $BChoice$ (but which is to be handled completely differently), the internals of `intByCons` will capture this ability and unwantedly handle it.

Second, determining the correct adjustment is cumbersome. Determining the accumulated adjustment $\langle BChoice,\ IChoice\rangle$ — although straightforward in this particular case — generally requires understanding the internals of the embracing handler.

We conclude that when composing handlers in this way, having an intermediate handling construct (`intByBool` issues effects handled by `boolByCons`) will inevitably cause intermediate effects to leak outside. We take up this issue again in Section 4.4.

4.2.3 Scenario 3: Intentionally Exposed Intermediate Effects

However, if we had a different intention about what `intByCons` should do, we could be satisfied with its type. Let us assume we give it the different name `boolIntByCons` and expect it to issue

both binary and integer decisions. In this case, the signature

```
boolIntByCons: {<BChoice, IChoice>X -> [Console]X}
```

is exactly what we would expect. We conclude that it depends on the intention whether the leakage of effects issued by intermediate handling constructs is acceptable or not. While it is not desired for `intByCons`, it is desired for `boolIntByCons`.

4.2.4 Scenario 4: Intermediate Handling Without Introducing New Effects

It is revealing to look into one particular case of intermediate handling constructs, namely that in which handlers handle and generate effects which are exposed to the outside anyway. Let us consider the following example of such an intermediate handler that increases every integer choice.

```
incrInt: {<IChoice>X -> [IChoice]X}
incrInt x = x
incrInt <ichoose -> k> = let n = ichoose! in incrInt (k (n + 1))
```

Let us define on top of `incrInt` a handler `doubleIncrInt` which increments every integer choice twice. We would like it to have the same type as `incrInt`, namely `{<IChoice>X -> [IChoice]X}`. In the same manner as in scenario 2) though, we have to line up the adjustment with the ambient ability and do not arrive at our desired type.

```
doubleIncrInt: {<IChoice, IChoice>X -> [IChoice]X}
doubleIncrInt <m> = incrInt (incrInt m!)
```

However, there is a difference to scenario 2) here. At the point where `m` is executed, the ambient ability is $[\epsilon \mid \mathit{IChoice}, \mathit{IChoice}, \mathit{IChoice}]$. Any `ichoose` command at this point is handled only by the handler offering the active instantiation of `IChoice`. The inactive instantiations of the ambient ability are irrelevant *at this point*. Similarly, the inactive instantiations of the ability of `m` are irrelevant, because a term can only exhibit commands of its active interface instantiations. Because of this, we are able to introduce a modification to the type system which we make precise in the next section. We already give away that it allows us to give `doubleIncrInt` the type we would like to give it: `{<IChoice>X -> [IChoice]X}`.

4.3 Relaxing the Type System via Up-to-Inactive-Instantiations

We make a modification to the Ty-App rule in that we don't require the ambient ability and the argument's ability to be *equal* ($=$) but *equal-up-to-inactive-instantiations* ($\sim$). We first give the modified rule of Ty-App and then define $\sim$.

$$\textsc{Ty-App}\quad \frac{\Gamma[\Sigma]\vdash m \Rightarrow \{\overline{\langle\Delta\rangle A \to [\Sigma']B}\} \qquad \Sigma' \sim \Sigma \qquad \overline{\Gamma[\Sigma \oplus \Delta]\vdash n : A}}{\Gamma[\Sigma]\vdash m\ \overline{n} \Rightarrow B}$$

When inferring the type of an application $m\ \overline{n}$ under an ambient ability Σ, we first infer the type of the operator m (left premise). We then check that for the ability of m's peg, Σ', we have $\Sigma \sim \Sigma'$ (center premise). This relaxes the original premise $\Sigma = \Sigma'$. Last, we check that the arguments' types match the ports of m (right premise). Let us now define $\sim$, based on the notion of the active fragment of an ability Σ.

Definition 4.3.1 *Given an ability Σ, we call $Act(\Sigma)$ the active fragment of Σ (all inactive interface instantiations are discarded).*

$$\begin{aligned} Act:\ [0 \mid I_1\ \overline{R_1}, ..., I_n\ \overline{R_n}] &\mapsto [0 \mid I_{i_1}\ \overline{R_{i_1}}, ..., I_{i_k}\ \overline{R_{i_k}}] \ \ s.t.\ I_{i_j}\ \overline{R_{i_j}} \in \{I_1\ \overline{R_1}, ..., I_n\ \overline{R_n}\} \\ &\qquad\qquad \textit{is active}\ \forall j \in \{1, ..., k\} \\ [E \mid I_1\ \overline{R_1}, ..., I_n\ \overline{R_n}] &\mapsto [E \mid I_{i_1}\ \overline{R_{i_1}}, ..., I_{i_k}\ \overline{R_{i_k}}] \ \ s.t.\ I_{i_j}\ \overline{R_{i_j}} \in \{I_1\ \overline{R_1}, ..., I_n\ \overline{R_n}\} \\ &\qquad\qquad \textit{is active}\ \forall j \in \{1, ..., k\} \end{aligned}$$

Definition 4.3.2 *Given Σ, Σ', we say that Σ is equal-up-to-inactive-instantiations to Σ' ($\Sigma \sim \Sigma'$) if $Act(\Sigma) = Act(\Sigma')$.*

For better illustration, consider these three examples:

- $[\epsilon|] \nsim [\epsilon \mid \mathit{Console}]$ (active instantiation *Console* missing on left hand side)
- $[\epsilon \mid \mathit{State\ Int},\ \mathit{State\ Bool}] \sim [\epsilon \mid \mathit{State\ Bool}]$ (instantiation *State Int* is inactive)
- $[\epsilon \mid \mathit{State\ Bool},\ \mathit{State\ Int}] \nsim [\epsilon \mid \mathit{State\ Bool}]$ (active instantiations do not match)

Revisiting Scenario 4). Let us briefly revisit scenario 4 to justify that `incrInt` can indeed be given the type `{<IChoice>X -> [IChoice]X}` instead of `{<IChoice,IChoice>X -> [IChoice]X}`:

```
doubleIncrInt: {<IChoice>X -> [IChoice]X}
doubleIncrInt x = x
doubleIncrInt <m> = incrInt (incrInt m!)
```

At the point when `m` is executed, the ambient ability is $[\epsilon \mid \mathit{IChoice},\ \mathit{IChoice},\ \mathit{IChoice}]$. The ability of the peg of `m` is $[\epsilon \mid \mathit{IChoice},\ \mathit{IChoice}]$. By $[\epsilon \mid \mathit{IChoice},\ \mathit{IChoice},\ \mathit{IChoice}] \sim [\epsilon \mid \mathit{IChoice},\ \mathit{IChoice}]$ and Ty-App, the expression type-checks. Note that as $\sim$ subsumes equality, the former definition of type `{<IChoice,IChoice>X -> [IChoice]X}` is still admissible.

4.4 Discussion and Future Work

We have seen that relaxing the Ty-App rule solves part of the issue of unintentionally exposed intermediate effects (scenario 2), namely the special case of scenario 4. The more general case of scenario 2 though still bears a shortcoming that is not solved yet in Frank. In the original Frank paper [20] (Sect. 6), Lindley et al. take up a similar issue where they consider to remove interface instantiations by a "negative adjustment" from the ambient ability. Although the presented issue is similar, we cannot see how negative adjustments can directly help us in scenario 2.

We have explored how the scenarios are represented in the effect-extended version of Links [12] and the programming language Koka [18] and obtain the same results about the unintentional leaking of intermediate effects (scenario 2) [6, 5].

An approach that seems promising to us is the idea of a hiding composition operator[2] that allows hiding of effects. We demonstrate the idea on the example of scenario 2 by first giving the original code, but this time rewritten with some pseudo-code (`o` represents ordinary composition).

```
intByCons: {<BChoice, IChoice>X -> [Console]X}
intByCons <m> = (boolByCons o intByBool) m!)
```

Let us imagine now that instead of an ordinary composition operator `o` we have a hiding composition operator $\mathtt{o}_{\mathtt{E}}$ where `E` is the "hiding adjustment", similar to the notion of a negative adjustment. We can now hide the intermediate effect.

```
intByCons: {<IChoice>X -> [Console]X}
intByCons <m> = (boolByCons oBChoice intByBool) m!)
```

A straightforward implementation of this operator consists of command renaming. We first explain the idea and then demonstrate it on the given example. Every operator has of multiple ports, each defining a set of commands that are handled, and a peg defining a set of commands that can potentially be requested. Imagine now that the compiler can rewrite these names for particular operator occurrences. The internal behaviour of the operator is still the same, but its interaction with the environment may change completely. By changing the output names (i.e., of requested commands) of one operator and the input names (i.e., of handled commands) of another in the *same way*, we achieve a tunnel between the two. Other operators will not interfere with this tunnel if the compiler makes sure that the commands were renamed in a

[2]This idea was first brought up by Sam Lindley via private communication.

unique way. With the renaming during compile time, we believe that the translated programs can be quite efficient.

Let us carry this idea over to our example `intByCons`, where a possible translation through the compiler could involve the following. The `intByBool` operator is modified s.t. every issued `bchoose` request becomes a `bchooseTunnel` request. The `boolByCons` operator is modified s.t. on its first port, instead of catching `bchoose` requests, it catches `bchooseTunnel` requests. This has as consequence that `boolByCons` does not add to the ambient ability, resulting in the ambient ability $[\epsilon \mid \mathit{Console},\ \mathit{IChoice}]$ (instead of $[\epsilon \mid \mathit{Console},\ \mathit{BChoice},\ \mathit{IChoice}]$) at the point where `m` is forced.

In a more general manner, multi-tunnels could be established between one effect-handling operator and several effect-exhibiting operators. The only condition is that the effect-handling operator is "wrapping" the effect-generating ones. An issue that still needs to be tackled is notation as the components of such a multi-tunnel can be at very different locations. Naming such a multi-tunnel and then explicitly referencing it seems thus a good approach to us.

Chapter 5

Polymorphic Commands

Interfaces are polymorphic, i.e., they can be parameterised by types as for example in `Send Int` where `Send` is parameterised by `Int`. The first major extension to the Frank language is to make commands polymorphic, too. This means that *on each command call*, the caller can choose a particular instantiation and the command's handler must be able deal with any such instantiation. The following simple example demonstrates both the new functionality and its usefulness. The `choose` command of `PolyChoice` can be seen as a mix of `if` (taking two suspended computations of which only one gets forced) and boolean choice (non-determinism).

```
interface PolyChoice = choose X: {[Polychoice]X} -> {[PolyChoice]X} -> X

nondeterministicGreeting : {[PolyChoice]String}
nondeterministicGreeting! = choose {"Good␣afternoon"} {"Guten␣Tag"}

pickLeft: {<PolyChoice>X -> X}
pickLeft <choose a b -> k> = pickLeft (k a!)
pickLeft x                 = x
```

As observable in the first line, type variables (for both value type and effect type) can be specified right between a command's name and the colon. A function requiring `PolyChoice` (e.g. `nondeterministicGreeting`) can now call `choose` and thereby implicitly instantiate the type variable `X` (e.g. with `String`).

Polymorphic commands are a straightforward extension as only the syntax and the type system need some modification. While widening the number of programs which type-check, we do not introduce new dynamic semantics and thus need not change anything concerning post-type-check processing.

In the next sections, we present the formal integration of the extensions into Frank and how it enables a flexible kind of exception handling. In Chapter 7.1, we present further how polymorphic commands enable generative effects such as referentiable state and in Chapter 7.3, 7.4 we will see that they are indispensable for concurrency abstractions such as the actor model.

5.1 Formal Extension

We formally present the changes made to the syntax and the type system of Frank. We begin with the syntax. Commands c are now polymorphic in the same way as polymorphic variables f. Since we only allow monomorphic terms, commands c need to be applied to type arguments $\overline{R}$ first to become usable. We update the abstract syntax as follows.

$$\text{(constructions)}\quad n ::= m \mid k\ \overline{n} \mid c\ \overline{R}\ \overline{n} \mid \{e\} \mid \textbf{let}\ f : P = n\ \textbf{in}\ n' \mid \textbf{letrec}\ \overline{f : P = e}\ \textbf{in}\ n$$

Furthermore, we update the abstract syntax of top-level definitions as follows.

$$\textit{Command} \ ::= \ c\ :\ \forall \overline{Z}.\overline{A \to B}$$

We now come to two modified typing rules. When inferring the type of a command use as described by Ty-Command, we have to consider a command now as being of polytype, not a mere type any more. We can thus infer *any* instantiation of this polytype.

$$\textsc{Ty-Command}\quad \frac{c : \forall \overline{Z}.\overline{A \to} \; B \in \Sigma \qquad \overline{\Gamma[\Sigma] \vdash n : A[\overline{R}/\overline{Z}]}}{\Gamma[\Sigma] \vdash c\ \overline{R}\ \overline{n} : B[\overline{R}/\overline{Z}]}$$

A similar change is done to the handling counterpart: When checking that a command pattern matches its corresponding adjustment in the Ty-P-Request rule, the type scheme of a command can simply be stripped because the type variables must be considered as arbitrarily instantiated.

$$\textsc{Ty-P-Request}\quad \frac{c : \forall \overline{Z}.\overline{A \to} B \in \varnothing \oplus \Delta \qquad (p_i : A_i \dashv \Gamma_i)_i}{\langle c\ \overline{p} \to z\rangle : \langle \Delta \rangle B' \dashv [\Sigma]\ \overline{\Gamma}, z : \langle \iota \rangle B \to [\Sigma \oplus \Delta] B'}$$

At last, we come to the dynamic semantics where we update the syntax of normal forms and evaluation contexts as well as a rule for bindings during pattern matching.

$$\begin{aligned} &\text{(normal forms)} && u ::= w \mid \mathcal{E}[c\ \overline{R}\ \overline{w}] \\ &\text{(evaluation contexts)} && \mathcal{E} ::= \bullet \mid \mathcal{E}\ \overline{n} \mid \{e\}\ (\overline{u}, \mathcal{E}, \overline{n}) \mid k\ (\overline{w}, \mathcal{E}, \overline{n}) \mid c\ \overline{R}\ (\overline{w}, \mathcal{E}, \overline{n}) \mid \\ &&& \quad \mathbf{let}\ f : P = \mathcal{E}\ \mathbf{in}\ n \mid (\mathcal{E} : A) \end{aligned}$$

$$\textsc{Bind-P-Request}\quad \frac{c : \forall \overline{Z}.\overline{B \to} \; C \in \Delta \qquad p_i : B_i \leftarrow w_i \dashv \sigma_i}{\langle c\ \overline{p}\ \to\ z\rangle : \langle \Delta \rangle A \leftarrow \mathcal{E}[c\ \overline{R}\ \overline{w}] \dashv [\Sigma]\ \sigma_1, ..., \sigma_n, \{z \mapsto (\{x \to \mathcal{E}[x]\} : \{C \to [\Sigma \oplus \Delta]A\})\}}$$

For the Bind-P-Request rule, nothing changes for the resulting mono-value substitutions as the handling expressions must handle commands of arbitrary instantiation.

5.2 Application: Exception Handling

Another useful application is providing convenient exception handling in Frank without resorting to additional language primitives. Consider the following interface marking the ability of raising exceptions.

```
interface Exception E = raise X: E -> X
```

The command `raise` is expected to produce a value for any type `X` whenever executed. There is of course no handler that can satisfy this, but the crux is that never to continue after a call of `raise` is exactly what one expects from an exception handler. The following presents 1) a fallible example operator `boolToInt` and 2) a handler `try` that executes a fallible program and returns `Either` its result or the raised exception.

```
boolToInt: {Bool -> [Exception String]Bool}
boolToInt "false" = false
boolToInt "true"  = true
boolToInt _   = raise "not␣a␣bool"

try: {<Exception E>X -> Either E X}
try <raise e -> _> = Left e
try x              = Right x
```

Without much effort we have integrated exception handling in a flexible manner: Any type can serve as an exception and exceptions can be caught and forwarded, using merely the core facilites of Frank.

5.2.1 Multiple Instances of Exceptions

We mention one shortcoming of exception handling in the current Frank setting. It is not possible to deal with multiple instantiations of the `Exception` interface at the same time. The reason for this is that different instantiations shadow each other and only the most recent one is active, e.g. `[Exception String, Exception Int]` behaves as `[Exception Int]`. This issue can be addressed by introducing interface instances which are distinguishable entities and do not shadow each other. Eff [2] is an example of a language that supports effect instances.

5.2.2 Non-informative Exceptions

A special case of exceptions — one without any exception object — is even simpler to define, use and handle.

```
interface Abort = abort X: X

stringToBool': {String -> [Abort]Bool}
stringToBool' "false" = false
stringToBool' "true"  = true
stringToBool' _   = abort!

try': {<Abort>X -> Maybe X}
try' <abort -> _> = Nothing
try' x            = Just x
```

This particular example can be directly compared to an example given in the original Frank paper [20]. Having no polymorphic commands at hand, the abort functionality had to be implemented on top of a vacuous pattern match to make `stringToBool'` and `try'` work in the same way (both definitions `stringToBool'` and `try'` are still valid when redefining `Abort` and `abort` as follows).

```
data Zero =
interface Abort = aborting: Zero

abort: {[Abort]X}
abort! = case aborting! {}
```

Although this does not save many lines of code in this example, it becomes clear that polymorphic commands do not only enlarge the spectrum of what can be implemented but can also offer a more direct way of expression.

Chapter 6

Type Soundness

We provide a proof of *type soundness* of the new Frank setting (i.e., including the enhancements) in the following. It guarantees that a terminating well-typed program yields a value of the same type. Next to classical value-type-safety this implies in particular effect-type safety, because any commands not provided in the ambient ability are guaranteed not to occur. No dangling unexpected command requests are therefore possible.

In the following, we first reiterate the syntax, typing rules and operational semantics due to the changes we have done to them in the last Chapters. Since they have been explained before, we only state the definitions and and highlight the updated parts. After introducing and proving further preliminaries, we establish *type soundness* by dividing it into two parts, *type preservation* and *progress*, and proving those properties separately.

$$
\begin{array}{rcl}
\mathit{Definition} & ::= & \mathit{DatatypeDef} \mid \mathit{InterfaceDef} \mid \mathit{TermDef} \mid \mathit{OperatorDef} \\
\mathit{DatatypeDef} & ::= & \mathbf{data}\ D\ \overline{Z}\ =\ \overline{\mathit{Constructor}} \\
\mathit{Constructor} & ::= & k\ \overline{A} \\
\mathit{InterfaceDef} & ::= & \mathbf{interface}\ I\ \overline{Z}\ =\ \overline{\mathit{Command}} \\
\mathit{Command} & ::= & c\ :\ \forall \overline{Z}.\overline{A \to B} \\
\mathit{TermDef} & ::= & f\ :\ P \\
 & & f = n \\
\mathit{OperatorDef} & ::= & f\ :\ \forall \overline{Z}.\{C\} \\
 & & \overline{f\ \overline{r}\ =\ n}
\end{array}
$$

Figure 6.1: Abstract Syntax of Top-Level Definitions (updated)

Types

$$
\begin{array}{lll}
\text{(value types)} & A, B & ::= D\ \overline{R} \mid \{C\} \mid X \\
\text{(computation types)} & C & ::= \overline{T \to}\ G \\
\text{(ports)} & T & ::= \langle \Delta \rangle A \\
\text{(pegs)} & G & ::= [\Sigma] A \\
\text{(type variables)} & Z & ::= X \mid [E] \\
\text{(type arguments)} & R & ::= A \mid [\Sigma] \\
\text{(polytype)} & P & ::= \forall \overline{Z}.A \\
\text{(abilities)} & \Sigma & ::= \varnothing \mid E \mid \Sigma, I\ \overline{R} \\
\text{(adjustments)} & \Delta & ::= \iota \mid \Delta + I\ \overline{R} \\
\text{(type environments)} & \Gamma & ::= \cdot \mid \Gamma, x : A \mid \Gamma, f : P
\end{array}
$$

with data types D
interfaces I
value type var.s X
effect type var.s E

Terms

$$
\begin{array}{lll}
\text{(uses)} & m & ::= x \mid f\ \overline{R} \mid m\ \overline{n} \mid (n : A) \\
\text{(constructions)} & n & ::= m \mid k\ \overline{n} \mid c\ \overline{R}\ \overline{n} \mid \{e\} \\
 & & \mid \textbf{let}\ f : P = n\ \textbf{in}\ n' \\
 & & \mid \textbf{letrec}\ \overline{f : P = e}\ \textbf{in}\ n \\
\text{(computations)} & e & ::= \overline{\overline{r} \to n} \\
\text{(comp. patterns)} & r & ::= p \mid \langle c\ \overline{p} \to z \rangle \mid \langle x \rangle \\
\text{(value patterns)} & p & ::= k\ \overline{p} \mid x
\end{array}
$$

with term var.s x, y, z
polyterm var.s f
constructors k
commands c

Figure 6.2: Abstract Syntax of Types and Terms [20] (updated)

$$
\begin{array}{lll}
\text{(use values)} & v & ::= (w : A) \\
\text{(construction values)} & w & ::= v \mid k\ \overline{w} \mid \{e\} \\
\text{(normal forms)} & u & ::= w \mid \mathcal{E}[c\ \overline{R}\ \overline{w}] \\
\text{(evaluation contexts)} & \mathcal{E} & ::= \bullet \mid \mathcal{E}\ \overline{n} \mid \{e\}\ (\overline{u}, \mathcal{E}, \overline{n}) \mid k\ (\overline{w}, \mathcal{E}, \overline{n}) \mid c\ \overline{R}\ (\overline{w}, \mathcal{E}, \overline{n}) \mid \\
 & & \quad \textbf{let}\ f : P = \mathcal{E}\ \textbf{in}\ n \mid (\mathcal{E} : A) \\
\text{(mono-value substitutions)} & \sigma & ::= \varnothing \mid \sigma, [x \mapsto (w : A)]
\end{array}
$$

$$
\begin{array}{ll}
TyEnv : & \sigma \mapsto \Gamma \\
TyEnv\ \ \overline{[x \mapsto (w : A)]} = \overline{x : A}
\end{array}
$$

Figure 6.3: Further Syntax w.r.t. Dynamic Semantics (updated)

$\boxed{\Gamma[\Sigma]\vdash m \Rightarrow A}$ *For environment* Γ *and ambient* Σ*, use* m *is inferred to have type* A*.*

$$\textsc{Ty-Var}\ \frac{x : A \in \Gamma}{\Gamma[\Sigma]\vdash x \Rightarrow A} \qquad \textsc{Ty-PolyVar}\ \frac{f : \forall \overline{Z}.A \in \Gamma}{\Gamma[\Sigma]\vdash f\ \overline{R} \Rightarrow A[\overline{R}/\overline{Z}]} \qquad \textsc{Ty-Coerce}\ \frac{\Gamma[\Sigma]\vdash n : A}{\Gamma[\Sigma]\vdash (n : A) \Rightarrow A}$$

$$\textsc{Ty-App}\ \frac{\Gamma[\Sigma]\vdash m \Rightarrow \{\overline{\langle\Delta\rangle A \to [\Sigma']B}\} \qquad \Sigma' \sim \Sigma \qquad \overline{\Gamma[\Sigma \oplus \Delta]\vdash n : A}}{\Gamma[\Sigma]\vdash m\ \overline{n} \Rightarrow B}$$

$\boxed{\Gamma[\Sigma]\vdash n : A}$ *For environment* Γ *and ambient* Σ*, construction* n *is checked to have* type A*.*

$$\textsc{Ty-Switch}\ \frac{\Gamma[\Sigma]\vdash m \Rightarrow A \qquad A = B}{\Gamma[\Sigma]\vdash m : B} \qquad \textsc{Ty-Data}\ \frac{k\ \overline{A} \in D\ \overline{R} \qquad \overline{\Gamma[\Sigma]\vdash n : A}}{\Gamma[\Sigma]\vdash k\ \overline{n} : D\ \overline{R}}$$

$$\textsc{Ty-Command}\ \frac{c : \forall \overline{Z}.\overline{A \to}\ B \in \Sigma \qquad \overline{\Gamma[\Sigma]\vdash n : A[\overline{R}/\overline{Z}]}}{\Gamma[\Sigma]\vdash c\ \overline{R}\ \overline{n} : B[\overline{R}/\overline{Z}]} \qquad \textsc{Ty-Let}\ \frac{P = \forall \overline{Z}.A \qquad \Gamma[\Sigma]\vdash n : A \qquad \Gamma, f : P\ [\Sigma]\vdash n' : B}{\Gamma[\Sigma]\vdash \mathbf{let}\ f : P = n\ \mathbf{in}\ n' : B}$$

$$\textsc{Ty-Operator}\ \frac{\Gamma \vdash e : C}{\Gamma[\Sigma]\vdash \{e\} : \{C\}} \qquad \textsc{Ty-LetRec}\ \frac{\overline{P = \forall \overline{Z}.\{C\}} \qquad \overline{\Gamma, \overline{f : P} \vdash e : C} \qquad \Gamma, \overline{f : P}\ [\Sigma]\vdash n : B}{\Gamma[\Sigma]\vdash \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n : B}$$

$\boxed{\Gamma \vdash e : C}$ *For environment* Γ *and ambient* Σ*, comp.* e *is checked to have comp. type* C*.*

$$\textsc{Ty-Comp}\ \frac{(r_{i,j} : T_j \dashv[\Sigma]\ \Gamma'_{i,j})_{i,j} \qquad (\Gamma, (\Gamma'_{i,j})_j\ [\Sigma]\vdash n_i : B)_i \qquad ((r_{i,j})_{i,j})\ \text{covers}\ (T_j)_j}{\Gamma \vdash ((r_{i,j})_j \to n_i)_i : (T_j \to)_j\ [\Sigma]B}$$

$\boxed{p : A \dashv \Gamma}$ *Value pattern* p *matches values of type* A *and provides bindings* Γ*.*

$$\textsc{Ty-P-Var}\ \frac{}{x : A \dashv x : A} \qquad \textsc{Ty-P-Data}\ \frac{k\ \overline{A} \in D\ \overline{R} \qquad \overline{p : A \dashv \Gamma}}{k\ \overline{p} : D\ \overline{R} \dashv \overline{\Gamma}}$$

$\boxed{r : T \dashv[\Sigma]\ \Gamma}$ *Comp. pat.* r *matches comp.s of port* T *under amb.* Σ *and provides bind.s* Γ*.*

$$\textsc{Ty-P-Value}\ \frac{p : A \dashv \Gamma}{p : \langle\Delta\rangle A \dashv[\Sigma]\ \Gamma} \qquad \textsc{Ty-P-Request}\ \frac{c : \forall \overline{Z}.\overline{A \to} B \in \varnothing \oplus \Delta \qquad (p_i : A_i \dashv \Gamma_i)_i}{\langle c\ \overline{p} \to z\rangle : \langle\Delta\rangle B' \dashv[\Sigma]\ \overline{\Gamma}, z : \langle\iota\rangle B \to [\Sigma \oplus \Delta]B'}$$

$$\textsc{Ty-P-CatchAll}\ \frac{}{\langle x\rangle : \langle\Delta\rangle A \dashv[\Sigma]\ x : \{[\Sigma \oplus \Delta]A\}}$$

Figure 6.4: Typing Rules [20] (updated)

$\boxed{p : A \leftarrow w \dashv \sigma}$ *Value pattern p for type A matches w and binds σ.*

$$\textsc{Bind-P-Var}\ \frac{}{x : A \leftarrow w \dashv \{x \mapsto (w : A)\}} \qquad \textsc{Bind-P-Data}\ \frac{k\ \overline{A} \in D\ \overline{R} \qquad p_i : A_i \leftarrow w_i \dashv \sigma_i}{k\ \overline{p} : D\ \overline{R} \leftarrow k\ \overline{w} \dashv \sigma_1, ..., \sigma_n}$$

$\boxed{r : T \leftarrow u \dashv[\Sigma]\ \sigma}$ *Comp. pattern r for peg T matches w under ambient Σ and binds σ.*

$$\textsc{Bind-P-Value}\ \frac{p : A \leftarrow w \dashv \sigma}{p : \langle\Delta\rangle A \leftarrow w \dashv[\Sigma]\ \sigma}$$

$$\textsc{Bind-P-Request}\ \frac{c : \forall \overline{Z}.\overline{B \rightarrow} C \in \Delta \qquad p_i : B_i \leftarrow w_i \dashv \sigma_i}{\langle c\ \overline{p}\ \rightarrow\ z\rangle : \langle\Delta\rangle A \leftarrow \mathcal{E}[c\ \overline{R}\ \overline{w}] \dashv[\Sigma]\ \sigma_1, ..., \sigma_n, \{z \mapsto (\{x \rightarrow \mathcal{E}[x]\} : \{C \rightarrow [\Sigma \oplus \Delta]A\})\}}$$

$$\textsc{Bind-P-CatchAll}\ \frac{}{\langle x\rangle : \langle\Delta\rangle A \leftarrow u \dashv[\Sigma]\ \{x \mapsto (\{u\} : \{[\Sigma \oplus \Delta]A\})\}}$$

Figure 6.5: Pattern Binding (updated)

$$\textsc{Step-Handle}\ \frac{(r_{i,j} : T_j \leftarrow u_j \dashv[\Sigma]\ \sigma_{i,j})_{i,j} \qquad i \text{ is minimal}}{\left(\begin{array}{l}\{r_{1,1}\ ...\ r_{1,l} \rightarrow n_1 \\ \mid ... \qquad\qquad : \{\overline{T \rightarrow}\ [\Sigma]B\} \\ \mid\ r_{k,1}\ ...\ r_{k,l} \rightarrow n_k\}\end{array}\right) u_1\ ...\ u_l\ \rightarrow\ n_i[\sigma_{i,1}, ..., \sigma_{i,l}]}$$

$$\textsc{Step-Let}\ \frac{}{\textbf{let}\ f : P = w\ \textbf{in}\ n \rightarrow n[(w : P)/f]}$$

$$\textsc{Step-LetRec}\ \frac{\overline{e = \overline{\overline{r} \rightarrow n}}}{\textbf{letrec}\ \overline{f : P = e}\ \textbf{in}\ n' \rightarrow n'[(\{\overline{r} \rightarrow \textbf{letrec}\ \overline{f : P = e}\ \textbf{in}\ n\} : P)/f]}$$

$$\textsc{Step-Coerce}\ \frac{}{(v : A) \rightarrow v} \qquad \textsc{Step-Ctx}\ \frac{n \rightarrow n'}{\mathcal{E}[n] \rightarrow \mathcal{E}[n']}$$

Figure 6.6: Small-Step Reduction Relation

6.1 Type Preservation

Type preservation means that reduction preserves typing for well-typed terms, i.e., after doing a reduction step, a well-typed term still has the same type. Before proving this property, we need to introduce some preliminaries. First of all, contexts need to be assigned context types. A context type of the form $A \Rrightarrow B$ specifies that when a context's hole is filled by a term of type A, then the resulting term is of type B. We have one rule per context form.

First, when filling the atomic context, the resulting term has the type of the hole (Ctx-Hole). The other rules lift up the hole's type to composed contexts. Let us examine one example, Ctx-App-L. The type of a context of the form $\mathcal{E}\ \overline{n}$ is determined by the type of operator $\mathcal{E}$ and the types of arguments $\overline{n}$. Because the operator is a context, its hole type A fixes the hole type for the expression $\mathcal{E}\ \overline{n}$. The result type B is determined by the operator's result type, in alignment to the normal typing rule Ty-App. The other rules follow this pattern of lifting the hole type while following the normal typing rules.

$\boxed{\Gamma[\Sigma]\vdash \mathcal{E} : A \Rrightarrow B}$ For environment Γ and ambient Σ, context $\mathcal{E}$ with hole type A has type B.

$$\textsc{Ctx-Hole}\ \frac{}{\Gamma[\Sigma]\vdash \bullet : A \Rrightarrow A} \qquad \textsc{Ctx-Coerce}\ \frac{\Gamma[\Sigma]\vdash \mathcal{E} : A \Rrightarrow B}{\Gamma[\Sigma]\vdash (\mathcal{E} : B) : A \Rrightarrow B}$$

$$\textsc{Ctx-App-L}\ \frac{\Gamma[\Sigma]\vdash \mathcal{E} : A \Rrightarrow \{\overline{\langle\Delta'\rangle B' \to B}\} \qquad \overline{\Gamma[\Sigma\oplus\Delta']\vdash n : B'}}{\Gamma[\Sigma]\vdash \mathcal{E}\ \overline{n} : A \Rrightarrow B}$$

$$\textsc{Ctx-App-R}\ \frac{\Gamma[\Sigma]\vdash \{e\} : \{\overline{\langle\Delta'\rangle B' \to} \langle\Delta''\rangle B'' \to \overline{\langle\Delta'''\rangle B''' \to} [\Sigma]B\} \quad \overline{\Gamma[\Sigma\oplus\Delta']\vdash u : B'} \quad \overline{\Gamma[\Sigma\oplus\Delta'']\vdash \mathcal{E} : A \Rrightarrow B''} \quad \overline{\Gamma[\Sigma\oplus\Delta''']\vdash n : B'''}}{\Gamma[\Sigma]\vdash \{e\}\ (\overline{u}, \mathcal{E}, \overline{n}) : A \Rrightarrow B}$$

$$\textsc{Ctx-Data}\ \frac{k\ (\overline{B}, B', \overline{B''}) \in D\ \overline{R} \qquad \overline{\Gamma[\Sigma]\vdash w : B} \qquad \overline{\Gamma[\Sigma]\vdash \mathcal{E} : A \Rrightarrow B'} \qquad \overline{\Gamma[\Sigma]\vdash n : B''}}{\Gamma[\Sigma]\vdash k\ (\overline{w}, \mathcal{E}, \overline{n}) : A \Rrightarrow D\ \overline{R}}$$

$$\textsc{Ctx-Command}\ \frac{c : \forall\overline{Z}.\overline{B \to} B' \to \overline{B'' \to} B''' \in \Sigma \quad \overline{\Gamma[\Sigma]\vdash w : B[\overline{R}/\overline{Z}]} \quad \Gamma[\Sigma]\vdash \mathcal{E} : A \Rrightarrow B'[\overline{R}/\overline{Z}] \quad \overline{\Gamma[\Sigma]\vdash n : B''[\overline{R}/\overline{Z}]}}{\Gamma[\Sigma]\vdash c\ \overline{R}\ (\overline{w}, \mathcal{E}, \overline{n}) : A \Rrightarrow B'''[\overline{R}/\overline{Z}]}$$

$$\textsc{Ctx-Let}\ \frac{\Gamma[\Sigma]\vdash \mathcal{E} : A \Rrightarrow C \qquad \Gamma, f : \forall\overline{Z}.C\ [\Sigma]\vdash n : B}{\Gamma[\Sigma]\vdash \mathbf{let}\ f : \forall\overline{Z}.C = \mathcal{E}\ \mathbf{in}\ n : A \Rrightarrow B}$$

Figure 6.7: Context Typing Rules

Typing judgements form a congruence with respect to contexts. This important property means that given a filled context, we can invert the context-filling, swap the filling (thereby respecting the hole type) and have the same judgement for the re-filled context. While this is a standard lemma, we mention one particularity of this lemma due to Frank's effect type system. A hole that is nested within a larger context may have access to a larger ambient ability. We deal with this by acknowledging that the filled term has access to the ambient ability Σ extended by some accumulated adjustment Δ. In some sense we can view this particularity as a technical detail that will not explicitly come into our way later.

Lemma 6.1.1 (Context Inversion) *Let $\mathcal{E}$ be a context. Then*

1. $\Gamma[\Sigma]\vdash \mathcal{E}[m] \Rightarrow B$ *implies* $\exists\ A, \Delta.\ \Gamma[\Sigma\oplus\Delta]\vdash m \Rightarrow A$ *and* $\Gamma[\Sigma]\vdash \mathcal{E} : A \Rrightarrow B$ *and* $(\forall\ m'.\ \Gamma[\Sigma\oplus\Delta]\vdash m' \Rightarrow A$ *implies* $\Gamma[\Sigma]\vdash \mathcal{E}[m'] \Rightarrow B)$
2. $\Gamma[\Sigma]\vdash \mathcal{E}[n] : B$ *implies* $\exists\ A, \Delta.\ \Gamma[\Sigma\oplus\Delta]\vdash n : A$ *and* $\Gamma[\Sigma]\vdash \mathcal{E} : A \Rrightarrow B$ *and* $(\forall\ n'.\ \Gamma[\Sigma\oplus\Delta]\vdash n' : A$ *implies* $\Gamma[\Sigma]\vdash \mathcal{E}[n'] : B)$

Proof By induction on $\mathcal{E}$. ∎

Next, we establish substitutivity of typing judgements, with respect to both mono-value substitutions σ and poly-value substitutions. Substitutivity is a standard and indispensable property stating that substituting values for free variables in terms preserves typing judgements, given that the values are of the right type.

Lemma 6.1.2 (Typing is Substitutive)

1. $\Gamma, (x : B)\ [\Sigma]\text{–}\ m \Rightarrow A$ *implies* $\Gamma\ [\Sigma]\text{–}\ m[x \mapsto (v : B)] \Rightarrow A$
   and $\Gamma\ [\Sigma]\text{–}\ v \Rightarrow B$
2. $\Gamma, (x : B)\ [\Sigma]\text{–}\ n : A$ *implies* $\Gamma\ [\Sigma]\text{–}\ n[x \mapsto (w : B)] : A$
   and $\Gamma\ [\Sigma]\text{–}\ w : B$
3. $\Gamma, (f : \forall \overline{Z}.B)\ [\Sigma]\text{–}\ m \Rightarrow A$ *implies* $\Gamma\ [\Sigma]\text{–}\ m[(v : \forall \overline{Z}.B)/f] \Rightarrow A$
   and $\Gamma\ [\Sigma]\text{–}\ v \Rightarrow B$
4. $\Gamma, (f : \forall \overline{Z}.B)\ [\Sigma]\text{–}\ n : A$ *implies* $\Gamma\ [\Sigma]\text{–}\ n[(w : \forall \overline{Z}.B)/f] : A$
   and $\Gamma\ [\Sigma]\text{–}\ w : B$

Proof For all 1), 2), 3), 4) by induction on the first premise. ∎

We now focus on pattern matching, one of core elements of computation in Frank. As defined before (Figure 6.5), patterns applied to values result in mono-value substitutions. We refine the notion of a mono-value substitution by the property of *well-typedness*, which we can expect of the substitutions yielded by pattern-matching.

Definition 6.1.3 (Well-Typed Mono-Value Substitutions) *We say that a mono-value substitution σ is well-typed if for all its entries $[x \mapsto (w : A)]$ it holds that for any Γ and Σ, $\Gamma[\Sigma]\text{–}\ w : A$.*

The following lemma fuses static and dynamic judgements about pattern matching. Given a typing judgement for a pattern r and a judgement describing the resulting substitution σ when matched against a value w, the typing judgements must agree with the types annotated in σ. Furthermore, as σ binds closed values (i.e., without occurrences of free variables), σ is guaranteed to be well-typed.

Lemma 6.1.4 (Pattern Typing Correspondence)

1. $p : A \dashv \Gamma$ *implies* $TyEnv(\sigma) = \Gamma$
   and $p : A \leftarrow w \dashv \sigma$ *and σ is well-typed*
2. $r : \langle \Delta \rangle A \dashv [\Sigma]\ \Gamma$ *implies* $TyEnv(\sigma) = \Gamma$
   and $r : \langle \Delta \rangle A \leftarrow u \dashv [\Sigma]\ \sigma$ *and σ is well-typed*

Proof For 1) by induction on p and for 2) by case analysis on r, then using 1). ∎

We need one further connection between *equality-up-to-inactive-instantiations* ($\Sigma \sim \Sigma'$) and typing judgements. Because a term can only exhibit command requests of an *active* interface instantiation, only the *active fragment* of the ambient ability is essential for typing. This means that a typing judgements under Σ holds also under Σ' for $\Sigma \sim \Sigma'$.

Lemma 6.1.5 (Typing Up-to-Inactive-Instantiations)

1. $\Gamma[\Sigma]\vdash m \Rightarrow A$ *implies* $\Gamma[\Sigma']\vdash m \Rightarrow A$ *and* $\Sigma \sim \Sigma'$
2. $\Gamma[\Sigma]\vdash n : A$ *implies* $\Gamma[\Sigma']\vdash n : A$ *and* $\Sigma \sim \Sigma'$

Proof By mutual induction on the typing judgements. ∎

Finally, we prove type preservation.

Theorem 6.1.6 (Type Preservation) *Let n, n' be terms with $n \to n'$. Then*

1. $\Gamma[\Sigma]\vdash m \Rightarrow A$ *implies* $\Gamma[\Sigma]\vdash m' \Rightarrow A$ *for* $m = n$, $m' = n'$
2. $\Gamma[\Sigma]\vdash n : A$ *implies* $\Gamma[\Sigma]\vdash n' : A$

Proof Induction on $n \to n'$.

- (Step-Coerce) $(v : A') \to v$. We prove 1) and 2).
  1. Let $\Gamma[\Sigma]\vdash (v : A') \Rightarrow A$. By inversion (Ty-Coerce), $A = A'$ and $\Gamma[\Sigma]\vdash v : A$. By inversion (Ty-Switch), $\Gamma[\Sigma]\vdash v \Rightarrow A$.
  2. Let $\Gamma[\Sigma]\vdash (v : A') : A$. By inversion (Ty-Switch, Ty-Coerce), $A = A'$ and $\Gamma[\Sigma]\vdash v : A$.
- (Step-Ctx) $\mathcal{E}[n] \to \mathcal{E}[n']$ with $n \to n'$. We prove 1) and 2).
  1. Let $\Gamma[\Sigma]\vdash \mathcal{E}[m] \Rightarrow B$. We have to show $\Gamma[\Sigma]\vdash \mathcal{E}[m'] \Rightarrow B$.
     By Lemma 6.1.1, there exist A, Δ s.t. $\Gamma[\Sigma \oplus \Delta]\vdash m \Rightarrow A$ and $(\forall\, m'.\ \Gamma[\Sigma \oplus \Delta]\vdash m' \Rightarrow A$ implies $[\Sigma]\vdash \mathcal{E}[m'] \Rightarrow B)$.
     By induction, we have $\Gamma[\Sigma \oplus \Delta]\vdash m' \Rightarrow A$. This implies $[\Sigma]\vdash \mathcal{E}[m'] \Rightarrow B$.
  2. Let $\Gamma[\Sigma]\vdash \mathcal{E}[n] : B$. We have to show $\Gamma[\Sigma]\vdash \mathcal{E}[n'] : B$.
     By Lemma 6.1.1, there exist A, Δ s.t. $\Gamma[\Sigma \oplus \Delta]\vdash n : A$ and $(\forall\, n'.\ \Gamma[\Sigma \oplus \Delta]\vdash n' : A$ implies $[\Sigma]\vdash \mathcal{E}[n'] \Rightarrow B)$.
     By induction, we have $\Gamma[\Sigma \oplus \Delta]\vdash n' : A$. This implies $[\Sigma]\vdash \mathcal{E}[n'] : B$.
- (Step-Handle) $\left(\begin{array}{l}\{r_{1,1}\ ...\ r_{1,l} \to n_1\\ \mid ... \qquad\qquad : \{\overline{T \to}\ [\Sigma']B\}\\ \mid r_{k,1}\ ...\ r_{k,l} \to n_k\}\end{array}\right) u_1\ ...\ u_l \ \to\ n_i[\sigma_{i,1}, ..., \sigma_{i,l}]$

  with $(r_{i,j} : T_j \leftarrow u_j \dashv[\Sigma']\ \sigma_{i,j})_{i,j}$. We prove 1) and 2).
  1. Let $\Gamma[\Sigma]\vdash (m : \{\overline{T \to}\ [\Sigma']B\})\ \overline{u} \Rightarrow B'$. We have to show $\Gamma[\Sigma]\vdash n_i[\sigma_{i,1}, ..., \sigma_{i,l}] \Rightarrow B'$.
     By inversion (Ty-App, Ty-Coerce), we have $B = B'$ and $\Sigma' \sim \Sigma$ and $\Gamma[\Sigma']\vdash m : \{\overline{T \to}\ [\Sigma]B\}$.
     By inversion (Ty-Operator, Ty-Comp), we have $(r_{i,j} : T_j \dashv[\Sigma']\ \Gamma'_{i,j})_{i,j}$ and $(\Gamma, (\Gamma'_{i,j})_j)\ [\Sigma']\vdash n_i : B)_i$.
     Together with the premise $(r_{i,j} : T_j \leftarrow u_j \dashv[\Sigma']\ \sigma_{i,j})_{i,j}$, we have $TyEnv(\sigma_{i,1}, ..., \sigma_{i,l}) = \Gamma'_{i,1}, ..., \Gamma'_{i,l}$ and well-typedness of $\sigma_{i,j}$ by Lemma 6.1.4.
     By Lemma 6.1.2, it suffices to show $(\Gamma, TyEnv(\sigma_{i,1}, ..., \sigma_{i,l})[\Sigma]\vdash n_i \Rightarrow B)_i$. This is equivalent to showing $(\Gamma, (\Gamma'_{i,1}, ..., \Gamma'_{i,l})[\Sigma]\vdash n_i \Rightarrow B)_i$. By $(\Gamma, (\Gamma'_{i,j})_j)\ [\Sigma']\vdash n_i : B)_i$ and $\Sigma' \sim \Sigma$ we can conclude with Lemma 6.1.5.
  2. Let $\Gamma[\Sigma]\vdash (m : \{\overline{T \to}\ [\Sigma]B\})\ \overline{u} : B'$. We have to show $\Gamma[\Sigma]\vdash n_i[\sigma_{i,1}, ..., \sigma_{i,l}] : B'$.
     By inversion (Ty-Switch), we have $\Gamma[\Sigma]\vdash (m : \{\overline{T \to}\ [\Sigma]B\})\ \overline{u} \Rightarrow B'$. By the same argument as in 1), we obtain $\Gamma[\Sigma]\vdash n_i[\sigma_{i,1}, ..., \sigma_{i,l}] \Rightarrow B'$. We conclude by Ty-Switch.

- (Step-Let) **let** $f : P = w$ **in** $n \to n[(w : P)/f]$. By inversion, we only need to prove 2). Let $\Gamma[\Sigma]\vdash$ **let** $f : P = w$ **in** $n : B$. We have to show $\Gamma[\Sigma]\vdash n[(w : P)/f] : B$.
  By inversion (Ty-Let) we have $P = \forall \overline{Z}.A$ and $\Gamma[\Sigma]\vdash w : A$ and $\Gamma, f : P\ [\Sigma]\vdash n : B$. We conclude by Lemma 6.1.2.

- (Step-LetRec) **letrec** $\overline{f : P = e}$ **in** $n' \to n'[\overline{(\{\overline{r} \to \textbf{letrec } \overline{f : P = e} \textbf{ in } n\} : P)/f}]$ with $\overline{e = \overline{r} \to n}$.
  By inversion, we only need to prove 2). Let $\Gamma[\Sigma]\vdash$ **letrec** $\overline{f : P = e}$ **in** $n' : B$.
  We have to show $\Gamma[\Sigma]\vdash n'[\overline{(\{\overline{r} \to \textbf{letrec } \overline{f : P = e} \textbf{ in } n\} : P)/f}] : B$.
  We argue in the following for every $(f_i)_i$ in the above substitution. By inversion (Ty-LetRec), we have $\overline{P = \forall \overline{Z}.\{\overline{\langle\Delta\rangle A \to A'}\}}$ and $\overline{\Gamma, \overline{f : P} \vdash e : \overline{\langle\Delta\rangle A \to A'}}$ and $\Gamma, \overline{f : P}\ [\Sigma]\vdash n' : B$.
  By Lemma 6.1.2, it suffices to show that $\Gamma[\Sigma]\vdash \overline{\{\overline{r} \to \textbf{letrec } \overline{f : P = e} \textbf{ in } n\}} : \{\overline{\langle\Delta\rangle A \to A'}\}$.
  By Ty-Operator, it suffices to show that $\Gamma[\Sigma]\vdash \overline{r} \to \textbf{letrec } \overline{f : P = e} \textbf{ in } n : \overline{\langle\Delta\rangle A \to A'}$.
  We can show this via Ty-Comp by providing the required premises through inversion of $\overline{\Gamma, \overline{f : P} \vdash e : \overline{\langle\Delta\rangle A \to A'}}$ which also gives us $(\Gamma, \overline{f : P}, (\Gamma'_{i,j})_j\ [\Sigma]\vdash n_i : A')_i$. It remains to show the second premise, $(\Gamma, (\Gamma'_{i,j})_j\ [\Sigma]\vdash$ **letrec** $\overline{f : P = e}$ **in** $n_i : A')_i$. We have this by Ty-LetRec and by weakening of type environment. ∎

6.2 Progress

Progress means that a well-typed term either is *normal* or can progress (i.e., reduce). Usually, *values* and *normal terms* are used synonymously. We make a distinction by having normal terms contain — next to values — terms whose next evaluation step requires the handling of a command, $\mathcal{E}[c\ \overline{w}]$. Bestowing this class of syntactic constructs with a constraint to an ambient ability, we have — next to values — a class of terms that we do not perceive as stuck.

Definition 6.2.1
For a term n with either $\Gamma[\Sigma]\vdash n \Rightarrow A$ or $\Gamma[\Sigma]\vdash n : A$, we say:

- *n is normal w.r.t. Σ if it is a value w or of the form $\mathcal{E}[c\ \overline{w}]$ where $c : \overline{A \to}\ B \in \Sigma$.*
- *n is non-stuck w.r.t. Σ if n is normal w.r.t. Σ or $\exists\ n'.\ n \to n'$.*

We need one more property of pattern matching that is related to progress in the sense that pattern matching must succeed in some way. We required in our typing rules (Figure 6.4) that the cases for each pattern-matching operator are exhaustive. Because we have not formally defined what it means for a pattern r to *cover* a type A, we cannot formally reason about it. We argue nevertheless that exhaustive pattern matching results in a binding σ_k for some k-th case.

Proposition 6.2.2 (Pattern Matching)

1. *Let $(p_i : A \dashv \Gamma'_i)_i$ be a collection of value patterns that cover A. Furthermore, let $\Gamma[\Sigma]\vdash w : A$. Then there exist k and σ_k s.t. $p_k : A \leftarrow w \dashv \sigma_k$ and k minimal.*
2. *Let $(r_i : \langle\Delta\rangle A \dashv[\Sigma]\ \Gamma'_i)_i$ be a collection of computation patterns that cover A. Furthermore, let $\Gamma[\Sigma \oplus \Delta]\vdash u : A$. Then there exist k and σ_k s.t. $r_k : \langle\Delta\rangle A \leftarrow u \dashv[\Sigma]\ \sigma_k$ and k minimal.*

Finally, we prove progress which together with type preservation gives type soundness.

Theorem 6.2.3 (Progress) *Let n be construction. Then:*

1. $\varnothing[\Sigma]\vdash m \Rightarrow A$ *implies* m *is non-stuck w.r.t.* Σ *for* $m = n$, $m' = n'$
2. $\varnothing[\Sigma]\vdash n : A$ *implies* n *is non-stuck w.r.t.* Σ

Proof Induction on n.

- n is a use m. We prove 1) and 2).
  1. Let $\varnothing[\Sigma]\vdash m \Rightarrow A$. By inversion, m can neither be x nor $f\ \overline{R}$ and we distinguish between two remaining cases.
     (a) (Ty-Coerce) $\varnothing[\Sigma]\vdash (n : A) \Rightarrow A$ with $\varnothing[\Sigma]\vdash n : A$. Then by induction, n is non-stuck w.r.t. Σ.
     - Let n be normal w.r.t. Σ. Then $(n : A)$ is normal w.r.t. Σ, too.
     - Let $n \to n'$. Then $(n : A) \to (n' : A)$ by Step-Ctx.

     (b) (Ty-App) $\varnothing[\Sigma]\vdash m\,\overline{n} \Rightarrow B$ with $\varnothing[\Sigma]\vdash m \Rightarrow \{\overline{\langle\Delta\rangle A \to [\Sigma']B}\}$ and $\overline{\varnothing[\Sigma\oplus\Delta]\vdash n : A}$ and $\Sigma' \sim \Sigma$.
     By induction, m is non-stuck w.r.t. Σ.
     - Let m be normal w.r.t. Σ. By induction, every n_i is non-stuck w.r.t. $\Sigma\oplus\Delta$. We distinguish two cases.
       * There exist k, n_k' s.t. $n_k \to n_k'$ and k is minimal.
         Then $m\ (n_1, ..., n_{k-1}, n_k, n_{k+1}, ...n_l) \to m\ (n_1, ..., n_{k-1}, n_k', n_{k+1}, ...n_l)$ by Step-Ctx.
       * All $n_j = u_j$ are normal w.r.t. $\Sigma\oplus\Delta$. Case distinction on normal m.
         · $m = (w : A')$. By inversion (Ty-Coerce), $\varnothing[\Sigma]\vdash w : \{\overline{\langle\Delta\rangle A \to [\Sigma']B}\}$. Again by inversion, w can only be of the form $\underline{w = \{e\}}$. Again, inversion (Ty-Operator, Ty-Comp) yields $\varnothing \vdash e : \overline{\langle\Delta\rangle A \to [\Sigma']B}$ with $(r_{i,j} : \langle\Delta_j\rangle A_j \dashv[\Sigma']\,\Gamma'_{i,j})_{i,j}$ and $(\Gamma'_{i,j})_j\ [\Sigma']\vdash n_i' : B)_i$ and $((r_{i,j})_{i,j})$ covers $(\langle\Delta_j\rangle A_j)_j$.

         We show that there exist minimal i, (σ_j) s.t. $(\{e\} : \{\overline{\langle\Delta\rangle A \to [\Sigma']B}\})\,\overline{n} \to n_i'[\overline{\sigma_j}]$. It suffices to show $(r_{i,j} : T_j \leftarrow u_j \dashv[\Sigma']\ \sigma_{i,j})_{i,j}$ for a minimal i. This is given by Proposition 6.2.2.
         · $m = \mathcal{E}[c\ \overline{w}]$. Then $m\ \overline{n} = (\mathcal{E}\ \overline{n})[c\ \overline{w}]$ is normal w.r.t. Σ.
     - Let $m \to m'$. Then $m\ \overline{n} \to m'\ \overline{n}$ by Step-Ctx.
  2. Let $\varnothing[\Sigma]\vdash m : A$. By inversion (Ty-Switch), we have $\varnothing[\Sigma]\vdash m \Rightarrow A$. By the same argument as in 1), we have that m is non-stuck w.r.t. Σ.
- $n = k\ \overline{n'}$. By inversion (Ty-Data) and induction, each n_i' is non-stuck w.r.t. Σ. If all n_i' are normal w.r.t. Σ, then $k\ \overline{n'}$ is normal w.r.t. Σ, too. Otherwise, $n_l' \to n_l''$ for minimal l and by Step-Ctx, $k\ \overline{n'}$ can do a step, too.
- $n = c\ \overline{R}\ \overline{n'}$. By inversion (Ty-Command) and induction, each n_i' is non-stuck w.r.t. Σ. Furthermore, we have $c : \forall\overline{Z}.\overline{A \to B} \in \Sigma$. If all n_i' are normal w.r.t. Σ, then $c\ \overline{R}\ \overline{n'}$ is normal w.r.t. Σ, too. Otherwise, $n_l' \to n_l''$ for minimal l and by Step-Ctx, $c\ \overline{R}\ \overline{n'}$ can do a step, too.
- $n = \{e\}$, which is normal w.r.t. Σ.
- $n = \mathbf{let}\ f : P = n'\ \mathbf{in}\ n''$. By inversion (Ty-Let) and induction, n' is non-stuck w.r.t. Σ. If n' is normal w.r.t. Σ, then by Step-Let, n can do a step. Otherwise there is a n''' s.t. $n' \to n'''$ and by Step-Ctx, n can do a step.
- $n = \mathbf{letrec}\ \overline{f : P = e}\ \mathbf{in}\ n'$. Then by Step-Letrec, n can do a step. ∎

Chapter 7

Generative Effects and Applications in Concurrency

Generative effects describe the ability to request freshly generated and thereby unique values of a type. The perhaps most prominent example is referentiable state, where fresh memory cells are issued on request. The perhaps simplest example is the ability to request fresh tokens which can be used as unique identifiers. Because referentiable state is an ability which relies upon the newly introduced polymorphic commands and sets the stage for a range of possibilities, we explore this construct in more detail in the following. Furthermore, we demonstrate the expressiveness gained by both referentiable state and polymorphic commands in the field of concurrency by giving implementations of a variant of the actor model and promises.

7.1 Referentiable State

Thanks to polymorphic commands, an interface providing ML-style memory allocation is easily defined.

```
data Ref X =

interface RefState = new X:   X -> Ref X
                   | read X:  Ref X -> X
                   | write X: Ref X -> X -> Unit
```

Both definitions are built-in because the only place to eventually handle `RefState` lies — in the current setting — outside of pure Frank, in the same way as the `IO` monad is executed outside of pure Haskell. This also justifies that `Ref X` has no data constructors: Reference cells cannot be constructed inside but only generated outside of Frank.

7.1.1 Imperative Programming

The built-in `RefState` allows programming in an imperative style. The only main primitive missing for efficient data manipulation are arrays, which can be added similarly to how standard memory cells have been added. Besides this missing feature, imperative programming is enabled just as in ML. This comprises the ability to define control structures like the `while`-loop in terms of Frank. The following gives the definition of `while` together with an imperative program `sumUpToN` using it and a `main` operator for a test run.

```
1  while: {{Bool} -> {Unit} -> Unit}
2  while cond body = if cond! { body!; while cond body }
3                             { unit }
4
5  sumUpToN: {Ref Int -> [RefState]Int}
6  sumUpToN n = let res = new 0 in
7               while { case (read n) { 0 -> false      -- while n != 0 do
```

```
8                                        | _ -> true} }
9                       { write res (read res + read n); -- res := res + arg
10                        write n (read n - 1) };        -- arg := arg - 1
11              read res
12
13  main: {[RefState]Int}
14  main! = sumUpToN (new 4)                            -- returns 10
```

The definition of `while` checks whether the `cond`ition is fulfilled; if so, it executes the `body` and recurses, otherwise it returns `unit`. Notice that `while` does not mention `RefState` at all, only when it is called in *line 7* its ambient is instantiated by `RefState`. In general, using a `while`-loop makes only sense if the ambient is instantiated by some effect that the boolean condition can depend upon. This dependence could be reading from a reference cell as in the example (*line 7*), but could also be through another ability such as `Console`, relying on console input.

Taking the input `n` as a reference to a non-negative integer, `sumUpToN` naively sums up numbers up to n by first allocating a reference cell `res` (*line 6*) initialised by 0, and then adding $n, n-1, ..., 1$ to it. It looks a little verbose and could be written more concise by adding syntactic sugar to Frank.

7.2 Cooperative Processes

Effect handling can be used to make concurrency implementations more flexible and accessible to the programmer [10, 11, 9] by introducing an interface between cooperative signals and how they are interpreted and scheduled. As Dolan et al. argue, this gives more control to the programmer, avoiding a monolithic baked-in mechanism that cannot adapt to particular program needs.

One of the simplest setups is one of cooperative processes as presented by Dolan et al., Hillerström and Lindley et al. [10, 11, 19] where a process can fork new processes and yield to other processes. Building on the example by Lindley et al. and relying on referentiable state, we present an implementation of the actor model as well as an implementation of promises. Remarkable about these concurrency libraries is that they are easily expressed in Frank and offer a convenient interface to programmers.

A cooperative process is marked by the interface `Co`, allowing a process to `fork` new processes (which inherit the parent's abilities) and to `yield` to other processes.

```
interface Co = fork:  {[Co]Unit} -> Unit
             | yield: Unit
```

The way in which `yield` is used depends on the programmer: It can be used for blocking when a process is waiting for a specific event (by yielding until the event occurs) but also to benevolently prevent starvation of other processes. The latter use shows that processes are expected to *cooperate* as there is no preemption (at least not in our Frank setting). This means that the `yield` and `fork` commands determine all possible interleavings, of which the handler/scheduler can pick one. Despite its simplicity, this interface serves well as a base for more sophisticated concurrency models as described in Sections 7.3 and 7.4. Let us first look at a use case of `Co` by considering the following counter.

```
counter: {[Co [Console], Console]Unit}
counter! = print "start␣";
           fork {print "one␣"; yield!; print "two␣"};
           fork {print "eins␣"; yield!; print "zwei␣"};
           yield!;
           print "done?␣"
```

As described, the eventual interleaving is only determined by the handler. Two of the possible outputs would thus be `"start␣one␣two␣eins␣zwei␣done?␣"` or `"start␣one␣eins␣done?␣zwei␣two␣"`.

7.2.1 Round-Robin Scheduling

The simplest standard way to schedule the processes is the Round-Robin strategy: Maintaining a queue of processes, we execute the head of the queue until the next `yield` or `fork`. We handle `yield` by pushing the continuation onto the queue and popping the next process. We handle `fork` by pushing the new process and continuing with the current process.

How can we implement this strategy? We present in the following the implementation as described by Lindley et al. [19][1]. First of all, we define a queue interface which provides an `enq`ueue and `deq`ueue operation.

```
interface Queue S = enq : S -> Unit
                  | deq : Maybe S
```

Furthermore, we define an operator `roundRobin` that takes a `Co` process and translates it recursively into a `Queue` manager, by which we refer to a computation with the ability to access a `Queue`. In particular, `roundRobin` results in a queue manager that executes the translated processes according to the described Round-Robin strategy. Because the enqueued items themselves are queue managers (i.e., may enqueue/dequeue new queue managers), we define a recursive data type `Proc` which wraps a queue manager. We can use the operators `pushProc` to enqueue a new queue manager, `popRunProc` to dequeue the next queue manager and execute it, and `popRunAllProcs` pop and execute all remaining queue managers. The `roundRobin` implements the mechanism describe before on top of the given queue operations.

```
data Proc = proc {[Queue Proc]Unit}

pushProc : {{[Queue Proc]Unit} -> [Queue Proc]Unit}
pushProc p = enq (proc p)

popRunProc : {[Queue Proc]Unit}
popRunProc! = case deq! { (just (proc x)) -> x!
                        | nothing         -> unit }

popRunAllProcs : {[Queue Proc]Unit}
popRunAllProcs! = case deq! { (just (proc x)) -> x!; popRunAllProcs!
                            | nothing         -> unit }

roundRobin : {<Co [Queue Proc]>Unit -> [Queue Proc]Unit}
roundRobin <yield -> k>  = pushProc {roundRobin (k unit)}; popRunProc!
roundRobin <fork p -> k> = pushProc {roundRobin p!}; roundRobin (k unit)
roundRobin unit          = popRunAllProcs!
```

7.2.2 Queue Implementation

Finally, the queue manager gets wrapped by a queue implementation which handles the `enq`ueue and `deq`ueue commands as expected. We use the fast queue implementation of Okasaki [31] that represents the queue by two lists of different orders. We have picked a formulation similar to how the state monad is run in Haskell: The `runZipQ` function takes a queue manager (`<Queue S>X`), an initial state (`ZipQ S`) and outputs the computed value and the remaining queue (`Pair X (ZipQ S)`). The function `evalZipQ` restricts the result to the computed value and `evalZipQEmpty` fixes the input queue as the empty queue.

```
data ZipQ S = zipq (List S) (List S)

runZipQ: {<Queue S>X -> ZipQ S -> Pair X (ZipQ S)}
runZipQ <enq q -> k> (zipq ps      qs) = runZipQ (k unit)
                                         (zipq ps (q :: qs))
runZipQ <deq -> k>   (zipq []      []) = runZipQ (k nothing)
                                         (zipq [] [])
runZipQ <deq -> k>   (zipq []      qs) = runZipQ (k deq!)
```

[1] We have rewritten parts for a more concise presentation, but the core idea is the same

```
                                                    (zipq (rev qs) [])
runZipQ <deq -> k>   (zipq (p::ps) qs) = runZipQ (k (just p))
                                                    (zipq ps qs)
runZipQ x            (zipq ps      qs) = pair x (zipq ps qs)

evalZipQ: {<Queue S>X -> ZipQ S -> X}
evalZipQ <t> q = case (runZipQ t! q) { (pair x _) -> x }

evalZipQEmpty : {<Queue S>X -> X}
evalZipQEmpty <m> = evalZipQ m! (zipq [] [])
evalZipQEmpty x   = x
```

7.2.3 Composing the Handlers

We have now everything we need to handle a program such as `counter` using the Round-Robin strategy:

```
main: {[Console]Unit}
main! = evalZipQEmpty (roundRobin counter!)
-- prints: start one eins done? two zwei
```

Factoring the composition out into a `runCo` function is very desirable to encapsulate the concrete handling cascade.

```
runCo: {<Co [Queue Proc], Queue Proc>Unit -> Unit}
runCo <c> = evalZipQEmpty (roundRobin c!)

main: {[Console]Unit}
main! = runCo counter!
-- prints: start one eins done? two zwei
```

However, when composing `evalZipQEmpty` and `roundRobin`, we encounter the issue of unintentionally exposed intermediate effects (see Chapter 4.2) by disclosing to a user of `runCo` that it internally uses `Queue Proc` effects.

7.3 Actor Model

It turns out that Frank is expressive enough to describe a variant of the actor model. We will first recall the actor model and present the Frank interface given to the programmer along with an example actor. Furthermore, we show how the actor model can be implemented in Frank on top of cooperative processes.

An actor is a computation of type `Unit` with the additional ability to spawn new actors and communicate with other actors. For the latter, each actor is assigned a mailbox through which it can receive messages. When asking for a new message via `recv`, the actor blocks until a new value arrives. Actors are addressed by their mailboxes, i.e., when an actor wants to send a message to another actor via `send`, it needs to know the receiver's mailbox. Spawning a new child actor via `spawn` results in a new mailbox which can then be addressed. In general, a mailbox can be seen as a buffer which makes communication between actors asynchronous. Consider now the `Actor` interface:

```
interface Actor X = spawn Y: {[Actor Y]Unit} -> Mailbox Y
                  | self:    Mailbox X
                  | recv:    X
                  | send Y:  Mailbox Y -> Y -> Unit
```

The parameter `X` represents the mailbox type, i.e., the type of the messages that can be received. One will notice that the type `Mailbox` has not been given yet. We give it in Section 7.3.1 as it has to be given concretely as part of the implementation. This is due to a shortcoming of Frank: What we really want is `Actor` to be polymorphic in the `Mailbox` functor; if we had higher kinds we could have written[2]:

[2]The same generalisation would be possible in the definition of all operators that do not handle actors.

```
interface Actor X M = spawn Y: {[Actor Y]Unit} -> M Y
                    | ...
```

We see higher kinds as a possibility for future work and for now have to settle with a concrete type of `Mailbox` which we give later.

When an actor receives a message, it often wants to reply to the sender later on. An easy way to enable this is to send its own mailbox address along the message such that the receiver obtains in addition the return address. This is where `self` comes into play by enabling self-reference. Because communicating the return address is such a common scenario, we introduce the following definition to enrich a message type with the return address.

```
data WithSender X Y = withSender (Mailbox X) Y
```

Let us now turn towards a very simple example of a concurrent program using actors. It demonstrates how a main actor `divConqActor` spawns two child actors (`doubleActor`), provides both of them with a computational task, awaits both responses, finishes the computation and then prints the result.

```
doubleActor: {[Actor (WithSender Int Int)]Unit}
doubleActor! = case recv! { (withSender sender n) -> send sender (n + n) }

divConqActor: {[Actor Int [Console], Console]Unit}
divConqActor! = let childA = spawn doubleActor in
                let childB = spawn doubleActor in
                send childA (withSender self! 1);
                send childB (withSender self! 2);
                print "calculating␣(1+1)␣+␣(2+2)...␣";
                case (recv! + recv!) { 6 -> print "6"
                                     | _ -> print "not␣6" }
```

One might be wondering why we would assign `divConqActor` the type:

```
{[Actor Int [Console], Console]Unit}
```

First of all, `Actor` has an implicit type variable ϵ which it passes on to any spawned actor computation. Why do we parameterise the `Actor` ability with `Console` in the type of `divConqActor`, despite the only spawned actor `doubleActor` not accessing the console? Indeed, admitting the `Console` ability to spawned processes within `divConqActor` seems unnecessary. The reason which makes this necessary though is that handling a whole actor system (parent actor with children, grand-children, ...) requires it to be uniform in its ability because it is handled in a uniform way, too. By providing the `Console` ability to the parent actor, the handler cannot strip `Console` off its ambient for handling the children: Ambients are accumulative and do not allow stripping off abilities.

7.3.1 Implementation Using Cooperative Processes

We now advance to an implementation of the actor model and base it on cooperative processes as presented in Section 7.2. The essential idea is that each actor will be translated into a process (i.e., `spawn`ing becomes `fork`ing) and each mailbox corresponds to a reference to a queue of messages. Let us begin with the postponed definition of the latter which the already implemented zip queue is based on:

```
data Mailbox X = mbox (Ref (ZipQ X))
```

An actor is made up of a mailbox component and a computational component. Combined as a whole, they can be translated into a cooperative process. Notice that handling allows us to translate the actor computation step by step, thereby yielding the overall translation.

```
1 step: {Mailbox X -> <Actor X [RefState, Co[RefState]]>Unit ->
2         [RefState, Co [RefState]]Unit}
3 step mine          <self -> k> = step mine (k mine)
4 step mine          <spawn you -> k> =
5                       let yours = mbox (new (zipq [] [])) in
```

```
6                       fork {step yours you!};
7                       step mine (k yours)
8  step (mbox mine') <recv -> k> = case (runZipQ deq! (read mine'))
9                                  { (pair nothing _)  ->
10                                      yield!;
11                                      step (mbox mine') (k recv!)
12                                 | (pair (just x) q) ->
13                                      write mine' q;
14                                      step (mbox mine') (k x) }
15 step mine          <send (mbox yours') x -> k> =
16                      let q = read yours' in
17                      case (execZipQ (enq x) q)
18                      { q' -> write yours' q';
19                              step mine (k unit) }
20 step mine          unit = unit
```

We denote the current mailbox of an actor as `mine` and the mailbox of a spawned child as `yours`. Let us briefly go through `step`. In *line 3*, the request for the own mailbox is straightforwardly handled. In *lines 4 to 7*, we spawn a new actor by first allocating a new mailbox for it, then forking its translation. We continue with the execution of the current actor. In *lines 8 to 14*, we handle a `recv` command. There are two cases: 1) The mailbox is empty. Blocking the actor is translated by a call to `yield` and a subsequent retry. 2) The mailbox is non-empty. We take out the first message and continue. In *lines 15 to 19* we handle `send` by enqueueing the new message into the addressed mailbox.

7.3.2 Composing the Handlers

We can now compose the operators in a similar way to how operators for the execution of cooperative processes were composed. Again, running a specific actor in a direct manner is straightforward as effects are implicitly instantiated.

```
main: {[Console, RefState]Unit}
main! = evalZipQEmpty (roundRobin (step (mbox (new (zipq [] []))))
                                          divConqActor!))
-- prints: calculating (1+1) + (2+2)... 6
```

The same problem as experienced in Section 7.2 occurs though when factoring the handler composition out into a `runActor` operator, and this time the type is severely larger, leaking the use of several intermediate effects.

```
runActor : {<RefState,
             Queue (Proc [RefState]),
             Co [Queue (Proc [RefState]), RefState],
             Actor X [RefState,
                      Queue (Proc [RefState]),
                      Co [Queue (Proc [RefState]), RefState]]>Unit ->
            [RefState]Unit}
runActor <m>  = evalZipQEmpty (roundRobin (step (mbox (new (zipq [] [])))
                                                  m!))
runActor unit = unit

main: {[Console, RefState]Unit}
main! = runActor divConqActor!
-- prints: calculating (1+1) + (2+2)... 6
```

Again, we see that this is a problem still to be solved.

7.4 Promises

We present how a second model of concurrency can be implemented in Frank, namely promises in the style of the *async* library for Haskell [27]. Recently, Dolan et al. implemented promises

in the same style in Multicore OCaml [9]. After recalling how concurrency based on promises works, we point out how this mechanism can take shape in Frank.

Promises enable a user to issue computational tasks, carry on with computation, and request the tasks' results when needed. Concurrency is thereby harnessed in that several tasks can be computed concurrently. On issuing a task, the issuing program receives a *promise*. When the result of the promise is demanded (by any program having a hold of the promise), the program blocks until the promise yields a result. Consider now the `Async` interface:

```
data PromiseStatus X = done X
                     | waiting (List {X -> Unit})

data Promise X = promise (Ref (PromiseStatus X))

interface Async = async X: {[Async]X} -> Promise X
                | await X: Promise X -> X
```

An operator implementing `Async` can issue a task via `async`, thereby receiving a `Promise`. At any later point, the promise can be waited for using the `await` command. A `Promise` is a reference to a `PromiseStatus`, marking the task either as `done` and providing a value or containing a list of operators that are `waiting` for the promise's result.

Let us now look at a simple example making use of promises. The binary operator `binOp` represents a resource-intensive task and is for now just modelled as the `+` operator. The operator `calcProg` takes three integers and exhibits `Async` effects by performing two independent resource-intensive tasks on the input. By spawning the tasks, two promises `p1` and `p2` are returned (*lines 5, 6*). Finally, the results of both `p1` and `p2` are requested (*lines 7, 8*) for and their sum is returned (*line 9*).

```
1 binOp : {Int -> Int -> Int}
2 binOp m n = m + n
3
4 calcProg : {Int -> Int -> Int -> [Async]Int}
5 calcProg a b c = let p1 = async {binOp a b} in
6                  let p2 = async {binOp b c} in
7                  let v1 = await p1 in
8                  let v2 = await p2 in
9                  v1 + v2
```

7.4.1 Implementation Using Cooperative Processes

How can we handle `Async` operators? Again, we resort to cooperative processes as presented in Chapter 7.2. The essential idea is that a `Promise` and an `Async` thunk (i.e., a task which results in a value) can together be translated recursively into a cooperative process that — when executed — computes the task and stores the result in the promise. Spawning from within the task via `async` is translated accordingly to `fork`ing. The translation is performed via a `step` operator which works similarly to its counterpart for actor translation. We first explain the different cases of `step` and afterwards give the auxiliary operator `execPromise` with explanations.

```
1  step: {Promise X [RefState, Co[RefState]] ->
2         <Async[RefState, Co[RefState]]>X ->
3         [RefState, Co[RefState]]Unit}
4  step (promise ref) v = write ref (done v)
5  step p <async c -> k> = let ref' = new (waiting []) in
6                          let job = {step (promise ref') c!} in
7                          fork { execWakeUp (promise ref') job };
8                          step p (k (promise ref'))
9  step p <await (promise ref') -> k> = case (read ref')
10                                      { (done v) -> step p (k v)
11                                      | (waiting js) ->
12                                          let job = { v -> step p (k v) } in
13                                          write ref' (waiting (job::js)) }
```

The first case describes the situation when the task has resulted in a value. This means that the promise has just been fulfilled, so the value is written to the promise's reference cell (*line 4*).

The second case handles `async` requests from the task. We create a blank non-fulfilled promise (*line 5*) and then a thunk which — as soon as executed — recursively executes the newly issued task (*line 6*). We call this thunk a job. The reason for not directly forking this job is that besides just executing the job, we need to retrieve the suspended computations in the waiting list in order to "wake them up" after executing the job. For this reason, we wrap the job by *execWakeUp* (*line 7*). Finally, after the new task is issued, we return the newly created promise to the task that issued the `async` command request (*line 8*).

The third case handles `await` requests from the task, given a promise whose fulfilment is to be awaited. There are two possibilities. First, if the promise has been fulfilled already, the stored value is passed on to the continuation (*line 10*). Second, if the promise has not been fulfilled yet, we enqueue the current computation (the continuation) as a further job into the promise's waiting list (*lines 12, 13*).

We now come to the `execWakeUp` wrapper operator that executes a job and delegates the value to the waiting list's computations.

```
1  execWakeUp: {Promise X [RefState, Co[RefState]] ->
2               {[RefState, Co[RefState]]Unit} ->
3               [RefState, Co[RefState]]Unit}
4  execWakeUp (promise ref) f = case (read ref)
5                               { (done _) -> unit   -- cannot happen
6                               | (waiting fs) ->
7                                   f!;
8                                   case (read ref)
9                                   { (done v) -> map { f -> f v } fs;
10                                                unit
11                                  | _ -> unit } }  -- cannot happen
```

The operator `execWakeUp` is applied to each task exactly once. Since the promise is blank in the beginning, we thus know that when examining the promise (*line 4*), the first case cannot happen. However, this semantic invariant is not statically available and we thus handle it by doing nothing (return `unit`, *line 5*). We come back to this point later.

When the promise is indeed found to contain a list of suspended tasks, we first execute the job which will fulfil the promise (*line 7*). This means that when looking up the promise again (*line 8*), we are guaranteed that it is fulfilled. However, again this semantic invariant is not statically available and we thus handle the second case again by doing nothing (return `unit`, *line 11*). When the promise is indeed found to contain a result value, it is fed to every suspended task.

Let us take up again the issue of missing static guarantees (*lines 5, 11*). These are due to the use of a single global referentiable state. By having a mechanism similar to the *ST monad* [16] in Haskell at hand, local state can be used to encapsulate the different tasks and obtain actual values of their executions. This requires the capability of "running" stateful computation, which is not possible with `RefState` as it is run outside of pure Frank.

7.4.2 Composing the Handlers

We can now compose the operators in a similar way to how operators for the execution of cooperative processes and actors were composed. We directly give an operator `runAsync` that factors the comprises the composition. As is clearly visible, the operator suffers severely from the problem of unintentionally exposed intermediate effects. It takes a task, creates a new promise (*line 10*), performs the task on the promise (*line 11*) and returns it (*line 12*).

```
1  runAsync : {<Co [Queue (Proc [RefState]), RefState],
2               Queue (Proc [RefState]),
3               Async[RefState,
4                     Co[Queue (Proc [RefState]), RefState],
5                     Queue (Proc [RefState])]>X ->
```

```
6               [RefState] (Promise X [RefState,
7                                      Co[RefState,
8                                         Queue (Proc [RefState])],
9                                      Queue (Proc [RefState])])}
10 runAsync <m>  = let resRef = new (waiting []) in
11                 evalZipQueueEmpty (roundRobin (step (promise resRef) m!));
12                 promise resRef
```

Finally, we can run the `calcProg` operator using `runAsync`. Again, we do not get a static guarantee that the promise returned by `runAsync` is indeed fulfilled, which is again due to not "executing" the referentiable state within Frank.

```
main: {[RefState]Int}
main! = case (runAsync (calcProg 1 2 3))
        { (promise resRef) -> case (read resRef)
                              { (done v) -> v
                              | _ -> 0 }      -- cannot happen
        }
-- returns: 8
```

As before, we have to check the promise's result and provide a dummy value for the case which will never happen. Executing the program results in `8` as expected.

Chapter 8

Further Technical Enhancements

While focusing on our main goals as laid out so far we have come across some rather technical issues and ideas of which we discuss the major ones here. As one of our first steps with the Frank compiler, we extended Frank by interface aliases. Furthermore, we eliminated a bug by providing an algorithm that identifies implicit effect variables of top-level definitions. Finally, in order to improve error reporting by displaying meta-information such as source code locations, we refactored the representation of the abstract syntax tree and discuss the taken approach.

8.1 Interface Aliases

An ability is made up of an (often implicit) effect variable and a set of instantiated interfaces. We introduce interface aliases as a way to comprise several interface instantiations together. Consider the following example:

```
interface Writer X = send: X -> Unit
interface Reader X = receive: X

interface State X = [Writer X, Reader X]
```

Formally, we update the abstract syntax of top-level definitions as follows.

$$
\begin{array}{rcl}
\mathit{Definition} & ::= & \mathit{DatatypeDef} \mid \mathit{InterfaceDef} \mid \mathit{InterfaceAliasDef} \mid \\
 & & \mathit{TermDef} \mid \mathit{OperatorDef} \\
\mathit{DatatypeDef} & ::= & \mathbf{data}\ D\ \overline{Z}\ =\ \overline{\mathit{Constructor}} \\
\mathit{Constructor} & ::= & k\ \overline{A} \\
\mathit{InterfaceDef} & ::= & \mathbf{interface}\ I\ \overline{Z}\ =\ \overline{\mathit{Command}} \\
\mathit{InterfaceAliasDef} & ::= & \mathbf{interface}\ I\ \overline{Z}\ =\ [\overline{I\ \overline{R}}] \\
\mathit{Command} & ::= & c\ :\ \forall \overline{Z}.\overline{A \to} B \\
\mathit{TermDef} & ::= & f\ :\ P \\
 & & f = n \\
\mathit{OperatorDef} & ::= & f\ :\ \forall \overline{Z}.\{C\} \\
 & & \overline{f\ \overline{r}\ =\ n}
\end{array}
$$

Interface aliases do not add any semantics but serve only as a mean of abbreviation. Therefore no changes to the type system or the dynamic semantics are required.

8.1.1 Implementation

Interface aliases are resolved before the type-checking phase in that they are substituted recursively by their definitions. The only critical issue to encounter here is that of detecting cyclic

definitions. This issue is easily eliminated though by keeping track of the definitions substituted so far in each recursion path. Whenever a definition is encountered twice, a cycle is detected and an error message is output.

8.2 Implicit Type Variables

Operators, constructors and interfaces are often polymorphic in their effects by providing an effect type variable. This is where implicit type variables can make types considerably more concise and readable. Consider for instance the definition of `MaybeThunk`...

```
data MaybeThunk X = thunk {X} | value X
```

...which is semantically equivalent to and internally pre-processed to...

```
data MaybeThunk X [ϵ] = thunk {[ϵ|]X} | value X
```

That is, the ambient ability bound by $[\epsilon]$ is pushed inwards and enabled for the thunk, too. An example of a definition without implicit effect variable is

```
data Bool = false | true
```

How is determined whether top-level definitions like `MaybeThunk` or `Bool` possess the implicit effect parameter $[\epsilon]$? Determining this is not as straightforward as one might think, because dependencies between definitions need to be considered. As these dependencies were not taken care of in an earlier version of the Frank compiler, we have fixed this bug and briefly sketch in the following how determining implicit effect variables can take place.

We have seen two examples for which we could decide whether an implicit $[\epsilon]$ parameter is present or not, merely by considering the two definitions by themselves. The data type `MaybeThunk` has an implicit $[\epsilon]$ parameter because the `thunk` constructor contains a thunk whose ability is implicitly bound to $[\epsilon]$. The data type `Bool` has no implicit $[\epsilon]$ parameter because none of its components implicitly are bound to one. Consider now the following definition.

```
interface Force = force X:   MaybeThunk X -> X
                | forceIf X: Bool -> MaybeThunk X ->MaybeThunk X
```

Whether `Force` has an implicit $[\epsilon]$ parameter or not depends on its components. The two contained components to be examined are `Bool` and `MaybeThunk`. Having already decided that `MaybeThunk` has an implicit $[\epsilon]$ and `Bool` has not, we decide that `Force` has an implicit $[\epsilon]$. It suffices that one dependency is decided positive (here `MaybeThunk`). The pre-processed version of the definition is thus:

```
interface Force [ϵ] = force X:   MaybeThunk X [ϵ|] -> X
                    | forceIf X: Bool ->
                                 MaybeThunk X [ϵ|] -> MaybeThunk X [ϵ|]
```

8.2.1 Examining a Definition

For brevity, we call a definition **positive** if it has an implicit $[\epsilon]$ and we call it **negative** otherwise. Given the definition of a data type, an interface or an interface alias, how can we either (1) directly decide whether it is positive/negative or (2) determine any definitions on which its decision depends? We need to examine the whole definition by recursively descending down its structure. The result of a definition's examination is correspondingly either:

(1) A is positive.

(2) Only if any of $\{B_1, ..., B_n\}$ is positive, then A is positive.

Note that (2) includes the negative decision ($n = 0$). Descending the structure of a definition works quite similar for the different components, so we only demonstrate how this is carried out by an example and refer to the source code [24] for more detail.

The examination of a data type definition requires examining its constructors. If any constructor is positive (1), so is the data type (1). Otherwise, the constructors' dependencies (2) are merged to become the data type's dependencies (2).

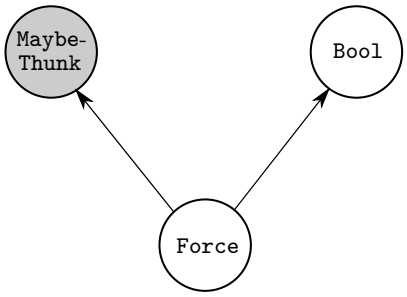
Having examined all definitions, we either have a definite decision or a set of dependencies for each definition. We now show how this information can be transformed into a final positive/negative decision for every definition.

8.2.2 Reduction to Graph Labeling Problem

Resolving the dependencies reduces to solving the following problem of labeling a directed graph: The **nodes** are the top-level definitions. An **edge from node A to node B** represents the dependency "If B is positive, then A is positive". Given an initial set Pos_{init} of positive nodes, the **problem** is to find the *smallest* set of positive nodes (subsuming Pos_{init}) such that the edges are respected. The solution is unique, because for each node A we can distinguish between two cases:

1. There is a path $A \to^* B$ for a node B s.t. $B \in Pos_{init}$ ("A can reach Pos_{init}"). Then A must be positive by transitivity of implication.
2. Otherwise ("A cannot reach Pos_{init}"). Then A may be either positive or negative without disrespecting the edges; but since we are constrained to finding the smallest set of positive nodes, it must be negative.

As an example, consider how the problem of determining the presence of $[\epsilon]$ for `MaybeThunk`, `Bool` and `Force` is reduced to the following graph problem (Figure 8.1).



`MaybeThunk` belongs to Pos_{init} (represented by a filled node) and has no dependencies. `Bool` has no dependencies either. `Force` has two dependencies. When labelling the graph, we observe that `MaybeThunk` is already decided positive, `Bool` does not reach any positive node and is thus decided negative, `Force` reaches the positive node `MaybeThunk` and is thus decided positive.

Figure 8.1: Graph for `MaybeThunk`, `Bool` and `Force`

8.2.3 Solving the Graph Labeling Problem

The algorithm consists of a simple graph traversal with the runtime linear in the number of nodes. We briefly sketch it in the following. Let there be a graph instance as described in the last section, i.e., a set of nodes, a set of edges and a set of pre-decidedly positive nodes Pos_{init}. As long as there is an undecided node A left, we decide it by $decideSubGraph(A, \emptyset)$.

decideSubGraph$(A, visited)$:
Case distinction.
1. A has no outgoing edges to any B with $B \notin visited$. Decide A negative.
2. A has outgoing edges to $B_1, ..., B_n \notin visited$.
   For each B_i, if not decided yet, do so by $decideSubGraph(B_i, visited \cup \{A\})$.

   Case distinction.
   (a) Some B_i is decided positive. Decide A positive.
   (b) All B_i are decided negative. Decide A negative.

The algorithm $decideSubGraph(A, visited)$ explores the subgraph consisting of all nodes and edges reachable from A and decides its contained nodes.

8.3 Improving Error Reporting

Informative error reporting with references to the source code (e.g. line and column numbers) are indispensable for the programmer. It requires the annotation with meta-information (e.g. line and column numbers) of each abstract syntax tree (AST) element that can be referred to in an error message. This annotation was not available in the original Frank compiler [21] and we refactored the code, thereby enabling informative error messages. We mention that we have not focused yet on the error messages themselves and not even on what meta-information is best provided. Instead we briefly describe how the abstract syntax tree is modelled in the current Frank compiler [23] and how annotations are integrated. We first collect the desiderata we have for such an AST model and then discuss the implementation in Haskell.

- **Annotations.** Meta-information like source code locations should be attachable to the AST nodes. Preferably, the meta information is flexible such that it can be changed, added and stripped off during different stages. For example, during the type-checking phase meta-information may additionally consist of unification history that can improve error reporting.
- **Adaptiveness to different processing stages.** In the current implementation, first the AST comes in raw form from the parser, then the AST is refined (by checking variable occurrences and more) and finally desugared. Although the structure remains essentially the same, there are some structural restrictions (e.g. some nodes may only exist after desugaring while others must not exist any more).
- **Structural consistency.** When a parent node has passed a certain processing stage (and therefore is labelled in a certain way), then the same should hold for its child nodes.
- **Easy accessibility.** When processing the AST nodes, easy decomposition (via pattern-matching) and easy composition (like with plain constructors) should be possible.

The original AST model [22] already distinguishes between different processing stages, using generalised abstract data types (GADTs). The data types are labelled by types `Raw`, `Refined` and `Desugared`. For an example, let us take a look at the definition of the `Pattern` data type that is used for modelling pattern-matching.

```
data Pattern a where
  MkVPat :: ValuePat a -> Pattern a                    -- value pattern
  MkCmdPat :: Id -> [ValuePat a] -> Id -> Pattern a -- command pattern
  MkThkPat :: Id -> Pattern a                          -- catch-all pattern
```

The type parameter `a` represents the label, and during the refinement stage for example, nodes of type `Pattern Raw` are refined into nodes of type `Pattern Refined`. Note that all AST types provide this labelling and that labels are handed down (in the example, `Pattern a` hands it label `a` down to `ValuePat a`) such that AST instances are always homogeneously labelled.

8.3.1 Implementation Approach 1: Using GADTs

In this original implementation and by virtue of GADTs, labelling as a type parameter can only distinguish between different processing stages. We have observed already that during different stages, different meta-information may need to be attached to the nodes. One way to solve this is to let the label types `Raw`, `Refined` and `Desugared` contain information-bearing values and attach them to each constructor as an additional parameter.

```
data Pattern a where
  MkVPat :: ValuePat a -> a -> Pattern a
  MkCmdPat :: Id -> [ValuePat a] -> Id -> a -> Pattern a
  MkThkPat :: Id -> a -> Pattern a
```

The `Raw` label for example could then contain information like a source location.

```
data Raw = Raw (Int, Int) -- line #, column #
```

Taking this approach seems to be straightforward and concise (by making only a slight extension to the original definitions) and effective (all of our desiderata are essentially met). The only disadvantage we see is that each constructor always contains a label, even if we might not need it. There is a way to structurally separate the AST node definition from labelling using a second approach we consider.

8.3.2 Implementation Approach 2: Using Fixed-Points of Functors

Haskell allows type constructors to take not only type arguments (i.e., of kind `*`), but general type constructors (e.g. of kind `* -> *`). This allows the definition of a fixed-point type constructor which takes a functor `f` (of kind `* -> *`) as argument and returns a type.

```
data Fix f = Fx (f (Fix f))
```

This fixed-point type constructor allows us to define data types in terms of functors. For demonstration purpose, we give the example of defining lists in terms of `Fix`.

```
data ListF a r where
  Nil :: ListF a r
  Cons :: a -> r -> ListF a r

type List a = Fix (ListF a)

oneTwoList :: List Int
oneTwoList = Fx (Cons 1 (Fx (Cons 2 (Fx Nil)))) -- represents [1, 2]
```

The `ListF` definition is a parameterised functor, with its parameter `a` representing the type of the list elements. The `r` parameter is the "recursor", representing the same type we are about to construct. Consider how `List a` is defined as `Fix (ListF a)`, i.e., as the fixed-point of `ListF a`. By definition of `Fix` then, `List a` is defined in terms of itself: `r` is instantiated exactly by `Fix (ListF a)` = `List a`.

The following implementation is inspired by Williams' Recursion Schemes talk [39]. In particular, he describes how one can compose data types based on functor definitions, giving the example of annotating data types which we rely on. The central idea is that we can take the fixed point of a functor that has been enriched by annotations. More concretely, we define an annotation transformer that takes an arbitrary functor and yields the same functor, but wrapping each value with an annotation.

```
data AnnotT a f r = AnnF (f r, a)
```

Type parameter `a` represents the annotation type (e.g. source code location), `f` represents the functor that is transformed and `r` is again the "recursor". Notice that the parameterisation results in `AnnotT` having the kind `(* -> *) -> (* -> *)`. The constructor `AnnF` serves as a wrapper for a pair consisting of the annotated object itself (of type `f r`) and the annotation (of type `a`).

The idea is now to define the AST data types in a similar way to how we defined `List`: In terms of a parameterised functor (`ListF` in the case of `List`). For demonstration purpose, we focus again on the example of the `Pattern` data type.

```
data PatternF :: ((* -> *) -> (* -> *)) -> * -> * where
  MkVPat :: TFix t ValuePatF -> PatternF t r
  MkCmdPat :: Id -> [TFix t ValuePatF] -> Id -> PatternF t r
  MkThkPat :: Id -> PatternF t r

data ValuePatF :: ((* -> *) -> (* -> *)) -> * -> * where
  ...
```

As the signature shows, `PatternF` can be understood as a functor parameterised by a functor transformer (which is of kind `(* -> *) -> (* -> *)`). All other AST node types (e.g. `ValuePatF`) are parameterised in the same way. We have not defined `TFix` yet which is used to construct the type `TFix t ValuePatF`. This is the point where structural consistency is enforced: The same given parameter `t` is handed down to the children nodes.

```
type TFix (t :: (* -> *) -> (* -> *))
          (f :: ((* -> *) -> (* -> *)) -> * -> *) = Fix (t (f t))
```

It is easy to get confused by the large signature, but it is really just a type synonym for `Fix (t (f t))`. In our example, `ValuePatF` is first parameterised with the same `t` as `PatternF`, resulting in a functor `ValuePatF t`. This functor is then tranformed by `t` (e.g. enriched by annotations), resulting in a functor `t (ValuePatF t)`. Finally, we take the fixed-point of this functor, resulting in `Fix (t (ValuePatF t))`.

We provide a specialised version of `TFix`, fixing the transformer as `AnnotT a`, but leaving the `a` as a parameter.

```
type AnnotTFix a f = TFix (AnnotT a) f
```

We can now define `Pattern` as a parameterised AST node type that we will primarily use, a fixed-point enriched by an annotation whose type is given as a parameter.

```
type Pattern a = AnnotTFix a PatternF
```

Notice that the kind of `Pattern` is `* -> *`, corresponding exactly to the definition given in approach 1). The difference is that because of its definition in terms of `Fix` and `AnnotT`, its values are "polluted" by `Fx` and `AnnF` constructors. This is no problem though as the "pollution" occurs very regularly. We can use *PatternSynonyms* [38] which allow the synonyms in the following manner.

```
pattern VPat vp a = Fx (AnnF (MkVPat vp, a))
```

The `Fx`, `AnnF` and `MkVPat` constructors are now hidden behind a single comprising `VPat` constructor. This feature ensures our desired property of having easy access to AST nodes.

8.3.3 Discussion

Both approaches satisfy our desiderata.

Approach 1) has the advantage of being easy to implement and causing only little overhead. On the other hand, every AST definition is bound to have exactly one annotation. For the current Frank compiler, this is sufficient, but potential future extensions like a second annotation layer are ruled out.

Approach 2) causes significantly more overhead. Not only need constructs like `Fix` and `AnnotT` be additionally defined, but there is also the need for guiding type class instantiation. In order to let the AST node types instantiate the type classes `Show` (to obtain string representations) and `Eq` (to check equality), we need to manually provide some hints. The hints are trivial though and besides the additional overhead this cannot be seen as a serious disadvantage. As described, the problem of "constructor pollution" is overcome by *pattern synonyms* such that together with few auxiliary functions, nothing of the internal fixed-point construction is noticed from the outside.

The current version of Frank [23] employs the second approach, we stress however that the source code that *uses* the syntax definitions remained almost the same if one had implemented the first approach.

Chapter 9

Conclusion

We conclude by a short final discussion of the targeted problems, the achieved enhancements and potential future work.

The enhancements we provide to Frank are built on top of each other. First, we establish polymorphic commands as a natural extension that has proven to be useful before [14]. Then we provide referentiable state by introducing a built-in interface relying on polymorphic commands for ML-style reference cells. Having referentiable state and polymorphic commands at hand, we are able to provide two concurrency models within Frank, namely actors and promises. Future work could include a more general exploration of generative effects, e.g. to establish local referentiable state in the style of the ST monad [16] in Haskell.

During the implementation of the concurrency models within Frank we came across a problem with operator composition, more specifically the problem of unintentionally exposed intermediate effects. We solve only a part of the problem and discuss the potential future approach of effect-hiding to tackle the more general problem. A first remedy could be effect-hiding composition operators, but we believe that this could be further generalised to effect-hiding multi-tunnels that gate effects from multiple output pegs (effect generators) to a single input port (effect handler).

All enhancements have been integrated into the Frank formalism and also implemented in the compiler [23]. By providing a direct operational small-step semantics, we 1) verify that the enhancements preserve type soundness and 2) establish a different perspective on the execution of Frank programs, forming a contrast to the perspective on the translation to Core Frank which is based on unary handlers.

We have encountered Frank as a junior and have been in charge of him for several weeks. On this joyful journey, we believe that besides having discovered some more of his potential, he has also gained in robustness. We wish him the best as he will further take up his journey to become a self-reliant member among his peers.

Bibliography

[1] Andrej Bauer and Matija Pretnar. "An effect system for algebraic effects and handlers". In: *International Conference on Algebra and Coalgebra in Computer Science*. Springer, 2013, pp. 1–16.

[2] Andrej Bauer and Matija Pretnar. "Programming with algebraic effects and handlers". In: *Journal of Logical and Algebraic Methods in Programming* 84.1 (2015), pp. 108–123.

[3] Edwin Brady. "Programming and reasoning with algebraic effects and dependent types". In: *ACM SIGPLAN Notices*. Vol. 48. ACM, 2013, pp. 133–144.

[4] Edwin Brady. "Resource-dependent algebraic effects". In: *International Symposium on Trends in Functional Programming*. Springer, 2014, pp. 18–33.

[5] Lukas Convent. *compos.kk (handler composition in Koka)*. 2016/17. URL: `https://github.com/frank-lang/frank/blob/cf33e60/examples/lukas-thesis/compos.kk`.

[6] Lukas Convent. *compos.links (handler composition in Links)*. 2016/17. URL: `https://github.com/frank-lang/frank/blob/cf33e60/examples/lukas-thesis/compos.links`.

[7] Lukas Convent. *Informatics Research Review: Enhancing a Modular Effectful Programming Language*. 2017.

[8] Stephen Dolan, Leo White, and Anil Madhavapeddy. "Multicore OCaml". In: *OCaml Users and Developers Workshop*. 2014.

[9] Stephen Dolan et al. "Concurrent System Programming with Effect Handlers". In: Proceedings of the Symposium on Trends in Functional Programming (2017).

[10] Stephen Dolan et al. "Effective concurrency through algebraic effects". In: *OCaml Workshop*. 2015.

[11] Daniel Hillerström. *MSc Thesis: Compilation of Effect Handlers and their Applications in Concurrency*. 2016.

[12] Daniel Hillerström. *MSc Thesis: Handlers for Algebraic Effects in Links*. 2015.

[13] Daniel Hillerström and Sam Lindley. "Liberating effects with rows and handlers". In: *Proceedings of the 1st International Workshop on Type-Driven Development*. ACM, 2016, pp. 15–27.

[14] Ohad Kammar, Sam Lindley, and Nicolas Oury. "Handlers in action". In: *ACM SIGPLAN Notices*. Vol. 48. ACM, 2013, pp. 145–158.

[15] Oleg Kiselyov, Amr Sabry, and Cameron Swords. "Extensible effects: an alternative to monad transformers". In: *ACM SIGPLAN Notices*. Vol. 48. ACM, 2013, pp. 59–70.

[16] John Launchbury and Simon L. Peyton Jones. "Lazy functional state threads". In: *ACM SIGPLAN Notices*. Vol. 29. ACM, 1994, pp. 24–35.

[17] Daan Leijen. "Koka: Programming with row polymorphic effect types". In: *arXiv preprint arXiv:1406.2061* (2014).

[18] Daan Leijen. "Type directed compilation of row-typed algebraic effects". In: *POPL. ACM*. 2017.

[19] Sam Lindley. *Implementation of cooperative processes in Frank*. URL: `https://github.com/frank-lang/frank/blob/4b7cb52/examples/coop.fk`.

[20] Sam Lindley, Conor McBride, and Craig McLaughlin. "Do be do be do". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM. 2017, pp. 500–514.

[21] Sam Lindley, Conor McBride, and Craig McLaughlin. *Frank junior (0.1)*. 2016/17. URL: https://github.com/frank-lang/frank/tree/4b7cb52.

[22] Sam Lindley and Craig McLaughlin. *Syntax.hs in Frank*. URL: https://github.com/frank-lang/frank/blob/4b7cb52/Syntax.hs.

[23] Sam Lindley et al. *Frank compiler (0.2)*. 2016/17. URL: https://github.com/frank-lang/frank/tree/cf33e60.

[24] Sam Lindley et al. *RefineSyntaxConcretiseEps.hs, part of Frank compiler (0.2)*. 2016/17. URL: https://github.com/frank-lang/frank/blob/cf33e60/RefineSyntaxConcretiseEps.hs.

[25] Sam Lindley et al. *Regression for unification order problem, part of Frank compiler (0.2)*. 2016/17. URL: https://github.com/frank-lang/frank/blob/cf33e60/tests/should-fail/r.unificationOrderMatters1.fk.

[26] John M. Lucassen and David K. Gifford. "Polymorphic effect systems". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1988, pp. 47–57.

[27] Simon Marlow. *The async package (2.1.1.1)*. 2012. URL: https://hackage.haskell.org/package/async.

[28] Conor McBride. *Frank (0.3)*. 2012. URL: http://hackage.haskell.org/package/Frank.

[29] Conor McBride. *How might effectful programs look like? (Workshop on Effects and Type Theory)*. 2007. URL: http://cs.ioc.ee/efftt/mcbride-slides.pdf.

[30] Conor McBride. *The Frank Manual*. 2012. URL: https://personal.cis.strath.ac.uk/conor.mcbride/pub/Frank/TFM.pdf.

[31] Chris Okasaki. *Purely functional data structures*. Carnegie Mellon University, 1996.

[32] Benjamin C. Pierce and David N. Turner. "Local type inference". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22.1 (2000), pp. 1–44.

[33] Gordon D. Plotkin and Matija Pretnar. "Handling Algebraic Effects". In: *Logical Methods in Computer Science* 9 (2013).

[34] Gordon Plotkin and John Power. "Adequacy for algebraic effects". In: *International Conference on Foundations of Software Science and Computation Structures*. Springer, 2001, pp. 1–24.

[35] Gordon Plotkin and John Power. "Algebraic operations and generic effects". In: *Applied Categorical Structures* 11.1 (2003), pp. 69–94.

[36] Gordon Plotkin and John Power. "Notions of computation determine monads". In: *International Conference on Foundations of Software Science and Computation Structures*. Springer, 2002, pp. 342–356.

[37] Gordon Plotkin and John Power. "Semantics for algebraic operations". In: *Electronic Notes in Theoretical Computer Science* 45 (2001), pp. 332–345.

[38] GHC Team. *GHC Extension: Pattern Synonyms*. 2011. URL: https://ghc.haskell.org/trac/ghc/wiki/PatternSynonyms.

[39] Tim Williams. *Recursion Schemes by Example*. 2013. URL: https://github.com/willtim/recursion-schemes/raw/master/slides-final.pdf.