Tampere University

Pauli Jaakkola

A TYPE SYSTEM FOR FIRST-CLASS RECURSIVE ML MODULES

Faculty of Information Technology and Communication Services
Master of Science Thesis
November 2020

ABSTRACT

Pauli Jaakkola: A Type System for First-Class Recursive ML Modules
Master of Science Thesis
Tampere University
Computing and Electrical Engineering
November 2020

Standard ML (SML), OCaml and other programming languages in the ML language family offer a particularly expressive module system. This *ML module system* provides hierarchical namespacing with *structures*, fine-grained interfaces with *translucent signatures*, implementation-side data abstraction with *sealing* as well as client side data-abstraction and generic programming with *functors*.

Languages with the ML module system are *stratified* into two levels: core ML and modules. There is extensive duplication of functionality between these levels: records and structures, types and signatures, functions and functors. The second-class nature of modules also limits expressivity as some features are available only in core ML or only in modules.

ML languages typically limit recursive definitions to very specific patterns: groups of mutually recursive functions or type definitions connected with `and` keywords. The restriction on type definitions ensures that *equirecursive types* are not needed. The restriction on function definitions ensures that recursive values are *well-founded*, preventing runtime use of uninitialized values.

Recursive modules are not part of the Definition of Standard ML but are supported by language extensions in several SML compilers as well as OCaml. Just like mutually recursive functions and types, mutually recursive modules are invariably required to be syntactically grouped together. Implementations of recursive ML modules also struggle with the *double vision* and *recursive linking* problems. In double vision a recursive module implementing abstract types fails to equate its exported abstract types with their implementations that should not be hidden inside the module. The recursive linking problem is to find an elegant and statically well-founded method of recursively linking modules not defined in the same group of recursive definitions or maybe not even the same compilation unit.

The paper "1ML – Core and Modules United" successfully collapsed core ML and modules into *one language* with *first-class modules*. Recursive modules have been deployed in production compilers and several methods to avoid double vision are known from the literature. However first-class and recursive modules have not been combined despite the availability of intermediate languages which can both act as targets of elaborating type systems like 1ML and support recursive modules.

This thesis develops *Recursive 1ML (R1ML)*, a type system for recursive first-class modules. R1ML is an elaborating type system in the style of 1ML. R1ML targets a variant of System F_c, which was created as an intermediate language for the Glasgow Haskell Compiler (GHC). Haskell does not have ML modules but the parametric polymorphism and type equality coercions of F_c are general enough to support recursive first-class modules while avoiding the double vision problem. This thesis also proves that R1ML is sound and decidable but incomplete in several (tolerable) ways. There was no time nor space to consider the recursive linking problem.

Keywords: programming languages, type systems, ML language family, ML modules, recursive modules, first-class modules

TIIVISTELMÄ

Pauli Jaakkola: Tyyppijärjestelmä ensiluokkaisille rekursiivisille ML-moduuleille
Diplomityö
Tampereen yliopisto
Tieto- ja sähkötekniikka
Marraskuu 2020

Standard ML (SML), OCaml ja useat muut kielet ML-kieliperheessä omaavat erityisen ilmaisuvoimaisen moduulijärjestelmän. Tämä *ML-moduulijärjestelmä* tarjoaa hierarkkiset nimiavaruudet *struktuureilla*, hienojakoiset rajapinnat *läpikuultavilla tunnisteilla*, toteuttajan data-abstraktion *sinetöinnillä* ja käyttäjän data-abstraktion *funktoreilla*.

ML-moduuleilla varustetut kielet on *kerrostettu* kahteen kerrokseen: ydin-ML:ään ja moduuleihin. Näiden tasojen välillä esiintyy laajamittaista toiminnallisuuden toistoa: tietueet ja struktuurit, tyypit ja tunnisteet, funktiot ja funktorit. Myös moduulien toisluokkainen luonne rajoittaa ilmaisuvoimaa koska jotkin ominaisuudet ovat saatavilla vain ydin-ML:ssä tai vain moduuleissa.

Tyypillisesti ML-kielet rajoittavat rekursion vain tiettyihin muotoihin: `and`-avainsanalla yhdistettyjen keskinäisrekursiivisten funktio- tai tyyppimääritelmien ryhmiin. Tyyppimääritelmien rajoittaminen varmistaa, ettei *ekvirekursiivisia tyyppejä* tarvita. Funktiomääritelmien rajoittaminen varmistaa, että rekursiiviset arvot ovat *hyvin perustettuja* estäen ajonaikaisen alustamattomien arvojen käytön.

Rekursiiviset moduulit eivät ole osa Standard ML:n määritelmää mutta ovat tuettuja kielilaajennuksilla useissa SML-kääntäjissä kuten myös OCamlissa. Aivan kuten keskinäisrekursiiviset funktiot ja tyypit, keskinäisrekursiiviset moduulit täytyy poikkeuksetta ryhmitellä syntaktisesti yhteen. Rekursiivisten ML-moduulien toteutukset kamppailevat myös *kaksinnäön* ja *rekursiivisen linkityksen* ongelmien kanssa. Kaksinnäössä abstrakteja tyyppejä toteuttava moduuli epäonnistuu samaistamaan viemänsä abstraktit tyypit niiden toteutuksiin, joiden ei pitäisi olla olla kätkettyjä moduulin sisällä. Rekursiivisen linkityksen ongelma on elegantin ja staattisesti hyvin perustetun menetelmän löytäminen sellaiseen rekursiiviseen moduulien linkitykseen, jossa linkitettäviä moduuleja ei ole määritelty samassa rekursiivisten määritelmien ryhmässä tai ehkei edes samassa käännösyksikössä.

Artikkeli "1ML – Core and Modules United" luhisti menestyksekkäästi ydin-ML:n ja moduulit *yhdeksi kieleksi*, jossa on *ensiluokkaiset moduulit*. Rekursiiviset moduulit on otettu käyttön tuotantokääntäjissä ja kirjallisuudesta tunnetaan useita tapoja välttää kaksinnäkö. Ensiluokkaisia ja rekursiivisia moduuleja ei ole kuitenkaan yhdistetty vaikka saatavilla on välikieliä, jotka voivat sekä toimia 1ML:n tapaisten kehittelevien tyyppijärjestelmien kohteina että tukea rekursiivisia moduuleja.

Tämä diplomityö kehittää *Rekursiivisen 1ML:n (R1ML)*, joka on ensiluokkaisilla rekursiivisilla moduuleilla varustettu tyyppijärjestelmä. R1ML on 1ML:n tyylinen kehittelevä tyypijärjestelmä. R1ML:n kehittelykohde on GHC:tä (Glasgow Haskell Compiler) varten kehitetty välikieli System F_c. Haskellissa ei ole ML-moduuleja, mutta System F_c:n parametrinen polymorfismi ja tyyppien yhtäläisyyspakotukset ovat riittävän yleisiä tukeakseen ensiluokkaisia rekursiivisia moduuleja välttäen kaksinnäön. Tämä diplomityö myös todistaa, että R1ML on tyyppiturvallinen ja algoritmisesti ratkeava, mutta epätäydellinen useilla (siedettävillä) tavoilla. Rekursiivisen linkityksen ongelmien tarkasteluun ei ollut aikaa eikä tilaa.

Avainsanat: ohjelmointikielet, tyyppijärjestelmät, ML-kieliperhe, ML-moduulit, rekursiiviset moduulit, ensiluokkaiset moduulit

PREFACE

This thesis work was conducted at Tampere University in 2019 and 2020. It was not commissioned by any organisation; the subject arose from my continuing interests and I performed this research for its own sake.

I would like to thank my examiners Kari Systä and Jyrki Nummenmaa for their feedback, especially given such an unusual subject and lengthy thesis. I am also grateful to my employer Metosin for granting me paid and unpaid study leave to engage in such impractical research.

A special thanks to Vesa Karvonen for unexpected and insightful discussions around this thesis and 1ML. Most of all thanks to my spouse Sinikka for unwavering moral support as well as proofreading all this gibberish.

Tampere, 17th November 2020

Pauli Jaakkola

CONTENTS

LIST OF FIGURES

LIST OF PROGRAMS AND ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

B	a binding
E, e	expressions
P	a pattern
T	a type
Γ, Θ, Δ	type environments
Σ	a non-existential semantic type
Ξ	a semantic type
α, β	(rigid) type variables
ι	an effect
ϵ	the empty sequence or grammar production
γ	a coercion type
$\hat{\alpha}, \hat{\beta}$	unification variables
κ	a kind
σ	a type scheme
τ	a monotype
θ	a substitution
$\underline{\Sigma}$	a type template
c	an axiom variable
f, g	function expressions
x, y	term variables
1ML	1st-class Module Language
Caml	Categorical Abstract Machine Language
CPS	Continuation Passing Style
FIFO	First In First Out
GHC	the Glasgow Haskell Compiler
HM	Hindley-Milner
INRIA	Institut national de recherche en sciences et technologies du numérique
LCF	Logic for Computable Functions

LIFO	Last In First Out
ML^F	ML raised to the power of System F
OCaml	Objective Categorical Abstract Machine Language
R1ML	Recursive 1st-class Module Language
RTG	Recursive Type Generativity
SML	Standard ML
SML/NJ	Standard ML of New Jersey
STLC	simply typed lambda-calculus
System F_η	System F modulo η-expansion

1 INTRODUCTION

Recently there has been an unprecedented surge of interest in functional programming. Some of it can be attributed to the rise of ubiquitous multiprocessors and the difficulty of sharing mutable state correctly between threads. But the referential transparency of pure functions (side-effectless procedures) greatly simplifies reasoning even in single-threaded contexts and largely eliminates the need for mocks and stubs in testing. Object-oriented programming promised reliability and composability by encapsulating state into objects. Perhaps programmers are starting to see reasoning about state as so complicated that reliability and composability is better approached by minimizing the amount of state altogether than by just sweeping it under a rug of objects.

Modularity is the key enabling technology for successful ‘programming in the large’ – the development and maintenance of large programs and also the pervasive sharing of libraries that characterizes modern software development. Objects support modularity and interfaces as well as polymorphism and encapsulation of state. The composability and reliability of pure functions increases reusability but how does functional programming support programming in the large?

Most programming methodologies do not share the totalitarian zeal of object-oriented programming. Functional programming is primarily the enemy of state and has few opinions on constructs for programming in the large. Nevertheless the ML family of functional-first languages has another crown jewel beside Hindley-Milner type inference: the ML module system.

The ML module system was introduced during the creation of the Definition of Standard ML (SML) and is also found in slightly different form in the other mainline ML variant Objective Categorical Abstract Machine Language (OCaml). Unlike most module systems in use today, the ML module system is far from a minimal afterthought. It provides hierarchical namespace management with *structures*, fine-grained interfaces with *translucent signatures*, implementation-side data abstraction with *sealing* and client side data-abstraction with *functors*. The ML module system has extensive support not only for modularity and information hiding but also ‘generic programming’ [12].

The ML module system is not specific to functional programming or linked with other features of ML. Indeed modules and core ML interact so little that the Definition of Standard ML [27] separates core and modules into distinct sets of chapters. The upside of this stratification is that ‘ML modules’ could also be transplanted to most non-ML programming languages. Leroy's Modular Module System [24] amply demonstrates both the expressive power and language-independence of the ML module system by using functors to build a module system that is generic over the core language. Despite such undeniable constructive proof the module system has not spread outside the ML family or even to some ML variants such as Haskell, F# and Elm.

The downside of stratification is that positioning modules strictly above the core language makes them second-class citizens[1], which is inherently limiting. 1st-class Module Language (1ML) [38] managed to collapse modules and core into *one* language, with *first-class* modules by replacing stratification with a predicativity restriction.

The second-class nature of modules is shared by most other module systems for statically typed languages such as Ada and Haskell. Another typical shortcoming of module systems that ML modules are not exempt of is the limited (OCaml) or non-existent (SML) support for mutual recursion between modules. Often refactoring modules to be non-recursive improves the separation of concerns, but claiming that this is always the case is akin to Stockholm syndrome. Making the module dependency graph acyclic might require compromising the separation of concerns by mixing mutually recursive concerns in the same module or devising complicated and brittle workarounds to achieve separation without cross-module recursion.

This thesis develops a type system for Recursive 1st-class Module Language (R1ML), a programming language in the ML language family. Unlike SML or OCaml, R1ML has no stratification between core and module languages, seamlessly supporting first class modules. This unification is due to R1ML being largely based on 1ML.

R1ML is also more flexible than other strict ML dialects in its support for recursion, especially recursive modules. The well-known double vision issue between recursive modules and abstract types is avoided by elaborating to a variant of System F_c [44] instead of standard System F_ω like 1ML.

We start with a brief introduction to the ML language family with a focus on the module system including its main features and the limitations on recursion and of stratification. This is followed by introductions to the basics of evaluation and type theory needed to understand the remaining chapters. With that out of the way a chapter on the "F-ing Modules" approach and its culmination 1ML follows. Then the variant of System F_c used during typechecking and as the elaboration target of R1ML is defined. Having outlined those two primary foundations the R1ML type system itself is described in detail. Finally the viability of the R1ML type system is estimated with the usual metatheoretic properties of soundness, decidability and completeness. The thesis is concluded with thoughts on directions for following research.

[1]Because modules cannot be used as values in the core language.

2 ML AND MODULES

The ML module system stands as a high-water mark of
programming language support for data abstraction.
Nevertheless, it is not in a fully evolved state.

Understanding and Evolving the ML Module System [8]

This chapter provides basic information on ML in general and the ML module system in particular. The limitations on recursion and the second class nature of modules are also explained.

2.1 A Brief History of Standard ML and OCaml

Starting in 1973, the original ML was the MetaLanguage (implementation and scripting language) of the Edinburgh Logic for Computable Functions (LCF) theorem prover. Later its successors became general purpose functional-first programming languages. But as often happens the acronym stuck because naming is *hard*. So in the context of programming languages and this thesis ML is not an acronym for Machine Learning – or anything else.

ML is much like a statically typed Lisp, with garbage collection, first-class functions and syntactic sugar for singly-linked lists. Unlike the statically typed imperative languages of the time like Pascal (1970) [49] and C (1972) [15], ML is *strongly typed* – i.e. actually type-safe (see Section 4.1.1). ML came with additional cutting edge features like exception handling, full type inference and algebraic datatypes with pattern matching. Embarrassingly those features except exception handling are still cutting edge in the overall programming language landscape. The type system still evokes admiration:

> “The Hindley-Milner type system ([3]; [26]) is a masterpiece of design. It offered a big step forward in expressiveness (parametric polymorphism) at very low cost. The cost is low in several dimensions: the type system is technically easy to describe, and a straightforward inference algorithm is both sound and complete with respect to the specification. And it does all this for programs with no type annotations at all!” [46]

Despite all these advancements but typically for a scripting language the original ML lacked support for programming in the large. This lack was acknowledged while the

Definition of SML (1990, revised 1997 [27]) was being created. The solution was the exceptionally expressive **ML module system** first introduced by MacQueen in [25]. As witnessed by the epigraph of this chapter the module system was another crown jewel for SML.

Meanwhile on the other side of the Channel Institut national de recherche en sciences et technologies du numérique (INRIA) created their own dialect of ML called Categorical Abstract Machine Language (Caml) (1987) to develop their own theorem prover Coq. The bytecode-driven Caml Light (1991) reimplementation of Caml adopted a minimal module system in the vein of Modula-2 instead of the 'heavyweight' ML module system [22]. The Caml Special Light (1995) edition added a native code compiler – and Leroy's variant of the ML module system [23]. The final step to OCaml (2000) added an object system [35] which is rather unique but seldom used in comparison to the module system.

2.2 ML Modules

This section explains the various ML module constructs, but cannot serve as a proper guide to programming in ML. We start using modules immediately, explaining the incidental core ML syntax encountered on the way. The examples are mostly in SML, but OCaml has very similar features.

2.2.1 Structures

Lists can be used as immutable stacks. We can be more explicit about this mode of usage by creating a separate module of list stacks in Listing 2.1. Record-like modules that consist of core definitions and whose members can be accessed (from outside the module) with the usual dot notation are called **structures** in the ML module system.

```
structure ListStack = struct
    type 'a t = 'a list

    val empty = []
    val isEmpty = null
    val push = op::
    val pop = List.getItem
    val append = List.@
    val reverse = List.rev
end
```

Listing 2.1. *An immutable stack structure implemented with standard lists*

In `ListStack` the stack type is called `t` and is just an alias for the standard `list` type. Like `list` it is parameterized over the element type which is here called `'a`. The parameterization makes `list` and `ListStack.t` type level functions. In SML and OCaml the syntax of type level function application is backwards; what would be `ListStack Int` in Haskell or `ListStack<Integer>` in Java is `int ListStack.t` here.

The value members are implemented with the empty list `[]` and various list functions from the SML Basis top level and `List` module. The infix operator `::` corresponds to the Lisp `cons` function; `x :: xs` prepends the element `x` to the list `xs`, allocating a new list. The `op` keyword takes an infix operator's function value. List concatenation `@` is also an infix operator but `xs List.@ ys` is invalid syntax so `List.@` produces the concatenation function without needing to involve `op`.

For technical reasons that will be explained in Section 4.6.1 type functions must always be fully applied in ML. In a system with **higher kinded types** like Haskell `Stack` by itself is a valid type and can be passed around by value on the type level just like a higher order function is passed around by value on the term level.

2.2.2 Paths

As the previous section already showed value members of modules may be used in core ML expressions (e.g. `ListStack.empty`) and types (e.g. `ListStack.t`) via **paths**. For example `ListStack.empty` is a path expression that denotes the empty stack and `ListStack.t` is a path type that denotes the higher-kinded type of stacks. SML uses 'dotted paths' which consist of a module name followed by a sequence of structure member selections while applicative functors (see Section 2.2.8) allow OCaml to extend paths to mixed sequences of member selections and functor applications. Furthermore type paths must denote a `type` member of some structure.

The presence of type paths means that ML modules provide **path-dependent types**. In **dependently typed** systems types may depend on terms (i.e. include value-level expressions). With path-dependent types the dependence on terms is limited to type paths.

2.2.3 Signatures

Every module has an interface. In the ML module system interfaces are simply the types of modules and are called **signatures**. Listing 2.2 shows the signature inferred for `ListStack`.

In this inferred signature `t` is a **transparent** type member which means that its implementation as `list` is present in the signature. So `ListStack.t` is not a distinct type but is equal to `list`. All the value members also have types that relate to any list and not just lists that are meant to be used as stacks.

The transparency of the type and values means that the module has no information hiding. Clients of this module can use and abuse the implementation of stacks as lists. So the representation of these stacks cannot be changed without potentially breaking such clients. The clients can also freely violate the invariants (i.e. the Last In First Out (LIFO)

```
sig
    type 'a t = 'a list

    val empty : 'a list
    val isEmpty : 'a list -> bool
    val push : 'a * 'a list -> 'a list
    val pop : 'a list -> ('a * 'a list) option
    val append : 'a list * 'a list -> 'a list
    val reverse : 'a list -> 'a list
end
```

Listing 2.2. *The inferred signature of `ListStack`*

discipline) of these stacks.

2.2.4 Signature Matching

Instead it would be desirable to create a distinct **abstract type** of stacks. Listing 2.3 defines a signature for modules implementing abstract immutable stacks.

```
signature STACK = sig
    type 'a t

    val empty : 'a t
    val push : 'a * 'a t -> 'a t
    val pop : 'a t -> ('a * 'a t) option
end
```

Listing 2.3. *A signature for abstract immutable stack structures*

In this `STACK` signature the type of stacks `t` is an **abstract** type member as its implementation type is left unspecified. The only value members specified are the empty stack and the fundamental push and pop operations. The types of the value members refer to the abstract type `t` instead of some pre-existing concrete type like `list`.

Signature matching is a form of **structural record subtyping** (Section 4.2) of structures augmented with **translucency** of type members. `ListStack` **satisfies** the `STACK` signature since it has all the required members (**width subtyping** of signatures) with compatible types (**depth subtyping** of signatures) and the kind of `t` is also compatible. Translucency means that the abstract `type 'a t` of `STACK` can be satisfied with other abstract type members but also with concrete type members like `type 'a t = 'a list` here.

2.2.5 Type Refinement

Abstract type members of signatures can be **refined** into transparent ones by specifying their implementation types with `where type`[1]. The example Listing 2.4 refines the stack type `'a t` to be equal to `'a list`.

```
signature LISTED_STACK = STACK where type 'a t = 'a list
```

Listing 2.4. *Refining the `STACK` signature with `where`*

This signature declaration is equivalent to the one in 2.5, but more maintainable (the Don't Repeat Yourself principle). Note that the signature `LISTED_STACK` is not the same as the inferred signature of `ListStack` in Listing 2.2 since `LISTED_STACK` only contains the members of `STACK`.

```
signature LISTED_STACK = sig
    type 'a t = 'a list

    val empty : 'a list
    val push : 'a * 'a list -> 'a list
    val pop : 'a list -> ('a * 'a list) option
end
```

Listing 2.5. *The refined `STACK` signature*

`ListStack` implements both `STACK` and `LISTED_STACK`. `LISTED_STACK` obviously satisfies `STACK`, being refined from it. Conversely implementations of `STACK` that do not implement `'a t` with `'a list` will not satisfy `LISTED_STACK`.

2.2.6 Sealing

Queues are another familiar family of abstract types. Listing 2.6 defines a signature for modules implementing abstract immutable queues.

```
signature QUEUE = sig
    type 'a t

    val empty : 'a t
    val enqueue : 'a t * 'a -> 'a t
    val dequeue : 'a t -> ('a * 'a t) option
end
```

Listing 2.6. *A signature for abstract immutable queue structures*

This queue signature is very similar to `STACK`. There is an abstract type of queues `t` parameterized over the element type `'a`. The empty queue value is required as well as

[1]OCaml uses `with type` instead.

the fundamental queue operations: `enqueue` to add an element to the rear of the queue and `dequeue` to split the queue into the first element and the rest of the queue if possible. The stack and queue operations have almost the same types but different semantics, stacks being LIFO and queues First In First Out (FIFO). But as in most languages those semantics are not actually specified in the interface itself.

A particularly elegant way to implement immutable queues is to use two immutable stacks [29, sec. 5.2]. Such an implementation is shown in Listing 2.7.

```
structure ListsQueue :> QUEUE = struct
    type 'a t = {front : 'a ListStack.t, back : 'a ListStack.t}

    val empty = {
        front = ListStack.empty,
        back = ListStack.empty
    }

    fun fromRawParts (front, back) =
        if not (ListStack.isEmpty front)
        then {front, back}
        else {
            front = ListStack.reverse back,
            back = ListStack.empty
        }

    fun enqueue ({front, back}, elem) =
        fromRawParts (front, ListStack.push (elem, back))

    fun dequeue {front, back} =
        case ListStack.pop front of
        | SOME (head :: front, back) =>
            SOME (head, fromRawParts (front, back))
        | NONE => NONE
end
```

Listing 2.7. *An immutable queue structure implemented with* `ListStack`

A `ListsQueue.t` is a record of `front` and `back` stacks. In an empty queue both stacks are empty. Enqueueing pushes an element on top of the back stack. Dequeueing pops the first element of the queue from the top of the front stack and builds a new queue from the remaining elements. If the front stack is empty `NONE` is returned.

The `dequeue` function relies on the invariant that the back stack is only empty if the front stack is; specifically the first element of the queue is always in the front stack if present at all. This invariant is maintained by constructing every queue (except `empty`) with `fromRawParts`. If `fromRawParts` gets an empty front stack it uses the reversed back stack candidate as the front stack and makes the back stack empty. The reversal is what creates a FIFO from two LIFOs. And since the reversal only happens when the front stack is empty the queue operations take amortized O(1) time, assuming that the queues are used linearly (i.e. whenever a new queue is constructed the client program discards any old ones).

Sealing (`:>`) `ListsQueue` with `QUEUE` checks that the structure satisfies that signature and causes the `ListsQueue` module variable to have a signature that is like `QUEUE` but with `t` implemented by a fresh abstract type. Freshness means that `t` is not equal to its implementing record type from the definition of `ListsQueue` or any other type, including the `t`:s of other modules satisfying or even sealed with `QUEUE`. Apart from the creation of abstract types sealing is essentially an upcast along the structural subtyping relation of modules.

Sealing `ListsQueue` with `QUEUE` accomplishes information hiding since the definition of `ListsQueue.t` is hidden and all the value members have types that are only compatible with the fresh abstract type created by the sealing. The auxiliary function `fromRawParts` is also hidden so new queues can only be created by (repeated) uses of `ListsQueue.enqueue` and `ListsQueue.dequeue` on `ListsQueue.empty`. All this hiding of implementation details also enables relying on and maintaining invariants inside modules, like the queue module does with `fromRawParts`.

2.2.7 Functors

The queues of `ListsQueue` are immutable but not *persistent* [29] because amortization does not happen if they are used in a ‘non-linear’ or ‘multiple history’ manner. Typically that might happen in a backtracking algorithm that includes an immutable queue as part of its state[2].

One way to implement a fully persistent queue is to use a variant of the ‘queue from stacks’ strategy that utilizes lazy sequences. On the other hand the operations on such a queue have higher constant factors than `ListsQueue`. And implementing both would create two modules with a high degree of code duplication between them.

Even though we did not seal `ListStack`, `ListsQueue` only relies on its signature (which is larger than `STACK`) and not on the fact that `ListStack.t = list`. Listing 2.8 abstracts over the stack module used in the implementation of queues with a **functor**[3], which is just a module level function.

The domain signature of the functor extends `STACK` by `include`:ing it and adding the `append` and `reverse` operation that are used in the body of the functor in addition to the basic stack functionality. Structures also support this sort of (multiple) inheritance with `open`. The codomain of the functor is `QUEUE`, i.e. the results of applications of this functor will be sealed with `QUEUE`.

[2] Immutable data structures are convenient because capturing a snapshot consists of just storing the current version. The tradeoff is that operations on immutable data structures tend to be slower than on their mutable alternatives

[3] Confusingly Haskell, C++ and Prolog each define ‘functor’ quite differently from the ML usage and each other.

```
functor BankersQueueFun (Stack : sig
    include STACK

    val append : 'a t * 'a t -> 'a t
    val reverse : 'a t -> 'a t
end) :> QUEUE = struct
    type 'a t = {
        front : 'a Stack.t, frontLength : int,
        back : 'a Stack.t, backLength : int
    }

    val empty = {
        back = Stack.empty, backLength = 0,
        front = Stack.empty, frontLength = 0
    }

    fun balance queue =
        let {front, frontLength, back, backLength} = queue
        in
            if frontLength >= backLength
            then queue
            else {
                front = Stack.append (front, Stack.reverse back),
                frontLength = frontLength + backLength,
                back = Stack.empty, backLength = 0
            }
        end

    fun enqueue ({front, frontLength, back, backLength}, x) =
        balance {
            front, frontLength,
            back = Stack.push (x, back),
            backLength = backLength + 1
        }

    fun dequeue {front, frontLength, back, backLength} =
        case Stack.pop front of
        | SOME (x, front) => SOME (x, balance {
            front, frontLength = frontLength - 1,
            back, backLength
        })
        | NONE => NONE
end
```

Listing 2.8. *An immutable queue functor*

The body of the functor now tracks the sizes of the front and back stacks explicitly on the side. This is required because the new invariant maintained with `balance` is that the front stack is at least as tall as the back stacks. That invariant not only ensures that the back stack is always empty if the front stack is but also that the operations of the persistent queue will achieve amortized O(1) time (the proof can be found in [29, sec. 6.3.2]).

Listing 2.9 recovers `ListsQueue` by applying the functor `BankersListQueue` to `ListStack`.

```
structure ListsQueue = BankersQueueFun(ListStack)
```

Listing 2.9. *A concrete instantiation of `BankersQueueFun` using `ListStack`*

Listing 2.10 applies `BankersQueueFun` to `LazySeqStack`, producing the fully persistent queue.

```
structure PersistentQueue = BankersQueueFun(LazySeqStack)
```

Listing 2.10. *A persistent queue structure from `BankersQueueFun`*

The source code for `LazySeqStack` is not provided, but it would be essentially like `ListStack` but using a lazy sequence type and its operations instead of standard lists. An implementation of lazy sequences as a `Stream` structure is given in [29].

2.2.8 Functor Generativity

`ListsQueue.t` and `PersistentQueue.t` or rather their instantiations like `int ListsQueue.t` and `int PersistentQueue.t` are not equal – should not be since their implementations are different. But consider another list-backed queue module `BankersQueueFun` in Listing 2.11.

```
structure ListsQueue' = BankersQueueFun(ListStack)
```

Listing 2.11. *Another list-backed queue module*

Are the types `int ListsQueue.t` and `int ListsQueue'.t` equal or not? In SML functors have **generative** semantics where every functor application creates a new module with fresh abstract types. So in SML `int ListsQueue.t` is no more equal to `int ListsQueue'.t` than it is to `int PersistentQueue.t`. Generative functors could be compared to Java class constructors which always allocate `new` objects.

OCaml uses **applicative** functor semantics [23], where types like `ListsQueue.t` and `ListsQueue'.t` that are created by the application of the same functor to the same argument are considered equal. Applicative functor semantics can be seen as an extension of type constructor behaviour where `int list` is equal to any other `int list` type expression.

Applicative semantics can be convenient, especially for functors like like `BankersQueueFun` that implement generic data structures. However applicative functors are somewhat subtle and more difficult to formalize and implement robustly than generative ones. OCaml has issues with **abstraction safety**, especially with but not limited to stateful modules [40].

2.3 Recursion in ML

ML languages typically limit recursive definitions to very specific patterns. The restriction on type definitions ensures that **equirecursive types** are not needed. The restriction on function definitions ensures that recursive values are **well-founded**, preventing runtime use of uninitialized values.

2.3.1 Recursive Types

The algebraic datatypes created with `datatype` can be recursive like the types in Figure 2.12. A `file` can be either a `regular_file` or a `directory` and a directory can contain more `file`s.

```
datatype file = RegularFile of regular_file
              | Directory of directory
withtype regular_file = {name: string, size: int}
and directory = {name: string, files: file list}
```

Listing 2.12. *File system types*

Mutual recursion requires grouping the types together connected with `and` (which replaces the `datatype` or `type` token). In SML all type recursion must go through algebraic `datatype`s although `withtype` allows including `type` aliases in the recursion group. The mandatory syntactic grouping of mutually recursive types can be annoying from a code organization standpoint but does not fundamentally limit expressiveness. It is possible to work around the grouping limitation with fixpointing tricks but in the long run they tend to be too complicated or confusing to be a net win.

2.3.2 Recursive Terms

Handling recursive types usually requires recursive functions. Figure 2.13 shows functions that compute the total size of file system subtrees using pattern matching. The `size` function computes the size of a `file` by delegating to `regularFileSize` and `directorySize`. The recursion is in `directorySize`, which computes the sizes of its children with `size` and then sums them.

```
fun size (RegularFile file) = regularFileSize file
  | size (Directory dir) = directorySize dir

and regularFileSize {name = _, size} = size

and directorySize {name = _, files} =
    List.foldl op+ 0 (map fileSize files)
```

Listing 2.13. *File system functions*

Like algebraic datatypes, functions defined with `fun` can be self-recursive, and mutually recursive when connected with `and` like these size functions. Sometimes the alternative `val rec f = ... and g = ...` syntax yields shorter definitions but it is no more expressive than `fun f ... and g ...` since `val rec` definiends must be function literals.

Recursion requires forward references and allowing arbitrary forward references in terms would lead to use of uninitialized values at runtime. Restricting term recursion to groups of function literals is a simple way to prevent such unsafety but can be very limiting. OCaml has a more permissive *relaxed value restriction*. These restrictions are not needed in a lazy language like Haskell since premature access to values will cause an infinite loop instead of using uninitialized memory.

2.3.3 Recursive Modules

It would be more maintainable to make a module for each type of file and seal them with a common `FILE` signature as in Figure 2.14. Especially if the operations were more numerous or used system calls to traverse the actual filesystem.

```
signature FILE = sig
    type t
    val size: t -> int
end

structure File :> FILE = struct
    datatype t = RegularFile of RegularFile.t
               | Directory of Directory.t

    fun size (RegularFile file) = RegularFile.size file
      | size (Directory dir) = Directory.size dir
end

structure RegularFile :> FILE = struct
    type t = {name: string, size: int}

    val size: t -> int = #size
end

structure Directory :> FILE = struct
    type t = {name: string, files: File.t list}

    fun size (dir: t) =
        List.foldl op+ 0 (map File.size (#files dir))
end
```

Listing 2.14. *Modules for the file types*

These modules are mutually recursive and SML does not support module-level forward references at all. Various SML compilers as well as OCaml have extensions to support recursive modules. Invariably those extensions require grouping the modules just like recursive types and terms. In practice there are much fewer modules than types or especially terms, so functor fixpointing, although arcane, is not too verbose like type and term fixpointing. Incidentally the R1ML prototype uses recursive modules and even functor fixpoints – as it should, in the name of metacircularity.

Recursive modules are surprisingly difficult to implement. The grouping requirement of recursive terms must be somehow relaxed to allow term recursion between modules like the reference to `File.size` inside `Directory.size`. Type recursion between modules also runs into issues, especially so-called **double vision** of abstract types detailed in the next subsection.

For example the OCaml recursive modules extension raises an exception on premature forward references at runtime and also has some very technical restrictions on module recursion to make the checks efficient. Such solutions are against the statically safe and theoretically clean spirit of ML. The extension also does not prevent double vision in all cases.

2.3.4 Double Vision

Recursion between sealed modules can lead to the double vision problem [8, s. 97–100] where an abstract type created by sealing a module is seen as distinct from its implementation even inside the sealed module. The double vision problem is demonstrated in Figure 2.15, adapted from [6].

In this contrived example the structure `A` defines an abstract type `t` implemented as `int` and the structure `B` defines another abstract type `u` implemented as `bool`. The functions `A.f` and `B.g` handle both `A.t`:s and `B.u`:s and are mutually recursive. The double vision problem manifests at the call to `B.g` from `A.f` where the argument is of type `int` but `B.g` expects a value of type `A.t`, which it sees as abstract and thus incompatible with `int`. However `t` is defined as an alias of `int` inside `A` and the call should thus be valid. That requires seeing the type alias `t` and the abstract type `A.t` as one and the same, which is easily overlooked and requires nonstandard type checking techniques.

2.4 First-Class Modules

Figure 2.16 defines a signature `ORD` of ordered types. Signatures like this are used for parameters of functors that implement generic sorted maps and sets.

One could also expect to parameterize sorting functions over structures of signature `ORD`. Figure 2.17 shows how that might work for a merge sort on lists.

Unfortunately the stratification of ML into core and modules means that it is not possible to parameterize core terms over modules like this. Instead the module parameterization must be done with a functor that returns a structure which redundantly has the specialized

```
structure A :> sig
    type t
    type u = B.u

    val f : t -> u * t
end = struct
    type t = int
    type u = B.u

    fun f (x : t) : u * t =
        let val (y, z) = B.g (x + 3)
        in (y, z + 5)
        end
end

structure B :> sig
    type t = A.t
    type u

    val g : t -> u * t
end = struct
    type t = A.t
    type u = bool

    fun g (x : t) : u * t = ... A.f (...) ...
end
```

Listing 2.15. *A demonstration of the double vision problem*

```
signature ORD = sig
    type t
    val compare: t * t -> order
end
```

Listing 2.16. *Signature for ordered types*

sorting function as its only member as in Figure 2.18.

The two-line usage syntax at the end of Figure 2.18 is quite verbose for the one-off usages typical of sorting functions. Another option is to parameterize over just the ordering function as in Figure 2.19.

The usage syntax is much more convenient, but core-level workarounds like this scale badly for modules with more complicated signatures. OCaml and various SML compilers have a **packaged modules** extension which allow injecting modules to and projecting them from core-level package values. As Listing 2.20 shows, OCaml packaged modules are sufficient to express simple cases like this.

Besides having verbose and potentially confusing syntax (you are not expected to understand Listing 2.20), packaged modules are also quite limited. For example it is not

```
fun merge (Elem : ORD) =
    fn (xs, []) => xs
     | ([], ys) => ys
     | (x :: xs, y :: ys) =>
         (case Elem.compare (x, y)
          of LESS | EQUAL => x :: merge (xs, y :: ys)
           | GREATER => y :: merge (x :: xs, ys))

fun mergeSort (Elem : ORD) =
    fn xs as [] => xs
     | xs as [_] => xs
     | xs => let val rec split =
                 fn xs as [] => (xs, xs)
                  | xs as [_] => (xs, [])
                  | x :: x' :: xs =>
                    let val (ls, rs) = split xs
                    in (x :: ls, x' :: rs)
                    end
             in merge Elem (split xs)
             end
```

Listing 2.17. *Generic merge sort*

```
functor MergeSortList(Elem: ORD) :> sig
    val mergeSort: Elem.t list -> Elem.t list
end = struct
    val merge = fn ...
    val mergeSort = fn ...
end

structure MergeSortIntList = MergeSortList(Int)
val sorted = MergeSortIntList.mergeSort [5, 42, 17, 23]
```

Listing 2.18. *Modules workaround for generic merge sort*

```
fun merge compare =
    ...
     | (x :: xs, y :: ys) =>
        (case compare (x, y) ...

fun mergeSort compare =
    ...
    in merge compare (split xs)
    end

val sorted = mergeSort Int.compare [5, 42, 17, 23]
```

Listing 2.19. *Core workaround for generic merge sort*

```
let merge (type a) (module Elem : ORD with type t = a) = function
    ...
    | ((x :: xs : a list), (y :: ys : a list) ->
        (match Elem.compare (x, y) with ...

let mergeSort (type a) (elem : module (ORD with type t = a)) =
    ...
    merge elem (split xs)

let sorted = mergeSort (module Int : ORD) [5; 42; 17; 23]
```
Listing 2.20. *Generic merge sort with OCaml packaged modules*

possible to abstract over modules of signature `STACK` due to the lack of higher kinded types in core ML. For example Figure 2.21 shows a generic stack `dup` function using type classes in Haskell that cannot be expressed with packaged modules. Such type class constraints (particularly the ever-present `Monad`) on higher-kinded types are pervasive in Haskell.

```
dup :: Stack t => t a -> Maybe (t a)
dup xs =
    case pop xs of
        Just (x, _) -> push x xs
        Nothing -> Nothing
```
Listing 2.21. *Generic `dup` function in Haskell*

The stratification between core and modules also prevents selecting between modules at runtime. One example would be selecting a graphics module based on what OpenGL version is supported and then using dependency injection with functors to specialize the entire application for the present GPU features.

Packaged modules partially recoup the loss of expressiveness due to the stratification between core and modules. But they do nothing to solve the language design issue of duplication between records and structures, types and signatures, functions and functors.

ML modules include path-dependent types and structure subtyping. The unification-based Hindley-Milner type inference algorithms are too weak to handle either subtyping or dependent types. Subtyping is a preorder and unification variables can only handle equality constraints. And dependent types require the evaluation of value-level terms. Additionally merging signatures into types would enable impredicative instantiation which in combination with the contravariance of functor subtyping would enable a pathological class of programs that causes the type checker to diverge, rendering the type system undecidable [38].

But the type paths of ML modules are not really dependent because they have to be **separable**: type paths must refer to type members and cannot depend on any dynamic computation. Type inference for structural records[4] is already incomplete in SML, without subtyping. And impredicativity may be ruled out by directly enforcing predicative instantiation instead of indirectly (but incredibly heavy-handedly) with stratification into core and module languages. Knowing all this the 1ML paper [38] could give a type system that merges core and modules into *one* language, with truly *first-class* modules. As a side effect of the unification of core and modules 1ML also gains various features like higher-rank types, existential types and record subtyping.

2.5 R1ML

1ML does not support recursive modules, as Rossberg feels that a satisfactory treatment of recursive modules requires a complete overhaul of the ML module system as in his earlier paper [39] on MixML with Dreyer. Inversely all module systems with recursive modules are stratified.

In this thesis I develop a type system with both first-class and recursive modules as well as partial type inference. The system is called R1ML, being mostly based on 1ML. I am not aware of any other type system with both first-class and recursive modules.

Because of the double vision problem recursive modules cannot be implemented by elaborating to standard System F_ω like 1ML. It would be natural to use an elaboration target specifically designed to support recursive modules like Dreyer's Recursive Type Generativity (RTG) [7] used in [6] and also in refined form for MixML. Instead I have chosen to use a variant of System F_c [44], which was created to serve as the main intermediate language in the Glasgow Haskell Compiler (GHC). Unlike RTG, System F_c has certainly been proven in the field. GHC elaborates many advanced constructs like GADTs, type classes and type families into System F_c [46]. However R1ML is the first to use System F_c to solve double vision.

[4]In OCaml records are nominative (like Java classes) instead of structural (like objects in Typescript).

3 UNTYPED LAMBDA-CALCULUS

There may, indeed, be other applications of the system than its use as a logic.

Alonzo Church

The untyped λ-calculus [32, ch. 5] is a minimal function application and variable reference -based programming language. Originally it was devised by Alonzo Church and his coworkers in the 1920s and '30s as a tool for research in the foundations of mathematics. The term 'untyped' might seem dismissive of dynamic typing but on the other hand dynamically typed languages usually have more types than the untyped lambda-calculus where everything is a function.

The abstract syntax of the untyped lambda-calculus is defined in Figure 3.1. In programming language formalizations abstract syntax definitions like this are more like `datatype` definitions for syntax trees than grammars for parsing.

$$e ::= x \mid \lambda x.e \mid e\, e$$

Figure 3.1. *Abstract syntax of λ-calculus*

The untyped λ-calculus consist of just function literals $\lambda x.e$, function application $e\,e$ and variable references x. The titular λ corresponds to the more descriptive `function`, `func`, `fun` or `fn` of practical programming languages. All functions have one parameter, but multiple parameters can be emulated by currying or tuple parameters. Function application is denoted by juxtaposition like multiplication in algebra.

3.1 Term Equivalence

An **equivalence relation** is a binary relation that is reflexive, symmetric and transitive. Intuitively, two lambda-calculus terms are equivalent if they implement the same function. This extensional notion of λ-term equivalence is defined syntactically by the rules in Figure 3.2.

The rules use the usual two-dimensional Gentzen-style notation where the expressions above the long horizontal inference line are premises and the expression under the line is the conclusion, i.e. $\frac{A\;B}{C}$ is just another way to write $A \wedge B \Rightarrow C$. The substitution notation $[e_a/x]e$ replaces each occurrence of x in the expression e with the expression e_a.

$$\textsc{Refl}\quad e = e \qquad \textsc{Symm}\quad \frac{e' = e}{e = e'} \qquad \textsc{Trans}\quad \frac{e = e'' \qquad e'' = e'}{e = e'}$$

$$\alpha\quad \frac{y \notin \mathsf{fv}(e)}{\lambda x.e = \lambda y.[y/x]e} \qquad \beta\quad (\lambda x.e)\, e_a = [e_a/x]e \qquad \eta\quad \frac{x \notin \mathsf{fv}(e)}{\lambda x.e\, x = e}$$

Figure 3.2. *Extensional equivalence of* λ*-terms*

We take reflexivity, symmetry and transitivity as definitions in the form of rules Refl, Symm and Trans. Rule Refl can also be read as the obvious statement that two syntactically identical terms are equivalent.

Lambda calculus also has three other well-known rules for term equivalence. **Alpha equivalence** in rule α formalizes the intuition that variable names do not matter; parameters can be renamed as long as **variable capture** is avoided (with the side condition $y \notin \mathsf{fv}(e)$). **Beta equivalence** of rule β defines immediate function application; a call to a known function is equivalent to the body of the function with the argument substituted for the parameter. **Eta equivalence** of rule η means that a function that just immediately calls another function on its parameter is equivalent to that other function. In more practical systems eta equivalence only holds if the inner function expression has no side effects, but plain lambda-calculus has no notion of side effects.

3.2 Evaluation

In day to day programming term equivalence is mostly interesting for small scale refactoring. Term equivalence has no directionality; β and η can be used to insert useless function applications and wrapper functions as well as removing them.

In practice we are more often interested in program execution. The corresponding formal notion is **evaluation** which consists of **rewriting** the program until it has been **reduced** to a **normal form**. A **reduction relation** is similar to an equivalence relation, consisting of rewriting rules that sometimes mirror the equivalence rules. It is also reflexive and transitive, but *not symmetric*, which gives it the desired directionality. The normal form form a term in a given rewriting system is one where it cannot be reduced any further, i.e. only reflexivity can be applied. Normal form terms are also called **values**.

Figure 3.3 defines the reduction relation for lambda-calculus. The use of an arrow is a constant reminder of that directionality.

So reduction in lambda-calculus is very similar to equality, but always proceeds in the direction of fewer beta and eta **redexes**, i.e. terms to which the rule β or η can be applied. Transitivity serves to apply β and η multiple times. Alpha renaming is only useful to ensure that the variable capture precondition of η can be satisfied and in the

$$\textsc{Refl}\quad e \to e \qquad\qquad \textsc{Trans}\quad \frac{e \to e'' \qquad e'' \to e'}{e \to e'}$$

$$\alpha\quad \frac{y \notin \mathsf{fv}(e)}{\lambda x.e \to \lambda y.[y/x]e} \qquad \beta\quad (\lambda x.e)\, e_a \to [e_a/x]e \qquad \eta\quad \frac{x \notin \mathsf{fv}(e)}{\lambda x.e\, x \to e}$$

Figure 3.3. *Reduction rules for λ-terms*

definition of capture avoiding substitution for β.

Usually reduction relations are presented with only the rules specific to that particular relation, reflexivity and transitivity left implicit. Similarly, the **Barendegt naming convention**, which states that variables are implicitly renamed whenever variable capture would occur, is often used. These conventions condense our reduction rules to Figure 3.4

$$\beta\quad (\lambda x.e)\, e_a \to [e_a/x]e \qquad\qquad \eta\quad \lambda x.e\, x \to e$$

Figure 3.4. *Compact presentation of reduction rules for λ-terms*

From now on, this thesis uses the Barendregt naming convention throughout. Reflexivity and transitivity are handled more specifically than the implicitness convention, as we shall soon see.

3.3 Evaluation Strategies

So reduction is directional, unlike equivalence. However the reduction relation in the previous section does not specify the *order* of reductions. In other words it is non-deterministic and must be refined to a specific algorithm to be implemented in practice. The algorithms also have corresponding normal forms that may differ from the normal form of the 'declarative' reduction relation of the previous section.

Full reduction does reductions everywhere in some order. Its normal form is the declarative normal form. But because lambda-calculus is Turing-complete, full reduction does not terminate on all expressions. Performing reductions inside functions is useless work in an interpreter or compiled code but is desirable for aggressive compiler optimizations like partial evaluation.

Call by value reduces the leftmost innermost redex first. Reductions are not made inside functions. More concretely the callee is evaluated, then the argument and then the resulting β-redex. The name comes from the fact that in this strategy the arguments in β-reductions are always in normal form. This corresponds to the runtime semantics of ML and most other programming languages. Call by value does less useless work than full reduction but is still susceptible to nontermination. It is easy to follow, which is important for debugging the order of side effects as well as performance tuning.

Call by name reduces the leftmost *outermost* redex first. Reductions are not made inside functions. In practice the callee is evaluated and applied to an unevaluated argument. This results in **weak head normal form**, which can contain redexes even outside function terms. Call by name is a **normal order** strategy, always finding the normal form of well-formed terms instead of nontermination. On the other hand unevaluated arguments can be duplicated, resulting in useless work.

Call by need avoids that duplication of work with memoization. **Lazy** programming languages like Haskell require a non-strict evaluation strategy, using call by need in practice. The work saving potential of lazy evaluation is often squandered in practice by its complex implementations and difficulty of performance tuning. The unpredictability of side effects has been praised as a blessing in disguise, keeping Haskell purely functional unlike Lisp or ML which used call by value and thus “succumbed to the siren call of side effects”.

This thesis always assumes that call by value is used for term evaluation, although often it does not even matter in type systems. Call by need is used for evaluation on the type level that is required for higher kinded types, because in that context it actually is likely to reduce the net amount of evaluation work.

We will omit η from our reduction relations, as is usually done in practical evaluators. Reduction algorithms will be presented in **one-step** form as in Figure 3.5, which shows call by value for lambda calculus.

$$\textsc{App}\quad \frac{e_f \longrightarrow \lambda x.e \qquad e_a \longrightarrow e'_a \qquad [e'_a/x]e \longrightarrow e'}{e_f\, e_a \longrightarrow e'}$$

Figure 3.5. *Call by value evaluation for lambda-calculus*

This definition is recursive, so that transitivity is not needed, even implicitly. The longer arrow is a reminder that applying this rule once does all the reductions needed to reach normal form.

4 TYPE THEORY PRELIMINARIES

Learn all there is to learn and then choose your own path.

Georg Friedrich Händel

This chapter is a fast-paced tutorial on the type theory prerequisites needed to understand Chapters 7 and 8, from the simply typed lambda-calculus up to System F_ω. Readers unfamiliar with type theory may wish to also study a textbook like [32] and type theory aficionados should expect to skim through the majority of this chapter.

4.1 Simply Typed Lambda-Calculus

The simply typed lambda-calculus [32, ch. 9] is a statically typed version of the lambda-calculus. It is 'simply typed' because it does not have any form of polymorphism (Sections 4.2 and 4.3) or nontrivial kinds (Section 4.6). The abstract syntax of the simply typed lambda-calculus is given in Figure 4.1

$$\begin{aligned} e &::= x \,|\, \lambda x : T \,.\, e \,|\, e\, e \\ T &::= T \to T \end{aligned}$$

Figure 4.1. *Abstract syntax of the simply typed lambda-calculus*

The only change from the term language of the untyped lambda-calculus is addition of compulsory type annotations on function parameters. The types of the simply typed lambda-calculus consist of just function types $T \to T$. In practice non-function terms and types like `true` and `bool` are required, because (finite) type expressions cannot consist of just $\to$:s.

4.1.1 Type Checking

Since type annotations are compulsory on function parameters, type checking terms in simply typed lambda-calculus is straightforward. The type checking rules are displayed in Figure 4.2.

The type of a variable reference is determined by rule VAR by looking it up from the **type environment** Γ. The type environment is a mapping from variable names to types. In type

$$\textsc{Var}\ \frac{(x:T)\in\Gamma}{\Gamma\vdash x:T}\qquad \textsc{Abs}\ \frac{\Gamma,x:T_d\vdash e:T_c}{\Gamma\vdash\lambda x:T_d.e:T_d\to T_c}\qquad \textsc{App}\ \frac{\Gamma\vdash e_f:T_d\to T_c\qquad \Gamma\vdash e_a:T_d}{\Gamma\vdash e_f\,e_a:T_c}$$

Figure 4.2. *Typing rules for the simply typed lambda-calculus*

theory the turnstile symbol $\vdash$ just separates the type environment from the expression being typechecked.

The codomain type T_c of a function abstraction $\lambda x.e$ is determined in rule Abs by typing the body e in a type environment extended with $x : T_d$. The domain type T_d is taken directly from the type annotation of the parameter x. Finally an arrow type (i.e. function type) $T_d \to T_c$ is built from the domain and codomain types. Since Abs assembles arrow types, it is an **introduction rule** and $\lambda x : T.e$ is an **introduction form** for arrow types.

Function applications are typed by rule App by typing the callee and argument expressions in the current type environment. The callability of the callee is ensured by checking that it has a function type. The type of the argument must be syntactically equal to the codomain type of the callee ($\Gamma \vdash e_a : T_c$), which ensures that the function will accept the argument. Since App requires values of arrow type and destructures said arrow type, it is an **elimination rule** and $e_f\,e_a$ is an **elimination form** for arrow types.

The simply typed lambda-calculus is **type-safe** or **sound**: the evaluation of a well-typed term will not result in a situation where the term has not reached normal form but no evaluation rule applies [32, sec. 8.3]. For contrast consider the dynamically typed Javascript where "undefined is not a function" is a frequent error message or the statically typed but unsound C with its numerous sources of **undefined behaviour**. More formally, a Turing-complete statically typed language is sound iff the evaluation of every well-typed term either produces a value (reaches normal form) or diverges (never halts). A type system can also be sound only for a particular evaluation strategy like call by value or call by name.

Unlike the untyped lambda-calculus, the simply typed lambda-calculus is *not Turing-complete*. One sign of this is that the Y-combinator is ill-typed in the simply typed lambda-calculus. Indeed the simply typed lambda-calculus is provably **strongly normalizing**. A statically typed language is strongly normalizing iff every sequence of reductions starting with a well-typed term eventually reaches normal form. Strong normalization implies that the type system is sound for every possible evaluation strategy, including full reduction.

The type system of an implementable language must be **decidable**. In general, a decision problem (a yes-no question) is decidable iff there exists an effective method to find the answer. In the case of type systems the decision problem is 'is a given program well-typed?' and the effective method is a typechecking algorithm.

Conversely to decidability, a type checking function is **complete** iff it will accept every well-typed program. When typechecking algorithms are incomplete they typically will reject well-typed programs in some obscure corner cases or the type checker diverges[1] on some pathological programs. For instance Haskell type checkers are incomplete because type classes allow certain classes of ambiguous programs [46] and the OCaml type checker can be made to diverge with contrived programs involving abstract signature members [38].

The simply typed lambda-calculus is decidable. The typechecking algorithm is a function which takes the expression and type environment and returns the type of the expression. The rules of the simply typed lambda-calculus are **syntax-directed**, which means that the shape of the expression completely determines which rule to apply. The typechecking function, sketched in Listing 4.1 can be constructed simply by transliterating the consequences as pattern matching[2] on the expression and the premises as the code in each branch.

```
datatype typ =
    | Arrow of typ * typ
    | Bool

datatype expr =
    | Abs of string * typ * expr
    | App of expr * expr
    | Var of string

structure Env = SortedMap(String)

fun typeof env = fn
    | Abs (param, domain, body) =>
        let val env = Env.insert env param domain
            val codomain = typeof env body
        in Arrow (domain, codomain)
        end
    | App (callee, arg) =>
        (case typeof env callee of
        | Arrow (domain, codomain) =>
            let val arg_t = typeof env arg
            in  if arg_t = domain
                then codomain
                else raise Fail "wrong␣argument␣type"
            end
        | _ => raise Fail "not␣a␣function")
    | Var name =>
        (case Env.find env name of
        | SOME t => t
        | NONE => raise Fail "unbound␣variable")
```

Listing 4.1. *Sketch of a type checker for the simply typed lambda-calculus*

[1]Usually some arbitrary limit on e.g. recursion depth is put into place to force the type checker to fail instead.

[2]Or branching in some other manner like an `if-else` chain or method dispatch.

To prove that the type checking function is an algorithm we only need to note that the rules are also **inductive**, which means that the problem gets smaller in every recursive invocation. Type systems where the rules can be directly read as a type checking algorithm are called **algorithmic**. Algorithmic type systems are usually syntax-directed, but not all syntax-directed systems are algorithmic.

4.1.2 Extensions

The minimal term and type language of the simply-typed lambda-calculus is of course impractical. In this section we extend it with various forms of terms that are relevant to the rest of this thesis. Figure 4.3 shows the extended terms and types. Subsequent sections of this chapter will focus on polymorphism and nontrivial kinding.

$$
\begin{aligned}
e ::= {} & x \mid \lambda x : T.e \mid e\, e \mid \mathsf{let}\ x : T = e\ \mathsf{in}\ e \mid \mathsf{let\ rec}\ \overline{x : T = e}\ \mathsf{in}\ e \mid e : T \\
& \mid \{\overline{x = e}\} \mid e.x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
T ::= {} & T \to T \mid \{\overline{x : T}\} \mid \mathsf{bool}
\end{aligned}
$$

Figure 4.3. *Abstract syntax of an extended simply typed lambda-calculus*

Here nonrecursive (`let`) and recursive (`let rec`) local definitions as well as type ascription $e : T$ have been added. We also have introduction and elimination forms and type expressions for records and booleans. The typing rules for these added constructs are in Figure 4.4.

LET
$$\frac{\Gamma \vdash e_x : T_x \qquad \Gamma, x : T_x \vdash e : T}{\Gamma \vdash \mathsf{let}\ x : T_x = e_x\ \mathsf{in}\ e : T}$$

LETREC
$$\frac{\overline{x} = \mathsf{dedup}(\overline{x}) \qquad \overline{\Gamma, \overline{x : T_x} \vdash e_x : T_x} \qquad \Gamma, \overline{x : T_x} \vdash e : T}{\Gamma \vdash \mathsf{let\ rec}\ \overline{x : T_x = e_x;}\ \mathsf{in}\ e : T}$$

ASCRIBE
$$\frac{\Gamma \vdash e : T}{\Gamma \vdash (e : T) : T}$$

RECORD
$$\frac{\overline{\Gamma \vdash e : T} \qquad \overline{x} = \mathsf{dedup}(\overline{x})}{\Gamma \vdash \{\overline{x = e}\} : \{\overline{: T}\}}$$

SELECT
$$\frac{\Gamma \vdash e : \{\overline{x_e : T_e}\} \qquad x : T \in \overline{x_e : T_e}}{\Gamma \vdash e.x : T}$$

TRUE
$$\Gamma \vdash \mathsf{true} : \mathsf{bool}$$

FALSE
$$\Gamma \vdash \mathsf{false} : \mathsf{bool}$$

IF
$$\frac{\Gamma \vdash e_c : \mathsf{bool} \qquad \Gamma \vdash e_t : T \qquad \Gamma \vdash e_f : T}{\Gamma \vdash \mathsf{if}\ e_c\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : T}$$

Figure 4.4. *Typing rules for extensions to the simply typed lambda-calculus*

Nonrecursive local definitions are very easy to type (rule LET). Just type e_x and check that the computed type matches the type annotation T_x. Then type the body expression e in a type environment extended with $x : T_x$. Since the value of a `let`-expression is that of its body, so is its type.

LET can actually derived from ABS and APP by considering `let` as syntactic sugar (and an optimization) for a β-redex $(\lambda x : T_x.e)\, e_x$. On the other hand, quite contrary to ABS, LET has the property that the type annotation T_x only serves as a type assertion or documention. The annotation is not strictly necessary for typechecking since the type of e_x can and must be computed without using it. Just making the type annotation optional provides a trivial amount of type inference but systems like the Hindley-Milner type system and ML^F go much further with special treatment for `let` (Section 4.4).

Groups of mutually recursive local definitions are not much harder to typecheck (rule LETREC). The $\overline{x}$ notation means "zero or more" and is used for all sorts of elementwise operations in 'computer science metanotation'. The definitions just have to be checked in a type environment that has already been extended with their types from the annotations $\overline{x : T_x}$. To prevent ambiguity each name must have just one definition ($\mathtt{dedup}(\overline{x})$). Thus unlike LET, LETREC crucially relies on the type annotations. Despite that, monomorphic type inference for `let rec` is quite easy. On the other hand type inference in the general polymorphic case is undecidable [32, p. 338].

In simply typed lambda-calculus type ascription $e : T$ just asserts statically that e has type T as can be seen from rule ASCRIBE. In polymorphic systems ascription can be generalized so that it can also **upcast** or **specialize** e to the type T. Here and even then the ascription rule can actually be derived by considering $e : T$ as syntactic sugar for let $x : T = e$ in x.

Records are central to this thesis thesis since structures are basically records that can additionally have types as members. According to rule RECORD, record construction $\{\overline{x = e}\}$ produces a record of type $\{\overline{x : T}\}$ where each field name x (**label**) is paired with the type T of its initializing expression e. Duplicate labels are not allowed. Rule SELECT states that projecting from a record (selecting a field by label) produces a value of the type corresponding to that label in the record's type.

The boolean constants `true` and `false` obviously have type `bool`. The `if` condition must also have type `bool`. The branches must have the same type. As we will see later, in more advanced systems matching up the branch types creates challenges specific to conditionals, all the more reason to include them in systems in this thesis.

4.1.3 Type Assignment

As we have seen, typechecking is straightforward when type annotations are compulsory on every variable definition. However typing those type annotations in even when they seem obvious or even redundant (e.g. `ArrayList<Foo> foos = new ArrayList<Foo>();`) quickly gets tedious. Any succeeding programming language has many more users than compiler writers, so reducing the amount of required type annotations is a profitable idea.

In **type assignment** systems terms contain no type annotations at all but types are still checked statically. Type checking is obviously much more difficult in such an **implicitly typed** system than in the explicitly typed one we have seen so far. For many systems the implicitly typed variant is altogether undecidable.

The terms of the implicitly simply typed lambda-calculus are those of the untyped lambda-calculus. Its type assignment rules are given in Figure 4.5.

$$\textsc{Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{Abs}\ \frac{\Gamma, x : T_d \vdash e : T_c}{\Gamma \vdash \lambda x.e : T_d \to T_c} \qquad \textsc{App}\ \frac{\Gamma \vdash e_f : T_d \to T_c \qquad \Gamma \vdash e_a : T_d}{\Gamma \vdash e_f\, e_a : T_c}$$

Figure 4.5. *Type assignment rules for the simply typed lambda-calculus*

These type assignment rules are identical to the type checking rules of Figure 4.2 except that in rule Abs the domain type T_d is just conjured up from thin air. Such guessing means that the rules do not directly describe an algorithm, but they still specify a logical relation.

Perhaps surprisingly the system of Figure 4.5 is decidable, using algorithms based on unification. But it does have the peculiarity that typings are not unique. Practically this means that terms such as

$$\mathsf{let}\ identity = \lambda x.x\ \mathsf{in}\ \mathsf{false}$$

do not fully constrain the types of variables; x could have any single simple type. The algorithm would have to leave the choice open like Prolog or use some arbitrary default. The Hindley-Milner system of Section 4.4 avoids the ambiguity by inferring a polymorphic type for $identity$.

4.2 Subtyping

A type T is a **subtype** [32, ch. 15] of another type T' if all the terms that have type T also have type T'. T' is also allowed to include terms that do not have type T, making subtyping more permissive than syntactic equality. Formally subtyping is a **preorder** on types while syntactic equality is an equivalence relation. Preorders must be reflexive and transitive but are not required to be symmetric like equivalence relations. Subtyping is also a form of **polymorphism** because terms can be ascribed more than one type.

The exact subtyping behaviour of a type system is specified by a **subtyping relation**. As a preorder the subtyping relation is usually denoted by an infix operator $<:$ or $\leq$. The left operand type of $<:$ contains fewer terms than the right operand type. The left operand is a **subtype** of the right operand and conversely the right operand is a **supertype** of the left operand.

In theory subtyping can be introduced into a type system like the simply typed lambda-calculus by the **subsumption rule** Sub in Figure 4.6. Sub simply states that values of type T can be used as values of type T', given that T is a subtype of T'.

$$\textsc{Sub}\quad \frac{\Gamma \vdash e : T \qquad T <: T'}{\Gamma \vdash e : T'}$$

Figure 4.6. *Declarative subsumption rule*

The subsumption rule is orthogonal which is nice for a declarative *specification* of well-typed terms. On the other hand it is not syntax-directed since it can be used for any expression e. Figure 4.7 shows how orthogonality can be traded off to regain a syntax-directed system.

$$\textsc{App}\quad \frac{\Gamma \vdash e_f : T_d \to T_c \qquad \Gamma \vdash e_a : T_a \qquad T_a <: T_d}{\Gamma \vdash e_f\, e_a : T_c}$$

Figure 4.7. *Application rule with domain subtyping*

Fortunately subsumption turns out to only be essential for arguments of function applications App so it can be integrated there instead of adding Sub. In the modified App rule the argument type can be any subtype of the domain instead of having to be identical. [32, ch. 16]

4.2.1 Record Subtyping

Intuitively, if a piece of code requires a value of structural record type $\{\overline{x : T}\}$ it should also work with records with a superset of the fields $\overline{x : T}$. Slightly counterintuitively a record type with *more* fields contains *fewer* values, e.g. $\{x = 5\}$ has types $\{x : int\}$ and $\{\}$ but not $\{x : int, y : int\}$. Both lines of thought lead to the structural record subtyping relation in Figure 4.8. This relation is reflexive and transitive. [32, ch. 16]

$$\textsc{SubFns}\quad \frac{T_d' <: T_d \qquad T_c <: T_c'}{T_d \to T_c <: T_d' \to T_c'} \qquad \textsc{RecordSub}\quad \frac{T <: T' \qquad \{\overline{x_1 : T_1}, \overline{x_2 : T_2}\} <: \{\overline{x_3 : T_3}\}}{\{\overline{x_1 : T_1}, x : T, \overline{x_2 : T_2}\} <: \{x : T', \overline{x_3 : T_3}\}} \qquad \textsc{RecordDrop}\quad \{\overline{x : T}\} <: \{\}$$

Figure 4.8. *Record subtyping*

According to rule RecordSub each field x of the record supertype must be found in the record subtype. RecordSub recurses on both the types of the field x and the fields beside x. The type T of the x field in the subtype must also be a subtype of the corresponding field type T' in the supertype. When a part of the subtype must be a subtype of the corresponding part in the supertype like this the subtyping relation is **covariant** in that part. The RecordSub recursion on the remaining fields ensures that all fields of the supertype are checked and the base case RecordDrop allows the subtype to have fields not found in the supertype.

Finally rule SubFns states that functions are also covariant in their codomains ($T_c <: T'_c$) but **contravariant** in their domains ($T'_d <: T_d$). The reasoning behind this is that a function *consumes* arguments with subtypes of its domain (parameter) type and *produces* return values of subtypes of its codomain type.

Structural record subtyping is very relevant to this thesis since it is a large part of module signature matching. It is quite different from the **nominative** subtyping of statically typed object-oriented programming which only considers the explicit inheritance hierarchy, ignoring the inner structure of classes like field and method definitions.

4.2.2 Impact on Other Extensions

Besides records which motivated the introduction of subtyping, the other extensions from Section 4.1.2 can also be incorporated into systems with subtyping. Figure 4.9 shows their typing rules modified to accomodate subtyping.

$$\textsc{Let}\quad \frac{\Gamma \vdash e_x : T'_x \qquad T'_x <: T_x \qquad \Gamma, x : T_x \vdash e : T}{\Gamma \vdash \text{let } x : T_x = e_x \text{ in } e : T}$$

$$\textsc{LetRec}\quad \frac{\overline{x} = \text{dedup}(\overline{x}) \qquad \overline{\Gamma, \overline{x : T_x} \vdash e_x : T'_x} \qquad \overline{T'_x <: T_x} \qquad \Gamma, \overline{x : T_x} \vdash e : T}{\Gamma \vdash \text{let rec } \overline{x : T_x = e_x;} \text{ in } e : T}$$

$$\textsc{Upcast}\quad \frac{\Gamma \vdash e : T' \qquad T' <: T}{\Gamma \vdash (e : T) : T} \qquad \textsc{If}\quad \frac{\Gamma \vdash e_c : \text{bool} \qquad \Gamma \vdash e_t : T_t \qquad \Gamma \vdash e_f : T_f}{\Gamma \vdash \text{if } e_c \text{ then } e_t \text{ else } e_f : T_t \vee T_f} \qquad \textsc{SubBool}\quad \text{bool} <: \text{bool}$$

Figure 4.9. *Local definitions, upcasting, booleans and conditionals with subtyping*

Local definitions and type ascription adapt easily to subtyping by just replacing the requirements of equality to type annotations with subtyping (rules Let, LetRec and Upcast). As the change of name suggests type ascription is thus promoted into an **upcasting** construct. The typing of `let` is still consistent with that of a β-redex and upcasting with let $x : T = e$ in x.

The subtyping of `bool` is syntactic equality (SubBool), so the typing of boolean constants is also unaffected by subtyping and thus omitted here. The branch types T_t and T_f of a conditional no longer need to match exactly, because in a declarative system Sub could be used on both branches. So a conditional could have any type that is a common supertype of the branch types. Choosing their **join** $\vee$ which is the **least common supertype** retains the maximum possible amount of type information. Not all subtype relations have joins. This simple record subtyping relation does, but ML modules and thus 1ML and R1ML types do not.

4.3 System F

Parametrically polymorphic values like the polymorphic identity function `(fn x => x) : 'a -> 'a` abstract over types as well as values. ML type inference and type syntax make the type abstraction very implicit so the Java version is more illustrative:

```
class Fun {
    public static <T> T identity(T x) { return x; }
}
```

Listing 4.2. *Parametrically polymorphic identity function in Java*

Here `identity` has a type parameter T which is used as the return type and the type of the (value) parameter x. The type argument can be passed at callsites like this: `Fun.<String>identity("foo")`[3].

Parametric polymorphism and subtyping are two different forms of polymorphism. The unqualified word ‘polymorphism’ usually means parametric polymorphism in functional programming. In object-oriented programming ‘polymorphism’ is usually subtyping and parametric polymorphism is labeled ‘generics’ instead. Parametric polymorphism is useful even with subtyping available as can be seen from this version of the identity function with subtyping:

```
class Funk {
    public static Object identity(Object x) { return x; }
}
```

Listing 4.3. *Subtyping identity function in Java*

Both `Fun.identity` and `Funk.identity` can be used on an argument of any type but while `Fun.identity("foo")` returns a `String`, `Funk.identity("foo")` returns an `Object`. Of course `String` could be restored with a downcast in the latter case but that would risk `ClassCastException`s, postponing some type errors to runtime. Not to mention that in C++ erroneous static casts result in undefined behaviour.

Adding abstraction of terms over types to the simply typed lambda-calculus produces the parametrically polymorphic lambda-calculus known as **System F** [32, ch. 23]. The terms and types of System F are shown in Figure 4.10.

The System F syntax for abstracting terms over types is simply $\Lambda\alpha.e$, where α is the parameter **type variable**, which is in scope within the body of the Λ-abstraction just like the term parameter x is in scope within the body of a λ-abstraction. Again analogously the type parameter can be instantiated with an argument type via a Λ-application $e\,\langle T\rangle$.

[3]Although Java does infer type arguments so `Fun.identity("foo")` is preferrable in practice.

$$\begin{aligned} e &::= x \mid \lambda x : T.e \mid e\, e \mid \Lambda\alpha.e \mid e\,\langle T\rangle \\ T &::= T \to T \mid \forall\alpha\,.\,T \mid \alpha \end{aligned}$$

Figure 4.10. *Abstract syntax of System F*

Type parameters may be referred to in types. Λ-abstractions have **universal types** $\forall\alpha.T$ where the type variable α is in scope within the body T of the universal type. The idea is that a Λ-abstraction has the type of its body *for all types* α that may be passed to it as arguments.

In this thesis Λ-abstractions and universal types form a part of the implementation of functors. For when a functor domain signature has type members the functor must work *for all implementations* of those type members.

4.3.1 Type Checking

System F is quite straightforward to typecheck since parameter annotations are compulsory like in the simply typed lambda-calculus and all type abstractions and applications are explicit. The typing rules for Λ-abstractions and applications that are to be added to the simply typed lambda-calculus rules of Figure 4.2 are given in Figure 4.11.

$$\textsc{ΛAbs}\;\frac{\Gamma, \alpha \vdash e : T}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.T} \qquad \textsc{ΛApp}\;\frac{\Gamma \vdash e : \forall\alpha.T}{\Gamma \vdash e\,\langle T_a\rangle : [T_a/\alpha]\,T}$$

Figure 4.11. *Introduction and elimination rules for universal types*

ΛABS typechecks the body e in a type environment extended with the type parameter α and universally quantifies the result type T, capturing any references to α in T. The overall structure is analogous to ABS.

The type application rule ΛAPP typechecks the callee, which must have a universal type, being a type abstraction. Then it **instantiates** that type by substituting the argument type T_a for the type parameter α. This is more similar to the *evaluation* of term applications than their typing. Despite the substitution in ΛAPP typechecking is decidable for System F.

4.3.2 Existential Types

By analogy with mathematical logic[4] we would expect that existential quantifiers could also be used to form useful types and that values of such **existential types** would con-

[4]The analogies "propositions as types, proofs as programs" are a common theme in type theory called the **Curry-Howard correspondence**.

cern *some unknown type*. And an extended version of System F, on display in Figure 4.12, does include existential types [32, ch. 24].

$$\begin{aligned}&e ::= x \mid \lambda x : T.e \mid e\, e \mid \Lambda\alpha.e \mid e\,\langle T\rangle \mid \mathsf{pack}\langle T, e\rangle \text{ as } \exists\alpha.T_\alpha \mid \mathsf{unpack}\langle\alpha, x\rangle = e \text{ in } e\\ &T ::= T \to T \mid \forall\alpha\,.\,T \mid \exists\alpha\,.\,T \mid \alpha\end{aligned}$$

Figure 4.12. *Abstract syntax of System F with existentials*

An existential type $\exists\alpha.T$ hides some part of the implementation of T behind the type variable α. Existential types are introduced by `pack` which hides the type T in the type of e behind the existentially quantified type variable α. Existential types are eliminated by `unpack` which is similar to `let` but also deconstructs the existential type and brings into scope a type variable α for the abstract type in addition to the term variable x for the unpacked value. Note that `unpack` does not and could not reveal the implementation of α, it just allows use of the packaged value and brings the type variable into scope like the type parameter of a Λ is in scope within its body.

1ML uses existentials to model the type abstraction aspect of module sealing as might be guessed from the similarity of `pack` to sealing. R1ML does not implement sealing with existential types because forward references and recursion are not supported by `pack` and `unpack`. 1ML also uses existential types to implement the type generativity of functors and so does R1ML. [38]

The introduction and elimination rules for existential types are given in Figure 4.13.

$$\textsc{Pack}\quad \frac{\Gamma \vdash e : [T/\alpha]\,T_\alpha}{\Gamma \vdash \mathsf{pack}\,\langle T, e\rangle \text{ as } \exists\alpha.T_\alpha : \exists\alpha.T_\alpha} \qquad \textsc{Unpack}\quad \frac{\Gamma \vdash e_x : \exists\alpha.T_\alpha \qquad \Gamma, \alpha, x : T_\alpha \vdash e : T}{\Gamma \vdash \mathsf{unpack}\,\langle\alpha, x\rangle = e_x \text{ in } e : T}$$

Figure 4.13. *Introduction and elimination rules for existentials*

The introduction rule Pack checks that the type of the expression e to be packed is the instantiation of the existential body T_α with the implementation type T. Existential packing is essentially type ascription but in an explicitly typed system the implementation type of the existentially quantified type variable must be given explicitly.

The elimination rule Unpack checks that the unpackee e_x has existential type and then types the body e in a type environment extended with the type variable α and term variable x. The type variable must be pushed to the environment before the term variable since the type T_α of the term variable can refer to the type variable. Existential unpacking is essentially `let` with the addition of the type variable handling.

While the universal introduction ΛAbs extends the type environment and the universal elimination ΛApp instantiates the universal, existential introduction instantiates the existential and elimination extends the type environment. Such mirroring in universal and existential handling is also common in logic. In type theory one way to understand this is the Continuation Passing Style (CPS)-flavoured encoding of existentials with universals $\exists\alpha.T = \forall\beta.(\forall\alpha.T \to \beta) \to \beta$ and the contravariance of function domains.

4.3.3 Type Assignment

A universal type like $\forall\alpha.\alpha \to \alpha$ is *more general* than any of its instantiations such as $\mathsf{int} \to \mathsf{int}$. This notion of relative generality is captured by the System F **instance relation** $\sqsubseteq_F$ in Figure 4.14.

$$\forall\overline{\alpha}.T \sqsubseteq_F \forall\overline{\beta}.[\overline{T_{arg}/\alpha}]T$$

Figure 4.14. *The System F instance relation* $\sqsubseteq_F$

The implicitly typed variant of System F is based on this instance relation. Like subtyping $<:$ the instance relation $\sqsubseteq_F$ is a preorder on types. So it is possible to introduce it into the implicitly simply typed lambda-calculus via the rule INST of Figure 4.15, analogous to the subsumption rule SUB.

$$\textsc{Inst}\ \frac{\Gamma \vdash e : T' \qquad T' \sqsubseteq_F T}{\Gamma \vdash e : T} \qquad\qquad \textsc{Gen}\ \frac{\Gamma \vdash e : T \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash e : \forall\alpha.T}$$

Figure 4.15. *Instantiation and generalization typing rules*

The implicit $\forall$-elimination rule INST replaces explicit Λ-application of explicitly typed System F. Figure 4.15 also introduces the corresponding implicit $\forall$-introduction rule GEN. The side condition of GEN prevents the quantifier from capturing of type variables that were already scoped to somewhere in the type environment.

Since INST and GEN remove the need for explicit abstraction and application of terms over types, the terms of implicitly typed System F are the same as in the implicitly simply typed lambda-calculus. INST and GEN do use universal types, so the types of implicitly typed System F are the same as in explicitly typed System F.

The implicitly typed System F is undecidable. This because it lacks the **principal types** property, where every term has a most general type according to the $\sqsubseteq_F$ preorder. [2]

4.4 HM and ML^F

The Hindley-Milner (HM) type system [26] used in core ML is a restriction of implicitly typed System F that does have principal types. Its types in Figure 4.16 are stratified into quantifier-free **monotypes** τ and prenex-quantified **type schemes** σ.

$$\begin{aligned}\tau &::= \tau \to \tau \mid \text{bool} \mid \alpha \\ \sigma &::= \forall\alpha\,.\,\sigma \mid \tau\end{aligned}$$

Figure 4.16. *Hindley-Milner monotypes and type schemes*

The HM instance relation $\sqsubseteq_{HM}$ in Figure 4.17 is the restriction of $\sqsubseteq_F$ to the prenex polymorphism of HM. Since quantifiers are limited to the outermost level of type schemes, type variables can only be instantiated with monotypes ($\overline{[\tau_{arg}/\alpha]}$).

$$\forall\overline{\alpha}.\tau \sqsubseteq_{HM} \forall\overline{\beta}.\overline{[\tau_{arg}/\alpha]}\tau$$

Figure 4.17. *The prenex polymorphism instance relation* $\sqsubseteq_{HM}$

The full typing rules of the declarative HM type system are given in Figure 4.18. All the rules are shown because of the pervasive changes from a single class of types T to monotypes and type schemes.

$$\textsc{Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_d \vdash e : \tau_c}{\Gamma \vdash \lambda x.e : \tau_d \to \tau_c} \qquad \textsc{App}\ \frac{\Gamma \vdash e_f : \tau_d \to \tau_c \qquad \Gamma \vdash e_a : \tau_d}{\Gamma \vdash e_f\, e_a : \tau_c}$$

$$\textsc{Let}\ \frac{\Gamma \vdash e_x : \sigma \qquad \Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \text{let } x = e_x \text{ in } e : \tau} \qquad \textsc{Inst}\ \frac{\Gamma \vdash e : \sigma' \qquad \sigma' \sqsubseteq_{HM} \sigma}{\Gamma \vdash e : \sigma} \qquad \textsc{Gen}\ \frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma}$$

Figure 4.18. *Typing rules of the Hindley-Milner type system*

Aside from the change of type symbols the rules are very similar to implicitly typed System F. Types are restricted to monotypes except in VAR, LET and of course instantiation and generalization which are about quantifier handling.

In HM `let` is not just syntactic sugar for a β-redex and is in fact essential to the system. The semantics of `let`-bound variables is that a variable reference is well-typed if it could be replaced with the `let`-definiend e_x of the variable[5]. For this semantics to work `let`-bound variables must be allowed to have a type scheme in LET and in VAR (although VAR also handles references to λ-parameters). This is the so-called **let-polymorphism**.

It turns out that the HM type system can be made syntax-directed by integrating INST into VAR and GEN into LET, similar to how subtyping could be integrated into APP earlier. Type inference algorithms like Milner's original algorithm W combine the syntax-directed system with the use of **unification** [36].

[5]Of course in practice `let` is not implemented with naive macroexpansion, but with either type erasure (like Java generics) or monomorphisation (like C++ templates).

The limitations of prenex polymorphism led Le Botlan and Rémy to develop ML raised to the power of System F ($\mathrm{ML^F}$) [18], which achieves principal types by extending type schemes to be *more* expressive than System F types. The monotypes and type schemes of $\mathrm{ML^F}$ are shown in Figure 4.19.

$$\begin{aligned}\tau &::= \tau \to \tau \mid \mathsf{bool} \mid \alpha\\ \sigma &::= \forall\alpha \geqslant \sigma.\sigma \mid \forall\alpha = \sigma.\sigma \mid \bot \mid \tau\end{aligned}$$

Figure 4.19. *$\mathrm{ML^F}$ monotypes and type schemes*

The distinguishing feature of $\mathrm{ML^F}$ type schemes are the polymorphism-constrained quantifers $\forall\alpha \geqslant \sigma.\sigma$ and $\forall\alpha = \sigma.\sigma$. A flexibly ($\alpha \geqslant \sigma$) quantified type variable can only be instantiated by instances of the constraining type scheme σ. Flexible quantification is required to obtain principal types in cases where System F does not have them.

A rigidly ($\alpha = \sigma$) quantified type variable can only be instantiated with a type equal to the type scheme σ. Rigid quantification can be seen as a generalization of nested quantifiers in System F types, the naming of nested quantifiers enabling types that are directed acyclic graphs instead of just trees like in System F. Rigidly quantified types are used to incorporate type annotations into type inference. Rigid quantification contributes to the fact that $\mathrm{ML^F}$ only requires type annotations on function parameters that are used polymorphically.

The $\mathrm{ML^F}$ typing rules are very similar to those of HM, but the instance relation $\sqsubseteq_{MLF}$ is much more complicated than $\sqsubseteq_F$ or $\sqsubseteq_{HM}$. Overall the instance relation and especially the type inference algorithms of $\mathrm{ML^F}$ are too involved to explain here.

4.5 System F_η

HM has decidable type assignment, also known as “full type inference”. But ML module signatures like `STACK` with polymorphic members cannot be expressed with prenex polymorphism, which is one reason for the stratification to core ML and the module system.

$\mathrm{ML^F}$ types do not have the limitations of HM types. However $\mathrm{ML^F}$ does not support covariant or contravariant type constructors. The deep instantiation of co- and contravariance has not been missed in $\mathrm{ML^F}$ and has also recently been removed from GHC [42]. But depth subtyping is an essential ingredient of ML module systems, so modules cannot be implemented on top of $\mathrm{ML^F}$ either.

4.5.1 Quantifier Subtyping

HM restricts the expressiveness of System F types to make all type annotations unnecessary. $\mathrm{ML^F}$ increases the expressiveness of System F types and the power of the instance relation and requires very few type annotations. A third option is to keep the System F type language and extend the instance relation with covariance and contravariance and treat it as a subtyping relation [28] [31] [9] [40]. A variant of this approach is shown in Figure 4.20.

$$\textsc{SForall}\quad \frac{\Gamma, \overline{\beta} \vdash [\overline{T_{arg}/\alpha}]T_l <: T_r \rightsquigarrow f}{\Gamma \vdash \forall\overline{\alpha}.T_l <: \forall\overline{\beta}.T_r \rightsquigarrow \lambda x.\Lambda\overline{\beta}.f\,(x\,\langle\overline{T_{arg}}\rangle)}$$

$$\textsc{SArrow}\quad \frac{\Gamma \vdash T_d' <: T_d \rightsquigarrow f \qquad \Gamma \vdash T_c <: T_c' \rightsquigarrow g}{\Gamma \vdash T_d \to T_c <: T_d' \to T_c' \rightsquigarrow \lambda x.\lambda y.g\,(x\,(f\,y))} \qquad \textsc{SRigid}\quad \Gamma[\alpha] \vdash \alpha <: \alpha \rightsquigarrow \lambda x.x$$

Figure 4.20. *Quantifier subtyping*

The central rule in this **quantifier subtyping** relation is SFORALL, an extension of $\sqsubseteq_F$ with covariance for the body types T_l and T_r. Covariance allows further instantiation to happen in the body before the requantification with $\forall\overline{\beta}$. SARROW is the standard contravariant-covariant function subtyping rule. Rule SRIGID shows that subtyping is syntactic equality for type variable references (that are in scope, $\Gamma[\alpha]$).

In Figure 4.20 the subtyping check also generates a **coercion function** in the greyed out parts of the rules. The coercion function is in explicitly typed System F, but this thesis follows [40] and [38] in omitting type annotations from generated terms to reduce clutter. These **retyping functions** can only adjust quantifier structure via Λ-abstractions and -applications. SARROW can result in **η-expansion**, i.e. the generated coercion $\lambda x.\lambda y.xy$ contains an η-redex $\lambda y.xy$ of the function x. Similarly SFORALL can produce (Λ-)η-redexes $\Lambda\overline{\beta}.x\,\langle\overline{\beta}\rangle$.

All the generated retyping functions can be seen as generalizations of η-expansion, which is why systems with quantifier subtyping like this are called System F modulo η-expansion (System F_η). System F_η originated in [28] where quantifier subtyping was called **polymorphic containment**.

The emission of coercion functions means that System F_η uses a **coercion semantics** [32, sec. 15.6] for quantifier subtyping. In coercion semantics the operational meaning of $T <: T'$ is that values of type T can be coerced to type T' by some additional generated code like a call to a retyping function. This is in contrast to a **subset semantics** where $T <: T'$ means that values of type T have a runtime representation which is compatible with that of T' without any conversion code. For example Typescript uses a subset semantics for structural subtyping by transpiling to Javascript which (conceptually) looks fields up by name (instead of just an address offset like C structs).

A type system which transforms the program in addition to typechecking is an **elaborating** type system. The typing rules of System F_η have to transform the program by inserting calls to the generated coercion functions, creating many β-redexes in addition to the η-redexes. Both types of redexes can be optimized away by later compilation passes [1, ch. 6] or avoided altogether by refining the elaboration algorithm with techniques from one-pass CPS conversion [4].

4.5.2 Elaborating Expression Typing and Focalization

System F$_\eta$ cannot just use the typing rules from Section 4.2 with the quantifier subtyping relation as $<:$ because elaborating typing rules are needed to incorporate the coercion functions produced by quantifier subtyping. Figure 4.21 shows suitable elaborating rules.

$$\textsc{Var}\;\frac{x : T \in \Gamma}{\Gamma \vdash x : T \leadsto x} \qquad \textsc{Abs}\;\frac{\Gamma, x : T_d \vdash e : T_c \leadsto e'}{\Gamma \vdash \lambda x : T_d.e : T_d \to T_c \leadsto \lambda x : T_d.e'} \qquad \Lambda\textsc{abs}\;\frac{\Gamma, \alpha \vdash e : T \leadsto e'}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.T \leadsto \Lambda\alpha.e'}$$

$$\textsc{App}\;\frac{\begin{array}{c}\Gamma \vdash e_f : T_f \leadsto e'_f \qquad T_f \gg \_ \to \_ \leadsto T_d \to T_c \leadsto f \\ \Gamma \vdash e_a : T_a \leadsto e'_a \qquad \Gamma \vdash T_a <: T_d \leadsto g\end{array}}{\Gamma \vdash e_f\, e_a : T_c \leadsto (f\, e'_f)\,(g\, e'_a)}$$

Figure 4.21. *System F$_\eta$ typing rules*

Var just produces a variable reference. Abs and Λabs just elaborate the function body and wrap it in their abstraction forms.

Quantifiers cause an additional challenge for elimination forms like function application; values with types that can be instantiated to a function type like $\forall\alpha.\alpha \to \alpha$ should be callable as well as those that already have a function type. Subtyping cannot be used for this because no specific function type is available to act as the supertype. So we introduce an auxiliary **focalization** judgment [9] [38] that takes a type template $\underline{T}$ that is like a type but can contain holes _. Like subtyping, focalization produces a coercion function. App types the callee e_f and focalizes its type T_f to any function type _ $\to$ _. The argument type is subtyped with the domain type as in Section 4.2. The elaborated application coerces the callee with the focalization coercion and the argument with the subtyping coercion before the expected application.

Focalization itself is based on instantiation like quantifier subtyping. Focalization is defined in Figure 4.22.

$$\gg\textsc{forall}\;\frac{[\overline{T_{arg}/\alpha}]T \gg \underline{T} \leadsto T' \leadsto f}{\forall\overline{\alpha}.T \gg \underline{T} \leadsto T' \leadsto \lambda x.f\,(x\,\langle\overline{T_{arg}}\rangle)} \qquad \gg\textsc{arrow}\;\; T_d \to T_c \gg \_ \to \_ \leadsto T_d \to T_c \leadsto \lambda x.x$$

Figure 4.22. *Focalization*

Rule $\gg$forall instantiates universal types, producing a coercion function with the corresponding Λ applications. Rule $\gg$arrow just checks that when the template is _ $\to$ _

(the only template used in this section) the focalizee is of arrow type.

A system like this infers all Λ-applications. Quantifier subtyping can also infer quite a few Λ-abstractions but Λ must be used to create polymorphism (via ΛABS). Enhancing the system with bidirectional typing [9] allows replacing Λ with polymorphic type annotations but even that system does not have the power of GEN to introduce polymorphism with no hints from the programmer. It has been argued that inference of applications is more important than inference of Λ-abstractions, which are far fewer in number and also benefit from type annotations as documentation [33]. More recently it has been shown that the vast majority of parametrically polymorphic definitions are at top level where they are very likely to be annotated in any case [45].

A more serious weakness of this system is that it is undecidable. The **predicative** variant which limits instantiation of type variables to monotypes τ is decidable. That variant is obtained by just replacing $[\overline{T_{arg}/\alpha}]$ with $[\overline{\tau_{arg}/\alpha}]$ (and $\overline{T_{arg}}$ with $\overline{\tau_{arg}}$).

Impredicative instantiation is rarely needed, so predicativity is not as limiting as it seems. On the other hand the rarity of impredicativity might just be additional evidence for the Sapir-Whorf hypothesis in programming languages, as Remy argues in [34] that impredicativity is essential. After all, people are quite satisfied with the even more limiting prenex polymorphism of HM even though “These cases are not all that common, but there are usually no workarounds; if you need higher-rank types, you really need them!” [31].

4.6 System F$_\omega$

The types of types are called **kinds** [32, ch. 29]. The systems discussed up to this point have been “dynamically kinded” or implicitly mono-kinded, all types having the kind `Type` and kind annotations omitted (because they would all be `Type`).

Nontrivial kinding is orthogonal to quantified types. However it is more practical to discuss System F$_\omega$ [32, ch. 30], which extends System F with nontrivial kinding.

The abstract grammar of System F$_\omega$ types in Figure 4.23 is a straightforward extension of System F with type level functions $\lambda\alpha : \kappa.T$ and applications $T\,T$ as well as kind annotations for quantifiers. Kinds κ are similar to the types of simply typed lambda-calculus; the atomic kind type for types and an arrow kind type $\rightarrow$ type for type level functions. Only types of kind type classify values.

$$\begin{aligned} e &::= x \mid \lambda x : T.e \mid e\,e \mid \Lambda\alpha : \kappa.e \mid e\,\langle T\rangle \\ T &::= T \rightarrow T \mid \forall\alpha : \kappa\,.\,T \mid \lambda\alpha : \kappa.T \mid T\,T \mid \alpha \\ \kappa &::= \kappa \rightarrow \kappa \mid \mathsf{type} \end{aligned}$$

Figure 4.23. *Abstract syntax of System F$_\omega$*

We are primarily interested in parameterized types like

```
type 'a point = {x : 'a, y : 'a}
```

whose Java equivalent[6] would be

```
class Point<T> {public T x; public T y;}
```

Those correspond to (second class) type level functions. Type level functions are also called **higher kinded types**. Other nontrivial kinds are useful in type level programming just like types are useful in regular term-level programming. Like SML and OCaml we will not support such advanced type level techniques.

Since only types of kind type classify values it is unsurprising that the impact of nontrivial kinding on the typing rules is minor. Figure 4.24 shows the required changes.

$$\textsc{Abs}\quad \frac{\Gamma \vdash T_d : \mathsf{type} \qquad \Gamma, x : T_d \vdash e : T_c}{\Gamma \vdash (\lambda x : T_d . e) : T_d \to T_c} \qquad\qquad \Lambda\textsc{Abs}\quad \frac{\Gamma, \alpha{:}\ \kappa \vdash e : T}{\Gamma \vdash \Lambda\alpha{:}\ \kappa . e : \forall\alpha{:}\ \kappa . T}$$

$$\Lambda\textsc{App}\quad \frac{\Gamma \vdash e : \forall\alpha{:}\ \kappa . T \qquad \Gamma \vdash T_a : \kappa}{\Gamma \vdash e\,\langle T_a\rangle : [T_a/\alpha]\,T} \qquad\qquad \textsc{Eq}\quad \frac{\Gamma \vdash e : T' \qquad T' \equiv T \qquad \Gamma \vdash T : \mathsf{type}}{\Gamma \vdash e : T}$$

Figure 4.24. *System F_ω changes from System F*

Quantifier rules are refined by adding kinds for type variables and well-kindedness checks $\Gamma \vdash T : \kappa$ for Λ applications. The addition of type level functions means that type equality is no longer simply syntactic. This is expressed with an explicit type equality relation $\equiv$ and a non-syntax-directed rule EQ, which is analogous to the subsumption rule in declarative subtyping.

At this point the impact of nontrivial kinding seems rather small. However we also need to define the details of well-kindedness $\Gamma \vdash T : \kappa$ and nonsyntactic type equality $\equiv$.

Since the new type expressions are just the simply typed lambda-calculus promoted to the level of types, the kinding rules in Figure 4.25 should be quite familiar by now. The kinding of function types and quantifiers is new but even simpler than the promoted rules.

The domain and codomain of functions must be of kind type and the combined function type also has kind type. In fact rule TARROW is redundant if we just consider $\to$ as an infix type operator of kind type $\to$ type. Some complication might be expected in the kinding of quantified types, but they just extend the type environment with their kinded type variable and have the kind that their body type has that extended environment.

As an equivalence relation $\equiv$ must be reflexive, symmetric and transitive. These properties are taken as definitions in Figure 4.26.

[6]Ignoring the curiosity that for historical reasons `Point` is a supertype of e.g. `Point<Integer>` in Java.

TVar
$$\frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa}$$

TAbs
$$\frac{\Gamma, \alpha : \kappa_d \vdash T : \kappa_c}{\Gamma \vdash (\lambda\alpha : \kappa_d.T) : \kappa_d \to \kappa_c}$$

TApp
$$\frac{\Gamma \vdash T_f : \kappa_d \to \kappa_c \qquad \Gamma \vdash T_a : \kappa_d}{\Gamma \vdash T_f\, T_a : \kappa_c}$$

TForall
$$\frac{\Gamma, \alpha : \kappa_\alpha \vdash T : \kappa}{\Gamma \vdash (\forall\alpha : \kappa_\alpha.T) : \kappa}$$

TArrow
$$\frac{\Gamma \vdash T_d : \mathsf{type} \qquad \Gamma \vdash T_c : \mathsf{type}}{\Gamma \vdash T_d \to T_c : \mathsf{type}}$$

Figure 4.25. *Kinding rules for System F$_\omega$*

EqRefl
$$T \equiv T$$

EqSymm
$$\frac{T' \equiv T}{T \equiv T'}$$

EqTrans
$$\frac{T \equiv T'' \qquad T'' \equiv T'}{T \equiv T'}$$

EqForall
$$\frac{T \equiv T'}{\forall\alpha : \kappa.T \equiv \forall\alpha : \kappa.T'}$$

EqArrow
$$\frac{T_d \equiv T_d' \qquad T_c \equiv T_c'}{T_d \to T_d \equiv T_d' \to T_c'}$$

EqFn
$$\frac{T \equiv T'}{\lambda\alpha : \kappa.T \equiv \lambda\alpha : \kappa.T'}$$

Eqβ
$$(\lambda\alpha : \kappa.T)\, T' \equiv [T'/\alpha]T$$

EqApp
$$\frac{T_f \equiv T_f' \qquad T_a \equiv T_a'}{T_f\, T_a \equiv T_f'\, T_a'}$$

Figure 4.26. *Type equivalence in System F$_\omega$*

The equivalence of both quantifiers and also type level functions follows the same formula: the types are equal if the kinds of the quantified variables are (syntactically) equivalent and if the body types are equivalent. It would seem that the names of the quantified variables would also need to be identical, but recall the Barendregt naming convention: the rules assume that the variables can be implicitly renamed to match. A practical type-checker would need to contain explicit code for this renaming or use a nameless representation like de Bruijn indices [32, ch. 6].

Rule Eβ is the real complication of type level functions; a β-redex $(\lambda\alpha : \kappa.T)\, T'$ is equal to its reduction $[T'/\alpha]T$, introducing type level computation. Equality is also recursively defined on any application expression (EApp) including the special case of function types (EArrow), which might seem redundant if function applications can be evaluated with Eβ. However applications of abstract type constructors like `list` or $\to$ actually cannot be reduced further, just like in SML `SOME 5` is a tagged value instead of a β-redex. Furthermore it should also be true that $\forall\alpha : \mathsf{type} \to \mathsf{type}.\alpha\,\mathsf{int} \equiv \forall\alpha : \mathsf{type} \to \mathsf{type}.\alpha\,\mathsf{int}$, where α stands for all types of kind $\mathsf{type} \to \mathsf{type}$.

4.6.1 Decidability

Type checking is decidable for explicitly typed System F$_\omega$. The non-syntax-directed rule Eq can be factored out, similar to Sub in Section 4.2. Like the simply typed lambda-calculus, kinding is syntax-directed and inductive so it is decidable. Type equivalence $\equiv$ is also decidable by separating evaluation from equality checking as in Section 7.3.2. Type level reduction is strongly normalizing, like the simply typed lambda-calculus. [32, ch. 30]

The type assignment variant of System F_ω is undecidable because it includes the already undecidable type assignment variant of System F. Furthermore the addition of type level functions requires higher order unification, which is also undecidable.

In predicative System F_η subtyping problems like

$$\{x : \mathsf{bool}\} <: \exists\alpha : \mathsf{type}.\{x : \alpha\}$$

could be solved straightforwardly with the substitution $[\mathsf{bool}/\alpha]$. But in F_ω we would encounter situations like

$$\{x : \mathsf{bool}\} <: \exists\alpha : \mathsf{type} \to \mathsf{type}.\{x : \alpha\,\mathsf{bool}\}$$

which could be solved with the substitution $[\lambda\beta : \mathsf{type}.\beta/\alpha]$ but also by $[\lambda\beta : \mathsf{type}.\mathsf{bool}/\alpha]$. In fact there are an infinitely many suitable type level functions, hence the undecidability of higher-order unification.

In ML higher kinded types must always be fully applied, making them second class (even on the type level). This restriction removes the requirement for higher order unification, keeping type inference decidable.

Haskell must have some support for higher kinded types since it is required by the essential `Monad` type class among others. Their solution is to limit higher kinded type terms to abstract types created by `data` and `newtype` declarations and partial applications thereof. Transparent `type` definitions are still second class. The limited higher kind type support of Haskell permits rather sophisticated constructions like the infamous monad transformers, but is quite unprincipled and leads to a proliferation of `newtype` wrapper boilerplate. It could not be reasonably combined with the translucency in ML modules (which Haskell does not have).

As we shall see later on in great detail, the subset of quantifier subtyping problems elaborated from ML modules is in fact decidable. We can already see why on the level of the surface language. A sealing problem

```
{type 'a t = 'a; x = true} :> {type 'a t; x : bool t}
```

can obviously only be solved with $\mathtt{t} = \lambda a.a$ since higher kinded types are only matched at `type` declarations. Furthermore without forward reference support in signatures the `type` declaration will always come up before any of its uses. First class modules and forward references complicate the situation somewhat but translucency remains workable.

5 F-ING MODULES AND 1ML

Altogether, we describe a comprehensive, unified, and yet simple semantics of a full-blown module language that – with the main exception of cross-module recursion – covers almost all interesting features that can be found in either the literature or in practical implementations of ML modules.

F-ing modules [40]

ML modules were first conceived as an application of dependent types. Nevertheless ML modules have been modeled with System F$_\omega$ encodings in papers for a long time. The “F-ing Modules” paper [40] finally introduced a full implementable module type checker which elaborates to System F$_\omega$. The follow-up 1ML paper [38] continued by replacing the stratification into module and core languages with a predicativity restriction.

In this chapter we review the main points of the ‘F-ing’ approach and 1ML in particular. To avoid introducing too much at once this chapter explains the explicitly typed 1ML variant. The abstract syntax of 1ML is repeated in Figure 5.1. The full surface syntax also includes various derived constructs to be desugared into these.

$$
\begin{aligned}
E ::=\ & X \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E : T \mid \{B\} \mid E.X \mid \mathsf{fun}\ (X : T) \Rightarrow E \mid X\ X \\
& \mid \mathsf{type}\ T \mid X :> T \\
B ::=\ & X = E \mid \mathsf{include}\ E \mid B; B \mid \epsilon \\
T ::=\ & E \mid \mathsf{bool} \mid \{D\} \mid (X : T) \Rightarrow T \mid (X : T) \rightarrow T \mid \mathsf{type} \mid = E \mid T\ \mathsf{where}\ (\overline{.X} : T) \\
D ::=\ & X : T \mid \mathsf{include}\ T \mid D; D \mid \epsilon
\end{aligned}
$$

Figure 5.1. *Abstract (desugared) syntax of 1ML*

Record expressions $\{B\}$ cover structure modules in addition to basic records. Bindings B are semicolon-connected sequences of member definitions $X = E$ and inheritance with include. Like `open` in SML, include E both copies the members of E into the record being built and brings them into scope as variables for the following bindings. As with both simple record types and structure modules the record types $\{D\}$ and declarations D parallel the record syntax.

The type T expression creates a type proxy value which carries no information at runtime but whose type includes the type T. On the type level `type` creates abstract types and is not tied to declarations D unlike in SML anc OCaml.

Type paths are not syntactically limited but can be any pure expression E of type `type`. The singleton type $= E$ is equal to the type of the expression E including the identity of abstract types.

Members of (nested) record types can be refined with `where`, which is a generalization of `where type` in SML[1]. A combination of `where` and a singleton type can emulate the `where structure` compiler extension of the Standard ML of New Jersey (SML/NJ) compiler.

The `fun` keyword starts function literals instead of function definitions like in SML. Functions can have both pure ($\Rightarrow$) and impure ($\rightarrow$) types. Functions subsume functors so the domain can be named in function types to let the codomain refer to the type components of the domain. Impure functors are generative and pure functors applicative.

Due to the presence of higher-kinded types ML module signatures and 1ML do not have joins. So in explicitly typed 1ML all conditionals must have a type signature to which both branches are upcasted (sealed). When type inference is added the type annotation is not required when both branches have monotypes.

Various subexpressions are limited to variables X to simplify typechecking. 1ML uses a kind of **A-normal form** conversion [11] before typechecking to lift these restrictions from source code programs.

5.1 Semantic Types

The types used in the 1ML type systems are a stylized subset of System F$_\omega$. That subset is shown in Figure 5.2.

$$
\begin{aligned}
\Xi &::= \exists\overline{\alpha}.\Sigma \\
\Sigma &::= \pi \mid \mathsf{bool} \mid [= \Xi] \mid \{\overline{l : \Sigma}\} \mid \forall\overline{\alpha}.\Sigma \rightarrow_\iota \Xi \\
\tau &::= \pi \mid \mathsf{bool} \mid [= \tau] \mid \{\overline{l : \tau}\} \mid \tau \rightarrow_\mathsf{I} \tau \\
\pi &::= \alpha \mid \pi\,\overline{\tau} \\
\iota &::= \mathsf{P} \mid \mathsf{I}
\end{aligned}
$$

Figure 5.2. *Semantic types of 1ML*

The base types are the large types Σ (Σ for module **s**ignature). Monotypes τ, called **small types** in 1ML have the same structure as large types but cannot contain existential or universal quantifiers. Here τ is used instead of the σ (lowercase or *small* Σ) in [38] for consistency with the rest of this thesis.

Type proxies (from `type` T) have carrier types $[= \Xi]$. Abstract types (from `type`) become

[1] `with type` in OCaml

paths π, which are either rigid type variables α or their (nested) applications to monotypes. Abstracted types Ξ can be existentially quantified and only appear in type proxies and function codomains.

Function types $\forall\bar{\alpha}.\Sigma \rightarrow_{\iota} \Xi$ have the usual arrow with domain and codomain types. Universal quantifiers are only needed in conjunction with function types, for abstract types in functor domain types. Function types also integrate an effect ι as a subscript to the arrow. The arrow effect is the effect resulting from calls to the function, either pure P (corresponding to $\Rightarrow$) or impure I (corresponding to $\rightarrow$). Monotyped ('non-functor') functions always have the impure effect to simplify type inference since parametric polymorphism is not available for effects.

5.2 Effects

Figure 5.3 defines effect subtyping and join. These rules are inevitable given the meaning of P and I.

ESREFL	ESIMPURIFY	EJREFL	EJIMPURIFYL	EJIMPURIFYR
$\iota <: \iota$	$\mathsf{P} <: \mathsf{I}$	$\iota \vee \iota = \iota$	$\mathsf{P} \vee \mathsf{I} = \mathsf{I}$	$\mathsf{I} \vee \mathsf{P} = \mathsf{I}$

Figure 5.3. *Effect subtyping and join*

Either effect is a subtype of itself, making effect subtyping reflexive (rule ESREFL). The other subtyping rule ESIMPURIFY makes P a subtype of I; when side effects are allowed, they are not compulsory. The only other combination $\mathsf{I} <: \mathsf{P}$ would make both effects equivalent and typing unsound.

The join of either effect with itself is that effect (EJREFL). Otherwise (EJIMPURIFYL and EJIMPURIFYR) the result is I because I is not a subtype of P as just discussed.

5.3 Structures

Figure 5.4 shows that 1ML essentially processes structure members as sequential variable bindings while also collecting them into a record. The effect ι of each subexpression is inferred in addition to the type, mostly by combining effects with the effect join. Since the initializing expressions of the members can create abstract types the rules also have a further unconventional aspect.

The existential parameters (if any) of the member type are inserted into the typing context along with the member name typed as the body of the existential. This enables subsequent members to refer to values involving those abstract types. For the type level unpacking of the existential to make sense with relation to the elaborated program an existential unpack is added to the generated code. To have a composable and tractable system the abstract types have to be re-packed into an existential that is returned. This

ESTR

$$\frac{\Gamma \vdash B :_\iota \Xi \rightsquigarrow e}{\Gamma \vdash \{\overline{B}\} :_\iota \Xi \rightsquigarrow e}$$

BVAR

$$\frac{\Gamma \vdash E :_\iota \exists\overline{\alpha}\,.\,\Sigma \rightsquigarrow e}{\Gamma \vdash X = E :_\iota \exists\overline{\alpha}\,.\,\{X : \Sigma\} \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in\ pack}\ \langle\overline{\alpha}, \{X = x\}\rangle}$$

BSEQ

$$\frac{\begin{array}{c}\Gamma \vdash B_1 :_{\iota_1} \exists\overline{\alpha_1}\,.\,\{\overline{X_1 : \Sigma_1}\} \rightsquigarrow e_1 \\ \Gamma, \overline{\alpha_1}, \overline{X_1 : \Sigma_1} \vdash B_2 :_{\iota_2} \exists\overline{\alpha_2}\,.\,\{\overline{X_2 : \Sigma_2}\} \rightsquigarrow e_2 \qquad \overline{X_1'} = \overline{X_1} - \overline{X_2} \qquad \overline{X_1' : \Sigma_1'} \subseteq \overline{X_1 : \Sigma_1}\end{array}}{\begin{array}{c}\Gamma \vdash B_1; B_2 :_{\iota_1 \vee \iota_2} \exists\overline{\alpha_2\alpha_2}\,.\,\{\overline{X_1' : \Sigma_1'}, \overline{X_2 : \Sigma_2}\} \\ \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha_1}, y_1\rangle = e_1\ \mathsf{in\ let}\ \overline{X_1 = y_1.X_1} \\ \mathsf{in\ unpack}\ \langle\overline{\alpha_1}, y_1\rangle = e_2 \\ \mathsf{in\ pack}\langle\overline{\alpha_1\alpha_1}, \{\overline{X_1' = y_1.X_1'}, \overline{X_2 = y_2.X_2}\}\rangle\end{array}}$$

BEMPTY

$$\Gamma \vdash \epsilon :_{\mathsf{P}} \{\} \rightsquigarrow \{\}$$

Figure 5.4. *1ML structures (excerpt)*

too needs to be accompanied by a value-level pack in the generated code. All this unpacking and re-packing maintains the invariant that existential quantifiers only occur at the outermost level of a type (or as the codomain of a function or inside a type carrier type $[= \exists\bar{\alpha}\,.\,\Sigma]$). The papers call this a 'monad of abstract type generation' which does provide the right intuition if one happens to be familiar with monads.

The set operations in rule BSEQ just ensure that the generated record will not have duplicated fields. The module expression can nevertheless employ name shadowing as in a sequential `let` with the additional consequence that the last value for a name is also collected into the record.

The 1ML structure signature elaboration rules are very similar to the analogous structure typing rules as one can see in Figure 5.5. Value and type variables are added to the context and existentials packed and unpacked exactly as in the module rules, just without the complications of generating the accompanying elaborated code.

TSTR

$$\frac{\Gamma \vdash D \rightsquigarrow \Xi}{\Gamma \vdash \{D\} \rightsquigarrow \Xi}$$

DVAR

$$\frac{\Gamma \vdash T \rightsquigarrow \exists\overline{\alpha}\,.\,\Sigma}{\Gamma \vdash X : T \rightsquigarrow \exists\overline{\alpha}\,.\,\{X : \Sigma\}}$$

DSEQ

$$\frac{\begin{array}{c}\Gamma \vdash D_1 \rightsquigarrow \exists\overline{\alpha_1}\,.\,\{\overline{X_1 : \Sigma_1}\} \\ \Gamma, \overline{\alpha_1}, \overline{X_1 : \Sigma_1} \vdash D_2 \rightsquigarrow \exists\overline{\alpha_2}\,.\,\{\overline{X_2 : \Sigma_2}\} \qquad \overline{X_1} \cap \overline{X_2} = \emptyset\end{array}}{\Gamma \vdash D_1; D_2 \rightsquigarrow \exists\overline{\alpha_1\alpha_2}\,.\,\{\overline{X_1 : \Sigma_1}, \overline{X_2 : \Sigma_2}\}}$$

DEMPTY

$$\Gamma \vdash \epsilon \rightsquigarrow \{\}$$

Figure 5.5. *1ML structure signatures (excerpt)*

It is perhaps surprising to see value bindings in type elaboration rules but this is how ML structure signatures are supposed to work. Such behaviour is characteristic of dependent typing but here it is implemented without dependent types.

The empty intersection assertion in rule DSEQ is almost analogous to the set operations in BSEQ. However it prohibits not only duplicate fields in the output type but also name shadowing, which is the usual ML signature behaviour even though it is inconsistent with BSEQ.

The use of existential packages in the output of these rules is the ‘F-ing Modules’ explanation for the fact that ML modules do not allow forward references. Abstract types (the parameters of $\exists$-quantifiers) are created at an `unpack` and are only in scope after that point. Other type definitions and even the types of values can and often do refer to these abstract types so they too might be invalid before the `unpack`. For consistent behaviour all forward references have been banned, even ones that do not refer to abstract types.

5.4 Type Members

Figure 5.6 shows the handling of type members. This is where the existentials that need to be constantly unpacked and packed originate.

$$\textsc{EType}\quad \frac{\Gamma \vdash T \rightsquigarrow \Xi}{\Gamma \vdash \mathsf{type}\, T :_{\mathsf{P}} [= \Xi] \rightsquigarrow [\Xi]} \qquad\qquad \textsc{TType}\quad \Gamma \vdash \mathsf{type} \rightsquigarrow \exists\alpha\,.\,[= \alpha]$$

Figure 5.6. *1ML first-class types (excerpt)*

Concrete types are represented as type carrier values $[= \Xi]$. Concrete type expressions yield carriers that just contain the elaborated type (rule ETYPE). Type members are transmitted between modules via these carrier values that serve no other purpose and could thus be elaborated further into e.g. the empty record `{}` by subsequent compilation passes. As mentioned earlier this is one of the few places where an existential quantifier can exist nested within a type.

The type `type` elaborates into a carrier type containing an existentially quantified type variable per rule TTYPE. This is the only place where existential quantifiers are created instead of merely being expanded and combined in the module and signature rules.

5.5 Sealing

Figure 5.7 shows the 1ML sealing rule. The rule states that only variables can be sealed, but the preprocessing phase mentioned earlier removes this restriction from the source-level syntax.

Sealing itself is surprisingly simple. The type of the expression is determined and the signature elaborated. Then the expression is just tested to be a subtype of the elaborated signature. The subtyping determines the implementations of the abstract type as the substitution δ and computes a value-level coercion function f, which are the used to

ESEAL

$$\frac{\Gamma \vdash X :_{\mathsf{P}} \Sigma_1 \rightsquigarrow e \qquad \Gamma \vdash T \rightsquigarrow \exists \overline{\alpha} \,.\, \Sigma_2 \qquad \Gamma \vdash \Sigma_1 <:_{\overline{\alpha}} \Sigma_2 \rightsquigarrow \delta; f}{\Gamma \vdash X :> T :_{\iota(\exists \overline{\alpha} \,.\, \Sigma_2)} \rightsquigarrow \mathsf{pack}\, \langle \delta \overline{\alpha}, f\, e \rangle}$$

Figure 5.7. *1ML sealing*

coerce and pack the value of the expression to the signature type. All this is familiar although the notation used here is slightly different from the rest of this thesis.

The notation $\iota(\exists \overline{\alpha} \,.\, \Sigma_2)$ states that the effect of the sealing is impure if the signature had any abstract types $\bar{\alpha}$ and pure otherwise (since a variable reference is always pure). Sealing does not have any runtime side-effects but treating existential creation as a side effect simplifies the handling of functor generativity in the next section. The “F-ing” and 1ML papers also describe more sophisticated language variants where sealing is pure.

5.6 Functors

The 1ML rules for functor (and function) types and values are shown in Figure 5.8. Applicative functors are also called pure because their bodies must be free of side effects. An impure applicative functor could cause breaches of abstraction as can happen in OCaml.

TFUN

$$\frac{\Gamma \vdash T_1 \rightsquigarrow \exists \overline{\alpha_1} \,.\, \Sigma_1 \qquad \Gamma, \overline{\alpha_1}, X : \Sigma_1 \vdash T_2 \rightsquigarrow \exists \overline{\alpha_2} \,.\, \Sigma_2}{\Gamma \vdash ((X : T_1) \to T_2 \rightsquigarrow \forall \overline{\alpha_1} \,.\, \Sigma_1 \to_{\mathsf{I}} \exists \overline{\alpha_2} \,.\, \Sigma_2}$$

TPFUN

$$\frac{\Gamma \vdash T_1 \rightsquigarrow \exists \overline{\alpha_1} \,.\, \Sigma_1 \qquad \Gamma, \overline{\alpha_1}, X : \Sigma_1 \vdash T_2 \rightsquigarrow \exists \overline{\alpha_2} \,.\, \Sigma_2 \qquad \overline{\alpha_2' : \overline{\kappa_{\alpha_1}} \to \kappa_{\alpha_2}}}{\Gamma \vdash ((X : T_1) \Rightarrow T_2 \rightsquigarrow \exists \overline{\alpha_2'} \,.\, \forall \overline{\alpha_1} \,.\, \Sigma_1 \to_{\mathsf{P}} [\overline{\alpha_2' \overline{\alpha_1} / \alpha_2}]\Sigma_2}$$

EFUN

$$\frac{\Gamma \vdash T \rightsquigarrow \exists \overline{\alpha} \,.\, \Sigma \qquad \Gamma, \overline{\alpha}, X : \Sigma \vdash E :_{\iota} \Xi \rightsquigarrow e}{\Gamma \vdash \mathsf{fun}\,(X : T) \Rightarrow E :_{\mathsf{P}} \forall \overline{\alpha} \,.\, \Sigma \to_{\iota} \Xi \rightsquigarrow \Lambda \overline{\alpha} \,.\, \lambda X : \Sigma \,.\, e}$$

EAPP

$$\frac{\Gamma \vdash X_1 :_{\mathsf{P}} \forall \overline{\alpha} \,.\, \Sigma_1 \to_{\iota} \Xi \rightsquigarrow e_1 \qquad \Gamma \vdash X_2 :_{\mathsf{P}} \Sigma_2 \rightsquigarrow e_2 \qquad \Gamma \vdash \Sigma_2 <:_{\overline{\alpha}} \Sigma_1 \rightsquigarrow \delta; f}{\Gamma \vdash X_1\, X_2 :_{\iota} \delta \Xi \rightsquigarrow e_1[\delta \overline{\alpha}](f\, e_2)}$$

Figure 5.8. *1ML functors (excerpt)*

Rules TFUN and TPFUN elaborate impure (generative) and pure (applicative) functors. The name X and abstract types $\overline{\alpha_1}$ of the domain are added to the typing context so that they are visible while elaborating the codomain type. Again this seems like dependent typing but is actually just a way to use quantifiers to enable flexible abstract types in modules.

The abstract types $\overline{\alpha_1}$ are requantified with a universal quantifier that contains their entire scope, the domain and the codomain. The functor will work for all implementations of the domain's abstract types, not just some hidden one so the quantifier must be changed from existential to universal. Changing the quantifier also fits in with the contravariance of function domain types.

As their output expressions suggest, EFUN and EAPP can be seen as combinations of previous λ- and Λ-abstraction and application rules respectively. Since sealing is impure in this simple 1ML variant the body of a pure functor cannot produce abstract types. The argument types of a functor are inferred as part of domain subtyping.

Impure functors just insert the elaborated codomain type $\exists\overline{\alpha_2}\,.\,\Sigma_2$ as the codomain of the elaborated functor type. This leads to the application expressions of such functors having an existential type like a `pack` expression. So the application expression generates abstract types. In other words, an impure functor behaves generatively.

Pure functors lift the abstract types into the outermost layer of the functor type, so that applications of pure functors will not have existential types and will not generate fresh abstract types. However the implementation types of the abstract codomain types $\overline{\alpha_2}$ can depend on the abstract domain types $\overline{\alpha_1}$ and each application of a pure functor where the abstract domain types get implemented differently should result in different abstract codomain types. This is achieved by parameterizing the abstract codomain types with the abstract domain types and replacing their usages in the codomain type with the type-level applications $\overline{\alpha_2'\overline{\alpha_1}}$. Now each application of a pure functor to the same argument will produce the same types but applying to a different argument will produce different types, so a pure functor behaves applicatively. This implementation of applicative functors does have the issue that argument identity is only tied to the domain type members and ignoring the identity of the values can result in breaches of abstraction even without runtime side effects. That hole was plugged with a more sophisticated system in [40], but with first-class modules their solution would seem to require impredicative instantiation.

In 1ML and R1ML applicative functors are essential since functors subsume type constructors which must have applicative behaviour to be usable: `list int` should be equal to all other `list int` type expressions. On the other hand a system that merges functors with functions also needs generative functors because ML functions can have side effects while abstraction-safe applicative functors require referential transparency.

6 SYSTEM F$_C$

While our motivating examples were GADTs and associated types, we believe that F$_c$ may have much wider application as a typed target for sophisticated HOT (higher-order typed) source languages.

System F with Type Equality Coercions [44]

The R1ML type system uses System F$_c$ [44] as its elaboration target. Naturally the needs of R1ML are somewhat different from those of GHC, so the System F$_c$ of this chapter is an extended subset of the one in [44]. R1ML is focused on abstract types and solving double vision and does not need as much coercion type machinery as GHC. On the other hand supporting the composable type abstraction of ML modules requires the addition or generalization of some constructs. The abstract syntax of our System F$_c$ variant is shown in Figure 6.1.

$$\begin{aligned}
e ::= {} & x \mid \lambda x : T\,.\,e \mid e\,e \mid \Lambda\alpha : \kappa\,.\,e \mid e\,\langle T\rangle \mid \mathsf{pack}\ \langle T, e\rangle\ \mathsf{as}\ T \mid \mathsf{unpack}\ \langle \alpha, x\rangle = e\ \mathsf{in}\ e \\
& \mid \mathsf{let\ \ rec}\ \overline{x : T = e}\ \mathsf{in}\ e \mid \mathsf{type}\ \alpha : \kappa\ \mathsf{in}\ e \mid \mathsf{axiom}\ c : T \sim T\ \mathsf{in}\ e \mid e \blacktriangleright \gamma \\
& \mid \{\overline{x = e}\} \mid e.x \mid [T] \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
T ::= {} & T \to T \mid \forall\alpha : \kappa\,.\,T \mid \exists\alpha : \kappa\,.\,T \mid T\,T \mid \{\overline{x : T}\} \mid [= T] \mid \mathsf{bool} \mid \alpha \\
\kappa ::= {} & \kappa \to \kappa \mid \mathsf{type} \\
\gamma ::= {} & T \mid \mathsf{sym}\ \gamma \mid \gamma \circ \gamma \mid c \mid \gamma @ T \mid \gamma\,\gamma \mid [= \gamma] \mid \{\overline{x : \gamma}\}
\end{aligned}$$

Figure 6.1. *Abstract syntax of System F$_c$*

As is to be expected, even the elaboration target of a realistic programming language needs many more constructs than the idealized calculi of Chapter 4. Fortunately most individual constructs are already familiar. The only new expression forms are local abstract type definitions $\mathsf{type}\ \alpha : \kappa\ \mathsf{in}\ e$, local axiom definitions $\mathsf{axiom}\ c : T \sim T\ \mathsf{in}\ e$ and coercion type -powered type casts $e \blacktriangleright \gamma$. The coercion types γ are also completely new in F$_c$. Aside from the separate syntactic category of coercion types there are no unfamiliar types or kinds.

The essence of System F$_c$ is similar to System F$_\omega$ augmented with type equality coercions. Like Haskell, System F$_c$ has nontrivial kinding and higher-kinded types, but not general type level functions $\lambda\alpha : \kappa\,.\,T$. General type level functions are needed in R1ML during type checking, but all of them can be β-reduced out from the final System F_c output.

6.1 Term Typing

Our System F_c typing rules are shown in Figure 6.2. As with syntactic forms there are quite a few but most of them are not new.

$$\textsc{Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{Abs}\ \frac{\Gamma \vdash T_d : \mathsf{type} \qquad \Gamma, x : T_d \vdash e : T_c}{\Gamma \vdash \lambda x : T_d \,.\, e : T_d \to T_c} \qquad \textsc{App}\ \frac{\Gamma \vdash e_f : T_d \to T_c \qquad \Gamma \vdash e_a : T_d}{\Gamma \vdash e_f\, e_a : T_c}$$

$$\textsc{AbsT}\ \frac{\Gamma, \alpha : \kappa \vdash e : T}{\Gamma \vdash \Lambda\alpha : \kappa \,.\, e : \forall\alpha : \kappa \,.\, T} \qquad \textsc{AppT}\ \frac{\Gamma \vdash e_f : \forall\alpha : \kappa \,.\, T \qquad \Gamma \vdash T_a : \kappa}{\Gamma \vdash e_f\, \langle T_a \rangle : [T_a/\alpha]\, T}$$

$$\textsc{Pack}\ \frac{\Gamma \vdash e : [T/\alpha]T_\alpha \qquad \Gamma \vdash \exists\alpha : \kappa \,.\, T_\alpha : \mathsf{type}}{\Gamma \vdash \mathsf{pack}\ \langle T, e \rangle\ \mathsf{as}\ \exists\alpha : \kappa \,.\, T_\alpha : \exists\alpha : \kappa \,.\, T_\alpha} \qquad \textsc{Unpack}\ \frac{\Gamma \vdash e_x : \exists\alpha : \kappa \,.\, T_\alpha \qquad \Gamma, \alpha : \kappa, x : T_\alpha \vdash e : T}{\Gamma \vdash \mathsf{unpack}\ \langle \alpha, x \rangle = e_x\ \mathsf{in}\ e : T}$$

$$\textsc{LetRec}\ \frac{\overline{x} = \mathsf{dedup}(\overline{x}) \qquad \overline{\Gamma, \overline{x : T_x} \vdash e_x : T_x} \qquad \Gamma, \overline{x : T_x} \vdash e : T}{\Gamma \vdash \mathsf{let\ rec}\ \overline{x : T_x = e_x;}\ \mathsf{in}\ e : T} \qquad \textsc{Type}\ \frac{\Gamma, \alpha : \kappa \vdash e : T}{\Gamma \vdash \mathsf{type}\ \alpha : \kappa\ \mathsf{in}\ e : T}$$

$$\textsc{Axiom}\ \frac{\Gamma \vdash T_1 : \kappa \qquad \Gamma \vdash T_2 : \kappa \qquad \Gamma, c : T_1 \sim T_2 \vdash e : T}{\Gamma \vdash \mathsf{axiom}\ c : T_1 \sim T_2\ \mathsf{in}\ e : T} \qquad \textsc{Cast}\ \frac{\Gamma \vdash e : T \qquad \Gamma \vdash_{\mathsf{co}} \gamma : T \sim T'}{\Gamma \vdash e \blacktriangleright \gamma : T'}$$

$$\textsc{Record}\ \frac{\overline{\Gamma \vdash e : T} \qquad \overline{x} = \mathsf{dedup}(\overline{x})}{\Gamma \vdash \{\overline{x = e}\} : \{\overline{x : T}\}} \qquad \textsc{Select}\ \frac{\Gamma \vdash e : \{\overline{x_e : T_e}\} \qquad x : T \in \overline{x_e : T_e}}{\Gamma \vdash e.x : T} \qquad \textsc{Proxy}\ \frac{\Gamma \vdash T : \mathsf{type}}{\Gamma \vdash [T] : [= T]}$$

$$\textsc{True}\ \ \Gamma \vdash \mathsf{true} : \mathsf{bool} \qquad \textsc{False}\ \ \Gamma \vdash \mathsf{false} : \mathsf{bool} \qquad \textsc{If}\ \frac{\Gamma \vdash e_c : \mathsf{bool} \qquad \Gamma \vdash e_t : T \qquad \Gamma \vdash e_f : T}{\Gamma \vdash \mathsf{if}\ e_c\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : T}$$

Figure 6.2. *System F_c typing rules*

The rules for variable references as well as λ- and Λ-abstractions and applications are the same as in standard System F_ω. The existential packing and unpacking rules are also the same as they would be in F_ω. With nontrivial kinding PACK needs to check that the existential type has the kind type as that must be the kind of the type of any packed value. This is similar to the kind check that was added to ABS for F_ω.

The rules for local definitions, records and booleans are the same as they would be in System F_ω or even the simply typed lambda-calculus. The rule PROXY is fairly obvious and can also be derived from the 1ML encoding $[T] = \{\mathsf{typ} = \lambda x : T \,.\, \{\}\}$.

The only F_c **type functions** needed by R1ML are parameterless and do not need to be differentiated from abstract types α. But unlike Haskell, ML modules require local abstract type generation. So while [44] had global type functions we have local abstract types with the completely obvious typing rule TYPE.

A cast expression $e \blacktriangleright \gamma$ swaps the type of the expression e, which is also the source type of the coercion γ to the target type of the coercion. The source and target types are extracted from the **equality kind** of the coercion which is computed by the kinding rules in Section 6.3. The coercion acts as an **equality witness** or proof that the cast is safe i.e. that the runtime representations of the source and target types are identical.

Equality axioms create new type equalities. To solve double vision R1ML needs local axioms instead of the global ones in [44]. The typing rule AXIOM is the obvious generalization of global axiom typing; the types to be equated are checked to have the same kind κ and no kinding errors. Then the proof variable c of the desired equality kind is added to the type environment while type checking of the body e.

The local axiom typing rule is simple but requires extending the auxiliary notion of **consistency** which is required for System F_c to be sound. Clearly allowing any axiom would be unsound since it has the same effect as arbitrary unchecked casts. Consistency becomes a slightly more challenging notion in the presence of scoped axioms even though it seems obvious that F_c code produced for modules should be sound since that is the case when plain System F or RTG is used as the elaboration target.

Definition 6.1 (Value type). A type T is a value type iff it is of form $\texttt{bool}$, $T_d \rightarrow T_c$, $\{\overline{l : T_l}\}$, $[= T_c]$, $\forall\alpha : \kappa \,.\, T_b$ or $\exists\alpha : \kappa \,.\, T_b$.

So a value type is the type of a data value (scalar or composite), a function or a quantified type. This value type terminology is unrelated to the value versus pointer type terminology that concerns immediate or stack values versus address or heap values.

Definition 6.2 (Consistency). Γ is consistent iff

1. If $\Gamma \vdash_{\mathsf{co}} \gamma : \mathsf{bool} \sim T$ and T is a value type, then $T = \mathsf{bool}$.
2. If $\Gamma \vdash_{\mathsf{co}} \gamma : T_d \rightarrow T_c \sim T$ and T is a value type, then $T = T'_d \rightarrow T'_c$.
3. If $\Gamma \vdash_{\mathsf{co}} \gamma : \{\overline{l : T_l}\} \sim T$ and T is a value type, then $T = \{\overline{l : T'_l}\}$.
4. If $\Gamma \vdash_{\mathsf{co}} \gamma : [= T_c] \sim T$ and T is a value type, then $T = [= T'_c]$.
5. If $\Gamma \vdash_{\mathsf{co}} \gamma : \forall\alpha : \kappa \,.\, T_b \sim T$ and T is a value type, then $T = \forall\alpha : \kappa \,.\, T'_b$.
6. If $\Gamma \vdash_{\mathsf{co}} \gamma : \exists\alpha : \kappa \,.\, T_b \sim T$ and T is a value type, then $T = \exists\alpha : \kappa \,.\, T'_b$.

So every coercion connecting two value types that can be constructed in the context Γ must have the same outermost type constructor on both sides. In the original System F_c consistency was only required of the toplevel context. Because we have local axioms we have to require consistency of all typing contexts. And we can just do that, since we do not need to abstract over coercions (for which [44] has variants of Λ and $\forall$). If we wanted to abstract over coercions we would have to require that any inconsistency is due to coercion parameters instead of axioms, which would not be as clear-cut than just requiring consistency of the toplevel or everywhere. The proof that consistency is sufficient for soundness can be found in [44, section 3.7].

In C casts have zero runtime overhead if the representations are identical but their validity is not checked at compile time or runtime and thus they are potentially unsafe. In Java casts are considered safe because their validity is checked at runtime. The validity of System F$_c$ casts is checked at compile time with the coercion kinding rules so they have both zero overhead (because the coercion type can be erased in later compilation passes) and safety (given consistent axioms).

6.2 Type Kinding

Because F$_c$ has nontrivial kinds it needs kinding rules. Like typing, the kinding rules in Figure 6.3 are mostly the same as in System F$_\omega$.

$$\textsc{TVar}\ \frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \textsc{TForall}\ \frac{\Gamma, \alpha : \kappa \vdash T : \text{type}}{\Gamma \vdash \forall \alpha : \kappa\,.\,T : \text{type}} \qquad \textsc{TExists}\ \frac{\Gamma, \alpha : \kappa \vdash T : \text{type}}{\Gamma \vdash \exists \alpha : \kappa\,.\,T : \text{type}}$$

$$\textsc{TApp}\ \frac{\Gamma \vdash T_f : \kappa_d \to \kappa_c \qquad \Gamma \vdash T_a : \kappa_d}{\Gamma \vdash T_f\,T_a : \kappa_c} \qquad \textsc{TArrow}\ \frac{\Gamma \vdash T_d : \text{type} \qquad \Gamma \vdash T_c : \text{type}}{\Gamma \vdash T_d \to T_c : \text{type}} \qquad \textsc{TRecord}\ \frac{\overline{\Gamma \vdash T : \text{type}}}{\Gamma \vdash \{\overline{x : T}\} : \text{type}}$$

$$\textsc{TProxy}\ \frac{\Gamma \vdash T : \text{type}}{\Gamma \vdash [= T] : \text{type}} \qquad \textsc{TBool}\ \Gamma \vdash \text{bool} : \text{type}$$

Figure 6.3. *Type kinding in System F$_c$*

The most interesting fact about these kinding rules is that the type level function elimination rule TAPP lacks the corresponding abstraction rule, since F$_c$ has no syntax to support such type level abstraction. The rules TRECORD, TPROXY and TBOOL are new but unremarkable: those types have the kind `type` and so must their constituents.

6.3 Coercion Kinding

A coercion type γ of kind $T \sim T'$ is proof that casting a value of the source type T to the target type T' is safe. The rules in Figure 6.4 determine the kinds of coercion types.

Types that are castable to each other have identical runtime value representations. Castability is an equivalence relation so naturally there are rules for reflexivity, symmetry and transitivity. Reflexivity REFL states that a value of type T can be cast to that same type and the type itself also acts as the proof coercion in this case. Symmetry SYM states that if γ proves that values of type T_1 can be cast to type T_2 then values of type T_2 can also be cast to type T_1 using sym γ. Transitivity TRANS states that coercions to cast T_1 to T_2

$$\textsc{Refl}\ \frac{\Gamma \vdash T : \kappa}{\Gamma \vdash_{\mathsf{CO}} T : T \sim T} \qquad \textsc{Sym}\ \frac{\Gamma \vdash_{\mathsf{CO}} \gamma : T' \sim T}{\Gamma \vdash_{\mathsf{CO}} \mathsf{sym}\ \gamma : T \sim T'} \qquad \textsc{Trans}\ \frac{\Gamma \vdash_{\mathsf{CO}} \gamma_1 : T_1 \sim T_2 \qquad \Gamma \vdash_{\mathsf{CO}} \gamma_2 : T_2 \sim T_3}{\Gamma \vdash_{\mathsf{CO}} \gamma_1 \circ \gamma_2 : T_1 \sim T_3}$$

$$\textsc{CoVar}\ \frac{c : T \sim T' \in \Gamma}{\Gamma \vdash_{\mathsf{CO}} c : T \sim T'} \qquad \textsc{Inst}\ \frac{\Gamma \vdash_{\mathsf{CO}} \gamma : \forall \alpha : \kappa \,.\, T \sim \forall \beta : \kappa \,.\, T' \qquad \Gamma \vdash T_{arg} : \kappa}{\Gamma \vdash_{\mathsf{CO}} \gamma @ T_{arg} : [T_{arg}/\alpha]T \sim [T_{arg}/\beta]T'}$$

$$\textsc{Comp}\ \frac{\Gamma \vdash_{\mathsf{CO}} \gamma : T \sim T' \qquad \Gamma \vdash_{\mathsf{CO}} \gamma_a : T_a \sim T'_a \qquad \Gamma \vdash T\,T_a : \kappa}{\Gamma \vdash_{\mathsf{CO}} \gamma\,\gamma_a : T\,T_a \sim T'\,T'_a}$$

$$\textsc{ArrowCo}\ \frac{\Gamma \vdash_{\mathsf{CO}} \gamma_d : T_d \sim T'_d \qquad \Gamma \vdash_{\mathsf{CO}} \gamma_c : T_c \sim T'_c \qquad \Gamma \vdash T_d \to T_c}{\Gamma \vdash_{\mathsf{CO}} \gamma_d \to \gamma_c : T_d \to T_c \sim T'_d \to T'_c}$$

$$\textsc{RecordCo}\ \frac{\overline{\Gamma \vdash_{\mathsf{CO}} \gamma : T \sim T'} \qquad \overline{\Gamma \vdash T : \mathsf{type}}}{\Gamma \vdash_{\mathsf{CO}} \{\overline{x : \gamma}\} : \{\overline{x : T}\} \sim \{\overline{x : T'}\}} \qquad \textsc{ProxyCo}\ \frac{\Gamma \vdash_{\mathsf{CO}} \gamma : T \sim T' \qquad \Gamma \vdash T : \mathsf{type}}{\Gamma \vdash_{\mathsf{CO}} [= \gamma] : [= T] \sim [= T']}$$

Figure 6.4. *Coercion kinding in System F_c*

and T_2 to T_3 can be combined with $\circ$ to directly cast T_1 to T_3.

As usual for variable rules, CoVar gets the equality kind from the typing context. Coercions between universal types with equal-kinded parameters can be specialized with @ according to rule CoInst, instantiating the universal types on both sides of the coercion's kind. In Comp application-like composition of coercions creates coercions between (well-kinded) type applications. As in kinding the function, record and type proxy rules are really just special cases of the type application logic (in this case rule Comp). Booleans are covered by rule Refl.

There are more coercion kinding rules than this in [44]. Apparently avoiding double vision is not a particularly demanding use of F_c, so R1ML gets by with just the functionality in Figure 6.4.

7 THE R1ML TYPE SYSTEM

Scope is everything!

Daniel P. Friedman

At long last this chapter contains and explains the actual subject of this thesis: the R1ML type system. As already stated this type system is largely based on 1ML [38] and System F$_c$ [44], but also heavily indebted to "Complete and Easy..." [9] and even MixML [39].

7.1 Syntax

This section lays out the abstract term and type syntax of R1ML. The subset of F$_c$ types that is used internally by the type system is also introduced. Figure 7.1 shows the abstract syntax of R1ML terms and types, which is broadly similar but not identical to that of 1ML.

$$\begin{aligned}
T \;&::= (x : T)\,(\rightarrow \mid \Rightarrow)\,T \mid {}'x \Rightarrow T \mid \{\} \mid \{\mathsf{extends}\ T;\ (x : T;)^*\} \mid T\ \mathsf{where}\ \ (\overline{.x} : T) \\
&\quad\mid E \mid = E \mid \mathsf{type} \mid \_ \mid \mathsf{bool} \\
E \;&::= x \mid \mathsf{fun}\ (x : T) \Rightarrow E \mid E\,E \;\mid\; \mathsf{let}\ (x : T? = E)^+\ \mathsf{in}\ E \mid x :> T \\
&\quad\mid \{\} \mid \{\mathsf{extends}\ E;\ (x = E;)^*\} \mid E.x \mid \mathsf{type}\ T \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E
\end{aligned}$$

Figure 7.1. *Abstract (desugared) syntax of R1ML*

Function types allow naming the parameter so that the codomain of a functor can depend on the type members of the domain. Although the function syntax is the same as in dependently typed systems, the semantics is much more limited since paths must have type `type`. The domain of implicit function types $'x \Rightarrow T$ is eliminated automatically if the type system implicitly applies the function. The domain of implicit functions is always `type`. Record types are more restricted than in 1ML, only supporting single (structural) inheritance via an initial `extends` declaration. Record types can be refined with `where` as usual in ML modules but the construct has the more general 1ML form. A semantic path can be any expression E. A singleton type $= E$ is the most specific type (including the identities of abstract types) of the expression E. Abstract types are introduced by the type literal `type`. Type holes _ can be left to be filled in by type inference. As before, builtin scalar types are represented by booleans `bool`.

Variable references, function application, record field selection, boolean constants and conditionals have the syntax we have come to expect. Function literals $\mathsf{fun}\ (x : T) \Rightarrow E$ are identical to those of 1ML. Local definitions with `let` are supported directly unlike in 1ML. Unlike other ML:s `let` is recursive by default, like in Haskell. Sealing uses the `:>` operator of SML and 1ML. Like record types, records are more restricted than in 1ML by only supporting single inheritance. A type expression $\mathtt{type}\ T$ creates a type proxy to support semantic paths.

R1ML does not desugar as aggressively as 1ML but quite a few constructs, shown in Figures 7.2 and 7.3, are implemented as syntactic sugar. The operator $:=$ here means 'desugars to', in contrast to the usual $::=$ 'is defined as' of grammar notation. Desugaring is a convenient strategy as it reduces the number of constructs that type checking (and later compilation passes) need to handle.

$$
\begin{aligned}
P &::= (x : T) \mid x := (x : \_) \\
T \to T' &:= (x : T) \to T' \\
T \Rightarrow T' &:= (x : T) \Rightarrow T' \\
\{\overline{x : T;}\} &:= \{\mathsf{extends}\ \{\}; \overline{x : T;}\} \\
T\ \mathsf{where}\ (\overline{.x}\,\overline{P} = E) &:= T\ \mathsf{where}\ \ (\overline{.x} : \overline{P \Rightarrow}(= E)) \\
T\ \mathsf{where}\ (\mathsf{type}\ \overline{.x}\,\overline{P} = T') &:= T\ \mathsf{where}\ \ (\overline{.x} : \overline{P \Rightarrow}(= \mathsf{type}\ T')) \\
x\,\overline{P} : T &:= x : \overline{P \Rightarrow} T \\
x\,\overline{P} = E &:= x : \overline{P \Rightarrow}(= E) \\
x\,\overline{'y} : T &:= x : \overline{'y \Rightarrow} T \\
\mathsf{type}\ x\,\overline{P} &:= x : \overline{P \Rightarrow}\ \mathsf{type} \\
\mathsf{type}\ x\,\overline{P} = T &:= x : \overline{P \Rightarrow}(=\ \mathsf{type}\ T)
\end{aligned}
$$

Figure 7.2. *Syntactic sugar for types and declarations*

Type annotations on parameters P can be made optional by assuming omitted annotations to be type holes _. Conversely missing function type domain variables may be filled with freshly generated ones. Noninheriting record types and records may be implemented by inheriting from $\{\}$. Multiparameter function literals can be curried into nested function literals.

$$
\begin{aligned}
\mathsf{fun}\ x \Rightarrow E &:= \mathsf{fun}\ (x : \_) \Rightarrow E \\
\mathsf{fun}\ \overline{P} \Rightarrow E &:= \overline{\mathsf{fun}\ P \Rightarrow} E \\
\{\overline{x = E;}\} &:= \{\mathsf{extends}\ \{\}; \overline{x = E;}\} \\
x\,\overline{P} = E &:= x = \mathsf{fun}\ \overline{P} \Rightarrow E \\
\mathsf{type}\ x\,\overline{P} = T &:= x = \mathsf{fun}\ \overline{P} \Rightarrow\ \mathsf{type}\ T
\end{aligned}
$$

Figure 7.3. *Syntactic sugar for expressions and definitions*

As in any language with first class functions parameterized declarations, definitions and `where` specifications can be reduced to nonparameterized ones by moving the parameters to the right-hand side as function type or function literal parameters. The `type` declarations and definitions of ML can be implemented by using abstract types, singleton types and first-class type expressions as in 1ML.

7.1.1 Semantic Types

The semantic types of Figure 7.4 are a subset of the $\mathsf{F_c}$ types of Chapter 6. There are some differences from both the semantic 1ML types of Section 5.1 and the $\mathsf{F_c}$ types of Chapter 6.

$$\begin{array}{rl}
\Xi & ::= \exists\bar{\alpha}.\Sigma \\
\Sigma & ::= \forall\bar{\alpha}.\Sigma \to_{\iota} \Xi \mid \forall\bar{\alpha}.\{\} \to_{\mathsf{A}} \Sigma \mid \{\overline{x : \Sigma}\} \mid [= \Xi] \mid \lambda\overline{\alpha}.\tau \mid \tau\,\overline{\tau} \mid \alpha \mid \hat{\alpha} \mid \mathsf{bool} \\
\tau & ::= \tau \to_{\mathsf{I}} \tau \mid \{\overline{x : \tau}\} \mid [= \tau] \mid \lambda\overline{\alpha}.\tau \mid \tau\,\overline{\tau} \mid \alpha \mid \hat{\alpha} \mid \mathsf{bool} \\
\iota & ::= \mathsf{P} \mid \mathsf{I} \\
\gamma & ::= \Xi \mid \mathsf{sym}\,\gamma \mid \gamma \circ \gamma \mid \gamma @ \overline{\tau} \mid \{\overline{x : \gamma}\} \mid [= \gamma] \mid \gamma\,\overline{\gamma} \mid c \mid \hat{\gamma} \\
\underline{\Sigma} & ::= \Sigma \mid \_ \to_{\_} \_ \mid \{x : \_, \_\} \mid [= \_]
\end{array}$$

Figure 7.4. *Semantic types*

The subsetting into and structure of large (Σ), small (τ) and abstracted (Ξ) types has been carried over from 1ML. As in the type inference algorithm of full 1ML [38, section 7] unification variables $\hat{\alpha}$ and implicit functor types $\forall\overline{\alpha}.\{\} \to_{\mathsf{A}} \Sigma$ have been added. The A arrow subscript is just a marker, not an effect. Implicit functions must always be pure since implicit side effects are considered harmful for obvious reasons. Effects ι are limited to just pure P and impure I as in 1ML.

The unification variable syntax is from [9] instead of 1ML. Inspired by [9] the unification variable mechanism has been extended to also cover the needs of translucency. This necessitated the expansion of semantic path syntax to $\tau\,\overline{\tau}$ and the addition of type level function literals $\lambda\overline{\alpha}.\tau$. Fortunately we do not actually need to extend System $\mathsf{F_c}$ with general type level functions since R1ML only produces type level function literals as part of β-redexes, which can be reduced away immediately after type checking.

Coercions γ have the same general structure as in Chapter 6. Naturally only semantic types (Ξ, which includes Σ and τ) are used as coercions. Since quantifiers can bind multiple type variables, instantiation coercions $\gamma @ \overline{\tau}$ can take multiple arguments. As a consequence of predicative instantiation those arguments are always small types.

Elimination rules require types with a particular outer structure. This is implemented by passing a type template $\underline{\Sigma}$ to focalization. There are templates for functions, records with a given field and type proxies. Any (large) type Σ may also be used as a template by only considering its outermost structure.

7.1.2 Type Environments

R1ML type environments are much more complex than usual. The complexity stems from the needs to support forward references without double vision and type inference that respects the scope of rigid type variables α, all in an elaborating type system.

$$\begin{aligned}
\Gamma, \Theta, \Delta &::= \epsilon \mid \Gamma, \{\overline{B}\} \mid \Gamma, \{\overline{B}; \overline{x : \Sigma}\} \mid \Gamma, \{\overline{\alpha}\} \mid \Gamma, \{\theta\}^{\wedge} \mid \Gamma, \{\overline{\alpha}; \theta\}^{\wedge} \mid \Gamma, \{\overline{c : \tau \sim \tau}\} \\
B &::= x :^{\circ} T \mid x :^{\bullet} T \mid x :^{\bullet} \overline{\alpha} \,.\, \Sigma \\
&\quad\;\; \mid x =^{\circ} E \mid x =^{\bullet} E \mid x :_{\iota} \Sigma =^{\bullet} e \\
\theta &::= \epsilon \mid \theta, \hat{\alpha} \mid \theta, \hat{\alpha} = \tau
\end{aligned}$$

Figure 7.5. *Type environments*

Because of the parallel nature of recursive binding constructs type environments Γ are sequences of scopes $\{...\}$ instead of just term variable type and type variable kind bindings as is usual. Ordering binding scopes $\{\overline{B}; \overline{x : \Sigma}\}$ correspond to record type (structure signature) declarations as well as record (structure) bindings while non-ordering binding scopes $\{\overline{B}\}$ are used for `let` bindings.

Abstract type scopes $\{\overline{\alpha}\}$ just wrap abstract type bindings which are used for quantifier handing as usual. Hoisting scopes $\{\overline{\alpha}; \theta\}^{\wedge}$ are unique to R1ML and accumulate a substitution θ of unification variables and a set of recursively abstract types $\overline{\alpha}$ that will be elaborated to System F$_\text{C}$ `type` definitions. Axiom scopes $\{\overline{c : \tau \sim \tau}\}$ are also novel and correspond to sequences of F$_\text{C}$ `axiom` definitions which locally define the implementation of abstract types.

Bindings B for record type declarations and annotated definitions ($x : T$) and unannotated definitions ($x = e$) each have the three states typical of general graph algorithms: unvisited (white $\circ$), in progress (grey $\bullet$) and visited (black $\bullet$). The unvisited and in progress states are identical to the binding in the source program. The visited state of a declaration or annotated binding replaces the type T with a semantic type Σ accompanied with the abstract types $\overline{\alpha}$ generated at the binding to avoid double vision. The visited state of an unannotated definition replaces the expression E with its elaboration e and also retains its type Σ and effect ι.

Substitutions θ are sets of unification variable bindings as usual for type inference algorithms. However usually substitutions only include the bound unification variable bindings $\hat{\alpha} = \tau$. Following [38] and [9] R1ML relates all unification variables to the scoping of rigid type variables α. Unlike the HM Algorithm W or 1ML type inference but following [9] R1ML integrates unification variable bindings into the type environment instead of having a single separate global substitution. As in [9], due to the integration of substitutions θ many judgements produce an output type environment ($\dashv \Delta$) in addition to taking an input environment ($\Gamma \vdash$), which is why type environments may also be denoted by Θ and Δ.

The partitioning of the type environment into scopes and the integration of unification variables into the type environment should translate into an extension of rank-based type inference [16] more readily than [38] or even [9]. Level-based type inference with mutable unification variables would remove the need for an output type environment as it eliminates the explicit substitutions θ.

R1ML employs special syntax, mostly derived from [9], for various operations on type environments. $\Gamma[\alpha]$ checks that the rigid type variable α is in scope in Γ and $\Gamma[\hat{\alpha}]$ does the same for a unification variable. Furthermore $\overline{B}[B']$ checks that the binding B' is one of $\overline{B}$. Conversely $[\hat{\alpha} = \tau]\Gamma$ binds the previously unbound unification variable $\hat{\alpha}$ in Γ, producing a new type environment. Similarly $[B]\Gamma$ changes the state of the binding of the variable bound by B.

$[\theta]\Xi$ applies the substitution θ as usual while $[\Gamma]\Xi$ applies all the substitutions in Γ similar to [9]. Most powerfully and hand-wavingly the 1ML-style syntax $\Gamma\ {}_{\Theta}\!\vdash$... applies the substitutions of Θ to all the inputs of the judgment. As the R1ML type system is (perhaps too) algorithmic the inputs can be equated with the parameters of the type checker function (procedure) corresponding to the judgement[1].

$\Gamma \uparrow \overline{\alpha}$ and $\Gamma \uparrow \overline{\hat{\alpha}}$ add the type or unification variable to the innermost hoisting scope in Γ. $\Gamma \uparrow \theta$ extends the innermost hoisting scope with an entire substitution. This is needed to salvage an incomplete substitution for later refinement when leaving its hoisting scope, a consideration which [9] seems to ignore and which is handled quite differently in full 1ML which uses a global substitution.

7.2 Lookup

In R1ML local definitions, record fields (structure members) and record type fields (signature members) are recursive. They cannot be handled like `let rec` in Section 4.1.2, by just pushing the types to the type environment before typechecking the definitions. Type elaboration by itself is not the issue since the types could just be elaborated first and then the elaborated types added to the environment. Unfortunately that is not possible because the types can refer to each other – including forward references – via path types E and singleton types $= E$, so the bindings need to be added to the environment before even elaborating the types.

The presence of unannotated definitions $x = E$ is another complication. It would be unsatisfactory to ban all forward references to unannotated variables since the Hindley-Milner type inference of conventional ML supports unannotated recursion for monotyped definitions. Forward references can also be desirable even without recursion, for top-down code organization. Having to add type annotations just to reorder definitions would be annoying. Anecdotally, avoiding such tedious addition of annotations when refactoring has been a major ceonsideration for ML$^{\mathsf{F}}$ [2, sec. 4.5].

The type of variable has to be determined when typechecking the definition of the variable

[1]Although compilers usually use imperative unification where explicit substitutions do not exist.

or at the first (transitive) forward reference to the variable. In either case the lookup judgement $\Gamma \vdash \mathit{def}(x) \Rightarrow \overline{\alpha}.\Sigma \dashv \Delta$ for annotated definitions and structure signature declarations or $\Gamma \vdash \mathit{def}(x) \Rightarrow_\iota \Sigma, e \dashv \Delta$ for unannotated definitions is invoked. The lookup rules are found in Figure 7.6.

LWHITEDECL

$$\frac{\Gamma, \{[x :^{\bullet} T]\overline{B}; \overline{y : \Sigma'}\} \vdash T \rightsquigarrow \exists\overline{\alpha}.\Sigma \dashv \Delta, \{\overline{B'}[x :^{\bullet} T]; \overline{y : \Sigma'}\}}{\Gamma, \{\overline{B}[x :^{\circ} T]; \overline{y : \Sigma'}\}, \Gamma' \vdash \mathit{def}(x) \Rightarrow \overline{\alpha}\,.\,\Sigma \dashv \Delta \uparrow \overline{\alpha}, \{[x :^{\bullet} \overline{\alpha}\,.\,\Sigma]\overline{B'}; \overline{y : \Sigma'}, x : \Sigma\}, \Gamma'}$$

LWHITE

$$\frac{\Gamma, \{[x =^{\bullet} E]\overline{B}; \overline{y : \Sigma'}\} \vdash E \Rightarrow_\iota \Sigma \dashv \Delta, \{\overline{B'}[x =^{\bullet} E]; \overline{y : \Sigma'}\} \rightsquigarrow e}{\Gamma, \{\overline{B}[x =^{\circ} E]; \overline{y : \Sigma'}\}, \Gamma' \vdash \mathit{def}(x) \Rightarrow_\iota \Sigma, e \dashv \Delta, \{[x :_\iota \Sigma =^{\bullet} e]\overline{B'}; \overline{y : \Sigma'}, x : \Sigma\}, \Gamma'}$$

LGREYDECL

$$\Gamma, \{\overline{B}[x :^{\bullet} T]; \overline{y : \Sigma'}\}, \Gamma' \vdash \mathit{def}(x) \Rightarrow \hat{\alpha} \dashv \Gamma \uparrow \hat{\alpha}, \{\overline{B}[x :^{\bullet} \hat{\alpha}]; \overline{y : \Sigma'}\}, \Gamma'$$

LGREY

$$\Gamma, \{\overline{B}[x =^{\bullet} E]; \overline{y : \Sigma'}\}, \Gamma' \vdash \mathit{def}(x) \Rightarrow \hat{\alpha} \dashv \Gamma \uparrow \hat{\alpha}, \{\overline{B}[x :^{\bullet} \hat{\alpha}]; \overline{y : \Sigma'}\}, \Gamma'$$

LWHITEDECL'

$$\frac{\begin{array}{c}\Gamma, \{[x :^{\bullet} T]\overline{B}; \overline{y : \Sigma'}\} \vdash T \rightsquigarrow \tau' \dashv \Theta, \{\overline{B'}[x :^{\bullet} \tau]; \overline{y : \Sigma'}\} \\ \Theta, \{\overline{B'}; \overline{y : \Sigma'}\}\, {}_{\Theta}\!\vdash \tau' \sim \tau \dashv \Delta, \{\overline{B''}; \overline{y : \Sigma'}\} \rightsquigarrow \gamma\end{array}}{\Gamma, \{\overline{B}[x :^{\circ} T]; \overline{y : \Sigma'}\}, \Gamma' \vdash \mathit{def}(x) \Rightarrow \tau \dashv \Delta, \{\overline{B'}; \overline{y : \Sigma'}, x : \tau\}, \Gamma'}$$

LWHITE'

$$\frac{\begin{array}{c}\Gamma, \{[x =^{\bullet} E]\overline{B}; \overline{y : \Sigma'}\} \vdash E \Rightarrow_\iota \Sigma \dashv \Theta, \{\overline{B'}[x :^{\bullet} \tau]; \overline{y : \Sigma'}\} \rightsquigarrow e \\ \Theta, \{\overline{B'}; \overline{y : \Sigma'}\}\, {}_{\Theta}\!\vdash \Sigma <:^{\text{check}} \tau \dashv \Delta, \{\overline{B''}; \overline{y : \Sigma'}\} \rightsquigarrow f\end{array}}{\Gamma, \{\overline{B}[x^{\circ} = E]\}, \Gamma' \vdash \mathit{def}(x) \Rightarrow_\iota \tau, f\ e \dashv \Delta, \{[x :_\iota \tau =^{\bullet} f\ e]\overline{B''}; \overline{y : \Sigma'}, x : \tau\}, \Gamma'}$$

LBLACKDECL

$$\Gamma[x :^{\bullet} \overline{\alpha}\,.\,\Sigma] \vdash \mathit{def}(x) \Rightarrow \overline{\alpha}\,.\,\Sigma \dashv \Gamma$$

LBLACK

$$\Gamma[x : \Sigma =^{\bullet}_{\iota} e] \vdash \mathit{def}(x) \Rightarrow_\iota \Sigma, e \dashv \Gamma$$

Figure 7.6. *Lookup* $\Gamma \vdash \mathit{def}(x) \Rightarrow (\overline{\alpha}\,.\,\Sigma \mid {}_\iota\Sigma, e) \dashv \Delta$

An unvisited binding is marked grey and then the type annotation is elaborated or the defining expression typed. If T or E does not (transitively) refer to its own variable x the binding will still be grey when the elaboration or typing is complete and LWHITEDECL or LWHITE applies. The type elaboration or expression typing results are essentially returned and also cached in the output type environment for LBLACKDECL and LBLACK which just return the cached type and its accessories. Any scopes nested inside the scope of the variable x must be ignored in the premises but copied to the output environment of lookup rules.

To support forward references LWHITEDECL hoists the abstract types ($\Delta \uparrow \overline{\alpha}$) so that they will gain System F$_\text{c}$ `type` definitions at some outer scope. The type of x will then refer to those hoisted types instead of being existential so the quantifier is dropped. The abstract types still need to be returned and cached so that the defining expression of an annotated definition can be sealed with the elaborated annotation without double vision.

LGREYDECL and LGREY produce and store in the binding a fresh unification variable as a placeholder for the eventual elaborated T. The unification variable restricts x to a monotype and the handling of unification variables elsewhere avoids cycles. LGREY opportunistically uses the same black binding form as LGREYDECL because it has no elaborated expression or effect for the black form of unannotated bindings. This is fine since the expression and effect are only needed at the definition site which always gets them after LWHITE(') is complete.

If LGREYDECL was used the binding will already be black by the time the annotation has been elaborated and LWHITEDECL' is used instead of LWHITEDECL. The elaborated type τ' must then unify with the monotype τ from LGREYDECL, which also implies that the elaborated type must be a monotype as well. Unification is used instead of the more flexible subtyping to adhere more closely to the type expressions supplied by the programmer. Besides, subtyping does not seem to make sense for record type declarations where coercion functions cannot be called. The coercion type γ has no use in structure type declarations and can also be ignored for annotated bindings by checking the defining expression against τ instead of τ'.

Similar to LWHITE' if LGREY was used the binding will already be black by the time the defining expression E has been typed and LWHITE' is used instead of LWHITE. Here subtyping can be used because the elaborated expression e is always available and there are no programmer-supplied type annotations to adhere to.

Abstract types (type, elaborates to $[= \alpha]$) can be subtyped by concrete types (type T, elaborates to $[= \Sigma]$). Matching higher-kinded abstract types to their implementations this way is only decidable at these so-called **roots** and not at use sites. So subtyping has to be manouvered into traversing the root sites before any use sites.

In conventional ML modules as well as 1ML it suffices to traverse the supertype fields in order of appearance in the source code. In those languages structure signatures' declarations do not permit forward references so the declarations must be topologically sorted in dependency order in any valid signature. Since the lookup judgement visits declarations in post-order it can also conveniently collect the elaborated record fields (signature members) $\overline{x : \Sigma}$ into the signature scope $\{\overline{B}; \overline{x : \Sigma}\}$ as can be seen in the lower right corners of LWHITEDECL and LWHITEDECL'. Those fields are wrapped into a record type by TEXTENDS (in Figure 7.8). This fine-tuning enables subtypings such as `{type t = int; x :  int} <:  {x :  t; type t}`.

Singleton types $= E$ may also produce structure signatures, so structure bindings need to be treated similarly to signature declarations. Their field types are collected in the lower right corners of LWHITE and LWHITE' and wrapped into a record type by EXTENDS and PEXTENDS (in Figure 7.14). The types of `let`-bindings do not end up in record types and so do not need to be collected, so they use a $\{\overline{B}\}$ scope as shown in Figure 7.13. These `let`-scopes are ignored in the lookup rules of Figure 7.6 since they can be obtained by just removing the field collection from the structure binding lookup rules.

7.3 Types

As the semantic types used for type checking differ from source level types, R1ML requires type elaboration rules to translate from surface type syntax to semantic types. Additionally the inclusion of limited higher-kinded types requires type normalization and the combination of quantifiers and unification variables requires a way to check the scoping of the type variables in a type against a type environment.

7.3.1 Elaboration

Type elaboration is broadly similar to that of 1ML. However the recursive scoping of record type declarations and the details of our general approach cause changes both to the overall structure of the elaboration and individual rules. The type elaboration judgement $\Gamma \vdash T \rightsquigarrow \Xi \dashv \Delta$ is outlined in Figures 7.7, 7.8 and 7.9.

$$\textsc{TTop}\quad \frac{\Gamma, \{;\}^{\wedge} \vdash T \rightsquigarrow! \, \Sigma \dashv \Delta, \{\overline{\alpha}; \theta\}^{\wedge}}{\Gamma \vdash T \rightsquigarrow \exists \overline{\alpha}.\Sigma \dashv \Delta \uparrow \theta}$$

$$\textsc{TPath}\quad \frac{\Gamma \vdash E \Rightarrow_{\mathsf{P}} \Sigma \dashv \Theta \rightsquigarrow e \qquad \Theta \; {}_{\Theta}{\vdash} \Sigma \gg [= \_] \rightsquigarrow [= \exists \overline{\alpha}.\Sigma'] \dashv \Delta \rightsquigarrow f}{\Gamma \vdash E \rightsquigarrow! \, \Sigma' \dashv \Delta \uparrow \overline{\alpha}}$$

$$\textsc{TSing}\quad \frac{\Gamma \vdash E \Rightarrow_{\mathsf{P}} \Sigma \dashv \Delta \rightsquigarrow e}{\Gamma \vdash = E \rightsquigarrow! \, \Sigma \dashv \Delta} \qquad \textsc{TType}\quad \frac{\alpha : \mathsf{type}}{\Gamma \vdash \mathsf{type} \rightsquigarrow! \, \alpha], [= \alpha] \dashv \Gamma \uparrow \alpha} \qquad \textsc{THole}\quad \Gamma \vdash \_ \rightsquigarrow! \, \hat{\alpha} \dashv \Gamma \uparrow \hat{\alpha}$$

$$\textsc{TBool}\quad \Gamma \vdash \mathsf{bool} \rightsquigarrow! \, \mathsf{bool} \dashv \Gamma$$

Figure 7.7. *Type elaboration I:* $\Gamma \vdash T \rightsquigarrow \Xi \dashv \Delta$ *and* $\Gamma \vdash T \rightsquigarrow! \, \Sigma \dashv \Delta$

As in 1ML, the abstract types in a type expression end up in an outer existential quantification. Like most R1ML judgements type elaboration differs from that of 1ML by including an output type environment Δ. The implementation of that judgement has been largely delegated to the 'imperative' variant $\Gamma \vdash T \rightsquigarrow! \, \Sigma \dashv \Delta$. Rule TTop just pushes a hoisting scope to the type environment to collect the abstract types $\overline{\alpha}$ from the elaboration of T and then existentially quantifies the abstract types and hoists the substitution θ in case it has any uninitialized unification variables.

Path types E and singleton types $= E$ are handled very similarly to (full) 1ML. The type of the expression is synthesized and the effect must be P. The type is then focalized to a carrier type $[= \Xi]$ and the inner type Ξ extracted. The abstract types are then hoisted to be caught by TTop and finally the type Σ is returned. The elaboration of singleton types starts with the same pure expression typing but the type Σ is returned directly and since it cannot be existentially quantified there are no abstract types to hoist.

A `type` expression produces a carrier type containing a fresh abstract type α. The abstract type is hoisted to the output environment to be later quantified by TTOP. THOLE produces a unification variable that is also hoisted but not wrapped in a carrier type as it does not need to have type `type`.

The boolean type just elaborates to $\mathsf{F_c}$ `bool`. The type environment is unchanged. Other builtin scalar types would behave similarly.

TEMPTY
$$\Gamma \vdash \{\} \rightsquigarrow! \{\} \dashv \Gamma$$

TEXTENDS
$$\frac{\Gamma \vdash T_s \rightsquigarrow! \{\overline{y : \Sigma}\} \dashv \Theta \qquad \overline{y}, \overline{x} = \mathsf{dedup}(\overline{y}, \overline{x}) \qquad \Theta, \{\overline{y :^\bullet \Sigma}, \overline{x :^\circ T_x}; \overline{y : \Sigma}\} \;{}_{\Theta}{\vdash} \overline{x : T_x} \dashv \Delta, \{\overline{y :^\bullet \Sigma}, \overline{x :^\bullet \Sigma_x}; \overline{y : \Sigma}, \overline{x' : \Sigma_x}\}}{\Gamma \vdash \{\mathsf{extends}\, T; \overline{x : T_x}\} \rightsquigarrow! \{\overline{y : \Sigma}, \overline{x' : \Sigma_x}\} \dashv \Delta}$$

DVAR
$$\frac{\Gamma \vdash def(x) \Rightarrow \Sigma_x \dashv \Theta \qquad \Theta \;{}_{\Theta}{\vdash} \overline{D} \rightsquigarrow! \dashv \Delta}{\Gamma \vdash x : T, \overline{D} \dashv \Delta}$$

DEMPTY
$$\Gamma \vdash \epsilon \dashv \Gamma$$

TWHERE
$$\frac{\Gamma \vdash T \rightsquigarrow \exists\overline{\alpha}.\Sigma \dashv \Theta \qquad \overline{\alpha}' = \overline{\alpha} \cap \mathsf{ftv}(\Sigma\overline{.x}) \qquad \overline{\alpha}'' = \overline{\alpha} \setminus \overline{\alpha}' \qquad \Theta \uparrow \overline{\alpha}'' \;{}_{\Theta}{\vdash} T_{\overline{x}} \rightsquigarrow! \Sigma_{\overline{x}} \dashv \Theta' \qquad \Theta', \{\overline{\hat{\alpha}}'\} \;{}_{\Theta'}{\vdash} \Sigma_{\overline{x}} <:^{\mathsf{check}} \left[\overline{\hat{\alpha}'/\alpha'}\right] \Sigma\overline{.x} \dashv \Delta, \{\theta\} \rightsquigarrow f}{\Gamma \vdash T \;\mathsf{where}\; (\overline{.x} : T_{\overline{x}}) \rightsquigarrow! [\theta][\overline{\hat{\alpha}'/\alpha'}]\Sigma \dashv \Delta}$$

Figure 7.8. *Type elaboration II: record types*

Since $\{\overline{x : T_x;}\}$ is just syntactic sugar for $\{\mathsf{extends}\ \{\}; \overline{x : T_x;}\}$ the elaboration of nonempty record types in TEXTENDS always begins with the elaboration of the supertype T. The fields of the elaborated supertype are then brought into scope as variables for the elaboration of the additional fields like in SML `open` – or Java `extends`. The super and added fields are then concatenated into a semantic record type. Because the scope is recursive there is no sensible way to pick between duplicate fields so duplicate fields are disallowed with a side condition. Multiple inheritance could be supported in this system but not with row types (see Section 9.5) so things are kept simple with single inheritance.

The new fields in TEXTENDS are elaborated by DVAR and the base case DEMPTY. DVAR just delegates the actual elaboration of the field type to lookup.

The elaboration and realization of field refinement in TWHERE begins by elaborating the original type T. The free type variables of Σ are then split into $\overline{\alpha}'$ which appear in the part selected by the simple dotted path $\overline{.x}$ and the rest $\overline{\alpha}''$. The latter will just become abstract types and are hoisted for TTOP to find. The refinement type $T_{\overline{x}}$ is elaborated with $\rightsquigarrow!$ so its abstract types also go to TTOP. The `where` operation can then be performed by replacing the $\overline{\alpha}'$ with unification variables in $\Sigma\overline{.x}$ and subtyping that with the elaborated refinement type. That amounts to partial instantiation of the original type with the refinement type, which is exactly what `where` should do. It is important to finally apply the substitution θ to the entire Σ to replace the abstract types everywhere in the original type which is also part of the semantics of `where`.

TFUNCTOR

$$\frac{\Gamma \vdash T_d \rightsquigarrow \exists\overline{\alpha}.\Sigma_d \dashv \Theta \qquad \Theta, \{\overline{\alpha}\}, \{x :^{\bullet} \Sigma_d\} {}_{\Theta}\!\vdash T_c \rightsquigarrow \Xi_c \dashv \Delta, \{\overline{\alpha}\}, \Theta'}{\Gamma \vdash (x : T_d) \to T_c \rightsquigarrow ! \forall\overline{\alpha}.\Sigma_d \to_{\mathsf{I}} \Xi_c \dashv \Delta}$$

TPFUNCTOR

$$\frac{\Gamma \vdash T_d \rightsquigarrow \exists\overline{\alpha}.\Sigma_d \dashv \Theta \qquad \Theta, \{\overline{\alpha}\}, \{x :^{\bullet} \Sigma_d\} {}_{\Theta}\!\vdash T_c \rightsquigarrow \exists\overline{\beta}.\Sigma_c \dashv \Delta, \{\overline{\alpha}\}, \Theta' \qquad \overline{\beta' : \overline{\kappa_\alpha \to} \kappa_\beta}}{\Gamma \vdash (x : T_d) \Rightarrow T_c \rightsquigarrow ! \left[\overline{\beta'\overline{\alpha}/\beta}\right] (\forall\overline{\alpha}.\Sigma_d \to_{\mathsf{P}} \Sigma_c) \dashv \Delta \uparrow \overline{\beta}'}$$

TIMPLICIT

$$\frac{\Gamma, \{\alpha\}, \{x :^{\bullet} \alpha], [= \alpha]\} \vdash T_c \rightsquigarrow \Sigma_c \dashv \Delta, \{\alpha\}, \Theta \qquad \alpha : \mathsf{type}}{\Gamma \vdash {}'x \Rightarrow T_c \rightsquigarrow ! \forall\alpha.\{\} \to_{\mathsf{A}} \Sigma_c \dashv \Delta}$$

Figure 7.9. *Type elaboration III: function types*

The elaboration of function types is quite similar to the typing of function expressions. The domain type is elaborated and both its abstract types $\overline{\alpha}$ and the unquantified interior using them are brought into scope before elaborating the codomain type. For the impure function types of TFUNCTOR the abstract domain type variables, domain type and codomain types are then just assembled into a semantic generative functor type. For the pure function types of TPFUNCTOR the abstract codomain types are lifted over the abstract domain types as in 1ML. As in the other R1ML rules those abstract types are then hoisted into the environment instead of being existentially bound as in 1ML.

The elaboration of implicit functions in TIMPLICIT is essentially the specialization of TFUNCTOR to domains of type `type`. The semantic implicit function type has an empty record as its domain instead of $[= \alpha]$ since `let`-generalization can create implicit functions with more than one type parameter and incorporating any number of type parameters into the domain type would create incidental complications.

7.3.2 Normalization

Since semantic types include type level functions and applications, their equivalence relation needs to include β-equivalence as in System F_ω. In type equivalence *algorithms* it is more convenient to first normalize the input types so that β-redexes will not be encountered during equivalence checking. R1ML uses subtyping and occasionally unification instead of type equivalence, but the normalize-first strategy can be used also for those judgements and focalization.

It would seem that subtyping and unification require the use of full reduction as the normalization strategy. On the other hand focalization only looks at the outermost structure of types so normalizing beyond weak head normal form would be a wasted effort. Having several evaluation strategies would be redundant. As we will see it is possible to avoid full reduction by interleaving subtyping, unification and focalization with call by name reduction. Figure 7.10 shows the call by name rules for R1ML type normalization.

$$\textsc{RApp}\quad \frac{\Gamma \vdash \tau_f \longrightarrow \tau_f' \rightsquigarrow \gamma_f \qquad \Gamma \vdash \tau_f' \cdot \overline{\tau_a} \longrightarrow \tau \rightsquigarrow \gamma}{\Gamma \vdash \tau_f\, \overline{\tau_a} \longrightarrow \tau \rightsquigarrow \gamma_f @\overline{\tau_a} \circ \gamma} \qquad \textsc{RAxiom}\quad \frac{\Gamma \vdash \tau \longrightarrow \tau' \rightsquigarrow \gamma}{\Gamma[c : \alpha \sim \tau] \vdash \alpha \longrightarrow \tau' \rightsquigarrow c \circ \gamma}$$

$$\textsc{RWhnf}\quad \Gamma \vdash \Sigma \longrightarrow \Sigma \rightsquigarrow \Sigma \qquad \text{RA}\beta\quad \frac{\Gamma \vdash [\overline{\tau_a/\alpha}]\tau \longrightarrow \tau' \rightsquigarrow \gamma}{\Gamma \vdash (\lambda\overline{\alpha : \kappa}.\tau) \cdot \overline{\tau_a} \longrightarrow \tau' \rightsquigarrow \gamma} \qquad \textsc{RARigid}\quad \Gamma \vdash \alpha \cdot \overline{\tau_a} \longrightarrow \alpha\, \overline{\tau_a} \rightsquigarrow \alpha\, \overline{\tau_a}$$

$$\textsc{RAApp}\quad \Gamma \vdash \tau\, \overline{\tau} \cdot \overline{\tau_a} \longrightarrow \tau\, \overline{\tau}\, \overline{\tau_a} \rightsquigarrow \tau\, \overline{\tau}\, \overline{\tau_a} \qquad \textsc{RAUni}\quad \Gamma \vdash \hat{\alpha} \cdot \overline{\tau_a} \longrightarrow \bot$$

Figure 7.10. *Type normalization* $\Gamma \vdash \Sigma \longrightarrow \Sigma|\bot \rightsquigarrow \gamma$ *and* $\Gamma \vdash \tau \cdot \overline{\tau} \longrightarrow \tau|\bot \rightsquigarrow \gamma$

RAPP evaluates the callee and the delegates to the 'apply' judgement $\Gamma \vdash \Sigma_f \cdot \Sigma_a \longrightarrow \Sigma' \rightsquigarrow \gamma$. RAXIOM reads through an axiom binding that is in scope for an abstract type preventing double vision (although we also have to arrange for axioms to be in scope for the right terms). If neither RAPP not RAXIOM applies the type must already be in weak head normal form and normalization is done (RWHNF).

Rule RAβ reduces a beta-redex by substitution and keeps reducing the result. RARIGID and RAAPP just reassemble the type application since abstract types behave as type constructors. When a unification variable is to be applied RAUNI fails explicitly by producing bottom $\bot$. Such an application may succeed later when the unification variable has been initialized so subtyping and unification know what to do when they receive $\bot$. To avoid clutter the normalization rules assume that $\bot$ is bubbled through them like an exception to the initiator of the normalization.

7.3.3 Well-Formedness

The well-formedness judgement for semantic types is presented in Figure 7.11. The goal is just to check type variable scoping. Semantic types are well kinded by construction; because type level functions are tied to applicative functors, any kind errors are just reflections of type errors. Types do not need to be normalized for this judgement.

$$\textsc{WExists}\quad \frac{\Gamma, \{\overline{\alpha}\} \vdash \Sigma}{\Gamma \vdash \exists\overline{\alpha}.\Sigma} \qquad \textsc{WForall}\quad \frac{\Gamma, \{\overline{\alpha}\} \vdash \Sigma}{\Gamma \vdash \forall\overline{\alpha}.\Sigma} \qquad \textsc{WArrow}\quad \frac{\Gamma \vdash \Sigma \qquad \Gamma \vdash \Xi}{\Gamma \vdash \Sigma \rightarrow_\iota \Xi} \qquad \textsc{WRecord}\quad \frac{\overline{\Gamma \vdash \Sigma}}{\Gamma \vdash \{\overline{x : \Sigma}\}} \qquad \textsc{WType}\quad \frac{\Gamma \vdash \Xi}{\Gamma \vdash [= \Xi]}$$

$$\textsc{WBool}\quad \Gamma \vdash \text{bool} \qquad \text{W}\lambda\quad \frac{\Gamma, \{\overline{\alpha}\} \vdash \tau}{\Gamma \vdash \lambda\overline{\alpha}.\tau} \qquad \textsc{WApp}\quad \frac{\Gamma \vdash \tau \qquad \overline{\Gamma \vdash \tau'}}{\Gamma \vdash \tau\, \overline{\tau}'} \qquad \textsc{WAVal}\quad \frac{\Gamma \vdash \tau}{\Gamma[c : \alpha \sim \tau] \vdash \alpha} \qquad \textsc{WRigid}\quad \Gamma[\alpha] \vdash \alpha \qquad \textsc{WUni}\quad \Gamma[\hat{\alpha}] \vdash \hat{\alpha}$$

Figure 7.11. *Well-formedness of semantic types* $\Gamma \vdash \Xi$

The central rules are WRIGID and WUNI, which check that references to type and unification variables are well-scoped. Otherwise the rules follow a traversal pattern we will see several more times; structural recursion which additionally adds quantified type variables to the type environment in WEXISTS, WFORALL and Wλ.

7.4 Terms

This section shows the R1ML term typing rules. Term typing is the core of any type system but as often happens R1ML term typing is mostly a straightforward traversal of the program that delegates most interesting decisions to other judgements.

7.4.1 Expression Type Synthesis

R1ML expression typing is **bidirectional** like [9], which means that instead of one typing judgement $E : \Sigma$ ("E has type Σ") we have **synthesis** $E \Rightarrow \Sigma$ ("the inferred type of E is Σ") and **checking** $E \Leftarrow \Sigma$ ("Σ is a valid type for E"). Expression typing is also elaborating like in 1ML, producing an expression $\rightsquigarrow e$. The expression type synthesis rules are divided into Figures 7.12, 7.13, 7.14 and 7.15.

VAR

$$\frac{\Gamma \vdash \mathit{def}(x) \Rightarrow \Sigma \dashv \Delta}{\Gamma \vdash x \Rightarrow_{\mathsf{P}} \Sigma \dashv \Delta \rightsquigarrow x}$$

ABS

$$\frac{\Gamma \vdash T \rightsquigarrow \exists\overline{\alpha}.\Sigma_d \dashv \Theta \qquad \Theta, \{\overline{\alpha}\}, \{x :^{\bullet} \Sigma_d\}, \{;\}^{\wedge} {}_{\Theta}\vdash E \Rightarrow_{\iota} \Sigma_c \dashv \Delta, \{\overline{\alpha}\}, \{x :^{\bullet} \Sigma_d\}, \{\overline{\beta}; \theta\}^{\wedge} \rightsquigarrow e}{\Gamma \vdash (\mathsf{fun}\ (x : T) \Rightarrow E) \Rightarrow_{\mathsf{P}} \forall\overline{\alpha}.\Sigma_d \rightarrow_{\iota} \exists\overline{\beta}.\Sigma_c \dashv \Delta \uparrow \theta \rightsquigarrow \Lambda\overline{\alpha}.\lambda x.\overline{\mathsf{type}\ \beta\ \mathsf{in}}\ \mathsf{pack}\ \langle\overline{\beta}, e\rangle}$$

APP

$$\frac{\begin{array}{cc}\Gamma \vdash E_f \Rightarrow_{\iota_f} \Sigma_f \dashv \Theta \rightsquigarrow e_f & \Theta\ {}_{\Theta}\vdash \Sigma_f \gg \_ \rightarrow_{\_} \_ \rightsquigarrow \Sigma_d \rightarrow_{\iota} \exists\overline{\alpha}.\Sigma_c \dashv \Theta' \rightsquigarrow f \\ \Theta'\ {}_{\Theta'}\vdash E_a \Leftarrow_{\iota_a} \Sigma_d \dashv \Theta'' \rightsquigarrow e_a & \Theta'' \uparrow \overline{\beta}, \{\overline{\alpha}\}, \{x :^{\bullet} \Sigma_c\}\ {}_{\Theta''}\vdash x \Leftarrow_{\iota_x} \overline{\beta}.\overline{[\beta/\alpha]}\Sigma_c \dashv \Delta \rightsquigarrow e_x\end{array}}{\Gamma \vdash E_f\, E_a \Rightarrow_{\iota_f \vee \iota_a \vee \iota \vee \iota_x} \overline{[\beta/\alpha]}\Sigma_c \dashv \Delta \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}, x\rangle = f\, e_f\, e_a\ \mathsf{in}\ e_x}$$

Figure 7.12. *Expression type synthesis I:* $\Gamma \vdash E \Rightarrow_{\iota} \Sigma \dashv \Delta \rightsquigarrow e$

Rule VAR just obtains the type of the variable with the lookup judgement. It ignores any accessories produced by the lookup judgement so it works for all variables. A variable reference has no side effects (because variables are immutable as usual in ML) and elaborates to a variable reference.

Function expression typing is largely based on that of 1ML. First the domain type T is elaborated into $\exists\overline{\alpha}.\Sigma_d$. The abstract types $\overline{\alpha}$ of the domain are pushed to the type environment so that the parameter type in the following scope $\{x : \Sigma_d\}$ is well scoped. A hoisting scope is also pushed to collect the abstract types $\overline{\alpha}$ generated by the function body. The body is typed in the extended environment, producing the codomain type Σ_c

and the effect ι.

The function type is assembled around the domain and codomain types. The body effect is suspended by attaching it to the function type; the effect of a function literal is always P regardless of the body and the side effects of the body happen (are ‘released’) at call sites instead. The abstract types are existentially quantified in the final codomain type, creating a generative functor. As in the basic 1ML system applicative functors can only be created by sealing a pure function with an applicative signature.

To match the existential quantification of the codomain the body expression needs to be existentially packed but on the other hand the typing of the body assumed that the abstract types will be bound by $\overline{\mathsf{type}\ \beta}$. The reconciliation is to pack the defined types as the implementations of the existentially quantified types ($\mathsf{pack}\ \langle\overline{\beta}, e\rangle$). The substitution θ is hoisted to the output environment as usual.

The main flow of function application APP is the usual bidirectional one where we synthesize the function type and check the argument against the domain from the function type. As in 1ML and [9] the function might also require coercion to get to the actual arrow type and the domain, which we do with the focalization judgement $\Gamma \vdash \Sigma \gg \Sigma\_ \rightsquigarrow \Sigma' \dashv \Delta \rightsquigarrow f$. Now we can apply the coerced function $f\ e_f$ to the likewise coerced argument e_a.

Calling a generative functor generates abstract types. To enable forward references the packing of type functions in ABS needs to be reversed, starting by unpacking the existential. Then we generate and hoist fresh type functions $\overline{\beta}$ to match the abstract types $\overline{\alpha}$ and check the fresh variable x against the pseudo-type $\overline{\beta}.[\overline{\beta/\alpha}]\Sigma_c$ to obtain the result expression e_x and the type generation effect ι_x. The effect of an application expression is the join of the callee, argument, arrow and type generation effects.

LET

$$\frac{\overline{x} = \mathsf{dedup}(\overline{x}) \qquad \Gamma, \{\}^\wedge, \{\overline{x(:^\circ T_x| =^\circ E_x)}\} \vdash \overline{x(: T_x)? = E_x} \Rightarrow_{\mathsf{I}} \dashv \Theta, \{\theta\}^\wedge, \{\overline{B}\} \rightsquigarrow \overline{x = e_x} \\ \Theta' = \Theta \uparrow \theta \qquad \Theta', \{\overline{B}\}\, {}_{\Theta'}{\vdash} E \Rightarrow_\iota \Sigma \dashv \Delta, \{\overline{B}\} \rightsquigarrow e}{\Gamma \vdash \mathsf{let}\ \overline{x(: T_x)? = E_x}\ \mathsf{in}\ E \Rightarrow_{\mathsf{I} \vee \iota} \Sigma \dashv \Delta \rightsquigarrow \mathsf{let\ rec}\ \overline{x = e_x}\ \mathsf{in}\ e}$$

PLET

$$\frac{\overline{x} = \mathsf{dedup}(\overline{x}) \\ \Gamma, \{\}^\wedge, \{\overline{x(:^\circ T_x| =^\circ E_x)}\} \vdash \overline{x(: T_x)? = E_x} \Rightarrow_{\mathsf{P}} \dashv \Theta, \{\theta\}^\wedge, \{\overline{x : \Sigma_x = e_x}\} \rightsquigarrow \overline{x = e_x} \\ \overline{\hat{\alpha}} = \mathsf{undet}(\theta) \qquad \overline{\alpha : \mathsf{type}} \qquad \Theta, \{x :^\bullet \forall\overline{\alpha}.\{\} \rightarrow_{\mathsf{A}} [\overline{\alpha/\hat{\alpha}}]\Sigma_x\}\, {}_{\Theta,\{\theta\}^\wedge}{\vdash} E \Rightarrow_\iota \Sigma \dashv \Delta, \{\overline{B}\} \rightsquigarrow e}{\Gamma \vdash \mathsf{let}\ \overline{x(: T_x)? = E_x}\ \mathsf{in}\ E \Rightarrow_{\mathsf{P} \vee \iota} \Sigma \dashv \Delta \rightsquigarrow \mathsf{let\ rec}\ \overline{x = \Lambda\overline{\alpha}.\lambda\_.[\overline{x\ \langle\overline{\alpha}\rangle\ \{\}/x}][\overline{\alpha/\hat{\alpha}}]e_x}\ \mathsf{in}\ e}$$

SEAL

$$\frac{\Gamma \vdash T \rightsquigarrow \exists\overline{\alpha}.\Sigma \dashv \Theta \qquad \Theta \uparrow \overline{\alpha}\, {}_{\Theta}{\vdash} E \Leftarrow_\iota \overline{\alpha}\,.\,\Sigma \dashv \Delta \rightsquigarrow e}{\Gamma \vdash E :> T \Rightarrow_\iota \Sigma \dashv \Delta \rightsquigarrow e}$$

TYPE

$$\frac{\Gamma \vdash T \rightsquigarrow \Xi \dashv \Delta}{\Gamma \vdash \mathsf{type}\ T \Rightarrow_{\mathsf{P}} [= \Xi] \dashv \Delta \rightsquigarrow [\Xi]}$$

Figure 7.13. *Expression type synthesis II*

The R1ML `let` is recursive by default unlike in SML and OCaml where it is sequential (without `rec` or `and`). The rule LET is essentially the same as we have seen before for `let rec`; the bindings for the definitions are pushed and then the definitions and the body are typed in that extended context. The $x(:^\circ T| =^\circ E)$ syntax tries to convey that for each annotated definition a $x :^\circ T$ binding should be created and for each unannotated definition a $x =^\circ E$ binding should be created. The definitions are typed by a separate judgement $\Gamma \vdash \overline{x(: T)? = E} \Rightarrow_\iota \overline{x : \Sigma} \dashv \Delta \rightsquigarrow \overline{x = e}$ defined in Section 7.4.3. The effect of a `let` is the join of the combined effect of the definitions ι_x and the body effect ι.

When none of the definitions have side effects, rule PLET is used instead of LET. In practice a type checker would perform the common prefix of LET and PLET and then branch on the combined effect of the definitions. Both LET and PLET push a substitution-only hoisting scope $\{\}^\wedge$. PLET uses the collected substitution θ to generalize the types $\overline{\Sigma_x}$ of the definitions. Generalization is implemented by wrapping the definiends in implicit functions and replacing the local unsolved unification variables $\mathtt{undet}(\theta)$ with the rigid universally quantified $\overline{\alpha}$. The use sites of the definiends also need to be changed to apply the implicit functions $([\overline{x \, \langle\overline{\alpha}\rangle \, \{\}/x}]e_x)$. LET just re-hoists the substitution instead because generalizing impure expressions is unsound as is well known from ML [27]. The use of an effect system (even if rudimentary) allows 1ML and R1ML to replace the very conservative **value restriction** of ML with this more permissive and also more intuitive **purity restriction** [38].

As usual for upcasting constructs in bidirectional typing rule SEAL switches from synthesis to checking. Of course the type annotation has to be elaborated first. Since existentials do not support forward references the abstract types are hoisted and the quantifier removed before checking E against the elaborated type and the type of the sealing is the unquantified Σ. All this is similar to but slightly simpler than the handling of the codomain in APP.

The typing of `type T` in rule TYPE just elaborates the type and wraps it into a carrier type. Type literals `type T` are considered to be constants so the effect is P.

Rule EXTENDS combines elements from both TEXTENDS and LET. The super-expression E is typed and its fields brought into scope for typing the additional fields $\overline{x(: T_x)? = E_x}$. The combined fields $\overline{y}, \overline{x}$ must contain no duplicates. The field definitions are typed like the definitions of a `let`. The elaborated program has to extract the super-fields $\overline{y}$ so that they are in scope for the additional fields and finally copy both sets of fields to the result record. The effect is the join of the super-expression and combined declaration effects. Like LET, EXTENDS has a generalizing sibling rule: PEXTENDS.

The typing of field selection is simple. SELECT types the record expression E and focalizes its type as a record with at least the field x. This record focalization fails if the type Σ is a unification variable as it would have to guess the labels of the other fields. The result type is extracted from the focalized type and the effect is just the effect of E.

The boolean constants `true` and `false` still have type `bool`. As constants they produce no side effects and elaborate to their counterparts in $\mathsf{F_c}$.

IF first types the condition and then focalizes that type to `bool`. Focalization is used to

EMPTY

$$\Gamma \vdash \{\} \Rightarrow_{\mathsf{P}} \{\} \dashv \Gamma \rightsquigarrow \{\}$$

EXTENDS

$$\frac{\begin{array}{c}\Gamma \vdash E \Rightarrow_{\iota} \dashv \Theta \rightsquigarrow e \qquad \overline{y}, \overline{x} = \mathsf{dedup}(\overline{y}, \overline{x}) \\ \Theta, \{\}^{\wedge}, \left\{\overline{y :^{\bullet} \Sigma}, \overline{x(:^{\circ} T_x | =^{\circ} E_x)}; \overline{y : \Sigma}\right\} {}_{\Theta}\vdash \overline{x(: T_x)? = E_x} \Rightarrow_{\mathsf{I}} \\ \dashv \Delta, \{\theta\}^{\wedge}, \{\overline{B}; \overline{y : \Sigma}, \overline{x' : \Sigma_x}\} \rightsquigarrow \overline{x = e_x}\end{array}}{\begin{array}{c}\Gamma \vdash \{\mathsf{extends}\, E; \overline{x(: T_x)? = E_x}\} \Rightarrow_{\iota \vee \mathsf{I}} \left\{\overline{y : \Sigma}, \overline{x' : \Sigma_x}\right\} \dashv \Delta \uparrow \theta \\ \rightsquigarrow \mathsf{let\ rec}\ x_s = e; \overline{y = x_s.y; x = e_x;}\ \mathsf{in}\ \left\{\overline{y = y}, \overline{x' = x'}\right\}\end{array}}$$

PEXTENDS

$$\frac{\begin{array}{c}\Gamma \vdash E \Rightarrow_{\iota} \dashv \Theta \rightsquigarrow e \qquad \overline{y}, \overline{x} = \mathsf{dedup}(\overline{y}, \overline{x}) \\ \Theta, \{\}^{\wedge}, \left\{\overline{y :^{\bullet} \Sigma}, \overline{x(:^{\circ} T_x | =^{\circ} E_x)}; \overline{y : \Sigma}\right\} {}_{\Theta}\vdash \overline{x(: T_x)? = E_x} \Rightarrow_{\mathsf{P}} \\ \dashv \Delta, \{\theta\}^{\wedge}, \{\overline{B}; \overline{y : \Sigma}, \overline{x' : \Sigma_x}\} \rightsquigarrow \overline{x = e_x} \qquad \overline{\hat{\alpha}} = \mathsf{undet}(\theta) \qquad \overline{\alpha : \mathsf{type}}\end{array}}{\begin{array}{c}\Gamma \vdash \{\mathsf{extends}\, E; \overline{x(: T_x)? = E_x}\} \Rightarrow_{\iota \vee \mathsf{P}} \left\{\overline{y : \Sigma}, \overline{x' : \forall \overline{\alpha}.\{\} \to_{\mathsf{A}} [\overline{\alpha / \hat{\alpha}}]\Sigma_x}\right\} \dashv \Delta \\ \rightsquigarrow \mathsf{let\ rec}\ x_s = e; \overline{y = x_s.y;} \overline{x = \Lambda \overline{\alpha}.\lambda\_.[\overline{x \langle \overline{\alpha} \rangle \{\} / x}][\overline{\alpha / \hat{\alpha}}]e_x;}\ \mathsf{in}\ \left\{\overline{y = y}, \overline{x' = x'}\right\}\end{array}}$$

SELECT

$$\frac{\Gamma \vdash E \Rightarrow_{\iota} \Sigma \dashv \Theta \rightsquigarrow e \qquad \Theta\, {}_{\Theta}\vdash \Sigma \gg \{x : \_, \_\} \rightsquigarrow \{\overline{y : \Sigma_y}, x : \Sigma_x, \overline{y' : \Sigma_{y'}}\} \dashv \Delta \rightsquigarrow f}{\Gamma \vdash E.x \Rightarrow_{\iota} \Sigma_x \dashv \Delta \rightsquigarrow (f\, y).x}$$

Figure 7.14. *Expression type synthesis III: records*

TRUE

$$\Gamma \vdash \mathsf{true} \Rightarrow_{\mathsf{P}} \mathsf{bool} \dashv \Gamma \rightsquigarrow \mathsf{true}$$

FALSE

$$\Gamma \vdash \mathsf{false} \Rightarrow_{\mathsf{P}} \mathsf{bool} \dashv \Gamma \rightsquigarrow \mathsf{false}$$

IFSYNTH

$$\frac{\begin{array}{c}\Gamma \vdash E_c \Rightarrow_{\iota_c} \Sigma_c \dashv \Theta \rightsquigarrow e_c \qquad \Theta\, {}_{\Theta}\vdash \Sigma_c \gg \mathsf{bool} \rightsquigarrow \mathsf{bool} \dashv \Theta' \rightsquigarrow f \\ \Theta'\, {}_{\Theta'}\vdash E_t \Rightarrow_{\iota_t} \Sigma_t \dashv \Theta'' \rightsquigarrow e_t \\ \Theta''\, {}_{\Theta''}\vdash E_f \Rightarrow_{\iota_f} \Sigma_f \dashv \Theta''' \rightsquigarrow e_f \qquad \Theta'''\, {}_{\Theta'''}\vdash \Sigma_f \sim \Sigma_t \dashv \Delta \rightsquigarrow \gamma\end{array}}{\Gamma \vdash \mathsf{if}\ E_c\ \mathsf{then}\ E_t\ \mathsf{else}\ E_f \Rightarrow_{\iota_c \vee \iota_t \vee \iota_f} \Sigma_t \dashv \Delta \rightsquigarrow \mathsf{if}\ f\, e_c\ \mathsf{then}\ e_t\ \mathsf{else}\ (e_f \blacktriangleright \gamma)}$$

Figure 7.15. *Expression type synthesis IV: booleans*

be consistent with the other elimination rules TPATH, APP and SELECT but $\Gamma \vdash E_c \Leftarrow_{\iota c}$ $\mathsf{bool} \dashv \Delta$ would be equally viable with such a simple type. The branch types have to be reconciled somehow. In a system with subtyping their join would usually be used but 1ML and R1ML types do not have joins. The `RankNTypes` paper [31] gives three options:

1. Only synthesize monomorphic types for multi-branch constructs, implemented by checking the branches against a fresh unification variable (which can only be initialized with monotypes).
2. Extend unification to handle polytypes and unify the branch types.
3. Use mutual subtyping for the branch types: $\Sigma_f <: \Sigma_t \wedge \Sigma_t <: \Sigma_f$.

They choose #3 because it is more powerful than #1 or even #2 and does not require the extra work to extend unification. Full 1ML behaves similarly to #1 (even though its typing is not bidirectional). I have chosen to use #2 because #3 would have to arbitrarily choose which coercion function to apply. While the decision to cast e_f (and not e_t) through the coercion type is still arbitrary, at least $\mathsf{F_c}$ casts have no runtime effect. Polytype unification is also useful elsewhere as witnessed by its inclusion in 1ML which chose #1 instead of #2. The effect of a conditional is the join of the condition and branch effects as is to be expected.

7.4.2 Expression Type Checking

The expression checking judgement $\Gamma \vdash E \Leftarrow_\iota \overline{\alpha}.\Sigma \dashv \Delta \rightsquigarrow e$ is detailed in Figure 7.16. It is rather trivial compared to systems like [9].The required type Σ can be accompanied with abstract types $\overline{\alpha}$ that have already been hoisted to be bound by type function declarations. Unlike the type, the effect is synthesized also in the checking judgement.

$$\textsc{IfCheck}\quad \frac{\begin{array}{cc}\Gamma \vdash E_c \Rightarrow_{\iota_c} \Sigma_c \dashv \Theta \rightsquigarrow e_c & \Theta\ {}_{\Theta}{\vdash} \Sigma_c \gg \mathsf{bool} \rightsquigarrow \mathsf{bool} \dashv \Theta' \rightsquigarrow f \\ \Theta'\ {}_{\Theta'}{\vdash} E_t \Leftarrow_{\iota_t} \overline{\alpha}\,.\,\Sigma \dashv \Theta'' \rightsquigarrow e_t & \Theta''\ {}_{\Theta''}{\vdash} E_f \Leftarrow_{\iota_f} \overline{\alpha}\,.\,\Sigma \dashv \Delta \rightsquigarrow e_f\end{array}}{\Gamma \vdash \mathsf{if}\ E_c\ \mathsf{then}\ E_t\ \mathsf{else}\ E_f \Leftarrow_{\iota_c \vee \iota_t \vee \iota_f} \overline{\alpha}\,.\,\Sigma \dashv \Delta \rightsquigarrow \mathsf{if}\ e_c\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f}$$

$$\textsc{Coerce}\quad \frac{\Gamma \uparrow \overline{\hat{\alpha}}, \{\overline{c : \alpha \sim \hat{\alpha}}\} \vdash E \Rightarrow_\iota \Sigma_E \dashv \Theta \rightsquigarrow e \qquad \Theta\ {}_{\Theta}{\vdash} \Sigma_E <:^{\mathsf{check}} \Sigma \dashv \Delta, \{\overline{c : \alpha \sim \tau}\} \rightsquigarrow f}{\Gamma \vdash E \Leftarrow_{\iota \vee \iota(\overline{\alpha})} \overline{\alpha}\,.\,\Sigma \dashv \Delta \rightsquigarrow \overline{\mathsf{axiom}\ c : \alpha \sim \tau\ \mathsf{in}}\ f\ e}$$

Figure 7.16. *Expression typechecking* $\Gamma \vdash E \Leftarrow_\iota \overline{\alpha}.\Sigma \dashv \Delta \rightsquigarrow e$

Conditionals are checked by IFCHECK which is similar to but simpler than IFSYNTH since the branch types can just be checked against the same type instead of being reconciled. This way the branches can implement the abstract types $\overline{\alpha}$ differently when a conditional is sealed and the integration of a type annotation into the `if` syntax in 1ML is not needed in R1ML. Furthermore nested conditionals can be sealed with just one type annotation instead of one per conditional.

The default checking rule in bidirectional typing with subtyping switches to synthesis and then checks that the synthesized type is a subtype of the required type. Rule COERCE does that but first pushes an axiom scope for the abstract types with as of yet unknown (i.e. unification variable) implementations. In concert with rule RAXIOM these axioms are the key to avoiding double vision in E. For the axiom scope to make sense the elaborated expression has the corresponding local axiom declarations. The effect is the effect of E joined with the **type generation effect** $\iota(\overline{\alpha})$ which is P if $\overline{\alpha}$ is empty and I otherwise.

7.4.3 Definition Typing

Sequences of definitions need to be typed for `let`, record expressions and at the top level. The definition sequence typing rules are shown in Figure 7.17.

The lookup rules have all the logic required to compute the type of the first definition.

ValAnn

$$\frac{\Gamma \vdash def(x) \Rightarrow \overline{\alpha}\,.\,\Sigma \dashv \Theta \qquad \Theta\;{}_{\Theta}{\vdash} E_x \Leftarrow_{\iota_x} \overline{\alpha}\,.\,\Sigma \dashv \Theta' \leadsto e_x \qquad \Theta'\;{}_{\Theta'}{\vdash} \overline{y(:T)? = E} \Rightarrow_{\iota} \dashv \Delta \leadsto \overline{y = e}}{\Gamma \vdash x : T_x = E_x ; \overline{y(:T)? = E} \Rightarrow_{\iota_x \vee \iota} \dashv \Delta \leadsto x = e_x ; \overline{y = e}}$$

Val

$$\frac{\Gamma \vdash def(x) \Rightarrow_{\iota_x} \Sigma, e_x \dashv \Theta \qquad \Theta\;{}_{\Theta}{\vdash} \overline{y(:T)? = E} \Rightarrow_{\iota} \dashv \Delta \leadsto \overline{y = e}}{\Gamma \vdash x = E_x ; \overline{y(:T)? = E} \Rightarrow_{\iota_x \vee \iota} \dashv \Delta \leadsto x = e_x ; \overline{y = e}}$$

NoDefs

$$\Gamma \vdash \epsilon \Rightarrow_{\mathsf{P}} \dashv \Gamma$$

Figure 7.17. *Definition typing* $\Gamma \vdash \overline{x(:T)? = E} \Rightarrow_{\iota} \dashv \Delta \leadsto \overline{x = e}$

When the first definition is annotated (ValAnn) the defining expression E_x is checked against the pseudo-type $\overline{\alpha}\,.\,\Sigma_x$. When the first definition is unnanotated (Val) the defining expression has already been typed. Both rules then recursively type the remaining definitions, appending their types and elaborated expressions to the type and definition results and joining the effects. If the definition sequence is empty the obvious base case NoDefs produces empty sequences and a pure effect.

7.4.4 Compilation Units

The type checker entry point is the compilation unit typing $\overline{x(:T)? = E} \Rightarrow \iota \leadsto e$ of Figure 7.18. The input definitions can come from text files or a REPL input line. As in SML and OCaml there is no entry point function even in batch compilation mode, the convention being that the last definition is of the form _ $= E$ where E is evaluated just for side effects as its result is not used.

Unit

$$\frac{\{;\}^{\wedge}, \left\{\overline{x(:^{\circ} T \mid =^{\circ} E)}\right\} \vdash \overline{x(:T)? = E} \Rightarrow_{\iota} \dashv \{\overline{\alpha}; \theta\}^{\wedge}, \Delta \leadsto \overline{x = e}}{\overline{x(:T)? = E} \Rightarrow \iota \leadsto \overline{\mathsf{type}\ \alpha\ \mathsf{in}}\ \mathsf{let\ rec}\ \overline{x = e}\ \mathsf{in}\ \{\}}$$

Figure 7.18. *Typing a compilation unit* $\overline{x(:T)? = E} \Rightarrow \iota \leadsto e$

The definitions are put into scope and typed with the rules of Section 7.4.3 like the local definitions of a `let` in Section 7.4.1. A hoisting scope also needs to be pushed to collect any abstract types generated outside function bodies. The effect of a program is unconstrained (although any useful program will have some sort of I/O side effect).

7.5 Type Matching

This section details the various type matching judgements that have been utilized in type elaboration and term typing. These type matching and conversion operations are the essence of polymorphism and translucency in R1ML.

7.5.1 Focalization and Articulation

As we have seen, R1ML elimination rules rely on focalization. Both the templates and focalizees in the focalization rules of Figure 7.19 are much more varied than in Section 4.5.2.

$$\textsc{FTop}\quad \frac{\Gamma \vdash \Sigma \longrightarrow \Sigma' \rightsquigarrow \gamma \qquad \Gamma \vdash \Sigma' \ggg \underline{\Sigma} \rightsquigarrow \Sigma'' \rightsquigarrow f}{\Gamma \vdash \Sigma \gg \underline{\Sigma} \rightsquigarrow \Sigma'' \rightsquigarrow \lambda x.f\,(x \blacktriangleright \gamma)}$$

$$\textsc{FForall}\quad \frac{\Gamma \uparrow \overline{\hat{\alpha}} \vdash [\overline{\hat{\alpha}/\alpha}]\Sigma \gg \underline{\Sigma} \rightsquigarrow \Sigma' \dashv \Delta \rightsquigarrow f}{\Gamma \vdash \forall\overline{\alpha}.\Sigma \ggg \underline{\Sigma} \rightsquigarrow \Sigma' \dashv \Delta \rightsquigarrow \lambda x.f\,(x\langle\overline{\hat{\alpha}}\rangle)}$$

$$\textsc{FImplicit}\quad \frac{\Gamma \vdash \Sigma \gg \underline{\Sigma} \rightsquigarrow \Sigma' \dashv \Delta \rightsquigarrow f}{\Gamma \vdash \{\} \rightarrow_{\mathsf{A}} \Sigma \ggg \underline{\Sigma} \rightsquigarrow \Sigma' \dashv \Delta \rightsquigarrow \lambda x.f\,(x\,\{\})}$$

$$\textsc{FArrow}\quad \Gamma \vdash \Sigma_d \rightarrow_\iota \Xi_c \ggg \_ \rightarrow_{\_} \_ \rightsquigarrow \Sigma_d \rightarrow_\iota \Xi_c \dashv \Gamma \rightsquigarrow \lambda x.x$$

$$\textsc{FRecord}\quad \Gamma \vdash \{\overline{y : \Sigma_y}, x : \Sigma_x, \overline{y' : \Sigma'_y}\} \ggg \{x : \_, \_\} \rightsquigarrow \{\overline{y : \Sigma_y}, x : \Sigma_x, \overline{y' : \Sigma'_y}\} \dashv \Gamma \rightsquigarrow \lambda x.x$$

$$\textsc{FType}\quad \Gamma \vdash [= \Xi] \ggg [= \_] \rightsquigarrow [= \Xi] \dashv \Gamma \rightsquigarrow \lambda x.x$$

$$\textsc{FArticulate}\quad \frac{\Gamma \vdash \hat{\alpha} :\gg \underline{\Sigma} \rightsquigarrow \Sigma \dashv \Delta}{\Gamma \vdash \hat{\alpha} \ggg \underline{\Sigma} \rightsquigarrow \Sigma \dashv \Delta \rightsquigarrow \lambda x.x}$$

Figure 7.19. *Focalization* $\Gamma \vdash \Sigma \gg \underline{\Sigma} \rightsquigarrow \Sigma \rightsquigarrow f$ *and* $\Gamma \vdash \Sigma \ggg \underline{\Sigma} \rightsquigarrow \Sigma \rightsquigarrow f$

Top level focalization $\gg$ consists of just FTOP, which uses type normalization $\longrightarrow$ to deal with β-redexes and read through axioms in scope and then delegates to the 'worker' focalization $\ggg$. If normalization gets stuck and produces $\bot$ focalization will also fail; unlike subtyping or unification, focalization cannot just go work on other things which would hopefully cause a retry to 'get unstuck'.

FFORALL instantiates a universal type by applying the elaborated Λ to fresh unification variables. Similarly FIMPLICIT removes the domain of an implicit function by applying the elaborated function to an empty record. The FARROW, FRECORD and FTYPE rules just return a type with right toplevel structure on encounter.

In HM unification variables only get instantiated to concrete monotypes. Focalization requires the capacity to add just enough structure to match the template, filling the rest in with fresh unification variables. To achieve that focalization rule FARTICULATE delegates to the **articulation judgement** $:\gg$ of Figure 7.20.

As we shall see shortly in Section 7.5.2, articulation is useful in subtyping as well as focalization. There we just use the sub- or supertype as the articulation template. Using a large type as a template does not threaten predicativity since articulation already has to assume that the template could contain holes and never assigns it to unification variables.

Although this thesis uses the terminology of [9], having separate judgments for general-purpose focalization and articulation is due to [38]. In [9] focalization was only used for and integrated into the application rules and rules for articulation had not been factored out at all.

AARROW
$$\Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\gg \underline{\Sigma_d} \to_\iota \underline{\Sigma_c} \rightsquigarrow \hat{\alpha}_d \to_{\mathsf{I}} \hat{\alpha}_c \dashv [\hat{\alpha} = \hat{\alpha}_d \to_{\mathsf{I}} \hat{\alpha}_c](\Delta \uparrow \hat{\alpha}_d \uparrow \hat{\alpha}_c)$$

ATYPE
$$\Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\gg [= \underline{\Sigma}] \rightsquigarrow [= \hat{\alpha}'] \dashv [\hat{\alpha} = [= \hat{\alpha}']](\Gamma \uparrow \hat{\alpha}')$$

AAPP
$$\Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\gg \underline{\Sigma_f}\, \overline{\underline{\Sigma_a}} \rightsquigarrow \hat{\alpha}_f\, \overline{\hat{\alpha}_a} \dashv [\hat{\alpha} = \hat{\alpha}_f\, \overline{\hat{\alpha}_a}](\Gamma \uparrow \hat{\alpha}_f \uparrow \overline{\hat{\alpha}_a})$$

ABOOL
$$\Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\gg \mathsf{bool} \rightsquigarrow \mathsf{bool} \dashv [\hat{\alpha} = \mathsf{bool}]\Gamma$$

ARIGID
$$\Gamma[\beta][\hat{\alpha}] \vdash \hat{\alpha} :\gg \beta \rightsquigarrow \beta \dashv [\hat{\alpha} = \beta]\Gamma$$

AUNI
$$\Gamma[\hat{\beta}][\hat{\alpha}] \vdash \hat{\alpha} :\gg \hat{\beta} \rightsquigarrow \hat{\beta} \dashv [\hat{\alpha} = \hat{\beta}]\Gamma$$

AUNIREV
$$\Gamma[\hat{\alpha}][\hat{\beta}] \vdash \hat{\alpha} :\gg \hat{\beta} \rightsquigarrow \hat{\alpha} \dashv [\hat{\beta} = \hat{\alpha}]\Gamma$$

Figure 7.20. *Articulation* $\Gamma \vdash \hat{\alpha} :\gg \underline{\Sigma} \dashv \Delta$

The rules AARROW, ATYPE, AAPP and ABOOL just generate a type whose outer structure matches the template and whose inner parts are filled in with fresh unification variables. Unification variables cannot be articulated to records, because that would require guessing all the labels missing from the template. SELECT in particular only supplies one field in the template. The templates $\underline{\Sigma}$ cover both templates from focalization and types from subtyping. Rigid type variables can just be assigned directly, like `bool`, but have to be in scope at least as high as the unification variable $\hat{\alpha}$. Another unification variable $\hat{\beta}$ is treated similarly but if it is below $\hat{\alpha}$ then $\hat{\beta}$ can be set instead of failing (this is called 'reaching' in [9]).

7.5.2 Subtyping

Like in System F_η and 1ML the subtyping rules in Figures 7.21 and 7.22 compose a coercion function to be called at the subtyping site. The subtyping judgement $\Gamma \vdash \Xi_l <:^{occ} \Xi_r \dashv \Delta \rightsquigarrow f$ requires a an occurs check flag occ in addition to the sub- and supertype. The occurs check flag prevents quadratic behaviour due to redundant occurs checks.

Similarly to focalization the actual subtyping work implemented by the auxiliary $\lesssim:$ judgement is interleaved with normalization in the main subtyping judgement $<:$. The normalizations return coercion types that can be used to reveal in-scope implementations of abstract types within the sub- and supertypes. When both normalizations succeed (STOP) the subtype coercion type γ_l is used in the coercion function to cast the subtype before applying the coercion function f from $\lesssim:$ and the result is abstracted into the supertype by casting with a sym-reversed γ_r.

The crux of universal subtyping is unchanged from Section 4.5 but in SFORALL it has

STOP

$$\frac{\Gamma \vdash \Xi_l \longrightarrow \Xi'_l \leadsto \gamma_l \qquad \Gamma \vdash \Xi_r \longrightarrow \Xi'_r \leadsto \gamma_r \qquad \Gamma \vdash \Xi'_l \lesssim:^{occ} \Xi'_r \dashv \Delta \leadsto f}{\Gamma \vdash \Xi_l <:^{occ} \Xi_r \dashv \Delta \leadsto (x \blacktriangleright \gamma_l) \blacktriangleright \mathsf{sym}\, \gamma_r}$$

SEXISTS

$$\frac{\Gamma, \{\overline{\alpha}; \overline{\hat{\beta}}\}^\wedge \vdash \Sigma_L <:^{occ} [\overline{\hat{\beta}/\beta}]\Sigma'_R \dashv \Delta, \{\overline{\alpha}; \theta\}^\wedge \leadsto f}{\Gamma \vdash \exists\overline{\alpha}.\Sigma_L \lesssim:^{occ} \exists\overline{\beta}.\Sigma'_R \dashv \Delta \leadsto \lambda x.\ \mathsf{unpack}\langle\overline{\alpha}, y\rangle = x\ \mathsf{in}\ \mathsf{pack}\langle\overline{\hat{\beta}}, f\, y\rangle}$$

SFORALL

$$\frac{\Gamma, \{\overline{\beta}; \overline{\hat{\alpha}}\}^\wedge \vdash [\overline{\hat{\alpha}/\alpha}]\Sigma_L <:^{occ} \Sigma_R \dashv \Delta, \{\overline{\beta}; \theta\}^\wedge \leadsto f}{\Gamma \vdash \forall\overline{\alpha}.\Sigma_L \lesssim:^{occ} \forall\overline{\beta}.\Sigma_R \dashv \Delta \leadsto \lambda x.\Lambda\overline{\beta}.f\,(x\,\overline{\hat{\alpha}})}$$

SIMPLICITR

$$\frac{\Gamma \vdash \Sigma_L <:^{occ} \Sigma_{Rc} \vdash \Delta \leadsto f}{\Gamma \vdash \Sigma_L \lesssim:^{occ} \{\} \to_{\mathsf{A}} \Sigma_{Rc} \dashv \Delta \leadsto \lambda x.\lambda\_.fx}$$

SIMPLICITL

$$\frac{\Gamma \vdash \Sigma_{Lc} <:^{occ} \Sigma_R \dashv \Delta \leadsto f}{\Gamma \vdash \{\} \to_{\mathsf{A}} \Sigma_{Lc} \lesssim:^{occ} \Sigma_R \dashv \Delta \leadsto \lambda x.f(x\,\{\})}$$

Figure 7.21. *Subtyping I:* $\Gamma \vdash \Xi <:^{occ} \Xi \dashv \Delta \leadsto f$ *and* $\Gamma \vdash \Xi \lesssim:^{occ} \Xi \dashv \Delta \leadsto f$

been supplemented with various general R1ML considerations. The instantiation is done with unification variables which removes the need to guess the final instantiation types at this point and indirectly implements the predicativity restriction required for decidability. A new hoisting scope is inserted for the unification variables so that they and also unification variables created within the recursive invocation of $<:$ can unify with the abstract types $\overline{\beta}$.

Existential subtyping in SEXISTS is a kind of mirror image of the universal handling as often happens; universals are *specialized* to a less general type by instantiation while existentials are shown to be *implemented* by some less abstracted type. A more mechanistic rationale is that the reverse behaviour is forced by the form of the coercion function body. Because universal quantifier bodies are always arrows in our semantic types we do not have to deal with instantiation conflicts $\forall\alpha.\exists\beta.\Sigma <:^{occ} \exists\beta'.\forall\alpha'.\Sigma'$ like [10] had to since types of the form $\forall\overline{\alpha}.\exists\overline{\beta}....$ never occur as semantic types.

Because implicit parameters always have type `type` the subtyping of implicit functions is a trivial analogue of universal subtyping. In SIMPLICITL the function is just applied to a unit argument like in focalization and in SIMPLICITR a useless λ wrapper is added. In a system with with nontrivial implicits like Scala or type classes like Haskell SIMPLICITL would have to invoke instance resolution which is akin to a Prolog interpreter – far from trivial.

Functions are still contravariant in their domain and covariant in their codomain. Additionally effect subtyping is invoked to prevent impure functions from being used as pure. Effect subtyping is identical to that of 1ML and not repeated in this chapter. Record subtyping is also familiar, with subsetting of labels and covariant field types. Function and record subtyping conjoin the constraints of their parts, which should be understood to produce an empty constraint $\emptyset$ if both of the conjoined constraints are empty.

SARROW
$$\frac{\Gamma \vdash \Sigma_{Rd} <:^{occ} \Sigma_{Ld} \dashv \Theta \rightsquigarrow f_1 \qquad \Theta\ {}_{\Theta}\!\vdash \Xi_{Lc} <:^{occ} \Xi_{Rc} \dashv \Delta \rightsquigarrow f_2 \qquad \iota_L <: \iota_R}{\Gamma \vdash \Sigma_{Ld} \rightarrow_{\iota_L} \Xi_{Lc} \lesssim:^{occ} \Sigma_{Rd} \rightarrow_{\iota_R} \Xi_{Rc} \dashv \Delta \rightsquigarrow \lambda x.\lambda y.f_2(x(f_1 y))}$$

SRECORD
$$\frac{\Gamma \vdash \Sigma_L <:^{occ} \Sigma_R \dashv \Theta \rightsquigarrow f \qquad \Theta\ {}_{\Theta}\!\vdash \{\rho\} \lesssim:^{occ} \{\overline{x' : \Sigma'_R}\} \dashv \Delta \rightsquigarrow g}{\Gamma \vdash \{\rho[x : \Sigma_L]\} \lesssim:^{occ} \{x : \Sigma_R, \overline{x' : \Sigma'_R}\} \dashv \Delta \rightsquigarrow \lambda x.\mathsf{let}\ y = g\,x\ \mathsf{in}\ \{x = f\,x.x, \overline{x' = y.x'}\}}$$

SEMPTY
$$\Gamma \vdash \{\overline{x : \Sigma}\} \lesssim:^{occ} \{\} \dashv \Gamma \rightsquigarrow \lambda x.\{\}$$

SFORGET
$$\frac{\begin{array}{c}\Gamma[\hat{\alpha}] \vdash \tau' \longrightarrow \hat{\alpha} \rightsquigarrow \gamma \qquad \Gamma = \Gamma', \{\overline{\beta}; \theta[\hat{\alpha}]\}^\wedge, \Gamma'', \{\overline{\alpha}; \theta'\}^\wedge, \Gamma''' \\ \hat{\alpha} \notin \mathsf{fuv}(\tau) \qquad \overline{\hat{\beta}} = \{\hat{\beta} \in \mathsf{fuv}(\tau) \mid \Gamma''[\hat{\beta}]\} \qquad \Delta = \Gamma', \{\overline{\beta}; \theta, \overline{\hat{\beta}}\}^\wedge \qquad \Delta \vdash \lambda\overline{\alpha}.\tau\end{array}}{\Gamma \vdash [= \tau] \lesssim:^{occ} [= \tau'\,\overline{\alpha}] \dashv [\hat{\alpha} = \lambda\overline{\alpha}.\tau]\Delta, (\Gamma'' - \overline{\hat{\beta}}), \{\overline{\alpha}; \theta'\}^\wedge, \Gamma''' \rightsquigarrow \lambda x.x}$$

STYPE
$$\frac{\Gamma \vdash \Xi_l \sim \Xi_r \dashv \Delta \rightsquigarrow \gamma}{\Gamma \vdash [= \Xi_l] \lesssim:^{occ} [= \Xi_r] \dashv \Delta \rightsquigarrow \lambda x.[\Xi'_r]}$$

SBOOL
$$\Gamma \vdash \mathsf{bool} \lesssim:^{occ} \mathsf{bool} \vdash \Gamma \rightsquigarrow \lambda x.x$$

SRIGID
$$\Gamma[\alpha] \vdash \alpha \lesssim:^{occ} \alpha \dashv \Gamma \rightsquigarrow \lambda x.x$$

SREFL
$$\Gamma[\hat{\alpha}] \vdash \hat{\alpha} \lesssim:^{occ} \hat{\alpha} \dashv \Gamma \rightsquigarrow \lambda x.x$$

SSOLVEL
$$\frac{\begin{array}{c}\hat{\alpha} \notin \mathsf{fuv}(\Sigma)\ \text{if}\ occ = \mathsf{check} \\ \Gamma \vdash \hat{\alpha} :\gg \Sigma \rightsquigarrow \tau \vdash \Theta \\ \Theta\ {}_{\Theta}\!\vdash \tau <:^{\mathsf{skip}} \Sigma \dashv \Delta \rightsquigarrow f\end{array}}{\Gamma\,[\hat{\alpha}] \vdash \hat{\alpha} \lesssim:^{occ} \Sigma \dashv \Delta \rightsquigarrow f}$$

SSOLVER
$$\frac{\begin{array}{c}\hat{\alpha} \notin \mathsf{fuv}(\Sigma)\ \text{if}\ occ = \mathsf{check} \\ \Gamma \vdash \hat{\alpha} :\gg \Sigma \rightsquigarrow \tau \vdash \Theta \\ \Theta\ {}_{\Theta}\!\vdash \Sigma <:^{\mathsf{skip}} \tau \dashv \Delta \rightsquigarrow f\end{array}}{\Gamma\,[\hat{\alpha}] \vdash \Sigma \lesssim:^{occ} \hat{\alpha} \dashv \Delta \rightsquigarrow f}$$

Figure 7.22. *Subtyping II*

Rule SFORGET applies at the roots of higher kinded abstract types. The abstract type must also be as of yet unimplemented, i.e. with the callee reducing to an unitialized unification variable. This is exactly the sort of type application that in non-root position causes normalization to return $\bot$.

But here at the root the higher-kinded type can be initialized. First an occurs check $\hat{\alpha} \notin \mathsf{fuv}(\tau)$ is made to prevent cyclic types. The occurs check is unconditional on the occurs flag occ which is meant only for SSOLVEL and SSOLVER. The unification variables $\overline{\hat{\beta}}$ that are free in τ and nested deeper than $\hat{\alpha}$ but shallower than $\overline{\alpha}$ are lifted to prevent abstract types in Γ'' and Γ''' from later escaping their scope. The implementation type $\lambda\overline{\alpha}.\tau$ must then be well-formed in the new upper context Δ of $\hat{\alpha}$. Having procured all these guarantees the abstract type can finally be initialized to a type-level lambda abstraction $\lambda\overline{\alpha}.\tau$ of the implementation type τ. In practice the occurs check, unification variable lifting and well-formedness check can be optimized to one non-orthogonal traversal of τ.

Unusually the λ parameters in SFORGET *must capture* the free occurrences of the abstract types $\overline{\alpha}$ in τ. Declaratively it should be possible for unification variables in τ that are bound in θ' or Γ''' to be later initialized to types containing references to $\overline{\alpha}$. However that would require being able to capture those $\overline{\alpha}$ references "after the fact", which would be prohibitively complicated to implement (if possible at all). Because SFORGET does not lift the unification variables bound in θ' or Γ''', such corner cases will result in type errors. This **capture restriction** is a source of incompleteness for R1ML. A somewhat less incomplete option would be to lift those unification variables, which would cause a type error only if the retroactive type variable capture would later be required. Unfortunately the delayed type error would be less specific and not indicative of the source of the problem.

Type carriers that are not the roots of higher kinded abstract types are handled *invariantly* by unifying the carried types. Due to limitations in its polytype unification 1ML uses mutual subtyping here, much like [31] did for multi-branch expressions. In case Ξ_l is a unification variable the unification is equivalent to SFORGET without type level functions.

Subtyping for booleans is still syntactic equality. Equal type and unification variables are subtypes of themselves as long as they are in scope (SRIGID and SREFL).

SSOLVEL and SSOLVER articulate the unification variable to match the outer structure of the other type Σ and then continue subtyping with the articulated type τ replacing the unification variable. To avoid cycles they need to do an occurs check at the first articulation but recursive invocations of these rules concern fresh unification variables created in articulation so the occurs check is unnecessary. The unnecessary occurs checks would make subtyping take quadratic time so the occ flag has been introduced to avoid that. 1ML does the unnecessary occurs checks while [9] avoids them but, lacking a separate articulation judgement, duplicates most of subtyping in its 'instantiation' rules.

7.5.3 Unification

Unification [36] is a generalization of equivalence which can also initialize unification variables to *make* its arguments equal. Like GHC's OutsideIn [46] constraint solver our unification can additionally make use of available explicit type equalities (derived from the axioms in scope) by producing a coercion type as a witness. However the flow of our unification is modeled after coercion subtyping instead of GHC's constraint solver. The unification rules in Figures 7.23 and 7.24 are mostly like restricted versions of the corresponding subtyping rules and produce a System F_c coercion *type* instead of a coercion *function*.

Like subtyping (and focalization) unification interleaves normalization in UTOP and UDELAY with actual unification work in an auxiliary $\approx$ judgement. The coercion types can be composed directly into the result unlike in subtyping where they have to be incorporated into casts in the coercion function. UDELAY produces a unification constraint along with a unification variable for patching the coercion type after solution of the constraint.

Rules UEXISTS and UFORALL implement polytype unification, relied upon by IFSYNTH

UTOP
$$\frac{\Gamma \vdash \Xi_l \longrightarrow \Xi_l' \rightsquigarrow \gamma_l \qquad \Gamma \vdash \Xi_r \longrightarrow \Xi_r' \rightsquigarrow \gamma_r \qquad \Gamma \vdash \Xi_l' \approx \Xi_r' \vdash \Delta \rightsquigarrow \gamma}{\Gamma \vdash \Xi_l \sim \Xi_r \dashv \Delta \rightsquigarrow \gamma_l \circ \gamma \circ \mathsf{sym}\gamma_r}$$

UEXISTS
$$\frac{\Gamma, \{\overline{\alpha}; \overline{\hat{\beta}}\}^\wedge \vdash \Sigma_l \sim [\overline{\hat{\beta}/\beta}]\Sigma_r \dashv \Delta, \{\overline{\alpha}; \theta\}^\wedge \rightsquigarrow \gamma \qquad \mathsf{img}(\theta) = \mathsf{dedup}(\mathsf{img}(\theta)) \subseteq \overline{\alpha} \cup \overline{\hat{\beta}}}{\Gamma \vdash \exists\overline{\alpha}.\Sigma_l \approx \exists\overline{\beta}.\Sigma_r \dashv \Delta \rightsquigarrow \exists\overline{\alpha}.\gamma}$$

UFORALL
$$\frac{\Gamma, \{\overline{\beta}; \overline{\hat{\alpha}}\}^\wedge \vdash [\overline{\hat{\alpha}/\alpha}]\Sigma_l \sim \Sigma_r \dashv \Delta, \{\overline{\beta}; \theta\}^\wedge \rightsquigarrow \gamma \qquad \mathsf{img}(\theta) = \mathsf{dedup}(\mathsf{img}(\theta)) \subseteq \overline{\beta} \cup \mathsf{dom}(\theta)}{\Gamma \vdash \forall\overline{\alpha}.\Sigma_l \approx \forall\overline{\beta}.\Sigma_r \dashv \Delta \rightsquigarrow \forall\overline{\alpha}.\gamma}$$

UIMPLICIT
$$\frac{\Gamma \vdash \Sigma_l \sim \Sigma_r \dashv \Delta \rightsquigarrow \gamma}{\Gamma \vdash \{\} \rightarrow_{\mathsf{A}} \Sigma_l \approx \{\} \rightarrow_{\mathsf{A}} \Sigma_r \dashv \Delta \rightsquigarrow \{\} \rightarrow \gamma}$$

Figure 7.23. *Unification I:* $\Gamma \vdash \Xi \sim \Xi \dashv \Delta \rightsquigarrow \gamma$ *and* $\Gamma \vdash \Xi \approx \Xi \dashv \Delta \rightsquigarrow \gamma$

as well as STYPE. The problem with quantifier unifications like these is that types that differ only in the order of the quantified variables or the addition of unused variables should be considered equivalent since those differences have no runtime effect. This is especially important in an elaborating system like this where quantifiers are merely a hidden implementation detail. At the very least the names of the quantified variables should not matter.

In a declarative type system the naming issue can be solved by just appealing to the Barendregt convention, but that does not directly lead to an algorithm. Here I have attempted to solve the unification of quantifiers by instantiating one of the types like in subtyping. However since unification should not actually instantiate quantifiers the unification variables are required to only unify with the quantified type variables from the other side or stay uninitialized to accommodate unused type variables on the instantiated side ($\mathsf{img}(\theta) \subseteq \overline{\beta} \cup \mathsf{dom}(\theta)$). Each unification variable must also unify with a different abstract type ($\mathsf{img}(\theta) = \mathsf{dedup}(\mathsf{img}(\theta))$). The substitution is not hoisted to prevent the unification variables from unifying with something forbidden later. All this should solve the naming, ordering and unused type variable issues without drifting from unification into subtyping.

Implicit functions cannot be implicitly applied or inserted in unification (UIMPLICIT). Indeed they behave like regular functions aside from always having the same domain and no effect (recalling that A is not an effect).

Function, record and type carrier unification is mostly straightforward recursion. Unlike in subtyping function effects must be equal and width subtyping of records is not permitted in rule UEMPTY.

Unsurprisingly `bool` unifies with itself. Syntactically equal type and unification variables also unify as long as they are in scope.

UARROW
$$\frac{\Gamma \vdash \Sigma_{Rd} \sim \Sigma_{Ld} \dashv \Theta \rightsquigarrow \gamma_d \qquad \Theta\,{}_{\Theta}\!\vdash \Sigma_{Lc} \sim\ \Sigma_{Rc} \dashv \Delta \rightsquigarrow \gamma_c}{\Gamma \vdash \Sigma_{Ld} \to_\iota \Sigma_{Lc} \approx \Sigma_{Rd} \to_\iota \Sigma_{Rc} \dashv \Delta \rightsquigarrow \gamma_d \to \gamma_c}$$

URECORD
$$\frac{\Gamma \vdash \Sigma_L \sim \Sigma_R \dashv \Theta \rightsquigarrow \gamma \qquad \Theta\,{}_{\Theta}\!\vdash \{\overline{y : \Sigma_y}, \overline{y' : \Sigma_{y'}}\} \approx \{\overline{x' : \Sigma'_R}\} \dashv \Delta \rightsquigarrow \{\overline{x' : \gamma'}\}}{\Gamma \vdash \{\overline{y : \Sigma_y}, x : \Sigma_L, \overline{y' : \Sigma_{y'}}\} \approx \{x : \Sigma_R, \overline{x' : \Sigma'_R}\} \dashv \Delta \rightsquigarrow \{x = \gamma, \overline{x' = \gamma'}\}}$$

UEMPTY
$$\Gamma \vdash \{\} \approx \{\} \dashv \Gamma \rightsquigarrow \{\}$$

UTYPE
$$\frac{\Gamma \vdash \Xi_l \sim \Xi_r \dashv \Delta \rightsquigarrow \gamma}{\Gamma \vdash [= \Xi_l] \approx [= \Xi_r] \dashv \Delta \rightsquigarrow [= \gamma]}$$

UBOOL
$$\Gamma \vdash \mathsf{bool} \approx \mathsf{bool} \vdash \Gamma \rightsquigarrow \mathsf{bool}$$

URIGID
$$\Gamma[\alpha] \vdash \alpha \approx \alpha \vdash \Gamma \rightsquigarrow \alpha$$

UREFL
$$\Gamma[\hat\alpha] \vdash \hat\alpha \approx \hat\alpha \dashv \Gamma \rightsquigarrow \hat\alpha$$

UINSTANTIATEL
$$\frac{\hat\alpha \notin \mathsf{fuv}(\tau) \qquad \overline{\hat\beta} = \{\hat\beta \in \mathsf{fuv}(\tau) \mid \Gamma'[\hat\beta]\} \qquad \Delta = \Gamma, \{\overline{\beta}; \theta, \overline{\hat\beta}\}^\wedge \qquad \Delta \vdash \tau}{\Gamma, \{\overline{\beta}; \theta[\hat\alpha]\}^\wedge, \Gamma' \vdash \hat\alpha \approx \tau \dashv [\hat\alpha = \tau]\Delta, (\Gamma' - \overline{\hat\beta}) \rightsquigarrow \tau}$$

UINSTANTIATER
$$\frac{\hat\alpha \notin \mathsf{fuv}(\tau) \qquad \overline{\hat\beta} = \{\hat\beta \in \mathsf{fuv}(\tau) \mid \Gamma'[\hat\beta]\} \qquad \Delta = \Gamma, \{\overline{\beta}; \theta, \overline{\hat\beta}\}^\wedge \qquad \Delta \vdash \tau}{\Gamma, \{\overline{\beta}; \theta[\hat\alpha]\}^\wedge, \Gamma' \vdash \tau \approx \hat\alpha \dashv [\hat\alpha = \tau]\Delta, (\Gamma' - \overline{\hat\beta}) \rightsquigarrow \tau}$$

Figure 7.24. *Unification II*

The initialization of unification variables in UINSTANTIATEL and UINSTANTIATER is basically an easier version of SFORGET. An occurs check $\hat\alpha \notin \mathsf{fuv}(\tau)$ is made, the unification variables $\overline{\hat\beta}$ below the level of $\hat\alpha$ are lifted and the type must be well scoped in the new upper context Δ of $\hat\alpha$ before finally initializing $\hat\alpha$.

8 METATHEORY OF R1ML

It turns out that a fair amount of careful analysis is required to avoid false and embarrassing claims of type soundness for programming languages.

Luca Cardelli

This section investigates the soundness, decidability and completeness of R1ML. These basic metatheoretic properties were defined way back in Section 4.1.1. As usual, detailed proofs are not provided because they would be very long and tedious.

Actually I would go as far as to argue that to be really convincing such proofs would need to be mechanized in a proof assistant like Coq as was done in [40]. But at this point I do not have the time to learn Coq and then write 13,000-line proof script.

8.1 Soundness

Usually the soundness of a type system is proven by proving **progress** and **preservation** [32, sec. 8.3]. Progress means that a normalization step applies to any well-typed expression and preservation means that the normalization step will produce a term of the same type.

Soundness is handled differently in elaborating type systems like R1ML. Given the soundness of the target language (System F_c in this case) what remains to be proven is that all elaborated terms are well-typed and all elaborated types well-kinded in the target system:

Theorem 8.1 (Correctness of R1ML elaboration)**.** *Let Γ (with scopes flattened out) be a well-formed F_c environment.*

1. *If* $\Gamma \vdash T \rightsquigarrow \Xi \dashv \Delta$*, then* $\Delta \vdash [\Delta]\Xi : \mathsf{type}$.
2. *If* $\Gamma \vdash E \Rightarrow_\iota \Sigma \dashv \Delta \rightsquigarrow e$ *or* $\Gamma \vdash E \Leftarrow_\iota \Sigma \dashv \Delta \rightsquigarrow e$*, then* $\Delta \vdash e : [\Delta]\Sigma$*.*
3. *If* $\Gamma \vdash \overline{x(: T)? = E} \Rightarrow_\iota \dashv \Delta[x : \Sigma] \rightsquigarrow \overline{x = e}$*, then* $\overline{\Delta \vdash e : \Sigma}$*.*
4. *If* $\Gamma \vdash \Xi <:^{occ} \Xi' \dashv \Theta \rightsquigarrow f$ *and* $\Gamma \vdash \Xi : \mathsf{type}$ *and* $\Gamma \vdash \Xi' : \mathsf{type}$ *then* $\Delta \vdash f : [\Delta]\Xi \to [\Delta]\Xi'$*.*
5. *If* $\Gamma \vdash \Xi \sim \Xi' \dashv \Theta \rightsquigarrow \gamma$ *and* $\Gamma \vdash \Xi : \mathsf{type}$ *and* $\Gamma \vdash \Xi' : \mathsf{type}$*, then* $\Delta \vdash \gamma : [\Delta]\Xi \sim [\Delta]\Xi'$*.*
6. *If* $\Gamma \vdash \Sigma \gg \underline{\Sigma} \rightsquigarrow \Sigma' \dashv \Delta \rightsquigarrow f$ *and* $\Gamma \vdash \Sigma : \mathsf{type}$ *and* $\Gamma \vdash \Sigma' : \mathsf{type}$*, then* $\Delta \vdash f : [\Delta]\Sigma \to [\Delta]\Sigma'$*.*

7. *If $\Gamma \vdash \hat{\alpha} :\gg \underline{\Sigma} \rightsquigarrow \Sigma \dashv \Delta$, then Σ has the same outer shape as $\underline{\Sigma}$.*
8. *All generated axioms are consistent.*

We also needed to prove that coercion functions had the expected type and coercion types the expected equality kind, although those can be regarded as special cases of well-typed terms and well-kinded types. The requirement for consistent axioms is unique to F_c.

Because elaboration is correct and F_c is sound given consistent axioms, R1ML is also sound:

Theorem 8.2 (Soundness of R1ML typing)**.** *If $\epsilon \vdash E \Rightarrow_\iota \Sigma \rightsquigarrow e$, then either $e \uparrow$ or $e \hookrightarrow^* v$ such that $\epsilon \vdash v : \Sigma$ and v is a value.*

8.2 Decidability

Articulation is non-recursive so it is not even potentially nonterminating. All recursive judgements except normalization, subtyping and unification are syntax-directed and inductive, which guarantees termination.

Normalization incorporates β-reduction in rule RAβ. In untyped lambda-calculus that can lead to nontermination. However as in F_ω our type level functions are essentially simply-typed lambda-calculus, which is strongly normalizing. And all R1ML contexts are well-formed since type elaboration only produces well-kinded types, so normalization will also terminate.

To prove termination of subtyping and unification some measures for semantic types are needed. These weight functions are defined in Figure 8.1.

Roughly, S is the number of abstract syntax tree nodes in the type and Q is the number of quantified variables. They can be combined into a total weight W. Weights can be added pointwise ($\langle Q, S\rangle + \langle Q', S'\rangle = \langle Q + Q', W + W'\rangle$ and compared lexicographically ($\langle Q, S\rangle < \langle Q', S'\rangle = Q < Q' \vee (Q = Q' \wedge S < S')$). A couple of key lemmas can be stated in terms of weights:

Lemma 8.3 (Weight reduction under substitution)**.**

1. $Q[\![[\theta]\Xi]\!] = Q[\![\Xi]\!]$
2. $W[\![[\theta]\Xi]\!] < \langle 1, 0\rangle + W[\![[\theta]\Xi]\!]$
3. $W[\![[\theta]\Xi]\!] \leq \langle |\theta|, 0\rangle + W[\![[\theta]\Xi]\!]$

Lemma 8.4 (Resolution progress)**.**

1. *Let Γ be a well-formed environment, $\Gamma \vdash \Xi$ and $\Gamma \vdash \Xi'$ and $\Gamma \vdash \Xi <:^{occ} \Xi' \dashv \Delta \rightsquigarrow f$, then $|\mathsf{unsolved}(\Delta)| - |\mathsf{unsolved}(\Gamma)| \leq Q[\![\Xi]\!] + Q[\![\Xi']\!]$.*
2. *Let Γ be a well-formed environment, $\Gamma \vdash \Xi$ and $\Gamma \vdash \Xi'$ and $\Gamma \vdash \Xi \sim \Xi' \dashv \Delta \rightsquigarrow \gamma$, then $|\mathsf{unsolved}(\Delta)| < |\mathsf{unsolved}(\Gamma)|$ or $|\mathsf{unsolved}(\Delta)| = |\mathsf{unsolved}(\Gamma)| = 0$.*

$$\begin{aligned}
S[\![\exists\overline{\alpha}.\Sigma]\!] &= S[\![\Sigma]\!] \\
S[\![\forall\overline{\alpha}.\Sigma \to_\iota \Xi]\!] &= 1 + S[\![\Sigma]\!] + S[\![\Xi]\!] \\
S[\![\forall\overline{\alpha}.\{\} \to_{\mathsf{A}} \Sigma]\!] &= 1 + S[\![\Sigma]\!] \\
S[\![\{\overline{x : \Sigma}\}]\!] &= 1 + \sum \overline{S[\![\Sigma]\!]} \\
S[\![[= \Xi]]\!] &= 1 + S[\![\Xi]\!] \\
S[\![\lambda\overline{\alpha}.\tau]\!] &= 1 + S[\![\tau]\!] \\
S[\![\tau_f\,\overline{\tau_a}]\!] &= 1 + S[\![\tau_f]\!] + \sum \overline{S[\![\tau_a]\!]} \\
S[\![\alpha]\!] &= 1 \\
S[\![\hat{\alpha}]\!] &= 1 \\
S[\![\mathsf{bool}]\!] &= 1
\end{aligned}$$

$$\begin{aligned}
Q[\![\exists\overline{\alpha}.\Sigma]\!] &= |\overline{\alpha}| + Q[\![\Sigma]\!] \\
Q[\![\forall\overline{\alpha}.\Sigma \to_\iota \Xi]\!] &= |\overline{\alpha}| + Q[\![\Sigma]\!] + Q[\![\Xi]\!] \\
Q[\![\forall\overline{\alpha}.\{\} \to_{\mathsf{A}} \Sigma]\!] &= |\overline{\alpha}| + Q[\![\Sigma]\!] \\
Q[\![\{\overline{x : \Sigma}\}]\!] &= \sum \overline{Q[\![\Sigma]\!]} \\
Q[\![[= \Xi]]\!] &= Q[\![\Xi]\!] \\
Q[\![\lambda\overline{\alpha}.\tau]\!] &= 0 \\
Q[\![\tau_f\,\overline{\tau_a}]\!] &= 0 \\
Q[\![\alpha]\!] &= 0 \\
Q[\![\hat{\alpha}]\!] &= 0 \\
Q[\![\mathsf{bool}]\!] &= 0
\end{aligned}$$

$$W[\![\Xi]\!] = \langle Q[\![\Xi]\!], S[\![\Xi]\!]\rangle$$

Figure 8.1. *Type weights*

In subtyping and unification the weight $W[\![\Xi]\!] + W[\![\Xi']\!] + \langle \mathsf{fuv}(\Xi) + \mathsf{fuv}(\Xi'), 0\rangle$ gets smaller for all premises. SSOLVEL and SSOLVER use articulation which increases the weight by adding unification variables but the following subtyping recursion will immediately solve the new unification variables.

The weight reduction in subtyping and unification shows that they are inductive after all and thus decidable. So in conclusion:

Theorem 8.5 (Decidability of R1ML elaboration)**.** *All R1ML elaboration judgements are decidable.*

8.3 Completeness

Since this thesis never defined a declarative version of the R1ML type system[1], it is not possible to formulate precise statements about completeness. However R1ML surely inherits the incompleteness issues of 1ML. Unification variables cannot handle width subtyping of records, in particular articulation fails on record types. There are also some rather obscure issues with the combination of functors and the value restriction. But as the 1ML paper points out, SML already had both issues (although not exactly since it does not have record subtyping).

The capture restriction in SFORGET is another source of incompleteness for R1ML. It can be triggered by certain combinations of applicative functors and implicit functions. Although the capture issue was not identified in 1ML [38], I suspect a similar restriction should actually be added to 1ML as well.

[1]For space reasons.

9 FUTURE WORK

1ML, as shown here, is but a first step. There are many possible improvements and extensions.

1ML – Core and Modules United [38]

With regard to this thesis I have achieved what I set out to do: an algorithmic type system for recursive first-class modules. In the bigger picture R1ML is not a language that is ready to be used tomorrow, lacking essential practical features and an implementation beyond a sketchy type checker prototype[1]. I also believe that engaging with some further research questions would result in a substantially more expressive design.

9.1 Declarative System

As remarked in the previous chapter, R1ML currently lacks a declarative type system. A declarative type system can act as the specification for typechecking in language definitions. It is mostly useful for determining the completeness of type checking algorithms. Almost every programmer relies on their internalized informal model of typing instead of a formal declarative type system. Indeed most practical programming languages do not even have formal semantics, static (declarative type system) or dynamic (evaluation), with SML being a notable exception.

The idea of a declarative elaborating type system is somewhat ill-defined, since semantic types and elaboration seem very much like algorithmic implementation details. In 1ML the declarative system seems to mean the use of non-syntax-directed generalization and instantiation rules and guessing of monotypes instead of introducing unification variables.

9.2 Practicality

As usual for research type system languages, R1ML is far from a complete language design. In fact the system presented in this thesis is not practical at all since it lacks basic facilities like integers and strings. At least first class recursive modules forced it to fully support recursive functions, parametric polymorphism and records.

Any ML dialect should have algebraic datatypes and pattern matching. Those features

[1]Available at `https://github.com/nilern/r1ml`.

are largely orthogonal to modules although pattern matching on modules with type members would require pattern matching extensions similar to existential datatypes [17] and the more advanced Generalized Algebraic Datatypes (GADTs) [10]. Even GADTs could be added since they are already implemented on top of System F_c in GHC [46].

Unlike the R1ML prototype, serious compilers should use surface syntax to hide the implementation detail of semantic types in type error messages. That should not be very difficult or theoretically interesting. However even non-elaborating type checkers tend to show too large or otherwise unintuitive types if compiler writers do not prioritize the quality of error messages.

9.3 Well-Founded Recursion

In SML and OCaml the well-foundedness of recursive definitions is guaranteed by syntactic restrictions. Those restrictions cannot be lifted without some other way to prevent forward references that access uninitialized values.

Using dynamic initialization checks like Scheme or the recursive modules of OCaml technically recovers soundness. But like Java's **ClassCastException**, that solution still leaves room for paranoia. It would also prevent the tracking of exceptions in an extended effect system (although we could designate a special untracked class of exceptions like Java's `RuntimeError` which could also include `OutOfMemory` exceptions and the like).

Dreyer has presented a type system for well-founded recursion [5]. However I know of no type inference algorithm for that system. I suspect that flow analysis like those used for definite assignment in Java [13, ch. 16] would be a more suitable solution.

9.4 Recursive Linking

This thesis and various preceding papers have solved the double vision problem which concerns type abstraction in the presence of recursive modules. There is also the problem of module abstraction (functors) in the presence of recursive modules. Functor fixpoints are the natural way to handle that and present no problems to type checking (assuming sufficient type annotations).

Unfortunately the recursion in functor fixpoints is not well-founded. Dreyer [5] provides a patchwork solution with explicit single-assignment references and a kind of parameter strictness effect. MixML argues that functors are fundamentally at odds with recursive linking and should be replaced with mixin modules, but it would be a shame to abandon the elegant unification of functors and functions, generativity and computational purity of 1ML. Lazy evaluation would sidestep the whole initialization issue by replacing access to uninitialized values with divergence, but it seems strange to treat premature access as an infinite loop.

Programming language features should be as few and general as possible. On the other hand it does not seem prudent to make pervasive changes to the ML module system just because of the rather obscure issues with functor fixpoints.

9.5 Row Typing

Using width subtyping for fields is natural for modules but makes type inference incomplete for records. It could be argued that this is not that bad since Standard ML does not even have record subtyping and still has incomplete type inference for records. Or width subtyping could be replaced with row inference [19]. That might be convenient for records but incomplete for modules since the predicativity restriction would also concern row variables.

9.6 Effect System

R1ML inherits 1ML's effect system. As explained in Chapter 5 effect-typing is necessary to prevent abstraction leaks from applicative functors. This very coarse-grained effect system (just P or I) and does not support effect polymorphism. If we are going to have an effect system, it should track individual side effects like mutation and exceptions separately, perhaps using row types for this as well like Koka [21] and Multicore OCaml. Incidentally the **operation signatures** of custom algebraic effects seem like a use case for module signatures.

A practical effect system should provide **effect polymorphism** to avoid code duplication between pure and impure versions of the same code, e.g. `fmap` and `traverse` in Haskell. Supporting effect polymorphism between generative and applicative functors leads to **generativity polymorphism** where the equality of abstract types can depend on runtime effects. Implementing such a type system has turned out to be quite complicated due to the different quantifier structure of pure and impure functors. [37]

As explained in [37], in a language with first class functors effect polymorphism implies generativity polymorphism. Perhaps System F_c type functions could be utilized, potentially using some methodology from the GHC type families implementation [41].

9.7 Impredicative Instantiation

Predicative instantiation prevents abstraction over large types. Arguably it is not often needed. However that may be, in a system of row-typed modules impredicative instantiation could become pervasive.

Despite its infamy, impredicative instantiation by itself is not that problematic. The core issue is just that when unifying a unification variable and a quantified type the decision between solving the unification variable with the quantified type (impredicative instantiation) on one hand and instantiating the quantifier (predicative instantiation) on the other hand is undecidable. For instance HMF [20] just requires type annotations in that ambiguous situation if impredicative instantiation is desired.

What really causes problems (undecidability in the form of nontermination instead of mere ambiguity) is the combination of impredicative instantiation with contravariance. Systems like ML^F and HMF only have invariant type constructors. GHC recently removed contravariance (and covariance) and subsequently was able to finally add impredicative instantiation [42].

But covariance and contravariance are essential to ML module signature matching, so we seem to be stuck with predicativity if we want a decidable system. On the other hand decidability can be imposed in practice with arbitrary recursion limit like in GHC with `UndecidableInstances`. Such arbitrary limits compromise completeness, but in practice one can just increase the limit enough to make a valid program compile; programs that trigger nontermination will hit any finite limit eventually. Also any system with Turing-complete macros is already subject to nonterminating compilation.

So impredicativity can be supported by unprincipled limits in compilers. But it would seem that that the undecidability of contravariant impredicative instantiation is not yet fully understood. Demonstrating that e.g. Turing machines can be encoded as System F_η typechecking problems proves undecidability but offers little insight for recovering decidability. Perhaps there is subtler way towards decidability than the complete removal of either impredicativity (like ML modules) or contravariance (like ML^F and GHC)?

9.8 Implicit Parameters

Like 1ML, R1ML only has implicit arguments of type `type`. Those are trivial to implement if type argument inference is already in place. Adding a resolution step in the style of [30] would enable ad-hoc polymorphism and convenient generic programming similar to Haskell type classes, Scala implicits and expecially the experimental OCaml modular implicits [47].

Since resolution potentially has the computational power of Prolog, it would seem like a substantial and complicated addition. However if one looks closely at the resolution rules of [30], it becomes clear that it is almost identical to our subtyping system. We would just need to add mechanisms for backtracking (i.e. undoing solution of unification variables and application of substitutions) and also for delaying and later triggering implicit resolution.

9.9 Non-Continuations

I suspect that the suggested continuations from 1ML [38] to extend applicative functors or go in the direction of irreducibly dependent types would be too complicated to be worth it. Especially when combined with the more practically enticing extensions towards recursive module abstraction, row inference, algebraic effects, modular implicits and impredicative instantiation.

10 CONCLUSION

Like the Hindley-Milner type system of core ML, the ML module system is a masterpiece of design. Like ALGOL 60, the hierarchical namespacing of structures, the direct representation of interfaces as module signatures, the flexible data abstraction of sealing and the statically typed generic programming of functors made the ML module system "so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors." [14].

On the other hand SML does have its limitations and inelegances. The prenex polymorphism of HM is very limited and the stratification into core and modules results in much duplication of functionality and a boundary that can become an obstacle.

Milner's type inference and MacQueen's modules are elegant and impressive. So is Rossberg's demonstration that they can be unified into 1ML instead of merely layered into SML. This chapter has mentioned my desire to also integrate implicit resolution and algebraic effects with first-class modules. Indeed the only language design statement that inspires me more than the 1ML paper is the customary introduction paragraph of Revised Reports on Scheme:

> "Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today." [43]

First and foremost I would like to make a language like 1ML a practical tool for software engineering instead of a few articles (and now a thesis!) about type theory. Too often research languages do not lead to useful implementations while pragmatic languages largely ignore research.

This thesis has shown a way to typecheck recursive first-class ML modules. Having read the most relevant papers [38] [6] [39] I was sure that I could achieve as much, which made it a usable subject for my master's thesis.

For quite a while now I have been far more intrigued by the truly tough problems encountered on the way. Can functor fixpoints be accommodated elegantly or is MixML the only way forward? Could impredicative instantiation coexist with contravariance? These seem like the very edges of tractability: “In mathematics and computability theory, self-reference (also known as Impredicativity) is the key concept in proving limitations of many systems.” [48].

REFERENCES

[1] A. W. Appel. *Compiling with Continuations*. USA: Cambridge University Press, 2007. ISBN: 052103311X.

[2] D. L. Botlan and D. Rémy. Recasting MLF. *Information and Computation* 207.6 (2009), 726–785. ISSN: 0890-5401. DOI: `https://doi.org/10.1016/j.ic.2008.12.006`. URL: `http://www.sciencedirect.com/science/article/pii/S0890540109000145`.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Jan. 1982), 207–212. DOI: `10.1145/582153.582176`.

[4] O. Danvy and L. R. Nielsen. A first-order one-pass CPS transformation. *Theoretical Computer Science* 308.1 (2003), 239–257. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/S0304-3975(02)00733-8`. URL: `http://www.sciencedirect.com/science/article/pii/S0304397502007338`.

[5] D. Dreyer. A Type System for Well-Founded Recursion. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: Association for Computing Machinery, 2004, 293–305. ISBN: 158113729X. DOI: `10.1145/964001.964026`. URL: `https://doi.org/10.1145/964001.964026`.

[6] D. Dreyer. A Type System for Recursive Modules. *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP '07. Freiburg, Germany: Association for Computing Machinery, 2007, 289–302. ISBN: 9781595938152. DOI: `10.1145/1291151.1291196`. URL: `https://doi.org/10.1145/1291151.1291196`.

[7] D. Dreyer. Recursive type generativity. *Journal of Functional Programming* 17.4-5 (2007), 433–471. DOI: `10.1017/S0956796807006429`.

[8] D. Dreyer, R. Harper and K. Crary. Understanding and Evolving the ML Module System. AAI3166274. PhD thesis. USA, 2005. ISBN: 0542015501.

[9] J. Dunfield and N. R. Krishnaswami. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. *CoRR* abs/1306.6032 (2013).

[10] J. Dunfield and N. R. Krishnaswami. Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types. *CoRR* abs/1601.05106 (2016). arXiv: `1601.05106`. URL: `http://arxiv.org/abs/1601.05106`.

[11] C. Flanagan, A. Sabry, B. F. Duba and M. Felleisen. The Essence of Compiling with Continuations. *SIGPLAN Not.* 28.6 (June 1993), 237–247. ISSN: 0362-1340. DOI: `10.1145/173262.155113`. URL: `https://doi.org/10.1145/173262.155113`.

[12] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek and J. Willcock. A Comparative Study of Language Support for Generic Programming. *SIGPLAN Not.* 38.11 (Oct. 2003), 115–134. ISSN: 0362-1340. DOI: `10.1145/949343.949317`. URL: `https://doi.org/10.1145/949343.949317`.

[13] J. Gosling, B. Joy, G. L. Steele, G. Bracha and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN: 013390069X.

[14] C. A. R. Hoare. *Hints on Programming Language Design.* Tech. rep. Stanford, CA, USA, 1973.

[15] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. USA: Prentice-Hall, Inc., 1978. ISBN: 0131101633.

[16] G. Kuan and D. MacQueen. Efficient Type Inference Using Ranked Type Variables. *Proceedings of the 2007 Workshop on Workshop on ML*. ML '07. Freiburg, Germany: Association for Computing Machinery, 2007, 3–14. ISBN: 9781595936769. DOI: `10.1145/1292535.1292538`. URL: `https://doi.org/10.1145/1292535.1292538`.

[17] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.* 16 (Sept. 1994), 1411–1430. DOI: `10.1145/186025.186031`.

[18] D. Le Botlan and D. Rémy. MLF: Raising ML to the Power of System F. *SIGPLAN Not.* 38.9 (Aug. 2003), 27–38. ISSN: 0362-1340. DOI: `10.1145/944746.944709`. URL: `https://doi.org/10.1145/944746.944709`.

[19] D. Leijen. Extensible records with scoped labels. *Trends in Functional Programming*. Vol. 6. Trends in Functional Programming. Intellect, 2005, 179–194.

[20] D. Leijen. HMF: Simple Type Inference for First-Class Polymorphism. *SIGPLAN Not.* 43.9 (Sept. 2008), 283–294. ISSN: 0362-1340. DOI: `10.1145/1411203.1411245`. URL: `https://doi.org/10.1145/1411203.1411245`.

[21] D. Leijen. *Koka: Programming with Row-Polymorphic Effect Types*. Tech. rep. MSR-TR-2013-79. Aug. 2013. URL: `https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types/`.

[22] X. Leroy. *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA, 1990.

[23] X. Leroy. Applicative Functors and Fully Transparent Higher-Order Modules. *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, 142–153. ISBN: 0897916921. DOI: `10.1145/199448.199476`. URL: `https://doi.org/10.1145/199448.199476`.

[24] X. Leroy. A Modular Module System. *J. Funct. Program.* 10.3 (May 2000), 269–303. ISSN: 0956-7968. DOI: `10.1017/S0956796800003683`. URL: `https://doi.org/10.1017/S0956796800003683`.

[25] D. MacQueen. Modules for Standard ML. *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: Associ-

ation for Computing Machinery, 1984, 198–207. ISBN: 0897911423. DOI: 10.1145/800055.802036. URL: https://doi.org/10.1145/800055.802036.

[26] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17.3 (1978), 348–375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: http://www.sciencedirect.com/science/article/pii/0022000078900144.

[27] R. Milner, M. Tofte and D. Macqueen. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0262631814.

[28] J. C. Mitchell. Polymorphic Type Inference and Containment. *Inf. Comput.* 76.2–3 (Feb. 1988), 211–249. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90009-0. URL: https://doi.org/10.1016/0890-5401(88)90009-0.

[29] C. Okasaki. *Purely Functional Data Structures*. USA: Cambridge University Press, 1998. ISBN: 0521631246.

[30] B. C. Oliveira, T. Schrijvers, W. Choi, W. Lee and K. Yi. The Implicit Calculus: A New Foundation for Generic Programming. *SIGPLAN Not.* 47.6 (June 2012), 35–44. ISSN: 0362-1340. DOI: 10.1145/2345156.2254070. URL: http://doi.acm.org/10.1145/2345156.2254070.

[31] S. Peyton Jones, D. Vytiniotis, S. Weirich and M. Shields. Practical Type Inference for Arbitrary-Rank Types. *J. Funct. Program.* 17.1 (Jan. 2007), 1–82. ISSN: 0956-7968. DOI: 10.1017/S0956796806006034. URL: https://doi.org/10.1017/S0956796806006034.

[32] B. C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091, 9780262162098.

[33] B. C. Pierce and D. N. Turner. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), 1–44. ISSN: 0164-0925. DOI: 10.1145/345099.345100. URL: https://doi.org/10.1145/345099.345100.

[34] D. Rémy. Simple, Partial Type-Inference for System F Based on Type-Containment. *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. Tallinn, Estonia: Association for Computing Machinery, 2005, 130–143. ISBN: 1595930647. DOI: 10.1145/1086365.1086383. URL: https://doi.org/10.1145/1086365.1086383.

[35] D. Rémy and J. Vouillon. Objective ML: A Simple Object-Oriented Extension of ML. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: Association for Computing Machinery, 1997, 40–53. ISBN: 0897918533. DOI: 10.1145/263699.263707. URL: https://doi.org/10.1145/263699.263707.

[36] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12.1 (Jan. 1965), 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253. URL: https://doi.org/10.1145/321250.321253.

[37] A. Rossberg. 1ML with Special Effects. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by S. Lindley, C. McBride, P. Trinder and D. Sannella. Cham: Springer International

Publishing, 2016, 336–355. ISBN: 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_18. URL: https://doi.org/10.1007/978-3-319-30936-1_18.

[38] A. Rossberg. 1ML - Core and modules united. *J. Funct. Program.* 28 (2018), e22.

[39] A. Rossberg and D. Dreyer. Mixin' Up the ML Module System. *ACM Trans. Program. Lang. Syst.* 35.1 (Apr. 2013). ISSN: 0164-0925. DOI: 10.1145/2450136.2450137. URL: https://doi.org/10.1145/2450136.2450137.

[40] A. Rossberg, C. Russo and D. Dreyer. F-ing modules. *Journal of Functional Programming* 24.5 (2014), 529–607. DOI: 10.1017/S0956796814000264.

[41] T. Schrijvers, S. Peyton Jones, M. Chakravarty and M. Sulzmann. Type Checking with Open Type Functions. *ICFP 2008*. Submitted to ICFP'08. Apr. 2008. URL: https://www.microsoft.com/en-us/research/publication/type-checking-with-open-type-functions/.

[42] A. Serrano, J. Hage, S. Peyton Jones and D. Vytiniotis. A quick look at impredicativity. *International Conference on Functional Programming (ICFP'20)*. ACM. ACM, Aug. 2020. URL: https://www.microsoft.com/en-us/research/publication/a-quick-look-at-impredicativity/.

[43] A. Shinn, J. Cowan and A. A. Gleckler. *Revised^7 Report on the Algorithmic Language Scheme*. 2013.

[44] M. Sulzmann, M. Chakravarty, S. Peyton Jones and K. Donnelly. System F with type equality coercions. *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, Jan. 2007, 53–66. ISBN: 1-59593-393-X. URL: https://www.microsoft.com/en-us/research/publication/system-f-with-type-equality-coercions/.

[45] D. Vytiniotis, S. Peyton Jones and T. Schrijvers. Let Should Not Be Generalized. *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '10. Madrid, Spain: Association for Computing Machinery, 2010, 39–50. ISBN: 9781605588919. DOI: 10.1145/1708016.1708023. URL: https://doi.org/10.1145/1708016.1708023.

[46] D. Vytiniotis, S. Peyton jones, T. Schrijvers and M. Sulzmann. Outsidein(x) Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21.4–5 (Sept. 2011), 333–412. ISSN: 0956-7968. DOI: 10.1017/S0956796811000098. URL: https://doi.org/10.1017/S0956796811000098.

[47] L. White, F. Bour and J. Yallop. Modular implicits. *Electronic Proceedings in Theoretical Computer Science* 198 (Dec. 2015), 22–63. ISSN: 2075-2180. DOI: 10.4204/eptcs.198.2. URL: http://dx.doi.org/10.4204/EPTCS.198.2.

[48] Wikipedia contributors. *Self-reference — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-July-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Self-reference&oldid=965391442.

[49] N. Wirth. The programming language Pascal. *Acta informatica* 1.1 (1971), 35–63.