Dependent Types in Practical Programming

Hongwei Xi

September 30th, 1998

Department of Mathematical Sciences
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Thesis Committee:

Frank Pfenning, Chair
Peter Andrews
Robert Harper
Dana Scott

This research was partially sponsored by the Defense Advanced Research Project Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. C533.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

Abstract

Programming is a notoriously error-prone process, and a great deal of evidence in practice has demonstrated that the use of a type system in a programming language can effectively detect program errors at compile-time. Moreover, some recent studies have indicated that the use of types can lead to significant enhancement of program performance at run-time. For the sake of practicality of type-checking, most type systems developed for general purpose programming languages tend to be simple and coarse, and this leaves ample room for improvement. As an advocate of types, this thesis addresses the issue of designing a type system for practical programming in which a notion of dependent types is available, leading to more accurate capture of program invariants with types.

In contrast to developing a type theory with dependent types and then designing upon it a functional programming language, we study practical methods for extending the type systems of existing programming languages with dependent types. We present an approach to enriching the type system of ML with a special form of dependent types, where type index objects are restricted to some constraint domains C, leading to the DML(C) language schema. The aim is to provide for specification and inference of significantly more precise type information compared with the current type system of ML, facilitating program error detection and compiler optimization. A major complication resulting from introducing dependent types is that pure type inference for the resulting system is no longer possible, but we show that type-checking a sufficiently annotated program in DML(C) can be reduced to constraint satisfaction in the constraint domain C. Therefore, type-checking in DML(C) can be made practical for those constraint domains C for which efficient constraint solvers can be provided. We prove that DML(C) is a conservative extension over ML, that is, a valid ML program is always valid in DML(C). Also we exhibit the unobtrusiveness of our approach through many practical examples. As a significant application, we also demonstrate the elimination of array bound checks in real code with the use of dependent types. All the examples have been verified in a prototype implementation of a type-checker for DML(C), where C is some constraint domain in which constraints are linear inequalities on integers. This is another attempt towards refining the type systems of existing programing languages, following the step of refinement types (Freeman and Pfenning 1991).

To my parents,

who have been waiting patiently in general and impatiently at the last moment.

Acknowledgements

Foremost I especially thank my advisor Frank Pfenning for suggesting to me the wonderful thesis topic. His sharp criticism on earlier versions of type-checking algorithms for DML was inspirational to the adoption of the current bi-directional version. DML would have been much less attractive without it. I have thus learned that a program can be trusted only if it can be clearly explained. More importantly, I have also developed a taste for simplicity, which will surely have some profound influence on my future research work. I also thank him for his unusual hospitality and patience in general.

I thank Peter Andrews for teaching me automated theorem proving and providing me with the opportunity to work as a research assistant on TPS, a theorem proving system based on higher-order classic logic. This really brought me a lot of first-hand experience with writing large programs in an untyped programming language (Common Lisp), and thus strongly motivated my research work. I also thank him for his kindness in general.

I feel lucky that I took Robert Harper's excellent course on *type theory and programming languages* in 1994. I have since determined to do research related to promoting the use of types in programming. The use of dependent types in array bound check elimination was partly motivated by a question he raised during my thesis proposal. He also suggested the use of dependent types in a typed assembly language, which seems to be a highly relevant and exciting research direction to follow. I also thank him for his encouragement.

I am also grateful to Dana Scott for his generous and timely help, and for his kindness in general, which I shall always look up to.

Thanks to Peter Lee and George Necula for providing me with interesting examples. Thanks to Rowan Davies and Chad Brown for both helpful technical and interesting non-technical discussions.

Thanks to Feng Tang for her enduring many hardships together. Also thanks to my parents for their being so patient, who once reasonably hoped that I could obtain a Ph.D degree before their retirement. Instead, they have now retired for several years.

Contents

List of Figures

Chapter 1

Introduction

Types play a pivotal rôle in the design and implementation of programming languages. The use of types for catching program errors at compile-time goes back to the early days of FORTRAN. A compelling reason for this practice is briefly explained in the following quote.

> Unfortunately one often pays a price for [languages which impose no discipline of types] in the time taken to find rather inscrutable bugs—anyone who mistakenly applies CDR to an atom in LISP and finds himself absurdly adding a property list to an integer, will know the symptoms. – Robin Milner
>
> *A Theory of Type Polymorphism in Programming* (Milner 1978)

It is also well-known that a well-designed type system such as that of ML (Milner, Tofte, and Harper 1990) can effectively enable the programmer to catch numerous program errors at compile-time. However, there are also various occasions in which many common program errors cannot be caught by the type system of ML. For instance, the error of taking the first element out of an empty list cannot be caught by the type system of ML because it does not distinguish empty lists from non-empty ones.

The use of types for compiler optimization, such as passing types to a polymorphic function to help eliminate boxing and/or tagging objects, is a much more recent discovery. However, the type system of ML is also inadequate in this direction. For instance, it is desirable to express the type of a *safe* array access operation since a compiler can then eliminate run-time array bound checks after type-checking, but it is not clear how to do this in the current type system of ML.

In the rest of this chapter we use concrete examples to illustrate the advantage of enriching the type system of ML with dependent types. We also describe the context in which this thesis exists, and then outline the rest of the thesis.

1.1 Introductory Examples

In this section we present several introductory examples to illustrate the expressiveness of the type system which we will soon formulate and study. We suggest that the reader pay further attention to these examples when studying the theoretical core of the thesis later. Some larger examples can be found in Appendix A.

Notice that a correct implementation of a reverse function on lists should return a list of the same length as that of its argument. Unfortunately, this property cannot be captured by the type

```
datatype 'a list = nil | cons of 'a * 'a list
typeref 'a list of nat (* indexing datatype 'a list with nat *)
with nil <| 'a list(0)
   | cons <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)

fun('a)
    reverse(l) =
      let
          fun rev(nil, ys) = ys
            | rev(cons(x, xs), ys) = rev(xs, cons(x, ys))
          where rev <| {m:nat}{n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
      in rev(l, nil) end
where reverse <| {n:nat} 'a list(n) -> 'a list(n)
```

Figure 1.1: The reverse function on lists

system of ML. The inadequacy can be remedied if we introduce *dependent types*. The example in Figure 1.1 is written in the style of Standard ML with some annotations, which will be explained shortly. We assume that we are working over the constraint domain of natural numbers with constants 0 and 1 and addition operation $+$. The polymorphic datatype `'a list` is defined to represent the type of lists. This datatype is indexed by a natural number, which stands for the length of a list in this case. The constructors associated with the datatype `'a list` are then assigned dependent types:

- `nil <| 'a list(0)` states that `nil` is a list of length 0.
- `cons <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)` states that `cons` yields a list of length $n + 1$ when given a pair consisting of an element and a list of length n. Note that `{n:nat}` means that n is universally quantified over natural numbers, usually written as $\Pi n : nat$.

The use of `fun('a)` is a recent feature of Standard ML (Milner, Tofte, Harper, and MacQueen 1997), which allows the programmer to explicitly control the scope of the type variable `'a`. The type of `reverse` is

```
{n:nat} 'a list(n) -> 'a list(n),
```

which states that `reverse` always returns an a list of length n if given one of length n. In this way, we have captured the information that the function `reverse` is length-preserving. Notice that we have also assigned the auxiliary function `rev` the following dependent type,

```
{m:nat}{n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
```

that is, `rev` always returns a list of length $m + n$ when given a pair of lists of lengths m and n. respectively. This invariant must be provided in order to type-check the entire code.

The next example in Figure 1.2 implements a quicksort function on `intlist`. The datatype `intlist`, which represents an integer list, is indexed by an integer which stands for the sum of all elements in the integer list. The following type of *quicksort*

```
{sum:int} intlist(sum) -> intlist(sum)
```

states that the the sum of all elements in the output `intlist` of the function always equals the sum of all elements in its input `intlist`. Therefore, if one mistakenly writes

`par(x, intCons(x,left),right, ys)` instead of `par(x, intCons(y,left), right, ys)`,

the error, which is not unusual, can be captured in the enriched type system. Notice that this error slips through the current type system of ML.

The above examples exhibit some potential use of dependent types in compile-time program error detection. We now show some potential use of dependent types in compiler optimization. The example in Figure 1.3 implements a binary search function on a polymorphic array. The asserted type of the subscript function `sub` precisely states that it returns an element of type `'a` when given an `'a array` of size n and and an integer equal to i such that $0 \leq i < n$ holds. Clearly, if the subscript function `sub` of this type is called, there is no need for inserting run-time array bound checks for checking possible memory violations. This not only enhances the robustness of the code but also its efficiency, illustrating that safety and efficiency issues can be complementary, sometimes.

Note that the programmer has to provide an appropriate type for the inner function *look* in order to have the code type-checked successfully. We will come back to this point later.

There is a common feature in the above three examples, that is all the type index objects are drawn from the integer constraint domain. The next example in Figure 1.4 and Figure 1.5 shows that we can also index datatypes with type index objects drawn from different constraint domains. Since this example is considerably involved, we present some detailed explanation.

The datatype `simple_type` represents simple types in a simply typed λ-calculus. The datatype type `context` is basically a list of simple types, which is used to assigns types to free variables in a λ-expression. The datatype `lambda_exp` is for formulating simply typed λ-expressions in the *de Bruijn's* notation (de Bruijn 1980). For instance, $\lambda x \lambda y.y(x)$ can be represented as

`Abs(Abs(App(One, Shift(One)))).`

The datatypes `closure` and `enviroment` are defined mutually recursively. An environment is a list of closures and a closure is a λ-abstraction associated with an environment which binds every free variable in the λ-abstraction to some closure.

We now refine some of these datatype types into dependent types in Figure 1.4. The datatype `lambda_exp` is made dependent on a pair `(t,ctx)`, where `t` stands for the simple type of a lambda-expression under the context `ctx`. Then we assign dependent types to the constructors associated with the datatype `lambda_exp`. For instance, the dependent type of `App` states that `App` yields an λ-expression of type `tb` under context `ctx` if given a pair of λ-expressions of types `Fun(ta,tb)` and `ta` under context `ctx`, respectively.

The datatype `closure` is made dependent on an index drawn from the type `simple_type`, which stands for the type of a closure. Also the datatype `environment`, is made dependent on an index drawn from the type `context`, which is a list of simple types corresponding to the list of closures in the environment.

We assign the function `call_by_value` the following type

`{t:simple_type} lambda_exp(t, CTXempty) -> closure(t)`

which states that this function always returns a closure of type `t` when given an argument of type `lambda_exp(t,CTXempty)`, i.e., a *closed* λ-expression of type `t`. This simply means that this

```
datatype intlist = intNil | intCons of int * intlist

typeref intlist of int (* sum *)
with intNil <| intlist(0)
  | intCons <| {i:nat, sum:int} int(i) * intlist(sum) -> intlist(i+sum)

fun intAppend(intNil, rs) = rs
  | intAppend(intCons(l, ls), rs) = intCons(l, intAppend(ls, rs))
where intAppend <| {sm:int, sn:int} intlist(sm) * intlist(sn) -> intlist(sm+sn)

fun quicksort(intNil) = intNil
  | quicksort(intCons(x,xs)) =
    let
        fun par(x, left, right, intNil) =
            intAppend(quicksort(left), intCons(x,quicksort(right)))
          | par(x, left, right, intCons(y,ys)) =
            if y <= x then par(x, intCons(y,left), right, ys)
            else par(x, left, intCons(y,right), ys)
        where par <| {i:int, sp:int, sq:int, sr:int}
                       int(i) * intlist(sp) * intlist(sq) * intlist(sr) ->
                       intlist(i+sp+sq+sr)
    in par(x, intNil, intNil, xs) end
where quicksort <| {sum:int} intlist(sum) -> intlist(sum)
```

Figure 1.2: Quicksort on integer lists

```
datatype 'a answer = NONE | SOME of int * 'a

assert sub <| {n:nat, i:int | 0 <= i < n } 'a array(n) * int(i) -> 'a
assert length  <| {n:nat} 'a array(n) -> int(n)

fun('a){size:nat}
bsearch cmp (key, arr) =
let
    fun look(lo, hi) =
        if hi >= lo then
            let
                val m = (hi + lo) div 2
                val x = sub(arr, m)
            in
                case cmp(key, x) of
                    LESS => look(lo, m-1)
                  | EQUAL => SOME(m, x)
                  | GREATER => look(m+1, hi)
            end
        else NONE
    where look <| {l:nat, h:int | 0 <= l <= size /\ 0 <= h+1 <= size }
                  int(l) * int(h) -> 'a answer
in
    look (0, length arr - 1)
end
where bsearch <| ('a * 'a -> order) -> 'a * 'a array(size) -> 'a answer
```

Figure 1.3: Binary search on arrays

```
datatype simple_type = Base of int | Fun of simple_type * simple_type

datatype context = CTXempty | CTXcons of simple_type * context

datatype lambda_exp =
  One | Shift of lambda_exp |
  Abs of lambda_exp |
  App of lambda_exp * lambda_exp

typeref lambda_exp of simple_type * context
with One <| {t:simple_type}{ctx:context} lambda_exp(t,CTXcons(t,ctx))
   | Shift <| {ta:simple_type}{tb:simple_type}{ctx:context}
              lambda_exp(ta,ctx) -> lambda_exp(ta,CTXcons(tb,ctx))
   | Abs <| {ta:simple_type}{tb:simple_type}{ctx:context}
            lambda_exp(tb,CTXcons(ta,ctx)) ->
            lambda_exp(Fun(ta,tb),ctx)
   | App <| {ta:simple_type}{tb:simple_type}{ctx:context}
            lambda_exp(Fun(ta,tb),ctx) * lambda_exp(ta,ctx) -> lambda_exp(tb,ctx)

datatype closure = Closure of lambda_exp * environment
and environment = ENVempty | ENVcons of closure * environment

typeref closure of simple_type
with Closure <| {t:simple_type}{ctx:context}
                lambda_exp(t,ctx) * environment(ctx) -> closure(t)
and environment of simple_type * context
with ENVempty <| environment(CTXempty)
   | ENVcons <| {t:simple_type}{ctx:context}
                closure(t) * environment(ctx) -> environment(CTXcons(t,ctx))
```

Figure 1.4: A call-by-value evaluator for simply typed λ-calculus (I)

implementation of an evaluator for the pure simply typed call-by-value λ-calculus *á la* Curry typing is a type-preserving function. Clearly, the programmer should have much more confidence in the correctness of the function `call_by_value` after the code passes type-checking.

1.2 Basic Overview

We outline in this section the historic context in which this thesis is developed, and mention some related work in the next section.

The problem of correctness of programs is ever present in programming. There has been a long history of research work on program verification.

The use of assertions to specify and prove correctness of flowchart programs was developed independently by Naur (Naur 1966) and Floyd (Floyd 1967). Hoare then constructed a partial-

```
fun call_by_value(e) =
  let
    fun cbv(One, ENVcons(clo, env)) = clo
      | cbv(Shift(e), ENVcons(clo, env)) = cbv(e, env)
      | cbv(Abs(e), env) = Closure(Abs(e), env)
      | cbv(App(f, e), env) =
        let
          val Closure(Abs(body), fenv) = cbv(f, env)
          val clo = cbv(e, env)
        in
          cbv(body, ENVcons(clo, fenv))
        end
    where cbv <| {t:simple_type, ctx:context}
                 lambda_exp(t,ctx) * environment(ctx) -> closure(t)
  in
    cbv(e, ENVempty)
  end
where call_by_value <| {t:simple_type} lambda_exp(t, CTXempty) -> closure(t)
```

Figure 1.5: A call-by-value evaluator for simply typed λ-calculus (II)

correctness system (Hoare 1969) which brought us *Hoare logic*. Then Dijkstra invented the notion of weakest preconditions (Dijkstra 1975) and explored it in more details, with many examples, in (Dijkstra 1976). As a generalization of the weakest-precondition approach, refinement logics have become an active research area in recent years. These approaches are in general notoriously difficult and expensive to put into software practice. Only small pieces of safety critical software can afford to be formally verified with such approaches. Although rapid progress has been made, there are still strong reservations on whether daily practical programming can benefit much from these approaches. However, these approaches are gaining grounds in the verification of hardware.

For functional programming languages we find two principal styles of reasoning: *equational* and *logical*.

Equational reasoning is performed through program transformation, which has its roots in (Church and Rosser 1936). Burstall and Darlington presented a transformation system for developing recursive programs (Burstall and Darlington 1977). Also we have Bird-Meertens calculus for derivation of functional programs from a specification (Bird 1990), which consists of a set of higher-order functions that operate on lists including map, fold, scan, filter, inits, tails, cross product and function composition. Equational reasoning also plays a fundamental role in FP and EML (Kahrs, Sannella, and Tarlecki 1994).

Logical reasoning is most often cast into type theory, which has its roots in (Church 1940; Martin-Löf 1984). This approach emphasizes the joint development of proofs and programs. Many systems such as NuPrl (Constable et al. 1986), Coq (Coquand 1991), LEGO (Pollack 1994), ALF (Augustsson, Coquand, and Nordström 1990) and PVS (Shankar 1996) are based on some variants of type theory, though this can also be done in a "type-free" setting as shown in PX (Hayashi and Nakano 1988). However, recently it has also been used to generate *post-hoc* proofs and proof

skeletons from functional programs together with specifications (Parent 1995).

There are several major difficulties associated with type-theoretic approaches.

1. Languages tend to be unrealistically small. Although pure type systems (Barendregt 1992) can be formulated concisely and elegantly, they contain too few language constructs to support practical programming.

2. It is unwieldy to add programming features into pure type theories. This is attested in the works such as allowing unlimited recursion (Constable and Smith 1987), introducing recursive types (Mendler 1987), and incorporating effects (Honsell, Mason, Smith, and Talcott 1995), exceptions (Nakano 1994) and input/output.

3. Type-checking is usually undecidable in systems enriched with recursion and dependent types. Therefore, type-checking programs requires a certain level of theorem proving. For systems such as NuPrl and PVS, type-checking is interactive and may often become a daunting task for programmers.

4. It is heuristic at best to do theorem proving by tactics during type-checking, and this requires a lot of user interactions. Since small changes in program may often mean a big change in a proof and there are many changes to be made during the program development cycle, the cost in effort and time often deters the user from programming in such a setting.

Instead, we propose to enrich the type systems of an existing functional programming language (ML). In contrast to adjusting programming language features such as recursion, effects and exceptions to a type theory, we study approaches to adjusting a type theory to these programming language features. We refine ML types with dependent types and introduce a restricted form of dependent types, borrowing ideas from type theory. This enables us to assert additional properties of programs in their types, providing significantly more information for program error detection and compiler optimization. In order to make type-checking manageable in this enriched type system, we require that type dependencies be taken from some restricted constraint domain C, leading to the DML(C) language schema. We then prove that type-checking a sufficiently annotated program in DML(C) can be reduced to constraint satisfaction in the constraint domain C. An immediate consequence of this reduction is that if we choose C to be some relative simple constraint domains for which there are practical approaches to solving constraints, then we can construct a practical type-checking algorithm for DML(C). We will focus on the case where C is some integer constraint domain in which the constraints are linear inequalities on integers.

1.3 Related Work

It is certainly beyond reasonable hope to mention even a moderate part of the research on the correctness of programs. This is simply because of the vastness of the field. We shall examine some efforts which have a close connection to our work, mostly concerning type theories and their applications. We start with Martin-Löf's constructive type theory.

1.3.1 Constructive Type Theory and Related Systems

The system of constructive type theory is based primarily on the work of Per Martin-Löf (Martin-Löf 1985; Martin-Löf 1984). Its core idea often reads *propositions as types*. This is a system which

is *simultaneously* a logic and a programming language. Programs are developed in such a way that they must behave according to their specifications. This is achieved through formal proofs which are written within the programs. The correctness of these proofs is verified by type-checkers.

NuPrl The NuPrl proof system was developed to allow the extraction of programs from the proof of specifications (Constable et al. 1986). Its logical basis is a sequent-calculus formulation of a descendant of constructive type theory. Similarly to LCF it features a goal-oriented proof engine employing tactics formulated in the ML programming language. The emphasis of NuPrl is logical, in that it is designed to support the top-down construction of derivations of propositions in a deduction system.

ALF The ALF (Another Logical Framework) system is an interactive proof editing environment where proof objects for mathematical theorems are constructed on screen. It is based on Martin-Löf's monomorphic type theory (Augustsson, Coquand, and Nordström 1990; Nordström 1993). The proof editor keeps a theory environment, a dictionary with abbreviations and a scratch area. The user navigates in the scratch area to build proofs in top-down and/or bottom-up fashion. A novelty of ALF lies in its use of pattern matching with dependent types (Coquand 1992) for defining functions. The totality of functions defined by pattern matching is guaranteed by some restrictions on recursive equational definitions. This allows the user to formulate significantly shorter proofs in ALF than in many other systems.

1.3.2 Computational Logic PX

Realizability models of intuitionistic formal systems also allow the extraction of computations from the systems. PX is such a system which is introduced in (Hayashi 1990) and described in detail in (Hayashi and Nakano 1988). PX is a logic for a *type-free* theory of computation based on Feferman's T_0 (Feferman 1979), from which LISP programs are extracted by a notion of realizability: *PX-realizability*. Hayashi argues that the requirement that a theory be total is too restrictive for practical programming, in justification of his logic being based around a system of possibly nonterminating computations.

Also Hayashi proposed a type system ATTT in (Hayashi 1991), which allows a notion of refinement types as in the type system for ML (Freeman and Pfenning 1991), plus intersection and union of refinement types and singleton refinement types. He demonstrated that singleton, union and intersection types allow the development of programs without unnecessary coding via a variant of the Curry-Howard isomorphism. More exactly, they give a way to write types as specifications of programs without unnecessary coding which is inevitable otherwise.

1.3.3 The Calculus of Constructions and Related Systems

Calculus of Constructions and Coq The calculus of constructions (CC) is a type system which basically enriches Girard's F_ω with types dependent on terms. It therefore relates to Martin-Löf's intuitionistic theory of types (TT) in this respect. CC was originally developed and implemented by Coquand and Huet (Coquand and Huet 1985; Coquand and Huet 1988). Coquand and Paulin-Mohring proposed to extend CC with primitive inductive definitions (Paulin-Mohring 1993), which led to the calculus of inductive constructions and its implementation in the Coq proof assistant consisting of a proof-checker for CC, a facility called *Mathematical Vernacular* for the high-level

notation of mathematical theories, and an interactive theorem prover based on tactics written in the Caml dialect of the ML language.

Recently, Parent (Parent 1995) proposed to reverse the process of extracting programs from constructive proofs in Coq, synthesizing, *post hoc*, proofs from programs. This approach has a close connection to ours, in that we are trying to use dependent types expressing additional properties of programs which are then verified by a type-checker. Relying on a weak extraction function which produces programs with annotations, Parent introduced a new language for annotated programs and proved that partial proof terms can be deterministically retrieved from given programs in this language and their specifications. Then she showed that such an extraction function is invertible, deducing an algorithm for reconstructing proofs from programs. She also proved the validity and completeness (in a certain sense) of this approach. Programs usually have prohibitively many annotations in the new language, preventing the user from writing sufficiently natural programs. A heuristic algorithm for generating partial proof terms was then proposed and implemented in Coq as a tactic. This tactic builds a partial proof term from a program and a specification, and then the usual Coq tactics are called to fulfill the proof obligations.

ECC and LEGO The Extended Calculus of Constructions (ECC) (Lou 1989) unifies ideas from Martin-Löf's type theory and the Calculus of Constructions. In (Lou 1991) a further extension of the framework by datatypes covered with a general form of schemata is proposed. The LEGO system implements ECC, in which the use of inductive definitions and pattern matching is appealing to practical work on proofs.

1.3.4 Software Model Checking

Model checking is superior to general theorem proving in a few aspects. Model checking need not invent lemmas or devise proof strategies, offering full automation. Also model checking can generate counterexamples when a check fails. Both software specifications and their intended properties can be expressed in a simple relational calculus (Jackson, Somesh, and Damon 1996). The claim that a specification satisfies a property becomes a relational formula that can then be checked automatically by enumerating the formula's interpretations if the number of interpretation is finite. Unfortunately, in software designs, state explosion arises more from the data structures of a single program than from the product of the control states of several programs. The result is that the number of different interpretations for a relational formula is in general vastly too great for brute-force enumerations to be feasible. Even worse, it is quite often the case where such a formula can have infinitely many interpretations. In (Jackson, Somesh, and Damon 1996), it is proposed to reduce the number of cases which a checker must consider by eliminating isomorphic interpretations. This strategy has been successfully tried in hardware verification. Also with great care one needs to downscale the state space of a system, bring it into the reach of a checker. This is based on the assumption that if a bug lies in the original system, then it is likely to cause a bug in the downscaled system. Experience suggests that enumerating all behaviors for the downscaled machine is a more reliable debugging method than exploring merely some cases for the original system.

As we will see, if we choose C to be some finite domain then model checking seems to be a natural approach to solving the constraints generated during type-checking programs in DML(C).

1.3.5 Extended ML

Sannella and Tarlecki proposed *Extended ML* (Sannella and Tarlecki 1989) as a framework for the formal development of programs in a pure fragment of Standard ML. The module system of Extended ML can not only declare the type of a function but also the axioms it satisfies. This leads to the need for theorem proving during type checking. We specify and check less information about functions which avoids general theorem proving. On the other hand, we currently do not address module-level issues, although we believe that our approach should extend naturally to signatures and functors without much additional machinery.

1.3.6 Refinement Types

Tim Freeman and Frank Pfenning proposed refinement types for ML (Freeman and Pfenning 1991). A user-defined ML datatype can be refined into a finite lattice of subtypes. In this extension, type inference is decidable and every well-typed expression has a principal type. The user is free to omit type declaration almost everywhere in a program. A prototype implementation (Freeman 1994) exhibits that this is a promising approach to enriching the type systems of ML. Our thesis work follows the paradigm of refinement types.

1.3.7 Shape Analysis

Jay and Sekanina (Jay and Sekanina 1996) introduced a technique for array bounds checking based on the notion of shape types. Shape checking is a kind of partial evaluation and has very different characteristics and source language when compared to DML(C), where C consists of linear integer equality and inequality constraints. We feel that their design is more restrictive and seems more promising for languages based on iteration schemas rather than general recursion.

1.3.8 Sized Types

Hughes, Pareto and Sabry (Hughes, Pareto, and Sabry 1996) introduced the notion of sized types for proving the correctness of reactive systems. Though there exist some similarities between sized types and datatype refinement in DML(C) for some domain C on natural numbers, the differences seem to be substantial. We feel that the language presented in (Hughes, Pareto, and Sabry 1996) is too restrictive for general purpose programming since the type system there can only handle (a minor variation of) primitive recursion. On the other hand, the use of sized types in the correctness proofs of reactive systems cannot be achieved in DML at this moment.

1.3.9 Indexed Types

So far the most closely related to our work is the system of *indexed types* developed independently by Zenger in his forthcoming Ph.D. Thesis (Zenger 1998) (an earlier version of which is described in (Zenger 1997)). He works in the context of of lazy functional programming. His language is clean and elegant and his applications (which significantly overlap with ours) are compelling. In general, his approach seems to require more changes to a given Haskell program to make it amenable to checking indexed types than is the case for our system and ML. This is particularly apparent in the case of existential dependent types, which are tied to data constructors. This has the advantage of a simpler algorithm for elaboration and type-checking than ours, but the program (and not just

the type) has to be more explicit. Also, since his language is pure, he does not consider a value restriction.

1.3.10 Cayenne

Cayenne (Augustsson 1998) is a Haskell-like language in which fully dependent types are available, that is, language expressions can be used as type index objects. The steep price for this is undecidable type-checking in Cayenne. We feel that Cayenne pays greater attention to making more programs typable than assigning programs more accurate types. In Cayenne, the `printf` in C, which is not typable in ML, can be made typable, and modules can be replaced with records, but the notion of datatype refinement does not exist. This clearly separates our language design from that of Cayenne.

1.4 Research Contributions

The notion of dependent types has been around for at least three decades, but it has not been made applicable to practical programming before. One major obstacle is the difficulty in designing a practical type-checking algorithm for dependent type systems.

The main contribution of this thesis is we convincingly demonstrate the use of a restricted form of dependent types in practical programming. We present a sound and practical approach to extending the type system of ML with dependent types, achieving this through theoretical work, actual implementation and evaluation. The following consists of some major steps which lead to the substantiation of this claim.

1. We separate type index objects from expressions in the programming language. More precisely, we require that type index objects be restricted to some constraint domains C. We then prove that type-checking a sufficiently annotated program in this setting can be reduced to constraint satisfaction in C. It is this crucial decision in our language design which makes type-checking practical in the case where there are feasible approaches to solving constraints in C.

2. We prove that our enriched language is a conservative extension of ML. Therefore, a program which uses no features of dependent types behaves exactly the same as in ML at both compile and run time.

3. We show that dependent types cope well with many important programming features such as polymorphism, mutable references and exceptions.

4. We exhibit the unobtrusiveness of dependent types in practical programming by writing programs as well as by modifying existing ML code. Though the programmer has to provide type annotations in many cases in order to successfully type-check the code, the amount of work is moderate (type annotations usually accounts for less than 20% of the entire code). On the other hand, all type annotations are type-checked mechanically, and therefore they can be fully trusted when used as program documentation.

5. We also demonstrate that the programmer can supply type annotations to safely remove array bound checks. This leads to not only more robust programs but also significantly more efficient code.

In a larger scale, the dependent types also have the following potential applications, for which we will provide illustrating examples.

1. The dependent types in the source code can be passed down to lower level languages. For instance, we are also in the process of designing a *dependently typed assembly language*, in which the dependent types passed down from the source code can be used to generate a proof asserting the memory integrity of the assembly code. Therefore, our source language is promising to act as a front-end for generating proof-carrying code (Necula 1997).

2. The dependent types can facilitate the elimination of redundant matches in pattern matching. On one hand, this can lead to more accurate error or warning message reports during type-checking. One the other hand, this opens an exciting avenue to *dependent type directed* partial evaluation as shown in Section 9.3.2.

1.5 Thesis Outline

The rest of the thesis is organized as follows.

In the next chapter, we start with an *untyped* language which is basically the call-by-value λ-calculus extended with general pattern matching. The importance of this language lies in its operational semantics, to which we will relate the operational semantics of typed languages formulated later. We then introduce a typed programing language ML_0, which is basically mini-ML extended with general pattern matching. We prove various well-known properties of ML_0, which mainly serve as the guidance for our further development. Also we study the operational equivalence relation in $\lambda^{\mathrm{pat}}_{\mathrm{val}}$, which is later needed in the proof of the correctness of elaboration algorithms in Chapters 4 and 5.

The language enriched with dependent types will be parameterized over a constraint domain from which the type index objects are drawn. We introduce a general constraint language in Chapter 3 upon which a constraint domain is formulated. We then present some concrete examples of constraint domains, including the integer domain needed for array bound check elimination.

In Chapter 4, we introduce the notion of *universal dependent types* and extend ML_0 with this form of types. This leads to the programming language $\mathrm{ML}^{\Pi}_0(C)$. We then prove various important properties of $\mathrm{ML}^{\Pi}_0(C)$ and relate its operation semantics to that of ML_0. This culminates with the conclusion that $\mathrm{ML}^{\Pi}_0(C)$ is a conservative extension of ML_0. In order to show the unobtrusiveness of universal dependent types in programming, we also formulate an external programming language $\mathrm{DML}_0(C)$ for $\mathrm{ML}^{\Pi}_0(C)$ which closely resembles that for mini-ML. We then present an elaboration mapping from $\mathrm{DML}_0(C)$ to $\mathrm{ML}^{\Pi}_0(C)$ and prove its correctness.

In Chapter 5, we explain some inadequacies of $\mathrm{ML}^{\Pi}_0(C)$ through examples and introduce the notion of *existential dependent types*. We extend $\mathrm{ML}^{\Pi}_0(C)$ with this form of types and obtain the programming language $\mathrm{ML}^{\Pi,\Sigma}_0(C)$. The external language $\mathrm{DML}_0(C)$ is extended accordingly. The initial development of this chapter is parallel to that of the previous one. However, it seems difficult to find an elaboration mapping from $\mathrm{DML}_0(C)$ to $\mathrm{ML}^{\Pi,\Sigma}_0(C)$ directly. We point out the difficulty and suggest some methods to overcome it. Then an elaboration mapping for $\mathrm{ML}^{\Pi,\Sigma}_0(C)$

is presented and proven to be correct. The theoretical core of the thesis consist of Chapter 4 and 5.

We study combining dependent types with polymorphism in Chapter 6. Though the development of dependent types is largely orthogonal to polymorphism, there are still some practical issues which we must address. We introduce $\mathrm{ML}_0^{\forall}$, a language which extends ML_0 with let-polymorphism, and set up the machinery for combining dependent types with let-polymorphism. Lastly, we present a two-phase elaboration algorithm for achieving full compatibility between $\mathrm{ML}_0^{\forall}$ and $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$, the language which extends $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ with let-polymorphism.

In Chapter 7, we study the interaction of dependent types with effects such as mutable references and exceptions. After spotting the problems, we adopt the *value restriction* approach, which solves these problems cleanly. We conclude with the formulation of a typed programing language $\mathrm{ML}_{0,\mathrm{exc,ref}}^{\forall,\Pi,\Sigma}(C)$ which includes features such as references, exceptions, let-polymorphism and dependent types. In other words, we have finally extended the core of ML, that is, ML without module level constructs, with dependent types.

We describe a prototype implementation in Chapter 8, and then present in Chapter 9 some applications of dependent types which include program error detection, array bound check elimination, redundant match elimination, etc. Lastly, we conclude and point out some directions for future research.

Chapter 2

Preliminaries

In this chapter, we first introduce an untyped language $\lambda^{\mathrm{pat}}_{\mathrm{val}}$ which is basically the call-by-value λ-calculus extended with general pattern matching. The importance of this language lies in its operational semantics, to which we will relate the operational semantics of other typed languages introduced later.

We then introduce an explicitly typed language upon which we will build our type system. We call this language ML_0, which is basically mini-ML extended with pattern matching. We present the typing rules and operational semantics for ML_0 and prove important properties of ML_0 such as the type preservation theorem, which are helpful for understanding what we develop later.

Lastly, we study the operational equivalence relation in $\lambda^{\mathrm{pat}}_{\mathrm{val}}$. This will be used later when we prove the correctness of elaboration algorithms for the languages $\mathrm{ML}_0^{\Pi}(C)$ and $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ in Chapter 4 and 5.

2.1 Untyped λ-calculus with Pattern Matching

A crucial point in many typed programming languages is that types are *indifferent* to program evaluation. Roughly speaking, one can erase all the type information in a program and evaluate it to reach the same result as one would while keeping all the type information during the evaluation. As matter of a fact, it is a common practice in many compilers to discard all the type information in a program after type-checking it. However, recent studies such as (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996; Morrisett, Walker, Crary, and Glew 1998) have demonstrated convincingly that this practice may not be wise because type information can be very helpful for compiler optimization.

Nonetheless it is necessary for us to show that types do not alter the operational semantics of programs in the various typed languages we formulate later in this thesis. For this purpose, we introduce an untyped language $\lambda^{\mathrm{pat}}_{\mathrm{val}}$. We then define an operational semantics for $\lambda^{\mathrm{pat}}_{\mathrm{val}}$ to which the operational semantics of other typed languages will relate.

The syntax of $\lambda^{\mathrm{pat}}_{\mathrm{val}}$ is given in Figure 2.1. We use x, y and f as meta variables for object language variables, c for constructors, e for expressions, u for value forms and v for values. Value forms are a special form of values and values are a special form of expressions. Also we use p for patterns and we emphasize that a variable can occur *at most* once in a given pattern. The signature is a list of constructors available in the language.

patterns	p	::=	$x \mid c \mid c(p) \mid \langle\rangle \mid \langle p_1, p_2\rangle$
matches	ms	::=	$(p \Rightarrow e) \mid (p \Rightarrow e \mid ms)$
expressions	e	::=	$x \mid \langle\rangle \mid \langle e_1, e_2\rangle \mid c(e) \mid (\textbf{case}\ e\ \textbf{of}\ ms) \mid (\textbf{lam}\ x.e) \mid e_1(e_2)$
			$\mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} \mid (\textbf{fix}\ f.u)$
value forms	u	::=	$c(u) \mid \langle\rangle \mid \langle u_1, u_2\rangle \mid (\textbf{lam}\ x.e)$
values	v	::=	$x \mid c(v) \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid (\textbf{lam}\ x.e)$
signatures	$\mathcal{S}$	::=	$\cdot \mid \mathcal{S}, c$
substitutions	θ	::=	$[] \mid \theta[x \mapsto e]$

Figure 2.1: The syntax for $\lambda^{\text{pat}}_{\text{val}}$

The set $\text{FV}(e)$ of free variables in an expression e is defined as follows.

$$\begin{aligned}
\text{FV}(x) &= \{x\} \\
\text{FV}(\langle\rangle) &= \emptyset \\
\text{FV}(c) &= \emptyset \\
\text{FV}(c(e)) &= \text{FV}(e) \\
\text{FV}(p \Rightarrow e) &= \text{FV}(e) \backslash \text{FV}(p) \\
\text{FV}(p \Rightarrow e \mid ms) &= \text{FV}(p \Rightarrow e) \cup \text{FV}(ms) \\
\text{FV}(\textbf{case}\ e\ \textbf{of}\ ms) &= \text{FV}(e) \cup \text{FV}(ms) \\
\text{FV}(\textbf{lam}\ x.e) &= \text{FV}(e) \backslash \{x\} \\
\text{FV}(e_1(e_2)) &= \text{FV}(e_1) \cup \text{FV}(e_2) \\
\text{FV}(\textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}) &= \text{FV}(e_1) \cup (\text{FV}(e_2) \backslash \{x\}) \\
\text{FV}(\textbf{fix}\ f.u) &= \text{FV}(u) \backslash \{f\}
\end{aligned}$$

Substitutions are defined in the standard way. We write $e[\theta]$ as the result of applying substitution θ to e. Since we allow substituting an expression containing free variables for a variable, we emphasize that α-conversion is always performed if necessary to avoid capturing free variables.

We use $\textbf{dom}(\theta)$ for the domain of substitution θ. If $x \notin \textbf{dom}(\theta)$, we use $\theta[x \mapsto e]$ for the substitution θ' such that $\textbf{dom}(\theta') = \textbf{dom}(\theta) \cup \{x\}$ and

$$\theta'(y) = \begin{cases} \theta(y) & \text{if } y \text{ is not } x; \\ e & \text{if } y \text{ is } x. \end{cases}$$

We use $[]$ for the empty substitution θ, and $[x \mapsto e]$ for the substitution θ such that $\textbf{dom}(\theta) = \{x\}$ and $\theta(x) = e$. Let θ_1 and θ_2 be two substitutions such that $\textbf{dom}(\theta_1) \cap \textbf{dom}(\theta_2) = \emptyset$. We define $\theta_1 \cup \theta_2$, the union of θ_1 and θ_2, as the substitution θ such that $\textbf{dom}(\theta) = \textbf{dom}(\theta_1) \cup \textbf{dom}(\theta_2)$ and

$$\theta(x) = \begin{cases} \theta_1(x) & \text{if } x \in \textbf{dom}(\theta_1); \\ \theta_2(x) & \text{if } x \in \textbf{dom}(\theta_2). \end{cases}$$

Similarly, $\theta_1 \circ \theta_2$, the composition of θ_1 and θ_2, is defined as the substitution θ such that $\textbf{dom}(\theta) = \textbf{dom}(\theta_1) \cup \textbf{dom}(\theta_2)$, and

$$\theta(x) = \begin{cases} (\theta_1(x))[\theta_2] & \text{if } x \in \textbf{dom}(\theta_1); \\ \theta_2(x) & \text{if } x \in \textbf{dom}(\theta_2). \end{cases}$$

A substitution θ is called a value substitution if $\theta(x)$ is a value for all $x \in \mathbf{dom}(\theta)$. We use $e[\theta]$ for the result of applying θ to e and $e[x_1, \ldots, x_n \mapsto e_1, \ldots, e_n]$ for $e[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$.

Proposition 2.1.1 *Given a value form u and an expression e, $u[x \mapsto e]$ is also a value form. Hence, value forms are closed under substitution.*

Proof This immediately follows from a structural induction on u. ∎

Proposition 2.1.2 *Given values v_1 and v_2, $v_2[x \mapsto v_1]$ is also a value. Hence, values are closed under value substitution.*

Proof This immediately follows from a structural induction on v_2.

- v_2 is a variable y. If y is x, then $v_2[x \mapsto v_1] = v_1$ is a value. Otherwise, $v_2[x \mapsto v_1] = y$ is also a value.
- v_2 is of form $\lambda y.e$. Then $v_2[x \mapsto v_1] = \lambda y.e[x \mapsto v_1]$ is obviously a value. Note we can assume that there are no free occurrences of y in v_1.

All other cases can be readily verified. ∎

Therefore, a significant difference between value forms and values is that the former are closed under all substitutions while the latter are only closed under value substitutions. This is the primary reason why we require that u be a value form in (**fix** $x.u$). This requirement also rules out troublesome expressions such as (**fix** $x.x$), which are of little use in practice.

2.1.1 Dynamic Semantics

We will present the operational semantics of $\lambda^{\mathrm{pat}}_{\mathrm{val}}$ in terms of *natural semantics* (Kahn 1987). This approach supports a short and clean formulation, but it prevents us from distinguishing a "stuck" program from a non-terminating one. An alternative would be using the "small-step" reduction semantics, which does enable us to distinguish a "stuck" program from a non-terminating one but its use in our setting is more involved. We feel that natural semantics suffices for our purpose, and therefore choose it over reduction semantics. Nonetheless, we will formulate the reduction semantics of $\lambda^{\mathrm{pat}}_{\mathrm{val}}$ when studying the operational equivalence relation in $\lambda^{\mathrm{pat}}_{\mathrm{val}}$.

Given a pattern p and a value v, a judgement of form $\mathbf{match}(p, v) \Longrightarrow \theta$, which means that matching a value v against a pattern p yields a substitution for the variables in p, can be derived with the application of the rules in Figure 2.2. Notice that the rule **(match-prod)** makes sense because p_1 and p_2 share no common variables.

The natural semantics for $\lambda^{\mathrm{pat}}_{\mathrm{val}}$ is given in Figure 2.3. Notice the presence of the rule **(ev-var)**, which means that we allow the evaluation of open code, that is code containing the occurrences of free variables. The main reason is that we hope that the theorems we prove are also applicable to program transformation, where the manipulation of open code is a necessity.

We will use constants $\overline{0}, \overline{1}, \overline{-1}, \ldots$ for integers and *nil*, *cons* for list constructors in our examples.

$$\frac{}{\mathbf{match}(x, v) \Longrightarrow [x \mapsto v]}\ \textbf{(match-var)}$$

$$\frac{}{\mathbf{match}(\langle\rangle, \langle\rangle) \Longrightarrow []}\ \textbf{(match-unit)}$$

$$\frac{\mathbf{match}(p_1, v_1) \Longrightarrow \theta_1 \quad \mathbf{match}(p_2, v_2) \Longrightarrow \theta_2}{\mathbf{match}(\langle p_1, p_2\rangle, \langle v_1, v_2\rangle) \Longrightarrow \theta_1 \cup \theta_2}\ \textbf{(match-prod)}$$

$$\frac{}{\mathbf{match}(c, c) \Longrightarrow []}\ \textbf{(match-cons-wo)}$$

$$\frac{\mathbf{match}(p, v) \Longrightarrow \theta}{\mathbf{match}(c(p), c(v)) \Longrightarrow \theta}\ \textbf{(match-cons-w)}$$

Figure 2.2: The pattern matching rules for $\lambda^{\text{pat}}_{\text{val}}$

$$\frac{}{x \hookrightarrow_0 x}\ \textbf{(ev-var)}$$

$$\frac{}{\langle\rangle \hookrightarrow_0 \langle\rangle}\ \textbf{(ev-unit)}$$

$$\frac{}{c \hookrightarrow_0 c}\ \textbf{(ev-cons-wo)}$$

$$\frac{e \hookrightarrow_0 v}{c(e) \hookrightarrow_0 c(v)}\ \textbf{(ev-cons-w)}$$

$$\frac{e_1 \hookrightarrow_0 v_1 \quad e_2 \hookrightarrow_0 v_2}{\langle e_1, e_2\rangle \hookrightarrow_0 \langle v_1, v_2\rangle}\ \textbf{(ev-prod)}$$

$$\frac{e_0 \hookrightarrow_0 v_0 \quad \mathbf{match}(v_0, p_k) \Longrightarrow \theta \text{ for some } 1 \le k \le n \quad e_k[\theta] \hookrightarrow_0 v}{(\mathbf{case}\ e_0\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)) \hookrightarrow_0 v}\ \textbf{(ev-case)}$$

$$\frac{}{(\mathbf{lam}\ x.e) \hookrightarrow_0 (\mathbf{lam}\ x.e)}\ \textbf{(ev-lam)}$$

$$\frac{e_1 \hookrightarrow_0 (\mathbf{lam}\ x.e) \quad e_2 \hookrightarrow_0 v_2 \quad e[x \mapsto v_2] \hookrightarrow_0 v}{e_1(e_2) \hookrightarrow_0 v}\ \textbf{(ev-app)}$$

$$\frac{e_1 \hookrightarrow_0 v_1 \quad e_2[x \mapsto v_1] \hookrightarrow_0 v_2}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \hookrightarrow_0 v_2}\ \textbf{(ev-let)}$$

$$\frac{}{(\mathbf{fix}\ f.u) \hookrightarrow_0 u[f \mapsto (\mathbf{fix}\ f.u)]}\ \textbf{(ev-fix)}$$

Figure 2.3: The evaluation rules for the natural semantics of $\lambda^{\text{pat}}_{\text{val}}$

Example 2.1.3 *Let $\mathcal{D}_1$ be the following derivation.*

$$\dfrac{\dfrac{\dfrac{}{\overline{0} \hookrightarrow_0 \overline{0}}\ \textbf{(ev-cons-wo)} \quad \dfrac{}{nil \hookrightarrow_0 nil}\ \textbf{(ev-cons-wo)}}{\langle \overline{0}, nil\rangle \hookrightarrow_0 \langle \overline{0}, nil\rangle}\ \textbf{(ev-prod)}}{cons(\langle \overline{0}, nil\rangle) \hookrightarrow_0 cons(\langle \overline{0}, nil\rangle)}$$

Let $\mathcal{D}_2$ be the following derivation.

$$\dfrac{\dfrac{\dfrac{}{\textbf{match}(x, \overline{0}) \Longrightarrow [x \mapsto \overline{0}]}\ \textbf{(match-var)} \quad \dfrac{}{\textbf{match}(xs, nil) \Longrightarrow [xs \mapsto nil]}\ \textbf{(match-var)}}{\textbf{match}(\langle x, xs\rangle, \langle \overline{0}, nil\rangle) \Longrightarrow [x \mapsto \overline{0}, xs \mapsto nil]}\ \textbf{(match-prod)}}{\textbf{match}(cons(\langle \overline{0}, nil\rangle), cons(\langle x, xs\rangle)) \Longrightarrow [x \mapsto \overline{0}, xs \mapsto nil]}\ \textbf{(match-cons-w)}$$

Let $tail = \lambda l.\textbf{case}\ l\ \textbf{of}\ cons(\langle x, xs\rangle) \Rightarrow xs$, and $tail(cons(\langle \overline{0}, nil\rangle)) \hookrightarrow_0 nil$ is derivable as follows.

$$\dfrac{\dfrac{}{tail \hookrightarrow_0 tail}\ \textbf{(ev-lam)} \quad \mathcal{D}_1 \quad \dfrac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \dfrac{}{nil \hookrightarrow_0 nil}\ \textbf{(ev-cons-wo)}}{\textbf{case}\ cons(\langle \overline{0}, nil\rangle)\ \textbf{of}\ cons(\langle x, xs\rangle) \Rightarrow xs \hookrightarrow_0 nil}\ \textbf{(ev-case)}}{tail(cons(\langle \overline{0}, nil\rangle)) \hookrightarrow_0 nil}\ \textbf{(ev-app)}$$

Notice that the rule **(ev-case)** introduces a certain amount of nondeterminism into the dynamic semantics of $\lambda^{\mathrm{pat}}_{\mathrm{val}}$ since it does not specify which matching clause is chosen if several are applicable. On the other hand, it is specified in ML that pattern matching is done sequentially, that is, the chosen matching clause is always the first one which is applicable. However, this difference is considerably a minor issue since in theory we can always require that all matching clauses be not overlapping

Theorem 2.1.4 *$v \hookrightarrow_0 v$ for every value v in $\lambda^{\mathrm{pat}}_{\mathrm{val}}$.*

Proof This immediately follows from a structural induction on v. We present a few cases.

- $v = \langle v_1, v_2\rangle$. By induction hypothesis $v_i \hookrightarrow_0 v_i$ are derivable for $i = 1, 2$. Hence we have the following derivation.
$$\dfrac{v_1 \hookrightarrow_0 v_1 \quad v_2 \hookrightarrow_0 v_2}{v \hookrightarrow_0 v}\ \textbf{(ev-prod)}$$
- $v = \textbf{lam}\ x.e$. Then we have the following.
$$\dfrac{}{v \hookrightarrow_0 v}\ \textbf{(ev-lam)}$$

All other cases are trivial. ∎

2.2 Mini-ML with Pattern Matching

We now introduce an explicitly typed programming language (ML_0) which basically extends mini-ML (Clément, Despeyroux, Despeyroux, and Kahn 1986) with general pattern matching. This is a simply typed version of $\lambda^{\mathrm{pat}}_{\mathrm{val}}$. The syntax of ML_0 is given in Figure 2.4. Given a context $\Gamma = x_1 : \tau_1, \ldots, x : \tau_n$ (we omit the leading $\cdot$ if the context is not empty), we always assume that all x_i are distinct for $i = 1, \ldots, n$. We write $\mathbf{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = \tau_i$ for $i = 1, \ldots, n$. A signature declares a list of constructors associated with their types. Notice that the type of a constructor is required to be of form either β or $\tau \to \beta$, where β is a (user-defined) base type, that is, a constructor is either without an argument or with exactly *one* argument.

base types	β	::=	$\texttt{bool} \mid \texttt{int} \mid$ (other user defined datatypes)
types	τ	::=	$\beta \mid \mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2$
patterns	p	::=	$x \mid c(p) \mid \langle\rangle \mid \langle p_1, p_2 \rangle$
matches	ms	::=	$(p \Rightarrow e) \mid (p \Rightarrow e \mid ms)$
expressions	e	::=	$x \mid \langle\rangle \mid \langle e_1, e_2 \rangle \mid c(e) \mid (\textbf{case}\ e\ \textbf{of}\ ms) \mid (\textbf{lam}\ x : \tau . e) \mid e_1(e_2)$ $\mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} \mid (\textbf{fix}\ f : \tau . u)$
value forms	u	::=	$c(u) \mid \langle\rangle \mid \langle u_1, u_2 \rangle \mid (\textbf{lam}\ x : \tau . e)$
values	v	::=	$x \mid c(v) \mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid (\textbf{lam}\ x : \tau . e)$
contexts	Γ	::=	$\cdot \mid \Gamma, x : \tau$
signatures	$\mathcal{S}$	::=	$\cdot \mid \mathcal{S}, c : \beta \mid \mathcal{S}, c : \tau \to \beta$
substitutions	θ	::=	$[] \mid \theta[x \mapsto e]$

Figure 2.4: The syntax for ML_0

$$\frac{}{x \downarrow \tau \rhd x : \tau}\ \textbf{(pat-var)}$$

$$\frac{}{\langle\rangle \downarrow \mathbf{1} \rhd \cdot}\ \textbf{(pat-unit)}$$

$$\frac{p_1 \downarrow \tau_1 \rhd \Gamma_1 \quad p_2 \downarrow \tau_2 \rhd \Gamma_2}{\langle p_1, p_2 \rangle \downarrow \tau_1 * \tau_2 \rhd \Gamma_1, \Gamma_2}\ \textbf{(pat-prod)}$$

$$\frac{\mathcal{S}(c) = \beta}{c \downarrow \beta \rhd \cdot}\ \textbf{(pat-cons-wo)}$$

$$\frac{\mathcal{S}(c) = \tau \to \beta \quad p \downarrow \tau \rhd \Gamma'}{c(p) \downarrow \beta \rhd \Gamma'}\ \textbf{(pat-cons-w)}$$

Figure 2.5: The pattern matching rules for ML_0

2.2.1 Static Semantics

Given a pattern p and a type τ, we can derive a judgement of form $p \downarrow \tau \rhd \Gamma$ with the rules in Figure 2.5, which reads that checking pattern p against type τ yields a context Γ.

In the following examples, we assume that `intlist` is a base type and *nil*, *cons* are constructors of type `intlist` and $\texttt{int} * \texttt{intlist} \to \texttt{intlist}$, respectively.

Example 2.2.1 *The following is a derivation of* $cons(\langle x, nil \rangle) \downarrow \texttt{intlist} \rhd x : \texttt{int}$.

$$\frac{\mathcal{S}(cons) = \texttt{int} * \texttt{intlist} \to \texttt{intlist} \quad \dfrac{\dfrac{}{x \downarrow \texttt{int} \rhd x : \texttt{int}}\ \textbf{(pat-var)} \quad \dfrac{\mathcal{S}(nil) = \texttt{intlist}}{nil \downarrow \texttt{intlist} \rhd \cdot}\ \textbf{(pat-cons-wo)}}{\langle x, nil \rangle \downarrow \texttt{int} * \texttt{intlist} \rhd x : \texttt{int}}\ \textbf{(pat-prod)}}{cons(\langle x, nil \rangle) \downarrow \texttt{intlist} \rhd x : \texttt{int}}\ \textbf{(pat-cons-w)}$$

The typing rules for ML_0 are given in Figure 2.6. We present an example of type inference in

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textbf{(ty-var)}$$

$$\frac{\mathcal{S}(c) = \beta}{\Gamma \vdash c : \beta}\ \textbf{(ty-cons-wo)}$$

$$\frac{\mathcal{S}(c) = \tau \to \beta \quad \Gamma \vdash e : \tau}{\Gamma \vdash c(e) : \delta}\ \textbf{(ty-cons-w)}$$

$$\frac{}{\Gamma \vdash \langle\rangle : \mathbf{1}}\ \textbf{(ty-unit)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 * \tau_2}\ \textbf{(ty-prod)}$$

$$\frac{p \downarrow \tau_1 \rhd \Gamma' \quad \Gamma, \Gamma' \vdash e : \tau_2}{\Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2}\ \textbf{(ty-match)}$$

$$\frac{\Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2 \quad \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\Gamma \vdash (p \Rightarrow e \mid ms) : \tau_1 \Rightarrow \tau_2}\ \textbf{(ty-matches)}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\Gamma \vdash (\textbf{case}\ e\ \textbf{of}\ ms) : \tau_2}\ \textbf{(ty-cases)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\textbf{lam}\ x : \tau_1.e) : \tau_1 \to \tau_2}\ \textbf{(ty-lam)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}\ \textbf{(ty-app)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}) : \tau_2}\ \textbf{(ty-let)}$$

$$\frac{\Gamma, f : \tau \vdash u : \tau}{\Gamma \vdash (\textbf{fix}\ f : \tau.u) : \tau}\ \textbf{(ty-fix)}$$

Figure 2.6: The typing rules for ML_0

ML_0.

Example 2.2.2 *The following is a derivation of* $\cdot \vdash (\textbf{lam}\ x : \mathtt{int}.cons(\langle x, nil\rangle)) : \mathtt{int} \to \mathtt{intlist}$.

$$\dfrac{\dfrac{\mathcal{S}(cons) = \mathtt{int} * \mathtt{intlist} \to \mathtt{intlist} \quad \dfrac{\dfrac{}{x : \mathtt{int} \vdash x : \mathtt{int}}\ \textbf{(ty-cons-wo)} \quad \dfrac{\mathcal{S}(nil) = \mathtt{intlist}}{x : \mathtt{int} \vdash nil : \mathtt{intlist}}\ \textbf{(ty-cons-wo)}}{x : \mathtt{int} \vdash \langle x, nil\rangle \vdash \mathtt{int} * \mathtt{intlist}}\ \textbf{(ty-prod)}}{x : \mathtt{int} \vdash cons(\langle x, nil\rangle) : \mathtt{intlist}}\ \textbf{(ty-cons-w)}}{\cdot \vdash (\textbf{lam}\ x : \mathtt{int}.cons(\langle x, nil\rangle)) : \mathtt{int} \to \mathtt{intlist}}\ \textbf{(ty-lam)}$$

Given Γ, Γ' and θ, a judgement of form $\Gamma \vdash \theta : \Gamma'$ can be derived with the application of the following rules. Such a judgement means that $\mathbf{dom}(\theta) = \mathbf{dom}(\Gamma')$ and $\Gamma \vdash \theta(x) : \Gamma'(x)$ is derivable for all $x \in \mathbf{dom}(\theta)$.

$$\frac{}{\Gamma \vdash [] : \cdot}\ \textbf{(subst-empty)} \qquad \frac{\Gamma \vdash \theta : \Gamma' \quad \Gamma \vdash e : \tau}{\Gamma \vdash \theta[x \mapsto e] : \Gamma', x : \tau}\ \textbf{(subst-var)}$$

The next proposition shows that judgement $\Gamma \vdash \theta : \Gamma'$ has the intended meaning.

Proposition 2.2.3 *We have the following.*

1. *If* $\Gamma \vdash \theta : \Gamma'$ *is derivable, then* $\mathbf{dom}(\theta) = \mathbf{dom}(\Gamma')$ *and* $\Gamma \vdash \theta(x) : \Gamma'(x)$ *is derivable for every* $x \in \mathbf{dom}(\theta)$.
2. *Given* θ_1 *and* θ_2 *such that* $\mathbf{dom}(\theta_1) \cap \mathbf{dom}(\theta_2) = \emptyset$, *then the following rule is admissible.*

$$\frac{\Gamma \vdash \theta_1 : \Gamma_1 \quad \Gamma \vdash \theta_2 : \Gamma_2}{\Gamma \vdash \theta_1 \cup \theta_2 : \Gamma_1, \Gamma_2} \textbf{ (subst-subst)}$$

Proof (1) follows from a structural induction on the derivation of $\Gamma \vdash \theta : \Gamma'$ and (2) follows from a structural induction on the derivation of $\Gamma \vdash \theta_2 : \Gamma_2$. We present the proof for (2).

- $\theta_2 = []$. This is trivial.
- $\theta_2 = \theta_2'[x \mapsto e]$. Suppose $\Gamma_2 = \Gamma_2', x : \tau$. Then we have the following derivation.

$$\frac{\Gamma \vdash \theta_2' : \Gamma_2' \quad \Gamma \vdash x : \tau}{\Gamma \vdash \theta_2'[x \mapsto e] : \Gamma_2', x : \tau} \textbf{ (subst-var)}$$

  By induction hypothesis, $\Gamma \vdash \theta_1 \cup \theta_2' : \Gamma_1, \Gamma_2'$ is derivable. This leads to the following derivation.

$$\frac{\Gamma \vdash \theta_1 \cup \theta_2' : \Gamma_1, \Gamma_2' \quad \Gamma \vdash x : \tau}{\Gamma \vdash (\theta_1 \cup \theta_2')[x \mapsto e] : \Gamma_1, \Gamma_2', x : \tau} \textbf{ (subst-var)}$$

  Since $\theta_1 \cup \theta_2$ is $(\theta_1 \cup \theta_2')[x \mapsto e]$ and Γ_2 is $\Gamma_2', x : \tau$, we are done.

■

Lemma 2.2.4 *If both* $\Gamma, \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash \theta : \Gamma'$ *are derivable, then* $\Gamma \vdash e[\theta] : \tau$ *is derivable.*

Proof The proof follows from a structural induction on the derivation $\mathcal{D}$ of $\Gamma, \Gamma' \vdash e : \tau$. We present a few cases.

$\mathcal{D} = \dfrac{\Gamma(x) = \tau}{\Gamma, \Gamma' \vdash x : \tau}$ Then $x \notin \mathbf{dom}(\Gamma')$. Since $\mathbf{dom}(\theta) = \mathbf{dom}(\Gamma')$ by Proposition 2.2.3, $x \notin \mathbf{dom}(\theta)$. This implies $x[\theta] = x$. Clearly, $\Gamma \vdash x : \tau$ is derivable.

$\mathcal{D} = \dfrac{\Gamma'(x) = \tau}{\Gamma, \Gamma' \vdash x : \tau}$ Since $\mathbf{dom}(\theta) = \mathbf{dom}(\Gamma')$ by Proposition 2.2.3, $x \in \mathbf{dom}(\theta)$. This implies $x[\theta] = \theta(x)$. Note $\Gamma \vdash \theta(x) : \tau$ is derivable by Proposition 2.2.3 since $\Gamma \vdash \theta : \Gamma'$ is.

$\mathcal{D} = \dfrac{\Gamma, \Gamma', x : \tau_1 \vdash e_1 : \tau_2}{\Gamma, \Gamma' \vdash (\mathbf{lam}\ x : \tau_1.e_1) : \tau_1 \to \tau_2}$ Then we can derive $\Gamma, x : \tau_1, \Gamma' \vdash e_1 : \tau_2$ and $\Gamma, x : \tau_1 \vdash \theta : \Gamma'$. By induction hypothesis, $\Gamma, x : \tau_1 \vdash e_1[\theta] : \tau_2$ is derivable, and this leads to the following derivation.

$$\frac{\Gamma, x : \tau_1 \vdash e_1[\theta] : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.e_1[\theta]) : \tau_2} \textbf{ (ty-lam)}$$

$$\frac{}{(\mathbf{lam}\ x : \tau.e) \hookrightarrow_0 (\mathbf{lam}\ x : \tau.e)}\ \textbf{(ev-lam)}$$

$$\frac{e_1 \hookrightarrow_0 (\mathbf{lam}\ x : \tau.e) \quad e_2 \hookrightarrow_0 v_2 \quad e[x \mapsto v_2] \hookrightarrow_0 v}{e_1(e_2) \hookrightarrow_0 v}\ \textbf{(ev-app)}$$

$$\frac{}{(\mathbf{fix}\ f : \tau.u) \hookrightarrow_0 u[f \mapsto (\mathbf{fix}\ f : \tau.u)]}\ \textbf{(ev-fix)}$$

Figure 2.7: Some evaluation rules for the natural semantics of ML_0

Note $x \notin \mathbf{dom}(\Gamma') = \mathbf{dom}(\theta)$. Since $\Gamma \vdash \theta : \Gamma'$, $x \notin \mathrm{FV}(\theta(y))$ for all $y \in \mathbf{dom}(\theta)$. Therefore, $(\lambda x : \tau_1.e_1)[\theta] = \lambda x : \tau_1.e_1[\theta]$.

All other cases can be handled similarly. ∎

If a value v matches a pattern p, then $\mathbf{match}(p, v) \Longrightarrow \theta$ is derivable for some substitution θ. The next lemma shows that if the type of v is given, then the type of $\theta(x)$ for every $x \in \mathbf{dom}(\theta)$ is fixed. This is crucial to proving the type preservation theorem for ML_0.

Lemma 2.2.5 *If* $\Gamma \vdash v : \tau$, $p \downarrow \tau \rhd \Gamma'$ *and* $\mathbf{match}(p, v) \Longrightarrow \theta$ *are derivable, then* $\Gamma \vdash \theta : \Gamma'$ *is derivable.*

Proof By a structural induction on the derivation $\mathcal{D}$ of $p \downarrow \tau \rhd \Gamma'$. We present one case as follows.

$$\mathcal{D} = \frac{\mathbf{match}(p_1, v_1) \Longrightarrow \theta_1 \qquad \mathbf{match}(p_2, v_2) \Longrightarrow \theta_2}{\mathbf{match}(\langle p_1, p_2\rangle, \langle v_1, v_2\rangle) \Longrightarrow \theta_1 \cup \theta_2}$$

By induction hypothesis, $\Gamma \vdash \theta_i : \Gamma_i$ are derivable for $i = 1, 2$. Hence we have the following derivation since **(subst-subst)** is an admissible rule by Proposition 2.2.3.

$$\frac{\Gamma \vdash \theta_1 : \Gamma_1 \quad \Gamma \vdash \theta_2 : \Gamma_2}{\Gamma \vdash \theta_1 \cup \theta_2 : \Gamma_1, \Gamma_2}\ \textbf{(subst-subst)}$$

All other cases are trivial. ∎

2.2.2 Dynamic Semantics

The natural semantics of ML_0 is almost the same as that of $\lambda^{\mathrm{pat}}_{\mathrm{val}}$. The only changes are made in the formulation of the rules in Figure 2.7, where types are carried around during evaluation. All other rules are unchanged.

Notice that types play no rôle in the formulation of the evaluation rules in Figure 2.7. To make this precise, we define a type erasure function $|\cdot|$ as follows, which maps an expression in ML_0 into

one in $\lambda_{\text{val}}^{\text{pat}}$.

$$\begin{array}{rcl} |x| & = & x \\ |c| & = & c \\ |p \Rightarrow e| & = & p \Rightarrow |e| \\ |(p \Rightarrow e \mid ms)| & = & p \Rightarrow |e| \mid |ms| \\ |\textbf{case } e \textbf{ of } ms| & = & \textbf{case } |e| \textbf{ of } |ms| \\ |\textbf{lam } x : \tau.e| & = & \textbf{lam } x.|e| \\ |e_1(e_2)| & = & |e_1|(|e_2|) \\ |\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end}| & = & \textbf{let } x = |e_1| \textbf{ in } |e_2| \textbf{ end} \\ |\textbf{fix } f : \tau.u| & = & \textbf{fix } f.|u| \end{array}$$

Theorem 2.2.6 *Given an expression e in* ML_0*, we have the following.*

1. *If $e \hookrightarrow_0 v$ is derivable in* ML_0*, then $|e| \hookrightarrow_0 |v|$ is derivable in $\lambda_{\text{val}}^{\text{pat}}$.*
2. *if $|e| \hookrightarrow_0 v_0$ is derivable in $\lambda_{\text{val}}^{\text{pat}}$, then $e \hookrightarrow_0 v$ is derivable in* ML_0 *for some v such that $|v| = v_0$.*

Proof (1) and (2) follow from a structural induction on the derivations of $e \hookrightarrow_0 v$ and $|e| \hookrightarrow_0 v_0$, respectively. ∎

Theorem 2.2.6 clearly exhibits the indifference of types to evaluation. However, one great advantage of imposing a type system on a language is that we are then able to prove certainly invariant properties about the evaluation of *well-typed* expressions.

2.2.3 Soundness

We are now ready to present the type preservation theorem for ML_0, which asserts that the evaluation rules for the natural semantics of ML_0 does not alter the types of the evaluated expressions. Notice that this theorem is closely related to but different from the subject reduction theorem (not presented in the thesis), which asserts that the (small-step) reduction semantics of ML_0 is type preserving.

The type preservation theorem is a fundamental theorem which relates the static semantics of ML_0, expressed in the form of type inference rules, to the dynamic semantics of ML_0, expressed in the form of natural semantics.

Since we allow the evaluation of open code, the formulation of the following type preservation theorem is slightly different from the standard one, which deals with only closed code and therefore needs no variable context to keep track of free variables in the code.

Theorem 2.2.7 *(Type preservation for* ML_0*) Given e, v where $e \hookrightarrow_0 v$ is derivable. If $\Gamma \vdash e : \tau$ is derivable then $\Gamma \vdash v : \tau$ is also derivable.*

Proof This follows from a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_0 v$. We present a few cases.

$\mathcal{D} = \dfrac{}{x \hookrightarrow_0 x}$ Trivially, $\Gamma \vdash x : \tau$ is derivable since $\Gamma \vdash x : \tau$ is derivable.

$\mathcal{D} = \dfrac{e_0 \hookrightarrow_0 v_0 \quad \mathbf{match}(v_0, p_k) \Longrightarrow \theta \text{ for some } 1 \leq k \leq n \quad e_k[\theta] \hookrightarrow_0 v}{(\mathbf{case}\ e_0\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)) \hookrightarrow_0 v}$ Then we have a derivation of the following form since $\Gamma \vdash (\mathbf{case}\ e_0\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)) : \tau$ is derivable.

$$\frac{\Gamma \vdash e_0 : \tau_1 \quad \Gamma \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \tau_1 \Rightarrow \tau}{\Gamma \vdash (\mathbf{case}\ e_0\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)) : \tau}\ \textbf{(ty-case)}$$

By induction hypothesis, $\Gamma \vdash v_0 : \tau_1$ is derivable. Notice $\Gamma \vdash p_i \Rightarrow e_i : \tau_1 \Rightarrow \tau$ are derivable for $1 \leq i \leq n$. Hence $p_k \downarrow \tau_1 \rhd \Gamma'$ is derivable for some Γ' and $\Gamma, \Gamma' \vdash e_k : \tau$ is derivable. By Lemma 2.2.5, $\Gamma \vdash \theta : \Gamma'$ is derivable. This leads to a derivation of $\Gamma \vdash e_k[\theta] : \tau$ by Lemma 2.2.4. By induction hypothesis, $\Gamma \vdash v : \tau$ is derivable.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_0 (\mathbf{lam}\ x : \tau_1.e_1') \quad e_2 \hookrightarrow_0 v_2 \quad e_1'[x \mapsto v_2] \hookrightarrow_0 v}{e_1(e_2) \hookrightarrow_0 v}$ Since $\Gamma \vdash e_1(e_2) : \tau$ is derivable, we have a derivation of the following form.

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau}\ \textbf{(ty-app)}$$

By induction hypothesis, both $\Gamma \vdash (\lambda x : \tau_1.e_1') : \tau_1 \to \tau$ and $\Gamma \vdash v_2 : \tau_1$ are derivable. Hence, $\Gamma \vdash e_1'[x \mapsto v_2] : \tau$ is derivable following Lemma 2.2.4. Again by induction hypothesis, $\Gamma \vdash v : \tau$ is derivable.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_0 v_1 \quad e_2[x \mapsto v_1] \hookrightarrow_0 v}{(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}) \hookrightarrow_0 v}$ Since $\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau$ is derivable, we have a derivation of the following form.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau}\ \textbf{(ty-let)}$$

By induction hypothesis, $\Gamma \vdash v_1 : \tau_1$ is derivable. Therefore, $\Gamma \vdash e_2[x \mapsto v_1] : \tau$ is derivable following Lemma 2.2.4. This yields that $\Gamma \vdash v : \tau$ is derivable by induction hypothesis.

$\mathcal{D} = \dfrac{}{(\mathbf{fix}\ f : \tau.u) \hookrightarrow_0 u[f \mapsto (\mathbf{fix}\ f : \tau.u)]}$ Since $\Gamma \vdash (\mathbf{fix}\ f : \tau.u) : \tau$ is derivable, we have a derivation of the following form.

$$\frac{\Gamma, f : \tau \vdash u : \tau}{\Gamma \vdash (\mathbf{fix}\ f : \tau.u) : \tau}\ \textbf{(ty-fix)}$$

Hence, $\Gamma \vdash u[f \mapsto (\mathbf{fix}\ f : \tau.u)] : \tau$ is derivable following Lemma 2.2.4.

All other cases can be handled similarly.

Notice that in the case where e is $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}$, the derivation of $\Gamma \vdash e_2[x \mapsto v_1] : \tau$ can be more complex that that of $\Gamma, x : \tau_1 \vdash e_2 : \tau$. Therefore, the proof could not have succeeded if we had proceeded by a structural induction on the derivation of $\Gamma \vdash e : \tau$. ∎

2.3 Operational Equivalence

We present some basics on operational equivalence in this section, which will be used later in Chapter 4 and Chapter 5 to prove the correctness of elaboration algorithms. This is also an appropriate place for us to mention something about the *reduction semantics* since it is based on the notion of *evaluation context* that we introduce as follows.

Definition 2.3.1 *We present the definition of evaluation contexts and (general) contexts as follows.*

$$
\begin{array}{llll}
\textit{(evaluation contexts)} & E & ::= & [] \mid \langle E, e\rangle \mid \langle v, E\rangle \mid c(E) \mid \textbf{case } E \textbf{ of } ms \\
 & & & \mid E(e) \mid v(E) \mid \textbf{let } x = E \textbf{ in } e \textbf{ end} \\
\textit{(match contexts)} & C_m & ::= & p \Rightarrow C \mid (p \Rightarrow e \mid C_m) \mid (p \Rightarrow C \mid ms) \\
\textit{(contexts)} & C & ::= & [] \mid \langle C, e\rangle \mid \langle e, C\rangle \mid c(C) \mid \textbf{case } C \textbf{ of } ms \mid \textbf{case } e \textbf{ of } C_m \\
 & & & \mid \textbf{lam } x.(C) \mid C(e) \mid e(C) \\
 & & & \mid \textbf{let } x = C \textbf{ in } e \textbf{ end} \mid \textbf{let } x = e \textbf{ in } C \textbf{ end} \mid \textbf{fix } f.C
\end{array}
$$

Given a context C and an expression e, $C[e]$ stands for the expression formulated by replacing with e the hole $[]$ in C. We emphasize that *this replacement is variable capturing*. For instance, given $C = \textbf{lam } x.[]$, then $C[x] = \textbf{lam } x.x$. Given two contexts C_1 and C_2, $C_1[C_2]$ is the context formulated by replacing with C_2 the hole $[]$ in C_1.

Proposition 2.3.2 *We have the following.*

1. *Given two evaluation contexts E_1 and E_2, $E_1[E_2]$ is also an evaluation context.*
2. *Given an evaluation context E and a value v, $E[x \mapsto v]$ is also an evaluation context.*
3. *Given an evaluation context E and an expression e, no free variables in e are captured when the hole $[]$ in E is replaced with e.*

Proof (1) simply follows from a structural induction on E_1. We present a few cases.

- $E_1 = []$. Then $E_1[E_2] = E_2$ is an evaluation context.
- $E_1 = \textbf{let } x = E_1' \textbf{ in } e \textbf{ end}$. Then $E_1'[E_2]$ is an evaluation context by induction hypothesis. Hence, $E_1[E_2] = \textbf{let } x = E_1'[E_2] \textbf{ in } e \textbf{ end}$ is also an evaluation context
- $E_1 = \textbf{case } E_1' \textbf{ of } ms$. Then $E_1'[E_2]$ is an evaluation context by induction hypothesis. Hence, $E_1[E_2] = \textbf{case } E_1'[E_2] \textbf{ of } ms$ is also an evaluation context

The rest of the cases can be handled similarly.

We omit the proofs of (2) and (3), which are based on a structural induction on E. ∎

Definition 2.3.3 *We define as follows redexes and their reductions on the left-hand and right-hand sides of $\mapsto$, respectively.*

$$
\begin{array}{rcl}
(\textbf{lam } x.e)(v) & \mapsto & e[x \mapsto v] \\
\textbf{let } x = v \textbf{ in } e \textbf{ end} & \mapsto & e[x \mapsto v] \\
\textbf{fix } f.u & \mapsto & u[f \mapsto (\textbf{fix } f.u)] \\
\textbf{case } v \textbf{ of } (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) & \mapsto & e_k[\theta], \\
\multicolumn{3}{l}{\textit{where } \textbf{match}(v, p_k) \Longrightarrow \theta \textit{ is derivable for some } 1 \leq k \leq n}
\end{array}
$$

The one-step reduction relation $\mapsto$ *is defined as follows.* $e_1 \mapsto e_2$ *if and only if* $e_1 = E[e]$ *for some evaluation context* E *and redex* e *and* $e_2 = E[e']$*, where* e' *is the reduction of* e*. We also say* e_1 *evaluates to* e_2 *in one step if* $e_1 \mapsto e_2$*.*

Notice that the relation $\mapsto$ is *context-sensitive*, that is, we cannot in general infer $C[e] \mapsto C[e']$ even if we have $e \mapsto e'$. However, this is true by Proposition 2.3.2 if C is an evaluation context. Let $\mapsto^*$ be the reflexive and transitive closure of $\mapsto$. The reduction semantics of $\lambda^{\mathrm{pat}}_{\mathrm{val}}$ states that e evaluates to v if $e \mapsto^* v$ holds. We point out that a redex of form **case** v **of** ms may have different reductions. Therefore, this reduction semantics contains a certain amount of nondeterminism.

Clearly, Proposition 2.3.2 implies $E[e] \mapsto^* E[e']$ if $e \mapsto^* e'$. We will use this property implicitly in the following presentation. The next theorem relates $\hookrightarrow_0$ and $\mapsto^*$ to each other.

Proposition 2.3.4 *We have the following.*

1. *If* e *is not a value, neither is* $E[e]$*.*
2. *If* $e = E[r]$ *for some redex* r *and* $e = E_1[e_1]$ *for some* e_1 *which is not a value, then* $e_1 = E_2[r]$ *for some* E_2 *and* $E = E_1[E_2]$*.*
3. *If* $E_1[r_1] = E_2[r_2]$ *for redexes* r_1 *and* r_2*, then* $E_1 = E_2$ *and* $r_1 = r_2$*.*
4. *If* $e = E[e_1] \mapsto^* v$*, then there is some value* v_1 *such that* $e = E[e_1] \mapsto^* E[v_1] \mapsto^* v$*.*

Proof (1) simply follows from the definition of values. We now proceed to prove (2) by a structural induction on E_1.

- $E_1 = []$. Then this is trivial.
- $E_1 = \langle E_1', e_2 \rangle$. Then $e = \langle E_1'[e_1], e_2 \rangle$. Since e_1 is not a value, (1) implies that $E_1'[e_1]$ is not a value. So E must be of form $\langle E', e_2 \rangle$. By induction hypothesis, $e_1 = E_2[r]$ for some E_2 such that $E' = E_1'[E_2]$. Note $E_1[E_2] = \langle E_1'[E_2], e_2 \rangle = \langle E', e_2 \rangle = E$, and we are done.
- $E_1 = \langle v, E_1' \rangle$. If E is of form $\langle E', e_2 \rangle$, then $v = E'[r]$. Since this contradicts (1), E must be of form $\langle v, E' \rangle$. By induction hypothesis, $e_1 = E_2[r]$ for some E_2 such that $E' = E_1'[E_2]$ Therefore, $E = E_1[E_2]$, and this concludes the case.

The rest of the cases can be treated similarly. (3) and (4) immediately follow from (2). ∎

Clearly, Proposition 2.3.4 (3) implies that if e can be reduced then there exist a unique evaluation context E and a redex r such that $e = E[r]$. However, r may have different reductions if r is of form **case** v **of** ms.

Theorem 2.3.5 *Given an expression* e *and a value* v *in* $\lambda^{\mathrm{pat}}_{\mathrm{val}}$*,* $e \hookrightarrow_0 v$ *if and only if* $e \mapsto^* v$

Proof We write $e_1 \mapsto^n e_2$ to mean that e_1 evaluates to e_2 in n steps. Assume $e \mapsto^n v$. We prove $e \hookrightarrow_0 v$ by an induction on n and the structure of e, lexicographically ordered. We do a case analysis on the structure of e.

- $e = \langle e_1, e_2 \rangle$. By Proposition 2.3.4 (4), there exists $0 \leq i, j \leq n$ such that $e_1 \mapsto^i v_1$ and $e_2 \mapsto^j v_2$ for some v_1 and v_2. By induction hypothesis, we can derive $e_1 \hookrightarrow_0 v_1$ and $e_2 \hookrightarrow_0 v_2$. This yields the following.

$$\frac{e_1 \hookrightarrow_0 v_1 \quad e_2 \hookrightarrow_0 v_2}{e \hookrightarrow_0 v} \textbf{ (ev-prod)}$$

- $e = e_1(e_2)$. Then there exists $0 \leq i, j < n$ such that $e_1 \mapsto^i v_1$ and $e_2 \mapsto v_2$ for some v_1 and v_2, where v_1 is of form $\textbf{lam}\ x.e_1'$. Hence we have the following.

$$e \mapsto \cdots \mapsto (\textbf{lam}\ x.e_1')(v_2) \mapsto e_1'[x \mapsto v_2] \mapsto \cdots \mapsto v$$

  By induction hypothesis, $e_1 \hookrightarrow_0 \textbf{lam}\ x.e_1'$, $e_2 \hookrightarrow_0 v_2$ and $e_1'[x \mapsto v_2] \hookrightarrow_0 v$ are derivable. This yields the following.

$$\frac{e_1 \hookrightarrow_0 \textbf{lam}\ x.e_1' \quad e_2 \hookrightarrow_0 v_2 \quad e_1'[x \mapsto v_2] \hookrightarrow_0 v}{e \hookrightarrow_0 v} \textbf{ (ev-app)}$$

- $e = \textbf{fix}\ f.u$. Then $e \mapsto u[f \mapsto (\textbf{fix}\ f.u)]$. Clearly, we have the following.

$$\frac{}{e \hookrightarrow_0 u[f \mapsto (\textbf{fix}\ f.u)]} \textbf{ (ev-fix)}$$

All other cases can be treated similarly.

We now assume that $e \hookrightarrow_0 v$ is derivable and prove $e \mapsto^* v$ by a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_0 v$. We present a few cases.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_0 v_1 \quad e_2 \hookrightarrow_0 v_2}{\langle e_1, e_2 \rangle \hookrightarrow_0 \langle v_1, v_2 \rangle}$ By induction hypothesis, We have $e_1 \mapsto^* v_1$ and $e_2 \mapsto^* v_2$. This yields the following since both $\langle [], e_2 \rangle$ and $\langle v_1, [] \rangle$ are evaluation contexts.

$$e = \langle e_1, e_2 \rangle \mapsto^* \langle v_1, e_2 \rangle \mapsto^* \langle v_1, v_2 \rangle$$

$\mathcal{D} = \dfrac{e_0 \hookrightarrow_0 v_0 \quad \textbf{match}(v_0, p_k) \Longrightarrow \theta \text{ for some } 1 \leq k \leq n \quad e_k[\theta] \hookrightarrow_0 v}{(\textbf{case}\ e_0\ \textbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)) \hookrightarrow_0 v}$ By induction hypothesis, we have $e_0 \mapsto^* v_0$ and $e_k[\theta] \mapsto^* v$. This leads to the following.

$$\textbf{case}\ e_0\ \textbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \mapsto^* \textbf{case}\ v_0\ \textbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \mapsto e_k[\theta] \mapsto^* v$$

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_0 (\textbf{lam}\ x.e_1') \quad e_2 \hookrightarrow_0 v_2 \quad e_1'[x \mapsto v_2] \hookrightarrow_0 v}{e_1(e_2) \hookrightarrow_0 v}$ By induction hypothesis, we have $e_1 \mapsto^* (\textbf{lam}\ x.e_1')$, $e_2 \mapsto^* v_2$ and $e_1'[x \mapsto v_2] \mapsto^* v$. This leads to the following.

$$e = e_1(e_2) \mapsto^* (\textbf{lam}\ x.e_1')(e_2) \mapsto^* (\textbf{lam}\ x.e_1')(v_2) \mapsto e_1'[x \mapsto v_2] \hookrightarrow_0 v$$

All other cases can be treated similarly. ∎

We will present elaboration algorithms in Chapter 4 and Chapter 5, which map a program written in an external language into one in an internal language. We will have to show that the elaboration of a program preserves its operational semantics. For this purpose, we introduce the notion of operational equivalence in $\lambda^{\text{pat}}_{\text{val}}$.

Definition 2.3.6 *Given two expression e_1 and e_2 in $\lambda^{\mathrm{pat}}_{\mathrm{val}}$, e_1 is operationally equivalent to e_2 if the following holds.*

- *Given any context C, $C[e_1] \mapsto^* \langle\rangle$ is derivable if and only if $C[e_2] \mapsto^* \langle\rangle$ is.*

We write $e_1 \cong e_2$ if e_1 is operationally equivalent to e_2.

Clearly $\cong$ is an equivalence relation. Our aim is to show that **let** $x = e$ **in** $E[x]$ **end** is operationally equivalent to $E[e]$ for any evaluation context E containing no free occurrences of x. However, this seemingly easy task turns out to be tricky. We will explain the need for the following definition in the proof of Lemma 2.3.11.

Definition 2.3.7 *The extended values and extended evaluation contexts are defined as follows.*

$$\begin{array}{lrcl} \textit{(extended values)} & w & := & x \mid c(w) \mid \langle\rangle \mid \langle w_1, w_2\rangle \mid (\textbf{lam}\ x.e) \mid (\textbf{fix}\ f.u) \\ \textit{(extended evaluation contexts)} & F & := & [] \mid \langle F, e\rangle \mid \langle w, F\rangle \mid c(F) \mid \textbf{case}\ F\ \textbf{of}\ ms \\ & & & \mid F(e) \mid w(F) \mid \textbf{let}\ x = F\ \textbf{in}\ e\ \textbf{end} \end{array}$$

$e_1 \mapsto_F e_2$ if $e_1 = F[e]$ for some F and redex e and $e_2 = F[e']$, where e' is the reduction of e. Let $\mapsto_F^*$ be the reflexive and transitive closure of $\mapsto_F$.

Clearly, the difference between extended values and values is that expression of form **fix** $f.u$ belongs the former but not latter. Informally speaking, it allows us to treat an expression of form **fix** $f.u$ as a value *when an extended evaluation context is formulated.* However, **fix** $f.u$ should not be regarded as a value when a redex is formulated. For instance, (**lam** $x.x$)(**fix** $f.u$) is *not* a redex.

Unlike the evaluation contexts, the extended evaluation contexts do not enjoy Proposition 2.3.4 (3). For instance, given $e = Fix(I(I))$, where $Fix =$ **fix** f.**lam** $x.f(x)$ and $I =$ (**lam** $x.x$), e can be reduced in one step to (**lam** $x.Fix(x))(I(I))$ or to (**fix** f.**lam** $x.f(x))(I)$. The next proposition states some relation between values (evaluation contexts) and extended values (extended evaluation contexts).

Proposition 2.3.8 *We have the following.*

1. *Given any extended value w, $w \mapsto^* v$ for some value v.*
2. *Given any extended evaluation context F and expression e, $F[e] \mapsto^* E[e]$ for some evaluation context E, where E is determined by F.*

Proof (1) follows from a structural induction on w. We present an interesting case.

- w is of form **fix** $f.u$. Then $w \mapsto u[f \mapsto w]$. Since $u[f \mapsto w]$ is a value, we are done.

We prove (2) by a structural induction on F. Here are a few cases.

- F is of form $w(F_1)$. Then by induction hypothesis $F_1[e] \mapsto^* E_1[e]$ for some E_1. By (1), $w \mapsto^* v$ for some v. Hence, we have

$$F[e] = w(F_1[e]) \mapsto^* v(F_1[e]) \mapsto^* v(E_1[e]) = E[e]$$

  for $E = v(E_1)$.

- F is of form **let** $x = F_1$ **in** e_1 **end**. By induction hypothesis, $F_1[e] \mapsto^* E_1[e]$ for some E_1. Hence, we have

$$F[e] = \textbf{let } x = F_1[e] \textbf{ in } e_1 \textbf{ end} \mapsto^* \textbf{let } x = E_1[e] \textbf{ in } e_1 \textbf{ end} = E[e]$$

  for $E = \textbf{let } x = E_1 \textbf{ in } e_1 \textbf{ end}$.

All other cases can be treated similarly. ■

We now relate $\mapsto_F$ to $\mapsto$. Clearly, $e_1 \mapsto e_2$ implies $e_1 \mapsto_F e_2$ since an evaluation context is an extended evaluation context. In the other direction, we have the following.

Lemma 2.3.9 *Given an expression e and a value v in $\lambda^{\mathrm{pat}}_{\mathrm{val}}$, if $e \mapsto^*_F v$ then $e \mapsto^* v$.*

Proof Assume $e \mapsto^n_F v$ and we proceed by an induction on n. If $n = 0$ then it is trivial. Assume $e = F[e_1] \mapsto_F F[e'_1] \mapsto^*_F v$ for some F, where e_1 is a redex and e'_1 is its reduction. By induction hypothesis, $F[e'_1] \mapsto^* v$. Note that $F[e_1] \mapsto^* E[e_1]$ and $F[e'_1] \mapsto^* E[e'_1]$ for some E by Proposition 2.3.8 (2). This leads to

$$e = F[e_1] \mapsto^* E[e_1] \mapsto E[e'_1] \mapsto^* v$$

■

Therefore, the operational semantics of $\lambda^{\mathrm{pat}}_{\mathrm{val}}$ is not affected even if we treat expressions of form **fix** $f.u$ as values *when formulating evaluation contexts.*

Definition 2.3.10 *A β_F-redex r is an expression of form* **let** $x = e$ **in** $F[x]$ **end**, *where there are no free occurrences of x in F. $e_1 \rightarrow_{\beta_F} e_2$ if e_1 is of form $C[r]$ for some β_F-redex $r =$* **let** $x = e$ **in** $F[x]$ **end** *and $e_2 = C[F[e]]$. We write $\rightarrow^*_{\beta_F}$ for the reflexive and transitive closure of $\rightarrow_{\beta_F}$*

Lemma 2.3.11 *Suppose $e_1 \rightarrow_{\beta_F} e_2$. We have the following.*

1. *If $e_1 \mapsto_F e'_1$, then for some e'_2, $e_2 \mapsto^{0/1}_F e'_2$ and $e'_1 \rightarrow^*_{\beta_F} e'_2$, where $e_2 \mapsto^{0/1}_F e'_2$ means either $e_2 = e'_2$ or $e_2 \mapsto_F e'_2$.*
2. *If $e_2 \mapsto_F e'_2$, then either $e_1 \mapsto_F e_2$ or for some e'_1, $e_1 \mapsto_F e'_1$ and $e'_1 \rightarrow^*_{\beta_F} e'_2$.*

Proof For (1), we proceed by a structural induction on e_1.

- e_1 is of form (**fix** $f.u_1$). Then $e_2 =$ (**fix** $f.u_2$) for some u_2 such that $u_1 \rightarrow_{\beta_F} u_2$. Note $e'_1 = u_1[f \mapsto e_1]$. Let $e'_2 = u_2[f \mapsto e_2]$, then $e_2 \mapsto_F e'_2$. If r is a β_F-redex in u_1, then we *observe* that $r[f \mapsto e_1]$ is a β_F-redex in e'_1. This is exactly the case which would not go through if we had not defined the notion of extended evaluation context.

  With this observation, it is not difficult to see that $e'_1 \rightarrow^*_{\beta_F} e'_2$.

All other cases can be handled similarly.

For (2), we also proceed by a structural induction on e_1.

- $e_1 =$ **let** $x = w$ **in** $F[x]$ **end**. Then there are several subcases.

- $e_1 \to_{\beta_F}$ **let** $x = w'$ **in** $F[x]$ **end** $= e_2$, where $w \to_{\beta_F} w'$. We have $e_1 \mapsto_F F[w] \to_{\beta_F} F[w']$ and $e_2 \mapsto_F F[w']$. So $e_2' = F[w']$. Let $e_1' = F[w]$, and we are done.
- $e_1 \to_{\beta_F} F[w] = e_2$. Then $e_1 \mapsto_F e_2$.
- $e_1 \to_{\beta_F}$ **let** $x = w$ **in** $F'[x]$ **end** $= e_2$, where $F[x] \to_{\beta_F} F'[x]$. We have $e_1 \mapsto_F F[w] \to_{\beta_F} F'[w]$ and $e_2 \mapsto_F F'[w]$. So $e_2' = F'[w]$. Let $e_1' = F[w]$ and we are done.

All other cases can be treated similarly. ■

Lemma 2.3.12 *Let e_1 and e_2 be two expressions in $\lambda_{\mathrm{val}}^{\mathrm{pat}}$ such that $e_1 \to_{\beta_F}^* e_2$. We have the following.*

1. *If $e_1 \mapsto_F^* v_1$ for some value v_1, then $e_2 \mapsto_F^* v_2$ for some value v_2 such that $v_1 \to_{\beta_F}^* v_2$.*
2. *If $e_2 \mapsto_F^* v_2$ for some value v_2, then $e_1 \mapsto_F^* v_1$ for some value v_1 such that $v_1 \to_{\beta_F}^* v_2$.*

Proof Assume $e_1 \mapsto_F^n v_1$. We prove (1) by induction on n.

1. $n = 0$. Then this is trivial.
2. $n > 0$. Then $e_1 \mapsto_F e_1' \mapsto_F^* v_1$ for some e_1'. Then by Proposition 2.3.11 (1), there exists e_2' such that $e_2 \mapsto_F^{0/1} e_2'$ and $e_1' \to_{\beta_F}^* e_2'$. By induction hypothesis, $e_2' \mapsto_F^* v_2$ for some value v_2 such that $v_1 \to_{\beta_F}^* v_2$.

Assume $e_2 \mapsto_F^n v_2$. We now prove (2) by induction on n.

1. $n = 0$. Then $e_1 \to_{\beta_F}^m e_2 = v_2$ for some m. It is straightforward to prove that $e_1 \mapsto_F^* v_1$ for some v_1 such that $v_1 \to_{\beta_F}^* v_2$ by induction on m.
2. $n > 0$. Then $e_2 \mapsto_F e_2' \mapsto_F^* v_2$ for some e_2'. Then by Proposition 2.3.11 (2), we have two cases.
   - $e_1 \mapsto_F e_2$. Then $e_1 \mapsto_F^* v_2$. Hence, let $v_1 = v_2$ and we are done.
   - $e_1 \mapsto_F e_1'$ for some e_1' such that $e_1' \to_{\beta_F}^* e_2'$. By induction hypothesis, $e_1' \mapsto_F^* v_1$ for some value v_1 such that $v_1 \to_{\beta_F}^* v_2$.

Therefore, both (1) and (2) hold. ■

Corollary 2.3.13 *For every extended evaluation context F and every expression e in $\lambda_{\mathrm{val}}^{\mathrm{pat}}$,*

$$\textbf{let } x = e \textbf{ in } F[x] \textbf{ end} \cong F[e]$$

holds if x has no occurrences in F.

Proof Notice **let** $x = e$ **in** $F[x]$ **end** is a β_F-redex. Hence, we have

$$C[\textbf{let } x = e \textbf{ in } F[x] \textbf{ end}] \to_{\beta_F} C[F[e]].$$

Suppose $C[\textbf{let } x = e \textbf{ in } F[x] \textbf{ end}] \mapsto^* \langle\rangle$. Then $C[F[e]] \mapsto v$ follows from Proposition 2.3.12 (1) such that $\langle\rangle \mapsto^* v$. Hence $v = \langle\rangle$.

Suppose $C[F[e]] \mapsto^* \langle\rangle$. Then $C[\textbf{let } x = e \textbf{ in } F[x] \textbf{ end}] \mapsto v$ follows from Proposition 2.3.12 (2) such that $v \mapsto^* \langle\rangle$. This implies $v = \langle\rangle$ since v is a value.

Therefore, $\textbf{let } x = e \textbf{ in } F[x] \textbf{ end} \cong F[e]$ by the definition of operational equivalence. ∎

Since an evaluation context is an extended evaluation context, we have derive the following operational equivalence for every evaluation context E in which there are no occurrences of x.

$$\textbf{let } x = e \textbf{ in } E[x] \textbf{ end} \cong E[e]$$

This equivalence will still hold after we extend the language with effects such as references and exceptions, although we will no longer present a proof.

Lastly, we list some properties which can be proven similarly.

Proposition 2.3.14 *we have the following.*

1. $(\textbf{lam } x.(\textbf{lam } y.e)(x)) \cong (\textbf{lam } y.e)$.
2. $(\textbf{fix } f.u) \cong u[f \mapsto (\textbf{fix } f.u)]$.
3. $\textbf{let } x = w \textbf{ in } e \textbf{ end} \cong e[x \mapsto w]$.

The need for introducing extended values and extended evaluation contexts stems from the adoption of the rule **(ev-fix)** in which the non-value $(\textbf{fix } f.u)$ is substituted for a variable f, which is regarded as a value. We now suggest two non-standard alternatives to coping with this problem.

1. The first alternative is that we classify variables into two categories. One category contains the variables which are regarded as values and the other category contains the variables which are not regarded as values. The variables bound by **lam** must be in the first category and the variables bound by **fix** must belong to the second one. This avoids substituting non-values for variables which are regarded as values.

2. The second alternative is to replace the rule **(ev-fix)** with the following evaluation rules. This readily guarantees that only values can be substituted for variables.

$$\frac{}{(\textbf{fix } f.u) \hookrightarrow_0 u[f \mapsto u^*]}$$

   where $u^* = u[f \mapsto (\textbf{fix } f.u)]$. This strategy is clearly justified by Proposition 2.3.14 (2).

2.4 Summary

We started with $\lambda^{\text{pat}}_{\text{val}}$, a untyped λ-calculus with general pattern matching. The importance of $\lambda^{\text{pat}}_{\text{val}}$ lies in its operational semantics, which is given in the style of natural semantics. We then introduced ML_0, the typed version of $\lambda^{\text{pat}}_{\text{val}}$. An important observation at this point is that types play no rôle in program evaluation. As we shall see, this property will be kept valid in all the typed languages that we introduce later in this thesis.

However, we emphasize that recent studies (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996; Morrisett, Walker, Crary, and Glew 1998) have convincingly shown that the use of types can be very helpful for detecting errors in compiler writing and enhance the performance of compiled

code. We will actually demonstrate in Chapter 9 that dependent types can indeed lead to more efficient code.

In addition, we studied the operation equivalence relation in $\lambda^{\text{pat}}_{\text{val}}$, which will be used later to prove the correctness of some type-checking algorithms. We are now ready to introduce dependent types into ML_0.

Chapter 3

Constraint Domains

Our enriched language will be parameterized over a constraint domain, from which the type index objects are drawn. Typical examples of constraints include linear equalities and inequalities over integers, equations over the algebraic terms (also called the Herbrand domain), first-order logic formulas over a finite domain, etc. Much of the work in this chapter is inspired and closely related to the CLP (Constraint Logic Programming) languages presented in (Jaffar and Maher 1994).

3.1 The General Constraint Language

We emphasize that the general constraint language itself is typed. In order to avoid potential confusion we call the types in the constraint language *index sorts*. We use b for base index sorts such as o for propositions and *int* for integers. A signature Σ declares a set of function symbols and associates with every function symbol an *index sort* defined below. A Σ-*structure* $\mathcal{D}$ consists of a set $\mathbf{dom}(\mathcal{D})$ and an assignment of functions to the function symbols in Σ.

We use f for interpreted function symbols, p for atomic predicates (that is, functions of sort $\gamma \to o$) and we assume we have constants such as equality, truth values $\top$ and $\bot$, conjunction $\wedge$, and disjunction $\vee$, all of which are interpreted as usual.

$$\begin{array}{lrcl} \text{index sorts} & \gamma & ::= & b \mid \mathbf{1} \mid \gamma_1 * \gamma_2 \mid \{a : \gamma \mid P\} \\ \text{index propositions} & P & ::= & \top \mid \bot \mid p(i) \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \end{array}$$

Here $\{a : \gamma \mid P\}$ is the subset index sort for those elements of index sort γ satisfying proposition P, where P is an index proposition. For instance, *nat* is an abbreviation for $\{a : \mathit{int} \mid a \geq 0\}$, that is, *nat* is a subset index sort of *int*.

We use a for index variables in the following formulation. We assume that there exists a predicate $\doteq$ of sort $\gamma * \gamma \to o$ for every index sort γ, which is interpreted as equality. Also we emphasize that all function symbols declared in Σ must be associated with index sorts of form $\gamma \to b$ or b. In other words, the constraint language is first-order.

$$\begin{array}{lrcl} \text{index objects} & i, j & ::= & a \mid \mathrm{f}(i) \mid \langle\rangle \mid \langle i, j\rangle \mid \mathbf{fst}(i) \mid \mathbf{snd}(i) \\ \text{index contexts} & \phi & ::= & \cdot \mid \phi, a : \gamma \mid \phi, P \\ \text{index constraints} & \Phi & ::= & i \doteq j \mid \top \mid \Phi_1 \wedge \Phi_2 \mid P \supset \Phi \mid \forall a : \gamma.\Phi \mid \exists a : \gamma.\Phi \\ \text{index substitutions} & \theta & ::= & [] \mid \theta[a \mapsto i] \\ \text{satisfiability relation} & & & \phi \models \Phi \end{array}$$

An index variable can be declared at most once in an index context. The domain of an index context is defined as follows.

$$\mathbf{dom}(\cdot) = \emptyset \qquad \mathbf{dom}(\phi, a : \gamma) = \mathbf{dom}(\phi) \cup \{a\} \qquad \mathbf{dom}(\phi, P) = \mathbf{dom}(\phi)$$

Also $\phi(a) = \gamma$ for every $a \in \mathbf{dom}(\phi)$ if $a : \gamma$ is declared in ϕ. A judgement of the form $\phi \vdash \theta : \phi'$ can be derived with the use of the following rules.

$$\frac{}{\phi \vdash [] : \cdot}\ \textbf{(subst-iempty)}$$

$$\frac{\phi \vdash \theta : \phi' \quad \phi \vdash i : \gamma}{\phi \vdash \theta[a \mapsto i] : \phi', a : \gamma}\ \textbf{(subst-ivar)}$$

$$\frac{\phi \vdash \theta : \theta' \quad \phi \models P[\theta]}{\phi \vdash \theta : \theta', P}\ \textbf{(subst-prop)}$$

Proposition 3.1.1 *If $\phi \vdash \theta : \phi'$ is derivable, then $\mathbf{dom}(\theta) = \mathbf{dom}(\phi')$ and $\phi \vdash \theta(a) : \phi'(a)$ is derivable for every $a \in \mathbf{dom}(\theta)$.*

Proof This simply follows from a structural induction on the derivation of $\phi \vdash \theta : \phi'$, parallel to that of Proposition 2.2.3. ∎

We present the sort formation and sorting rules for type index objects in Figure 3.1. We explain the meanings of these judgements as follows. A judgement of form $\vdash \phi[\mathbf{ictx}]$ means that ϕ is a valid index context, and a judgement of form $\phi \vdash \gamma : *_s$ means that γ is a valid sort under ϕ, and a judgement of form $\phi \vdash i : \gamma$ means that i is of sort γ under ϕ. Since the constraint language is explicitly sorted, sort-checking can be done straightforwardly following the presented sorting rules. Details on sort-checking, which involves constraint satisfaction, can be found in Subsection 4.2.6.

We could certainly allow *any* first-order logic formula to be a constraint. However, in practice, we often consider a subset of formulas closed under the above definition to be constraints. We use $\mathcal{L}$ for a class of Σ-formulas (constraints), and we call the pair $\langle \mathcal{D}, \mathcal{L} \rangle$ a *constraint domain*, where $\mathcal{D}$ is a Σ-structure. Sometimes, we simply use C for a constraint domain.

We define $(\phi)\Phi$ as follows.

$$\begin{aligned}
&(\cdot)\Phi = \Phi \\
&(a : b)\Phi = \forall a : b.\Phi \\
&(a : \gamma_1 * \gamma_2)\Phi = (a_1 : \gamma_1)(a_2 : \gamma_2)\Phi[a \mapsto \langle a_1, a_2 \rangle] \\
&(\phi, \{a : \gamma \mid P\})\Phi = (\phi)(a : \gamma)(P \supset \Phi) \\
&(\phi, P)\Phi = (\phi)(P \supset \Phi)
\end{aligned}$$

We say that $\phi \models \Phi$ is satisfiable in $C = \langle \mathcal{D}, \mathcal{L} \rangle$ if $(\phi)\Phi$ is true in $\mathcal{D}$ in the model-theoretic sense, that is, the interpretation of $(\phi)\Phi$ in $\mathcal{D}$ is true.

We also present some basic rules for reasoning about the satisfiability of $\phi \models \Phi$ as follows. Note that there also exist other rules such as induction and model checking, which are associated with certain special constraint domains.

$$\frac{}{\vdash \cdot[\mathbf{ictx}]}\ \textbf{(ictx-empty)}$$

$$\frac{\vdash \phi[\mathbf{ictx}] \quad \phi \vdash \gamma : *_s}{\vdash \phi, a : \gamma[\mathbf{ictx}]}\ \textbf{(ictx-ivar)}$$

$$\frac{\vdash \phi[\mathbf{ictx}]}{\phi \vdash b : *_s}\ \textbf{(sort-base)}$$

$$\frac{\vdash \phi[\mathbf{ictx}]}{\phi \vdash \mathbf{1} : *_s}\ \textbf{(sort-unit)}$$

$$\frac{\phi \vdash \gamma_1 : *_s \quad \phi \vdash \gamma_2 : *_s}{\phi \vdash \gamma_1 * \gamma_2 : *_s}\ \textbf{(sort-prod)}$$

$$\frac{\phi \vdash \gamma : *_s \quad \phi, a : \gamma \vdash P : o}{\phi \vdash \{a : \gamma \mid P\} : *_s}\ \textbf{(sort-subset)}$$

$$\frac{\vdash \phi[\mathbf{ictx}] \quad \phi(a) = \gamma}{\phi \vdash a : \gamma}\ \textbf{(index-var)}$$

$$\frac{\vdash \phi[\mathbf{ictx}]}{\phi \vdash \langle\rangle : \mathbf{1}}\ \textbf{(index-unit)}$$

$$\frac{\phi \vdash i_1 : \gamma_1 \quad \phi \vdash i_2 : \gamma_2}{\phi \vdash \langle i_1, i_2 \rangle : \gamma_1 * \gamma_2}\ \textbf{(index-prod)}$$

$$\frac{\phi \vdash i : \gamma_1 * \gamma_2}{\phi \vdash \mathbf{fst}(i) : \gamma_1}\ \textbf{(index-first)}$$

$$\frac{\phi \vdash i : \gamma_1 * \gamma_2}{\phi \vdash \mathbf{snd}(i) : \gamma_2}\ \textbf{(index-second)}$$

$$\frac{\phi \vdash a_1 : \{a_2 : \gamma \mid P\}}{\phi \vdash a_1 : \gamma}\ \textbf{(index-var-subset)}$$

$$\frac{\phi \vdash i : \gamma \quad \phi, a : \gamma \vdash P : o \quad \phi \models P[a \mapsto i]}{\phi \vdash i : \{a : \gamma \mid P\}}\ \textbf{(index-subset)}$$

$$\frac{\Sigma(\mathrm{f}) = b}{\phi \vdash \mathrm{f} : b}\ \textbf{(index-cons)}$$

$$\frac{\Sigma(\mathrm{f}) = \gamma \to b \quad \phi \vdash i : \gamma}{\phi \vdash \mathrm{f}(i) : b}\ \textbf{(index-fun)}$$

Figure 3.1: The sort formation and sorting rules for type index objects

$$\frac{\phi \models \Phi_1 \quad \phi \models \Phi_2}{\phi \models \Phi_1 \wedge \Phi_2}\ \textbf{(sat-conj)} \qquad \frac{\phi, P \models \Phi}{\phi \models P \supset \Phi}\ \textbf{(sat-impl)}$$

$$\frac{\phi, a : \gamma \models \Phi}{\phi \models \forall a : \gamma.\Phi}\ \textbf{(sat-forall)} \qquad \frac{\phi \models \Phi[a \mapsto i] \quad \phi \vdash i : \gamma}{\phi \models \exists a : \gamma.\Phi}\ \textbf{(sat-exists)}$$

Clearly, these rules are not enough. We have to be able to verify the derivability of a satisfiability relation of form $\phi \models P$. We say that $\phi \models P$ is derivable in a constraint domain $C = \langle \mathcal{D}, \mathcal{L} \rangle$ if $(\phi)P$ is satisfiable in $\mathbf{dom}(\mathcal{D})$. In order to verify whether $(\phi)P$ is satisfiable in $\mathcal{D}$, one may use some special methods associated with C such as model-checking for finite domains. We can readily prove that $(\phi)\Phi$ is satisfiable if $\phi \models \Phi$ is derivable. This establishes the soundness of this approach to solving constraints. Clearly, this may not be a complete approach. For instance, even if $\exists a : \gamma.\Phi$ is satisfiable in $\mathbf{dom}(\mathcal{D})$, there may not exist an index i expressible in the constraint language such that $\Phi[a \mapsto i]$ is satisfiable. Also, the special methods employed to verify the the satisfiability of $(\phi)P$ may not be complete.

Proposition 3.1.2 *We have the following.*

1. *If both $\phi \models P$ and $\phi, P \models \Phi$ are derivable, then $\phi \models \Phi$ is derivable.*
2. *If both $\phi \vdash i : \gamma$ and $\phi, a : \gamma \models \Phi$ are derivable, then $\phi \models \Phi[a \mapsto i]$ is also derivable.*
3. *If both $\phi \vdash \theta : \phi'$ and $\phi, \phi' \models \Phi$ are derivable, then $\phi \models \Phi[\theta]$ is also derivable.*

Proof All these are straightforward. ∎

Note that the rule **(sat-exists)** is *not* syntax-directed. This could be a serious problem which hinders the efficiency of a constraint solver. In Subsection 4.2.6, we will introduce a procedure which eliminates existential variables in the constraints generated during type-checking. In the prototype implementation, we simply reject a constraint if some existential quantifiers in it cannot be eliminated. The practical significance of this decision is to make constraint solving as feasible as possible for typical use. Another important reason is that this can significantly help generate comprehensible error messages as our experience indicates.

Not much of our development depends on the precise form of the constraint domain, except that the constructs above must be present in order to reduce dependent type-checking to constraint satisfaction. For example, implication $P \supset \Phi$ is necessary to express constraints arising from pattern matching. Though subset sorts $\{a : \gamma \mid P\}$ are not strictly required in the formulation of the type system, they are crucial to making the system expressive enough for practical use.

3.2 A Constraint Domain over Algebraic Terms

We present a constraint domain over algebraic terms. In the signature Σ_{alg} of this domain, a declaration is of form $f : b_1 * \cdots * b_n \to b$. If it is preferred to have an unsorted constraint domain, then one can assume that there is only one base sort *term*, which stands for the sort of all terms.

Let us present an interesting example, in which the type index objects are drawn from Σ_{alg}. We use the following datatype to represent pure untyped lambda-terms in de Bruijn's notation.

$$\frac{a \in \mathbf{dom}(\phi_0)}{\cdot \models[\phi_0]\ a \doteq a} \qquad \frac{\cdot \models[\phi_0]\ i_1 \doteq j_1 \ \cdots \ \cdot \models[\phi_0]\ i_n \doteq j_n}{\cdot \models[\phi_0]\ f(i_1,\ldots,i_n) \doteq f(j_1,\ldots,j_n)}$$

$$\frac{\cdot \models[\phi_0]\ P_1}{\cdot \models[\phi_0]\ P_1 \vee P_2} \qquad \frac{\cdot \models[\phi_0]\ P_2}{\cdot \models[\phi_0]\ P_1 \vee P_2} \qquad \frac{P_1 \models[\phi_0]\ P_2}{\cdot \models[\phi_0]\ P_1 \supset P_2}$$

$$\frac{P_1, P_2, \phi_p \models[\phi_0]\ P}{P_1 \wedge P_2, \phi_p \models[\phi_0]\ P} \qquad \frac{i_1 \doteq j_1, \ldots, i_n \doteq j_n, \phi_p \models[\phi_0]\ P}{f(i_1,\ldots,i_n) \doteq f(j_1,\ldots,j_n), \phi_p \models[\phi_0]\ P}$$

$$\frac{\phi_p[a \mapsto i] \models[\phi_0]\ P[a \mapsto i]}{a \doteq i, \phi_p \models[\phi_0]\ P} \qquad \frac{\phi_p[a \mapsto i] \models[\phi_0]\ P[a \mapsto i]}{i \doteq a, \phi_p \models[\phi_0]\ P}$$

$$\frac{P_1, \phi_p \models[\phi_0]\ P \quad P_2, \phi_p \models[\phi_0]\ P}{P_1 \vee P_2, \phi_p \models[\phi_0]\ P} \qquad \frac{\cdot \models[\phi_0, a:b]\ P}{\cdot \models[\phi_0]\ \forall a:b.P}$$

Figure 3.2: The rules for satisfiability verification

```
datatype lambda_term = One | Shift of lambda_term |
                       Abs of lambda_term |
                       App of lambda_term * lambda_term
```

Suppose that there is a base sort *level*, and the following function symbols are declared in Σ_{alg}.

$$zero : level \quad \text{and} \quad next : level \to level$$

This enables us to refine the datatype `lambda_term` into the following dependent type.

```
typeref lambda_term of level
with One <| {l:level} lambda_term(next(l))
   | Shift <| {l:level} lambda_term(l) -> lambda_term(next(l))
   | Abs <| {l:level} lambda_term(next(l)) -> lambda_term(l)
   | App <| {l:level} lambda_term(l) * lambda_term(l) -> lambda_term(l)
```

Roughly speaking, if the de Bruijn's notation of a λ-term is of type `lambda_term(l)`, where $l = next(\cdots(zero)\cdots)$ contains n occurrences of *next*, then there are at most n free variables in the λ-term. Therefore, the type of all *closed* λ-terms is `lambda_term(zero)`.

This is a very simple constraint domain. Given ϕ and P, the rules in Figure 3.2 can be used verify if $(\phi)P$ is satisfiable. Notice that ϕ_0 and ϕ_p are index contexts of forms $a_1 : b_1, \ldots, a_n : b_n$ and $P_1, \ldots, P_n$, respectively. We say that $(\phi)P$ is satisfiable if $\cdot \models[\cdot]\ (\phi)P$ is derivable. It is clear that Σ_{alg} should not to be fixed so that the programmer can then be allowed to declare the sorts of function symbols. The simple reason for this is that the rules for satisfiability verification in this

domain are not effected by such declarations. The following is a sample derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \models [a:level, b:level]\ b \doteq b}}{a \doteq b \models [a:level, b:level]\ a \doteq b}}{next(a) \doteq next(b) \models [a:level, b:level]\ a \doteq b}}{\cdot \models [a:level, b:level]\ next(a) \doteq next(b) \supset a \doteq b}}{\cdot \models [a:level]\ \forall (b:level).next(a) \doteq next(b) \supset a \doteq b}}{\cdot \models [\cdot]\ \forall (a:level)\forall (b:level).next(a) \doteq next(b) \supset a \doteq b}$$

Lastly, we remark that if disequations are allowed in this constraint domain then the rules for satisfiability verification can be extended straightforwardly.

3.3 A Constraint Domain over Integers

We present an integer constraint domain in this section. The signature of the domain is given in Figure 3.3. We also list some sample constraints in Figure 3.4, which are generated during type-checking the binary search program in Figure 1.3.

Unfortunately, there exist no practical constraint solving algorithms for this constraint domain in its full generality. This poses a very serious problem since our objective is to design a dependent type system for general purpose practical programming. In Subsection 4.2.6, a procedure is introduced to eliminate existential quantifiers in constraints generated during type-checking. We currently simply reject a constraint if some existential quantifiers in it cannot be eliminated. Therefore, the constraints which are finally passed to a constraint solver consist of only linear inequalities, for which there exist practical solvers.

3.3.1 A Constraint Solver for Linear Inequalities

When all existential variables have been eliminated (Subsection 4.2.6) and the resulting constraints collected, we check them for linearity. We currently reject non-linear constraints rather than postponing them as *hard constraints* (Michaylov 1992), which is planned for future work. If the constraints are linear, we negate them and test for unsatisfiability. Our technique for solving linear constraints is mainly based on Fourier-Motzkin variable elimination (Dantzig and Eaves 1973), but there are many other methods available for this purpose such as the SUP-INF method (Shostak 1977) and the well-known simplex method. We have chosen Fourier-Motzkin's method mainly for its simplicity.

We now briefly explain this method. We use x for integer variables, a for integers, and l for linear expressions. Given a set of inequalities S, we would like to show that S is unsatisfiable. We fix a variable x and transform all the linear inequalities into one of the forms $l \leq ax$ or $ax \leq l$ for $a \geq 0$. For every pair $l_1 \leq a_1 x$ and $a_2 x \leq l_2$, where $a_1, a_2 > 0$, we introduce a new inequality $a_2 l_1 \leq a_1 l_2$ into S, and then remove from S all the inequalities involving x. Clearly, this is a sound but incomplete procedure. If x were a real variable, then the elimination would also be complete.

In order to handle modular arithmetic, we also perform another operation to rule out non-integer solutions: we transform an inequality of form

$$a_1 x_1 + \cdots + a_n x_n \leq a$$

$$
\begin{array}{rcrcl}
\Sigma_{\text{int}} & = & abs & : & int \to int \\
 & & sgn & : & int \to int \\
 & & succ & : & int \to int \\
 & & pred & : & int \to int \\
 & & \sim & : & int \to int \\
 & & + & : & int * int \to int \\
 & & - & : & int * int \to int \\
 & & * & : & int * int \to int \\
 & & div & : & int * int \to int \\
 & & min & : & int * int \to int \\
 & & max & : & int * int \to int \\
 & & mod & : & int * int \to int \\
 & & < & : & int * int \to o \\
 & & \leq & : & int * int \to o \\
 & & = & : & int * int \to o \\
 & & \geq & : & int * int \to o \\
 & & > & : & int * int \to o \\
 & & \neq & : & int * int \to o
\end{array}
$$

Figure 3.3: The signature of the integer domain

$$\forall h : int.\forall l : nat.\forall size : nat.(0 \leq h+1 \leq size \wedge 0 \leq l \leq size \wedge h \geq l) \supset (l + (h-l)/2) \leq size$$
$$\forall h : int.\forall l : nat.\forall size : nat.(0 \leq h+1 \leq size \wedge 0 \leq l \leq size \wedge h \geq l) \supset 0 \leq l + (h-l)/2 - 1 + 1$$
$$\forall h : int.\forall l : nat.\forall size : nat.(0 \leq h+1 \leq size \wedge 0 \leq l \leq size \wedge h \geq l) \supset l + (h-l)/2 - 1 + 1 \leq size$$
$$\forall h : int.\forall l : nat.\forall size : nat.(0 \leq h+1 \leq size \wedge 0 \leq l \leq size \wedge h \geq l) \supset 0 \leq l + (h-l)/2 + 1$$
$$\forall h : int.\forall l : nat.\forall size : nat.(0 \leq h+1 \leq size \wedge 0 \leq l \leq size \wedge h \geq l) \supset l + (h-l)/2 + 1 \leq size$$

Figure 3.4: Sample constraints

into

$$a_1x_1 + \cdots + a_nx_n \leq a',$$

where a' is the largest integer such that $a' \leq a$ and the greatest common divisor of $a_1, \ldots, a_n$ divides a'. This is used in type-checking an optimized byte copy function in Section A.5.

The above elimination method can be extended to be both sound and complete while remaining practical (see, for example, (Pugh and Wonnacott 1992; Pugh and Wonnacott 1994)). We hope to use such more sophisticated methods which still appear to be practical, although we have not yet found the need to do so in the context of our current experiments.

3.3.2 An Example

We show how the following constraint is solved with the above approach.

$$\forall h : int.\forall l : nat.\forall size : nat.(0 \leq h+1 \leq size \wedge 0 \leq l \leq size \wedge h \geq l) \supset l + (h-l)/2 + 1 \leq size$$

The first step is to negate the constraint and transform it into the following form.

$$l \geq 0 \quad size \geq 0 \quad 0 \leq h+1 \quad h+1 \leq size \quad l \leq size \quad h \geq l \quad l + (h-l)/2 + 1 > size$$

Then we replace $(h-l)/2$ with D and add $h-l-1 \leq 2D \leq h-l$ into the set of linear inequalities. We now test for the unsatisfiability of the following set of linear inequalities.

$$\begin{gathered} l \geq 0 \quad size \geq 0 \quad 0 \leq h+1 \quad h+1 \leq size \quad l \leq size \quad h \geq l \\ h-l-1 \leq 2D \quad 2D \leq h-l \quad l + D \geq size \end{gathered}$$

We now eliminate variable *size*, yielding the following set of inequalities.

$$\begin{gathered} l \geq 0 \quad l + D \geq 0 \quad 0 \leq h+1 \quad h+1 \leq l+D \quad l \leq l+D \quad h \geq l \\ h-l-1 \leq 2D \quad 2D \leq h-l \end{gathered}$$

We then eliminate variable D and generate the following set of inequalities.

$$l \geq 0 \quad -2l \leq h-l \quad 0 \leq h+1 \quad 2h-2l+2 \leq h-l \quad 0 \leq h-l \quad h \geq l \quad h-l-1 \leq h-l$$

If we eliminate variable h at this stage, the inequality $l \leq l-1$ is then produced, which leads to a contradiction. Therefore, the original constraint has been verified.

The Fourier variable elimination method can be expensive in practice. We refer the reader to (Pugh and Wonnacott 1994) for a detailed analysis on this issue. However, we feel that this method is intuitive and therefore can facilitate informative type error message report if some constraints can not be verified.

We have observed that an overwhelming majority of the constraints gathered in practice are trivial ones and can be solved with a sound and highly efficient (but incomplete) constraint solver such as one based on the simplex method for *reals*. Therefore, a promising strategy is to use such an efficient constraint solver to filter out trivial constraints and then use a sound and complete (but relatively slow) constraint solver to handle the rest of the constraints.

3.4 Summary

In this chapter, we have presented a general constraint language in which constraint domains can be constructed. It will soon be clear that the dependent type system that we develop parameterizes over a given constraint domain. The ability to find a practical constraint solver for a constraint domain is crucial to making type-checking feasible in the dependent type system parameterizing over it.

At this moment, there is no mechanism to allow the user to define a constraint solver for a declared constraint domain. Some study on formulating such a mechanism can be found in (Frühwirth 1992). Also there is a great deal of study on how to define constraint solvers and make them more efficient in the *constraint logic programming* community, and (Jaffar and Maher 1994) is an excellent source to draw inspiration from.

Chapter 4

Universal Dependent Types

In this chapter we enrich the type system of ML_0 with universal dependent types, yielding a language $\mathrm{ML}_0^{\Pi}(C)$, where C is some fixed constraint domain. We then present the typing rules and operational semantics for $\mathrm{ML}_0^{\Pi}(C)$ and prove some crucial properties, which include the type preservation theorem and the relation between the operational semantics of $\mathrm{ML}_0^{\Pi}(C)$ and that of ML_0. Also we prove that $\mathrm{ML}_0^{\Pi}(C)$ is a conservative extension of ML_0.

In order to make $\mathrm{ML}_0^{\Pi}(C)$ a practical programming language, we design an external language $\mathrm{DML}_0(C)$ for $\mathrm{ML}_0^{\Pi}(C)$. We address the issue of unobtrusiveness of programming in $\mathrm{DML}_0(C)$ through an elaboration mapping which maps a program in $\mathrm{DML}_0(C)$ into one in $\mathrm{ML}_0^{\Pi}(C)$. We then prove the correctness of the elaboration. This elaboration process, which reduces type-checking a program into constraint satisfaction, accounts for a major contribution of the thesis. Finally, we use a concrete example to illustrate the elaboration in full details since it is a considerably involved process.

This extension primarily serves as the core of the language that we will eventually develop, and it also demonstrates cleanly the language design approach we take for making dependent types available in practical programming.

4.1 Universal Dependent Types

We now present $\mathrm{ML}_0^{\Pi}(C)$, which is an extension of ML_0 with universal dependent types. Given a constraint domain C, the syntax of $\mathrm{ML}_0^{\Pi}(C)$ is given in Figure 4.1. We use δ for base type families, where we use $\delta(\langle\rangle)$ for an unindexed type. Type and context formation rules are listed in Figure 4.2. A judgement of form $\phi \vdash \tau : *$ means that τ is a well-formed type under index context ϕ, and a judgement of form $\phi \vdash \Gamma[\mathbf{ctx}]$ means that Γ is a well-formed context under ϕ. Notice that a major type is a type which does not begin with a quantifier.

The domains of Γ and ϕ are defined as usual. Note that every substitution θ can be thought of as the union of two substitutions θ_ϕ and θ_Γ, where $\mathbf{dom}(\theta_\phi)$ contains only index variables and $\mathbf{dom}(\theta_\Gamma)$ contains only (ordinary) variables.

We do not specify here how new type families or constructor types are actually declared, but assume only that they can be processed into the form given above. Our implementation provides indexed refinement of datatype declarations as shown in Section 1.1. The syntax for such declarations will be mentioned in Chapter 8.

families	δ	::=	(family of refined datatypes)
signature	$\mathcal{S}$	::=	$\cdot_{\mathcal{S}} \mid S, \delta : \gamma \to *$
			$\mid \mathcal{S}, c : \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n . \delta(i)$
			$\mid \mathcal{S}, c : \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n . \tau \to \delta(i)$
major types	μ	::=	$\delta(i) \mid \mathbf{1} \mid (\tau_1 * \tau_2) \mid (\tau_1 \to \tau_2)$
types	τ	::=	$\mu \mid (\Pi a : \gamma . \tau)$
patterns	p	::=	$x \mid c[a_1] \ldots [a_n] \mid c[a_1] \ldots [a_n](p) \mid \langle\rangle \mid \langle p_1, p_2 \rangle$
matches	ms	::=	$(p \Rightarrow e) \mid (p \Rightarrow e \mid ms)$
expressions	e	::=	$x \mid \langle\rangle \mid \langle e_1, e_2 \rangle \mid c[i_1] \ldots [i_n] \mid c[i_1] \ldots [i_n](e)$
			$\mid (\mathbf{case}\ e\ \mathbf{of}\ ms) \mid (\mathbf{lam}\ x : \tau . e) \mid e_1(e_2)$
			$\mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \mid (\mathbf{fix}\ f : \tau . u)$
			$\mid (\lambda a : \gamma . e) \mid e[i]$
value forms	u	::=	$c[i_1] \ldots [i_n] \mid c[i_1] \ldots [i_n](u) \mid \langle\rangle \mid \langle u_1, u_2 \rangle$
			$\mid (\mathbf{lam}\ x : \tau . e) \mid (\lambda a : \gamma . u)$
values	v	::=	$x \mid c[i_1] \ldots [i_n] \mid c[i_1] \ldots [i_n](v) \mid \langle\rangle \mid \langle v_1, v_2 \rangle$
			$\mid (\mathbf{lam}\ x : \tau . e) \mid (\lambda a : \gamma . v)$
contexts	Γ	::=	$\cdot \mid \Gamma, x : \tau$
index contexts	ϕ	::=	$\cdot \mid \phi, a : \gamma \mid \phi, P$
substitutions	θ	::=	$[] \mid \theta[x \mapsto e] \mid \theta[a \mapsto i]$

Figure 4.1: The syntax for $\mathrm{ML}_0^{\Pi}(C)$

$$\frac{\mathcal{S}(\delta) = \gamma \to * \quad \phi \vdash i : \gamma}{\phi \vdash \delta(i) : *}\ \textbf{(type-datatype)} \qquad \frac{\phi \vdash \tau_1 : * \quad \phi \vdash \tau_2 : *}{\phi \vdash \tau_1 \Rightarrow \tau_2 : *}\ \textbf{(type-match)}$$

$$\frac{\vdash \phi[\mathbf{ictx}]}{\phi \vdash \mathbf{1} : *}\ \textbf{(type-unit)} \qquad \frac{\phi \vdash \tau_1 : * \quad \phi \vdash \tau_2 : *}{\phi \vdash \langle \tau_1, \tau_2 \rangle : *}\ \textbf{(type-prod)}$$

$$\frac{\phi \vdash \tau_1 : * \quad \phi \vdash \tau_2 : *}{\phi \vdash \tau_1 \to \tau_2 : *}\ \textbf{(type-fun)} \qquad \frac{\phi, a : \gamma \vdash \tau}{\phi \vdash \Pi a : \gamma . \tau}\ \textbf{(type-pi)}$$

$$\frac{}{\phi \vdash \cdot[\mathbf{ctx}]}\ \textbf{(ctx-empty)} \qquad \frac{\phi \vdash \Gamma[\mathbf{ctx}] \quad \phi \vdash \tau : *}{\phi \vdash \Gamma, x : \tau[\mathbf{ctx}]}\ \textbf{(ctx-var)}$$

Figure 4.2: The type formation rules for ML_0

$$\frac{}{x \downarrow \tau \rhd (\cdot; x:\tau)}\ \textbf{(pat-var)}$$

$$\frac{}{\langle\rangle \downarrow \mathbf{1} \rhd (\cdot;\cdot)}\ \textbf{(pat-unit)}$$

$$\frac{p_1 \downarrow \tau_1 \rhd (\phi_1;\Gamma_1) \quad p_2 \downarrow \tau_2 \rhd (\phi_2;\Gamma_2)}{\langle p_1, p_2\rangle \downarrow \tau_1 * \tau_2 \rhd (\phi_1,\phi_2;\Gamma_1,\Gamma_2)}\ \textbf{(pat-prod)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1:\gamma_1 \ldots \Pi a_n:\gamma_n.\delta(i)}{c[a_1]\ldots[a_n] \downarrow \delta(j) \rhd (a_1:\gamma_1,\ldots,a_n:\gamma_n, i \doteq j;\cdot)}\ \textbf{(pat-cons-wo)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1:\gamma_1 \ldots \Pi a_n:\gamma_n.(\tau \to \delta(i)) \quad p \downarrow \tau \rhd (\phi;\Gamma)}{c[a_1]\ldots[a_n](p) \downarrow \delta(j) \rhd (a_1:\gamma_1,\ldots,a_n:\gamma_n, i \doteq j, \phi;\Gamma)}\ \textbf{(pat-cons-w)}$$

Figure 4.3: Typing rules for patterns

4.1.1 Static Semantics

We start with the typing rules for patterns, which are listed in Figure 4.3. The judgment $p \downarrow \tau \rhd (\phi;\Gamma)$ expresses that the index and ordinary variables in pattern p have the types declared in ϕ and Γ, respectively, if we know that p must have type τ.

We write $\phi \models \tau \equiv \tau'$ for the congruent extension of $\phi \models i \doteq j$ from index objects to types, which is determined by the following rules.

$$\frac{\phi \models i \doteq j}{\phi \models \delta(i) \equiv \delta(j)} \qquad \frac{\phi \models \tau_1 \equiv \tau_1' \quad \phi \models \tau_2 \equiv \tau_2'}{\phi \models \tau_1 * \tau_2 \equiv \tau_1' * \tau_2'}$$

$$\frac{\phi \models \tau_1' \equiv \tau_1 \quad \phi \models \tau_2 \equiv \tau_2'}{\phi \models \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \frac{\phi, a:\gamma \models \tau \equiv \tau'}{\phi \models \Pi a:\gamma.\tau \equiv \Pi a:\gamma.\tau'}$$

Proposition 4.1.1 *If both $\phi \vdash \theta : \phi'$ and $\phi, \phi' \models \tau_1 \equiv \tau_2$ are derivable, then $\phi \models \tau_1[\theta] \equiv \tau_2[\theta]$ is also derivable.*

Proof This simply follows from a structural induction on the derivation of $\phi \models \tau_1 \equiv \tau_2$, with the application of Proposition 3.1.2 (3). ∎

We now present the typing rules for $\mathrm{ML}_0^{\Pi}(C)$ in Figure 4.4. We require that there be no free occurrences of a in $\Gamma(x)$ for every $x \in \mathbf{dom}(\Gamma)$ when the rule **(ty-ilam)** is applied. Also note that one premise $\phi \vdash \tau_2 : *$ of the rule **(ty-match)** enforces that all index variables in τ are declared in ϕ. The rule **(ty-cons-wo)** applies only if c is a constructor without an argument. If c is with one argument, the rule **(ty-cons-w)** applies.

Proposition 4.1.2 *(Inversion) If $\phi;\Gamma \vdash e : \tau$ is derivable, then the last inference rule of any derivation of $\phi;\Gamma \vdash e : \tau$ is either* **(ty-eq)** *or uniquely determined by the structure of e.*

Proof By an inspection of all the typing rules in Figure 4.4. ∎

This proposition will be frequently used to do structural induction on typing derivations since it allows us to determinate the last applied rule in such derivations.

$$\frac{\phi;\Gamma \vdash e : \tau_1 \quad \phi \models \tau_1 \equiv \tau_2}{\phi;\Gamma \vdash e : \tau_2}\ \textbf{(ty-eq)}$$

$$\frac{\phi \vdash \Gamma[\mathbf{ctx}] \quad \Gamma(x) = \tau}{\phi;\Gamma \vdash x : \tau}\ \textbf{(ty-var)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.\delta(i) \quad \phi \vdash i_1 : \gamma_1 \ \cdots \ \phi \vdash i_n : \gamma_n \quad \phi \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash c[i_1]\ldots[i_n] : \delta(i[a_1,\ldots,a_n \mapsto i_1,\ldots,i_n])}\ \textbf{(ty-cons-wo)}$$

$$\frac{\begin{array}{c}\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots a_n : \gamma_n.\tau \to \delta(i) \\ \phi \vdash i_1 : \gamma_1 \ \cdots \ \phi \vdash i_n : \gamma_n \quad \phi;\Gamma \vdash e : \tau[a_1,\ldots,a_n \mapsto i_1,\ldots,i_n]\end{array}}{\phi;\Gamma \vdash c[i_1]\ldots[i_n](e) : \delta(i[a_1,\ldots,a_n \mapsto i_1,\ldots,i_n])}\ \textbf{(ty-cons-w)}$$

$$\frac{\phi \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash \langle\rangle : \mathbf{1}}\ \textbf{(ty-unit)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \tau_1 \quad \phi;\Gamma \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 * \tau_2}\ \textbf{(ty-prod)}$$

$$\frac{p \downarrow \tau_1 \rhd (\phi';\Gamma') \quad \phi,\phi';\Gamma,\Gamma' \vdash e : \tau_2 \quad \phi \vdash \tau_2 : *}{\phi;\Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2}\ \textbf{(ty-match)}$$

$$\frac{\phi;\Gamma \vdash (p \Rightarrow e) : \tau_1 \Rightarrow \tau_2 \quad \phi;\Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\phi;\Gamma \vdash (p \Rightarrow e \mid ms) : \tau_1 \Rightarrow \tau_2}\ \textbf{(ty-matches)}$$

$$\frac{\phi;\Gamma \vdash e : \tau_1 \quad \phi;\Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\phi;\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ ms) : \tau_2}\ \textbf{(ty-case)}$$

$$\frac{\phi, a : \gamma;\Gamma \vdash e : \tau}{\phi;\Gamma \vdash (\lambda a : \gamma.e) : (\Pi a : \gamma.\tau)}\ \textbf{(ty-ilam)}$$

$$\frac{\phi;\Gamma \vdash e : \Pi a : \gamma.\tau \quad \phi \vdash i : \gamma}{\phi;\Gamma \vdash e[i] : \tau[a \mapsto i]}\ \textbf{(ty-iapp)}$$

$$\frac{\phi;\Gamma, x : \tau_1 \vdash e : \tau_2}{\phi;\Gamma \vdash (\mathbf{lam}\ x : \tau_1.e) : \tau_1 \to \tau_2}\ \textbf{(ty-lam)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \phi;\Gamma \vdash e_2 : \tau_1}{\phi;\Gamma \vdash e_1(e_2) : \tau_2}\ \textbf{(ty-app)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \tau_1 \quad \phi;\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau_2}\ \textbf{(ty-let)}$$

$$\frac{\phi;\Gamma, f : \tau \vdash u : \tau}{\phi;\Gamma \vdash (\mathbf{fix}\ f : \tau.u) : \tau}\ \textbf{(ty-fix)}$$

Figure 4.4: Typing Rules for $\mathrm{ML}_0^{\Pi}(C)$

$$\frac{}{\mathbf{match}(x, v) \Longrightarrow [x \mapsto v]}\ \textbf{(match-var)}$$

$$\frac{}{\mathbf{match}(\langle\rangle, \langle\rangle) \Longrightarrow []}\ \textbf{(match-unit)}$$

$$\frac{\mathbf{match}(p_1, v_1) \Longrightarrow \theta_1 \quad \mathbf{match}(p_2, v_2) \Longrightarrow \theta_2}{\mathbf{match}(\langle p_1, p_2\rangle, v) \Longrightarrow \theta_1 \cup \theta_2}\ \textbf{(match-prod)}$$

$$\frac{}{\mathbf{match}(c[a_1]\ldots[a_n], c[i_1]\ldots[i_n]) \Longrightarrow [a_1 \mapsto i_1, \ldots, a_n \mapsto i_n] \cup []}\ \textbf{(match-cons-wo)}$$

$$\frac{\mathbf{match}(p, v) \Longrightarrow \theta}{\mathbf{match}(c[a_1]\ldots[a_n](p), c[i_1]\ldots[i_n](v)) \Longrightarrow [a_1 \mapsto i_1, \ldots, a_n \mapsto i_n] \cup \theta}\ \textbf{(match-cons-w)}$$

Figure 4.5: The pattern matching rules for $\mathrm{ML}_0^{\Pi}(C)$

Next we turn to the operational semantics. Matching a pattern p against a value v yields a substitution θ, whose domain includes both index and ordinary variables, written as the judgment $\mathbf{match}(p, v) \Longrightarrow \theta$.

Given $\Gamma, \phi, \Gamma', \phi'$ and θ, a judgement of form $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ can be derived through the application of the following rules.

$$\frac{}{\phi; \Gamma \vdash [] : (\cdot; \cdot)}\ \textbf{(subst-empty)}$$

$$\frac{\phi; \Gamma \vdash \theta : (\phi'; \Gamma') \quad \phi; \Gamma \vdash e : \tau}{\phi; \Gamma \vdash \theta[x \mapsto e] : (\phi'; \Gamma', x : \tau)}\ \textbf{(subst-var)}$$

$$\frac{\phi; \Gamma \vdash \theta : (\phi'; \Gamma') \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash \theta[a \mapsto i] : (\phi', a : \gamma; \Gamma')}\ \textbf{(subst-ivar)}$$

$$\frac{\phi; \Gamma \vdash \theta : (\phi'; \Gamma') \quad \phi, \phi' \vdash P : o \quad \phi \models P[\theta]}{\phi; \Gamma \vdash \theta : (\phi', P; \Gamma')}\ \textbf{(subst-iprop)}$$

The meaning of a judgement of form $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ is given in the proposition below.

Proposition 4.1.3 *If $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ is derivable, then*

$$\mathbf{dom}(\Gamma') = \mathbf{dom}(\theta_\Gamma) \ \textit{ and } \ \mathbf{dom}(\phi') = \mathbf{dom}(\theta_\phi),$$

and $\phi \models P[\theta]$ is derivable for every index proposition P declared in ϕ'.

Proof This directly follows from a structural induction on the derivation $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$. ∎

Lemma 4.1.4 *(Substitution) If $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$ and $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ are derivable, then $\phi; \Gamma \vdash e[\theta] : \tau[\theta]$ is derivable.*

Proof This follows from a structural induction on the derivation $\mathcal{D}$ of $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$, parallel to the proof of Lemma 2.2.4. We present some cases.

$$\mathcal{D} = \frac{\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau_1 \qquad \phi, \phi' \models \tau_1 \equiv \tau_2}{\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau_2}$$ By induction hypothesis, $\phi; \Gamma \vdash e[\theta] : \tau_1[\theta]$ is derivable. Clearly, $\phi \vdash \theta_\phi : \phi'$ is also derivable, and this implies that $\phi \models \tau_1[\theta_\phi] \equiv \tau_2[\theta_\phi]$ is derivable. We then have the following.

$$\frac{\phi; \Gamma \vdash e[\theta] : \tau_1[\theta_\phi] \quad \phi \models \tau_1[\theta_\phi] \equiv \tau_2[\theta_\phi]}{\phi; \Gamma \vdash e[\theta] : \tau_2[\theta_\phi]} \textbf{ (ty-eq)}$$

By the definition of θ_ϕ, $\tau_i[\theta_\phi] = \tau_i[\theta]$ for $i = 1, 2$. This concludes the case.

$$\mathcal{D} = \frac{\phi, \phi'; \Gamma, \Gamma' \vdash e_1 : \tau_1 \qquad \phi, \phi'; \Gamma, \Gamma' \vdash e_2 : \tau_2}{\phi, \phi'; \Gamma, \Gamma' \vdash \langle e_1, e_2 \rangle : \tau_1 * \tau_2}$$ By induction hypothesis, $\phi; \Gamma \vdash e_i[\theta] : \tau_i[\theta]$ are derivable for $i = 1, 2$. This leads to the following derivation.

$$\frac{\phi; \Gamma \vdash e_1[\theta] : \tau_1[\theta] \quad \phi; \Gamma \vdash e_2[\theta] : \tau_2[\theta]}{\phi; \Gamma \vdash \langle e_1[\theta], e_2[\theta] \rangle : \tau_1[\theta] * \tau_2[\theta]} \textbf{ (ty-prod)}$$

Since $\langle e_1, e_2 \rangle[\theta] = \langle e_1[\theta], e_2[\theta] \rangle$ and $(\tau_1 * \tau_2)[\theta] = \tau_1[\theta] * \tau_2[\theta]$, we are done.

All other cases can be handled similarly. ∎

Lemma 4.1.5 *Assume that there is no $a \in \mathbf{dom}(\phi)$ which occurs in pattern p. If $\phi; \Gamma \vdash v : \tau$, $p \downarrow \tau \rhd (\phi'; \Gamma')$ and $\mathbf{match}(p, v) \Longrightarrow \theta$ are derivable, then $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ is derivable.*

Proof This follows from a structural induction on the derivation $\mathcal{D}$ of $p \downarrow \tau \rhd (\phi'; \Gamma')$, parallel to the proof of Lemma 2.2.5. Since there is no $a \in \mathbf{dom}(\phi)$ which occurs in pattern p, $\mathbf{dom}(\phi) \cap \mathbf{dom}(\theta_\phi) = \emptyset$. We present one interesting case where $v = c[a_1] \ldots [a_n](v_1)$.

$$\mathcal{D} = \frac{\mathbf{match}(p_1, v_1) \Longrightarrow \theta_1}{\mathbf{match}(c[a_1] \ldots [a_n](p_1), c[i_1] \ldots [i_n](v_1)) \Longrightarrow [a_1 \mapsto i_1, \ldots, a_n \mapsto i_n] \cup \theta_1}$$ Then the derivation of $p \downarrow \tau \rhd (\phi'; \Gamma')$ must be of the following form,

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n . (\tau_1 \to \delta(i)) \quad p_1 \downarrow \tau_1 \rhd (\phi_1'; \Gamma')}{c[a_1] \ldots [a_n](p_1) \downarrow \delta(j) \rhd (a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi_1'; \Gamma')} \textbf{ (pat-cons-w)}$$

where $\tau = \delta(j)$ and $\phi' = a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi_1'$. By induction hypothesis, $\phi; \Gamma \vdash \theta_1 : (\phi_1'; \Gamma')$ is derivable. Let us first suppose that the derivation of $\phi; \Gamma \vdash v : \tau_1$ is of the following form,

$$\frac{\begin{array}{c}\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots a_n : \gamma_n . \tau_1 \to \delta(i) \\ \phi \vdash i_1 : \gamma_1 \ \cdots \ \phi \vdash i_n : \gamma_n \quad \phi; \Gamma \vdash v_1 : \tau_1[a_1, \ldots, a_n \mapsto i_1, \ldots, i_n]\end{array}}{\phi; \Gamma \vdash c[i_1] \ldots [i_n](v_1) : \delta(i[a_1, \ldots, a_n \mapsto i_1, \ldots, i_n])} \textbf{ (ty-cons-w)}$$

where $i[a_1, \ldots, a_n \mapsto i_1, \ldots, i_n]$ is j. Clearly, we have $\phi \models i[\theta] \doteq j[\theta]$ since

$$i[\theta] = i[a_1, \ldots, a_n \mapsto i_1, \ldots, i_n] = j = j[\theta].$$

It then immediately follows that $\phi;\Gamma \vdash \theta : (\phi';\Gamma')$ is derivable. Note that $\phi;\Gamma \vdash v : \tau$ can also be derived as follows,

$$\frac{\phi;\Gamma \vdash v : \tau_1 \quad \phi \models \tau_1 \equiv \tau}{\phi;\Gamma \vdash v : \tau}\ \textbf{(ty-eq)}$$

where $\tau_1 = \delta(j_1)$ for some j_1 and $\phi;\Gamma \vdash v : \tau_1$ is derived with an application of **(ty-cons-w)**. Then j_1 is $i[a_1,\ldots,a_n \mapsto i_1,\ldots,i_n]$. We can infer $\phi \models j_1 \doteq j$ from $\phi \models \tau_1 \equiv \tau$. This implies $\phi \models i[\theta] = j_1 \doteq j = j[\theta]$, leading to a derivation of $\phi;\Gamma \vdash \theta : (\phi';\Gamma')$.

All other cases can be treated similarly. ■

Lemma 4.1.5 is crucial to proving the type preservation theorem for $\mathrm{ML}_0^{\Pi}(C)$, which is formulated as Theorem 4.1.6.

4.1.2 Dynamic Semantics

The natural semantics of $\mathrm{ML}_0^{\Pi}(C)$ is given through the rules in Figure 4.6. Note that $e \hookrightarrow_d v$ means that e reduces to a value v in this semantics.

Notice that type indices are never evaluated. This highlights the language design decision we have made: there exist no direct interactions between indices and code execution. The reasoning on type indices requires constraint satisfaction done statically during type-checking.

Theorem 4.1.6 *(Type preservation in $\mathrm{ML}_0^{\Pi}(C)$) Given e, v in $\mathrm{ML}_0^{\Pi}(C)$ such that $e \hookrightarrow_d v$ is derivable. If $\phi;\Gamma \vdash e : \tau$ is derivable, then $\phi;\Gamma \vdash v : \tau$ is derivable.*

Proof The theorem follows from a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_d v$ and the derivation of $\phi;\Gamma \vdash e : \tau$, lexicographically ordered. If the last rule in the derivation of $\phi;\Gamma \vdash e : \tau$ is

$$\frac{\phi;\Gamma \vdash e : \tau' \quad \phi \vdash \tau' \equiv \tau}{\phi;\Gamma \vdash e : \tau}\ \textbf{(ty-eq)},$$

then by induction hypothesis $\phi;\Gamma \vdash v : \tau'$ is derivable, and therefore we have the following.

$$\frac{\phi;\Gamma \vdash v : \tau' \quad \phi \models \tau' \equiv \tau}{\phi;\Gamma \vdash v : \tau}\ \textbf{(ty-eq)}$$

This allows us to assume that the last rule in the derivation of $\phi;\Gamma \vdash e : \tau$ is *not* **(ty-eq)** in the rest of the proof. We present several cases.

$$\mathcal{D} = \boxed{\frac{e_0 \hookrightarrow_d v_0 \quad \textbf{match}(v_0, p_k) \Longrightarrow \theta \text{ for some } 1 \le k \le n \quad e_k[\theta] \hookrightarrow_d v}{(\textbf{case } e_0 \textbf{ of } p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \hookrightarrow_d v}}$$

Then by Proposition 4.1.2, the last rule in the derivation of $\phi;\Gamma \vdash e : \tau$ is of the following form.

$$\frac{\phi;\Gamma \vdash e_0 : \tau_0 \quad \phi;\Gamma \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \tau_0 \Rightarrow \tau}{\phi;\Gamma \vdash (\textbf{case } e_0 \textbf{ of } (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)) : \tau}\ \textbf{(ty-case)}$$

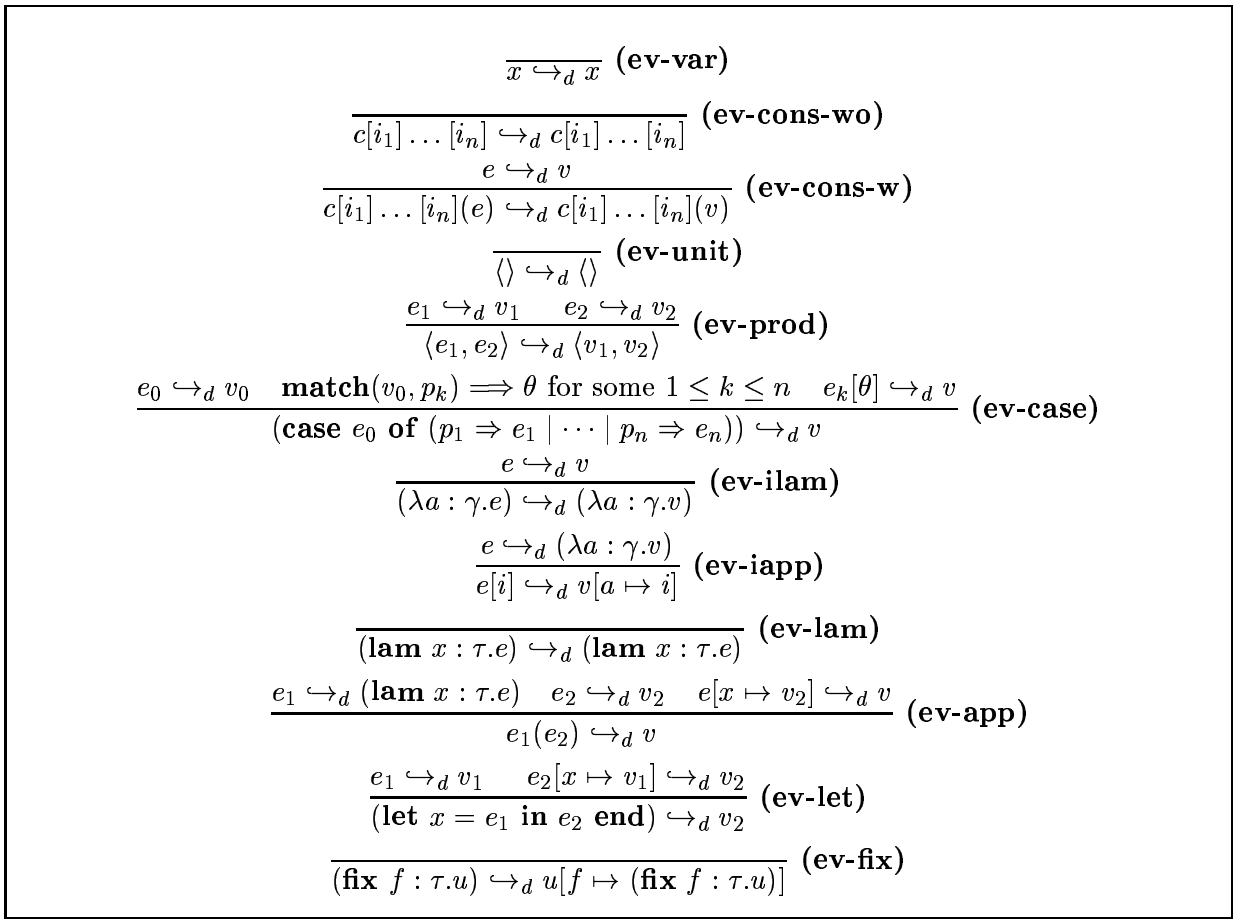

Figure 4.6: Natural Semantics for $\mathrm{ML}_0^{\Pi}(C)$

Clearly, we also have the following.

$$\frac{p_k \downarrow \tau_0 \rhd (\phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e_k : \tau \quad \phi \vdash \tau : *}{\phi; \Gamma \vdash (p_k \Rightarrow e_k) : (\tau_0 \Rightarrow \tau)} \textbf{ (ty-match)}$$

By induction hypothesis, $\phi; \Gamma \vdash v_0 : \tau_0$ is derivable. Therefore, $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ is derivable by Lemma 4.1.5. This implies that $\phi; \Gamma \vdash e_k[\theta] : \tau$ is derivable by Lemma 4.1.4 since $\tau = \tau[\theta]$. By induction hypothesis, $\phi; \Gamma \vdash v : \tau$ is derivable.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_d v_1}{(\lambda a : \gamma.e_1) \hookrightarrow_d (\lambda a : \gamma.v_1)}$ Then by Proposition 4.1.2, $\phi, a : \gamma; \Gamma \vdash e_1 : \tau_1$ is derivable, where $\Pi a : \gamma.\tau_1 = \tau$. By induction hypothesis, $\phi, a : \gamma; \Gamma \vdash v_1 : \tau_1$ is derivable, and this yields the following.

$$\frac{\phi, a : \gamma; \Gamma \vdash v_1 : \tau_1}{\phi; \Gamma \vdash (\lambda a : \gamma.v_1) : (\Pi a : \gamma.\tau_1)} \textbf{ (ty-ilam)}$$

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_d (\lambda a : \gamma.v_1)}{e_1[i] \hookrightarrow_d v_1[a \mapsto i]}$ Then by Proposition 4.1.2, we have a derivation of the following form,

$$\frac{\phi; \Gamma \vdash e_1 : (\Pi a : \gamma.\tau_1) \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash e_1[i] : \tau} \textbf{ (ty-iapp)}$$

where $\tau = \tau_1[a \mapsto i]$. By induction hypothesis, $\phi; \Gamma \vdash (\lambda a : \gamma.v_1) : (\Pi a : \gamma.\tau_1)$ is derivable, and this yields that $\phi, a : \gamma; \Gamma \vdash v_1 : \tau_1$ is derivable. By Lemma 4.1.4, $\phi; \Gamma \vdash v_1[a \mapsto i] : \tau_1[a \mapsto i]$ is derivable.

All other cases can be handled similarly. ∎

We have no intention to construct an interpreter or a compiler following the natural semantics of $\mathrm{ML}_0^{\Pi}(C)$. Instead, we intend to use existing compilers of ML to compile programs written in $\mathrm{ML}_0^{\Pi}(C)$. The following index erasure function $\|\cdot\|$ is mainly introduced for this purpose. Note that this is different from the type erasure function $|\cdot|$. Roughly speaking, the index erasure function erases everything related to type index objects, mapping $\mathrm{ML}_0^{\Pi}(C)$ programs into ML_0 ones.

Definition 4.1.7 *The index erasure function $\|\cdot\|$ is defined in Figure 4.7. which maps an expression in $\mathrm{ML}_0^{\Pi}(C)$ into one in ML_0.*

In order to justify that the index erasure function does what it is supposed to do, we have to show that the index erasure of an $\mathrm{ML}_0^{\Pi}(C)$ program behaves properly in the following sense.

1. Given an $\mathrm{ML}_0^{\Pi}(C)$ program e which evaluates to v according to the natural semantics of $\mathrm{ML}_0^{\Pi}(C)$, we must verify that $\|e\|$ evaluates to $\|v\|$ according to the natural semantics of ML_0.

2. Given an $\mathrm{ML}_0^{\Pi}(C)$ program e whose erasure $\|e\|$ evaluates to v_0 according to the natural semantics of ML_0, we must verify that e evaluates to some v according to the natural semantics of $\mathrm{ML}_0^{\Pi}(C)$ such that $\|v\| = v_0$.

$$
\begin{array}{lcl}
\|\mathbf{1}\| & = & \mathbf{1} \\
\|\delta(i)\| & = & \delta \\
\|\Pi a:\gamma.\tau\| & = & \|\tau\| \\
\|\tau_1 * \tau_2\| & = & \|\tau_1\| * \|\tau_2\| \\
\|\tau_1 \to \tau_2\| & = & \|\tau_1\| \to \|\tau_2\| \\
\|x\| & = & x \\
\|c[i_1]\ldots[i_n]\| & = & c \\
\|c[i_1]\ldots[i_n](e)\| & = & c(\|e\|) \\
\|\langle\rangle\| & = & \langle\rangle \\
\|\langle e_1, e_2\rangle\| & = & \langle\|e_1\|, \|e_2\|\rangle \\
\|p \Rightarrow e \mid ms\| & = & \|p\| \Rightarrow \|e\| \mid \|ms\| \\
\|(\mathbf{case}\ e\ \mathbf{of}\ ms)\| & = & (\mathbf{case}\ \|e\|\ \mathbf{of}\ \|ms\|) \\
\|(\mathbf{lam}\ x:\tau.e)\| & = & (\mathbf{lam}\ x:\|\tau\|.\|e\|) \\
\|e_1(e_2)\| & = & \|e_1\|(\|e_2\|) \\
\|(\lambda a:\gamma.e)\| & = & \|e\| \\
\|e[i]\| & = & \|e\| \\
\|\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}\| & = & \mathbf{let}\ x = \|e_1\|\ \mathbf{in}\ \|e_2\|\ \mathbf{end} \\
\|\mathbf{fix}\ f:\tau.u\| & = & \mathbf{fix}\ f:\|\tau\|.\|u\| \\
\|\cdot\| & = & \cdot \\
\|\Gamma, x:\tau\| & = & \|\Gamma\|, x:\|\tau\| \\
\|\cdot_{\mathcal{S}}\| & = & \cdot_{\mathcal{S}} \\
\|\mathcal{S}, \delta:\gamma\to *\| & = & \|\mathcal{S}\|, \delta:* \\
\|\mathcal{S}, c:\tau\| & = & \|\mathcal{S}\|, c:\|\tau\| \\
\|[]\| & = & [] \\
\|\theta[a\mapsto i]\| & = & \|\theta\| \\
\|\theta[x\mapsto e]\| & = & \|\theta\|[x\mapsto\|e\|]
\end{array}
$$

Figure 4.7: The definition of erasure function $\|\cdot\|$

(1) and (2) will be proven as Theorem 4.1.10 and Theorem 4.1.12, respectively.

Proposition 4.1.8 *We have the following.*

1. $\|\tau[\theta]\| = \|\tau\|$ *and* $\|e[\theta]\| = \|e\|[\|\theta\|]$.
2. $\|u\|$ *is a value form in* ML_0 *if* u *is a value form in* $\mathrm{ML}_0^{\Pi}(C)$.
3. $\|v\|$ *is a value in* ML_0 *if* v *is a value in* $\mathrm{ML}_0^{\Pi}(C)$.
4. *If* $p \downarrow \tau \rhd (\phi;\Gamma)$ *is derivable, then* $\|p\| \downarrow \|\tau\| \rhd \|\Gamma\|$ *is derivable.*
5. *If* $\mathbf{match}(p, v) \Longrightarrow \theta$ *is derivable in* $\mathrm{ML}_0^{\Pi}(C)$, *then* $\mathbf{match}(\|p\|, \|v\|) \Longrightarrow \|\theta\|$ *is derivable in* ML_0.

6. *Given* v, p *in* $\mathrm{ML}_0^{\Pi}(C)$ *such that* $\phi; \Gamma \vdash v : \tau$ *and* $p \downarrow \tau \Longrightarrow (\phi; \Gamma)$ *are derivable. If* $\mathbf{match}(\|p\|, \|v\|) \Longrightarrow \theta_0$ *is derivable, then* $\mathbf{match}(p, v) \Longrightarrow \theta$ *is derivable for some* θ *and* $\|\theta\| = \theta_0$.

7. *If* $\phi \models \tau_1 \equiv \tau_2$ *is derivable, then* $\|\tau_1\| = \|\tau_2\|$.

Proof We omit the proofs of (1), (2) and (3), which are straightforward. (4) is proven by a structural induction on the derivation $\mathcal{D}$ of $p \downarrow \tau \rhd (\phi; \Gamma)$, and we present one case below. Let $\mathcal{D}$ be a derivation of the following form.

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.(\tau \to \delta(i)) \quad p \downarrow \tau \rhd (\phi; \Gamma)}{c[a_1] \ldots [a_n](p) \downarrow \delta(j) \rhd (a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi; \Gamma)} \textbf{ (pat-cons-w)}$$

By induction hypothesis, we have the following derivation.

$$\frac{\|\mathcal{S}\|(c) = \|\tau\| \to \delta \quad \|p\| \downarrow \|\tau\| \rhd \|\Gamma\|}{c(\|p\|) \downarrow \delta \rhd \|\Gamma\|} \textbf{ (pat-cons-w)}$$

Notice that $\|c[a_1] \ldots [a_n](p)\| = c(\|p\|)$, $\|\delta(j)\| = \delta$ and

$$\|(a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi; \Gamma)\| = \|\Gamma\|.$$

Hence we are done.

(5) follows from a straightforward structural induction on the derivation $\mathcal{D}$ of $\mathbf{match}(p, v) \Longrightarrow \theta$. We present one case below.

$\mathcal{D} = \dfrac{\mathbf{match}(p, v) \Longrightarrow \theta}{\mathbf{match}(c[a_1] \ldots [a_n](p), c[i_1] \ldots [i_n](v)) \Longrightarrow [a_1 \mapsto i_1, \ldots, a_n \mapsto i_n] \cup \theta}$ By induction hypothesis, $\mathbf{match}(\|p\|, \|v\|) \Longrightarrow \|\theta\|$. This leads to the following.

$$\frac{\mathbf{match}(\|p\|, \|v\|) \Longrightarrow \|\theta\|}{\mathbf{match}(c(\|p\|), c(\|v\|)) \Longrightarrow \|\theta\|} \textbf{ (match-cons-w)}$$

Since $\|c[a_1] \ldots [a_n](p)\| = c(\|p\|)$, $\|c[i_1] \ldots [i_n](v)\| = c(\|v\|)$ and

$$\|[a_1 \mapsto i_1, \ldots, a_n \mapsto i_n] \cup \theta\| = \|\theta\|,$$

we are done.

All other cases can be treated similarly.

The proof of (6) proceeds by a structural induction on the derivation of $\mathbf{match}(\|p\|, \|v\|) \Longrightarrow \theta_0$, parallel to that of (5). (7) is then proven by a structural induction on the derivation of $\phi \models \tau_1 \equiv \tau_2$. ∎

Theorem 4.1.9 *If* $\phi; \Gamma \vdash e : \tau$ *is derivable in* $\mathrm{ML}_0^{\Pi}(C)$, *then* $\|\Gamma\| \vdash \|e\| : \|\tau\|$ *is derivable in* ML_0.

Proof This simply follows from a structural induction on the derivation of $\phi; \Gamma \vdash e : \tau$. ∎

We say that an expression e in $\mathrm{ML}_0^{\Pi}(C)$ (ML_0) is typable if $\phi; \Gamma \vdash e : \tau$ ($\Gamma \vdash e : \tau$) is derivable for some ϕ, Γ, τ in $\mathrm{ML}_0^{\Pi}(C)$ (for some Γ, τ in ML_0). Also we say that an untyped expression e in $\lambda_{\mathrm{val}}^{\mathrm{pat}}$ is typable in $\mathrm{ML}_0^{\Pi}(C)$ (ML_0) if e is the *type* erasure of some typable expression in $\mathrm{ML}_0^{\Pi}(C)$ (ML_0). In this sense, it is clear from Theorem 4.1.9 that there are no more expressions in $\lambda_{\mathrm{val}}^{\mathrm{pat}}$ which are typable in $\mathrm{ML}_0^{\Pi}(C)$ than are typable in ML_0. On the other hand, there has been a great deal of research on designing type systems so that strictly more expressions in $\lambda_{\mathrm{val}}^{\mathrm{pat}}$ are typable in these type systems than are typable in ML_0. For instance, the type system extending ML_0 with let-polymorphism allows more expressions in $\lambda_{\mathrm{val}}^{\mathrm{pat}}$ to be typable. In this respect, our work is significantly different. Roughly speaking, our objective is to assign expressions more accurate types rather than make more expressions typable.

Theorem 4.1.10 *If $e \hookrightarrow_d v$ derivable in $\mathrm{ML}_0^{\Pi}(C)$, then $\|e\| \hookrightarrow_0 \|v\|$ is derivable.*

Proof This simply follows from a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_d v$. We present a few cases as follows.

$$\mathcal{D} = \frac{e_0 \hookrightarrow_d v_0 \quad \mathbf{match}(v_0, p_k) \Longrightarrow \theta \text{ for some } 1 \leq k \leq n \quad e_k[\theta] \hookrightarrow_d v}{(\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \hookrightarrow_d v}$$

Then by induction hypothesis, $\|e_0\| \hookrightarrow_0 \|v_0\|$ is derivable. By Proposition 4.1.8 (5), $\mathbf{match}(\|p_k\|, \|v_0\|) \Longrightarrow \|\theta\|$ is derivable. By induction hypothesis, $\|e_k\|[\|\theta\|] \hookrightarrow_0 \|v\|$ is derivable since $\|e_k[\theta]\| = \|e_k\|[\|\theta\|]$ by proposition 4.1.8 (1). This leads to the following.

$$\frac{\|e_0\| \hookrightarrow_0 \|v_0\| \quad \mathbf{match}(\|v_0\|, \|p_k\|) \Longrightarrow \theta \text{ for some } 1 \leq k \leq n \quad \|e_k\|[\|\theta\|] \hookrightarrow_0 \|v\|}{(\mathbf{case}\ \|e_0\|\ \mathbf{of}\ (\|p_1\| \Rightarrow \|e_1\| \mid \cdots \mid \|p_n\| \Rightarrow \|e_n\|)) \hookrightarrow_0 \|v\|}\ \textbf{(ev-case)}$$

Note that $\|\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n\|$ is

$$\mathbf{case}\ \|e_0\|\ \mathbf{of}\ (\|p_1\| \Rightarrow \|e_1\| \mid \cdots \mid \|p_n\| \Rightarrow \|e_n\|),$$

and we are done.

$$\mathcal{D} = \frac{e_1 \hookrightarrow_d v_1}{(\lambda a : \gamma.e_1) \hookrightarrow_d (\lambda a : \gamma.v_1)}$$

Then by induction hypothesis, $\|e_1\| \hookrightarrow_0 \|v_1\|$ is derivable. Note that $\|(\lambda a : \gamma.e_1)\| = \|e_1\|$ and $\|(\lambda a : \gamma.v_1)\| = \|v_1\|$. Hence we are done.

$$\mathcal{D} = \frac{e_1 \hookrightarrow_d (\lambda a : \gamma.v_1)}{e_1[i] \hookrightarrow_d v_1[a \mapsto i]}$$

Then by induction hypothesis, $\|e_1\| \hookrightarrow_0 \|(\lambda a : \gamma.v_1)\| = \|v_1\|$ is derivable. Note that $\|e_1[i]\| = \|e_1\|$. Also $\|v_1[a \mapsto i]\| = \|v_1\|$ by Proposition 4.1.8 (1). Hence, we are done.

All the rest of the cases can be handled similarly. ∎

Theorem 4.1.10 is a reconfirmation that type indices do not interact with program evaluation. This is a soundness argument in the sense that we have proven that index erasure is sound with respect to evaluation. We now prove that index erasure is also complete with respect to evaluation, formulated as Theorem 4.1.12. The following lemma will be needed in its proof.

Lemma 4.1.11 *Given a value v_1 in $\mathrm{ML}_0^{\Pi}(C)$ such that $\phi;\cdot \vdash v_1 : \Pi a:\gamma.\tau$ is derivable, v_1 must be of form $\lambda a:\gamma.v_2$ for some value v_2.*

Proof This follows from a structural induction on the derivation $\mathcal{D}$ of $\phi;\Gamma \vdash v_1 : (\Pi a:\gamma.\tau)$.

$\mathcal{D} = \dfrac{\phi;\cdot \vdash v_1 : \tau_1 \quad \phi \models \tau_1 \equiv \Pi a:\gamma.\tau}{\phi;\cdot \vdash v_1 : \Pi a:\gamma.\tau}$ Then τ_1 must of form $\Pi a:\gamma.\tau_1'$. By induction hypothesis, v_1 has the claimed form.

$\mathcal{D} = \dfrac{\phi, a:\gamma;\cdot \vdash v : \tau}{\phi;\cdot \vdash (\lambda a:\gamma.v) : (\Pi a:\gamma.\tau)}$ Then v_1 is $\lambda a:\gamma.v$, and we are done.

Note that the last applied rule in $\mathcal{D}$ cannot be **(ty-var)**. Since v_1 is a value, no other rules can be the last applied rule in $\mathcal{D}$. This concludes the proof. ∎

Theorem 4.1.12 *Given $\phi;\cdot \vdash e : \tau$ derivable in $\mathrm{ML}_0^{\Pi}(C)$. If $e^0 = \|e\| \hookrightarrow_0 v^0$ is derivable for some v^0 in ML_0, then there exists v in $\mathrm{ML}_0^{\Pi}(C)$ such that $e \hookrightarrow_d v$ is derivable and $\|v\| = v^0$.*

Proof The theorem follows from a structural induction on the derivation of $e^0 \hookrightarrow_0 v^0$ and the derivation $\mathcal{D}$ of $\phi;\cdot \vdash e : \tau$, lexicographically ordered. If the last applied rule in the derivation of $\phi;\cdot \vdash e : \tau$ is

$$\frac{\phi;\cdot \vdash e : \tau' \quad \phi \vdash \tau' \equiv \tau}{\phi;\cdot \vdash e : \tau}\ \textbf{(ty-eq)},$$

then by induction hypothesis $e \hookrightarrow_d v$ is derivable for some v and $\|v\| = v^0$. This allows us to assume that the last applied rule in the derivation of $\phi;\cdot \vdash e : \tau$ is *not* **(ty-eq)** in the rest of the proof. We present several cases.

$\mathcal{D} = \dfrac{\phi;\cdot \vdash e_0 : \tau_0 \quad \phi;\cdot \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \tau_0 \Rightarrow \tau}{\phi;\cdot \vdash (\textbf{case}\ e_0\ \textbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)) : \tau}$ Then the derivation of $e^0 \hookrightarrow_0 v^0$ must be of the following form.

$$\frac{e_0^0 \hookrightarrow_d v_0^0 \quad \textbf{match}(v_0^0, p_k^0) \Longrightarrow \theta_0 \text{ for some } 1 \le k \le n \quad e_k^0[\theta_0] \hookrightarrow_d v^0}{(\textbf{case}\ e_0^0\ \textbf{of}\ p_1^0 \Rightarrow e_1^0 \mid \cdots \mid p_n^0 \Rightarrow e_n^0) \hookrightarrow_d v^0}\ \textbf{(ev-case)},$$

where $\|e_0\| = e_0^0$, $\|p_k\| = p_k^0$ and $\|e_k\| = e_k^0$ for all $1 \le k \le n$. Clearly, we also have

$$\frac{p_k \downarrow \tau_0 \rhd (\phi';\Gamma') \quad \phi, \phi';\cdot, \Gamma' \vdash e_k : \tau \quad \phi \vdash \tau : *}{\phi;\cdot \vdash p_k \Rightarrow e_k : \tau_0 \Rightarrow \tau}\ \textbf{(ty-match)}.$$

By induction hypothesis, $e_0 \hookrightarrow_d v_0$ is derivable for some v_0 and $\|v_0\| = v_0^0$. Hence, $\phi;\cdot \vdash v_0 : \tau_0$ is derivable by Theorem 4.1.6. By Proposition 4.1.8 (6), $\textbf{match}(p_k, v_0) \Longrightarrow \theta$ is derivable for some θ and $\|\theta\| = \theta_0$. Note $e_k^0[\theta_0] = \|e_k[\theta]\|$ by Proposition 4.1.8 (1) and $\phi;\cdot \vdash \theta : (\phi';\Gamma')$ is derivable by Lemma 4.1.5. This yields that $\phi;\cdot \vdash e_k[\theta] : \tau$ is derivable by Lemma 4.1.4. By induction hypothesis, $e_k[\theta] \hookrightarrow_0 v$ is derivable for some v and $\|v\| = v^0$.

$\mathcal{D} = \dfrac{\phi, a:\gamma; \cdot \vdash e_1 : \tau}{\phi; \cdot \vdash (\lambda a:\gamma.e_1) : (\Pi a:\gamma.\tau_1)}$ Hence we have $\|\lambda a:\gamma.e_1\| = \|e_1\| \hookrightarrow_0 v^0$ for some v^0. By induction hypothesis, $e_1 \hookrightarrow_d v_1$ for some v_1 such that $\|v_1\| = v^0$. Hence we have the following.

$$\frac{e_1 \hookrightarrow_d v_1}{\lambda a:\gamma.e_1 \hookrightarrow_d \lambda a:\gamma.v_1}\ \textbf{(ev-ilam)}$$

Note $\|\lambda a:\gamma.v_1\| = \|v_1\| = v^0$, and this concludes the case.

$\mathcal{D} = \dfrac{\phi; \cdot \vdash e_1 : \Pi a:\gamma.\tau \quad \phi \vdash i : \gamma}{\phi; \cdot \vdash e_1[i] : \tau[a \mapsto i]}$ Then we have $\|e_1[i]\| = \|e_1\| \hookrightarrow_0 v_0$ for some v_0. By induction hypothesis, $e_1 \hookrightarrow_d v_1$ for some v_1. By Theorem 4.1.6, $\phi; \cdot \vdash v_1 : \Pi a:\gamma.\tau$ is derivable. Notice that v_1 is of form $\lambda a:\gamma.v_2$ by Lemma 4.1.11. This leads to the following.

$$\frac{e_1 \hookrightarrow_d v_1}{e_1[i] \hookrightarrow_d v_2[a \mapsto i]}\ \textbf{(ev-iapp)}$$

Since $\|v_2[a \mapsto i]\| = \|v_2\| = \|v_1\| = v^0$, we are done.

All other cases can be treated similarly. ∎

Now we have completely justified the following evaluation strategy for $\mathrm{ML}_0^{\Pi}(C)$: given a well-typed expression e in $\mathrm{ML}_0^{\Pi}(C)$, we can erase all type indices in e to obtain a well-typed expression $\|e\|$ in ML_0 and then evaluate it in ML_0. By Theorem 4.1.10 and Theorem 4.1.12, this yields the expected result.

We are now at the stage to report an interesting phenomenon in $\mathrm{ML}_0^{\Pi}(C)$.

Example 4.1.13 *There is no closed expression e in $\mathrm{ML}_0^{\Pi}(C)$ of type*

$$\Pi m:nat.\Pi n:nat.\mathtt{intlist}(m+n) \to \mathtt{intlist}(m)$$

such that $\|e\|(cons(\langle\overline{0}, nil\rangle))$ evaluates to a value in ML_0.

Suppose $\|e\|(cons(\langle\overline{0}, nil\rangle))$ evaluates to v. Then by Theorem 4.1.12, there are some v_1 of type $\mathtt{intlist}(1)$ and v_2 of type $\mathtt{intlist}(0)$ such that

$$e[1][0](cons[1](\langle\overline{0}, nil\rangle)) \hookrightarrow_d v_1 \text{ and } e[0][1](cons[1](\langle\overline{0}, nil\rangle)) \hookrightarrow_d v_2$$

and $\|v_1\| = v = \|v_2\|$. This is a contradiction since v cannot be a list of length both zero and one.

However, this does not mean that we could not define a function in $\mathrm{ML}_0^{\Pi}(C)$ to be of the type $\Pi m:nat.\Pi n:nat.\mathtt{intlist}(m+n) \to \mathtt{intlist}(m)$. As a matter of fact, the following function is of this type.

$$\lambda m:nat.\lambda n:nat.\mathbf{lam}\ x:\mathtt{intlist}(m+n).\mathbf{case}\ x\ \mathbf{of}\ nil \Rightarrow nil$$

If we call the above expression e, then the reader can readily verify that $e[1][0](cons[1](\langle\overline{0}, nil\rangle))$ does not evaluates to any value.

It turns out that this kind of types can also significantly complicate the constraints generated during an elaboration process which we will develop in the next section. The main reason lies in that the existential variable elimination approach introduced in Subsection 4.2.6 does not cope well with the constraints produced when such types are checked.

Since such types seem to have little practical use, we intend to find an syntactic approach to disallowing them. This will be a future research topic.

It is a straightforward observation on the typing rules for $\mathrm{ML}_0^{\Pi}(C)$ that the following theorem holds.

Theorem 4.1.14 *$\mathrm{ML}_0^{\Pi}(C)$ is a conservative extension of ML_0, that is, given Γ, e, v, τ in ML_0, $\cdot; \Gamma \vdash e : \tau$ and $e \hookrightarrow_d v$ are derivable in $\mathrm{ML}_0^{\Pi}(C)$ if and only if $\Gamma \vdash e : \tau$ and $e \hookrightarrow_0 v$ are derivable in ML_0.*

Proof The "if" part immediately follows from an inspection of the typing and evaluation rules for ML_0, which are all allowed in $\mathrm{ML}_0^{\Pi}(C)$. We now show the "only if" part. Since e is ML_0, neither rule **(ty-ilam)** nor rule **(ty-iapp)** can be applied in the derivation of $\cdot; \Gamma \vdash e : \tau$. Therefore, this derivation can easily lead to a derivation of $\Gamma \vdash e : \tau$ in ML_0. Similarly, the derivation of $e \hookrightarrow_d v$ can readily yield a derivation of $e \hookrightarrow_0 v$. ■

The novelty of our approach to enriching the type system of ML with dependent types is precisely the introduction of a *restricted* form of dependent types, where type index objects and language expressions are separated. This, however, does not prevent us from reasoning about the values of expressions in the type system because we can introduce *singleton* types to relate the value of an expression to that of an index. For example, we refine the type `int` into infinitely many singleton types `int(i)` for $i = 0, 1, -1, 2, -2, \ldots$, each of which contains only the integer i. If we can type-check that an expression e is of type `int(i)`, then we know that the run-time value of e must equal i. This, for instance, allows us to determine at compile-time whether the value of an expression of type `int` is within certain range. Please see Section 9.2 for more details on this issue.

We emphasize that both ML_0 and $\mathrm{ML}_0^{\Pi}(C)$ in Theorem 4.1.14 are *explicitly typed internal* languages, and hence we *cannot* simply conclude that if the programmer does not index any types in his programs then these programs are valid for $\mathrm{ML}_0^{\Pi}(C)$ if they are valid for ML_0. The obvious reason is that the programmer almost always writes programs in an *external* language, which may not be fully explicitly typed. Therefore, these programs needs to be elaborated into the corresponding explicitly typed ones in an internal language.

In order to guarantee that valid programs written in an external language for ML_0 can be successfully elaborated into explicitly typed programs in $\mathrm{ML}_0^{\Pi}(C)$, we will design a two phase type-checking algorithm in Chapter 6, achieving full compatibility.

4.2 Elaboration

We have so far presented an *explicitly typed* language $\mathrm{ML}_0^{\Pi}(C)$. This presentation has a serious drawback from a programmer's point of view: *one would quickly get overwhelmed with types when programming in such a setting.* It then becomes apparent that it is necessary to provide an *external language* $\mathrm{DML}_0(C)$ together with a mapping from $\mathrm{DML}_0(C)$ to the *internal language* $\mathrm{ML}_0^{\Pi}(C)$. This mapping is called *elaboration.* Note that we also use the phrase type-checking to mean elaboration, sometimes.

4.2.1 The External Language $\mathrm{DML}_0(C)$ for $\mathrm{ML}_0^{\Pi}(C)$

The syntax for $\mathrm{DML}_0(C)$ is given as follows.

$$
\begin{array}{lllll}
\text{patterns} & p & ::= & x \mid c \mid c(p) \mid \langle\rangle \mid \langle p_1, p_2\rangle \\
\text{matches} & ms & ::= & (p \Rightarrow e) \mid (p \Rightarrow e \mid ms) \\
\text{expressions} & e & ::= & x \mid c(e) \mid \langle\rangle \mid \langle e_1, e_2\rangle \\
 & & & \mid (\textbf{case}\ e\ \textbf{of}\ ms) \mid (\textbf{lam}\ x.e) \mid (\textbf{lam}\ x : \tau.e) \mid e_1(e_2) \\
 & & & \mid (\textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}) \mid (\textbf{fix}\ f : \tau.u) \mid \lambda a : \gamma.e \mid (e : \tau)
\end{array}
$$

$(e : \tau)$ means that e is annotated with type τ. Type annotations in a program will be crucial to elaboration. Also, the need for $\lambda a : \gamma.e$ is explained in Section 8.3, which is used in a very restricted way.

Note that the syntax of $\mathrm{DML}_0(C)$ is basically the syntax of ML_0, though types here could be dependent types. This partially attests to the unobtrusiveness of our enrichment. The type erasure function $|\cdot|$ on expressions in $\mathrm{ML}_0^{\Pi}(C)$ is defined in the obvious way. Again please note that $|\cdot|$ is different from the index erasure function $\|\cdot\|$, which maps an $\mathrm{ML}_0^{\Pi}(C)$ expression into an ML_0 one.

4.2.2 Elaboration as Static Semantics

We illustrate some intuition behind the elaboration rules while presenting them. Elaboration, which incorporates type checking, is defined via two mutually recursive judgments: one to synthesize a type where this can be done in a most general way, and one to check a term against a type where synthesis is not possible. The synthesizing judgement has the form $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow e^*$ and means that e elaborates into e^* *with* type τ. The checking judgement has the form $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*$ and means that e elaborates into e^* *against* type τ. In general, we use e, p, ms for external expressions, patterns and matches, and e^*, p^*, ms^* for their internal counterparts.

The purpose of first two rules is to eliminate universal quantifiers. For instance, let us assume that $e_1(e_2)$ is in the code and a type of form $\Pi a : \gamma.\tau$ is synthesized for e_1; then we must apply the rule **(elab-pi-elim)** to remove the quantifier in the type; we continue doing so until a major type is reached, which must be of form $\tau_1 \to \tau_2$ (if the code is type-correct). Note that the actual index i is not locally determined, but becomes an existential variable for the constraint solver. The rule **(elab-pi-intro-1)** is simpler since we check against a given dependent functional type. Of course, we require that there be no free occurrences of a in $\Gamma(x)$ for all $x \in \mathbf{dom}(\Gamma)$ when **(elab-pi-intro-1)** is applied.

$$
\frac{\phi; \Gamma \vdash e \uparrow \Pi a : \gamma.\tau \Rightarrow e^* \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash e \uparrow \tau[a \mapsto i] \Rightarrow e^*[i]}\ \textbf{(elab-pi-elim)}
$$

$$
\frac{\phi, a : \gamma; \Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi; \Gamma \vdash e \downarrow \Pi a : \gamma.\tau \Rightarrow (\lambda a : \gamma.e^*)}\ \textbf{(elab-pi-intro-1)}
$$

The next rule is for lambda abstraction, which checks a **lam**-expression against a type. The rule for the fixed point operator is similar. We emphasize that we never synthesize types for either **lam** or **fix**-expressions (for which principal types do not exist in general).

$$
\frac{\phi; \Gamma, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow e^*}{\phi; \Gamma \vdash (\textbf{lam}\ x.e) \downarrow \tau_1 \to \tau_2 \Rightarrow (\textbf{lam}\ x : \tau_1.e_1^*)}\ \textbf{(elab-lam)}
$$

$$\frac{}{x \downarrow \tau \Rightarrow (x; \cdot; x : \tau)} \textbf{ (elab-pat-var)}$$

$$\frac{}{\langle\rangle \downarrow \mathbf{1} \Rightarrow (\langle\rangle; \cdot; \cdot)} \textbf{ (elab-pat-unit)}$$

$$\frac{p_1 \downarrow \tau_1 \Rightarrow (p_1^*; \phi_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \Rightarrow (p_2^*; \phi_2; \Gamma_2)}{\langle p_1, p_2\rangle \downarrow \tau_1 * \tau_2 \Rightarrow (\langle p_1^*, p_2^*\rangle; \phi_1, \phi_2; \Gamma_1, \Gamma_2)} \textbf{ (elab-pat-prod)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.\delta(i)}{c \downarrow \delta(j) \Rightarrow (c[a_1] \ldots [a_n]; a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi; \Gamma)} \textbf{ (elab-pat-cons-wo)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.\tau \to \delta(i) \quad p \downarrow \tau \Rightarrow (p^*; \phi; \Gamma)}{c(p) \downarrow \delta(j) \Rightarrow (c[a_1] \ldots [a_n](p^*); a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi; \Gamma)} \textbf{ (elab-pat-cons-w)}$$

Figure 4.8: The elaboration rules for patterns

The next rule is for function application, where the interaction between the two kinds of judgments takes place. After synthesizing a major type $\tau_1 \to \tau_2$ for e_1, we simply check e_2 against τ_1—synthesis for e_2 is unnecessary.

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \to \tau_2 \Rightarrow e_1^* \quad \phi; \Gamma \vdash e_2 \downarrow \tau_1 \Rightarrow e_2^*}{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_2 \Rightarrow e_1^*(e_2^*)} \textbf{ (elab-app-up)}$$

We maintain the invariant that the shape of types of variables in the context is always determined, modulo possible index constraints which may need to be solved. This means that with the rules above we can already check all normal forms. A term which is not in normal form will most often be a **let**-expression, but in any case will require a type annotation, as illustrated in the following one of two rules for **let**-expressions.

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow e_1^* \quad \phi; \Gamma, x : \tau_1 \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*}{\phi; \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \downarrow \tau_2 \Rightarrow \textbf{let } x = e_1^* \textbf{ in } e_2^* \textbf{ end}} \textbf{ (elab-let-down)}$$

Even if we are checking against a type, we must synthesize the type of e_1. If e_1 is a function or fixpoint, its type must be given, in practice mostly be writing **let** $x : \tau = e_1$ **in** e_2 **end** which abbreviates **let** $x = (e_1 : \tau)$ **in** e_2 **end**. The following rule allows us to take advantage of such annotations.

$$\frac{\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi; \Gamma \vdash (e : \tau) \uparrow \tau \Rightarrow e^*} \textbf{ (elab-anno-up)}$$

As a result, the only types appearing in realistic programs are due to declarations of functions and a few cases of polymorphic instantiation. The latter will be explained later in Subsection 6.2.3.

Moreover, in the presence of existential dependent types, which will be introduced in Chapter 5, a pure ML type without dependencies obtained in the first phase of type-checking is assumed if no explicit type annotation is given. This makes our extension truly conservative in the sense that pure ML programs will work exactly as before, not requiring any annotations.

Elaboration rules for patterns are particularly simple, due to the constraint nature of the types for constructors. We elaborate a pattern p against a type τ, yielding an internal pattern p^* and

index context ϕ and (ordinary) context Γ, respectively. This is written as $p \downarrow \tau \Rightarrow (p^*; \phi; \Gamma)$ in Figure 4.8. This judgment is used in the rules for pattern matching. The generated index context ϕ' are assumed into the index context ϕ while elaborating e as shown in the rule **(elab-match)** below. For constraint satisfaction, these are treated as hypotheses.

$$\frac{p \downarrow \tau_1 \Rightarrow (p^*; \phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e \downarrow \tau_2 \Rightarrow e^* \quad \phi \vdash \tau_2 : *}{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (p^* \Rightarrow e^*)} \textbf{(elab-match)}$$

For instance, if the constructor *cons* is of type $\tau = \Pi a : nat.\mathtt{int} * \mathtt{intlist}(a) \rightarrow \mathtt{intlist}(a+1)$, then we have the following.

$$\frac{\mathcal{S}(cons) = \tau \qquad \dfrac{\overline{x \downarrow \mathtt{int} \Rightarrow (x; \cdot; x : \mathtt{int})} \quad \overline{xs \downarrow \mathtt{intlist}(a) \Rightarrow (xs; \cdot; xs : \mathtt{intlist}(a))}}{\langle x, xs \rangle \downarrow \mathtt{int} * \mathtt{intlist}(a) \Rightarrow (\langle x, xs \rangle; \cdot; x : \mathtt{int}, xs : \mathtt{intlist}(a))}}{cons(\langle x, xs \rangle) \downarrow \mathtt{intlist}(n+1) \Rightarrow (cons[a](\langle x, xs \rangle); a : nat, a + 1 \doteq n + 1; x : \mathtt{int}, xs : \mathtt{intlist}(a))}$$

Lemma 4.2.1 *If $p \downarrow \tau \Rightarrow (p^*; \phi; \Gamma)$ is derivable, then $p = \|p^*\|$ and $p^* \downarrow \tau \rhd (\phi; \Gamma)$ is derivable.*

Proof The proof proceeds by a structural induction on the derivation of $p \downarrow \tau \Rightarrow (p^*; \phi; \Gamma)$. We present some cases as follows.

$\mathcal{D} = \dfrac{p_1 \downarrow \tau_1 \Rightarrow (p_1^*; \phi_1; \Gamma_1) \qquad p_2 \downarrow \tau_2 \Rightarrow (p_2^*; \phi_2; \Gamma_2)}{\langle p_1, p_2 \rangle \downarrow \tau_1 * \tau_2 \Rightarrow (\langle p_1^*, p_2^* \rangle; \phi_1, \phi_2; \Gamma_1, \Gamma_2)}$ By induction hypothesis, for $i = 1, 2$, $p_i = \|p_i^*\|$ and $p_i^* \downarrow \tau_i \rhd (\phi_i; \Gamma_i)$ are derivable. Therefore, we have $\langle p_1, p_2 \rangle = \|\langle p_1^*, p_2^* \rangle\|$, and we can derive $\langle p_1^*, p_2^* \rangle \downarrow \tau_1 * \tau_2 \rhd (\phi_1, \phi_2; \Gamma_1, \Gamma_2)$ as follows.

$$\frac{p_1^* \downarrow \tau_1 \rhd (\phi_1; \Gamma_1) \quad p_2^* \downarrow \tau_2 \rhd (\phi_2; \Gamma_2)}{\langle p_1^*, p_2^* \rangle \downarrow \tau_1 * \tau_2 \rhd (\phi_1, \phi_2; \Gamma_1, \Gamma_2)} \textbf{(elab-pat-prod)}$$

This concludes the case.

$\mathcal{D} = \dfrac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.\tau \rightarrow \delta(i) \quad p \downarrow \tau \Rightarrow (p^*; \phi; \Gamma)}{c(p) \downarrow \delta(j) \Rightarrow (c[a_1] \ldots [a_n](p^*); a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi; \Gamma)}$ By induction hypothesis, $p = \|p^*\|$ and $p^* \downarrow \tau \rhd (\phi; \Gamma)$ is derivable. Hence, $c(p) = \|c[a_1] \ldots [a_n](p^*)\|$ and the following is derivable.

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.(\tau \rightarrow \delta(i)) \quad p^* \downarrow \tau \rhd (\phi; \Gamma)}{c[a_1] \ldots [a_n](p^*) \downarrow \delta(j) \rhd (a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi; \Gamma)} \textbf{(elab-pat-cons-w)}$$

This concludes the case.

All other cases are straightforward. ∎

We now present the complete list of elaboration rules for $\mathrm{ML}_0^{\Pi}(C)$ in Figure 4.9 and Figure 4.10. The correctness of these rules are justified by Theorem 4.2.2.

There is a certain amount of nondeterminism in the formulation of these elaboration rules. For instance, there is a contention between the rules **(elab-pi-intro-1)** and **(elab-pi-intro-2)** when both of them are applicable. In this case, we always choose the former over the latter. Also

$$\frac{\phi;\Gamma \vdash e \uparrow \Pi a:\gamma.\tau \Rightarrow e^* \quad \phi \vdash i:\gamma}{\phi;\Gamma \vdash e \uparrow \tau[a \mapsto i] \Rightarrow e^*[i]}\ \textbf{(elab-pi-elim)}$$

$$\frac{\phi, a:\gamma;\Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi;\Gamma \vdash e \downarrow \Pi a:\gamma.\tau \Rightarrow (\lambda a:\gamma.e^*)}\ \textbf{(elab-pi-intro-1)}$$

$$\frac{\phi, a:\gamma;\Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi;\Gamma \vdash \lambda a:\gamma.e \downarrow \Pi a:\gamma.\tau \Rightarrow (\lambda a:\gamma.e^*)}\ \textbf{(elab-pi-intro-2)}$$

$$\frac{\Gamma(x) = \tau \quad \phi \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash x \uparrow \tau \Rightarrow x}\ \textbf{(elab-var-up)}$$

$$\frac{\phi;\Gamma \vdash x \uparrow \mu_1 \Rightarrow e^* \quad \phi \models \mu_1 \equiv \mu_2}{\phi;\Gamma \vdash x \downarrow \mu_2 \Rightarrow e^*}\ \textbf{(elab-var-down)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1:\gamma_1 \ldots \Pi a_n:\gamma_n.\delta(i) \quad \phi \vdash i_1:\gamma_1 \ \cdots \ \phi \vdash i_n:\gamma_n}{\phi;\Gamma \vdash c \uparrow \delta(i[a_1,\ldots,a_n \mapsto i_1,\ldots i_n]) \Rightarrow c[i_1]\ldots[i_n]}\ \textbf{(elab-cons-wo-up)}$$

$$\frac{\phi;\Gamma \vdash c \uparrow \mu_1 \Rightarrow e^* \quad \phi \models \mu_1 \equiv \mu_2}{\phi;\Gamma \vdash c \downarrow \mu_2 \Rightarrow e^*}\ \textbf{(elab-cons-wo-down)}$$

$$\frac{\begin{array}{c}\mathcal{S}(c) = \Pi a_1:\gamma_1 \ldots \Pi a_n:\gamma_n.\tau \to \delta(i) \\ \phi;\Gamma \vdash e \downarrow \tau[a_1,\ldots,a_n \mapsto i_1,\ldots i_n] \Rightarrow e^* \\ \phi \vdash i_1:\gamma_1 \ \cdots \ \phi \vdash i_n:\gamma_n\end{array}}{\phi;\Gamma \vdash c(e) \uparrow \delta(i[a_1,\ldots,a_n \mapsto i_1,\ldots i_n]) \Rightarrow c[i_1]\ldots[i_n](e^*)}\ \textbf{(elab-cons-w-up)}$$

$$\frac{\phi;\Gamma \vdash c(e) \uparrow \mu_1 \Rightarrow e^* \quad \phi \models \mu_1 \equiv \mu_2}{\phi;\Gamma \vdash c(e) \downarrow \mu_2 \Rightarrow e^*}\ \textbf{(elab-cons-w-down)}$$

$$\frac{}{\phi;\Gamma \vdash \langle\rangle \uparrow \mathbf{1} \Rightarrow \langle\rangle}\ \textbf{(elab-unit-up)}$$

$$\frac{}{\phi;\Gamma \vdash \langle\rangle \downarrow \mathbf{1} \Rightarrow \langle\rangle}\ \textbf{(elab-unit-down)}$$

$$\frac{\phi;\Gamma \vdash e_1 \uparrow \mu_1 \Rightarrow e_1^* \quad \phi;\Gamma \vdash e_2 \uparrow \mu_2 \Rightarrow e_2^*}{\phi;\Gamma \vdash \langle e_1, e_2\rangle \uparrow \mu_1 * \mu_2 \Rightarrow \langle e_1^*, e_2^*\rangle}\ \textbf{(elab-prod-up)}$$

$$\frac{\phi;\Gamma \vdash e_1 \downarrow \tau_1 \Rightarrow e_1^* \quad \phi;\Gamma \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*}{\phi;\Gamma \vdash \langle e_1, e_2\rangle \downarrow \tau_1 * \tau_2 \Rightarrow \langle e_1^*, e_2^*\rangle}\ \textbf{(elab-prod-down)}$$

Figure 4.9: The elaboration rules for $\mathrm{ML}_0^{\Pi}(C)$ (I)

$$\frac{p \downarrow \tau_1 \Rightarrow (p^*; \phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e \downarrow \tau_2 \Rightarrow e^* \quad \phi \vdash \tau_2 : *}{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (p^* \Rightarrow e^*)} \textbf{ (elab-match)}$$

$$\frac{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (p^* \Rightarrow e^*) \quad \phi; \Gamma \vdash ms \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow ms^*}{\phi; \Gamma \vdash (p \Rightarrow e \mid ms) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (p^* \Rightarrow e^* \mid ms^*)} \textbf{ (elab-matches)}$$

$$\frac{\phi; \Gamma \vdash e \uparrow \tau_1 \Rightarrow e^* \qquad \phi; \Gamma \vdash ms \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow ms^*}{\phi; \Gamma \vdash (\textbf{case } e \textbf{ of } ms) \downarrow \tau_2 \Rightarrow (\textbf{case } e^* \textbf{ of } ms^*)} \textbf{ (elab-case)}$$

$$\frac{\phi; \Gamma, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow e^*}{\phi; \Gamma \vdash (\textbf{lam } x.e) \downarrow \tau_1 \to \tau_2 \Rightarrow (\textbf{lam } x : \tau_1.e^*)} \textbf{ (elab-lam)}$$

$$\frac{\phi; \Gamma, x_1 : \tau_1, x : \tau \vdash e \downarrow \tau_2 \Rightarrow e^* \quad \phi; \Gamma, x_1 : \tau_1 \vdash x_1 \downarrow \tau \Rightarrow e_1^*}{\phi; \Gamma \vdash (\textbf{lam } x : \tau.e) \downarrow \tau_1 \to \tau_2 \Rightarrow (\textbf{lam } x_1 : \tau_1.\textbf{let } x = e_1^* \textbf{ in } e^* \textbf{ end})} \textbf{ (elab-lam-anno)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \to \tau_2 \Rightarrow e_1^* \quad \phi; \Gamma \vdash e_2 \downarrow \tau_1 \Rightarrow e_2^*}{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_2 \Rightarrow e_1^*(e_2^*)} \textbf{ (elab-app-up)}$$

$$\frac{\phi; \Gamma \vdash e_1(e_2) \uparrow \mu_1 \Rightarrow e^* \quad \phi \models \mu_1 \equiv \mu_2}{\phi; \Gamma \vdash e_1(e_2) \downarrow \mu_2 \Rightarrow e^*} \textbf{ (elab-app-down)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow e_1^* \quad \phi; \Gamma, x : \tau_1 \vdash e_2 \uparrow \tau_2 \Rightarrow e_2^*}{\phi; \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \uparrow \tau_2 \Rightarrow \textbf{let } x = e_1^* \textbf{ in } e_2^* \textbf{ end}} \textbf{ (elab-let-up)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow e_1^* \quad \phi; \Gamma, x : \tau_1 \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*}{\phi; \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \downarrow \tau_2 \Rightarrow \textbf{let } x = e_1^* \textbf{ in } e_2^* \textbf{ end}} \textbf{ (elab-let-down)}$$

$$\frac{\phi; \Gamma, f : \tau \vdash u \downarrow \tau \Rightarrow u^*}{\phi; \Gamma \vdash (\textbf{fix } f : \tau.u) \uparrow \tau \Rightarrow (\textbf{fix } f : \tau.u^*)} \textbf{ (elab-fix-up)}$$

$$\frac{\phi; \Gamma, f : \tau \vdash u \downarrow \tau \Rightarrow u^* \quad \phi; \Gamma, x : \tau \vdash x \downarrow \tau' \Rightarrow e^*}{\phi; \Gamma \vdash (\textbf{fix } f : \tau.u) \downarrow \tau' \Rightarrow \textbf{let } x = (\textbf{fix } f : \tau.u^*) \textbf{ in } e^* \textbf{ end}} \textbf{ (elab-fix-down)}$$

$$\frac{\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi; \Gamma \vdash (e : \tau) \uparrow \tau \Rightarrow e^*} \textbf{ (elab-anno-up)}$$

$$\frac{\phi; \Gamma \vdash (e : \tau) \uparrow \mu_1 \Rightarrow e^* \quad \phi \models \mu_1 \equiv \mu_2}{\phi; \Gamma \vdash (e : \tau) \downarrow \mu_2 \Rightarrow e^*} \textbf{ (elab-anno-down)}$$

Figure 4.10: The elaboration rules for $\mathrm{ML}_0^{\Pi}(C)$ (II)

notice the occurrences of major types, that is types which do not begin with a Π quantifier, in the elaboration rules. The use of major types is mostly a pragmatic strategy which aims for making the elaboration more flexible. After introducing the existential dependent types in the next chapter, we will introduce a coercion function in Subsection 5.2.1 to replace this strategy.

Theorem 4.2.2 *We have the following.*

1. *If* $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow e^*$ *is derivable, then* $\phi; \Gamma \vdash e^* : \tau$ *is derivable and* $|e| \cong |e^*|$.
2. *If* $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*$ *is derivable, then* $\phi; \Gamma \vdash e^* : \tau$ *is derivable and* $|e^*| \cong |e|$.

Proof (1) and (2) follow straightforwardly from a simultaneous structural induction on the derivations $\mathcal{D}$ of $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow e^*$ and $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*$. We present a few cases.

$$\mathcal{D} = \frac{\phi; \Gamma \vdash e \uparrow \Pi a : \gamma.\tau \Rightarrow e^* \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash e \uparrow \tau[a \mapsto i] \Rightarrow e^*[i]}$$

By induction hypothesis, $\phi; \Gamma \vdash e^* : (\Pi a : \gamma.\tau)$ is derivable and $|e^*| \cong |e|$. This leads to the following.

$$\frac{\phi; \Gamma \vdash e^* : (\Pi a : \gamma.\tau) \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash e^*[i] : \tau[a \mapsto i]}\ \textbf{(ty-iapp)}$$

Clearly, $|e^*[i]| = |e^*| \cong |e|$.

$$\mathcal{D} = \frac{\phi; \Gamma \vdash x \uparrow \mu_1 \Rightarrow e^* \qquad \phi \models \mu_1 \equiv \mu_2}{\phi; \Gamma \vdash x \downarrow \mu_2 \Rightarrow e^*}$$

By induction hypothesis, $\phi; \Gamma \vdash e^* : \mu_1$ is derivable and $|e^*| \cong x$. Hence we have the following.

$$\frac{\phi; \Gamma \vdash e^* : \mu_1 \quad \phi \models \mu_1 \equiv \mu_2}{\phi; \Gamma \vdash e^* : \mu_2}\ \textbf{(ty-eq)}$$

$$\mathcal{D} = \frac{\phi; \Gamma, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow e^*}{\phi; \Gamma \vdash (\textbf{lam}\ x.e) \downarrow \tau_1 \to \tau_2 \Rightarrow (\textbf{lam}\ x : \tau_1.e_1^*)}$$

By induction hypothesis, $\phi; \Gamma, x : \tau_1 \vdash e^* : \tau_2$ is derivable and $|e^*| \cong |e|$. This yields the following.

$$\frac{\phi; \Gamma, x : \tau_1 \vdash e^* : \tau_2}{\phi; \Gamma \vdash (\textbf{lam}\ x : \tau_1.e^*) : \tau_1 \to \tau_2}\ \textbf{(ty-lam)}$$

Note $|\textbf{lam}\ x : \tau_1.e^*| = \textbf{lam}\ x.|e^*| \cong \textbf{lam}\ x.|e| = |\textbf{lam}\ x.e|$. Hence, we are done.

$$\mathcal{D} = \frac{\phi; \Gamma, x_1 : \tau_1, x : \tau \vdash e \downarrow \tau_2 \Rightarrow e^* \quad \phi; \Gamma, x_1 : \tau_1 \vdash x_1 \downarrow \tau \Rightarrow e_1^*}{\phi; \Gamma \vdash (\textbf{lam}\ x : \tau.e) \downarrow \tau_1 \to \tau_2 \Rightarrow (\textbf{lam}\ x_1 : \tau_1.\textbf{let}\ x = e_1^*\ \textbf{in}\ e^*\ \textbf{end})}$$

By induction hypothesis, both $\phi; \Gamma, x_1 : \tau_1, x : \tau \vdash e^* : \tau$ and $\phi; \Gamma, x_1 : \tau_1 \vdash e_1^* : \tau$ are derivable, and $|e^*| \cong |e|$ and $|e_1^*| \cong x_1$. This leads to the following.

$$\frac{\dfrac{\phi; \Gamma, x_1 : \tau_1 \vdash e_1^* : \tau \quad \phi; \Gamma, x_1 : \tau_1, x : \tau \vdash e^* : \tau_2}{\phi; \Gamma, x_1 : \tau_1 \vdash \textbf{let}\ x = e_1^*\ \textbf{in}\ e^*\ \textbf{end} : \tau_2}\ \textbf{(ty-let)}}{\phi; \Gamma \vdash (\textbf{lam}\ x_1 : \tau_1.\textbf{let}\ x = e_1^*\ \textbf{in}\ e^*\ \textbf{end}) : \tau_1 \to \tau_2}\ \textbf{(ty-lam)}$$

Notice that we have the following.

$$\begin{array}{rcl} |\mathbf{lam}\ x_1 : \tau_1.\mathbf{let}\ x = e_1^*\ \mathbf{in}\ e^*\ \mathbf{end}| & = & \mathbf{lam}\ x_1.\mathbf{let}\ x = |e_1^*|\ \mathbf{in}\ |e^*|\ \mathbf{end} \\ & \cong & \mathbf{lam}\ x_1.\mathbf{let}\ x = x_1\ \mathbf{in}\ |e^*|\ \mathbf{end} \\ & \cong & \mathbf{lam}\ x.|e^*| \cong \mathbf{lam}\ x.|e| = |\mathbf{lam}\ x.e|. \end{array}$$

This concludes the case.

$\mathcal{D} = \dfrac{\phi; \Gamma, f : \tau \vdash u \downarrow \tau \Rightarrow u^* \qquad \phi; \Gamma, x : \tau \vdash x \downarrow \tau' \Rightarrow e^*}{\phi; \Gamma \vdash (\mathbf{fix}\ f : \tau.u) \downarrow \tau' \Rightarrow \mathbf{let}\ x = (\mathbf{fix}\ f : \tau.u^*)\ \mathbf{in}\ e^*\ \mathbf{end}}$ By induction hypothesis, $\phi; \Gamma, f : \tau \vdash u^* : \tau$ and $\phi; \Gamma, x : \tau \vdash e^* : \tau'$ are derivable. This leads to the following.

$$\dfrac{\dfrac{\phi; \Gamma, f : \tau \vdash u^* : \tau}{\phi; \Gamma \vdash (\mathbf{fix}\ f : \tau.u^*) : \tau}\ \textbf{(ty-fix)} \qquad \phi; \Gamma, x : \tau \vdash e^* : \tau'}{\phi; \Gamma \vdash \mathbf{let}\ x = (\mathbf{fix}\ f : \tau.u^*)\ \mathbf{in}\ e^*\ \mathbf{end} : \tau'}\ \textbf{(ty-let)}$$

Also by induction hypothesis, $|u^*| \cong |u|$ and $x \cong |e^*|$. This yields the following.

$$\begin{array}{rcl} |\mathbf{let}\ x = (\mathbf{fix}\ f : \tau.u^*)\ \mathbf{in}\ e^*\ \mathbf{end}| & = & \mathbf{let}\ x = (\mathbf{fix}\ f.|u^*|)\ \mathbf{in}\ |e^*|\ \mathbf{end} \\ & \cong & \mathbf{let}\ x = (\mathbf{fix}\ f.|u|)\ \mathbf{in}\ x\ \mathbf{end} \\ & \cong & (\mathbf{fix}\ f.|u|) = |(\mathbf{fix}\ f.u)| \end{array}$$

Note $\mathbf{let}\ x = (\mathbf{fix}\ f.|u|)\ \mathbf{in}\ x\ \mathbf{end} \cong (\mathbf{fix}\ f.|u|)$ follows from Corollary 2.3.13.

All other cases can be treated similarly. ∎

The description of type reconstruction as static semantics is intuitively appealing, but there is still a gap between the description and its implementation. There, elaboration rules explicitly generate constraints, and thus reduce dependent type-checking to constraint satisfaction. This is the subject of the next subsection.

4.2.3 Elaboration as Constraint Generation

Our objective is to turn the elaboration rules in Figure 4.9 and Figure 4.10 into rules which generate constraints immediately when applied. For this purpose, we extend the language for type index objects as follows.

existential variables	A		
index objects	i, j	$::=$	$\cdots \mid A$
existential contexts	ψ	$::=$	$\cdot \mid \psi, A : \gamma$
existential substitutions	θ	$::=$	$[] \mid \theta[A \mapsto i]$

Intuitively speaking, the existential variables are used to represent unknown type indices during elaboration so that we can postpone the solutions to these indices until we have enough information on them.

We now list all the constraint generation rules in Figure 4.11 and Figure 4.12. Note that we assume A is not declared in ψ when when we expand ψ to $\psi, A : \gamma$. Also we always assume that ψ_1 and ψ_2 share no common existential variables when we form the context ψ_1, ψ_2. Also notice the occurrence of a^{ψ} in the rules **(constr-pi-intro-1)** and **(constr-pi-intro-2)**. We decorate a with

ψ to prevent any existential variable declared in ψ from unifying with an index i in which there are free occurrences of a^ψ. Note $\vec{a}^\psi$ and ϕ^ψ stand for $a_1^\psi, \ldots, a_n^\psi$ and $a_1^\psi : \gamma_1, \ldots, a_n^\psi : \gamma_n$, respectively, where $\vec{a} = a_1, \ldots, a_n$ and $\phi = a_1 : \gamma_1, \ldots, a_n : \gamma_n$. If a proposition P is also declared in ϕ, then label all the free index variables in P with ψ. We define $\mathbf{label}(\phi)$ as follows.

$$\mathbf{label}(\cdot) = \emptyset \qquad\qquad \mathbf{label}(\phi, a^\psi) = \mathbf{label}(\phi) \cup \mathbf{dom}(\psi)$$

A judgement of form $\phi \rhd \theta : \psi$ can be derived through the following rules.

$$\frac{\vdash \phi[\mathbf{ictx}]}{\phi \vdash [] : \cdot} \qquad\qquad \frac{\phi_1 \vdash i : \gamma \quad A \notin \mathbf{label}(\phi_1) \quad \phi_1, \phi_2[A \mapsto i] \vdash \theta : \psi}{\phi_1, \phi_2 \vdash \theta[A \mapsto i] : A : \gamma, \psi}$$

Also we use $\exists(\psi).\Phi$ for $\exists A_1 : \gamma_1 \ldots \exists A_n : \gamma_n.\Phi$, where $\psi = A_1 : \gamma_1, \ldots, A_n : \gamma_n$.

Given an index context ϕ and an existential context ψ, we can form a mixed context $(\phi \mid \psi)$ as follows.

$$\begin{array}{lcl} (\cdot \mid \psi) & = & \psi \\ (\phi \mid \cdot) & = & \phi \\ (a^{\psi_1} : \gamma_1, \phi \mid A : \gamma, \psi) & = & \left\{ \begin{array}{ll} A : \gamma, (a^{\psi_1} : \gamma_1 \mid \phi, \psi) & \text{if } A \in \mathbf{dom}(\psi_1). \\ a^{\psi_1} : \gamma_1, (\phi \mid A : \gamma, \psi) & \text{if } A \notin \mathbf{dom}(\psi_1); \end{array} \right. \end{array}$$

Judgements of forms $(\phi \mid \psi) \vdash i : \gamma$ and $(\phi \mid \psi) \vdash \tau : *$ are derived as usual, that is, similar to judgements of forms $\phi \vdash i : \gamma$.

Proposition 4.2.3 *Assume that $\phi \rhd \theta : \psi$ is derivable.*

1. *If $(\phi \mid \psi) \vdash i : \gamma$ is derivable then so is $\phi[\theta] \vdash i[\theta] : \gamma[\theta]$.*
2. *If $(\phi \mid \psi) \vdash \tau : *$ is derivable then so is $\phi[\theta] \vdash \tau[\theta] : *$.*
3. *If $(\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]$ is derivable then so is $\phi[\theta] \vdash \Gamma[\theta][\mathbf{ctx}]$.*

Proof These immediately follows from structural induction on the derivations of $(\phi \mid \psi) \vdash i : \gamma$, $(\phi \mid \psi) \vdash \tau : *$, and $(\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]$, respectively. ∎

A judgement of form $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow [\psi]\ \Phi$ basically means that e elaborates into some expression with a *synthesized* type τ while generating the constraint Φ in which all existential variables are declared in ψ. Similarly, a judgement of form $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow [\psi]\ \Phi$ means that e elaborates into some expression against a *given* type τ while generating the constraint Φ in which all existential variables are declared in ψ. Therefore, we have finally turned type-checking into constraint satisfaction.

Given an index context ϕ and a constraint formula Φ, we define $\forall(\phi).\Phi$ as follows.

$$\forall(\cdot).\Phi = \Phi \qquad \forall(\phi, a : \gamma).\Phi = \forall(\phi).\forall a : \gamma.\Phi \qquad \forall(\phi, P).\Phi = \forall(\phi).P \supset \Phi$$

Proposition 4.2.4 *Suppose that either $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow [\psi]\ \Phi$ or $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow [\psi]\ \Phi$ is derivable. Then $(\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]$, $(\phi \mid \psi) \vdash \tau : *$ and $(\phi \mid \psi) \vdash \Phi : o$ are derivable.*

$$\frac{\phi;\Gamma \vdash e \uparrow \tau \Rightarrow[\psi]\ \Phi \quad (\phi \mid \psi) \vdash \gamma : *_s}{\phi;\Gamma \vdash e \uparrow \tau \Rightarrow[\psi, A:\gamma]\ \Phi}\ \textbf{(constr-weak)}$$

$$\frac{\phi;\Gamma \vdash e \uparrow \Pi a:\gamma.\tau \Rightarrow[\psi]\ \Phi}{\phi;\Gamma \vdash e \uparrow \tau[a \mapsto A] \Rightarrow[\psi, A:\gamma]\ \Phi}\ \textbf{(constr-pi-elim)}$$

$$\frac{\phi, a^{\psi}:\gamma;\Gamma \vdash e[a \mapsto a^{\psi}] \downarrow \tau \Rightarrow[\psi]\ \Phi \quad (\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash \lambda a:\gamma.e \downarrow \Pi a:\gamma.\tau \Rightarrow[\psi]\ \forall(a^{\psi}:\gamma).\Phi}\ \textbf{(constr-pi-intro-1)}$$

$$\frac{\phi, a^{\psi}:\gamma;\Gamma \vdash e \downarrow \tau \Rightarrow[\psi]\ \Phi \quad (\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash e \downarrow \Pi a:\gamma.\tau \Rightarrow[\psi]\ \forall(a^{\psi}:\gamma).\Phi}\ \textbf{(constr-pi-intro-2)}$$

$$\frac{\Gamma(x) = \tau \quad (\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash x \uparrow \tau \Rightarrow[\psi]\ \top}\ \textbf{(constr-var-up)}$$

$$\frac{\phi;\Gamma \vdash x \uparrow \mu_1 \Rightarrow[\psi_2, \psi_1]\ \top \quad (\phi \mid \psi_2) \vdash \mu_2 : * \quad (\phi \mid \psi_2) \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash x \downarrow \mu_2 \Rightarrow[\psi_2]\ \exists(\psi_1).\mu_1 \equiv \mu_2}\ \textbf{(constr-var-down)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1:\gamma_1 \ldots \Pi a_n:\gamma_n.\delta(i) \quad (\phi;\psi) \vdash \Gamma[\mathbf{ictx}]}{\phi;\Gamma \vdash c \uparrow \delta(i[a_1,\ldots,a_n \mapsto A_1,\ldots,A_n]) \Rightarrow[\psi, A_1:\gamma_1,\ldots,A_n:\gamma_n]\ \top}\ \textbf{(constr-cons-wo-up)}$$

$$\frac{\begin{array}{c}\phi;\Gamma \vdash c \uparrow \delta(i_1) \Rightarrow[\psi_2, \psi_1]\ \top \\ (\phi \mid \psi_2) \vdash \delta(i_2) : * \quad (\phi \mid \psi_2) \vdash \Gamma[\mathbf{ctx}]\end{array}}{\phi;\Gamma \vdash c \downarrow \delta(i_2) \Rightarrow[\psi_2]\ \exists(\psi_1).\delta(i_1) \equiv \delta(i_2)}\ \textbf{(constr-cons-wo-down)}$$

$$\frac{\begin{array}{c}\mathcal{S}(c) = \Pi a_1:\gamma_1 \ldots \Pi a_n:\gamma_n.\tau \to \delta(i) \\ \phi;\Gamma \vdash e \downarrow \tau[a_1,\ldots,a_n \mapsto A_1,\ldots,A_n] \Rightarrow[\psi, A_1:\gamma_1,\ldots,A_n:\gamma_n]\ \Phi\end{array}}{\phi;\Gamma \vdash c(e) \uparrow \delta(i[a_1,\ldots,a_n \mapsto A_1,\ldots,A_n]) \Rightarrow[\psi, A_1:\gamma_1,\ldots,A_n:\gamma_n]\ \Phi}\ \textbf{(constr-cons-w-up)}$$

$$\frac{\begin{array}{c}\phi;\Gamma \vdash c(e) \uparrow \delta(i_1) \Rightarrow[\psi_2, \psi_1]\ \Phi \\ (\phi \mid \psi_2) \vdash \delta(i_2) : * \quad (\phi \mid \psi_2) \vdash \Gamma[\mathbf{ctx}]\end{array}}{\phi;\Gamma \vdash c(e) \downarrow \delta(i_2) \Rightarrow[\psi_2]\ \exists(\psi_1).\Phi \wedge \delta(i_1) \equiv \delta(i_2)}\ \textbf{(constr-cons-w-down)}$$

$$\frac{(\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash \langle\rangle \uparrow \mathbf{1} \Rightarrow[\psi]\ \top}\ \textbf{(constr-unit-up)}$$

$$\frac{(\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash \langle\rangle \downarrow \mathbf{1} \Rightarrow[\psi]\ \top}\ \textbf{(constr-unit-down)}$$

$$\frac{\phi;\Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow[\psi]\ \Phi_1 \quad \phi;\Gamma \vdash e_2 \uparrow \tau_2 \Rightarrow[\psi]\ \Phi_2}{\phi;\Gamma \vdash \langle e_1, e_2\rangle \uparrow \tau_1 * \tau_2 \Rightarrow[\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-prod-up)}$$

$$\frac{\phi;\Gamma \vdash e_1 \downarrow \tau_1 \Rightarrow[\psi]\ \Phi_1 \quad \phi;\Gamma \vdash e_2 \downarrow \tau_2 \Rightarrow[\psi]\ \Phi_2}{\phi;\Gamma \vdash \langle e_1, e_2\rangle \downarrow \tau_1 * \tau_2 \Rightarrow[\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-prod-down)}$$

Figure 4.11: The constraint generation rules for $\mathrm{ML}_0^{\Pi}(C)$ (I)

$$\frac{p \downarrow \tau_1 \Rightarrow (p^*; \phi_1; \Gamma_1) \quad \phi, \phi_1^{\psi}; \Gamma, \Gamma_1 \vdash e \downarrow \tau_2 \Rightarrow [\psi]\ \Phi \quad (\phi \mid \psi) \vdash \tau_1 \Rightarrow \tau_2 : * \quad (\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow [\psi]\ \forall(\phi_1^{\psi}).\Phi}\ \textbf{(constr-match)}$$

$$\frac{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow [\psi]\ \Phi_1 \quad \phi; \Gamma \vdash ms \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash (p \Rightarrow e \mid ms) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-matches)}$$

$$\frac{\phi; \Gamma \vdash e \uparrow \tau_1 \Rightarrow [\psi]\ \Phi_1 \quad \phi; \Gamma \vdash ms \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ ms) \downarrow \tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-case)}$$

$$\frac{\phi; \Gamma, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow [\psi]\ \Phi}{\phi; \Gamma \vdash (\mathbf{lam}\ x.e) \downarrow \tau_1 \to \tau_2 \Rightarrow [\psi]\ \Phi}\ \textbf{(constr-lam)}$$

$$\frac{\phi; \Gamma, x : \tau \vdash e \downarrow \tau_2 \Rightarrow [\psi]\ \Phi \quad \phi; \Gamma, x : \tau_1 \vdash x \downarrow \tau \Rightarrow [\psi]\ \Phi_1}{\phi; \Gamma \vdash (\mathbf{lam}\ x : \tau.e) \downarrow \tau_1 \to \tau_2 \Rightarrow [\psi]\ \Phi \wedge \Phi_1}\ \textbf{(constr-lam-anno)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \to \tau_2 \Rightarrow [\psi]\ \Phi_1 \quad \phi; \Gamma \vdash e_2 \downarrow \tau_1 \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-app-up)}$$

$$\frac{\phi; \Gamma \vdash e_1(e_2) \uparrow \mu_1 \Rightarrow [\psi_2, \psi_1]\ \Phi \quad (\phi \mid \psi_2) \vdash \mu_2 : * \quad (\phi \mid \psi_2) \vdash \Gamma[\mathbf{ctx}]}{\phi; \Gamma \vdash e_1(e_2) \downarrow \mu_2 \Rightarrow [\psi_2]\ \exists(\psi_1).\Phi \wedge \mu_1 \equiv \mu_2}\ \textbf{(constr-app-down)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow [\psi]\ \Phi_1 \quad \phi; \Gamma, x : \tau_1 \vdash e_2 \uparrow \tau_2 \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}) \uparrow \tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-let-up)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow [\psi]\ \Phi_1 \quad \phi; \Gamma, x : \tau_1 \vdash e_2 \downarrow \tau_2 \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}) \downarrow \tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-let-down)}$$

$$\frac{\phi; \Gamma, f : \tau \vdash u \downarrow \tau \Rightarrow [\psi]\ \Phi}{\phi; \Gamma \vdash (\mathbf{fix}\ f : \tau.u) \uparrow \tau \Rightarrow [\psi]\ \Phi}\ \textbf{(constr-fix-up)}$$

$$\frac{\phi; \Gamma, f : \tau \vdash u \downarrow \tau \Rightarrow [\psi]\ \Phi \quad \phi; \Gamma, x : \tau \vdash x \downarrow \tau_1 \Rightarrow [\psi]\ \Phi_1}{\phi; \Gamma \vdash (\mathbf{fix}\ f : \tau.u) \downarrow \tau_1 \Rightarrow [\psi]\ \Phi \wedge \Phi_1}\ \textbf{(constr-fix-down)}$$

$$\frac{\phi_1, \phi_2; \Gamma \vdash e \downarrow \tau \Rightarrow [\psi]\ \Phi \quad \phi_1 \vdash \tau : *}{\phi_1, \phi_2; \Gamma \vdash (e : \tau) \uparrow \tau \Rightarrow [\psi]\ \Phi}\ \textbf{(constr-anno-up)}$$

$$\frac{\phi; \Gamma \vdash (e : \tau) \uparrow \mu_1 \Rightarrow [\psi_2, \psi_1]\ \top \quad (\phi \mid \psi_2) \vdash \mu_2 : * \quad (\phi \mid \psi_2) \vdash \Gamma[\mathbf{ctx}]}{\phi; \Gamma \vdash (e : \tau) \downarrow \mu_2 \Rightarrow [\psi_2]\ \exists(\psi_1).\mu_1 \equiv \mu_2}\ \textbf{(constr-anno-down)}$$

Figure 4.12: The constraint generation rules for $\mathrm{ML}_0^{\Pi}(C)$ (II)

Proof This simply follows from a simultaneous structural induction on the derivations of $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow [\psi]\ \Phi$ and $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow [\psi]\ \Phi$. ∎

Theorem 4.2.5 relates the constraint generation rules to the elaboration rules in Figure 4.9 and Figure 4.10, justifying the correctness of these constraint generation rules.

Theorem 4.2.5 *We have the following.*

1. *Suppose that $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow [\psi]\ \Phi$ is derivable. If $\phi[\theta] \models \Phi[\theta]$ is provable for some θ such that $\phi \rhd \theta : \psi$ is derivable, then there exists e^* such that $\phi[\theta]; \Gamma[\theta] \vdash e \uparrow \tau[\theta] \Rightarrow e^*$ is derivable.*

2. *Suppose that $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow [\psi]\ \Phi$ is derivable. If $\phi[\theta] \models \Phi[\theta]$ is provable for some θ such that $\phi \rhd \theta : \psi$ is derivable, then there exists e^* such that $\phi[\theta]; \Gamma[\theta] \vdash e \downarrow \tau[\theta] \Rightarrow e^*$ is derivable.*

Proof (1) and (2) follows from a simultaneous structural induction on the derivations $\mathcal{D}$ of $\Gamma \vdash e \uparrow \tau \Rightarrow [\psi]\ \Phi$ and $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow [\psi]\ \Phi$. We present several cases as follows.

$$\mathcal{D} = \frac{\phi, a^{\psi} : \gamma; \Gamma \vdash e \downarrow \tau \Rightarrow [\psi]\ \Phi}{\phi; \Gamma \vdash e \downarrow \Pi a : \gamma.\tau \Rightarrow [\psi]\ \forall (a^{\psi} : \gamma).\Phi}$$

Note that $(\forall (a^{\psi} : \gamma).\Phi)[\theta] = \forall (a^{\psi} : \gamma[\theta]).\Phi[\theta]$ since $\phi \rhd \theta : \psi$ is derivable. The derivation of $\phi[\theta] \models \forall (a^{\psi} : \gamma[\theta]).\Phi[\theta]$ must be of the following form.

$$\frac{\phi[\theta], a^{\psi} : \gamma[\theta] \models \Phi[\theta]}{\phi[\theta] \models \forall (a^{\psi} : \gamma[\theta]).\Phi[\theta]}$$

By induction hypothesis, $\phi[\theta], a^{\psi} : \gamma[\theta]; \Gamma[\theta] \vdash e \downarrow \tau[\theta] \Rightarrow e^*$ is derivable. This leads to the following.

$$\frac{\phi[\theta], a^{\psi} : \gamma[\theta]; \Gamma[\theta] \vdash e \downarrow \tau[\theta] \Rightarrow e^*}{\phi[\theta]; \Gamma[\theta] \vdash e \downarrow \Pi a^{\psi} : \gamma[\theta].\tau[\theta] \Rightarrow e^*}\ \textbf{(elab-pi-intro-1)}$$

Note that $\Pi (a^{\psi} : \gamma[\theta]).\tau[\theta]$ is $(\Pi (a^{\psi} : \gamma).\tau)[\theta]$, and we are done.

$$\mathcal{D} = \frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow [\psi]\ \Phi_1 \qquad \phi; \Gamma \vdash e_2 \uparrow \tau_2 \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash \langle e_1, e_2 \rangle \uparrow \tau_1 * \tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}$$

Then there exists θ such that $\phi \models (\Phi_1 \wedge \Phi_2)[\theta]$ is derivable. This implies that both $\phi \models \Phi_1[\theta]$ and $\phi \models \Phi_2[\theta]$ are derivable. By induction hypothesis, for $i = 1, 2$, $\phi[\theta]; \Gamma[\theta] \vdash e_i \uparrow \tau_i[\theta] \Rightarrow e_i^*$ are derivable for some e_i^*. This leads to the following.

$$\frac{\phi[\theta]; \Gamma[\theta] \vdash e_1 \uparrow \tau_1[\theta] \Rightarrow e_1^* \quad \phi[\theta]; \Gamma[\theta] \vdash e_2 \uparrow \tau_2[\theta] \Rightarrow e_2^*}{\phi[\theta]; \Gamma[\theta] \vdash e \uparrow \tau_1[\theta_1] * \tau_2[\theta_2] \Rightarrow \langle e_1^*, e_2^* \rangle}\ \textbf{(elab-prod-up)}$$

Note that $(\tau_1 * \tau_2)[\theta] = \tau_1[\theta] * \tau_2[\theta]$. Hence we are done.

$$\mathcal{D} = \frac{\phi; \Gamma \vdash e_0 \uparrow \tau_0 \Rightarrow [\psi]\ \Phi_1 \qquad \phi; \Gamma \vdash ms \downarrow (\tau_0 \Rightarrow \tau) \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash (\textbf{case}\ e_0\ \textbf{of}\ ms) \downarrow \tau \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}$$

Then $\phi[\theta] \models (\Phi_1 \wedge \Phi_2)[\theta_2]$ is derivable for some θ such that $\phi \rhd \theta : \psi$ holds. This implies $\phi \models \Phi_1[\theta]$ and $\phi \models \Phi_2[\theta]$ are derivable.

By induction hypothesis, $\phi[\theta]; \Gamma[\theta] \vdash e_0 \uparrow \tau_0[\theta] \Rightarrow e_0^*$ and $\phi[\theta]; \Gamma[\theta] \vdash ms \downarrow \tau_0[\theta] \Rightarrow \tau[\theta] \Rightarrow ms^*$ are derivable for some e_0^* and ms^*. This leads to the following.

$$\frac{\phi[\theta]; \Gamma[\theta] \vdash e_0 \uparrow \tau_0[\theta] \Rightarrow e^* \qquad \phi; \Gamma \vdash ms \downarrow (\tau_0[\theta] \Rightarrow \tau[\theta]) \Rightarrow ms^*}{\phi[\theta]; \Gamma[\theta] \vdash (\textbf{case}\ e_0\ \textbf{of}\ ms) \downarrow \tau[\theta] \Rightarrow (\textbf{case}\ e^*\ \textbf{of}\ ms^*)}\ \textbf{(elab-case)}$$

Hence we are done.

$\mathcal{D} = \dfrac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \to \tau_2 \Rightarrow [\psi]\ \Phi_1 \qquad \phi; \Gamma \vdash e_2 \downarrow \tau_1 \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}$ Then $\phi[\theta] \models (\Phi_1 \wedge \Phi_2)[\theta]$ is derivable for some θ such that $\phi \rhd \theta : \psi$ is derivable. Hence, $\phi[\theta] \models \Phi_1[\theta]$ is derivable, and this yields that $\phi[\theta]; \Gamma[\theta] \vdash e_1 \uparrow \tau_1[\theta] \to \tau_2[\theta] \Rightarrow e_1^*$ is derivable for some e_1^*. Also by induction hypothesis, $\phi[\theta]; \Gamma[\theta] \vdash e_2 \downarrow \tau_1[\theta] \Rightarrow e_2^*$ is derivable for some e_2^* since $\phi[\theta] \models \Phi_2[\theta]$ is derivable. This yields the following.

$$\frac{\phi[\theta]; \Gamma[\theta] \vdash e_1 \uparrow \tau_1[\theta] \to \tau_2[\theta] \Rightarrow e_1^* \quad \phi[\theta]; \Gamma[\theta] \vdash e_2 \downarrow \tau_1[\theta] \Rightarrow e_2^*}{\phi[\theta]; \Gamma[\theta] \vdash e_1(e_2) \uparrow \tau_2[\theta] \Rightarrow e_1^*(e_2^*)}\ \textbf{(elab-app-up)}$$

Hence we are done.

$\mathcal{D} = \dfrac{\phi; \Gamma \vdash e_1(e_2) \uparrow \mu_1 \Rightarrow [\psi_2, \psi_1]\ \Phi \qquad (\phi \mid \psi_2) \vdash \mu_2 : * \qquad (\phi \mid \psi_2) \vdash \Gamma[\textbf{ctx}]}{\phi; \Gamma \vdash e_1(e_2) \downarrow \mu_2 \Rightarrow [\psi_2]\ \exists(\psi_1).\Phi \wedge \mu_1 \equiv \mu_2}$ Then the following

$$\phi[\theta_2] \models (\exists(\psi_1).\Phi \wedge \mu_1 \equiv \mu_2)[\theta_2]$$

is derivable for some θ_2 such that $\phi \rhd \theta : \psi_2$ holds. Note that

$$(\exists(\psi_1).\Phi \wedge \mu_1 \equiv \mu_2)[\theta_2] = \exists(\psi_1).\Phi[\theta_2] \wedge \mu_1[\theta_2] \equiv \mu_2[\theta_2],$$

and therefore $\phi[\theta_2] \models \exists(\psi_1).\Phi[\theta_2] \wedge \mu_1[\theta_2] \equiv \mu_2[\theta_2]$ is derivable. This means that $(\phi[\theta_2] \models \Phi[\theta_2] \wedge \mu_1[\theta_2] \equiv \mu_2[\theta_2])[\theta_1]$ is derivable for some θ_1 such that $\phi[\theta_2] \vdash \theta_1 : \psi_1$ holds. This implies that $\phi \rhd \theta_2 \cup \theta_1 : \psi_2, \psi_1$ is also derivable. By induction hypothesis, $\phi[\theta]; \Gamma[\theta] \vdash e_1(e_2) \uparrow \tau_2[\theta] \Rightarrow e^*$ is derivable for $\theta = \theta_2 \cup \theta_1$. Since both $(\phi \mid \psi_2) \vdash \Gamma[\textbf{ctx}]$ and $(\phi \mid \psi_2) \vdash \tau_2 : *$ are derivable, we have $\phi[\theta] = \phi[\theta_2][\theta_1] = \phi[\theta_2]$, $\Gamma[\theta] = \Gamma[\theta_2][\theta_1] = \Gamma[\theta_2]$, and $\tau_2[\theta] = \tau_2[\theta_2][\theta_1] = \tau_2[\theta_2]$. Therefore, $\phi[\theta_2]; \Gamma[\theta_2] \vdash e_1(e_2) \uparrow \tau_2[\theta_2] \Rightarrow e^*$ is derivable.

$\mathcal{D} = \dfrac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow [\psi]\ \Phi_1 \qquad \phi; \Gamma, x : \tau_1 \vdash e_2 \downarrow \tau_2 \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}) \downarrow \tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}$ Then $\phi[\theta] \models (\Phi_1 \wedge \Phi_2)[\theta]$ is derivable for some θ such that $\phi \rhd \theta : \psi_2$ holds. Clearly, $(\Phi_1 \wedge \Phi_2)[\theta] = \Phi_1[\theta] \wedge \Phi_2[\theta]$, and therefore, both $\phi[\theta] \models \Phi_1[\theta]$ and $\phi[\theta] \vdash \Phi_2[\theta]$ are derivable. By induction hypothesis, both $\phi[\theta]; \Gamma[\theta] \vdash e_1 \uparrow \tau_1[\theta] \Rightarrow e_1^*$ and $\phi[\theta]; \Gamma[\theta], x : \tau_1[\theta] \vdash e_2 \downarrow \tau_2[\theta] \Rightarrow e_2^*$ are derivable. This leads to the following.

$$\frac{\phi[\theta]; \Gamma[\theta] \vdash e_1 \uparrow \tau_1[\theta] \Rightarrow e_1^* \quad \phi[\theta]; \Gamma[\theta], x : \tau_1[\theta] \vdash e_2 \downarrow \tau_2[\theta] \Rightarrow e_2^*}{\phi[\theta]; \Gamma[\theta] \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} \downarrow \tau_2[\theta] \Rightarrow \textbf{let}\ x = e_1^*\ \textbf{in}\ e_2^*\ \textbf{end}}\ \textbf{(elab-let-down)}$$

Hence we are done.

$$\mathcal{D} = \frac{\phi; \Gamma \vdash e \downarrow \tau \Rightarrow [\psi]\ \Phi \qquad \phi \vdash \tau : *}{\phi; \Gamma \vdash (e : \tau) \uparrow \tau \Rightarrow [\psi]\ \Phi}$$

Then $\phi[\theta] \models \Phi[\theta]$ is derivable for some θ such that $\phi \rhd \theta : \psi$ holds. By induction hypothesis, $\phi[\theta]; \Gamma[\theta] \vdash e \downarrow \tau[\theta] \Rightarrow e^*$ is derivable for some e^*. Since τ cannot contain any existential variables, $\tau[\theta] = \tau$. This leads to the following.

$$\frac{\phi[\theta]; \Gamma[\theta] \vdash e \downarrow \tau \Rightarrow e^*}{\phi[\theta]; \Gamma[\theta] \vdash (e : \tau) \uparrow \tau \Rightarrow e^*}\ \textbf{(elab-anno-up)}$$

All other cases can be handled similarly. ■

Given a closed expression e in the external language $\mathrm{DML}_0(C)$, we try to derive a judgement of form $\cdot; \cdot \vdash e \uparrow \tau \Rightarrow [\psi]\ \Phi$. This can succeed if there are enough type annotations in e. By Theorem 4.2.5, e is typable if and only if $\cdot \models \exists(\psi).\Phi$ is provable. In this way, type-checking in $\mathrm{ML}_0^{\Pi}(C)$ is reduced to constraint satisfaction.

There is still some indeterminacy in the constraint generation rules, which has to be handled in an implementation. For instance, if both of the rules **(constr-pi-intro-1)** and **(constr-pi-intro-2)** are applicable, it must be decided which one is to be applied. We will explain some of these issues in Chapter 8.

4.2.4 Some Informal Explanation on Constraint Generation Rules

We first explain why the rule **(constr-weak)** is needed. Note that in the following rule

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow [\psi]\ \Phi_1 \quad \phi; \Gamma \vdash e_2 \uparrow \tau_2 \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash \langle e_1, e_2 \rangle \uparrow \tau_1 * \tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-prod-up)}$$

the two premises must have the same existential variable declaration ψ. However, it is most likely that $\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow [\psi_1]\ \Phi_1$ and $\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow [\psi_2]\ \Phi_1$ are derived for different ϕ_1 and ϕ_2. In order to obtain the same ϕ, the rule **(constr-weak)** needs to be applied. Now the question is why we do not replace the rule **(constr-prod-up)** with the following.

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow [\psi_1]\ \Phi_1 \quad \phi; \Gamma \vdash e_2 \uparrow \tau_2 \Rightarrow [\psi_2]\ \Phi_2}{\phi; \Gamma \vdash \langle e_1, e_2 \rangle \uparrow \tau_1 * \tau_2 \Rightarrow [\psi_1, \psi_2]\ \Phi_1 \wedge \Phi_2}$$

Unfortunately, this replacement can readily invalidate Proposition 4.2.4, and thus breaks down the proof of Theorem 4.2.5. We present such an example. Suppose we try to derive the following for some Φ.

$$a : \gamma; x : \delta(A_1), y : \delta(A_2) \vdash \textbf{let}\ z = \langle x, y \rangle\ \textbf{in}\ z\ \textbf{end} \downarrow \delta(a) * \delta(a) \Rightarrow [A_1 : \gamma, A_2 : \gamma]\ \Phi$$

Hence, we need to derive the following for some τ and Φ_0.

$$a : \gamma; x : \delta(A_1), y : \delta(A_2) \vdash \langle x, y \rangle \uparrow \tau \Rightarrow [A_1 : \gamma, A_2 : \gamma]\ \Phi_0$$

However, it is impossible to find ψ_1 and ψ_2 such that $\psi_1, \psi_2 = A_1 : \gamma, A_2 : \gamma$ and both

$$a : \gamma; x : \delta(A_1), y : \delta(A_2) \vdash x \uparrow \tau_1 \Rightarrow [\psi_1]\ \Phi_1 \text{ and } a : \gamma; x : \delta(A_1), y : \delta(A_2) \vdash y \uparrow \tau_2 \Rightarrow [\psi_2]\ \Phi_2$$

are derivable for some τ_1, Φ_1 and τ_2, Φ_2, respectively. For instance, if $\psi_1 = A_1 : \gamma$, then the judgement $a : \gamma; x : \delta(A_1), y : \delta(A_2) \vdash x \uparrow \tau_1 \Rightarrow[\psi_1]\ \Phi_1$ is ill-formed since A_2 is not declared anywhere.

We now briefly mention how these constraint generation rules are implemented. We associate a function *up* with the judgements of form $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow[\psi]\ \Phi$, which, when given a triple (ϕ, Γ, e), returns a triple (τ, ψ, Φ). Similarly, we associate a function *down* with the judgements of form $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow[\psi]\ \Phi$, which returns Φ when given $(\phi, \Gamma, e, \psi, \tau)$. There are also occasions where we need a variant *up'* of *up* which returns a pair (τ, Φ) when given a quadruple (ϕ, Γ, e, ψ). For instance, when computing $down(\phi, \Gamma, \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}, \psi, \tau_2)$, we need to compute $up'(\phi, \Gamma, e_1, \psi)$ to get a pair (τ_1, Φ_1) and then compute $down(\phi, (\Gamma, x : \tau_1), e_2, \psi, \tau_2)$ to get Φ_2. The result of $down(\phi, \Gamma, \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}, \psi, \tau_2)$ is then $\Phi_1 \wedge \Phi_2$. The actual implementation simply follows the constraint generation rules, and therefore we omit the further details.

4.2.5 An Example on Elaboration

We now present a simple example in full details to illustrate how the constraint generation rules in Figure 4.11 and Figure 4.12 are applied. Unlike in ML_0, the type-checking is rather involved in $\mathrm{ML}_0^{\Pi}(C)$, and therefore we strongly recommend that the reader follow through these details carefully. This will be especially helpful if the reader intends to understand how type-checking is performed for existential dependent types, which is a highly complicated subject in the next chapter.

The following is basically the auxiliary tail-recursive function in the body of the reverse function in Figure 1.1, but we have replaced the polymorphic type `'a list` with the monomorphic type `intlist`. We will not introduce polymorphic types until Chapter 6.

```
fun rev(nil, ys) = ys
  | rev(x::xs, ys) = rev(xs, x::ys)
where rev <| {m:nat} {n:nat} intlist(m) * intlist(n) -> intlist(m+n)
```

This code corresponds to the following expression in the formal external language $\mathrm{DML}_0(C)$,

$$\textbf{fix}\ rev : (\Pi m : nat.\Pi n : nat.\texttt{intlist}(m) * \texttt{intlist}(n) \rightarrow \texttt{intlist}(m+n)).body$$

where

$$body = \textbf{lam}\ pair.\textbf{case}\ pair\ \textbf{of}\ \langle nil, ys\rangle \Longrightarrow ys \mid \langle cons(\langle x, xs\rangle), ys\rangle \Longrightarrow rev(\langle xs, cons(\langle x, ys\rangle)\rangle)$$

For the sake of simplicity, we will omit the parts of a constraint generation rule that do not generate constraints when we write out constraint generation rules in the following presentation.

Let *revCode* be the above $\mathrm{DML}_0(C)$ expression. We aim for constructing a derivation of the following judgement

$$\cdot; \cdot \vdash revCode \uparrow \tau \Rightarrow[\cdot]\ \Phi_0$$

for some τ and Φ_0. Hence, the derivation must be of the following form

$$\frac{\cdot; rev : \tau \vdash body \downarrow \tau \Rightarrow[\cdot]\ \Phi_0}{\cdot; \cdot \vdash revCode \uparrow \tau \Rightarrow[\cdot]\ \Phi_0}\ \textbf{(constr-fix-up)}$$

and

$$\tau = \Pi m : nat.\Pi n : nat.\texttt{intlist}(m) * \texttt{intlist}(n) \rightarrow \texttt{intlist}(m+n).$$

Then we should have a derivation of the following form for some Φ_1,

$$\dfrac{\dfrac{m : nat, n : nat; rev : \tau \vdash body \downarrow \mu \Rightarrow [\cdot]\ \Phi_1}{m : nat; rev : \tau \vdash body \downarrow \Pi n : nat.\mu \Rightarrow [\cdot]\ \forall n : nat.\Phi_1}\ \textbf{(constr-pi-intro-2)}}{\cdot; rev : \tau \vdash body \downarrow \tau \Rightarrow [\cdot]\ \forall m : nat.\forall n : nat.\Phi_1}\ \textbf{(constr-pi-intro-2)}$$

where $\Phi_0 = \forall m : nat.\forall n : nat.\Phi_1$ and $\mu = \texttt{intlist}(m) * \texttt{intlist}(n) \rightarrow \texttt{intlist}(m+n)$. Then we should reach a derivation of the following form,

$$\dfrac{\phi; \Gamma \vdash \textbf{case}\ pair\ \textbf{of}\ ms \downarrow \texttt{intlist}(m+n) \Rightarrow [\cdot]\ \Phi_1}{m : nat, n : nat; rev : \tau \vdash body \uparrow \mu \Rightarrow [\cdot]\ \Phi_1}\ \textbf{(constr-lam)}$$

where $\phi = m : nat, n : nat$, $\Gamma = rev : \tau, pair : \texttt{intlist}(m) * \texttt{intlist}(n)$, and ms is

$$\langle nil, ys\rangle \Longrightarrow ys \mid \langle cons(\langle x, xs\rangle), ys\rangle \Longrightarrow rev(\langle xs, cons(\langle x, ys\rangle)\rangle)$$

Then we should reach a derivation of the following form for some τ_3, Φ_2 and Φ_3 such that $\Phi_1 = \Phi_2 \wedge \Phi_3$.

$$\dfrac{\phi; \Gamma \vdash pair \uparrow \tau_3 \Rightarrow [\cdot]\ \Phi_2 \quad \phi; \Gamma \vdash ms \downarrow (\tau_3 \Rightarrow \texttt{intlist}(m+n)) \Rightarrow [\cdot]\ \Phi_3}{\phi; \Gamma \vdash \textbf{case}\ pair\ \textbf{of}\ ms \downarrow \texttt{intlist}(m+n) \Rightarrow [\cdot]\ \Phi_2 \wedge \Phi_3}\ \textbf{(constr-case)}$$

Clearly, we have the following derivation for $\tau_3 = \texttt{intlist}(m) * \texttt{intlist}(n)$ and $\Phi_2 = \top$.

$$\dfrac{\Gamma(pair) = \tau_3}{\phi; \Gamma \vdash pair \uparrow \tau_3 \Rightarrow [\cdot]\ \Phi_2}\ \textbf{(constr-var-up)}$$

Then we should reach a derivation of the following form for $\Phi_3 = \Phi_4 \wedge \Phi_5$,

$$\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2}{\phi; \Gamma \vdash ms \downarrow (\texttt{intlist}(m) * \texttt{intlist}(n) \Rightarrow \texttt{intlist}(m+n)) \Rightarrow [\cdot]\ \Phi_4 \wedge \Phi_5}\ \textbf{(constr-matches)}$$

where $\mathcal{D}_1$ is a derivation of

$$\phi; \Gamma \vdash \langle nil, ys\rangle \Longrightarrow ys \downarrow (\texttt{intlist}(m) * \texttt{intlist}(n) \Rightarrow \texttt{intlist}(m+n)) \Rightarrow [\cdot]\ \Phi_4$$

and $\mathcal{D}_2$ is a derivation of

$$\phi; \Gamma \vdash \langle cons(\langle x, xs\rangle), ys\rangle \Longrightarrow ys \downarrow (\texttt{intlist}(m) * \texttt{intlist}(n) \Rightarrow \texttt{intlist}(m+n)) \Rightarrow [\cdot]\ \Phi_5$$

Clearly, $\mathcal{D}_1$ is of the following form for some p_1, ϕ_1 and Γ_1,

$$\dfrac{\langle nil, ys\rangle \downarrow \tau_3 \rhd (p_1; \phi_1; \Gamma_1) \quad \phi, \phi_1; \Gamma, \Gamma_1 \vdash ys \downarrow \tau_4 \Rightarrow [\cdot]\ \Phi_4}{\phi; \Gamma \vdash \langle nil, ys\rangle \Longrightarrow ys \downarrow \tau_3 \Rightarrow \tau_4 \Rightarrow [\cdot]\ \Phi_4}\ \textbf{(constr-match)}$$

where $\tau_4 = \texttt{intlist}(m+n)$. Notice that we have the following derivation for $p_1 = \langle nil, ys\rangle, \phi_1 = 0 \doteq m$ and $\Gamma_1 = ys : \texttt{intlist}(n)$.

$$\frac{\dfrac{\mathcal{S}(nil) = \texttt{intlist}(0)}{nil \downarrow \texttt{intlist}(m) \rhd (nil; 0 \doteq m; \cdot)} \quad \dfrac{}{ys \downarrow \texttt{intlist}(n) \rhd (ys; \cdot; ys : \texttt{intlist}(n))}}{\langle nil, ys\rangle \downarrow \tau_3 \rhd (p_1; \phi_1; \Gamma_1)}$$

Hence we have the following derivation for $\Phi_4 = \forall(\phi_1).\texttt{intlist}(n) \equiv \texttt{intlist}(m+n)$.

$$\frac{\dfrac{\dfrac{\Gamma_1(ys) = \texttt{intlist}(n)}{\phi, \phi_1; \Gamma, \Gamma_1 \vdash ys \uparrow \texttt{intlist}(n) \Rightarrow[\cdot]\ \top}}{\phi, \phi_1; \Gamma, \Gamma_1 \vdash ys \downarrow \tau_4 \Rightarrow[\cdot]\ \texttt{intlist}(n) \equiv \texttt{intlist}(m+n)}\ \textbf{(constr-var-down)}}{\phi; \Gamma \vdash \langle nil, ys\rangle \Longrightarrow ys \downarrow \tau_3 \Rightarrow \tau_4 \Rightarrow[\cdot]\ \Phi_4}\ \textbf{(constr-match)}$$

Now let us turn our attention to $\mathcal{D}_2$. Clearly, $\mathcal{D}_2$ is of the following form for some p_2, ϕ_2 and Γ_2,

$$\frac{\langle cons(\langle x, xs\rangle), ys\rangle \downarrow \tau_3 \rhd (p_2; \phi_2; \Gamma_2) \quad \phi, \phi_2; \Gamma, \Gamma_2 \vdash rev(\langle xs, cons(\langle x, ys\rangle)\rangle) \downarrow \tau_4 \Rightarrow[\cdot]\ \Phi_5'}{\phi; \Gamma \vdash \langle cons(\langle x, xs\rangle), ys\rangle \Longrightarrow ys \downarrow (\tau_3 \Rightarrow \tau_4) \Rightarrow[\cdot]\ \Phi_5}$$

where $\Phi_5 = \forall(\phi_2).\Phi_5'$. Notice that we have the following derivation $\mathcal{D}_3$

$$\frac{\mathcal{S}(cons) = \tau_{cons} \qquad \dfrac{\dfrac{}{x \downarrow \texttt{int} \rhd (x; \cdot; x : \texttt{int})} \quad \dfrac{}{xs \downarrow \texttt{intlist}(a) \rhd (xs; \cdot; xs : \texttt{intlist}(a))}}{\langle x, xs\rangle \rhd (\langle x, xs\rangle; \cdot; x : \texttt{int}, xs : \texttt{intlist}(a))}}{cons(\langle x, xs\rangle) \downarrow \texttt{intlist}(m) \rhd (cons[a](\langle x, xs\rangle); a : nat, a+1 \doteq m; x : \texttt{int}, xs : \texttt{intlist}(a))}$$

where $\tau_{cons} = \Pi a : nat.\texttt{int} * \texttt{intlist}(a) \to \texttt{intlist}(a+1)$. This leads to the derivation below for $p_2 = \langle cons[a](\langle x, xs\rangle), ys\rangle, \phi_2 = a : nat, a+1 \doteq m$ and $\Gamma_2 = x : \texttt{int}, xs : \texttt{intlist}(a), ys : \texttt{intlist}(n)$.

$$\frac{\mathcal{D}_3 \quad \dfrac{}{ys \downarrow \texttt{intlist}(n) \rhd (ys; \cdot; ys : \texttt{intlist}(n))}}{\langle cons(\langle x, xs\rangle), ys\rangle \downarrow \tau_3 \rhd (p_2; \phi_2; \Gamma_2)}\ \textbf{(elab-pat-prod)}$$

We now have the task to construct a derivation of the following form for some τ_1, τ_2 and ψ,

$$\frac{\dfrac{\dfrac{\phi, \phi_2; \Gamma, \Gamma_2 \vdash rev \uparrow \tau_1 \to \tau_2 \Rightarrow[\psi]\ \Phi_6 \quad \phi, \phi_2; \Gamma, \Gamma_2 \vdash \langle cons(\langle x, xs\rangle), ys\rangle \downarrow \tau_2 \Rightarrow[\psi]\ \Phi_7}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash rev(\langle xs, cons(\langle x, ys\rangle)\rangle) \uparrow \tau_1 \Rightarrow[\psi]\ \Phi_6 \wedge \Phi_7}}{\vdots}}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash rev(\langle xs, cons(\langle x, ys\rangle)\rangle) \downarrow \tau_4 \Rightarrow[\cdot]\ \Phi_5'}\ \textbf{(constr-app-down)}$$

Obviously, we have the following derivation for

$$\tau_1 = \texttt{intlist}(M) * \texttt{intlist}(N) \qquad \tau_2 = \texttt{intlist}(M+N) \qquad \psi = M : nat, N : nat \qquad \Phi_6 = \top.$$

$$\frac{\dfrac{\phi, \phi_2; \Gamma, \Gamma_2 \vdash rev \uparrow \Pi m : nat.\Pi n : nat.\texttt{intlist}(m) * \texttt{intlist}(n) \to \texttt{intlist}(m+n) \Rightarrow[\cdot]\ \top}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash rev \uparrow \Pi n : nat.\texttt{intlist}(M) * \texttt{intlist}(n) \to \texttt{intlist}(M+n) \Rightarrow[M : nat]\ \top}}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash rev \uparrow \tau_1 \to \tau_2 \Rightarrow[\psi]\ \Phi_6}$$

Then we need to construct a derivation of the following form,

$$\frac{\phi, \phi_2; \Gamma, \Gamma_2 \vdash \downarrow \mathtt{intlist}(M) \Rightarrow[\psi]\ \Phi_8 \quad \phi, \phi_2; \Gamma, \Gamma_2 \vdash \mathit{cons}(\langle x, ys\rangle) \downarrow \mathtt{intlist}(N) \Rightarrow[\psi]\ \Phi_9}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash \langle xs, \mathit{cons}(\langle x, ys\rangle)\rangle \downarrow \tau_1 \Rightarrow[\psi]\ \Phi_8 \wedge \Phi_9}$$

where $\Phi_7 = \Phi_8 \wedge \Phi_9$.

Clearly, we have the following derivation for $\Phi_8 = \mathtt{intlist}(a) \equiv \mathtt{intlist}(M)$.

$$\dfrac{\dfrac{\Gamma_2(xs) = \mathtt{intlist}(n)}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash xs \uparrow \mathtt{intlist}(n) \Rightarrow[\psi]\ \top}\ \textbf{(constr-var-up)}}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash xs \downarrow \mathtt{intlist}(M) \Rightarrow[\psi]\ \Phi_8}\ \textbf{(constr-var-down)}$$

Let $\mathcal{D}_4$ be following derivation,

$$\dfrac{\dfrac{\Gamma_2(x) = \mathtt{int}}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash x \uparrow \mathtt{int} \Rightarrow[\psi, L : \mathit{nat}]\ \top}\ \textbf{(constr-var-up)}}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash x \downarrow \mathtt{int} \Rightarrow[\psi, L : \mathit{nat}]\ \top \wedge \mathtt{int} \equiv \mathtt{int}}\ \textbf{(constr-var-down)}$$

and $\mathcal{D}_5$ be the following derivation.

$$\dfrac{\dfrac{\Gamma_2(ys) = \mathtt{intlist}(n)}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash ys \uparrow \mathtt{intlist}(n) \Rightarrow[\psi, L : \mathit{nat}]\ \top}\ \textbf{(constr-var-up)}}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash ys \downarrow \mathtt{intlist}(L) \Rightarrow[\psi, L : \mathit{nat}]\ \top \wedge \mathtt{intlist}(n) \equiv \mathtt{intlist}(L)}\ \textbf{(constr-var-down)}$$

Therefore, we have the following derivation

$$\dfrac{\dfrac{\dfrac{\mathcal{S}(\mathit{cons}) = \tau_{\mathit{cons}} \quad \dfrac{\mathcal{D}_4 \quad \mathcal{D}_5}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash \langle x, ys\rangle \downarrow \mathtt{int} * \mathtt{intlist}(L) \Rightarrow[\psi, L : \mathit{nat}]\ \Phi_{10}}\ \textbf{(constr-prod-down)}}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash \mathit{cons}(\langle x, ys\rangle) \uparrow \mathtt{intlist}(L+1) \Rightarrow[\psi, L : \mathit{nat}]\ \top \wedge \Phi_{10}}\ \textbf{(constr-cons-w-up)}}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash \mathit{cons}(\langle x, ys\rangle) \downarrow \mathtt{intlist}(N) \Rightarrow[\psi]\ \Phi_9}\ \textbf{(constr-app-down)}$$

for $\Phi_9 = \exists(L : \mathit{nat}).\top \wedge \Phi_{10} \wedge \mathtt{intlist}(L+1) \equiv \mathtt{intlist}(N)$, where $\Phi_{10} = \top \wedge \mathtt{int} \equiv \mathtt{int} \wedge \top \wedge \mathtt{intlist}(n) \equiv \mathtt{intlist}(L)$. So far we have constructed a derivation of

$$\phi, \phi_2; \Gamma, \Gamma_2 \vdash \mathit{rev}(\langle xs, \mathit{cons}(\langle x, ys\rangle)\rangle) \uparrow \tau_2 \Rightarrow[\psi]\ \Phi_6 \wedge \Phi_7,$$

which then leads to the following for $\Phi_5' = \exists(\psi).\Phi_6 \wedge \Phi_7 \wedge \mathtt{intlist}(M+N) \equiv \mathtt{intlist}(m+n)$.

$$\frac{\phi, \phi_2; \Gamma, \Gamma_2 \vdash \mathit{rev}(\langle xs, \mathit{cons}(\langle x, ys\rangle)\rangle) \uparrow \tau_2 \Rightarrow[\psi]\ \Phi_6 \wedge \Phi_7}{\phi, \phi_2; \Gamma, \Gamma_2 \vdash \mathit{rev}(\langle xs, \mathit{cons}(\langle x, ys\rangle)\rangle) \downarrow \tau_4 \Rightarrow[\cdot]\ \Phi_5'}\ \textbf{(constr-app-down)}$$

We have finally finished the construction of a derivation of $\cdot; \cdot \vdash \mathit{revCode} \uparrow \tau \Rightarrow[\cdot]\ \Phi_0$ for

$$\begin{aligned}
\Phi_0 &= \forall m : \mathit{nat}.\forall n : \mathit{nat}.\Phi_1 = \forall m : \mathit{nat}.\forall n : \mathit{nat}.\Phi_2 \wedge \Phi_3 = \forall m : \mathit{nat}.\forall n : \mathit{nat}.\top \wedge \Phi_4 \wedge \Phi_5 \\
&= \forall m : \mathit{nat}.\forall n : \mathit{nat}.\top \wedge (0 \doteq m \supset \mathtt{intlist}(n) \equiv \mathtt{intlist}(m+n)) \wedge \Phi_5 \\
\Phi_5 &= \forall a : \mathit{nat}.a + 1 \doteq m \supset \Phi_5' \\
&= \forall a : \mathit{nat}.a + 1 \doteq m \supset \exists M : \mathit{nat}.\exists N : \mathit{nat}.\Phi_6 \wedge \Phi_7 \wedge \mathtt{intlist}(M+N) \equiv \mathtt{intlist}(m+n) \\
&= \forall a : \mathit{nat}.a + 1 \doteq m \supset \exists M : \mathit{nat}.\exists N : \mathit{nat}.\top \wedge \Phi_7 \wedge \mathtt{intlist}(M+N) \equiv \mathtt{intlist}(m+n) \\
\Phi_7 &= \Phi_8 \wedge \Phi_9 = \\
&= \exists(L : \mathit{nat}).\mathtt{intlist}(a) \equiv \mathtt{intlist}(M) \wedge \top \wedge \Phi_{10} \wedge \mathtt{intlist}(L+1) \equiv \mathtt{intlist}(N) \\
\Phi_{10} &= \top \wedge \mathtt{int} \equiv \mathtt{int} \wedge \top \wedge \mathtt{intlist}(n) \equiv \mathtt{intlist}(L)
\end{aligned}$$

If we replace $\delta(i) \equiv \delta(j)$ with $i \doteq j$ and remove all $\top$, then Φ_0 can be reduced to the following.

$$\begin{array}{l}\forall m : nat.\forall n : nat. \\ \quad (0 \doteq m \supset n \doteq\ m + n) \wedge \\ \quad \forall a : nat. a + 1 \doteq m \supset \wedge \\ \quad\quad \exists M : nat.\exists N : nat. \\ \quad\quad\quad (\exists (L : nat). a \doteq M \wedge n \doteq L \wedge L + 1 \doteq N) \wedge M + N \doteq m + n\end{array}$$

If we eliminate all the existential quantifiers in Φ_0 by substituting n for L, a for M and $n+1$ for N, we obtain the following constraint.

$$\begin{array}{l}\forall m : nat.\forall n : nat. \\ \quad (0 \doteq m \supset n \doteq\ m + n) \wedge \\ \quad \forall a : nat. a + 1 \doteq m \supset (a \doteq a \wedge n \doteq n \wedge n + 1 \doteq n + 1 \wedge a + (n + 1) \doteq m + n)\end{array}$$

The validity of the constraint can be readily verified. Therefore $\models \Phi$ is derivable, implying that *revCode* is well-typed.

The elimination of existential quantifiers is crucial to simplifying constraints, and therefore crucial to the practicality of our approach. We address this issue in the next subsection.

4.2.6 Elimination of Existential Variables

It is shown that all existential variables can be eliminated from the constraint generated after the example in the last subsection is elaborated. Our observation indicates that this is the case for almost of all the examples in our experiment. This suggests that we eliminate as many existential quantifiers as possible in a constraint before passing it to a constraint solver.

The rule for eliminating existential quantifiers in constraints are presented in Figure 4.13. A judgement of form $\phi \vdash i : \gamma \Rightarrow \Phi$ means that $\phi \vdash i : \gamma$ is derivable if $\phi \models \Phi$ is. This is reflected in the following proposition.

Theorem 4.2.6 *If both $\phi \vdash i : \gamma \Rightarrow \Phi$ and $\phi \models \Phi$ are derivable, then $\phi \vdash i : \gamma$ is also derivable.*

Proof This simply follows from a structural induction on the derivation $\mathcal{D}$ of $\phi \vdash i : \gamma \Rightarrow \Phi$. We present one case.

$$\mathcal{D} = \frac{\phi \vdash i : \gamma \Rightarrow \Phi_1 \qquad \phi, a : \gamma \vdash P : o \Rightarrow \Phi_2}{\phi \vdash i : \{a : \gamma \mid P\} \Rightarrow P[a \mapsto i] \wedge \Phi_1 \wedge \forall (a : \gamma).\Phi_2}$$

Since $\phi \models P[a \mapsto i] \wedge \Phi_1 \wedge \forall (a : \gamma).\Phi_2$ is derivable, $\phi \models P[a \mapsto i]$, $\phi \models \Phi_1$ and $\phi, a : \gamma \vdash \Phi_2$ are also derivable. By induction hypothesis, $\phi \vdash i : \gamma$ and $\phi, a : \gamma \vdash P : o$ is derivable. This leads to the following.

$$\frac{\phi \vdash i : \gamma \quad \phi, a : \gamma \vdash P : o \quad \phi \models P[a \mapsto i]}{\phi \vdash i : \{a : \gamma \mid P\}}\ \textbf{(index-subset)}$$

All other cases can be handled similarly. ∎

We use $\textbf{solve}(A : \gamma; \Phi) \downarrow (i; \Phi')$ to mean that solving Φ for A yields an index i and a constraint Φ'. Also $\textbf{solves}(\psi; \Phi) \downarrow (\theta; \Phi')$ means that solving Φ for the existential variables declared in ψ generates a substitution θ with domain ψ and a constraint Φ'. Finally, $\textbf{elimExt}(\Phi) \downarrow \Phi'$ means that eliminating all the existential variables in Φ yields a constraint Φ'.

Proposition 4.2.7 *We have the following.*

1. *Suppose* $\phi; \phi_1 \vdash \mathbf{solve}(A : \gamma; \Phi) \downarrow (i; \Phi')$ *is derivable. If* $\phi, \phi_1 \models \Phi'[A \mapsto i]$ *is derivable then so is* $\phi, \phi_1 \models \Phi[A \mapsto i]$.

2. *Suppose* $\phi \vdash \mathbf{solves}(\psi; \Phi) \downarrow (\theta; \Phi')$. *If* $\phi \models \Phi'$ *is derivable then so is* $\phi \models \Phi[\theta]$.

3. *Suppose* $\phi \vdash \mathbf{elimExt}(\Phi) \downarrow \Phi'$ *is derivable. If* $\phi \models \Phi'$ *is derivable then so is* $\phi \models \Phi$.

Proof (1) follows from a structural induction on the derivation of $\phi; \phi_1 \vdash \mathbf{solve}(A : \gamma; \Phi) \downarrow (i; \Phi')$, and (2) follows from a structural induction on the derivation of $\phi \vdash \mathbf{solves}(\psi; \Phi) \downarrow (\theta; \Phi')$ with the help of (1). (3) then follows from (2). ■

We have thus established the correctness of the rules for eliminating existential variables in constraints.

4.3 Summary

The language $\mathrm{ML}_0^{\Pi}(C)$, which extends the language ML_0 with universal dependent types, is formulated to parameterize over a given constraint domain C.

We call the type system of $\mathrm{ML}_0^{\Pi}(C)$ a restricted form of dependent type system for the following reason. We view both index objects and expressions in $\mathrm{ML}_0^{\Pi}(C)$ as *terms*. In this view, the type of a term can depend on the *value* of terms. For instance, the type of $reverse[n](l)$, which is $\mathtt{intlist}(n)$, depends on n. An alternative is to view index objects as types, and therefore to regard the type system of $\mathrm{ML}_0^{\Pi}(C)$ as a polymorphic type system. However, this alternative leads some serious complications. For instance, it is unclear what expressions are of type i if i is an index object. Also this view complicates the interpretation of subset sorts significantly.

The operational semantics of $\mathrm{ML}_0^{\Pi}(C)$ is presented in the style of natural semantics, in which type indices are never evaluated. This highlights our language design decision which requires the reasoning on type indices be done statically. It is then proven that $\mathrm{ML}_0^{\Pi}(C)$ enjoys the type preservation property (Theorem 4.1.6). We emphasize that one *can* always evaluate type indices if one chooses to. However, there is simply no such a need for doing this. Clearly, this must be changed if run-time type-checking becomes necessary, but we currently reject all programs which cannot pass (dependent) type-checking.

Another important aspect of $\mathrm{ML}_0^{\Pi}(C)$ is that there are no more untyped expressions which are typable in $\mathrm{ML}_0^{\Pi}(C)$ than in ML_0 (Theorem 4.1.9). This distinguishes our study from those which emphasize on enriching a type system to make more expressions typable. *Our objective is to assign expressions more accurate types rather than make more expressions typable.*

Theorem 4.2.5 constitutes a major contribution of the thesis. It yields a strong justification for the methodology which we have adopted for developing dependent type systems in practical programming. Dependent types and their usefulness in programming have been noticed for at least three decades. However, the great difficulty in designing a type-checking algorithm for dependent type system has always been a major obstacle which hinders the wide use of dependent types in programming. We briefly explain the reason as follows.

In a *fully* dependent type system such as the one which underlies LF (Harper, Honsell, and Plotkin 1993) or Coq (Coquand and Huet 1986), there is no differentiation between the type index

$$\frac{\phi \vdash a : b}{\phi \vdash a : b \Rightarrow \top}$$

$$\frac{\Sigma(f) = \gamma \to b \quad \phi \vdash i : \gamma \Rightarrow \Phi}{\phi \vdash f(i) : b \Rightarrow \Phi}$$

$$\frac{\phi \vdash i_1 : \gamma_1 \Rightarrow \Phi_1 \quad \phi \vdash i_2 : \gamma_1 \Rightarrow \Phi_2}{\phi \vdash \langle i_1, i_2 \rangle : \gamma_1 * \gamma_2 \Rightarrow \Phi_1 \wedge \Phi_2}$$

$$\frac{\phi \vdash i : \gamma \Rightarrow \Phi_1 \quad \phi, a : \gamma \vdash P : o \Rightarrow \Phi_2}{\phi \vdash i : \{a : \gamma \mid P\} \Rightarrow P[a \mapsto i] \wedge \Phi_1 \wedge \forall (a : \gamma).\Phi_2}$$

$$\frac{\phi \vdash i : \gamma \Rightarrow \Phi \quad A \notin \mathbf{label}(\phi)}{\phi; \phi_1 \vdash \mathbf{solve}(A : \gamma; A \doteq i) \downarrow (i; \Phi)}$$

$$\frac{\phi \vdash i : \gamma \Rightarrow \Phi \quad A \notin \mathbf{label}(\phi)}{\phi; \phi_1 \vdash \mathbf{solve}(A : \gamma; i \doteq A) \downarrow (i; \Phi)}$$

$$\frac{\phi; \phi_1, P \vdash \mathbf{solve}(A : \gamma; \Phi) \downarrow (i; \Phi')}{\phi; \phi_1 \vdash \mathbf{solve}(A : \gamma; P \supset \Phi) \downarrow (i; P \supset \Phi')}$$

$$\frac{\phi; \phi_1, a : \gamma_1 \vdash \mathbf{solve}(A : \gamma; \Phi) \downarrow (i; \Phi')}{\phi; \phi_1 \vdash \mathbf{solve}(A : \gamma; \forall a : \gamma_1.\Phi) \downarrow (i; \forall a : \gamma_1.\Phi')}$$

$$\frac{\phi; \phi_1 \vdash \mathbf{solve}(A : \gamma; \Phi_1) \downarrow (i; \Phi_1')}{\phi; \phi_1 \vdash \mathbf{solve}(A : \gamma; \Phi_1 \wedge \Phi_2) \downarrow (i; \Phi_1' \wedge \Phi_2)}$$

$$\frac{\phi; \phi_1 \vdash \mathbf{solve}(A : \gamma; \Phi_2) \downarrow (i; \Phi_2')}{\phi; \phi_1 \vdash \mathbf{solve}(A : \gamma; \Phi_1 \wedge \Phi_2) \downarrow (i; \Phi_1 \wedge \Phi_2')}$$

$$\frac{}{\phi \vdash \mathbf{solves}(\cdot; \Phi) \downarrow ([]; \Phi)}$$

$$\frac{\phi, A : \gamma \vdash \mathbf{solves}(\psi; \Phi) \downarrow (\theta, \Phi') \quad \phi; \cdot \vdash \mathbf{solve}(A : \gamma; \Phi') \downarrow \Phi''}{\phi \vdash \mathbf{solves}(A : \gamma, \psi; \Phi) \downarrow (\theta \circ [A \mapsto i], \Phi''[A \mapsto i])}$$

$$\frac{}{\phi \vdash \mathbf{elimExt}(P) \downarrow P}$$

$$\frac{\phi \vdash \mathbf{elimExt}(\Phi_1) \downarrow \Phi_1' \quad \phi \vdash \mathbf{elimExt}(\Phi_2) \downarrow \Phi_2'}{\phi \vdash \mathbf{elimExt}(\Phi_1 \wedge \Phi_2) \downarrow \Phi_1' \wedge \Phi_2'}$$

$$\frac{\phi \vdash \mathbf{elimExt}(\Phi) \downarrow \Phi'}{\phi \vdash \mathbf{elimExt}(P \supset \Phi) \downarrow P \supset \Phi'}$$

$$\frac{\phi, a : \gamma \vdash \mathbf{elimExt}(\Phi) \downarrow \Phi'}{\phi \vdash \mathbf{elimExt}(\forall (a : \gamma).\Phi) \downarrow \forall (a : \gamma).\Phi'}$$

$$\frac{\phi, \psi \vdash \mathbf{elimExt}(\Phi) \downarrow \Phi' \quad \phi; \cdot \vdash \mathbf{solves}(\psi; \Phi') \downarrow (\theta; \Phi'')}{\phi \vdash \mathbf{elimExt}(\exists (\psi).\Phi) \downarrow \Phi''}$$

Figure 4.13: The rules for eliminating existential variables

objects and the expressions in the system. In other words, every expression can be used as a type index object. Suppose that we extend the type system of ML_0 with such a fully dependent type system. In this setting, the constraint domain C is the same as the programming language itself, and therefore, Theorem 4.2.5 offers little benefit since constraint satisfaction is as difficult as program verification, which seems to be intractable in practical programming. This intuitive argument suggests that it may not be such an attractive idea to use fully dependent types in a programming language.

On the other hand, if we choose C to be some relatively simple constraint domain for which there are practical approaches to constraint satisfaction, then we are guaranteed by Theorem 4.2.5 that elaboration in $\mathrm{ML}_0^{\Pi}(C)$ can be made practical. For instance, the integer constraint domain presented in Chapter 3 falls into this category.

Although it is the burden of the programmer to provide sufficient type annotations in code, our experience suggests that this requirement is not overwhelming (the part of type annotations usually consist of less than 20% of the entire code). Also type annotations can be fully trusted as program documentation since they are always verified mechanically, avoiding the "code-changes-but-comments-stay-the-same" common symptom in programming. Given the effectiveness of dependent types in program error detection and compiler optimization (Chapter 9) and the moderate number of type annotations needed for type-checking a program, we feel that the practicality of our approach has gained some solid justification.

Chapter 5

Existential Dependent Types

In this chapter, we further enrich the type system of $\mathrm{ML}_0^{\Pi}(C)$ with *existential dependent types*, yielding the language $\mathrm{ML}_0^{\Pi,\Sigma}(C)$. We illustrate through examples the need for existential dependent types, and then formulate the corresponding typing rules and elaboration algorithm. This is similar to the development presented in the last chapter, although it is significantly more involved.

5.1 Existential Dependent Types

The need for existential dependent types is immediate. The following example clearly illustrates one aspect of this point.

```
fun filter pred nil = nil
  | filter pred (x::xs) = if pred(x) then x::filter(xs) else filter(xs)
```

The function *filter* eliminates all elements in a list which do not satisfy a given predicate. Given a predicate p and a list l, we cannot calculate the length of $filter(p)(l)$ in general if we only know the types of p and l. Therefore, it is impossible to assign *filter* a dependent type of form $\Pi n : nat.\mathtt{intlist}(n) \to \mathtt{intlist}(i)$ for any index i. Intuitively, we should be able to assign *filter* the type

$$\Pi m : nat.\mathtt{intlist}(m) \to \Sigma n : nat.\mathtt{intlist}(n),$$

where $\Sigma n : nat.\mathtt{intlist}(n)$ roughly means an integer list with some *unknown* length.

Another main reason for introducing existential dependent types is to cope with existing (library) code. For instance, let *lib* be a function in a library with a (non-dependent) type `intlist` $\to$ `intlist`. In general, we cannot refine the type of *lib* without the access to the source code of *lib*. Again intuitively, we should be able to assign the function *lib* the following type

$$(\Sigma n : nat.\mathtt{intlist}(n)) \to (\Sigma n : nat.\mathtt{intlist}(n))$$

in order to check the code in which *lib* is called (if `intlist` has been refined). This provides a smooth interaction between dependent and non-dependent types.

Also existential dependent types can facilitate array bound check elimination. For example, in some implementation of Knuth-Morris-Pratt string search algorithm, one computes an integer array A whose elements are used later to index another array B. If we could assign array A the

type $(\Sigma n : nat.int(n))\ array$, i.e., an array of natural numbers, then we would only have to check whether an element i in array A is less than the size of array B when we use it to index array B. It is unnecessary to check whether i is nonnegative since the type of i, $\Sigma n : nat.int(n)$, already implies this. We refer the reader to the code in Section A.1 for more details.

Our experience indicates that existential dependent types are indispensable in practice. For instance, almost all the examples in Appendix A use some existential dependent types.

We now enrich the language $\mathrm{ML}_0^{\Pi}(C)$ with existential dependent types, and call the enriched language $\mathrm{ML}_0^{\Pi,\Sigma}(C)$. In addition to the syntax of $\mathrm{ML}_0^{\Pi}(C)$, we need the following.

$$\begin{array}{lrcl} \text{types} & \tau & ::= & \ldots \mid (\Sigma a : \gamma.\tau) \\ \text{expressions} & e & ::= & \ldots \mid \langle i \mid e \rangle \mid \textbf{let } \langle a \mid x \rangle = e_1 \textbf{ in } e_2 \textbf{ end} \\ \text{value forms} & u & ::= & \ldots \mid \langle i \mid u \rangle \\ \text{values} & v & ::= & \ldots \mid \langle i \mid v \rangle \end{array}$$

The formation of an existential dependent type is given as follows.

$$\frac{\phi, a : \gamma \vdash \tau : *}{\phi \vdash (\Sigma a : \gamma.\tau) : *}\ \textbf{(type-sig)}$$

Also the following rule is needed for extending the type congruence relation to including existential dependent types.

$$\frac{\phi, a : \gamma \models \tau \equiv \tau'}{\phi \models \Sigma a : \gamma.\tau \equiv \Sigma a : \gamma.\tau'}$$

The typing rules for existential dependent types are given below. Note that **(ty-sig-elim)** can be applied only if a has no free occurrence in Γ and τ_2.

$$\frac{\phi; \Gamma \vdash e : \tau[a \mapsto i] \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash \langle i \mid e \rangle : (\Sigma a : \gamma.\tau)}\ \textbf{(ty-sig-intro)}$$

$$\frac{\phi; \Gamma \vdash e_1 : (\Sigma a : \gamma.\tau_1) \quad \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \Gamma \vdash \textbf{let } \langle a \mid x \rangle = e_1 \textbf{ in } e_2 \textbf{ end} : \tau_2}\ \textbf{(ty-sig-elim)}$$

In addition to the evaluation rules in Figure 4.6, we need the following rules to formulate the natural semantics of $\mathrm{ML}_0^{\Pi,\Sigma}(C)$.

$$\frac{e \hookrightarrow_d v}{\langle i \mid e \rangle \hookrightarrow_d \langle i \mid v \rangle}\ \textbf{(ev-sig-intro)}$$

$$\frac{e_1 \hookrightarrow_d \langle i \mid v_1 \rangle \quad e_2[a \mapsto i][x \mapsto v_1] \hookrightarrow_d v_2}{\textbf{let } \langle a \mid x \rangle = e_1 \textbf{ in } e_2 \textbf{ end} \hookrightarrow_d v_2}\ \textbf{(ev-sig-elim)}$$

Now let us prove some expected properties of $\mathrm{ML}_0^{\Pi,\Sigma}(C)$. This part of the development of $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ is parallel to that of $\mathrm{ML}_0^{\Pi}(C)$.

Theorem 5.1.1 *(Type preservation in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$) Given e, v in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ such that $e \hookrightarrow_d v$ is derivable. If $\phi; \Gamma \vdash e : \tau$ is derivable, then $\phi; \Gamma \vdash v : \tau$ is derivable.*

Proof The theorem follows from a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_d v$ and the derivation of $\phi; \Gamma \vdash e : \tau$, lexicographically ordered. This is similar to the proof of Theorem 4.1.6 We present several cases.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_d v_1}{\langle i \mid e_1 \rangle \hookrightarrow_d \langle i \mid v_1 \rangle}$ The last applied rule in the derivation $\phi; \Gamma \vdash e : \tau$ is of the following form.

$$\frac{\phi; \Gamma \vdash e_1 : \tau_1[a \mapsto i] \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash \langle i \mid e_1 \rangle : \Sigma a : \gamma.\tau_1} \textbf{ (ty-sig-intro)}$$

By induction hypothesis, $\phi; \Gamma \vdash v_1 : \tau_1[a \mapsto i]$ is derivable, and this leads to the following.

$$\frac{\phi; \Gamma \vdash v_1 : \tau_1[a \mapsto i] \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash \langle i \mid v_1 \rangle : \Sigma a : \gamma.\tau_1} \textbf{ (ty-sig-intro)}$$

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_d \langle i \mid v_1 \rangle \qquad e_2[a \mapsto i][x \mapsto v_1] \hookrightarrow_d v_2}{\textbf{let } \langle a \mid x \rangle = e_1 \textbf{ in } e_2 \textbf{ end} \hookrightarrow_d v_2}$ The last rule in the derivation of $\phi; \Gamma \vdash e : \tau$ is of form:

$$\frac{\phi; \Gamma \vdash e_1 : \Sigma a : \gamma.\tau_1 \qquad \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \Gamma \vdash \textbf{let } \langle a \mid x \rangle = e_1 \textbf{ in } e_2 \textbf{ end} : \tau} \textbf{ (ty-sig-elim)}.$$

By induction hypothesis, $\phi; \Gamma \vdash \langle i \mid v_1 \rangle : \Sigma a : \gamma.\tau_1$ is derivable. This implies that $\phi \vdash i : \gamma$ and $\phi; \Gamma \vdash v_1 : \tau_1[a \mapsto i]$ are derivable. Since $\phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2$ is derivable and a has no free occurrences in Γ and τ_2, a proof of $\phi; \Gamma \vdash e_2[a \mapsto i][x \mapsto v_1] : \tau_2$ can also be constructed. By induction hypothesis, $\phi; \Gamma \vdash v_2 : \tau_2$ is derivable.

The other cases can be treated similarly. ■

We extend the definition of the index erasure function $\|\cdot\|$ as follows.

$$\begin{array}{lcl} \|\langle i \mid e \rangle\| & = & \|e\| \\ \|\textbf{let } \langle a \mid x \rangle = e_1 \textbf{ in } e_2 \textbf{ end}\| & = & \textbf{let } x = \|e_1\| \textbf{ in } \|e_2\| \textbf{ end} \end{array}$$

Then Theorem 4.1.9, Theorem 4.1.10 and Theorem 4.1.12 all have their corresponding versions in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$, which we mention briefly as follows.

Theorem 5.1.2 *If $\phi; \Gamma \vdash e : \tau$ is derivable in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$, then $\|\Gamma\| \vdash \|e\| : \|\tau\|$ is derivable in ML_0.*

Proof This simply follows from a structural induction on the derivation $\mathcal{D}$ of $\phi; \Gamma \vdash e : \tau$. We present some cases.

$\mathcal{D} = \dfrac{\phi; \Gamma \vdash e_1 : \tau_1[a \mapsto i] \qquad \phi \vdash i : \gamma}{\phi; \Gamma \vdash \langle i \mid e_1 \rangle : \Sigma a : \gamma.\tau_1}$ By induction hypothesis, $\|\Gamma\| \vdash \|e_1\| : \|\tau_1[a \mapsto i]\|$ is derivable. Since $\|\tau_1[a \mapsto i]\| = \|\tau_1\| = \|\Sigma a : \gamma.\tau_1\|$ and $\|\langle i \mid e_1 \rangle\| = \|e_1\|$, we are done.

$\mathcal{D} = \dfrac{\phi; \Gamma \vdash e_1 : (\Sigma a : \gamma.\tau_1) \qquad \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \Gamma \vdash \textbf{let } \langle a \mid x \rangle = e_1 \textbf{ in } e_2 \textbf{ end} : \tau_2}$ By induction hypothesis, $\|\Gamma\| \vdash \|e_1\| : \|\Sigma a : \gamma.\tau_1\|$ and $\|\Gamma, x : \tau_1\| \vdash \|e_2\| : \|\tau_2\|$ are derivable. Since $\|\Sigma a : \gamma.\tau_1\| = \|\tau_1\|$ and $\|\Gamma, x : \tau_1\| = \|\Gamma\|, x : \|\tau_1\|$, this leads to the following.

$$\frac{\|\Gamma\| \vdash \|e_1\| : \|\tau_1\| \quad \|\Gamma\|, x : \|\tau_1\| \vdash \|e_2\| : \|\tau_2\|}{\|\Gamma\| \vdash \textbf{let } x = \|e_1\| \textbf{ in } \|e_2\| \textbf{ end} : \|\tau_2\|} \textbf{ (ty-let)}$$

Note that $\|\textbf{let } \langle a \mid x\rangle = e_1 \textbf{ in } e_2 \textbf{ end}\|$ is $\textbf{let } x = \|e_1\| \textbf{ in } \|e_2\| \textbf{ end}$, and we are done.

All other cases can be treated similarly. ∎

Like $\mathrm{ML}_0^{\Pi}(C)$, the evaluation in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ can be simulated by the evaluation in ML_0. This is stated in the theorem below.

Theorem 5.1.3 *If $e \hookrightarrow_d v$ derivable in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$, then $\|e\| \hookrightarrow_0 \|v\|$ is derivable in ML_0.*

Proof This simply follows from a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_d v$. We present a few cases as follows.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_d v_1}{\langle i \mid e_1\rangle \hookrightarrow_d \langle i \mid v_1\rangle}$ Then $\|e_1\| \hookrightarrow_0 \|v_1\|$ is derivable by induction hypothesis. Since

$$\|\langle i \mid e_1\rangle\| = \|e_1\| \quad \text{and} \quad \|\langle i \mid v_1\rangle\| = \|v_1\|,$$

we are done.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_d \langle i \mid v_1\rangle \qquad e_2[a \mapsto i][x \mapsto v_1] \hookrightarrow_d v}{\textbf{let } \langle a \mid x\rangle = e_1 \textbf{ in } e_2 \textbf{ end} \hookrightarrow_d v}$ By induction hypothesis, both

$$\|e_1\| \hookrightarrow_0 \|\langle i \mid v_1\rangle\| \quad \text{and} \quad \|e_2[a \mapsto i][x \mapsto v_1]\| \hookrightarrow_0 \|v\|$$

are derivable. Note $\|\langle i \mid v_1\rangle\| = \|v_1\|$ and $\|e_2[a \mapsto i][x \mapsto v_1]\| = \|e_2\|[x \mapsto \|v_1\|]$. This leads to the following.

$$\frac{\|e_1\| \hookrightarrow_0 \|v_1\| \quad \|e_2\|[x \mapsto \|v_1\|] \hookrightarrow_0 \|v\|}{\textbf{let } x = \|e_1\| \textbf{ in } \|e_2\| \textbf{ end} \hookrightarrow_0 v}\ \textbf{(ev-let)}$$

Since $\|\textbf{let } \langle a \mid x\rangle = e_1 \textbf{ in } e_2 \textbf{ end}\|$ is $\textbf{let } x = \|e_1\| \textbf{ in } \|e_2\| \textbf{ end}$, we are done.

All other cases can be handled similarly. ∎

Like Lemma 4.1.11, the following lemma is needed in the proof of Theorem 5.1.5.

Lemma 5.1.4 *Given a value v_1 in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ such that $\phi; \cdot \vdash v_1 : \Sigma a : \gamma.\tau$ is derivable, v_1 must be of form $\langle i \mid v_2\rangle$ for some value v_2.*

Proof This follows from a structural induction on the derivation $\mathcal{D}$ of v_1.

$\mathcal{D} = \dfrac{\phi; \cdot \vdash v_1 : \tau_1 \qquad \phi \models \tau_1 \equiv \Sigma a : \gamma.\tau}{\phi; \cdot \vdash v_1 : \Sigma a : \gamma.\tau}$ Then τ_1 must be of form $\Sigma a : \gamma.\tau_1'$. By induction hypothesis, v_1 has the claimed form.

$\mathcal{D} = \dfrac{\phi; \cdot \vdash v : \tau[a \mapsto i] \qquad \phi \vdash i : \gamma}{\phi; \cdot \vdash \langle i \mid v\rangle : (\Sigma a : \gamma.\tau)}$ Then v_1 is $\langle i \mid v\rangle$, and we are done.

Note that the last applied rule in $\mathcal{D}$ cannot be **(ty-var)**. Since v_1 is a value, no other rules can be the last applied rule in $\mathcal{D}$. This concludes the proof. ∎

Theorem 5.1.5 *Given $\phi;\cdot \vdash e : \tau$ derivable in $\mathrm{ML}_0^{\Pi}(C)$. If $e^0 = \|e\| \hookrightarrow_0 v^0$ is derivable for some value v^0 in ML_0, then there exists a value v in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ such that $e \hookrightarrow_d v$ is derivable and $\|v\| = v^0$.*

Proof The theorem follows from a structural induction on the derivation of $e^0 \hookrightarrow_0 v^0$ and the derivation $\mathcal{D}$ of $\phi;\cdot \vdash e : \tau$, lexicographically ordered. We present a few cases.

$$\mathcal{D} = \frac{\phi;\cdot \vdash e_1 : \tau_1[a \mapsto i] \qquad \phi \vdash i : \gamma}{\phi;\cdot \vdash \langle i \mid e_1\rangle : (\Sigma a : \gamma.\tau_1)}$$

Then $\|\langle i \mid e_1\rangle\| = \|e_1\| \hookrightarrow_0 v^0$ is derivable in ML_0. By induction hypothesis, $e_1 \hookrightarrow_d v_1$ is derivable in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ such that $\|v_1\| = v^0$. This yields the following.

$$\frac{e_1 \hookrightarrow_d v_1}{\langle i \mid e_1\rangle \hookrightarrow_d \langle i \mid v_1\rangle}\ \textbf{(ev-sig-intro)}$$

Note that $\|\langle i \mid v_1\rangle\| = \|v_1\| = v^0$, and we are done.

$$\mathcal{D} = \frac{\phi;\cdot \vdash e_1 : (\Sigma a : \gamma.\tau_1) \qquad \phi, a : \gamma; x : \tau_1 \vdash e_2 : \tau}{\phi;\cdot \vdash \textbf{let } \langle a \mid x\rangle = e_1 \textbf{ in } e_2 \textbf{ end} : \tau}$$

Then the derivation of $\|e\| \hookrightarrow_0 v^0$ is of the following form

$$\frac{\|e_1\| \hookrightarrow_0 v_1^0 \quad \|e_2\|[x \mapsto v_1^0] \hookrightarrow_0 v^0}{\textbf{let } x = \|e_1\| \textbf{ in } \|e_2\| \textbf{ end} \hookrightarrow_0 v^0}\ \textbf{(ev-let)}$$

By induction hypothesis, $e_1 \hookrightarrow_d v_1$ is derivable for some v_1 such that $\|v_1\| = v_1^0$. By Theorem 5.1.1, $\phi;\cdot \vdash v_1 : (\Sigma a : \gamma.\tau_1)$ is derivable. Therefore, Lemma 5.1.4 implies that v_1 is of form $\langle i \mid v_2\rangle$ for some v_2. It then follows that both $\phi;\cdot \vdash v_2 : \tau_1[a \mapsto i]$ and $\phi \vdash i : \gamma$ are derivable. This leads to a derivation of $\phi;\cdot \vdash e_2[a \mapsto i][x \mapsto v_2] : \tau$ since τ contains no free occurrences of a. Notice $\|e_2[a \mapsto i][x \mapsto v_2]\| = \|e_2\|[x \mapsto v_1^0]$. By induction hypothesis, $e_2[a \mapsto i][x \mapsto v_2] \hookrightarrow_d v$ is derivable for some v such that $\|v\| = v^0$. Hence, we have the following, and we are done.

$$\frac{e_1 \hookrightarrow_d \langle i \mid v_2\rangle \quad e_2[a \mapsto i][x \mapsto v_2] \hookrightarrow_d v}{\textbf{let } \langle a \mid x\rangle = e_1 \textbf{ in } e_2 \textbf{ end} \hookrightarrow_d v}\ \textbf{(ev-sig-elim)}$$

All other cases can be handled similarly. ∎

As a consequence, it is straightforward to conclude that $\mathrm{ML}_0^{\Pi,\Sigma}(C)$, like $\mathrm{ML}_0^{\Pi}(C)$, is also a conservative extension of ML_0.

5.2 Elaboration

In order to make $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ suitable as a *practical* programming language, we have to be able to design a satisfactory elaboration algorithm from $\mathrm{DML}(C)$ to $\mathrm{ML}_0^{\Pi,\Sigma}(C)$, where $\mathrm{DML}(C)$ is basically the external language $\mathrm{DML}_0(C)$ present in Section 4.2 except that existential dependent types are allowed now. This turns out to be a challenging task.

We present a typical conflict which we are facing in order to do elaboration in this setting. Let us assign the type $\Pi n : \mathit{nat}.\mathtt{intlist}(n) \to \mathtt{intlist}(n)$ to the function rev which reverses

an integer list. Suppose that $rev(l)$ occurs in the code, where l is an integer list. Intuitively, we synthesize rev to $rev[i]$ for some index i subject to the satisfiability of the index constraints, and then we check l against type $\texttt{intlist}(i)$. Suppose we then need to synthesize l, obtaining some l^* with type $\Sigma n : nat.\texttt{intlist}(n)$. We now get stuck because l^* cannot be (successfully) checked against $\texttt{intlist}(i)$ for whatever i is, and a type error should then be reported. Nonetheless, it seems quite natural in this case to elaborate $rev(l)$ into

$$\textbf{let } \langle a \mid x \rangle = l^* \textbf{ in } \langle a \mid rev[a](x) \rangle \textbf{ end},$$

which is of type $\Sigma a : nat.\texttt{intlist}(a)$. This justifies the intuition that *reversing a list with unknown length yields a list with unknown length* [1]. The crucial step is to unpack l before we synthesize rev to $rev[i]$. Also notice that this elaboration of $rev(l)$ does not alter the operational semantics of $rev(l)$, although it changes the structure of the expression significantly.

This example suggest that we transform $rev(l)$ into $\textbf{let } x = l \textbf{ in } rev(x) \textbf{ end}$ before elaboration. In general, we can define a variant of A-normal transform (Moggi 1989; Sabry and Felleisen 1993) as follows, which transforms expressions e in DML(C) into $\underline{e}$.

$$\begin{array}{rcl}
\underline{x} & = & x \\
\underline{\textbf{lam } x.e} & = & \textbf{lam } x.\underline{e} \\
\underline{\textbf{lam } x : \tau.e} & = & \textbf{lam } x : \tau.\underline{e} \\
\underline{\textbf{fix } f.e} & = & \textbf{fix } f.\underline{e} \\
\underline{\textbf{fix } f : \tau.e} & = & \textbf{fix } f : \tau.\underline{e} \\
\underline{\langle\rangle} & = & \langle\rangle \\
\underline{c} & = & c \\
\underline{c(e)} & = & \textbf{let } x = \underline{e} \textbf{ in } c(x) \textbf{ end} \\
\underline{\textbf{case } e \textbf{ of } ms} & = & \textbf{let } x = \underline{e} \textbf{ in case } x \textbf{ of } \underline{ms} \textbf{ end} \\
\underline{p \Rightarrow e} & = & p \Rightarrow \underline{e} \\
\underline{\langle e_1, e_2 \rangle} & = & \textbf{let } x_1 = \underline{e_1} \textbf{ in let } x_2 = \underline{e_2} \textbf{ in } \langle x_1, x_2 \rangle \textbf{ end end} \\
\underline{e_1(e_2)} & = & \textbf{let } x_1 = \underline{e_1} \textbf{ in let } x_2 = \underline{e_2} \textbf{ in } x_1(x_2) \textbf{ end end} \\
\underline{\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end}} & = & \textbf{let } x = \underline{e_1} \textbf{ in } \underline{e_2} \textbf{ end} \\
\underline{e : \tau} & = & \underline{e} : \tau
\end{array}$$

The following proposition shows that $\underline{e}$ preserves the operational semantics of the transformed expression e.

Proposition 5.2.1 *We have* $|e| \cong |\underline{e}|$ *for all expressions* e *in* DML(C).

Proof With Corollary 2.3.13, this follows from a structural induction on e. ∎

The strategy to transform e into $\underline{e}$ before elaborating e means that we must synthesize the types of e_1 and e_2 in order to synthesize the type of an application $e_1(e_2)$ since it is transformed into $\textbf{let } x_1 = \underline{e_1} \textbf{ in let } x_2 = \underline{e_2} \textbf{ in } x_1(x_2) \textbf{ end end}$. Clearly, this strategy rules out the following style of elaboration, which would otherwise exist. For instance, let us assume that the type of e_1

[1] It is tempting to require that reversing a list with unknown length yield a list with the *same* unknown length. This, however, is not helpful to justify that $\langle l, rev(l) \rangle$ is a pair of lists with the same length if we enrich our language further to include effects. If l has no effects, this can be achieved using $\textbf{let } x = l \textbf{ in } \langle x, rev(x) \rangle \textbf{ end}$.

is $\Pi a : \gamma.(\delta(a) \to \delta(a)) \to \delta(a)$ and e_2 is **lam** $x.x$; then synthesizing the type of e_2 is clearly impossible but the type of $e_1(e_2)$ can nonetheless be synthesized as follows.

$$\frac{\dfrac{\phi;\Gamma \vdash e_1 \uparrow \Pi a : \gamma.(\delta(a) \to \delta(a)) \to \delta(a) \Rightarrow e_1^*}{\phi;\Gamma \vdash e_1 \uparrow (\delta(i) \to \delta(i)) \to \delta(i) \Rightarrow e_1^*[i]} \qquad \phi;\Gamma \vdash e_2 \downarrow \delta(i) \to \delta(i) \Rightarrow (\textbf{lam}\ x : \delta(i).x)}{\phi;\Gamma \vdash e_1(e_2) \uparrow \delta(i) \Rightarrow e_1^*[i](\textbf{lam}\ x : \delta(i).x)}$$

It has been observed that this style of programming does occur occasionally in practice. Therefore, we are prompted with a question about whether the above transform should always be performed before elaboration begins. There is no clear answer to this question at this moment. On one hand, we may require that the programmer perform the transform manually but this could be too much a burden. On the other hand, if the transform is always performed automatically, then we may lose the ability to elaborate some programs which would otherwise be possible. More importantly, this could make it much harder to report informative error messages during type-checking. Given that this issue has yet to be settled in practice, it is desirable for us to separate from elaboration the issue of transforming programs. We will address in Chapter 8 the practical issues involving program transform before elaboration.

In the following presentation, we will use $\vec{a}$ for a (possibly empty) sequence of index variables and $\vec{\gamma}$ for a (possibly empty) sequence of sorts. Also we use $\vec{a} : \vec{\gamma}$ for a sequence of declarations $a_1 : \gamma_1, \ldots, a_n : \gamma_n$, where $\vec{a} = a_1, \ldots, a_n$ and $\vec{\gamma} = \gamma_1, \ldots, \gamma_n$, and $\Sigma(\vec{a} : \vec{\gamma}).\tau$ for the following.

$$\Sigma(a_1 : \gamma_1) \ldots \Sigma(a_n : \gamma_n).\tau.$$

We use $\langle \vec{a} \mid e \rangle$ and **let** $\langle \vec{a} \mid x \rangle = e_1$ **in** e_2 **end** for the abbreviations defined as follows. If $\vec{a}$ is empty, we have

$$\langle \vec{a} \mid e \rangle = e \qquad\qquad \textbf{let}\ \langle \vec{a} \mid x \rangle = e_1\ \textbf{in}\ e_2\ \textbf{end} = \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}$$

and if $\vec{a}$ is $a, \vec{a_1}$, we have

$$\begin{array}{rcl} \langle \vec{a} \mid e \rangle & = & \langle a \mid \langle \vec{a_1} \mid e \rangle \rangle \\ \textbf{let}\ \langle \vec{a} \mid x \rangle = e_1\ \textbf{in}\ e_2\ \textbf{end} & = & \textbf{let}\ \langle a \mid x_1 \rangle = e_1\ \textbf{in let}\ \langle \vec{a_1} \mid x \rangle = x_1\ \textbf{in}\ e_2\ \textbf{end end} \end{array}$$

The following proposition presents some properties related to these abbreviations.

Proposition 5.2.2 *We have the following.*

1. $|\textbf{let}\ \langle \vec{a} \mid x \rangle = e_1\ \textbf{in}\ e_2\ \textbf{end}| \cong \textbf{let}\ x = |e_1|\ \textbf{in}\ |e_2|\ \textbf{end}$ *for expressions* e_1, e_2 *in* $\mathrm{ML}_0^{\Pi,\Sigma}(C)$.
2. *Suppose that both* $\phi;\Gamma \vdash e_1 : \Sigma(\vec{a} : \vec{\gamma}).\tau_1$ *and* $\phi, \vec{a} : \vec{\gamma};\Gamma, x : \tau_1 \vdash e_2 : \tau_2$ *are derivable. If none of the variables in* $\vec{a}$ *have free occurrences in* τ_2 *then* $\phi;\Gamma \vdash \textbf{let}\ \langle \vec{a} \mid x \rangle = e_1\ \textbf{in}\ e_2\ \textbf{end} : \tau_2$ *is derivable.*

Proof (1) simply follows from Corollary 2.3.13, and (2) follows from an induction on the number of index variables declared in $\vec{a}$. ∎

In addition, the above $rev(l)$ example suggests that we turn both the rules **(elab-let-up)** and **(elab-let-down)** into the following forms, respectively. In other words, we always unpack a **let**-bound expression if its synthesized type begins with existential quantifiers.

$$\frac{\phi;\Gamma \vdash e_1 \uparrow \Sigma(\vec{a} : \vec{\gamma}).\tau_1 \Rightarrow e_1^* \quad \phi, \vec{a} : \vec{\gamma};\Gamma, x : \tau_1 \vdash e_2 \uparrow \tau_2 \Rightarrow e_2^*}{\phi;\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} \uparrow \Sigma(\vec{a} : \vec{\gamma}).\tau_2 \Rightarrow \textbf{let}\ \langle \vec{a} \mid x \rangle = e_1^*\ \textbf{in}\ \langle \vec{a} \mid e_2^* \rangle\ \textbf{end}}$$

$$\frac{\phi;\Gamma \vdash e_1 \uparrow \Sigma(\vec{a}:\vec{\gamma}).\tau_1 \Rightarrow e_1^* \quad \phi,\vec{a}:\vec{\gamma};\Gamma,x:\tau_1 \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*}{\phi;\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \downarrow \tau_2 \Rightarrow \textbf{let } \langle \vec{a} \mid x\rangle = e_1^* \textbf{ in } e_2^* \textbf{ end}}$$

The rule **(elab-case)** must be dealt with similarly.

There is yet another issue. Suppose that we need to check an expression e against a type τ. If e is a variable or an application, we must synthesize the type of e, obtaining some type τ'. At this stage, we need to check whether an expression of type τ' can be *coerced* into one of type τ. The strategy used in the elaboration for $\mathrm{ML}_0^{\Pi}(C)$ is simply to check whether $\tau' \equiv \tau$ holds. However, this strategy is highly unsatisfactory for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ in practice. We are thus motivated to design a more effective approach to coercion.

5.2.1 Coercion

Given types τ_1 and τ_2 in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ such that $\|\tau_1\| = \|\tau_2\|$, a coercion from τ_1 to τ_2 is an *evaluation context* E such that for every expression e of type τ_1, $E[e]$ is of type τ_2 and $|e| \cong |E[e]|$.

In Figure 5.1 we present the rules for coercion in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$. A judgement of form $\phi \vdash \textbf{coerce}(\tau,\tau') \Rightarrow E$ means that every expression e of type τ can be coerced into expression $E[e]$ of type τ'.

Example 5.2.3 *We show that the type* $\tau = \Pi a : \gamma.\delta(a) \to \delta(a)$ *can be coerced into the type* $\tau' = \Pi a : \gamma.\delta(a) \to \Sigma b : \gamma.\delta(b)$.

$$\dfrac{\dfrac{\dfrac{a:\gamma \models a \doteq a}{a:\gamma \vdash \textbf{coerce}(\delta(a),\delta(a)) \Rightarrow []} \quad \dfrac{\dfrac{a:\gamma \models a \doteq a}{a:\gamma \vdash \textbf{coerce}(\delta(a),\delta(a)) \Rightarrow []}}{a:\gamma \vdash \textbf{coerce}(\delta(a),\Sigma b:\gamma.\delta(b)) \Rightarrow \langle a \mid []\rangle}}{a:\gamma \vdash \textbf{coerce}(\delta(a)\to\delta(a),\delta(a)\to\Sigma b:\gamma.\delta(b)) \Rightarrow \textbf{let } x_1 = [] \textbf{ in lam } x_2:\delta(a).\langle a \mid x_1(x_2)\rangle \textbf{ end}}}{a:\gamma \vdash \textbf{coerce}(\tau,\delta(a)\to\Sigma b:\gamma.\delta(b)) \Rightarrow \textbf{let } x_1 = [][a] \textbf{ in lam } x_2:\delta(a).\langle a \mid x_1(x_2)\rangle \textbf{ end}}}{\cdot \vdash \textbf{coerce}(\tau,\tau') \Rightarrow \lambda a:\gamma.\textbf{let } x_1 = [][a] \textbf{ in lam } x_2:\delta(a).\langle a \mid x_1(x_2)\rangle \textbf{ end}}$$

We are ready to prove the correctness of these coercion rules, which is stated as Theorem 5.2.4.

Theorem 5.2.4 *If* $\phi;\Gamma \vdash e : \tau$ *and* $\phi \vdash \textbf{coerce}(\tau,\tau') \Rightarrow E$ *are derivable, then* $\phi;\Gamma \vdash E[e] : \tau$ *is also derivable and* $|e| \cong |E[e]|$.

Proof This follows from a structural induction on the derivation $\mathcal{D}$ of $\phi \vdash \textbf{coerce}(\tau,\tau') \Rightarrow E$. We present several cases.

$\mathcal{D} = \dfrac{\phi \vdash \textbf{coerce}(\tau_1,\tau_1') \Rightarrow E_1 \quad \phi \vdash \textbf{coerce}(\tau_2,\tau_2') \Rightarrow E_2}{\phi \vdash \textbf{coerce}(\tau_1 * \tau_2, \tau_1' * \tau_2') \Rightarrow \textbf{case } [] \textbf{ of } \langle x_1,x_2\rangle \Rightarrow \langle E_1[x_1], E_2[x_2]\rangle}$ By induction hypothesis, $\phi;\Gamma,x_1:\tau_1,x_2:\tau_2 \vdash E_1[x_1] : \tau_1'$ and $\phi;\Gamma,x_1:\tau_1,x_2:\tau_2 \vdash E_2[x_2] : \tau_2'$ are derivable. This leads to the following derivation,

$$\dfrac{\phi;\Gamma \vdash e : \tau_1 * \tau_2 \quad \dfrac{\langle x_1,x_2\rangle \downarrow \tau_1 * \tau_2 \rhd (\cdot; x_1:\tau_1, x_2:\tau_2) \quad \mathcal{D}_0}{\phi;\Gamma \vdash \langle x_1,x_2\rangle \Rightarrow \langle E_1[x_1],E_2[x_2]\rangle : \tau_1*\tau_2 \Rightarrow \tau_1'*\tau_2'}\ \textbf{(ty-match)}}{\phi;\Gamma \vdash (\textbf{case } e \textbf{ of } \langle x_1,x_2\rangle \Rightarrow \langle E_1[x_1],E_2[x_2]\rangle) : \tau_1'*\tau_2'}\ \textbf{(ty-case)}$$

$$\frac{\phi \models i \doteq j}{\phi \vdash \mathbf{coerce}(\delta(i), \delta(j)) \Rightarrow []} \textbf{ (coerce-datatype)}$$

$$\frac{\vdash \phi[\mathbf{ictx}]}{\phi \vdash \mathbf{coerce}(\mathbf{1}, \mathbf{1}) \Rightarrow []} \textbf{ (coerce-unit)}$$

$$\frac{\phi \vdash \mathbf{coerce}(\tau_1, \tau_1') \Rightarrow E_1 \quad \phi \vdash \mathbf{coerce}(\tau_2, \tau_2') \Rightarrow E_2}{\phi \vdash \mathbf{coerce}(\tau_1 * \tau_2, \tau_1' * \tau_2') \Rightarrow \mathbf{case}\ []\ \mathbf{of}\ \langle x_1, x_2 \rangle \Rightarrow \langle E_1[x_1], E_2[x_2] \rangle} \textbf{ (coerce-prod)}$$

$$\frac{\phi \vdash \mathbf{coerce}(\tau_1', \tau_1) \Rightarrow E_1 \quad \phi \vdash \mathbf{coerce}(\tau_2, \tau_2') \Rightarrow E_2}{\phi \vdash \mathbf{coerce}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') \Rightarrow \mathbf{let}\ x_1 = []\ \mathbf{in}\ \mathbf{lam}\ x_2 : \tau_1'.E_2[x_1(E_1[x_2])]\ \mathbf{end}} \textbf{ (coerce-fun)}$$

$$\frac{\phi \vdash \mathbf{coerce}(\tau_1[a \mapsto i], \tau) \Rightarrow E \quad \phi \vdash i : \gamma}{\phi \vdash \mathbf{coerce}(\Pi a : \gamma.\tau_1, \tau) \Rightarrow E[[]][i]]} \textbf{ (coerce-pi-l)}$$

$$\frac{\phi, a : \gamma \vdash \mathbf{coerce}(\tau_1, \tau) \Rightarrow E}{\phi \vdash \mathbf{coerce}(\tau_1, \Pi a : \gamma.\tau) \Rightarrow \lambda a : \gamma.E} \textbf{ (coerce-pi-r)}$$

$$\frac{\phi, a : \gamma \vdash \mathbf{coerce}(\tau_1, \tau) \Rightarrow E}{\phi \vdash \mathbf{coerce}(\Sigma(a : \gamma).\tau_1, \tau) \Rightarrow \mathbf{let}\ \langle a \mid x \rangle = []\ \mathbf{in}\ E[x]\ \mathbf{end}} \textbf{ (coerce-sig-l)}$$

$$\frac{\phi \vdash \mathbf{coerce}(\tau_1, \tau[a \mapsto i]) \Rightarrow E \quad \phi \vdash i : \gamma}{\phi \vdash \mathbf{coerce}(\tau_1, \Sigma(a : \gamma).\tau) \Rightarrow \langle i \mid E \rangle} \textbf{ (coerce-sig-r)}$$

Figure 5.1: The derivation rules for coercion

where $\mathcal{D}_0$ is the following.

$$\frac{\phi;\Gamma, x_1:\tau_1, x_2:\tau_2 \vdash E_1[x_1]:\tau_1' \quad \phi;\Gamma, x_1:\tau_1, x_2:\tau_2 \vdash E_2[x_2]:\tau_2'}{\phi;\Gamma, x_1:\tau_1, x_2:\tau_2 \vdash \langle E_1[x_1], E_2[x_2]\rangle : \tau_1' * \tau_2'}\ \textbf{(ty-prod)}$$

In addition, we have $x_1 \cong |E_1[x_1]|$ and $x_2 \cong |E_2[x_2]|$. Therefore,

$$|\textbf{case } e \textbf{ of } \langle x_1, x_2\rangle \Rightarrow \langle E_1[x_1], E_2[x_2]\rangle| \cong |\textbf{case } e \textbf{ of } \langle x_1, x_2\rangle \Rightarrow \langle x_1, x_2\rangle|$$

Since e is of type $\tau_1 * \tau_2$, it can then be shown that $|e| \cong |\textbf{case } e \textbf{ of } \langle x_1, x_2\rangle \Rightarrow \langle x_1, x_2\rangle|$.

$$\mathcal{D} = \frac{\phi \vdash \textbf{coerce}(\tau_1', \tau_1) \Rightarrow E_1 \quad \phi \vdash \textbf{coerce}(\tau_2, \tau_2') \Rightarrow E_2}{\phi \vdash \textbf{coerce}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') \Rightarrow \textbf{let } x_1 = [] \textbf{ in lam } x_2 : \tau_1'.E_2[x_1(E_1[x_2])] \textbf{ end}}$$

By induction hypothesis, $\phi;\Gamma, x_2 : \tau_1' \vdash E_1[x_2] : \tau_1$ is derivable. This leads to the following.

$$\frac{\phi;\Gamma, x_1:\tau_1\to\tau_2, x_2:\tau_1' \vdash x_1 : \tau_1 \to \tau_2 \quad \phi;\Gamma, x_1:\tau_1\to\tau_2, x_2:\tau_1' \vdash E_1[x_2]:\tau_1}{\phi;\Gamma, x_1:\tau_1\to\tau_2, x_2:\tau_1' \vdash x_1(E_1[x_2]) : \tau_2}\ \textbf{(ty-app)}$$

Then by induction hypothesis again, $\phi;\Gamma, x_1 : \tau_1 \to \tau_2, x_2 : \tau_1' \vdash E_2[x_1(E_1[x_2])] : \tau_2'$ is derivable, and this yields the following.

$$\frac{\phi;\Gamma \vdash e : \tau_1 \to \tau_2 \quad \dfrac{\phi;\Gamma, x_1:\tau_1\to\tau_2, x_2:\tau_1' \vdash E_2[x_1(E_1[x_2])]:\tau_2'}{\phi;\Gamma, x_1:\tau_1\to\tau_2 \vdash \textbf{lam } x_2:\tau_1'.E_2[x_1(E_1[x_2])] : \tau_1' \to \tau_2'}\ \textbf{(ty-lam)}}{\phi;\Gamma \vdash \textbf{let } x_1 = e \textbf{ in lam } x_2 : \tau_1'.E_2[x_1(E_1[x_2])] \textbf{ end} : \tau_1' \to \tau_2'}\ \textbf{(ty-let)}$$

Also we have the following since $\phi;\Gamma \vdash e : \tau_1 \to \tau_2$ is derivable.

$$\begin{array}{rl} & |\textbf{let } x_1 = e \textbf{ in lam } x_2 : \tau_1'.E_2[x_1(E_1[x_2])] \textbf{ end}| \\ = & \textbf{let } x_1 = |e| \textbf{ in lam } x_2.|E_2[x_1(E_1[x_2])]| \textbf{ end} \\ \cong & \textbf{let } x_1 = |e| \textbf{ in lam } x_2.x_1(x_2) \textbf{ end} \\ \cong & \textbf{let } x_1 = |e| \textbf{ in } x_1 \textbf{ end} \ \text{(by Proposition 2.3.14 (1))} \\ \cong & |e| \end{array}$$

This wraps up the case.

$$\mathcal{D} = \frac{\phi \vdash \textbf{coerce}(\tau_1[a \mapsto i], \tau) \Rightarrow E \quad \phi \vdash i : \gamma}{\phi \vdash \textbf{coerce}(\Pi a : \gamma.\tau_1, \tau) \Rightarrow E[[]i]}$$

Since $\phi;\Gamma \vdash e : \Pi a : \gamma.\tau_1$, we have the following.

$$\frac{\phi;\Gamma \vdash e : \Pi a : \gamma.\tau_1 \quad \phi \vdash i : \gamma}{\phi;\Gamma \vdash e[i] : \tau_1[a \mapsto i]}\ \textbf{(ty-iapp)}$$

By induction hypothesis, $\phi;\Gamma \vdash E[e[i]] : \tau$ is derivable and $|e[i]| \cong |E[e[i]]|$. Note $|e| = |e[i]|$, and we are done.

$$\mathcal{D} = \frac{\phi, a : \gamma \vdash \textbf{coerce}(\tau_1, \tau) \Rightarrow E}{\phi \vdash \textbf{coerce}(\tau_1, \Pi a : \gamma.\tau) \Rightarrow \lambda a.\gamma.E}$$

By induction hypothesis, $\phi, a : \gamma;\Gamma \vdash E[e] : \tau$ is derivable and $|e| \cong |E[e]|$. Since there are no free occurrences of a in the types of the variables declared in Γ, we have the following.

$$\frac{\phi, a:\gamma;\Gamma \vdash E[e] : \tau}{\phi, a:\gamma;\Gamma \vdash \lambda a : \gamma.E[e] : \tau}\ \textbf{(ty-ilam)}$$

Also $|\lambda a : \gamma.E[e]| = |E[e]| \cong |e|$. Hence we are done.

$\mathcal{D} = \dfrac{\phi, a : \gamma \vdash \mathbf{coerce}(\tau_1, \tau) \Rightarrow E}{\phi \vdash \mathbf{coerce}(\Sigma(a : \gamma).\tau_1, \tau) \Rightarrow \mathbf{let}\ \langle a \mid x \rangle = [\,]\ \mathbf{in}\ E[x]\ \mathbf{end}}$ By induction hypothesis, $\phi, a : \gamma; \Gamma, x : \tau_1 \vdash E[x] : \tau$ is derivable. This leads to the following.

$$\frac{\phi; \Gamma \vdash e : \Sigma(a : \gamma).\tau_1 \quad \phi, a : \gamma; \Gamma, x : \tau_1 \vdash E[x] : \tau}{\phi; \Gamma \vdash \mathbf{let}\ \langle a \mid x \rangle = e\ \mathbf{in}\ E[x]\ \mathbf{end}}\ \textbf{(ty-sig-elim)}$$

Notice that

$$|\mathbf{let}\ \langle a \mid x \rangle = e\ \mathbf{in}\ E[x]\ \mathbf{end}| = \mathbf{let}\ x = |e|\ \mathbf{in}\ |E[x]|\ \mathbf{end} \cong \mathbf{let}\ x = |e|\ \mathbf{in}\ x\ \mathbf{end} \cong |e|.$$

Hence we are done.

$\mathcal{D} = \dfrac{\phi \vdash \mathbf{coerce}(\tau_1, \tau[a \mapsto i]) \Rightarrow E \qquad \phi \vdash i : \gamma}{\phi \vdash \mathbf{coerce}(\tau_1, \Sigma(a : \gamma).\tau) \Rightarrow \langle i \mid E \rangle}$ By induction hypothesis, $\phi; \Gamma \vdash E[e] : \tau[a \mapsto i]$ is derivable and $|e| \cong |E[e]|$. This leads to the following.

$$\frac{\phi; \Gamma \vdash E[e] : \tau[a \mapsto i] \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash \langle i \mid E[e] \rangle : \Sigma a : \gamma.\tau}\ \textbf{(ty-sig-intro)}$$

Also $|\langle i \mid E[e] \rangle| = |E[e]| \cong |e|$, and we are done.

All the rest of cases can be treated similarly. ∎

As usual, there is a gap between the elaboration rules for coercion and their implementation. We bridge the gap by presenting the constraint generation rules for coercion in Figure 5.2. A judgement of form $\phi \vdash [\psi]\ \mathbf{coerce}(\tau, \tau') \Rightarrow \Phi$ means that coercing τ into τ' under context ϕ yields a constraint Φ in which all existential variables are declared in ψ.

Theorem 5.2.5 *Assume that $\phi \vdash [\psi]\ \mathbf{coerce}(\tau, \tau') \Rightarrow \Phi$ is derivable. If $\phi[\theta] \models \Phi[\theta]$ is derivable for some existential substitution θ such that $\phi \rhd \theta : \psi$ holds, then $\phi[\theta] \vdash \mathbf{coerce}(\tau[\theta], \tau'[\theta]) \Rightarrow E$ is derivable for some evaluation context E.*

Proof The proof proceeds by a structural induction on the derivation $\mathcal{D}$ of $\phi \vdash [\psi]\ \mathbf{coerce}(\tau, \tau') \Rightarrow \Phi$. We present a few cases.

$\mathcal{D} = \dfrac{\phi \vdash [\psi]\ \mathbf{coerce}(\tau_1, \tau_1') \Rightarrow \Phi_1 \qquad \phi \vdash [\psi]\ \mathbf{coerce}(\tau_2, \tau_2') \Rightarrow \Phi_2}{\phi \vdash [\psi]\ \mathbf{coerce}(\tau_1 * \tau_2, \tau_1' * \tau_2') \Rightarrow \Phi_1 \wedge \Phi_2}$ Then $\phi[\theta] \models (\Phi_1 \wedge \Phi_2)[\theta]$ is derivable, and this implies both $\phi[\theta] \models \Phi_1[\theta]$ and $\phi[\theta] \models \Phi_2[\theta]$ are derivable. By induction hypothesis, there are evaluation contexts E_1 and E_2 such that $\phi[\theta] \vdash \mathbf{coerce}(\tau_1[\theta], \tau_1'[\theta]) \Rightarrow E_1$ and $\phi[\theta] \vdash \mathbf{coerce}(\tau_2[\theta], \tau_2'[\theta]) \Rightarrow E_2$ are derivable. This yields the following.

$$\frac{\phi[\theta] \vdash \mathbf{coerce}(\tau_1[\theta], \tau_1'[\theta]) \Rightarrow E_1 \quad \phi[\theta] \vdash \mathbf{coerce}(\tau_2[\theta], \tau_2'[\theta]) \Rightarrow E_2}{\phi[\theta] \vdash \mathbf{coerce}(\tau_1[\theta] * \tau_2[\theta], \tau_1'[\theta] * \tau_2'[\theta]) \Rightarrow \mathbf{case}\ [\,]\ \mathbf{of}\ \langle x_1, x_2 \rangle \Rightarrow \langle E_1[x_1], E_2[x_2] \rangle}$$

$$\frac{(\phi \mid \psi) \vdash \delta(i) \quad (\phi \mid \psi) \vdash \delta(j)}{\phi \vdash [\psi]\ \mathbf{coerce}(\delta(i), \delta(j)) \Rightarrow i \doteq j}\ \textbf{(co-constr-datatype)}$$

$$\frac{\vdash (\phi \mid \psi)[\mathbf{ictx}]}{\phi \vdash [\psi]\ \mathbf{coerce}(\mathbf{1}, \mathbf{1}) \Rightarrow \top}\ \textbf{(co-constr-unit)}$$

$$\frac{\phi \vdash [\psi]\ \mathbf{coerce}(\tau_1, \tau_1') \Rightarrow \Phi_1 \quad \phi \vdash [\psi]\ \mathbf{coerce}(\tau_2, \tau_2') \Rightarrow \Phi_2}{\phi \vdash [\psi]\ \mathbf{coerce}(\tau_1 * \tau_2, \tau_1' * \tau_2') \Rightarrow \Phi_1 \wedge \Phi_2}\ \textbf{(co-constr-prod)}$$

$$\frac{\phi \vdash [\psi]\ \mathbf{coerce}(\tau_1', \tau_1) \Rightarrow \Phi_1 \quad \phi \vdash [\psi]\ \mathbf{coerce}(\tau_2, \tau_2') \Rightarrow \Phi_2}{\phi \vdash [\psi]\ \mathbf{coerce}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') \Rightarrow \Phi_1 \wedge \Phi_2}\ \textbf{(co-constr-fun)}$$

$$\frac{\phi \vdash [\psi, A : \gamma]\ \mathbf{coerce}(\tau_1[a \mapsto A], \tau) \Rightarrow \Phi}{\phi \vdash [\psi]\ \mathbf{coerce}(\Pi a : \gamma.\tau_1, \tau) \Rightarrow \exists A : \gamma.\Phi}\ \textbf{(co-constr-pi-l)}$$

$$\frac{\phi, a^{\psi} : \gamma \vdash [\psi]\ \mathbf{coerce}(\tau_1, \tau[a \mapsto a^{\psi}]) \Rightarrow \Phi}{\phi \vdash [\psi]\ \mathbf{coerce}(\tau_1, \Pi a : \gamma.\tau) \Rightarrow \forall (a^{\psi} : \gamma).\Phi}\ \textbf{(co-constr-pi-r)}$$

$$\frac{\phi, a^{\psi} : \gamma \vdash [\psi]\ \mathbf{coerce}(\tau_1[a \mapsto a^{\psi}], \tau) \Rightarrow \Phi}{\phi \vdash [\psi]\ \mathbf{coerce}(\Sigma(a : \gamma).\tau_1, \tau) \Rightarrow \forall (a^{\psi} : \gamma).\Phi}\ \textbf{(co-constr-sig-l)}$$

$$\frac{\phi \vdash [\psi, A : \gamma]\ \mathbf{coerce}(\tau_1, \tau[a \mapsto A]) \Rightarrow \Phi}{\phi \vdash [\psi]\ \mathbf{coerce}(\tau_1, \Sigma(a : \gamma).\tau) \Rightarrow \exists A : \gamma.\Phi}\ \textbf{(co-constr-sig-r)}$$

Figure 5.2: The constraint generation rules for coercion

$\mathcal{D} = \dfrac{\phi \vdash [\psi, A : \gamma]\ \mathbf{coerce}(\tau_1[a \mapsto A], \tau) \Rightarrow \Phi_1}{\phi \vdash [\psi]\ \mathbf{coerce}(\Pi a : \gamma.\tau_1, \tau) \Rightarrow \exists A : \gamma.\Phi_1}$ Then $\phi[\theta] \models (\exists A : \gamma.\Phi_1)[\theta]$ is derivable. Since $(\exists A : \gamma.\Phi_1)[\theta]$ is $\exists A : \gamma[\theta].\Phi_1[\theta]$, there exists some i such that $\phi[\theta] \vdash i : \gamma[\theta]$ and $\phi[\theta] \models \Phi_1[\theta_1]$ for $\theta_1 = \theta[A \mapsto i]$. Clearly, $\phi[\theta_1] = \phi[\theta]$. By induction hypothesis,

$$\phi[\theta_1] \vdash \mathbf{coerce}(\tau[\theta_1], \tau'[\theta_1]) \Rightarrow E_1$$

is derivable for some evaluation context E_1. Note $(\tau_1[a \mapsto A])[\theta_1] = (\tau_1[a \mapsto i])[\theta]$ and $\tau[\theta_1] = \tau[\theta]$. This leads to the following.

$$\frac{\phi[\theta] \vdash \mathbf{coerce}(\tau_1[\theta][a \mapsto i], \tau[\theta]) \Rightarrow E_1 \quad \phi[\theta] \vdash i : \gamma[\theta]}{\phi[\theta] \vdash \mathbf{coerce}(\Pi a : \gamma.\tau_1[\theta], \tau[\theta]) \Rightarrow E_1[[][i]]}\ \textbf{(co-constr-pi-l)}$$

$\mathcal{D} = \dfrac{\phi, a^\psi : \gamma \vdash [\psi]\ \mathbf{coerce}(\tau_1[a \mapsto a^\psi], \tau) \Rightarrow \Phi_1}{\phi \vdash [\psi]\ \mathbf{coerce}(\Sigma a : \gamma.\tau_1, \tau) \Rightarrow \forall (a^\psi : \gamma).\Phi_1}$ Then $\phi[\theta] \models (\Pi a^\psi : \gamma.\Phi_1)[\theta]$ is derivable for some θ such that $\phi \rhd \theta : \psi$ holds. Notice that $(\Pi a^\psi : \gamma.\Phi_1)[\theta]$ is $\Pi a^\psi : \gamma[\theta].\Phi_1[\theta]$. Hence, $\phi[\theta], a^\psi : \gamma[\theta] \models \Phi_1[\theta]$ is derivable. By induction hypothesis, the following is derivable for some E_1.

$$\phi[\theta], a^\psi : \gamma[\theta] \vdash \mathbf{coerce}((\tau_1[a \mapsto a^\psi])[\theta], \tau[\theta]) \Rightarrow E_1$$

Note that $(\tau_1[a \mapsto a^\psi])[\theta] = \tau_1[\theta][a \mapsto a^\psi]$. This leads to the following.

$$\frac{\phi[\theta], a^\psi : \gamma[\theta] \vdash \mathbf{coerce}(\tau_1[\theta][a \mapsto a^\psi], \tau[\theta]) \Rightarrow E_1}{\phi[\theta] \vdash \mathbf{coerce}(\Sigma a : \gamma[\theta].\tau_1[\theta], \tau[\theta]) \Rightarrow \mathbf{let}\ \langle a \mid x \rangle = []\ \mathbf{in}\ E_1[x]\ \mathbf{end}}\ \textbf{(co-constr-sig-l)}$$

Hence, we are done.

All other cases can be handled similarly. ∎

We now have justified the correctness of the constraint generation rules for coercion. However, there is still some indeterminacy in these rules, which we will address in Chapter 8.

5.2.2 Elaboration as Static Semantics

We list the elaboration rules for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ in Figure 5.3 and Figure 5.4. The meaning of the judgements $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow e^*$ and $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*$ are basically the same as that of the judgements given in Figure 4.9 and Figure 4.10.

The following theorem justifies the correctness of these rules.

Theorem 5.2.6 *We have the following.*

1. *If $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow e^*$ is derivable, then $\phi; \Gamma \vdash e^* : \tau$ is derivable and $|e| \cong |e^*|$.*
2. *If $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*$ is derivable, then $\phi; \Gamma \vdash e^* : \tau$ is derivable and $|e| \cong |e^*|$.*

Proof The proof is parallel to that of Theorem 4.2.2. (1) and (2) follow straightforwardly from a simultaneous structural induction on the derivations $\mathcal{D}$ of $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow e^*$ and $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*$. We present a few cases.

$$\frac{\phi;\Gamma \vdash e \uparrow \Pi a:\gamma.\tau \Rightarrow e^* \quad \phi \vdash i:\gamma}{\phi;\Gamma \vdash e \uparrow \tau[a \mapsto i] \Rightarrow e^*[i]}\ \textbf{(elab-pi-elim)}$$

$$\frac{\phi, a:\gamma;\Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi;\Gamma \vdash e \downarrow \Pi a:\gamma.\tau \Rightarrow (\lambda a:\gamma.e^*)}\ \textbf{(elab-pi-intro-1)}$$

$$\frac{\phi, a:\gamma;\Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi;\Gamma \vdash \lambda a:\gamma.e \downarrow \Pi a:\gamma.\tau \Rightarrow (\lambda a:\gamma.e^*)}\ \textbf{(elab-pi-intro-2)}$$

$$\frac{\phi;\Gamma \vdash e \downarrow \tau[a \mapsto i] \Rightarrow e^* \quad \phi \vdash i:\gamma}{\phi;\Gamma \vdash e \downarrow \Sigma a:\gamma.\tau \Rightarrow \langle i \mid e^* \rangle}\ \textbf{(elab-sig-intro)}$$

$$\frac{\Gamma(x) = \tau \quad \phi \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash x \uparrow \tau \Rightarrow x}\ \textbf{(elab-var-up)}$$

$$\frac{\phi;\Gamma \vdash x \uparrow \tau_1 \Rightarrow e^* \quad \phi \vdash \mathbf{coerce}(\tau_1, \tau_2) \Rightarrow E}{\phi;\Gamma \vdash x \downarrow \tau_2 \Rightarrow E[e^*]}\ \textbf{(elab-var-down)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1:\gamma_1 \ldots \Pi a_n:\gamma_n.\delta(i) \quad \phi \vdash i_1:\gamma_1 \ \cdots\ \phi \vdash i_n:\gamma_n}{\phi;\Gamma \vdash c \uparrow \delta(i[a_1, \ldots, a_n \mapsto i_1, \ldots i_n]) \Rightarrow c[i_1] \ldots [i_n]}\ \textbf{(elab-cons-wo-up)}$$

$$\frac{\phi;\Gamma \vdash c \uparrow \delta(i) \Rightarrow e^* \quad \phi \models i \doteq j}{\phi;\Gamma \vdash c \downarrow \delta(j) \Rightarrow e^*}\ \textbf{(elab-cons-wo-down)}$$

$$\frac{\begin{array}{c}\mathcal{S}(c) = \Pi a_1:\gamma_1 \ldots \Pi a_n:\gamma_n.\tau \to \delta(i) \\ \phi;\Gamma \vdash e \downarrow \tau[a_1, \ldots, a_n \mapsto i_1, \ldots i_n] \Rightarrow e^* \\ \phi \vdash i_1:\gamma_1 \ \cdots\ \phi \vdash i_n:\gamma_n\end{array}}{\phi;\Gamma \vdash c(e) \uparrow \delta(i[a_1, \ldots, a_n \mapsto i_1, \ldots i_n]) \Rightarrow c[i_1] \ldots [i_n](e^*)}\ \textbf{(elab-cons-w-up)}$$

$$\frac{\phi;\Gamma \vdash c(e) \uparrow \delta(i) \Rightarrow e^* \quad \phi \models i \doteq j}{\phi;\Gamma \vdash c(e) \downarrow \delta(j) \Rightarrow e^*}\ \textbf{(elab-cons-w-down)}$$

$$\frac{}{\phi;\Gamma \vdash \langle\rangle \uparrow \mathbf{1} \Rightarrow \langle\rangle}\ \textbf{(elab-unit-up)}$$

$$\frac{}{\phi;\Gamma \vdash \langle\rangle \downarrow \mathbf{1} \Rightarrow \langle\rangle}\ \textbf{(elab-unit-down)}$$

$$\frac{\phi;\Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow e_1^* \quad \phi;\Gamma \vdash e_2 \uparrow \tau_2 \Rightarrow e_2^*}{\phi;\Gamma \vdash \langle e_1, e_2\rangle \uparrow \tau_1 * \tau_2 \Rightarrow \langle e_1^*, e_2^*\rangle}\ \textbf{(elab-prod-up)}$$

$$\frac{\phi;\Gamma \vdash e_1 \downarrow \tau_1 \Rightarrow e_1^* \quad \phi;\Gamma \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*}{\phi;\Gamma \vdash \langle e_1, e_2\rangle \downarrow \tau_1 * \tau_2 \Rightarrow \langle e_1^*, e_2^*\rangle}\ \textbf{(elab-prod-down)}$$

Figure 5.3: The elaboration rules for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ (I)

$$\frac{p \downarrow \tau_1 \Rightarrow (p^*; \phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e \downarrow \tau_2 \Rightarrow e^* \quad \phi \vdash \tau_2 : *}{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (p^* \Rightarrow e^*)} \textbf{ (elab-match)}$$

$$\frac{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (p^* \Rightarrow e^*) \quad \phi; \Gamma \vdash ms \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow ms^*}{\phi; \Gamma \vdash (p \Rightarrow e \mid ms) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (p^* \Rightarrow e^* \mid ms^*)} \textbf{ (elab-matches)}$$

$$\frac{\phi; \Gamma \vdash e \uparrow \tau_1 \Rightarrow e^* \qquad \phi; \Gamma \vdash ms \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow ms^*}{\phi; \Gamma \vdash (\textbf{case } e \textbf{ of } ms) \downarrow \tau_2 \Rightarrow (\textbf{case } e^* \textbf{ of } ms^*)} \textbf{ (elab-case)}$$

$$\frac{\phi; \Gamma, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow e^*}{\phi; \Gamma \vdash (\textbf{lam } x.e) \downarrow \tau_1 \to \tau_2 \Rightarrow (\textbf{lam } x : \tau_1.e_1^*)} \textbf{ (elab-lam)}$$

$$\frac{\phi; x : \tau \vdash e \downarrow \tau_2 \Rightarrow e^* \quad \phi \vdash \textbf{coerce}(\tau_1, \tau) \Rightarrow E}{\phi; \Gamma \vdash (\textbf{lam } x : \tau.e) \downarrow \tau_1 \to \tau_2 \Rightarrow (\textbf{lam } x_1 : \tau_1.\textbf{let } x = E[x_1] \textbf{ in } e^* \textbf{ end})} \textbf{ (elab-lam-anno)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \to \tau_2 \Rightarrow e_1^* \quad \phi; \Gamma \vdash e_2 \downarrow \tau_1 \Rightarrow e_2^*}{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_2 \Rightarrow e_1^*(e_2^*)} \textbf{ (elab-app-up)}$$

$$\frac{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_1 \Rightarrow e^* \quad \phi \vdash \textbf{coerce}(\tau_1, \tau_2) \Rightarrow E}{\phi; \Gamma \vdash e_1(e_2) \downarrow \tau_2 \Rightarrow E[e^*]} \textbf{ (elab-app-down)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \Sigma(\vec{a} : \vec{\gamma}).\tau_1 \Rightarrow e_1^* \quad \phi, \vec{a} : \vec{\gamma}; \Gamma, x : \tau_1 \vdash e_2 \uparrow \tau_2 \Rightarrow e_2^*}{\phi; \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \uparrow \Sigma(\vec{a} : \vec{\gamma}).\tau_2 \Rightarrow \textbf{let } \langle \vec{a} \mid x \rangle = e_1^* \textbf{ in } \langle \vec{a} \mid e_2^* \rangle \textbf{ end}} \textbf{ (elab-let-up)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \Sigma(\vec{a} : \vec{\gamma}).\tau_1 \Rightarrow e_1^* \quad \phi, \vec{a} : \vec{\gamma}; \Gamma, x : \tau_1 \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*}{\phi; \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \downarrow \tau_2 \Rightarrow \textbf{let } \langle \vec{a} \mid x \rangle = e_1^* \textbf{ in } e_2^* \textbf{ end}} \textbf{ (elab-let-down)}$$

$$\frac{\phi; \Gamma, f : \tau \vdash u \downarrow \tau \Rightarrow u^*}{\phi; \Gamma \vdash (\textbf{fix } f : \tau.u) \uparrow \tau \Rightarrow (\textbf{fix } f : \tau.u^*)} \textbf{ (elab-fix-up)}$$

$$\frac{\phi; \Gamma, f : \tau \vdash u \downarrow \tau \Rightarrow u^* \quad \phi \vdash \textbf{coerce}(\tau, \tau') \Rightarrow E}{\phi; \Gamma \vdash (\textbf{fix } f : \tau.u) \downarrow \tau' \Rightarrow \textbf{let } x = (\textbf{fix } f : \tau.u^*) \textbf{ in } E[x] \textbf{ end}} \textbf{ (elab-fix-down)}$$

$$\frac{\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi; \Gamma \vdash (e : \tau) \uparrow \tau \Rightarrow e^*} \textbf{ (elab-anno-up)}$$

$$\frac{\phi; \Gamma \vdash (e : \tau) \uparrow \tau_1 \Rightarrow e^* \quad \phi \vdash \textbf{coerce}(\tau_1, \tau_2) \Rightarrow E}{\phi; \Gamma \vdash (e : \tau) \downarrow \tau_2 \Rightarrow E[e^*]} \textbf{ (elab-anno-down)}$$

Figure 5.4: The elaboration rules for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ (II)

$\mathcal{D} = \dfrac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \to \tau_2 \Rightarrow e_1^* \qquad \phi; \Gamma \vdash e_2 \downarrow \tau_1 \Rightarrow e_2^*}{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_2 \Rightarrow e_1^*(e_2^*)}$ Then by induction hypothesis, both $\phi; \Gamma \vdash e_1^* : \tau_1 \to \tau_2$ and $\phi; \Gamma \vdash e_2^* : \tau_1$ are derivable. This leads to the following.

$$\frac{\phi; \Gamma \vdash e_1^* : \tau_1 \to \tau_2 \quad \phi; \Gamma \vdash e_2^* : \tau_1}{\phi; \Gamma \vdash e_1^*(e_2^*) : \tau_2} \textbf{ (ty-app)}$$

Note $|e_1^*(e_2^*)| = |e_1^*|(|e_2^*|) \cong |e_1|(|e_2|)$, and we are done.

$\mathcal{D} = \dfrac{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_1 \Rightarrow e^* \qquad \phi \vdash \textbf{coerce}(\tau_1, \tau_2) \Rightarrow E}{\phi; \Gamma \vdash e_1(e_2) \downarrow \tau_2 \Rightarrow E[e^*]}$ Then by induction hypothesis, $\phi; \Gamma \vdash e^* : \tau_1$ is derivable and $|e^*| \cong |e_1(e_2)|$. Since $\phi \vdash \textbf{coerce}(\tau_1, \tau_2) \Rightarrow E$ holds, $\phi; \Gamma \vdash E[e^*] : \tau_2$ is derivable by Theorem 5.2.4 and $|e^*| \cong |E[e^*]|$. Hence, $|e_1(e_2)| \cong |E[e^*]|$ and we are done.

$\mathcal{D} = \dfrac{\phi; \Gamma \vdash e_1 \uparrow \Sigma(\vec{a} : \vec{\gamma}).\tau_1 \Rightarrow e_1^* \qquad \phi, \vec{a} : \vec{\gamma}; \Gamma, x : \tau_1 \vdash e_2 \uparrow \tau_2 \Rightarrow e_2^*}{\phi; \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \uparrow \Sigma(\vec{a} : \vec{\gamma}).\tau_2 \Rightarrow \textbf{let } \langle \vec{a} \mid x \rangle = e_1^* \textbf{ in } \langle \vec{a} \mid e_2^* \rangle \textbf{ end}}$ By induction hypothesis, both $\phi; \Gamma \vdash e_1^* : \Sigma(\vec{a} : \vec{\gamma}).\tau_1$ and $\phi, \vec{a} : \vec{\gamma}; \Gamma, x : \tau_1 \vdash e_2^* : \tau_2$ are derivable. Hence, $\phi, \vec{a} : \vec{\gamma}; \Gamma, x : \tau_1 \vdash e_2^* : \Sigma(\vec{a} : \vec{\gamma}).\tau_2$ is also derivable by applying the rule **(trule-sig-intro)** repeatedly. Then by Proposition 5.2.2, the following is derivable.

$$\phi; \Gamma \vdash \textbf{let } \langle \vec{a} \mid x \rangle = e_1^* \textbf{ in } e_2^* \textbf{ end} : \Sigma(\vec{a} : \vec{\gamma}).\tau_2$$

Note that we have the following.

$$|\textbf{let } \langle \vec{a} \mid x \rangle = e_1^* \textbf{ in } e_2^* \textbf{ end}| = \textbf{let } x = |e_1^*| \textbf{ in } |e_2^*| \textbf{ end} \cong \textbf{let } x = |e_1| \textbf{ in } |e_2| \textbf{ end}$$

Hence we are done.

$\mathcal{D} = \dfrac{\phi; \Gamma \vdash e_1 \uparrow \Sigma(\vec{a} : \vec{\gamma}).\tau_1 \Rightarrow e_1^* \qquad \phi, \vec{a} : \vec{\gamma}; \Gamma, x : \tau_1 \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*}{\phi; \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \downarrow \tau_2 \Rightarrow \textbf{let } \langle \vec{a} \mid x \rangle = e_1^* \textbf{ in } e_2^* \textbf{ end}}$ By induction hypothesis, both $\phi; \Gamma \vdash e_1^* : \Sigma(\vec{a} : \vec{\gamma}).\tau_1$ and $\phi, \vec{a} : \vec{\gamma}; \Gamma, x : \tau_1 \vdash e_2^* : \tau_2$ are derivable. Therefore, the following is derivable by Proposition 5.2.2.

$$\phi; \Gamma \vdash \textbf{let } \langle \vec{a} \mid x \rangle = e_1^* \textbf{ in } e_2^* \textbf{ end} : \tau_2$$

Note that we have the following.

$$|\textbf{let } \langle \vec{a} \mid x \rangle = e_1^* \textbf{ in } e_2^* \textbf{ end}| = \textbf{let } x = |e_1^*| \textbf{ in } |e_2^*| \textbf{ end} \cong \textbf{let } x = |e_1| \textbf{ in } |e_2| \textbf{ end}$$

Hence we are done.

$\mathcal{D} = \dfrac{\phi; \Gamma, f : \tau \vdash u \downarrow \tau \Rightarrow u^*}{\phi; \Gamma \vdash (\textbf{fix } f : \tau.u) \uparrow \tau \Rightarrow (\textbf{fix } f : \tau.u^*)}$ By induction hypothesis, $\phi; \Gamma, f : \tau \vdash u^* : \tau$ is derivable. This yields the following derivation.

$$\frac{\phi; \Gamma, f : \tau \vdash u^* : \tau}{\phi; \Gamma \vdash (\textbf{fix } f : \tau.u^*) : \tau} \textbf{ (ty-fix)}$$

Also we have $|\textbf{fix } f : \tau.u^*| = \textbf{fix } f.|u^*| \cong \textbf{fix } f.|u| = |\textbf{fix } f : \tau.u|$. This concludes the case.

All other cases can be handled similar. ∎

5.2.3 Elaboration as Constraint Generation

As usual, there is still a gap between the description of elaboration rules for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ and an actual implementation. In order to bridge the gap, we list the constraint generation rules in Figure 5.5 and Figure 5.6.

The correctness of the constraint generation rules for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ is justified by the following theorem, which corresponds to Theorem 4.2.5.

Theorem 5.2.7 *We have the following.*

1. *Suppose that* $\Gamma \vdash e \uparrow \tau \Rightarrow[\psi]\ \Phi$ *is derivable. If* $\phi[\theta] \models \Phi[\theta]$ *is provable for some* θ *such that* $\phi \rhd \theta : \psi$ *is derivable, then there exists* e^* *such that* $\phi[\theta]; \Gamma[\theta] \vdash e \uparrow \tau[\theta] \Rightarrow e^*$ *is derivable.*
2. *Suppose that* $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow[\psi]\ \Phi$ *is derivable. If* $\phi[\theta] \models \Phi[\theta]$ *is provable for some* θ *such that* $\phi \rhd \theta : \psi$ *is derivable, then there exists* e^* *such that* $\phi[\theta]; \Gamma[\theta] \vdash e \downarrow \tau[\theta] \Rightarrow e^*$ *is derivable.*

Proof (1) and (2) are proven simultaneously by a structural induction on the derivations $\mathcal{D}$ of $\Gamma \vdash e \uparrow \tau \Rightarrow[\psi]\ \Phi$ and $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow[\psi]\ \Phi$. The proof is parallel to that of Theorem 4.2.5. We present a few cases.

$$\mathcal{D} = \frac{\phi; \Gamma, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow[\psi]\ \Phi}{\phi; \Gamma \vdash (\mathbf{lam}\ x.e) \downarrow \tau_1 \to \tau_2 \Rightarrow[\psi]\ \Phi}$$

By induction hypothesis, $\phi[\theta]; \Gamma[\theta], x : \tau_1[\theta] \vdash e \downarrow \tau[\theta] \Rightarrow e^*$ is derivable, and this yields the following.

$$\frac{\phi[\theta]; \Gamma[\theta], x : \tau_1[\theta] \vdash e \downarrow \tau[\theta] \Rightarrow e^*}{\phi[\theta]; \Gamma[\theta] \vdash (\mathbf{lam}\ x.e) \downarrow \tau_1[\theta] \to \tau[\theta] \Rightarrow \mathbf{lam}\ x : \tau_1[\theta].e^*}\ \textbf{(elab-lam)}$$

Note that $(\tau_1 \to \tau)[\theta]$ is $\tau_1[\theta] \to \tau[\theta]$, and we are done.

$$\mathcal{D} = \frac{\begin{array}{c}\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_1 \Rightarrow[\psi_1]\ \Phi_1 \qquad (\phi \mid \psi_2) \vdash \tau_2 : * \\ (\phi \mid \psi_2, \psi_1) \vdash [\cdot]\ \mathbf{coerce}(\tau_1, \tau_2) \Rightarrow \Phi_2\end{array}}{\phi; \Gamma \vdash e_1(e_2) \downarrow \tau_2 \Rightarrow[\psi_2]\ \exists(\psi_1).\Phi_1 \wedge \Phi_2}$$

Note that $(\exists\psi.\Phi_1 \wedge \Phi_2)[\theta]$ is $\exists\psi.\Phi_1[\theta] \wedge \Phi_2[\theta]$. Hence, there is an existential substitution θ_1 such that $\phi[\theta] \vdash \theta_1 \rhd \psi_1$ holds and $\phi[\theta_2] \models \Phi_1[\theta_2] \wedge \Phi_2[\theta_2]$ is derivable for $\theta_2 = \theta, \theta_1$. Hence, $\phi[\theta_2] \models \Phi_1[\theta_2]$ and $\phi[\theta_2] \models \Phi_2[\theta_2]$ are derivable. By induction hypothesis, $\phi[\theta_2]; \Gamma[\theta_2] \vdash e_1(e_2) \uparrow \tau_1[\theta_2] \Rightarrow e^*$ is derivable. Also $\phi[\theta_2] \vdash \mathbf{coerce}(\tau_1[\theta_2], \tau_2[\theta_2]) \Rightarrow E$ is derivable for some E by Theorem 5.2.5. This leads to the following.

$$\frac{\phi[\theta_2]; \Gamma[\theta_2] \vdash e_1(e_2) \uparrow \tau_1[\theta_2] \Rightarrow e^* \quad \phi[\theta_2] \vdash \mathbf{coerce}(\tau_1[\theta_2], \tau_2[\theta_2]) \Rightarrow E}{\phi[\theta_2]; \Gamma[\theta_2] \vdash e_1(e_2) \downarrow \tau_2[\theta_2] \Rightarrow E[e^*]}\ \textbf{(elrule-app-down)}$$

Note that $\phi[\theta] = \phi[\theta_2]$, $\Gamma[\theta] = \Gamma[\theta_2]$ and $\tau_2[\theta] = \tau_2[\theta_2]$, and $|e_1(e_2)| \cong |e^*| \cong |E[e^*]|$. Hence, we are done.

$$\mathcal{D} = \frac{\begin{array}{c}\phi; \Gamma \vdash e_1 \uparrow \Sigma(\vec{a} : \vec{\gamma}).\tau_1 \Rightarrow[\psi]\ \Phi_1 \\ \phi, \vec{a}^{\psi} : \gamma; \Gamma, x : \tau_1[\vec{a} \mapsto \vec{a}^{\psi}] \vdash e_2 \uparrow \tau_2 \Rightarrow[\psi]\ \Phi_2\end{array}}{\phi; \Gamma \vdash (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}) \uparrow \Sigma(\vec{a}^{\psi} : \vec{\gamma}).\tau_2 \Rightarrow[\psi]\ \Phi_1 \wedge \forall(\vec{a}^{\psi} : \vec{\gamma}).\Phi_2}$$

Then by assumption, the following is derivable.

$$\phi[\theta] \models (\Phi_1 \wedge \forall(\vec{a}^{\psi}).\Phi_2)[\theta]$$

$$\frac{\phi;\Gamma \vdash e \uparrow \tau \Rightarrow[\psi]\ \Phi \quad (\phi \mid \psi) \vdash \gamma : *_s}{\phi;\Gamma \vdash e \uparrow \tau \Rightarrow[\psi, A:\gamma]\ \Phi}\ \textbf{(constr-weak)}$$

$$\frac{\phi;\Gamma \vdash e \uparrow \Pi a:\gamma.\tau \Rightarrow[\psi]\ \Phi}{\phi;\Gamma \vdash e \uparrow \tau[a \mapsto A] \Rightarrow[\psi, A:\gamma]\ \Phi}\ \textbf{(constr-pi-elim)}$$

$$\frac{\phi, a^{\psi}:\gamma;\Gamma \vdash e[a \mapsto a^{\psi}] \downarrow \tau \Rightarrow[\psi]\ \Phi \quad (\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash \lambda a:\gamma.e \downarrow \Pi a:\gamma.\tau \Rightarrow[\psi]\ \forall(a^{\psi}:\gamma).\Phi}\ \textbf{(constr-pi-intro-1)}$$

$$\frac{\phi, a^{\psi}:\gamma;\Gamma \vdash e \downarrow \tau \Rightarrow[\psi]\ \Phi \quad (\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash e \downarrow \Pi a:\gamma.\tau \Rightarrow[\psi]\ \forall(a^{\psi}:\gamma).\Phi}\ \textbf{(constr-pi-intro-2)}$$

$$\frac{\phi;\Gamma \vdash e \downarrow \tau[a \mapsto A] \Rightarrow[\psi, A:\gamma]\ \Phi}{\phi;\Gamma \vdash e \downarrow \Sigma a:\gamma.\tau \Rightarrow[\psi]\ \exists A:\gamma.\Phi}\ \textbf{(constr-sig-intro)}$$

$$\frac{\Gamma(x) = \tau \quad (\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash x \uparrow \tau \Rightarrow[\psi]\ \top}\ \textbf{(constr-var-up)}$$

$$\frac{\begin{array}{c}\phi;\Gamma \vdash x \uparrow \tau_1 \Rightarrow[\psi_2, \psi_1]\ \top \quad (\phi \mid \psi_2) \vdash \tau_2 : * \\ (\phi \mid \psi_2) \vdash \Gamma[\mathbf{ctx}] \quad (\phi \mid \psi_2, \psi_1) \vdash [\cdot]\ \mathbf{coerce}(\tau_1, \tau_2) \Rightarrow \Phi\end{array}}{\phi;\Gamma \vdash x \downarrow \tau_2 \Rightarrow[\psi_2]\ \exists(\psi_1).\Phi}\ \textbf{(constr-var-down)}$$

$$\frac{\mathcal{S}(c) = \Pi(\vec{a}:\vec{\gamma}).\delta(i) \quad \phi \vdash \Gamma[\mathbf{ictx}]}{\phi;\Gamma \vdash c \uparrow \delta(i[\vec{a} \mapsto \vec{A}]) \Rightarrow[\vec{A}:\vec{\gamma}]\ \top}\ \textbf{(constr-cons-wo-up)}$$

$$\frac{\phi;\Gamma \vdash c \uparrow \delta(i_1) \Rightarrow[\psi_2, \psi_1]\ \top \quad (\phi \mid \psi_2) \vdash \delta(i_2) : *}{\phi;\Gamma \vdash c \downarrow \delta(i_2) \Rightarrow[\psi_2]\ \exists(\psi_1).\delta(i_1) \equiv \delta(i_2)}\ \textbf{(constr-cons-wo-down)}$$

$$\frac{\mathcal{S}(c) = \Pi(\vec{a}:\vec{\gamma}).\tau \to \delta(i) \quad \phi;\Gamma \vdash e \downarrow \tau[\vec{a} \mapsto \vec{A}] \Rightarrow[\psi, \vec{A}:\vec{\gamma}]\ \Phi}{\phi;\Gamma \vdash c(e) \uparrow \delta(i[\vec{a} \mapsto \vec{A}]) \Rightarrow[\psi, \vec{A}:\vec{\gamma}]\ \Phi}\ \textbf{(constr-cons-w-up)}$$

$$\frac{\begin{array}{c}\phi;\Gamma \vdash c(e) \uparrow \delta(i_1) \Rightarrow[\psi_2, \psi_1]\ \Phi \\ (\phi \mid \psi_2) \vdash \delta(i_2) : * \quad (\phi \mid \psi_2) \vdash \Gamma[\mathbf{ctx}]\end{array}}{\phi;\Gamma \vdash c(e) \downarrow \delta(i_2) \Rightarrow[\psi_2]\ \exists(\psi_1).\Phi \wedge \delta(i_1) \equiv \delta(i_2)}\ \textbf{(constr-cons-w-down)}$$

$$\frac{(\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash \langle\rangle \uparrow \mathbf{1} \Rightarrow[\psi]\ \top}\ \textbf{(constr-unit-up)}$$

$$\frac{(\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi;\Gamma \vdash \langle\rangle \downarrow \mathbf{1} \Rightarrow[\psi]\ \top}\ \textbf{(constr-unit-down)}$$

$$\frac{\phi;\Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow[\psi]\ \Phi_1 \quad \phi;\Gamma \vdash e_2 \uparrow \tau_2 \Rightarrow[\psi]\ \Phi_2}{\phi;\Gamma \vdash \langle e_1, e_2\rangle \uparrow \tau_1 * \tau_2 \Rightarrow[\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-prod-up)}$$

$$\frac{\phi;\Gamma \vdash e_1 \downarrow \tau_1 \Rightarrow[\psi]\ \Phi_1 \quad \phi;\Gamma \vdash e_2 \downarrow \tau_2 \Rightarrow[\psi]\ \Phi_2}{\phi;\Gamma \vdash \langle e_1, e_2\rangle \downarrow \tau_1 * \tau_2 \Rightarrow[\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-prod-down)}$$

Figure 5.5: The constraint generation rules for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ (I)

$$\frac{p \downarrow \tau_1 \Rightarrow (p^*; \phi_1; \Gamma_1) \quad \phi, \phi_1^{\psi}; \Gamma, \Gamma_1 \vdash e \downarrow \tau_2 \Rightarrow [\psi]\ \Phi \quad (\phi \mid \psi) \vdash \tau_1 \Rightarrow \tau_2 : * \quad (\phi \mid \psi) \vdash \Gamma[\mathbf{ctx}]}{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow [\psi]\ \forall(\phi_1^{\psi}).\Phi}\ \textbf{(constr-match)}$$

$$\frac{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow [\psi]\ \Phi_1 \quad \phi; \Gamma \vdash ms \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash (p \Rightarrow e \mid ms) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-matches)}$$

$$\frac{\phi; \Gamma \vdash e \uparrow \tau_1 \Rightarrow [\psi]\ \Phi_1 \quad \phi; \Gamma \vdash ms \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ ms) \downarrow \tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-case)}$$

$$\frac{\phi; \Gamma, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow [\psi]\ \Phi}{\phi; \Gamma \vdash (\mathbf{lam}\ x.e) \downarrow \tau_1 \to \tau_2 \Rightarrow [\psi]\ \Phi}\ \textbf{(constr-lam)}$$

$$\frac{\phi; \Gamma, x : \tau \vdash e \downarrow \tau_2 \Rightarrow [\psi]\ \Phi \quad \phi; \Gamma, x : \tau_1 \vdash x \downarrow \tau \Rightarrow [\psi]\ \Phi_1}{\phi; \Gamma \vdash (\mathbf{lam}\ x : \tau.e) \downarrow \tau_1 \to \tau_2 \Rightarrow [\psi]\ \Phi \wedge \Phi_1}\ \textbf{(constr-lam-anno)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \to \tau_2 \Rightarrow [\psi]\ \Phi_1 \quad \phi; \Gamma \vdash e_2 \downarrow \tau_1 \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \Phi_2}\ \textbf{(constr-app-up)}$$

$$\frac{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_1 \Rightarrow [\psi_2, \psi_1]\ \Phi_1 \quad (\phi \mid \psi_2) \vdash \tau_2 : * \quad (\phi \mid \psi_2) \vdash \Gamma[\mathbf{ctx}] \quad (\phi \mid \psi_2, \psi_1) \vdash [\cdot]\ \mathbf{coerce}(\tau_1, \tau_2) \Rightarrow \Phi_2}{\phi; \Gamma \vdash e_1(e_2) \downarrow \tau_2 \Rightarrow [\psi_2]\ \exists(\psi_1).\Phi_1 \wedge \Phi_2}\ \textbf{(constr-app-down)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \Sigma(\vec{a} : \vec{\gamma}).\tau_1 \Rightarrow [\psi]\ \Phi_1 \quad \phi, \vec{a}^{\psi} : \gamma; \Gamma, x : \tau_1[\vec{a} \mapsto \vec{a}^{\psi}] \vdash e_2 \uparrow \tau_2 \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}) \uparrow \Sigma(\vec{a}^{\psi} : \vec{\gamma}).\tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \forall(\vec{a}^{\psi} : \vec{\gamma}).\Phi_2}\ \textbf{(constr-let-up)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \Sigma(\vec{a} : \vec{\gamma}).\tau_1 \Rightarrow [\psi]\ \Phi_1 \quad \phi, \vec{a}^{\psi} : \gamma; \Gamma, x : \tau_1[\vec{a} \mapsto \vec{a}^{\psi}] \vdash e_2 \downarrow \tau_2 \Rightarrow [\psi]\ \Phi_2}{\phi; \Gamma \vdash (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}) \downarrow \tau_2 \Rightarrow [\psi]\ \Phi_1 \wedge \forall(\vec{a}^{\psi} : \vec{\gamma}).\Phi_2}\ \textbf{(constr-let-down)}$$

$$\frac{\phi; \Gamma, f : \tau \vdash u \downarrow \tau \Rightarrow [\psi]\ \Phi}{\phi; \Gamma \vdash (\mathbf{fix}\ f : \tau.u) \uparrow \tau \Rightarrow [\psi]\ \Phi}\ \textbf{(constr-fix-up)}$$

$$\frac{\phi; \Gamma, f : \tau \vdash u \downarrow \tau \Rightarrow [\psi]\ \Phi \quad \phi; \Gamma, x : \tau \vdash x \downarrow \tau_1 \Rightarrow [\psi]\ \Phi_1}{\phi; \Gamma \vdash (\mathbf{fix}\ f : \tau.u) \downarrow \tau_1 \Rightarrow [\psi]\ \Phi \wedge \Phi_1}\ \textbf{(constr-fix-down)}$$

$$\frac{\phi; \Gamma \vdash e \downarrow \tau \Rightarrow [\psi]\ \Phi}{\phi; \Gamma \vdash (e : \tau) \uparrow \tau \Rightarrow [\psi]\ \Phi}\ \textbf{(constr-anno-up)}$$

$$\frac{\phi; \Gamma \vdash (e : \tau) \uparrow \tau_1 \Rightarrow [\psi]\ \top \quad (\phi \mid \psi) \vdash \tau_2 : * \quad (\phi \mid \psi) \vdash [\cdot]\ \mathbf{coerce}(\tau_1, \tau_2) \Rightarrow \Phi}{\phi; \Gamma \vdash (e : \tau) \downarrow \tau_2 \Rightarrow [\psi]\ \Phi}\ \textbf{(constr-anno-down)}$$

Figure 5.6: The constraint generation rules for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ (II)

This implies that both $\phi[\theta] \vdash \Phi_1[\theta]$ and $\phi[\theta] \vdash \forall(\vec{a}^{\psi} : \vec{\gamma}[\theta]).\Phi_2[\theta]$ are derivable. By induction hypothesis, the following is derivable for some e_1^* such that $|e_1| \cong |e_1^*|$, where $\vec{\gamma}[\theta]$ is $\gamma_1[\theta], \ldots, \gamma_n[\theta]$ for $\vec{\gamma} = \gamma_1, \ldots, \gamma_n$.

$$\phi[\theta]; \Gamma[\theta] \vdash e_1 \uparrow \Sigma(\vec{a}^{\psi} : \vec{\gamma}[\theta]).\tau_1[\theta] \Rightarrow e_1^*$$

Notice that the derivability of $\phi[\theta] \vdash \forall(\vec{a}^{\psi} : \vec{\gamma}[\theta]).\Phi_2[\theta]$ implies that of $\phi[\theta], \vec{a}^{\psi} : \vec{\gamma}[\theta] \models \Phi_2[\theta]$. By induction hypothesis, we have the following derivable for some e_2^* such that $|e_2| \cong |e_2^*|$.

$$\phi[\theta], \vec{a}^{\psi}[\theta] : \gamma[\theta]; \Gamma[\theta], x : \tau_1[\vec{a} \mapsto \vec{a}^{\psi}][\theta] \vdash e_2 \uparrow \tau_2[\theta] \Rightarrow e_2^*$$

This yields the following derivation.

$$\frac{\begin{array}{c}\phi[\theta]; \Gamma[\theta] \vdash e_1 \uparrow \Sigma(\vec{a} : \vec{\gamma}[\theta]).\tau_1[\theta] \Rightarrow e_1^* \\ \phi[\theta], \vec{a}^{\psi} : \vec{\gamma}[\theta]; \Gamma, x : \tau_1[\vec{a} \mapsto \vec{a}^{\psi}][\theta] \vdash e_2 \uparrow \tau_2[\theta] \Rightarrow e_2^*\end{array}}{\phi[\theta]; \Gamma[\theta] \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \uparrow \Sigma(\vec{a}^{\psi} : \vec{\gamma}[\theta]).\tau_2[\theta] \Rightarrow \mathbf{let}\ \langle \vec{a} \mid x \rangle = e_1^*\ \mathbf{in}\ \langle \vec{a} \mid e_2^* \rangle\ \mathbf{end}}$$

So the case wraps up.

$$\mathcal{D} = \frac{\phi; \Gamma, f : \tau \vdash u \downarrow \tau \Rightarrow [\psi]\ \Phi}{\phi; \Gamma \vdash (\mathbf{fix}\ f : \tau.u) \uparrow \tau \Rightarrow [\psi]\ \Phi}$$

By induction hypothesis, $\phi[\theta]; \Gamma[\theta], f : \tau[\theta] \vdash u \downarrow \tau[\theta] \Rightarrow u^*$ is derivable for some u^* such that $|u| \cong |u^*|$, and this leads to the following.

$$\frac{\phi[\theta]; \Gamma[\theta], f : \tau[\theta] \vdash u \downarrow \tau[\theta] \Rightarrow u^*}{\phi[\theta]; \Gamma[\theta] \vdash (\mathbf{fix}\ f : \tau.u) \downarrow \tau[\theta] \Rightarrow (\mathbf{fix}\ f : \tau[\theta].u^*)}\ \textbf{(elab-fix)}$$

Note that $|\mathbf{fix}\ f : \tau.u| = \mathbf{fix}\ f.|u| \cong \mathbf{fix}\ f.|u^*| = |\mathbf{fix}\ f : \tau[\theta].u^*|$, and we are done.

All other cases can be treated in a similar manner. ∎

Given a program, that is, a closed expression e in DML(C), we can use the constraint generation rules to derive a judgement of form $\cdot; \cdot \vdash e \uparrow \tau \Rightarrow [\psi]\ \Phi$ for some ψ, τ and Φ. Assume that this process succeeds. By Theorem 5.2.7 and Theorem 5.2.6, we know that e can be elaborated into an expression e^* in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ such that $|e| \cong |e^*|$ if $\cdot \models \exists(\psi).\Phi$ can be derived. In this sense, we say that type-checking in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ has been reduced to constraint satisfaction.

5.3 Summary

In this section, $\mathrm{ML}_0^{\Pi}(C)$ is extended with existential dependent types, leading to the language $\mathrm{ML}_0^{\Pi,\Sigma}(C)$. This extension seems to be indispensable in practical programming. For instance, existential dependent types are used in all the examples presented in Appendix A. Like $\mathrm{ML}_0^{\Pi}(C)$, $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ enjoys the type preservation property and its operational semantics can be simulated by that of ML_0 (Theorem 5.1.3 and Theorem 5.1.5). Consequently, $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ is a conservative extension of ML_0.

$\mathrm{ML}_0^{\Pi,\Sigma}(C)$ is an explicitly typed internal programming language, and therefore, a practical elaboration from the external language DML(C) to $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ is crucial if $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ is intended for general purpose programming. As for $\mathrm{ML}_0^{\Pi}(C)$, we achieve this by presenting a set of elaboration

rules and then a set of constraint generation rules. The correctness of these rules are justified by Theorem 5.2.6 and Theorem 5.2.7, respectively.

However, there is a significant issue which involves whether a variant of A-normal transform should be performed on programs in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ before elaboration. This transform enables us to elaborate a very common form of expressions which could otherwise not be elaborated, but it also prevents us from elaborating a less common form of expressions. A serious disadvantage of performing the transform is that it can complicate reporting comprehensible error messages during elaboration since the programmer may have to understand how the programs are transformed. An alternative is to allow the programmer to control the transform with the help of some sugared syntax. This has yet to be settled in practice. We point out that the transform is performed in our current prototype implementation.

This chapter has further solidified the justification for the practicality of our approach to extending programming languages with dependent types. The theoretic core of this thesis consists of Chapter 4 and Chapter 5. We are now ready to study the issues on extending $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ with let-polymorphism, effects such as references and exception mechanism, aiming for adding dependent types to the entire core of ML.

Chapter 6

Polymorphism

Polymorphism is the ability to abstract expressions over types. Such expressions with universally quantified types can then assume different types when the universally quantified type variables are instantiated differently. Therefore, polymorphism provides an approach to promoting certain form of *code reuse*, which is an important issue in software engineering. In this chapter, we extend the language ML_0 to $\mathrm{ML}_0^{\forall}$ with ML-style of let-polymorphism and prove some relevant results. We then extend the language $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ to $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$, combining dependent types with let-polymorphism. The relation between $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ and $\mathrm{ML}_0^{\forall}$ is established, parallel to that between $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ and ML_0.

Although the development of dependent types is largely orthogonal to polymorphism, it is nonetheless noticeably involved to combine these two features together. Also there are some practical issues showing up when elaboration is concerned, which must be addressed carefully.

6.1 Extending ML_0 to $\mathrm{ML}_0^{\forall}$

In this section, we extend ML_0 with ML-style of let-polymorphism, yielding a polymorphic programming language $\mathrm{ML}_0^{\forall}$. The syntax of $\mathrm{ML}_0^{\forall}$ enriches that of ML_0 with the following.

type variables	α		
type constructors	δ		
types	τ	::=	$\cdots \mid \alpha \mid (\tau_1, \ldots, \tau_n)\delta$
type schemes	σ	::=	$\tau \mid \forall\alpha.\sigma$
patterns	p	::=	$\cdots \mid c(\vec{\alpha}) \mid c(\vec{\alpha})(p)$
expressions	e	::=	$\cdots \mid c(\vec{\tau}) \mid c(\vec{\tau})(e) \mid x(\vec{\tau}) \mid \Lambda\alpha.e$
value forms	u	::=	$\cdots \mid c(\vec{\tau}) \mid c(\vec{\tau})(u)$
values	v	::=	$\cdots \mid x(\vec{\tau}) \mid c(\vec{\tau}) \mid c(\vec{\tau})(v) \mid \Lambda\alpha.v$
type var contexts	Δ	::=	$\cdot \mid \Delta, \alpha$
signature	$\mathcal{S}$	::=	$\cdots \mid \mathcal{S}, \delta : * \to \cdots \to * \mid c : \forall\vec{\alpha}.(\vec{\alpha})\delta$
substitutions	θ	::=	$\cdots \mid \theta[\alpha \mapsto \tau]$

We use $\vec{\tau}$ for a (possibly empty) sequence of types $\tau_1, \ldots, \tau_m$. In addition, given $\vec{\tau} = \tau_1, \ldots, \tau_m$, $(\vec{\tau})\delta$, $c(\vec{\tau})$ and $x(\vec{\tau})$ are abbreviations for $(\tau_1, \ldots, \tau_m)\delta$, $c(\tau_1)\ldots(\tau_m)$ and $x(\tau_1)\ldots(\tau_m)$, respectively. We may also write $\forall\vec{\alpha}.\sigma$ for $\forall\alpha_1 \ldots \forall\alpha_n.\sigma$, where $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$.

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha : *}\ \textbf{(type-var)} \qquad \frac{}{\Delta \vdash \beta : *}\ \textbf{(type-base)}$$

$$\frac{\mathcal{S}(\delta) = * \to \cdots \to * \to * \quad \Delta \vdash \tau_1 : * \quad \cdots \quad \Delta \vdash \tau_m : *}{\Delta \vdash (\tau_1, \ldots, \tau_m)\delta : *}\ \textbf{(type-datatype)}$$

$$\frac{}{\Delta \vdash \mathbf{1} : *}\ \textbf{(type-unit)} \qquad \frac{\Delta \vdash \tau_1 : * \quad \Delta \vdash \tau_2 : *}{\Delta \vdash \tau_1 * \tau_2 : *}\ \textbf{(type-prod)}$$

$$\frac{\Delta \vdash \tau_1 : * \quad \Delta \vdash \tau_2 : *}{\Delta \vdash \tau_1 \to \tau_2 : *}\ \textbf{(type-fun)} \qquad \frac{\Delta, \alpha \vdash \sigma}{\Delta \vdash \forall \alpha.\sigma}\ \textbf{(type-poly)}$$

Figure 6.1: Type formation rules for $\mathrm{ML}_0^\forall$

The *types* in $\mathrm{ML}_0^\forall$ are basically those defined in ML_0 but they may contain type variables in this setting. A *type scheme* σ must be of the form $\forall\alpha_1 \cdots \forall\alpha_n.\tau$ and σ is τ if $n = 0$.

Notice that the treatment of patterns is non-standard. In ML, the type variables do not occur in patterns. We take this approach since it naturally follows the one we adopted for handling universal dependent types in Section 4.1. However, the difference is largely cosmetic.

6.1.1 Static Semantics

The rules for forming legal types in $\mathrm{ML}_0^\forall$ are presented in Figure 6.1. Clearly, if $\Delta \vdash \sigma : *$ is derivable, then all free type variables in σ are declared in Δ.

We present the typing rules for pattern matching in Figure 6.2. We then list all the type inference rules for $\mathrm{ML}_0^\forall$ in Figure 6.3. Of course, we require that there be no free occurrences of α in $\Gamma(x)$ for every $x \in \mathbf{dom}(\Gamma)$ when the rule **(ty-poly-intro)** is introduced. The rules closely resemble those for ML_0 except that we now use a type variable context Δ in every judgement to keep track of free type variables. The let-polymorphism is enforced because **(ty-let)** is the only rule which can eliminate from (ordinary) variable context the variables whose types contain $\forall$ quantifiers.

Given a substitution θ, we define

$$x(\vec{\tau})[\theta] = v[\vec{\alpha} \mapsto \vec{\tau}]$$

if $\theta(x) = \Lambda\vec{\alpha}.v$. Notice that $\vec{\alpha}$ and $\vec{\tau}$ must have the same length. Otherwise, $x(\vec{\tau})[\theta]$ is undefined. This definition obviates the need for introducing expressions of form $e(\vec{\tau})$ for non-variable expressions e, which cannot occur in $\mathrm{ML}_0^\forall$ since only let-polymorphism is allowed.

Lemma 6.1.1 *If $\Delta \vdash \tau_i : *$ are derivable for $i = 1, \ldots, n$ and $\Delta, \vec{\alpha}; \Gamma \vdash e : \sigma$ is also derivable in $\mathrm{ML}_0^\forall$, then $\Delta; \Gamma[\vec{\alpha} \mapsto \vec{\tau}] \vdash e[\vec{\alpha} \mapsto \vec{\tau}] : \sigma[\vec{\alpha} \mapsto \vec{\tau}]$ is derivable, where $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$ and $\vec{\tau} = \tau_1, \ldots, \tau_n$.*

Proof This simply follows from a structural induction on the derivation of $\Delta, \vec{\alpha}; \Gamma \vdash e : \sigma$. ∎

$$\frac{}{x \downarrow \tau \rhd (x : \tau)}\ (\textbf{pat-var})$$

$$\frac{}{\langle\rangle \downarrow \mathbf{1} \rhd \cdot}\ (\textbf{pat-unit})$$

$$\frac{p_1 \downarrow \tau_1 \rhd \Gamma_1 \quad p_2 \downarrow \tau_2 \rhd \Gamma_2}{\langle p_1, p_2\rangle \downarrow \tau_1 * \tau_2 \rhd \Gamma_1, \Gamma_2}\ (\textbf{pat-prod})$$

$$\frac{\mathcal{S}(c) = \forall\alpha_1 \ldots \forall\alpha_m.(\alpha_1, \ldots, \alpha_m)\delta}{c(\alpha_1)\ldots(\alpha_m) \downarrow (\tau_1, \ldots, \tau_m)\delta \rhd \cdot}\ (\textbf{pat-cons-wo})$$

$$\frac{\mathcal{S}(c) = \forall\alpha_1 \ldots \forall\alpha_m.(\tau \to (\alpha_1, \ldots, \alpha_m)\delta) \quad p \downarrow \tau[\alpha_1, \ldots, \alpha_m \mapsto \tau_1, \ldots, \tau_m] \rhd \Gamma}{c(\alpha_1)\ldots(\alpha_m)(p) \downarrow (\tau_1, \ldots, \tau_m)\delta \rhd \Gamma}\ (\textbf{pat-cons-w})$$

Figure 6.2: Typing rules for pattern matching in $\mathrm{ML}_0^\forall$

Lemma 6.1.2 *If both* $\Delta;\Gamma \vdash v : \sigma_1$ *and* $\Delta;\Gamma, x : \sigma_1 \vdash e : \sigma$ *are derivable, then* $\Delta;\Gamma \vdash e[x \mapsto v] : \sigma$ *is also derivable.*

Proof This simply follows from a structural induction on the derivation of $\Delta;\Gamma, x : \sigma_1 \vdash e : \sigma$. ■

6.1.2 Dynamic Semantics

The evaluation rules for formulating the natural semantics of $\mathrm{ML}_0^\forall$ are those for ML_0 plus the following rule **(ev-poly)**, which is needed for evaluation under Λ.

$$\frac{e \hookrightarrow_0 v}{\Lambda\alpha.e \hookrightarrow_0 \Lambda\alpha.v}\ (\textbf{ev-poly})$$

Note that we do not need a rule for evaluating $e(\tau)$ because this expression can never occur in $\mathrm{ML}_0^\forall$.

As usual, the type preservation theorem holds in $\mathrm{ML}_0^\forall$.

Theorem 6.1.3 *(Type preservation for* $\mathrm{ML}_0^\forall$*) If* $e \hookrightarrow_0 v$ *and* $\Delta;\Gamma \vdash e : \sigma$ *are derivable, then* $\Delta;\Gamma \vdash v : \sigma$ *is also derivable.*

Proof This proof proceeds by a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_0 v$, parallel to that of Theorem 2.2.7. We present several cases.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_0 v_1 \quad e_2[x \mapsto v_1] \hookrightarrow_0 v}{(\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end}) \hookrightarrow_0 v}$ Then we also have the following derivation.

$$\frac{\Delta;\Gamma \vdash e_1 : \sigma_1 \quad \Delta;\Gamma, x_1 : \sigma_1 \vdash e_2 : \tau}{\Delta;\Gamma \vdash \textbf{let } x_1 = e_1 \textbf{ in } e_2 \textbf{ end} : \tau}\ (\textbf{ty-let})$$

$$\frac{\Delta \vdash \tau_1 : * \quad \cdots \quad \Delta \vdash \tau_n : * \quad \Gamma(x) = \forall\alpha_1 \cdots \forall\alpha_n.\tau}{\Delta;\Gamma \vdash x(\tau_1)\ldots(\tau_n) : \tau[\alpha_1,\ldots,\alpha_n \mapsto \tau_1,\ldots,\tau_n]} \textbf{ (ty-poly-var)}$$

$$\frac{\mathcal{S}(c) = \forall\alpha_1 \cdots \forall\alpha_n.(\alpha_1,\ldots,\alpha_n)\delta \quad \Delta \vdash \tau_1 : * \quad \cdots \quad \Delta \vdash \tau_n : *}{\Delta;\Gamma \vdash c(\tau_1,\ldots,\tau_n) : (\tau_1,\ldots,\tau_n)\delta} \textbf{ (ty-poly-cons-wo)}$$

$$\frac{\begin{array}{c}\mathcal{S}(c) = \forall\alpha_1 \cdots \forall\alpha_n.\tau \to \delta \\ \Delta \vdash \tau_1 : * \quad \cdots \quad \Delta \vdash \tau_n : * \quad \Delta;\Gamma \vdash e : \tau[\alpha_1,\ldots,\alpha_n \mapsto \tau_1,\ldots,\tau_n]\end{array}}{\Delta;\Gamma \vdash c(\tau_1,\ldots,\tau_n)(e) : (\tau_1,\ldots,\tau_n)\delta} \textbf{ (ty-poly-cons-w)}$$

$$\frac{}{\Delta;\Gamma \vdash \langle\rangle : \mathbf{1}} \textbf{ (ty-unit)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \quad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 * \tau_2} \textbf{ (ty-prod)}$$

$$\frac{\Delta \vdash \tau_1 : * \quad p \downarrow \tau_1 \rhd \Gamma' \quad \Delta;\Gamma,\Gamma' \vdash e : \tau_2}{\Delta;\Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2} \textbf{ (ty-match)}$$

$$\frac{\Delta;\Gamma \vdash (p \Rightarrow e) : \tau_1 \Rightarrow \tau_2 \quad \Delta;\Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\Delta;\Gamma \vdash (p \Rightarrow e \mid ms) : \tau_1 \Rightarrow \tau_2} \textbf{ (ty-matches)}$$

$$\frac{\Delta;\Gamma \vdash e : \tau_1 \quad \Delta;\Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\Delta;\Gamma \vdash (\textbf{case } e \textbf{ of } ms) : \tau_2} \textbf{ (ty-case)}$$

$$\frac{\Delta;\Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta;\Gamma \vdash (\textbf{lam } x : \tau_1.e) : \tau_1 \to \tau_2} \textbf{ (ty-lam)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta;\Gamma \vdash e_2 : \tau_1}{\Delta;\Gamma \vdash e_1(e_2) : \tau_2} \textbf{ (ty-app)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \sigma \quad \Delta;\Gamma, x : \sigma \vdash e_2 : \tau}{\Delta;\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau} \textbf{ (ty-let)}$$

$$\frac{\Delta;\Gamma, f : \tau \vdash u : \tau}{\Delta;\Gamma \vdash (\textbf{fix } f : \tau.u) : \tau} \textbf{ (ty-fix)}$$

$$\frac{\Delta,\alpha;\Gamma \vdash e : \sigma}{\Delta;\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\sigma} \textbf{ (ty-poly-intro)}$$

Figure 6.3: Typing Rules for $\mathrm{ML}_0^\forall$

By induction hypothesis, $\Delta;\Gamma \vdash v_1 : \sigma_1$ is derivable. Therefore, $\Delta;\Gamma \vdash e_2[x_1 \mapsto v_1] : \tau$ is derivable by Lemma 6.1.2. This leads to a derivation of $\Delta;\Gamma \vdash v : \sigma_1$ by induction hypothesis.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_0 v_1}{\Lambda\alpha.e_1 \hookrightarrow_0 \Lambda\alpha.v_1}$ Then we also have the following derivation.

$$\frac{\Delta, \alpha;\Gamma \vdash e_1 : \sigma_1}{\Delta;\Gamma \vdash \Lambda\alpha.e_1 : \forall\alpha.\sigma_1}\ \textbf{(ty-poly-intro)}$$

By induction hypothesis, $\cdot;\Delta,\alpha;\Gamma \vdash v_1 : \sigma_1$ is derivable. This readily leads to a derivation of $\Delta;\Gamma \vdash \Lambda\alpha.v_1 : \forall\alpha.\sigma_1$.

■

As in ML_0, types play no rôle in program evaluation. Extending the definition of the type erasure function $|\cdot|$ as follows, we capture the indifference of types to evaluation in $\mathrm{ML}_0^{\forall}$ through Theorem 6.1.4.

$$|x(\vec{\tau})| = x \qquad |c(\vec{\alpha})| = c \qquad |c(\vec{\alpha})(e)| = c(|e|) \qquad |\Lambda\alpha.e| = |e|$$

Theorem 6.1.4 *Given an expression e in $\mathrm{ML}_0^{\forall}$, we have the following.*

1. *If $e \hookrightarrow_0 v$ is derivable in $\mathrm{ML}_0^{\forall}$, then $|e| \hookrightarrow_0 |v|$ is derivable in $\lambda_{\mathrm{val}}^{\mathrm{pat}}$.*
2. *If $\Delta;\Gamma \vdash e : \sigma$ is derivable in $\mathrm{ML}_0^{\forall}$ and $|e| \hookrightarrow_0 v_0$ derivable in $\lambda_{\mathrm{val}}^{\mathrm{pat}}$, then $e \hookrightarrow_0 v$ is derivable in $\mathrm{ML}_0^{\forall}$ for some v such that $|v| = v_0$.*

Proof (1) and (2) follow from a structural induction on the derivations of $e \hookrightarrow_0 v$ and $|e| \hookrightarrow_0 v_0$, respectively. ■

We have now finished setting up the machinery for combining dependent types with the ML style of let-polymorphism.

6.2 Extending $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ to $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$

The language $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ is extended to the language $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ as follows. We use $\vec{i}$ for a (possibly empty) sequence of type indices. In addition, given $\vec{\tau} = \tau_1, \ldots, \tau_m$ and $\vec{i} = i_1, \ldots, i_n$, $c(\vec{\alpha})[\vec{i}]$ is an abbreviation for $c(\tau_1)\ldots(\tau_m)[i_1]\ldots[i_n]$.

type variables	α		
types	τ	::=	$\cdots \mid \alpha$
type schemes	σ	::=	$\tau \mid \forall\alpha.\sigma$
patterns	p	::=	$\cdots \mid c(\vec{\alpha})[\vec{i}] \mid c(\vec{\alpha})[\vec{i}](p)$
expressions	e	::=	$\cdots \mid c(\vec{\tau})[\vec{i}] \mid c(\vec{\tau})[\vec{i}](e) \mid x(\vec{\tau}) \mid \Lambda\alpha.e$
value forms	u	::=	$\cdots \mid c(\vec{\tau})[\vec{i}] \mid c(\vec{\tau})[\vec{i}](u)$
values	v	::=	$\cdots \mid x(\vec{\tau}) \mid c(\vec{\tau})[\vec{i}] \mid c(\vec{\tau})[\vec{i}](v) \mid \Lambda\alpha.v$
signature	$\mathcal{S}$	::=	$\cdots \mid \mathcal{S}, \delta : * \to \cdots \to * \to \gamma \to * \mid \mathcal{S}, c : \forall\vec{\alpha}.\forall\vec{a} : \vec{\gamma}.(\vec{\alpha})\delta(i)$
substitutions	θ	::=	$\cdots \mid \theta[\alpha \mapsto \tau]$

$$\frac{\alpha \in \Delta \quad \vdash \phi[\mathbf{ictx}]}{\phi; \Delta \vdash \alpha : *}\ \textbf{(type-var)} \qquad \frac{\vdash \phi[\mathbf{ictx}]}{\phi; \Delta \vdash \mathbf{1} : *}\ \textbf{(type-unit)}$$

$$\frac{\mathcal{S}(\delta) = * \to \cdots \to * \to \gamma \to * \quad \phi; \Delta \vdash \tau_1 : * \quad \cdots \quad \phi; \Delta \vdash \tau_m : * \quad \phi \vdash i : \gamma}{\phi; \Delta \vdash (\tau_1, \ldots, \tau_m)\delta(i) : *}\ \textbf{(type-datatype)}$$

$$\frac{\phi; \Delta \vdash \tau_1 : * \quad \phi; \Delta \vdash \tau_2 : *}{\phi; \Delta \vdash \tau_1 * \tau_2 : *}\ \textbf{(type-prod)} \qquad \frac{\phi; \Delta \vdash \tau_1 : * \quad \phi; \Delta \vdash \tau_2 : *}{\phi; \Delta \vdash \tau_1 \to \tau_2 : *}\ \textbf{(type-fun)}$$

$$\frac{\phi, a : \gamma; \Delta \vdash \tau : *}{\phi; \Delta \vdash \Pi a : \gamma.\tau : *}\ \textbf{(type-pi)} \qquad \frac{\phi, a : \gamma; \Delta \vdash \tau : *}{\phi; \Delta \vdash \Sigma a : \gamma.\tau : *}\ \textbf{(type-sig)} \qquad \frac{\cdot; \Delta, \alpha \vdash \sigma : *}{\cdot; \Delta \vdash \forall \alpha.\sigma : *}\ \textbf{(type-poly)}$$

Figure 6.4: Type formation rules for $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$

The *types* in $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ are basically the types defined in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ but they may contain type variables in this setting. A type scheme σ must then be of the form $\forall\alpha_1 \cdots \forall\alpha_n.\tau$ and σ is τ if $n = 0$. Notice that this disallows $\forall$ quantifiers to occur in the scope of a Π or Σ quantifier. For instance, the following is an illegal type.

$$\Pi n : nat.\forall\alpha.(\alpha)list(n) \to (\alpha)list(n)$$

This restriction is also necessary for the two-phase type-checking algorithm we introduce shortly.

6.2.1 Static Semantics

We present the rules for forming legal types in Figure 6.4.

Also we need the following additional rules for handling the type congruence relation.

$$\frac{}{\phi \models \alpha \equiv \alpha}$$

$$\frac{\phi \models \tau_1 \equiv \tau_1' \quad \cdots \quad \phi \models \tau_n \equiv \tau_n' \quad \phi \models i \doteq i'}{\phi \models (\tau_1, \ldots, \tau_n)\delta(i) \equiv (\tau_1', \ldots, \tau_n')\delta(i')}$$

We present the typing rules for pattern matching in Figure 6.5. Notice that in the rule **(pat-cons-w)**, the type of a constructor c associated with a datatype constructor δ is always of form

$$\forall\alpha_1 \ldots \forall\alpha_m.\Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.(\tau \to (\alpha_1, \ldots, \alpha_m)\delta(i))$$

For instance, it is *not* allowed in SML to declare a datatype as follows.

```
datatype bottom = Bottom of 'a
```

because this declaration assigns *Bottom* the type $\forall\alpha.\alpha \to bottom$, which clearly is not of the required form $\forall\alpha.\tau \to (\alpha)bottom$.

The following proposition is parallel to Proposition 4.1.8 for $\mathrm{ML}_0^{\Pi}(C)$.

Proposition 6.2.1 *We have the following.*

$$\frac{}{x \downarrow \tau \rhd (\cdot; x : \tau)}\ \textbf{(pat-var)}$$

$$\frac{}{\langle\rangle \downarrow \mathbf{1} \rhd (\cdot;\cdot)}\ \textbf{(pat-unit)}$$

$$\frac{p_1 \downarrow \tau_1 \rhd (\phi_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \rhd (\phi_2; \Gamma_2)}{\langle p_1, p_2\rangle \downarrow \tau_1 * \tau_2 \rhd (\phi_1, \phi_2; \Gamma_1, \Gamma_2)}\ \textbf{(pat-prod)}$$

$$\frac{\mathcal{S}(c) = \forall\alpha_1 \ldots \forall\alpha_m.\Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.(\alpha_1, \ldots, \alpha_m)\delta(i)}{c(\alpha_1)\ldots(\alpha_m)[a_1]\ldots[a_n] \downarrow (\tau_1, \ldots, \tau_m)\delta(j) \rhd (a_1 : \gamma_1, \ldots, a_n : \gamma; \cdot)}\ \textbf{(pat-cons-wo)}$$

$$\frac{\begin{array}{c}\mathcal{S}(c) = \forall\alpha_1 \ldots \forall\alpha_m.\Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.(\tau \to (\alpha_1, \ldots, \alpha_m)\delta(i)) \\ p \downarrow \tau[\alpha_1, \ldots, \alpha_m \mapsto \tau_1, \ldots, \tau_m] \rhd (\phi; \Gamma)\end{array}}{c(\alpha_1)\ldots(\alpha_m)[a_1]\ldots[a_n](p) \downarrow (\tau_1, \ldots, \tau_m)\delta(j) \rhd (a_1 : \gamma_1, \ldots, a_n : \gamma_n, i \doteq j, \phi; \Gamma)}\ \textbf{(pat-cons-w)}$$

Figure 6.5: Typing rules for patterns

1. $\|\tau[\theta]\| = \|\tau\|$ *and* $\|e[\theta]\| = \|e\|[\|\theta\|]$.
2. $\|u\|$ *is a value form in* $\mathrm{ML}_0^{\forall}$ *if* u *is a value form in* $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$.
3. $\|v\|$ *is a value in* $\mathrm{ML}_0^{\forall}$ *if* v *is a value in* $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$.
4. *If* $p \downarrow \tau \rhd (\phi'; \Gamma')$ *is derivable. then* $\|p\| \downarrow \|\tau\| \rhd \|\Gamma'\|$ *is derivable.*
5. *If* $\mathbf{match}(p, v) \Longrightarrow \theta$ *is derivable in* $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$, *then* $\mathbf{match}(\|p\|, \|v\|) \Longrightarrow \|\theta\|$ *is derivable in* $\mathrm{ML}_0^{\forall}$.
6. *Given* v, p *in* $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ *such that* $\phi; \Gamma \vdash v : \tau$ *and* $p \downarrow \tau \Longrightarrow (\phi'; \Gamma')$ *are derivable. If* $\mathbf{match}(\|p\|, \|v\|) \Longrightarrow \theta_0$ *is derivable, then* $\mathbf{match}(p, v) \Longrightarrow \theta$ *is derivable for some* θ *and* $\|\theta\| = \theta_0$.
7. *If* $\phi \vdash \tau_1 \equiv \tau_2$ *is derivable, then* $\|\tau_1\| = \|\tau_2\|$.

Proof Please refer to the proof of Proposition 4.1.8. ∎

We list all the type inference rules for $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ in Figure 6.6. The rules resemble those for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ very closely except that we now use a type variable context Δ in a judgement to keep track of free type variables. The let-polymorphism is enforced because **(ty-let)** is the only rule which can eliminate from (ordinary) variable context a variable whose type begins with a $\forall$ quantifier.

Example 6.2.2 *We present an example of type derivation in* $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$. *Let* $\mathcal{D}_1$ *be the following derivation,*

$$\cfrac{\cfrac{\cfrac{}{\cdot; \alpha; x : \alpha \vdash x : \alpha}\ \textbf{(ty-poly-var)}}{\cdot; \alpha; \cdot \vdash \lambda x : \alpha.x : \alpha \to \alpha}\ \textbf{(ty-lam)}}{\cdot; \cdot; \cdot \vdash (\Lambda\alpha.\lambda x : \alpha.x) : \forall\alpha.\alpha \to \alpha}\ \textbf{(ty-poly-intro)}$$

$$\frac{\phi; \Delta; \Gamma \vdash e : \tau_1 \quad \phi \vdash \tau_1 \equiv \tau_2}{\phi; \Delta; \Gamma \vdash e : \tau_2} \textbf{ (ty-eq)}$$

$$\frac{\phi; \Delta \vdash \tau_1 : * \quad \cdots \quad \phi; \Delta \vdash \tau_n : * \quad \Gamma(x) = \forall \alpha_1 \cdots \forall \alpha_n . \tau}{\phi; \Delta; \Gamma \vdash x(\tau_1) \ldots (\tau_n) : \tau[\alpha_1, \ldots, \alpha_n \mapsto \tau_1, \ldots, \tau_n]} \textbf{ (ty-poly-var)}$$

$$\frac{\phi; \Delta \vdash \tau_1 : * \quad \cdots \quad \phi; \Delta \vdash \tau_n : * \quad \mathcal{S}(c) = \forall \alpha_1 \cdots \forall \alpha_n . \tau}{\phi; \Delta; \Gamma \vdash c(\tau_1) \cdots (\tau_n) : \tau[\alpha_1, \ldots, \alpha_n \mapsto \tau_1, \ldots, \tau_n]} \textbf{ (ty-poly-cons)}$$

$$\frac{}{\phi; \Delta; \Gamma \vdash \langle\rangle : \mathbf{1}} \textbf{ (ty-unit)}$$

$$\frac{\phi; \Delta; \Gamma \vdash e_1 : \tau_1 \quad \phi; \Delta; \Gamma \vdash e_2 : \tau_2}{\phi; \Delta; \Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 * \tau_2} \textbf{ (ty-prod)}$$

$$\frac{p \downarrow \tau_1 \rhd (\phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e : \tau_2}{\phi; \Delta; \Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2} \textbf{ (ty-match)}$$

$$\frac{\phi; \Delta; \Gamma \vdash (p \Rightarrow e) : \tau_1 \Rightarrow \tau_2 \quad \phi; \Delta; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\phi; \Delta; \Gamma \vdash (p \Rightarrow e \mid ms) : \tau_1 \Rightarrow \tau_2} \textbf{ (ty-matches)}$$

$$\frac{\phi; \Delta; \Gamma \vdash e : \tau_1 \quad \phi; \Delta; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\phi; \Delta; \Gamma \vdash (\textbf{case } e \textbf{ of } ms) : \tau_2} \textbf{ (ty-case)}$$

$$\frac{\phi, a : \gamma; \Delta; \Gamma \vdash e : \tau}{\phi; \Delta; \Gamma \vdash (\lambda a : \gamma . e) : (\Pi a : \gamma . \tau)} \textbf{ (ty-ilam)}$$

$$\frac{\phi; \Delta; \Gamma \vdash e : \Pi a : \gamma . \tau \quad \phi \vdash i : \gamma}{\phi; \Delta; \Gamma \vdash e[i] : \tau[a \mapsto i]} \textbf{ (ty-iapp)}$$

$$\frac{\phi; \Delta; \Gamma \vdash e : \tau[a \mapsto i] \quad \phi \vdash i : \gamma}{\phi; \Delta; \Gamma \vdash \langle i \mid e \rangle : (\Sigma a : \gamma . \tau)} \textbf{ (ty-sig-intro)}$$

$$\frac{\phi; \Delta; \Gamma \vdash e_1 : \Sigma a : \gamma . \tau_1 \quad \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \Delta; \Gamma \vdash \textbf{let } \langle a \mid x \rangle = e_1 \textbf{ in } e_2 \textbf{ end} : \tau_2} \textbf{ (ty-sig-elim)}$$

$$\frac{\phi; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\phi; \Delta; \Gamma \vdash (\textbf{lam } x : \tau_1 . e) : \tau_1 \to \tau_2} \textbf{ (ty-lam)}$$

$$\frac{\phi; \Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \phi; \Delta; \Gamma \vdash e_2 : \tau_1}{\phi; \Delta; \Gamma \vdash e_1(e_2) : \tau_2} \textbf{ (ty-app)}$$

$$\frac{\phi; \Delta; \Gamma \vdash e_1 : \sigma \quad \phi, \phi_1; \Delta; \Gamma, x : \sigma \vdash e_2 : \tau}{\phi, \phi_1; \Delta; \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau} \textbf{ (ty-let)}$$

$$\frac{\phi; \Delta; \Gamma, f : \tau \vdash u : \tau}{\phi; \Delta; \Gamma \vdash (\textbf{fix } f : \tau . u) : \tau} \textbf{ (ty-fix)}$$

$$\frac{\cdot; \Delta, \alpha; \Gamma \vdash e : \sigma}{\cdot; \Delta; \Gamma \vdash \Lambda \alpha . e : \forall \alpha . \sigma} \textbf{ (ty-poly-intro)}$$

Figure 6.6: Typing Rules for $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$

and $\mathcal{D}_2$ be the following one,

$$\dfrac{\dfrac{}{\cdot;\cdot;f:\forall\alpha.\alpha\to\alpha\vdash\overline{0}:\mathtt{int}}\quad\dfrac{\dfrac{}{\cdot;\cdot\vdash\mathtt{int}:*}}{\cdot;\cdot;f:\forall\alpha.\alpha\to\alpha\vdash f(\mathtt{int}):\mathtt{int}\to\mathtt{int}}\ \textbf{(ty-poly-var)}}{\cdot;\cdot;f:\forall\alpha.\alpha\to\alpha\vdash f(\mathtt{int})(\overline{0}):\mathtt{int}}\ \textbf{(ty-app)}$$

and $\mathcal{D}_3$ be the following one.

$$\dfrac{\dfrac{}{\cdot;\cdot;f:\forall\alpha.\alpha\to\alpha\vdash\mathbf{false}:\mathtt{bool}}\quad\dfrac{\dfrac{}{\cdot;\cdot\vdash\mathtt{bool}:*}}{\cdot;\cdot;f:\forall\alpha.\alpha\to\alpha\vdash f(\mathtt{bool}):\mathtt{bool}\to\mathtt{bool}}\ \textbf{(ty-poly-var)}}{\cdot;\cdot;f:\forall\alpha.\alpha\to\alpha\vdash f(\mathtt{bool})(\mathbf{false}):\mathtt{bool}}\ \textbf{(ty-app)}$$

Then we have the following derivation.

$$\dfrac{\mathcal{D}_1\quad\dfrac{\mathcal{D}_2\quad\mathcal{D}_3}{\cdot;\cdot;f:\forall\alpha.\alpha\to\alpha\vdash\langle f(\mathtt{int})(\overline{0}),f(\mathtt{bool})(\mathbf{false})\rangle:\mathtt{int}*\mathtt{bool}}\ \textbf{(ty-prod)}}{\cdot;\cdot;\cdot\vdash\mathbf{let}\ f=\Lambda\alpha.\lambda x:\alpha.x\ \mathbf{in}\ \langle f(\mathtt{int})(\overline{0}),f(\mathtt{bool})(\mathbf{false})\rangle\ \mathbf{end}:\mathtt{int}*\mathtt{bool}}\ \textbf{(ty-let)}$$

Lemma 6.2.3 *If $\phi;\Delta\vdash\tau_i:*$ are derivable for $1=1,\ldots,n$ and $\phi;\Delta,\vec{\alpha};\Gamma\vdash e:\sigma$ is also derivable, then $\phi;\Delta;\Gamma[\vec{\alpha}\mapsto\vec{\tau}]\vdash e[\vec{\alpha}\mapsto\vec{\tau}]:\sigma[\vec{\alpha}\mapsto\vec{\tau}]$ is derivable, where $\vec{\alpha}=\alpha_1,\ldots,\alpha_n$ and $\vec{\tau}=\tau_1,\ldots,\tau_n$.*

Proof This simply follows from a structural induction on the derivation of $\phi;\Delta,\vec{\alpha};\Gamma\vdash e:\sigma$. ■

Lemma 6.2.4 *If both $\phi;\Delta;\Gamma\vdash v:\sigma_1$ and $\phi,\phi_1;\Delta;\Gamma,x:\sigma_1\vdash e:\sigma$ are derivable, then $\phi,\phi_1;\Delta;\Gamma\vdash e[x\mapsto v]:\sigma$ is also derivable.*

Proof The proof follows from a structural induction on the derivation $\mathcal{D}$ of $\phi,\phi_1;\Delta;\Gamma,x:\sigma_1\vdash e:\sigma$. We present one case as follows.

$\mathcal{D}=\dfrac{\phi,\phi_1;\Delta\vdash\tau_1:*\quad\cdots\quad\phi,\phi_1;\Delta\vdash\tau_n:*}{\phi,\phi_1;\Delta;\Gamma,x:\forall\vec{\alpha}.\tau\vdash x(\vec{\tau}):\tau[\vec{\alpha}\mapsto\vec{\tau}]}$ Since $\phi;\Delta;\Gamma\vdash v:\forall\vec{\alpha}.\tau$ is derivable, v is of form $\Lambda\vec{\alpha}.v_1$ and $\phi;\Delta,\vec{\alpha};\Gamma\vdash v:\tau$ is also derivable by inverting the rule **(ty-poly-intro)**. We require that $\vec{\alpha}$ have no free occurrences in the types of the variables declared in Γ. This implies that $\phi,\phi_1;\Delta,\vec{\alpha};\Gamma\vdash v:\tau$ is also derivable.

Notice $x(\vec{\tau})[x:=\Lambda\vec{\alpha}.v_1]=v_1[\vec{\alpha}\mapsto\vec{\tau}]$. By Lemma 6.2.3, $\phi,\phi_1;\Delta;\Gamma\vdash v[\vec{\alpha}\mapsto\vec{\tau}]:\tau[\vec{\alpha}\mapsto\vec{\tau}]$ is derivable since $\Gamma=\Gamma[\vec{\alpha}\mapsto\vec{\tau}]$.

All other cases can be treated similarly. ■

6.2.2 Dynamic Semantics

In addition to the evaluation rules for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$, we also need the following rule to formulate the natural semantics of $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$.

$$\dfrac{e\hookrightarrow_d v}{\Lambda\alpha.e\hookrightarrow_d\Lambda\alpha.v}\ \textbf{(ev-poly)}$$

Theorem 6.2.5 *(Type preservation for* $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$*) If both* $e \hookrightarrow_d v$ *and* $\phi;\Delta;\Gamma \vdash e : \sigma$ *are derivable, then* $\phi;\Delta;\Gamma \vdash v : \sigma$ *is also derivable.*

Proof The proof, parallel to that of Theorem 5.1.1, is based on a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_d v$ and the derivation of $\phi;\Delta;\Gamma \vdash e : \sigma$, lexicographically ordered. We present a few interesting cases as follows.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_d v_1 \quad e_2[x \mapsto v_1] \hookrightarrow_d v}{(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}) \hookrightarrow_d v}$ Then we also have the following derivation.

$$\frac{\phi;\Delta;\Gamma \vdash e_1 : \sigma_1 \quad \phi,\phi_1;\Delta;\Gamma,x_1 : \sigma_1 \vdash e_2 : \tau}{\phi,\phi_1;\Delta;\Gamma \vdash \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau}\ (\mathbf{ty\text{-}let})$$

By induction hypothesis, $\phi;\Delta;\Gamma \vdash v_1 : \sigma_1$ is derivable. Therefore, $\phi,\phi_1;\Delta;\Gamma \vdash e_2[x_1 \mapsto v_1] : \tau$ by Lemma 6.2.4. This leads to a derivation of $\phi,\phi_1;\Delta;\Gamma \vdash v : \tau$.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_d v_1}{\Lambda\alpha.e_1 \hookrightarrow_d \Lambda\alpha.v_1}$ Then we also have the following derivation.

$$\frac{\cdot;\Delta,\alpha;\Gamma \vdash e_1 : \sigma_1}{\cdot;\Delta;\Gamma \vdash \Lambda\alpha.e_1 : \forall\alpha.\sigma_1}\ \mathbf{(ty\text{-}poly\text{-}intro)}$$

By induction hypothesis, $\cdot;\Delta,\alpha;\Gamma \vdash v_1 : \sigma_1$ is derivable. This readily leads to a derivation of $\cdot;\Delta;\Gamma \vdash \Lambda\alpha.v_1 : \forall\alpha.\sigma_1$

The rest of the cases can be treated similarly. ∎

Clearly, the definition of the index erasure function $\|\cdot\|$ can be extended as follows.

$$\begin{array}{lcl}
\|\alpha\| & = & \alpha \\
\|\forall\alpha.\sigma\| & = & \forall\alpha.\|\sigma\| \\
\|\Lambda\alpha.e\| & = & \Lambda\alpha.\|e\| \\
\|x(\vec{\tau})\| & = & x(\|\vec{\tau}\|) \\
\|c(\vec{\tau})[\vec{i}]\| & = & c(\|\vec{\tau}\|) \\
\|c(\vec{\tau})[\vec{i}](e)\| & = & c(\|\vec{\tau}\|)(\|e\|)
\end{array}$$

Now an immediate question is whether we still have the corresponding versions of Theorem 5.1.2, Theorem 5.1.3 and Theorem 5.1.5 in $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$. Unsurprisingly, the answer is positive.

The relation between $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ and $\mathrm{ML}_0^{\forall}$ is similar to that between $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ and ML_0. The following theorem corresponds to Theorem 5.1.2. Therefore, if an (untyped) expression in $\lambda_{\mathrm{val}}^{\mathrm{pat}}$ is typable in $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$, it is already typable in $\mathrm{ML}_0^{\forall}$. This reiterates that the objective of our work is to assign programs more accurate types rather than make more programs typable.

Theorem 6.2.6 *If* $\phi;\Delta;\Gamma \vdash e : \sigma$ *is derivable in* $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$*, then* $\Delta;\|\Gamma\| \vdash \|e\| : \|\sigma\|$ *is derivable in* $\mathrm{ML}_0^{\forall}$.

Proof The proof follows from a structural induction on the derivation of $\phi;\Delta;\Gamma \vdash e : \sigma$ ∎

Theorem 6.2.7 *If $e \hookrightarrow_d v$ is derivable in $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$, then $\|e\| \hookrightarrow_0 \|v_0\|$ is derivable in $\mathrm{ML}_0^{\forall}$.*

Proof This follows a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_d v$. We present a few interesting cases.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_d v_1 \quad e_2[x \mapsto v_1] \hookrightarrow_d v}{(\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end}) \hookrightarrow_d v}$ By induction hypothesis, $\|e_1\| \hookrightarrow_0 \|v_1\|$ and $\|e_2[x \mapsto v_1]\| \hookrightarrow_0 \|v\|$ are derivable. It can be readily verified that $\|e_2[x \mapsto v_1]\| = \|e_2\|[x \mapsto \|v_1\|]$. This leads to the following derivation.

$$\frac{\|e_1\| \hookrightarrow_0 \|v_1\| \quad \|e_2\|[x \mapsto \|v_1\|] \hookrightarrow_0 \|v\|}{\textbf{let } x = \|e_1\| \textbf{ in } \|e_2\| \textbf{ end} \hookrightarrow_0 \|v\|} \textbf{ (ev-let)}$$

Hence, $\|\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end}\| \hookrightarrow_0 \|v\|$ is derivable.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_d v_1}{\Lambda\alpha.e_1 \hookrightarrow_d \Lambda\alpha.v_1}$ By induction hypothesis, $\|e_1\| \hookrightarrow_0 \|v_1\|$ is derivable in $\mathrm{ML}_0^{\forall}$. Since $\|\Lambda\alpha.e_1\| = \Lambda\alpha.\|e_1\|$ and $\|\Lambda\alpha.v_1\| = \Lambda\alpha.\|v_1\|$, $\|\Lambda\alpha.e_1\| \hookrightarrow_d \|\Lambda\alpha.v_1\|$ is derivable in $\mathrm{ML}_0^{\forall}$.

■

Theorem 6.2.8 *Given $\phi;\Gamma \vdash e : \sigma$ derivable in $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$. If $e^0 = \|e\| \hookrightarrow_0 v^0$ is derivable for some v^0 in $\mathrm{ML}_0^{\forall}$, then there exists v in $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ such that $e \hookrightarrow_d v$ is derivable and $\|v\| = v^0$.*

Proof The proof is similar to that of Theorem 5.1.5, and therefore we omit it here. ■

6.2.3 Elaboration

We slightly extend the external language $\mathrm{DML}_0(C)$ as follows, yielding the external language $\mathrm{DML}(C)$ for $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$.

$$\text{expressions} \quad e \quad ::= \quad \cdots \mid \Lambda\alpha.e$$

Theoretically, there are no technical obstacles which prevent us from directly formulating elaboration rules and then constraint generation rules for $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ as is done for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$. However, in practice there are some serious disadvantages for doing so, which we briefly explain as follows.

In Chapter 1, we used the following example demonstrating how to refine a polymorphic datatype into a polymorphic dependent type.

```
datatype 'a list = nil | cons of 'a * 'a list
typeref 'a list of nat (* indexing datatype 'a list with nat *)
with nil <| 'a list(0)
   | cons <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)
```

After this declaration, *cons* is of type

$$\forall\alpha.\Pi n : nat.\alpha * (\alpha)list(n) \to (\alpha)list(n+1).$$

Suppose that we have already refined the type `int`, assigning the types $\texttt{int}(0)$ and $\texttt{int}(1)$ to $\overline{0}$ and $\overline{1}$, respectively. Now let us see how to elaborate the expression $cons(\langle\overline{0}, cons(\langle\overline{1}, nil\rangle)\rangle)$. Intuitively, we should instantiate the type of the first *cons* to

$$\texttt{int}(0) * (\texttt{int}(0))list(n) \to (\texttt{int}(0))list(n+1),$$

and then check $cons(\langle\overline{1}, nil\rangle)$ against $(\texttt{int}(0))list(n+1)$. This leads to the instantiation of the type of the second *cons* to

$$\texttt{int}(0) * (\texttt{int}(0))list(n) \to (\texttt{int}(0))list(n+1),$$

and we then check $\overline{1}$ against $\texttt{int}(0)$. This results in a type error since $\overline{1}$ cannot be of type $\texttt{int}(0)$. In contrast, there exists no problem elaborating $cons(\overline{0}, cons(\overline{1}, nil))$ into an expression of type $(\texttt{int})list$ in $\mathrm{ML}_0^{\forall}$. This would destroy the precious compatibility property we expect, that is, a valid ML program written in an external language for ML can always be treated as a valid DML(C) program. Fortunately, the reader can readily verify that the elaboration of $cons(\overline{0}, cons(\overline{1}, nil))$ would have succeeded if we had started *checking* it against the type $\Sigma a : int.\texttt{int}(a)$. This example shows that it is highly questionable to directly combine the dependent type-checking with polymorphic type-checking.

There is yet another disadvantage. One main objective of designing a dependent type system is to enable the programmer to capture more program errors at compile time. Therefore, it is crucial that *adequately informative* type error message can be issued once type-checking fails. This, however, would be greatly complicated if errors resulted from both dependent type-checking and polymorphic type-checking are mingled together, especially given that it is already difficult enough to report only errors from polymorphic type-checking.

These practical issues prompt us to adopt a two-phase elaboration for $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$.

Phase One

Theorem 6.2.6 states that if e is well-typed in $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ then its index erasure $\|e\|$ is well-typed in $\mathrm{ML}_0^{\forall}$. Therefore, given a program e in $\mathrm{DML}(C)$, if e can be successfully elaborated in $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$, then its index erasure $\|e\|$ can be elaborated in $\mathrm{ML}_0^{\forall}$. We use the W-algorithm for polymorphic type-checking in ML (Milner 1978) to check whether $\|e\|$ is well-typed in $\mathrm{ML}_0^{\forall}$. This is a crucial step towards guaranteeing full compatibility of $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ with $\mathrm{ML}_0^{\forall}$ in the sense that a program written in an *external* language for $\mathrm{ML}_0^{\forall}$ should always be accepted by $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ if it is by $\mathrm{ML}_0^{\forall}$. For the parts of a program which use dependent types, we expect this phase of elaboration to be highly efficient since there are abundant programmer-supplied type annotations available. In practice, this leads to accurate type error message report because type-checking is essentially performed in a top-down fashion.

Phase Two

After the first phase of elaboration, we perform the following.

- If a declared function is not annotated, we annotate it with the ML-type inferred for this function from phase one.

$$\frac{}{(\alpha, \alpha', \mathbf{pos})} \qquad \frac{(\alpha, \tau_1, \mathbf{pos}) \quad (\alpha, \tau_2, \mathbf{pos})}{(\alpha, \tau_1 * \tau_2, \mathbf{pos})} \qquad \frac{(\alpha, \tau_1, \mathbf{neg}) \quad (\alpha, \tau_2, \mathbf{pos})}{(\alpha, \tau_1 \to \tau_2, \mathbf{pos})}$$

$$\frac{\alpha \text{ is not } \alpha'}{(\alpha, \alpha', \mathbf{neg})} \qquad \frac{(\alpha, \tau_1, \mathbf{neg}) \quad (\alpha, \tau_2, \mathbf{neg})}{(\alpha, \tau_1 * \tau_2, \mathbf{neg})} \qquad \frac{(\alpha, \tau_1, \mathbf{pos}) \quad (\alpha, \tau_2, \mathbf{neg})}{(\alpha, \tau_1 \to \tau_2, \mathbf{neg})}$$

$$\frac{(\alpha, \tau_1, s(1)) \quad \cdots \quad (\alpha, \tau_m, s(m))}{(\alpha, (\tau_1, \ldots, \tau_n)\delta, \mathbf{pos})} \qquad \frac{(\alpha, \tau_1, \overline{s}(1)) \quad \cdots \quad (\alpha, \tau_m, \overline{s}(m))}{(\alpha, (\tau_1, \ldots, \tau_n)\delta, \mathbf{neg})}$$

Figure 6.7: The inference rules for datatype constructor status

- For a let-expression **let** $x = e_1$ **in** e_2 **end**, if the inferred type scheme of x is of form $\forall\vec{\alpha}.\tau$, we replace every free occurrence of x in e_2 with $x(\vec{\tau})$ for some appropriate $\vec{\tau}$ inferred from the first phase of elaboration. Notice that these $\vec{\tau}$ are ML-types. If the programmer would like to instantiate $\vec{\alpha}$ with some dependent types, this must be written in the program. For instance, the array subscript function *sub* is of the following type;

  $$\forall\alpha.(\alpha)array * \mathtt{int} \to \alpha$$

  if we need a subscript function which only acts on an array of natural numbers in a block of code, we can declare **let** $subNat = sub(\Sigma i : nat.\mathtt{int}(i))$ **in** ... **end**; this assures that the type variable α in the type of *sub* is instantiated with the *dependent type* $\Sigma i : nat.\mathtt{int}(i)$, which is the type of natural numbers.

- If a datatype constructor δ is refined with index objects from sort γ, then we replace all occurrences of $(\tau_1, \ldots, \tau_n)\delta$ with $\Sigma a : \gamma.(\tau_1, \ldots, \tau_n)\delta(a)$. This process is then performed recursively on τ_i for $i = 1, \ldots, n$.

After the above processing is done, we can readily elaborate the program in the way described in Section 5.2. This concludes the informal description of a two-phase elaboration for $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$.

6.2.4 Coercion

Coercion between polymorphic datatypes needs some special care. An informal view is given as follows. Assume that type τ_1 can be coerced into type τ_2; if α occurs *positively* in $(\alpha)\delta$, then $(\tau_1)\delta$ should be able to coerce into $(\tau_2)\delta$; if α occurs *negatively* in $(\alpha)\delta$, then $(\tau_2)\delta$ should be able to coerce into $(\tau_1)\delta$. In order to handle more general cases, we introduce the notion of status as follows.

Let δ be a datatype constructor declared in ML and c_i are constructors of type $\forall\alpha_1 \ldots \forall\alpha_m.\tau_i \to (\alpha_1, \ldots, \alpha_m)\delta$ associated with δ for $i = 1, \ldots, n$. A status s for δ is a function with domain $\mathbf{dom}(s) = \{1, \ldots, m\}$ and range $\{\mathbf{pos}, \mathbf{neg}\}$. We use $\overline{s}$ for the dual status of s, that is $\overline{s}(k) = \mathbf{neg}$ if and only if $s(k) = \mathbf{pos}$ for $k = 1, \ldots, m$.

We say that δ has status s if for every $k \in \mathbf{dom}(s)$, $(\alpha_k, \tau_i, s(k))$ can be derived for $i = 1, \ldots, n$ with the rules in Figure 6.7.

This can be readily extended to mutually recursively declared datatype constructors in ML.

Assume that a datatype constructor δ is of status s. We say that $(\tau_1, \ldots, \tau_m)\delta(i)$ can be coerced into $(\tau'_1, \ldots, \tau'_m)\delta(i)$ if τ_k coerces into τ'_k for those k such that $s(k) = \mathbf{pos}$ and τ'_k coerces into τ_k for the rest.

We currently disallow coercions between $(\tau_1, \ldots, \tau_m)\delta(i)$ and $(\tau'_1, \ldots, \tau'_m)\delta(i)$ if δ cannot be assigned a status. Clearly, it is possible to extend the range of a status function to containing **neutral** and **mixed**, which roughly mean "both positive and negative" and "neither positive nor negative", respectively. However, it is yet to see whether such extension would be of some practical relevance.

6.3 Summary

Polymorphism is largely orthogonal to the development of dependent types. In this chapter, ML_0 is extended to $\mathrm{ML}_0^{\forall}$ with let-polymorphism, and this sets up the machinery we need for combining dependent types with let-polymorphism. Then the language $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ is introduced, which extends $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ with let-polymorphism. The relation between $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ and $\mathrm{ML}_0^{\forall}$ is parallel to that between $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ and ML_0. However, some serious problems show up when elaboration is concerned. This prompts us to adopt a two-phase elaboration process, which does the usual ML-type checking in the first phase and the dependent type-checking in the second phase. This seems to be a clean and practical solution.

$\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ is a pure call-by-value functional programming language, that is, it contains no imperative features. Therefore, the natural move is to extend $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ with some imperative features, which consists the topic of the next chapter.

Chapter 7

Effects

We have so far developed the type theory of dependent types in a pure functional programming language $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$, which lacks the imperative features of ML. In this chapter, we extend the language $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ to accommodate exceptions and references. We will examine the potential problems and present the approaches to solving them. The organization of the chapter is as follows.

We first extend the language ML_0 with the exception mechanism and formulate the language $\mathrm{ML}_{0,\mathrm{exc}}$. After proving the type preservation theorem for $\mathrm{ML}_{0,\mathrm{exc}}$, we extend it with the references. This yields the language $\mathrm{ML}_{0,\mathrm{exc,ref}}$. Again, we prove the type preservation theorem for $\mathrm{ML}_{0,\mathrm{exc,ref}}$. We then exhibit what the problems are if we extend $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ with references and exception mechanism. This leads to adopting the *value restriction* approach (Wright 1995). Finally, we study the relation between $\mathrm{ML}_{0,\mathrm{exc,ref}}$ and $\mathrm{ML}_{0,\mathrm{exc,ref}}^{\forall,\Pi,\Sigma}(C)$.

7.1 Exception Mechanism

The exception mechanism is an important feature of ML which allows programs to perform non-local "jumps" in the flow of control by setting a *handler* during evaluation of an expression that may be invoked by *raising* an exception. Exceptions are value-carrying in the sense that they can pass values to exception handlers. Because of the dynamic nature of exception handlers, it is required that all the exception values have a single datatype **Exc**, which can then be extended by the programmer. This is called *extensible datatype*. We assume that **Exc** is a distinguished built-in base type, but do not concern ourselves with how constructors in this datatype are created.

7.1.1 Static Semantics

The language ML_0 is extended to the language $\mathrm{ML}_{0,\mathrm{exc}}$ as follows. An *answer* is either a value of an *uncaught* exception.

base types	β	::=	$\cdots \mid \mathbf{Exc}$
expressions	e	::=	$\cdots \mid \mathbf{raise}(e) \mid \mathbf{handle}\ e\ \mathbf{with}\ ms$
answers	ans	::=	$\cdots \mid \mathbf{raise}(v)$

In addition to the typing rules for ML_0, we need the following ones for handling the newly introduced language constructs.

$$\frac{}{x \hookrightarrow_0 x}\ \textbf{(ev-var)}$$

$$\frac{}{\langle\rangle \hookrightarrow_0 \langle\rangle}\ \textbf{(ev-unit)}$$

$$\frac{e \hookrightarrow_0 \textbf{raise}(v)}{c(e) \hookrightarrow_0 \textbf{raise}(v)}\ \textbf{(ev-cons-1)}$$

$$\frac{e \hookrightarrow_0 v}{c(e) \hookrightarrow_0 c(v)}\ \textbf{(ev-cons-2)}$$

$$\frac{e_1 \hookrightarrow_0 \textbf{raise}(v)}{\langle e_1, e_2\rangle \hookrightarrow_0 \textbf{raise}(v)}\ \textbf{(ev-prod-1)}$$

$$\frac{e_1 \hookrightarrow_0 v_1 \quad e_2 \hookrightarrow_0 \textbf{raise}(v)}{\langle e_1, e_2\rangle \hookrightarrow_0 \textbf{raise}(v)}\ \textbf{(ev-prod-2)}$$

$$\frac{e_1 \hookrightarrow_0 v_1 \quad e_2 \hookrightarrow_0 v_2}{\langle e_1, e_2\rangle \hookrightarrow_0 \langle v_1, v_2\rangle}\ \textbf{(ev-prod-3)}$$

$$\frac{e \hookrightarrow_0 \textbf{raise}(v)}{\textbf{case}\ e\ \textbf{of}\ ms \hookrightarrow_0 \textbf{raise}(v)}\ \textbf{(ev-case-1)}$$

$$\frac{e_0 \hookrightarrow_0 v_0 \quad \textbf{match}(v_0, p_k) \Longrightarrow \theta \text{ for some } 1 \leq k \leq n \quad e_k[\theta] \hookrightarrow_0 ans}{(\textbf{case}\ e_0\ \textbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)) \hookrightarrow_0 ans}\ \textbf{(ev-case-2)}$$

Figure 7.1: The natural semantics for $\mathrm{ML}_{0,\mathrm{exc}}$ (I)

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash ms : \textbf{Exc} \Rightarrow \tau}{\Gamma \vdash (\textbf{handle}\ e\ \textbf{with}\ ms) : \tau}\ \textbf{(ty-handle)}$$

$$\frac{\Gamma \vdash e : \textbf{Exc}}{\Gamma \vdash \textbf{raise}(e) : \tau}\ \textbf{(ty-raise)}$$

7.1.2 Dynamic Semantics

We now present the evaluation rules for $\mathrm{ML}_{0,\mathrm{exc}}$ in Figure 7.1 and Figure 7.2, upon which the natural semantics of $\mathrm{ML}_{0,\mathrm{exc}}$ is established. Notice that a successful evaluation of an expression e result in either a value or an uncaught exception.

Theorem 7.1.1 *(Type preservation) Assume that* $\Gamma \vdash e : \tau$ *is derivable in* $\mathrm{ML}_{0,\mathrm{exc}}$. *If* $e \hookrightarrow_0 ans$ *for some answer* ans, *then* $\Gamma \vdash ans : \tau$ *is derivable.*

Proof The proof is parallel to the proof of Theorem 2.2.7, following from a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_0 v$. We present a few cases.

$\mathcal{D} = \dfrac{e_1 \hookrightarrow_0 \textbf{raise}(v_1)}{\textbf{raise}(e_1) \hookrightarrow_0 \textbf{raise}(v_1)}$ The derivation of $\Gamma \vdash \textbf{raise}(e_1) : \tau$ must be of the following

$$\frac{e_1 \hookrightarrow_0 \mathbf{raise}(v)}{e_1(e_2) \hookrightarrow_0 \mathbf{raise}(v)}\ \textbf{(ev-app-1)}$$

$$\frac{e_1 \hookrightarrow_0 (\mathbf{lam}\ x : \tau.e) \quad e_2 \hookrightarrow_0 \mathbf{raise}(v)}{e_1(e_2) \hookrightarrow_0 \mathbf{raise}(v)}\ \textbf{(ev-app-2)}$$

$$\frac{e_1 \hookrightarrow_0 (\mathbf{lam}\ x : \tau.e) \quad e_2 \hookrightarrow_0 v_2 \quad e[x \mapsto v_2] \hookrightarrow_0 ans}{e_1(e_2) \hookrightarrow_0 ans}\ \textbf{(ev-app-3)}$$

$$\frac{e_1 \hookrightarrow_0 \mathbf{raise}(v)}{(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}) \hookrightarrow_0 \mathbf{raise}(v)}\ \textbf{(ev-let-1)}$$

$$\frac{e_1 \hookrightarrow_0 v_1 \quad e_2[x \mapsto v_1] \hookrightarrow_0 ans}{(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}) \hookrightarrow_0 ans}\ \textbf{(ev-let-2)}$$

$$\frac{}{(\mathbf{fix}\ f : \tau.u) \hookrightarrow_0 u[f \mapsto (\mathbf{fix}\ f : \tau.u)]}\ \textbf{(ev-fix)}$$

$$\frac{e \hookrightarrow_0 \mathbf{raise}(v)}{\mathbf{raise}(e) \hookrightarrow_0 \mathbf{raise}(v)}\ \textbf{(ev-raise-1)}$$

$$\frac{e \hookrightarrow_0 v}{\mathbf{raise}(e) \hookrightarrow_0 \mathbf{raise}(v)}\ \textbf{(ev-raise-2)}$$

$$\frac{e \hookrightarrow_0 \mathbf{raise}(v)}{\mathbf{handle}\ e\ \mathbf{with}\ ms \hookrightarrow_0 \mathbf{raise}(v)}\ \textbf{(ev-handler-1)}$$

$$\frac{e_0 \hookrightarrow_0 \mathbf{raise}(v_0) \quad \mathbf{match}(v_0, p_k) \Longrightarrow \theta \text{ for some } 1 \leq k \leq n \quad e_k[\theta] \hookrightarrow_0 ans}{\mathbf{handle}\ e_0\ \mathbf{with}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \hookrightarrow_0 ans}\ \textbf{(ev-handler-2)}$$

$$\frac{e \hookrightarrow_0 v}{\mathbf{handle}\ e\ \mathbf{with}\ ms \hookrightarrow_0 v}\ \textbf{(ev-handler-3)}$$

Figure 7.2: The natural semantics for $\mathrm{ML}_{0,\mathrm{exc}}$ (II)

form.

$$\frac{\Gamma \vdash e_1 : \mathbf{Exc}}{\Gamma \vdash \mathbf{raise}(e_1) : \tau}\ \textbf{(ty-raise)}$$

By induction hypothesis, $\Gamma \vdash \mathbf{raise}(v_1) : \mathbf{Exc}$ is derivable. Hence, we have a derivation of $\Gamma \vdash v_1 : \mathbf{Exc}$. This leads to the following.

$$\frac{\Gamma \vdash v_1 : \mathbf{Exc}}{\Gamma \vdash \mathbf{raise}(v_1) : \tau}\ \textbf{(ty-raise)}$$

$\mathcal{D} = \dfrac{e_0 \hookrightarrow_0 \mathbf{raise}(v_0) \quad \mathbf{match}(v_0, p_k) \Longrightarrow \theta \text{ for some } 1 \leq k \leq n \quad e_k[\theta] \hookrightarrow_0 ans}{\mathbf{handle}\ e_0\ \mathbf{with}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \hookrightarrow_0 ans}$ Then we have a derivation of the following form.

$$\frac{\Gamma \vdash e_0 : \tau \quad \Gamma \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \mathbf{Exc} \Rightarrow \tau}{\Gamma \vdash (\mathbf{handle}\ e_0\ \mathbf{with}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)) : \tau}\ \textbf{(ty-handle)}$$

By induction hypothesis, $\Gamma \vdash \mathbf{raise}(v_0) : \tau$ is derivable. This leads to the following derivation.

$$\frac{\Gamma \vdash v_0 : \mathbf{Exc}}{\Gamma \vdash \mathbf{raise}(v_0) : \tau}\ \textbf{(ty-raise)}$$

Notice $\Gamma \vdash p_i \Rightarrow e_i : \mathbf{Exc} \Rightarrow \tau$ are derivable for $1 \leq i \leq n$. Hence $p_k \downarrow \mathbf{Exc} \rhd \Gamma'$ is derivable for some Γ' and $\Gamma, \Gamma' \vdash e_k : \tau$ is derivable. By Lemma 2.2.5, $\Gamma \vdash \theta : \Gamma'$ is derivable. This leads to a derivation of $\Gamma \vdash e_k[\theta] : \tau$ by Lemma 2.2.4. By induction hypothesis, $\Gamma \vdash ans : \tau$ is derivable.

$\mathcal{D} = \dfrac{e_0 \hookrightarrow_0 v}{\mathbf{handle}\ e_0\ \mathbf{with}\ ms \hookrightarrow_0 v}$ Then we have a derivation of the following form.

$$\frac{\Gamma \vdash e_0 : \tau \quad \Gamma \vdash ms : \mathbf{Exc} \Rightarrow \tau}{\Gamma \vdash (\mathbf{handle}\ e_0\ \mathbf{with}\ ms) : \tau}\ \textbf{(ty-handle)}$$

By induction hypothesis, $\Gamma \vdash ans : \tau$ is derivable. Hence, we are done.

All other cases can be treated similarly. ∎

7.2 References

A unique aspect of ML is the use of reference types to segregate mutable data structures from immutable ones. Given a type τ, the reference type τ **ref** stands for the type of reference cell which can only store a value of type τ.

7.2.1 Static Semantics

The language $\mathrm{ML}_{0,\mathrm{exc}}$ is extended to the language $\mathrm{ML}_{0,\mathrm{exc,ref}}$ as follows. An *answer* is either a value or an uncaught exception associated with a piece of memory.

$$\begin{array}{lrcl}
\text{types} & \tau & ::= & \cdots \mid \tau\ \mathbf{ref} \\
\text{expressions} & e & ::= & \cdots \mid \mathbf{letref}\ M\ \mathbf{in}\ e\ \mathbf{end} \mid e_1 := e_2 \mid !e \\
\text{memory} & M & ::= & \cdot \mid M, x : \tau\ \mathbf{is}\ v \\
\text{programs} & \mathit{prog} & ::= & \mathbf{letref}\ M\ \mathbf{in}\ e\ \mathbf{end} \\
\text{answers} & \mathit{ans} & ::= & \mathbf{letref}\ M\ \mathbf{in}\ v\ \mathbf{end} \mid \mathbf{letref}\ M\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}
\end{array}$$

Let $\mathbf{dom}(M)$ be defined as follows.

$$\mathbf{dom}(\cdot) = \emptyset \qquad \mathbf{dom}(M, x : \tau\ \mathbf{is}\ v) = \mathbf{dom}(M) \cup \{x\}$$

For every $x \in \mathbf{dom}(M)$, $M(x)$ is v if $x : \tau\ \mathbf{is}\ v$ is declared in M. For $x \in \mathbf{dom}(M)$, we use $M[x := v]$ for the memory which replaces with $x : \tau\ \mathbf{is}\ v$ the declaration $x : \tau\ \mathbf{is}\ v_{\mathrm{old}}$ in M for some τ and v_{old}.

We need the following typing rules for handling the newly introduced language constructs.

$$\frac{\Gamma' = x_1 : \tau_1\ \mathbf{ref}, \ldots, x_n : \tau_n\ \mathbf{ref} \quad \Gamma, \Gamma' \vdash v_i : \tau_i \quad (1 \le i \le n)}{\Gamma \vdash (x_1 : \tau_1\ \mathbf{is}\ v_1, \ldots, x_n : \tau_1\ \mathbf{is}\ v_n) : \Gamma'}\ \textbf{(ty-memo)}$$

$$\frac{\Gamma \vdash M : \Gamma' \quad \Gamma, \Gamma' \vdash e : \tau}{\Gamma \vdash \mathbf{letref}\ M\ \mathbf{in}\ e\ \mathbf{end} : \tau}\ \textbf{(ty-letref)}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \mathbf{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \mathbf{1}}\ \textbf{(ty-assign)}$$

$$\frac{\Gamma \vdash e : \tau\ \mathbf{ref}}{\Gamma \vdash !e : \tau}\ \textbf{(ty-deref)}$$

Note that we use $\mathbf{Ref}(e)$ as an abbreviation for $\mathbf{let}\ x = e\ \mathbf{in}\ \mathbf{letref}\ y : \tau\ \mathbf{is}\ x\ \mathbf{in}\ y\ \mathbf{end}\ \mathbf{end}$.

Example 7.2.1 *Given a derivation $\mathcal{D}$ of $\Gamma \vdash e : \tau$, we can construct the following derivation of $\Gamma \vdash \mathbf{Ref}(e) : \tau\ \mathbf{ref}$.*

$$\dfrac{\mathcal{D} \quad \dfrac{\dfrac{\dfrac{}{\Gamma, x : \tau, y : \tau\ \mathbf{ref} \vdash x : \tau}}{\Gamma, x : \tau \vdash (y : \tau\ \mathbf{is}\ x) : (y : \tau\ \mathbf{ref})} \quad \dfrac{}{\Gamma, x : \tau, y : \tau\ \mathbf{ref} \vdash y : \tau\ \mathbf{ref}}}{\Gamma, x : \tau \vdash \mathbf{letref}\ y : \tau\ \mathbf{is}\ x\ \mathbf{in}\ y\ \mathbf{end} : \tau\ \mathbf{ref}}\ \textbf{(ty-letref)}}{\Gamma \vdash \mathbf{Ref}(e) : \tau\ \mathbf{ref}}\ \textbf{(ty-let)}$$

7.2.2 Dynamic Semantics

The natural semantics of $\mathrm{ML}_{0,\mathrm{exc,ref}}$ is given in Figure 7.3 and Figure 7.4.

Proposition 7.2.2 *If the following is derivable, then $\mathbf{dom}(M_1) \subseteq \mathbf{dom}(M_2)$.*

$$\mathbf{letref}\ M_1\ \mathbf{in}\ e\ \mathbf{end} \hookrightarrow_0 \mathbf{letref}\ M_2\ \mathbf{in}\ v\ \mathbf{end}$$

$$\frac{}{\textbf{letref } M \textbf{ in } \langle\rangle \textbf{ end} \hookrightarrow_0 \textbf{letref } M \textbf{ in } \langle\rangle \textbf{ end}} \textbf{ (ev-unit)}$$

$$\frac{}{\textbf{letref } M \textbf{ in } c \textbf{ end} \hookrightarrow_0 \textbf{letref } M \textbf{ in } c \textbf{ end}} \textbf{ (ev-cons-wo)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}}{\textbf{letref } M_1 \textbf{ in } c(e) \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-cons-w-1)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } v \textbf{ end}}{\textbf{letref } M_1 \textbf{ in } c(e) \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } c(v) \textbf{ end}} \textbf{ (ev-cons-w-2)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}}{\textbf{letref } M_1 \textbf{ in } \langle e_1, e_2\rangle \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-prod-1)}$$

$$\frac{\begin{array}{c}\textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } v_1 \textbf{ end} \\ \textbf{letref } M_2 \textbf{ in } e_2 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_3 \textbf{ in raise}(v) \textbf{ end}\end{array}}{\textbf{letref } M_1 \textbf{ in } \langle e_1, e_2\rangle \textbf{ end} \hookrightarrow_0 \textbf{letref } M_3 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-prod-2)}$$

$$\frac{\begin{array}{c}\textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } v_1 \textbf{ end} \\ \textbf{letref } M_2 \textbf{ in } e_2 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_3 \textbf{ in } v_2 \textbf{ end}\end{array}}{\textbf{letref } M_1 \textbf{ in } \langle e_1, e_2\rangle \textbf{ end} \hookrightarrow_0 \textbf{letref } M_3 \textbf{ in } \langle v_1, v_2\rangle \textbf{ end}} \textbf{ (ev-prod-3)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}}{\textbf{letref } M_1 \textbf{ in case } e \textbf{ of } ms \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-case-1)}$$

$$\frac{\begin{array}{c}\textbf{letref } M_1 \textbf{ in } e_0 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } v_0 \textbf{ end} \\ \textbf{match}(v_0, p_k) \Longrightarrow \theta \text{ for some } 1 \leq k \leq n \\ \textbf{letref } M_2 \textbf{ in } e_k[\theta] \textbf{ end} \hookrightarrow_0 ans\end{array}}{\textbf{letref } M_1 \textbf{ in case } e_0 \textbf{ of } (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \textbf{ end} \hookrightarrow_0 ans} \textbf{ (ev-case-2)}$$

$$\frac{}{\textbf{letref } M \textbf{ in lam } x : \tau.e \textbf{ end} \hookrightarrow_0 \textbf{letref } M \textbf{ in lam } x : \tau.e \textbf{ end}} \textbf{ (ev-lam)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}}{\textbf{letref } M_1 \textbf{ in } e_1(e_2) \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-app-1)}$$

$$\frac{\begin{array}{c}\textbf{letref } M_1 \textbf{ in } e_1 \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in lam } x : \tau.e \textbf{ end end} \\ \textbf{letref } M_2 \textbf{ in } e_2 \hookrightarrow_0 \textbf{letref } M_3 \textbf{ in raise}(v) \textbf{ end end}\end{array}}{\textbf{letref } M_1 \textbf{ in } e_1(e_2) \textbf{ end} \hookrightarrow_0 \textbf{letref } M_3 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-app-2)}$$

$$\frac{\begin{array}{c}\textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } (\textbf{lam } x : \tau.e) \textbf{ end} \\ \textbf{letref } M_2 \textbf{ in } e_2 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_3 \textbf{ in } v_2 \textbf{ end} \\ \textbf{letref } M_3 \textbf{ in } e[x \mapsto v_2] \textbf{ end} \hookrightarrow_0 ans\end{array}}{\textbf{letref } M_1 \textbf{ in } e_1(e_2) \textbf{ end} \hookrightarrow_0 ans} \textbf{ (ev-app-3)}$$

Figure 7.3: The natural semantics for $\mathrm{ML}_{0,\mathrm{exc,ref}}$ (I)

$$\frac{\textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}}{\textbf{letref } M_1 \textbf{ in } (\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end}) \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-let-1)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e_1 \hookrightarrow_0 v_1 \textbf{ end} \quad \textbf{letref } M_2 \textbf{ in } e_2[x \mapsto v_1] \textbf{ end} \hookrightarrow_0 ans}{\textbf{letref } M_1 \textbf{ in } (\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end}) \textbf{ end} \hookrightarrow_0 ans} \textbf{ (ev-let-2)}$$

$$\frac{}{\textbf{letref } M \textbf{ in fix } f : \tau.u \textbf{ end} \hookrightarrow_0 \textbf{letref } M \textbf{ in } u[f \mapsto (\textbf{fix } f : \tau.u)] \textbf{ end}} \textbf{ (ev-fix)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}}{\textbf{letref } M_1 \textbf{ in raise}(e) \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-raise-1)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } v \textbf{ end}}{\textbf{letref } M_1 \textbf{ in raise}(e) \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-raise-2)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}}{\textbf{letref } M_1 \textbf{ in handle } e \textbf{ with } ms \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-handle-1)}$$

$$\frac{\begin{array}{c}\textbf{letref } M_1 \textbf{ in } e_0 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v_0) \textbf{ end} \\ \textbf{match}(v_0, p_k) \Longrightarrow \theta \text{ for some } 1 \leq k \leq n \\ \textbf{letref } M_2 \textbf{ in } e_k[\theta] \textbf{ end} \hookrightarrow_0 ans\end{array}}{\textbf{letref } M_1 \textbf{ in handle } e_0 \textbf{ with } (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \textbf{ end} \hookrightarrow_0 ans} \textbf{ (ev-handle-2)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } v \textbf{ end}}{\textbf{letref } M_1 \textbf{ in handle } e \textbf{ with } ms \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } v \textbf{ end}} \textbf{ (ev-handle-3)}$$

$$\frac{\textbf{letref } M_1, M_2 \textbf{ in } e \textbf{ end} \hookrightarrow_0 ans}{\textbf{letref } M_1 \textbf{ in letref } M_2 \textbf{ in } e \textbf{ end end} \hookrightarrow_0 ans} \textbf{ (ev-extrusion)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}}{\textbf{letref } M_1 \textbf{ in } e_1 := e_2 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-assign-1)}$$

$$\frac{\begin{array}{c}\textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } x \textbf{ end} \\ \textbf{letref } M_2 \textbf{ in } e_2 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_3 \textbf{ in raise}(v) \textbf{ end}\end{array}}{\textbf{letref } M_1 \textbf{ in } e_1 := e_2 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_3 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-assign-2)}$$

$$\frac{\begin{array}{c}\textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } x \textbf{ end} \\ \textbf{letref } M_2 \textbf{ in } e_2 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_3 \textbf{ in } v \textbf{ end}\end{array}}{\textbf{letref } M_1 \textbf{ in } e_1 := e_2 \textbf{ end} \hookrightarrow_0 \textbf{letref } M_3[x := v] \textbf{ in } \langle\rangle \textbf{ end}} \textbf{ (ev-assign-3)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}}{\textbf{letref } M_1 \textbf{ in } !e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in raise}(v) \textbf{ end}} \textbf{ (ev-deref-1)}$$

$$\frac{\textbf{letref } M_1 \textbf{ in } e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } x \textbf{ end}}{\textbf{letref } M_1 \textbf{ in } !e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } M_2(x) \textbf{ end}} \textbf{ (ev-deref-2)}$$

Figure 7.4: The natural semantics for $\mathrm{ML}_{0,\mathrm{exc,ref}}$ (II)

Proof This simply follows from the formulation of the evaluation rules. Note that **(ev-extrusion)** is the only rule which can expand the memory. ■

Theorem 7.2.3 *(Type preservation) Given a program* $P = \textbf{letref}\ M\ \textbf{in}\ e\ \textbf{end}$*, if* $\cdot \vdash P : \tau$ *and* $P \hookrightarrow_0 ans$ *are derivable in* $\mathrm{ML}_{0,\mathrm{exc,ref}}$*, then* $\cdot \vdash ans : \tau$ *is also derivable in* $\mathrm{ML}_{0,\mathrm{exc,ref}}$*.*

Proof The proof proceeds by a structural induction on the derivation $\mathcal{D}$ of $P \hookrightarrow_0 ans$. We present a few cases.

$$\mathcal{D} = \frac{\textbf{letref}\ M_1, M_2\ \textbf{in}\ e\ \textbf{end} \hookrightarrow_0 ans}{\textbf{letref}\ M_1\ \textbf{in letref}\ M_2\ \textbf{in}\ e\ \textbf{end end} \hookrightarrow_0 ans}$$

Then we have the following derivation.

$$\frac{\cdot \vdash M_1 : \Gamma_1 \quad \dfrac{\Gamma_1 \vdash M_2 : \Gamma_2 \quad \Gamma_1, \Gamma_2 \vdash e : \tau}{\Gamma_1 \vdash \textbf{letref}\ M_2\ \textbf{in}\ e\ \textbf{end} : \tau}\ \textbf{(ty-letref)}}{\cdot \vdash \textbf{letref}\ M_1\ \textbf{in letref}\ M_2\ \textbf{in}\ e\ \textbf{end end} : \tau}\ \textbf{(ty-letref)}$$

Since $\cdot \vdash M_1 : \Gamma_1$ and $\Gamma_1 \vdash M_2 : \Gamma_2$ are derivable, $\cdot \vdash M_1, M_2 : \Gamma_1, \Gamma_2$ is derivable. This leads to the following.

$$\frac{\cdot \vdash M_1, M_2 : \Gamma_1, \Gamma_2 \quad \Gamma_1, \Gamma_2 \vdash e : \tau}{\cdot \vdash \textbf{letref}\ M_1, M_2\ \textbf{in}\ e\ \textbf{end} : \tau}\ \textbf{(ty-letref)}$$

By induction hypothesis, $\cdot \vdash ans : \tau$ is derivable.

$$\mathcal{D} = \frac{\begin{array}{c}\textbf{letref}\ M_1\ \textbf{in}\ e_1\ \textbf{end} \hookrightarrow_0 \textbf{letref}\ M_2\ \textbf{in}\ x\ \textbf{end} \\ \textbf{letref}\ M_2\ \textbf{in}\ e_2\ \textbf{end} \hookrightarrow_0 \textbf{letref}\ M_3\ \textbf{in}\ v\ \textbf{end}\end{array}}{\textbf{letref}\ M_1\ \textbf{in}\ e_1 := e_2\ \textbf{end} \hookrightarrow_0 \textbf{letref}\ M_3[x := v]\ \textbf{in}\ \langle\rangle\ \textbf{end}}$$

Then we have a derivation of the following form.

$$\frac{\cdot \vdash M_1 : \Gamma_1 \quad \dfrac{\Gamma_1 \vdash e_1 : \tau\ \textbf{ref} \quad \Gamma_1 \vdash e_2 : \tau}{\Gamma_1 \vdash e_1 := e_2 : \textbf{1}}\ \textbf{(ty-assign)}}{\cdot \vdash \textbf{letref}\ M_1\ \textbf{in}\ e_1 := e_2\ \textbf{end} : \textbf{1}}\ \textbf{(ty-letref)}$$

This leads to the following.

$$\frac{\cdot \vdash M_1 : \Gamma_1 \quad \Gamma_1 \vdash e_1 : \tau\ \textbf{ref}}{\cdot \vdash \textbf{letref}\ M_1\ \textbf{in}\ e_1\ \textbf{end} : \tau\ \textbf{ref}}\ \textbf{(ty-letref)}$$

By induction hypothesis, $\cdot \vdash \textbf{letref}\ M_2\ \textbf{in}\ x\ \textbf{end} : \tau\ \textbf{ref}$ is derivable. This implies that $\cdot \vdash M_2 : \Gamma_2$ is derivable for some Γ_2 such that $\Gamma_2(x) = \tau\ \textbf{ref}$. By Proposition 7.2.2, $M_1 \subseteq M_2$. Hence, $\Gamma_1 \subseteq \Gamma_2$, and we have the following derivation.

$$\frac{\cdot \vdash M_2 : \Gamma_2 \quad \Gamma_2 \vdash e_2 : \tau}{\cdot \vdash \textbf{letref}\ M_2\ \textbf{in}\ e_2\ \textbf{end}}\ \textbf{(ty-letref)}$$

By induction hypothesis, $\cdot \vdash \textbf{letref}\ M_3\ \textbf{in}\ v\ \textbf{end} : \tau$ is derivable. This implies that we can derive $\cdot \vdash M_3 : \Gamma_3$ for some Γ_3 and $\Gamma_2 \subseteq \Gamma_3$. Therefore, $\cdot \vdash M_3[x := v] : \Gamma_3$ is also derivable, and this yields the following.

$$\frac{\cdot \vdash M_3[x := v] : \Gamma_3 \quad \Gamma_3 \vdash \langle\rangle : \textbf{1}}{\cdot \vdash \textbf{letref}\ M_3\ \textbf{in}\ \langle\rangle\ \textbf{end} : \textbf{1}}\ \textbf{(ty-letref)}$$

$$\mathcal{D} = \frac{\textbf{letref } M_1 \textbf{ in } e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } x \textbf{ end}}{\textbf{letref } M_1 \textbf{ in } !e \textbf{ end} \hookrightarrow_0 \textbf{letref } M_2 \textbf{ in } M_2(x) \textbf{ end}}$$

Then we have a derivation of the following form.

$$\dfrac{\dfrac{\cdot \vdash M_1 : \Gamma_1 \quad \dfrac{\Gamma_1 \vdash e : \tau \textbf{ ref}}{\Gamma_1 \vdash !e : \tau}\ \textbf{(ty-deref)}}{\cdot \vdash \textbf{letref } M_1 \textbf{ in } !e \textbf{ end} : \tau}\ \textbf{(ty-letref)}}$$

This leads to the following.

$$\frac{\cdot \vdash M_1 : \Gamma_1 \quad \Gamma_1 \vdash e_1 : \tau \textbf{ ref}}{\cdot \vdash \textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} : \tau \textbf{ ref}}\ \textbf{(ty-letref)}$$

By induction hypothesis, $\cdot \vdash \textbf{letref } M_2 \textbf{ in } x \textbf{ end} : \tau \textbf{ ref}$ is derivable. This implies $\cdot \vdash M_2 : \Gamma_2$ is derivable for some Γ_2 such that $\Gamma_2(x) = \tau \textbf{ ref}$. This then implies that $\Gamma_2 \vdash M_2(x) : \tau$ is derivable. Therefore, we have the following.

$$\frac{\cdot \vdash M_2 : \Gamma_2 \quad \Gamma_2 \vdash M_2(x) : \tau}{\cdot \vdash \textbf{letref } M_2 \textbf{ in } M_2(x) \textbf{ end} : \tau}\ \textbf{(ty-letref)}$$

The rest of the cases can be handled in a similar manner. ■

The next theorem generalizes Theorem 2.1.4.

Theorem 7.2.4 *We have* **letref** M **in** v **end** $\hookrightarrow_0$ **letref** M **in** v **end** *for all memory* M *and values* v *in* $\mathrm{ML}_{0,\mathrm{exc,ref}}$.

Proof This simply follows from a structural induction on v. ■

7.3 Value Restriction

We first mention some problems if we extend $\mathrm{ML}_{0,\mathrm{exc,ref}}$ with dependent types and/or polymorphism. Let us take a look at the following evaluation rules.

$$\frac{e \hookrightarrow_d v}{(\lambda a : \gamma.e) \hookrightarrow_d (\lambda a : \gamma.v)}\ \textbf{(ev-ilam)}$$

$$\frac{}{(\textbf{lam } x : \tau.e) \hookrightarrow_d (\textbf{lam } x : \tau.e)}\ \textbf{(ev-lam)}$$

$$\frac{e \hookrightarrow_d v}{\Lambda\alpha.e \hookrightarrow_d \Lambda\alpha.v}\ \textbf{(ev-poly)}$$

Clearly, evaluation can occur under both λ and Λ but cannot under **lam**. This can introduce a serious problem when we extend the language $\mathrm{ML}_0^{\forall,\Pi,\Sigma}(C)$ with effects such as exceptions and references. For instance, the following cases arise immediately.

1. If evaluation is allowed under λ, then the following rule must be adopted since an exception may be raised during the evaluation of e.

$$\frac{e \hookrightarrow_d \textbf{raise}(v)}{(\lambda a : \gamma.e) \hookrightarrow_d \textbf{raise}(v)}\ \textbf{(ev-ilam-raise)}$$

However, v may contain some free occurrences of a when this rule is applied.

2. Similarly, we must adopt the following rule if evaluation is allowed under λ.

$$\frac{\textbf{letref } M_1, M_2 \textbf{ in } \lambda a : \gamma.e \textbf{ end} \hookrightarrow_0 ans}{\textbf{letref } M_1 \textbf{ in } \lambda a : \gamma.\textbf{letref } M_2 \textbf{ in } e \textbf{ end end} \hookrightarrow_0 ans} \textbf{ (ev-ilam-extrusion)}$$

   However, M_2 may contain some free occurrences of a when this rule is applied.

3. If evaluation is allowed under Λ, then we need the following rule since an exception may be raised during the evaluation of e.

$$\frac{e \hookrightarrow_d \textbf{raise}(v)}{(\Lambda\alpha.e) \hookrightarrow_d \textbf{raise}(v)} \textbf{ (ev-poly-raise)}$$

   The problem is that v may contain some free occurrences of α.

4. Similarly, the following rule is also needed.

$$\frac{\textbf{letref } M_1, M_2 \textbf{ in } \Lambda\alpha.e \textbf{ end} \hookrightarrow_0 ans}{\textbf{letref } M_1 \textbf{ in } \Lambda\alpha.\textbf{letref } M_2 \textbf{ in } e \textbf{ end end} \hookrightarrow_0 ans} \textbf{ (ev-poly-extrusion)}$$

   The problem is that M_2 may contain some free occurrences of α.

In all of these cases, some bound variables become unbound after the evaluation. Clearly, this must be addressed if we extend $\mathrm{ML}_{0,\mathrm{exc,ref}}$ with let-polymorphism as well as dependent types.

A radical solution to *all* the problems above is to make sure that we never evaluate under either λ or Λ. In other words, we should adopt instead the following rules.

$$\frac{}{(\lambda a : \gamma.e) \hookrightarrow_d (\lambda a : \gamma.e)} \textbf{ (ev-ilam)} \qquad \frac{}{\Lambda\alpha.e \hookrightarrow_d \Lambda\alpha.e} \textbf{ (ev-poly)}$$

This seems to be a clean solution. Unfortunately, the adoption of the above rules immediately falsifies Theorem 6.2.7 and Theorem 6.2.8 for the obvious reason that neither $\|\lambda a : \gamma.e\|$ nor $\|\Lambda\alpha.e\|$ is a value if $\|e\|$ is not. In order to overcome this difficulty, we require that e be a value whenever either $\lambda a : \gamma.e$ or $\Lambda\alpha.e$ occurs in an expression. This can be achieved if we require $\|e\|$ to be a value when the following typing rules are applied.

$$\frac{\phi, a : \gamma; \Delta; \Gamma \vdash e : \tau}{\phi; \Delta; \Gamma \vdash (\lambda a : \gamma.e) : (\Pi a : \gamma.\tau)} \textbf{ (ty-ilam)}$$

$$\frac{\cdot; \Delta, \alpha; \Gamma \vdash e : \sigma}{\cdot; \Delta; \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\sigma} \textbf{ (ty-poly-intro)}$$

This is called *value restriction*. In other words, we should formulate the above rules as follows.

$$\frac{\phi, a : \gamma; \Delta; \Gamma \vdash v : \tau}{\phi; \Delta; \Gamma \vdash (\lambda a : \gamma.v) : (\Pi a : \gamma.\tau)} \textbf{ (ty-ilam)}$$

$$\frac{\cdot; \Delta, \alpha; \Gamma \vdash v : \sigma}{\cdot; \Delta; \Gamma \vdash \Lambda\alpha.v : \forall\alpha.\sigma} \textbf{ (ty-poly-intro)}$$

From now on, we always assume that value restriction is imposed unless it is stated otherwise explicitly.

7.4 Extending $\mathrm{ML}_{0,\mathrm{exc,ref}}$ with Polymorphism and Dependent Types

In this section, we extend $\mathrm{ML}_{0,\mathrm{exc,ref}}$ with let-polymorphism and dependent types, leading to the language $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$. Therefore, we have finally designed a language in which there are features such as references, exception mechanism, let-polymorphism and both universal and existential dependent types. Since the core of ML, that is ML without module level constructs, is basically $\mathrm{ML}_{0,\mathrm{exc,ref}}$ with let-polymorphism, we claim that we have presented a practical approach to extending the core of ML with dependent types. We regard this as the key contribution of the thesis.

The complete syntax of $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$ is given in Figure 7.5. The typing rules for $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$ are those presented in Figure 6.6 plus those in Figure 7.6. Also the natural semantics of $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$ is given in terms of the evaluation rules listed in Figure 7.3 and Figure 7.4 plus those in Figure 7.7.

Lemma 7.4.1 *(Substitution) We have the following.*

1. *If both* $\phi \vdash i : \gamma$ *and* $\phi, a : \gamma; \Delta; \Gamma \vdash e : \tau$ *are derivable, then* $\phi; \Delta; \Gamma[a \mapsto i] \vdash e[a \mapsto i] : \tau[a \mapsto i]$ *is also derivable.*

2. *If both* $\Delta \vdash \tau : *$ *and* $\phi; \Delta, \alpha; \Gamma \vdash e : \sigma$ *are derivable, then* $\phi; \Delta; \Gamma[\alpha \mapsto \tau] \vdash e[\alpha \mapsto \tau] : \sigma[\alpha \mapsto \tau]$ *is also derivable.*

3. *If both* $\phi; \Delta; \Gamma \vdash v : \sigma_1$ *and* $\phi; \Delta; \Gamma, x : \sigma_1 \vdash e : \sigma$ *are derivable, then* $\phi; \Delta; \Gamma \vdash e[x \mapsto v] : \sigma$ *is also derivable.*

Proof The proof is standard and therefore omitted here. Please see the proof of Lemma 4.1.4 for some relevant details. ∎

Theorem 7.4.2 *(Type preservation for* $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$*) If both* $e \hookrightarrow_d ans : \sigma$ *and* $\cdot;\cdot;\cdot \vdash e : \sigma$ *are derivable in* $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$*, then* $\cdot;\cdot;\cdot \vdash ans : \sigma$ *is also derivable* $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$.

Proof The proof follows from a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_d ans$ and the derivation of $\cdot;\cdot;\cdot \vdash e : \sigma$, lexicographically ordered. We present a few cases.

$$\mathcal{D} = \frac{\textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} \hookrightarrow_d \textbf{letref } M_2 \textbf{ in } \lambda a : \gamma.v \textbf{ end}}{\textbf{letref } M_1 \textbf{ in } e_1[i] \textbf{ end} \hookrightarrow_d \textbf{letref } M_2 \textbf{ in } v[a \mapsto i] \textbf{ end}}$$

Then we have a derivation of the following form since $\cdot;\cdot;\cdot \vdash \textbf{letref } M_1 \textbf{ in } e_1[i] \textbf{ end} : \sigma$ is derivable, where $\sigma = \tau[a \mapsto i]$.

$$\frac{\dfrac{\cdot;\cdot;\Gamma_1 \vdash e_1 : \Pi a : \gamma.\tau \quad \cdot \vdash i : \gamma}{\cdot;\cdot;\Gamma_1 \vdash e_1[i] : \tau[a \mapsto i]}\ (\textbf{ty-iapp}) \qquad \cdot;\cdot;\cdot \vdash M_1 : \Gamma_1}{\cdot;\cdot;\cdot \vdash \textbf{letref } M_1 \textbf{ in } e_1[i] \textbf{ end} : \tau[a \mapsto i]}\ (\textbf{ty-letref})$$

This yields the following derivation.

$$\frac{\cdot;\cdot;\Gamma_1 \vdash e_1 : \Pi a : \gamma.\tau \quad \cdot;\cdot;\cdot \vdash M_1 : \Gamma_1}{\cdot;\cdot;\cdot \vdash \textbf{letref } M_1 \textbf{ in } e_1 \textbf{ end} : \Pi a : \gamma.\tau}\ (\textbf{ty-letref})$$

families	δ	::=	(family of refined datatypes)
signatures	$\mathcal{S}$	::=	$\cdot_{\mathcal{S}} \mid \mathcal{S}, \delta : * \to \cdots \to * \to \gamma \to *$
			$\mid \mathcal{S}, c : \Lambda\alpha_1. \ldots \Lambda\alpha_m.\Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.(\alpha_1, \ldots, \alpha_m)\delta(i)$
			$\mid \mathcal{S}, c : \Lambda\alpha_1. \ldots \Lambda\alpha_m.\Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.\tau \to (\alpha_1, \ldots, \alpha_m)\delta(i)$
major types	μ	::=	$\alpha \mid (\alpha_1, \ldots, \alpha_m)\delta(i) \mid \mathbf{1} \mid (\tau_1 * \tau_2) \mid (\tau_1 \to \tau_2)$
types	τ	::=	$\mu \mid (\Pi a : \gamma.\tau) \mid (\Sigma a : \gamma.\tau)$
type schemes	σ	::=	$\tau \mid \Lambda\alpha.\sigma$
patterns	p	::=	$x \mid c(\alpha_1) \ldots (\alpha_m)[a_1] \ldots [a_n] \mid c(\alpha_1) \ldots (\alpha_m)[a_1] \ldots [a_n](p)$
			$\mid \langle\rangle \mid \langle p_1, p_2 \rangle$
matches	ms	::=	$(p \Rightarrow e) \mid (p \Rightarrow e \mid ms)$
expressions	e	::=	$x \mid \langle\rangle \mid \langle e_1, e_2 \rangle$
			$\mid c(\tau_1) \ldots (\tau_m)[i_1] \ldots [i_n] \mid c(\tau_1) \ldots (\tau_m)[i_1] \ldots [i_n](e)$
			$\mid (\mathbf{case}\ e\ \mathbf{of}\ ms) \mid (\mathbf{lam}\ x : \tau.e) \mid e_1(e_2)$
			$\mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \mid (\mathbf{fix}\ f : \tau.v)$
			$\mid \mathbf{raise}(e) \mid \mathbf{handle}\ e\ \mathbf{with}\ ms$
			$\mid e_1 := e_2 \mid !e$
			$\mid \mathbf{letref}\ M\ \mathbf{in}\ e\ \mathbf{end}$
			$\mid (\lambda a : \gamma.v) \mid e[i]$
			$\mid \langle i \mid e \rangle \mid \mathbf{let}\ \langle a \mid x \rangle = e_1\ \mathbf{in}\ e_2\ \mathbf{end}$
			$\mid \Lambda\alpha.v$
value forms	u	::=	$c(\tau_1) \ldots (\tau_m)[i_1] \ldots [i_n] \mid c(\tau_1) \ldots (\tau_m)[i_1] \ldots [i_n](u) \mid \langle\rangle$
			$\mid \langle u_1, u_2 \rangle \mid \mathbf{lam}\ x : \tau.e \mid (\lambda a : \gamma.u) \mid \langle i \mid u \rangle$
values	v	::=	$x(\tau_1) \ldots (\tau_m) \mid c(\tau_1) \ldots (\tau_m)[i_1] \ldots [i_n] \mid c(\tau_1) \ldots (\tau_m)[i_1] \ldots [i_n](v)$
			$\mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid (\mathbf{lam}\ x : \tau.e) \mid (\lambda a : \gamma.v) \mid \langle i \mid v \rangle \mid (\Lambda\alpha.v)$
memories	M	::=	$\cdot \mid M, x : \tau\ \mathbf{is}\ v$
programs	$prog$	::=	$\mathbf{letref}\ M\ \mathbf{in}\ e\ \mathbf{end}$
answers	ans	::=	$\mathbf{letref}\ M\ \mathbf{in}\ v\ \mathbf{end} \mid \mathbf{letref}\ M\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}$
contexts	Γ	::=	$\cdot \mid \Gamma, x : \sigma$
type var ctxts	Δ	::=	$\cdot \mid \Delta, \alpha$
index contexts	ϕ	::=	$\cdot \mid \phi, a : \gamma$
substitutions	θ	::=	$[] \mid \theta[x \mapsto v] \mid \theta[a \mapsto i] \mid \theta[\alpha \mapsto \tau]$

Figure 7.5: The syntax for $\mathrm{ML}_{0,\mathrm{exc},\mathrm{ref}}^{\forall,\Pi,\Sigma}(C)$

$$\frac{\phi;\Delta;\Gamma \vdash e : \tau \quad \phi;\Delta;\Gamma \vdash ms : \mathbf{Exc} \Rightarrow \tau}{\phi;\Delta;\Gamma \vdash (\mathbf{handle}\ e\ \mathbf{with}\ ms) : \tau}\ \textbf{(ty-handle)}$$

$$\frac{\phi;\Delta \vdash \tau : * \quad \phi;\Delta;\Gamma \vdash e : \mathbf{Exc}}{\phi;\Delta;\Gamma \vdash \mathbf{raise}(e) : \tau}\ \textbf{(ty-raise)}$$

$$\frac{\Gamma' = x_1 : \tau_1\ \mathbf{ref}, \ldots, x_n : \tau_n\ \mathbf{ref} \quad \phi;\Delta;\Gamma', \Gamma \vdash v_i : \tau_i \quad (1 \leq i \leq n)}{\phi;\Delta;\Gamma \vdash (x_1 : \tau_1\ \mathbf{is}\ v_1, \ldots, x_n : \tau_n\ \mathbf{is}\ v_n) : \Gamma'}\ \textbf{(ty-memo)}$$

$$\frac{\phi;\Delta;\Gamma \vdash M : \Gamma' \quad \phi;\Delta;\Gamma, \Gamma' \vdash e : \tau}{\phi;\Delta;\Gamma \vdash \mathbf{letref}\ M\ \mathbf{in}\ e\ \mathbf{end} : \tau}\ \textbf{(ty-letref)}$$

$$\frac{\phi;\Delta;\Gamma \vdash e_1 : \tau\ \mathbf{ref} \quad \phi;\Delta;\Gamma \vdash e_2 : \tau}{\phi;\Delta;\Gamma \vdash e_1 := e_2 : \mathbf{1}}\ \textbf{(ty-assign)}$$

$$\frac{\phi;\Delta;\Gamma \vdash e : \tau\ \mathbf{ref}}{\phi;\Delta;\Gamma \vdash\, !e : \tau}\ \textbf{(ty-deref)}$$

Figure 7.6: Additional typing rules for $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$

$$\frac{}{\mathbf{letref}\ M\ \mathbf{in}\ \lambda a : \gamma.v\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M\ \mathbf{in}\ \lambda a : \gamma.v\ \mathbf{end}}\ \textbf{(ev-ilam)}$$

$$\frac{\mathbf{letref}\ M_1\ \mathbf{in}\ e\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}}{\mathbf{letref}\ M_1\ \mathbf{in}\ e[i]\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}}\ \textbf{(ev-iapp-1)}$$

$$\frac{\mathbf{letref}\ M_1\ \mathbf{in}\ e\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ \lambda a : \gamma.v\ \mathbf{end}}{\mathbf{letref}\ M_1\ \mathbf{in}\ e[i]\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ v[a \mapsto i]\ \mathbf{end}}\ \textbf{(ev-iapp-2)}$$

$$\frac{\mathbf{letref}\ M_1\ \mathbf{in}\ e\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}}{\mathbf{letref}\ M_1\ \mathbf{in}\ \langle i \mid e\rangle\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}}\ \textbf{(ev-sig-intro-1)}$$

$$\frac{\mathbf{letref}\ M_1\ \mathbf{in}\ e\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ v\ \mathbf{end}}{\mathbf{letref}\ M_1\ \mathbf{in}\ \langle i \mid e\rangle\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ \langle i \mid v\rangle\ \mathbf{end}}\ \textbf{(ev-sig-intro-1)}$$

$$\frac{\mathbf{letref}\ M_1\ \mathbf{in}\ e_1\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}}{\mathbf{letref}\ M_1\ \mathbf{in}\ \mathbf{let}\ \langle a \mid x\rangle = e_1\ \mathbf{in}\ e_2\ \mathbf{end}\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}}\ \textbf{(ev-sig-elim-1)}$$

$$\frac{\begin{array}{c}\mathbf{letref}\ M_1\ \mathbf{in}\ e_1\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ \langle i \mid v\rangle\ \mathbf{end} \\ \mathbf{letref}\ M_2\ \mathbf{in}\ e_2[a \mapsto i][x \mapsto v]\ \mathbf{end} \hookrightarrow_d ans\end{array}}{\mathbf{letref}\ M_1\ \mathbf{in}\ \mathbf{let}\ \langle a \mid x\rangle = e_1\ \mathbf{in}\ e_2\ \mathbf{end}\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M_2\ \mathbf{in}\ ans\ \mathbf{end}}\ \textbf{(ev-sig-elim-2)}$$

$$\frac{}{\mathbf{letref}\ M\ \mathbf{in}\ \Lambda\alpha.v\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M\ \mathbf{in}\ \Lambda\alpha.v\ \mathbf{end}}\ \textbf{(ev-poly)}$$

Figure 7.7: Additional evaluation rules for $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$

By induction hypothesis, $\cdot;\cdot;\cdot \vdash \textbf{letref}\ M_2\ \textbf{in}\ \lambda a:\gamma.v\ \textbf{end} : \Pi a:\gamma.\tau$ is derivable. Therefore, we have a derivation of the following form.

$$\frac{\cdot;\cdot;\Gamma_2 \vdash \lambda a:\gamma.v : \Pi a:\gamma.\tau \quad \cdot;\cdot;\cdot \vdash M_2 : \Gamma_2}{\cdot;\cdot;\cdot \vdash \textbf{letref}\ M_2\ \textbf{in}\ \lambda a:\gamma.v\ \textbf{end} : \Pi a:\gamma.\tau}\ (\textbf{ty-letref})$$

By inversion, we can assume that $a:\gamma;\cdot;\Gamma_2 \vdash v : \tau$ is derivable. Therefore, by Lemma 7.4.1 (1), $\cdot;\cdot;\Gamma_2 \vdash v[a \mapsto i] : \tau[a \mapsto i]$ is derivable since a has no free occurrences in Γ_2.

This leads to the following derivation of $\cdot;\cdot;\cdot \vdash \textbf{letref}\ M_2\ \textbf{in}\ v[a \mapsto i]\ \textbf{end} : \tau[a \mapsto i]$.

$$\frac{\cdot;\cdot;\Gamma_2 \vdash v[a \mapsto i] : \tau[a \mapsto i] \quad \cdot;\cdot;\cdot \vdash M_2 : \Gamma_2}{\cdot;\cdot;\cdot \vdash \textbf{letref}\ M_2\ \textbf{in}\ v[a \mapsto i]\ \textbf{end} : \tau[a \mapsto i]}\ (\textbf{ty-letref})$$

$$\mathcal{D} = \frac{\begin{array}{c}\textbf{letref}\ M_1\ \textbf{in}\ e_1\ \textbf{end} \hookrightarrow_d \textbf{letref}\ M_2\ \textbf{in}\ \langle i \mid v\rangle\ \textbf{end} \\ \textbf{letref}\ M_2\ \textbf{in}\ e_2[a \mapsto i][x \mapsto v]\ \textbf{end} \hookrightarrow_d ans\end{array}}{\textbf{letref}\ M_1\ \textbf{in let}\ \langle a \mid x\rangle = e_1\ \textbf{in}\ e_2\ \textbf{end end} \hookrightarrow_d \textbf{letref}\ M_2\ \textbf{in}\ ans\ \textbf{end}}$$ Then we have a derivation of the following form.

$$\frac{\dfrac{\cdot;\cdot;\Gamma_1 \vdash e_1 : \Sigma a:\gamma.\tau \quad a:\gamma;\cdot;\Gamma_1, x:\tau \vdash e_2 : \sigma}{\cdot;\cdot;\Gamma_1 \vdash \textbf{let}\ \langle a \mid x\rangle = e_1\ \textbf{in}\ e_2\ \textbf{end} : \sigma}\ (\textbf{ty-sig-elim}) \quad \cdot;\cdot;\cdot \vdash M_1 : \Gamma_1}{\cdot;\cdot;\cdot \vdash \textbf{letref}\ M_1\ \textbf{in let}\ \langle a \mid x\rangle = e_1\ \textbf{in}\ e_2\ \textbf{end} : \sigma\ \textbf{end}}\ (\textbf{ty-letref})$$

This yields the following derivation.

$$\frac{\cdot;\cdot;\Gamma_1 \vdash e_1 : \Sigma a:\gamma.\tau \quad \cdot;\cdot;\cdot \vdash M_1 : \Gamma_1}{\cdot;\cdot;\cdot \vdash \textbf{letref}\ M_1\ \textbf{in}\ e_1\ \textbf{end} : \Sigma a:\gamma.\tau}\ (\textbf{ty-letref})$$

By induction hypothesis, $\cdot;\cdot;\cdot \vdash \textbf{letref}\ M_2\ \textbf{in}\ \langle i \mid v\rangle\ \textbf{end} : \Sigma a:\gamma.\tau$ is derivable. Therefore, we have a derivation of the following form.

$$\frac{\dfrac{\cdot;\cdot;\Gamma_2 \vdash v : \tau[a \mapsto i] \quad \phi \vdash i : \gamma}{\cdot;\cdot;\Gamma_2 \vdash \langle i \mid v\rangle : \Sigma a:\gamma.\tau}\ (\textbf{ty-sig-intro}) \quad \cdot;\cdot;\cdot \vdash M_2 : \Gamma_2}{\cdot;\cdot;\cdot \vdash \textbf{letref}\ M_2\ \textbf{in}\ \langle i \mid v\rangle\ \textbf{end} : \Sigma a:\gamma.\tau}\ (\textbf{ty-letref})$$

Note that $\cdot;\cdot;\Gamma_1 \vdash e_2[a \mapsto i][x \mapsto v] : \sigma$ is also derivable by Lemma 7.4.1. This leads to the following.

$$\frac{\cdot;\cdot;\Gamma_2 \vdash e_2[a \mapsto i][x \mapsto v] : \sigma \quad \cdot;\cdot;\cdot \vdash M_2 : \Gamma_2}{\cdot;\cdot;\cdot \vdash \textbf{letref}\ M_2\ \textbf{in}\ e_2[a \mapsto i][x \mapsto v]\ \textbf{end} : \sigma}\ (\textbf{ty-letref})$$

By induction hypothesis, *ans* is of type σ.

All other cases can be dealt with in a similar manner. ∎

Given $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$, it is straightforward to form the language $\mathrm{ML}^{\forall}_{0,\mathrm{exc,ref}}$, which extends $\mathrm{ML}_{0,\mathrm{exc,ref}}$ with let-polymorphism. Note that value restriction is also imposed to guarantee the soundness of the type system of $\mathrm{ML}^{\forall}_{0,\mathrm{exc,ref}}$. We leave the details for the interested reader.

We now extend the definition of the index erasure function as follows.

$$\begin{array}{lcl}
\|\cdot\| & = & \cdot \\
\|\mathbf{raise}(e)\| & = & \mathbf{raise}(\|e\|) \\
\|\mathbf{handle}\ e\ \mathbf{with}\ ms\| & = & \mathbf{handle}\ \|e\|\ \mathbf{with}\ \|ms\| \\
\|M, x\ \mathbf{is}\ v\| & = & \|M\|, x\ \mathbf{is}\ \|v\| \\
\|\mathbf{letref}\ M\ \mathbf{in}\ e\ \mathbf{end}\| = & = & \mathbf{letref}\ \|M\|\ \mathbf{in}\ \|e\|\ \mathbf{end}
\end{array}$$

Theorem 7.4.3 *Suppose that* $\cdot;\cdot;\cdot \vdash e : \sigma$ *is derivable in* $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$. *If* $e \hookrightarrow_d ans$ *is also derivable in* $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$, *then* $\|e\| \hookrightarrow_0 \|ans\|$ *is derivable in* $\mathrm{ML}^{\forall}_{0,\mathrm{exc,ref}}$.

Proof This follows from a structural induction on the derivation $\mathcal{D}$ of $e \hookrightarrow_d ans$ and the derivation of $\cdot;\cdot;\cdot \vdash e : \sigma$, lexicographically ordered. We present a few cases.

$\mathcal{D} = \dfrac{}{\mathbf{letref}\ M\ \mathbf{in}\ \lambda a : \gamma.v\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M\ \mathbf{in}\ \lambda a : \gamma.v\ \mathbf{end}}$ Notice that we have the following.

$$\|\mathbf{letref}\ M\ \mathbf{in}\ \lambda a : \gamma.v\ \mathbf{end}\| = \mathbf{letref}\ \|M\|\ \mathbf{in}\ \|v\|\ \mathbf{end}.$$

By Proposition 7.2.4, we have

$$\mathbf{letref}\ \|M\|\ \mathbf{in}\ \|v\|\ \mathbf{end} \hookrightarrow_0 \mathbf{letref}\ \|M\|\ \mathbf{in}\ \|v\|\ \mathbf{end}$$

since $\|v\|$ is obviously a value.

$\mathcal{D} = \dfrac{}{\mathbf{letref}\ M\ \mathbf{in}\ \Lambda\alpha.v\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M\ \mathbf{in}\ \Lambda\alpha.v\ \mathbf{end}}$ Notice that we have the following.

$$\|\mathbf{letref}\ M\ \mathbf{in}\ \Lambda\alpha.v\ \mathbf{end}\| = \mathbf{letref}\ \|M\|\ \mathbf{in}\ \|v\|\ \mathbf{end}.$$

By Proposition 7.2.4, we have

$$\mathbf{letref}\ \|M\|\ \mathbf{in}\ \|v\|\ \mathbf{end} \hookrightarrow_0 \mathbf{letref}\ \|M\|\ \mathbf{in}\ \|v\|\ \mathbf{end}$$

since $\|v\|$ is obviously a value.

All other cases can be treated as done in the proof of Theorem 6.1.3. ∎

Suppose that we formulate a reduction semantics for $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$. Then a legitimate question to ask is whether an expression of form $\Lambda\alpha.e$ $(\lambda a : \gamma.e)$ for some non-value e can be generated during the reduction of a program p in which there are no such expressions. The answer is negative since $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$ is a call-by-value language. Therefore, not surprisingly, a type preservation theorem for $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$ can also be formulated and proven using reduction semantics. Usually, such a theorem is called *subject reduction* theorem. We leave the details for the interested reader.

Theorem 7.4.4 *Suppose that* $\cdot;\cdot;\cdot \vdash e : \sigma$ *is derivable in* $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$. *If* $\|e\| \hookrightarrow_0 ans_0$ *is derivable in* $\mathrm{ML}^{\forall}_{0,\mathrm{exc,ref}}$, *then* $e \hookrightarrow_d ans$ *is also derivable in* $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$ *for some ans such that* $\|ans\| = ans_0$.

Proof The proof proceeds by a structural induction on the derivation $\mathcal{D}_0$ of $\|e\| \hookrightarrow_0 ans_0$ and the derivation $\mathcal{D}$ of $\cdot;\cdot;\cdot \vdash e : \sigma$, lexicographically ordered. We present one case.

$$\mathcal{D} = \frac{\cdot;\cdot;\cdot \vdash e_0 : \tau \qquad \cdot;\cdot;\cdot \vdash ms : \mathbf{Exc} \Rightarrow \tau}{\cdot;\cdot;\cdot \vdash (\mathbf{handle}\ e_0\ \mathbf{with}\ ms) : \tau}$$ Then we have

$$\|e\| = \|\mathbf{handle}\ e_0\ \mathbf{with}\ ms\| = \mathbf{handle}\ \|e_0\|\ \mathbf{with}\ \|ms\|.$$

The derivation $\mathcal{D}_0$ of $\|e\| \hookrightarrow_0 ans_0$ must be one of the following forms.

$$\mathcal{D}_0 = \frac{\mathbf{letref}\ \cdot\ \mathbf{in}\ \|e_0\|\ \mathbf{end} \hookrightarrow_0 \mathbf{letref}\ M^0\ \mathbf{in}\ \mathbf{raise}(v^0)\ \mathbf{end}}{\mathbf{letref}\ \cdot\ \mathbf{in}\ \mathbf{handle}\ \|e_0\|\ \mathbf{with}\ \|ms\|\ \mathbf{end} \hookrightarrow_0 \mathbf{letref}\ M^0\ \mathbf{in}\ \mathbf{raise}(v^0)\ \mathbf{end}}$$ By induction hypothesis, $\mathbf{letref}\ \cdot\ \mathbf{in}\ e_0\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}$ is derivable for some M and v such that $\|M\| = M^0$ and $\|v\| = v^0$. This leads to the following.

$$\frac{\mathbf{letref}\ \cdot\ \mathbf{in}\ e_0\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}}{\mathbf{letref}\ \cdot\ \mathbf{in}\ \mathbf{handle}\ e_0\ \mathbf{with}\ ms\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}}\ \textbf{(ev-handle-1)}$$

Hence, we are done.

$$\mathcal{D}_0 = \frac{\begin{array}{c}\mathbf{letref}\ \cdot\ \mathbf{in}\ \|e_0\|\ \mathbf{end} \hookrightarrow_0 \mathbf{letref}\ M^0\ \mathbf{in}\ \mathbf{raise}(v^0)\ \mathbf{end} \\ \mathbf{match}(v^0, \|p_k\|) \Longrightarrow \theta_0 \text{ for some } 1 \le k \le n \\ \mathbf{letref}\ M^0\ \mathbf{in}\ \|e_k\|[\theta_0]\ \mathbf{end} \hookrightarrow_0 ans_0\end{array}}{\mathbf{letref}\ \cdot\ \mathbf{in}\ \mathbf{handle}\ \|e_0\|\ \mathbf{with}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)\ \mathbf{end} \hookrightarrow_0 ans_0}$$ By induction hypothesis, $\mathbf{letref}\ \cdot\ \mathbf{in}\ e_0\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end}$ is derivable for some M and v such that $\|M\| = M^0$ and $\|v\| = v^0$. By Theorem 7.4.2, $\cdot;\cdot;\cdot \vdash v : \sigma$ is derivable. By Proposition 6.2.1, $\mathbf{match}(v, p_k) \Longrightarrow \theta$ is derivable for some θ such that $\|\theta\| = \theta_0$, and therefore, $\|e_k[\theta]\| = \|e_k\|[\theta_0]$. By induction hypothesis, $\mathbf{letref}\ M\ \mathbf{in}\ e_k[\theta]\ \mathbf{end} \hookrightarrow_d ans$ for some ans such that $\|ans\| = ans_0$. This leads to the following.

$$\frac{\begin{array}{c}\mathbf{letref}\ \cdot\ \mathbf{in}\ e_0\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M\ \mathbf{in}\ \mathbf{raise}(v)\ \mathbf{end} \\ \mathbf{match}(v, p_k) \Longrightarrow \theta \text{ for some } 1 \le k \le n \\ \mathbf{letref}\ M\ \mathbf{in}\ e_k[\theta]\ \mathbf{end} \hookrightarrow_d ans\end{array}}{\mathbf{letref}\ \cdot\ \mathbf{in}\ \mathbf{handle}\ e_0\ \mathbf{with}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)\ \mathbf{end} \hookrightarrow_d ans}\ \textbf{(ev-handle-2)}$$

This concludes the subcase.

$$\mathcal{D}_0 = \frac{\mathbf{letref}\ \cdot\ \mathbf{in}\ \|e_0\|\ \mathbf{end} \hookrightarrow_0 \mathbf{letref}\ M^0\ \mathbf{in}\ v^0\ \mathbf{end}}{\mathbf{letref}\ \cdot\ \mathbf{in}\ \mathbf{handle}\ \|e_0\|\ \mathbf{with}\ \|ms\|\ \mathbf{end} \hookrightarrow_0 \mathbf{letref}\ M^0\ \mathbf{in}\ v^0\ \mathbf{end}}$$ By induction hypothesis, $\mathbf{letref}\ \cdot\ \mathbf{in}\ e_0\ \mathbf{end} \hookrightarrow_d \mathbf{letref}\ M\ \mathbf{in}\ v\ \mathbf{end}$ is derivable for some M and v such that $\|M\| = M^0$ and $\|v\| = v^0$. This leads to the following.

$$\frac{\mathbf{letref}\ \cdot\ \mathbf{in}\ e_0\ \mathbf{end} \hookrightarrow_0 \mathbf{letref}\ M\ \mathbf{in}\ v\ \mathbf{end}}{\mathbf{letref}\ \cdot\ \mathbf{in}\ \mathbf{handle}\ e_0\ \mathbf{with}\ ms\ \mathbf{end} \hookrightarrow_0 \mathbf{letref}\ M\ \mathbf{in}\ v\ \mathbf{end}}\ \textbf{(ev-handle-3)}$$

Hence, we are done.

All other cases can be treated similarly.

■

We have thus extended the entire core of ML with dependent types. Given the comprehensive features of the core of ML, this really is a solid justification on the feasibility of our approach to making dependent types available in practical programming. Naturally, the next move is to enrich the module system of ML with dependent types, which we regard as a primary future research topic.

7.5 Elaboration

We briefly explain how elaboration for $\mathrm{ML}_{0,\mathrm{exc,ref}}^{\forall,\Pi,\Sigma}(C)$ is performed. We concentrate on the newly introduced language constructs rather than present all the elaboration rules as done for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$, which is simply too overwhelming in this case. We also ignore type variables since polymorphism is large orthogonal to dependent types as explained in Chapter 6.

The elaboration rules for references and exception mechanism are listed in Figure 7.8. We omit the formulation of the corresponding constraint generation rules. Also it is a routine to formulate and prove a similar version of Theorem 5.2.6 for $\mathrm{ML}_{0,\mathrm{exc,ref}}^{\forall,\Pi,\Sigma}(C)$, which justifies the correctness of these elaboration rules. We leave out details since we have adequately presented in the previous chapters the techniques needed for fulfilling such a task.

7.6 Summary

In this chapter we studied the interactions between dependent types and effects such as references and exception mechanism. Like polymorphism, dependent types cannot be combined with effects directly for the type system would be unsound otherwise. A clean solution to this problem is to adopt value restriction on formulating expressions of dependent function types. The development seems to be straightforward after this adoption. However, this problem also exhibits another inadequate aspect of the type system of ML for it cannot distinguish the functions which have effects from those which do not. It will be interesting to see how this can be remedied in future research.

The type system of $\mathrm{ML}_{0,\mathrm{exc,ref}}^{\forall,\Pi,\Sigma}(C)$, which includes let-polymorphism, effects and dependent types, has reached the stage where it is difficult to manipulate without mechanical assistance. For instance, we presented only one case in the proof of Theorem 7.4.4, and left out dozens. Since almost all the proofs in this thesis are based on some sort of structural induction, it seems highly relevant to investigate whether an interactive theorem prover with certain automation features can accomplish the task of fulfilling the cases that we omitted. The interested reader can find some related research in (Schümann and Pfenning 1998).

$$\frac{\phi \vdash \tau : * \quad \phi; \Gamma \vdash e \downarrow \mathbf{Exc} \Rightarrow e^*}{\phi; \Gamma \vdash \mathbf{raise}(e) \downarrow \tau \Rightarrow \mathbf{raise}(e^*)} \ \textbf{(elab-raise)}$$

$$\frac{\phi; \Gamma \vdash e \uparrow \tau \Rightarrow e^* \quad \phi; \Gamma \vdash ms \downarrow (\mathbf{Exc} \Rightarrow \tau) \Rightarrow ms^*}{\phi; \Gamma \vdash (\mathbf{handle}\ e\ \mathbf{with}\ ms) \uparrow \tau \Rightarrow (\mathbf{handle}\ e^*\ \mathbf{with}\ ms^*)} \ \textbf{(elab-handle-up)}$$

$$\frac{\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^* \quad \phi; \Gamma \vdash ms \downarrow (\mathbf{Exc} \Rightarrow \tau) \Rightarrow ms^*}{\phi; \Gamma \vdash (\mathbf{handle}\ e\ \mathbf{with}\ ms) \downarrow \tau \Rightarrow (\mathbf{handle}\ e^*\ \mathbf{with}\ ms^*)} \ \textbf{(elab-handle-down)}$$

$$\frac{\phi; \Gamma \vdash e \uparrow \tau \Rightarrow e^*}{\phi; \Gamma \vdash \mathbf{Ref}(e) \uparrow \tau\ \mathbf{ref} \Rightarrow \mathbf{Ref}(e^*)} \ \textbf{(elab-ref-up)}$$

$$\frac{\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi; \Gamma \vdash \mathbf{Ref}(e) \downarrow \tau\ \mathbf{ref} \Rightarrow \mathbf{Ref}(e^*)} \ \textbf{(elab-ref-down)}$$

$$\frac{\phi; \Gamma \vdash e \uparrow \tau\ \mathbf{ref} \Rightarrow e^*}{\phi; \Gamma \vdash !e \uparrow \tau \Rightarrow !e^*} \ \textbf{(elab-deref-up)}$$

$$\frac{\phi; \Gamma \vdash e \downarrow \tau\ \mathbf{ref} \Rightarrow e^*}{\phi; \Gamma \vdash !e \downarrow \tau \Rightarrow !e^*} \ \textbf{(elab-deref-down)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau\ \mathbf{ref} \Rightarrow e_1^* \quad \phi; \Gamma \vdash e_2 \downarrow \tau \Rightarrow e_2^*}{\phi; \Gamma \vdash e_1 := e_2 \uparrow \mathbf{1} \Rightarrow e_1^* := e_2^*} \ \textbf{(elab-assign-up)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau\ \mathbf{ref} \Rightarrow e_1^* \quad \phi; \Gamma \vdash e_2 \downarrow \tau \Rightarrow e_2^*}{\phi; \Gamma \vdash e_1 := e_2 \downarrow \mathbf{1} \Rightarrow e_1^* := e_2^*} \ \textbf{(elab-assign-down)}$$

Figure 7.8: Some elaboration rules for references and exception mechanism

Chapter 8

Implementation

We have finished a prototype implementation of dependent type inference in Standard ML of New Jersey, version 110. The implementation corresponds closely to the theory developed in the previous chapters. All the examples presented in Appendix A have been verified in this implementation.

In this chapter, we account for some decisions we made during this implementation. However, this chapter is not meant to be complete instructions for using the prototype implementation. The syntax for the expressions recognized by the implementation is similar to that of the external language DML(C) for $\mathrm{ML}_{0,\mathrm{exc,ref}}^{\forall,\Pi,\Sigma}(C)$, including let-polymorphism, references, exception mechanism, universal and existential dependent types. The record types, which can be regarded as a sugared version of product types, are not available at this moment. Most of the features can be found in the examples presented in Appendix A.

The grammar for a sugared version of DML(C) closely resembles that of Standard ML in the sense that a DML(C) program becomes an SML one if all syntax related to type index objects is erased. Therefore, we will only briefly go over the syntax related to dependent types. Also note that the explanation will be given in an informal way since most of the syntax for DML(C) is likely to change in future implementations.

Lastly, we will move on to mention some issues on implementing the elaboration algorithm presented in Chapter 5.

8.1 Refinement of Built-in Types

We have refined the built-in types `int`, `bool` and `'a array` in ML as follows.

- `int` is refined into infinitely many singleton types `int`(n), where n are of integer values. In other words, if a value v is of type `int`(n) for some n, then v is equal to n. As a consequence, `int` becomes a shorthand for $\Sigma n : int.$`int`(n).

- `bool` is refined into two singleton types `bool`(b), where b is either $\top$ or $\bot$. $true$ and $false$ are assigned types `bool`($\top$) and `bool`($\bot$) respectively. As a consequence, `bool` is a shorthand for $\Sigma b : o.$`bool`(b).

- `'a array`is refined into infinitely many dependent types `'a array(n)` where `n` stands for the size of the array.

$$
\begin{array}{rcl}
+ & : & \Pi m : int.\Pi n : int.int(m) * int(n) \rightarrow int(m+n) \\
- & : & \Pi m : int.\Pi n : int.int(m) * int(n) \rightarrow int(m-n) \\
\times & : & \Pi m : int.\Pi n : int.int(m) * int(n) \rightarrow int(m*n) \\
\div & : & \Pi m : int.\Pi n : int.int(m) * int(n) \rightarrow int(div(m,n)) \\
\% & : & \Pi m : int.\Pi n : int.int(m) * int(n) \rightarrow int(mod(m,n)) \\
< & : & \Pi m : int.\Pi n : int.int(m) * int(n) \rightarrow bool(m<n) \\
\leq & : & \Pi m : int.\Pi n : int.int(m) * int(n) \rightarrow bool(m \leq n) \\
= & : & \Pi m : int.\Pi n : int.int(m) * int(n) \rightarrow bool(m=n) \\
> & : & \Pi m : int.\Pi n : int.int(m) * int(n) \rightarrow bool(m>n) \\
\geq & : & \Pi m : int.\Pi n : int.int(m) * int(n) \rightarrow bool(m \geq n) \\
array & : & \Lambda\alpha.\Pi n : nat.\alpha * int(n) \rightarrow (\alpha)array(n) \\
length & : & \Lambda\alpha.\Pi n : nat.(\alpha)array(n) \rightarrow int(n)
\end{array}
$$

Figure 8.1: Dependent types for some built-in functions

Also we have assigned dependent types to some built-in functions on integers, booleans and arrays.

8.2 Refinement of Datatypes

During the development of various dependent type systems in previous chapters, we implicitly assumed that a declared (polymorphic) datatype constructor $\delta : * \rightarrow \cdots \rightarrow * \rightarrow *$ in ML can be refined into a dependent datatype constructor $\delta : * \rightarrow \cdots \rightarrow * \rightarrow \gamma \rightarrow *$ for some index sort γ, and every constructor c associated with δ of type $\Lambda\alpha_1.\cdots.\Lambda\alpha_m.\tau \rightarrow \delta$ is then assigned a dependent type of form

$$\Lambda\alpha_1 \ldots \Lambda\alpha_m.\Pi a_1 : \gamma. \ldots \Pi a_n : \gamma_n.\tau \rightarrow (\alpha_1, \ldots, \alpha_m)\delta(a_1, \ldots, a_n),$$

where $\gamma_1 * \cdots * \gamma_n = \gamma$. We now use an example to illustrate how a datatype refinement declaration is formulated in the implementation.

Given the datatype constructor *tree* as follows,

```
datatype 'a tree = Leaf | Branch of 'a * 'a tree * 'a tree
```

the following is a datatype refinement declaration for *tree*.

```
typeref 'a tree of nat with
  Leaf <| 'a tree(0)
| Branch <|
  {sl:nat, sr:nat} 'a * 'a tree(sl) * 'a tree(sr) -> 'a tree(1+sl+sr)
```

This declaration states that the datatype constructor $tree : * \rightarrow *$ has been refined into a dependent datatype constructor $tree : * \rightarrow nat \rightarrow *$. Also the associated constructors $Leaf$ and $Branch$ have

been assigned the following types, respectively.

$$\Lambda\alpha.(\alpha)tree(0) \quad \text{and} \quad \Lambda\alpha.\Pi sl : nat.\Pi sr : nat.\alpha * (\alpha)tree(sl) * (\alpha)tree(sr) \rightarrow (\alpha)tree(1 + sl + sr)$$

Clearly, the meaning of the type index i in $(\alpha)tree(i)$ is the size of the tree. If one would like to index a *tree* with its height, then the following declaration suffices.

```
typeref 'a tree of nat with
  Leaf <| 'a tree(0)
| Branch <|
  {hl:nat, hr:nat} 'a * 'a tree(sl) * 'a tree(sr) -> 'a tree(1+max(hl, hr))
```

Moreover, if one would like to index a *tree* with both its size and its height, then the declaration can be written as follows.

```
typeref 'a tree of nat * nat with
  Leaf <| 'a tree(0, 0)
| Branch <|
  {{sl:nat, sr:nat, hl:nat, hr:nat}
  'a * 'a tree(sl, hl) * 'a tree(sr, hr) -> 'a tree(1+sl+sr, 1+max(hl, hr))
```

More sophisticated datatype refinement declarations can be found in the examples presented in Appendix A. Note that a datatype can be refined at most once in the current implementation for the sake of simplicity.

8.3 Type Annotations

The constraint generation rules for elaboration presented in Chapter 5 require that the programmer supply adequate type annotations. Roughly speaking, the dependent types of declared function should be determined by the programmer rather than synthesized during elaboration. The main reason for this is that, unlike in ML, there exists no notion of principal types in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$.

The type annotation for a function can be supplied through the use of a `where` clause following the function declaration. Suppose that the following datatype refinement has been declared.

```
datatype 'a list = nil | cons of 'a * 'a list

typeref 'a list of nat with
  nil <| 'a list(0)
| cons <| {n:nat} 'a * 'a list(n) -> 'a * 'a list(n+1)
```

Then the following function declaration contains a type annotation for the declared function *reverse*.

```
fun('a)
    reverse(nil) = nil
  | reverse(cons(x, xs)) = reverse(xs) @ cons(x, nil)
where reverse <| {n:nat} 'a list(n) -> 'a list(n)
```

The type annotation states that the *reverse* is a function of type $\Pi n : nat.(\alpha)list(n) \rightarrow (\alpha)list(n)$. The above declaration roughly corresponds to the following expression in DML(C).

$$\Lambda\alpha.\textbf{fix}\ reverse : \Pi n : nat.(\alpha)list(n) \rightarrow (\alpha)list(n).$$
$$\lambda n.\textbf{lam}\ l.\textbf{case}\ l\ \textbf{of}\ nil \Rightarrow nil \mid cons(\langle x, xs\rangle) \Rightarrow reverse(xs)@cons(\langle x, nil\rangle)$$

There is another form of type annotation shown in the following example, which is a slight variant of the example in Figure 1.1.

```
fun('a){n:nat}
    reverse(l) =
      let
          fun rev(nil, ys) = ys
            | rev(cons(x, xs), ys) = rev(xs, cons(x, ys))
          where rev <| {m:nat}{n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
      in rev(l, nil) end
where reverse <| 'a list(n) -> 'a list(n)
```

reverse is now defined in the tail-recursive style. Notice that `{n:nat}` follows `fun('a)` in this declaration, which corresponds to the following expression in DML(C).

$$\Lambda\alpha.\lambda n : nat.\textbf{fix}\ reverse : (\alpha)list(n) \rightarrow (\alpha)list(n).$$
$$\textbf{let}\ rev = \textbf{fix}\ rev : \Pi m : nat.\Pi n : nat.(\alpha)list(m) * (\alpha)list(n) \rightarrow (\alpha)list(m+n).$$
$$\lambda m.\lambda n.\textbf{lam}\ l.$$
$$\textbf{case}\ l\ \textbf{of}\ \langle nil, ys\rangle \Rightarrow ys$$
$$\mid\ \langle cons(\langle x, xs\rangle), ys\rangle \Rightarrow rev(\langle xs, cons(\langle x, ys\rangle)\rangle)$$
$$\textbf{in}\ rev(\langle l, nil\rangle)\ \textbf{end}$$

Another kind of type annotation is essentially like the type annotation in ML except that `<|` is used instead of `:` and a dependent type is supplied. For instance, the type annotation in the following code, extracted from the example in Section A.5, captures the relation between *front* and *srcalign*.

```
fun{srcalign:int}
aligned(src, srcpos, endsrc, dest, destpos, srcalign, bytes) =
   let
      val front =
          (case srcalign of
               0 => 0
             | 1 => 3
             | 2 => 2
             | 3 => 1) <| [i:nat |
                           (srcalign = 0 /\ i = 0) \/
                           (srcalign = 1 /\ i = 3) \/
                           (srcalign = 2 /\ i = 2) \/
                           (srcalign = 3 /\ i = 1) ] int(i)
```

```
    ...
in
    ...
end
```

We list as follows some less common syntax in the implementation and its corresponding part in $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$, helping the reader to understand examples.

implementation	$\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc,ref}}(C)$
e `<|` τ	$e : \tau$
P_1 `/\` P_2	$P_1 \wedge P_2$
P_1 `\/` P_2	$P_1 \vee P_2$
$\{a_1 : \gamma_1, \ldots, a_n : \gamma_n\}$	$\forall a_1 : \gamma_1 \ldots \forall a_n : \gamma_n.$
$\{a_1 : \gamma_1, \ldots, a_n : \gamma_n \mid P\}$	$\forall a_1 : \gamma_1 \ldots \forall a_n : \{a : \gamma_n \mid P[a_n \mapsto a]\}.$
$[a_1 : \gamma_1, \ldots, a_n : \gamma_n]$	$\exists a_1 : \gamma_1 \ldots \exists a_n : \gamma_n.$
$[a_1 : \gamma_1, \ldots, a_n : \gamma_n \mid P]$	$\exists a_1 : \gamma_1 \ldots \exists a_n : \{a : \gamma_n \mid P[a_n \mapsto a]\}.$

8.4 Program Transformation

There is a significant issue on whether a variant of A-normal transform should be performed on programs before they are elaborated. The advantage of doing the transform is that a common form of expressions are then able to be elaborated which would otherwise not be possible. However, the transform also prevents us from elaborating a less common form of expressions. This drawback, however, can be largely remedied by define $\underline{e_1(e_2)}$ as follows.

$$\underline{e_1(e_2)} = \begin{cases} \textbf{let } x_1 = \underline{e_1} \textbf{ in } x_1(e_2) \textbf{ end} & \text{if } e_2 \text{ is a value;} \\ \textbf{let } x_1 = \underline{e_1} \textbf{ in let } x_2 = \underline{e_2} \textbf{ in } x_1(x_2) \textbf{ end end} & \text{otherwise.} \end{cases}$$

A more serious disadvantage of performing the transform is that it can significantly complicate for the programmer the issue of understanding the error messages reported during elaboration since he or she may have to understand how the programs are transformed.

The transform *is performed* in the current prototype implementation. Since little attention is paid to reporting error messages in this implementation, the issue has yet to be addressed in future implementations. We would also like to study the feasibility of allowing the programmer to guide the transform with some syntax.

8.5 Indeterminacy in Elaboration

The constraint generation rules for coercion as presented in Figure 5.2 contain a certain amount of indeterminacy. Since we disallow backtracking in elaboration for the sake of practicality, we have imposed the following precedence on the application of these rules, that is, the rule with a higher precedence is chosen over the other if both of them are applicable.

$$\textbf{(coerce-pi-r)} > \textbf{(coerce-sig-l)} > \textbf{(coerce-pi-l)} > \textbf{(coerce-sig-r)}$$

Given $\phi \vdash \tau : *$ derivable, the above strategy guarantees that $\phi \vdash [\cdot]$ **coerce**$(\tau, \tau) \Rightarrow \Phi$ is derivable for some Φ such that $\phi \models \Phi$ is derivable. However, the programmer must use language constructs to guide coercion, sometimes. For instance, given a function f of type $\tau_1 = \Pi a : \gamma.\delta(a) \to \delta(i)$, one can define g as **lam** x.**let** $y = x$ **in** $f(y)$ **end** and assign it the type $\tau_2 = (\Sigma a : \gamma.\delta(a)) \to (\Sigma a : \gamma.\delta(a))$. This type-checks. Notice that it could not have succeeded with the precedence above if we had coerced τ_1 into τ_2 directly.

Similarly, the rule **(crule-pi-intro-1)** is always chosen over **(crule-pi-intro-2)** if both are applicable. There is yet another issue. Suppose that we have synthesized the type τ of an expression e for $\tau = \Pi a : \gamma.\tau_1$. Clearly, the rule **(crule-pi-elim)** is applicable now. Should we apply the rule? In the implementation, we apply the rule only if e occurs as a subexpression of $e(e')$ or **case** e **of** ms.

This pretty much summarizes how indeterminacy in elaboration is dealt with in the prototype implementation.

8.6 Summary

We have finished a prototype implementation in which there are features such as datatype declarations, high-order functions, let-polymorphism, references, exception mechanism, and both universal and existential dependent types. The only missing main feature in the core of ML is *records*, which can be regarded as a variant of product. The implementation sticks tightly to the theory developed in the previous chapters.

In the implementation of the elaboration described in Section 5.2, we have to cope with some indeterminacy in the constraint generation rules for elaboration and coercion. The important decision we adopt is that we disallow the use of backtracking in type-checking. The main reason for this decision is that backtracking can not only significantly slow down type-checking but also make it almost impossible to report type-error messages in an acceptable manner. We are now ready to harvest the fruit of our hard labor, mentioning some interesting applications of dependent types in the next chapter.

Chapter 9

Applications

In this chapter, we present some concrete examples to demonstrate various applications of dependent types in practical programming. All the examples in Section 9.1 and Section 9.2 have been verified in the prototype implementation. The ones in Section 9.3 are for the future research.

9.1 Program Error Detection

It was our original motivation to use dependent types to capture more programming errors at compile-time. We report some rather common errors which can be captured with the dependent type system developed in this thesis. Notice that all these errors slip through the type system of ML.

We have found that it is significantly beneficial for the programmer to be able to verify certain properties about the lengths of lists in programs. For instance, the following is an implementation of the quicksort algorithm on lists.

```
fun('a)
    quickSort cmp [] = []
  | quickSort cmp (x::xs) = par cmp (x, [], [], xs)
where quickSort <|
{n:nat} ('a * 'a -> bool) -> 'a list(n) -> 'a list(n)

and('a)
    par cmp (x, left, right, xs) =
    case xs of
        [] => (quickSort cmp left) @ (x :: (quickSort cmp right))
      | y::ys =>
            if cmp(y, x) then par cmp (x, y::left, right, ys)
            else par cmp (x, left, y::right, ys)
where par <| {p:nat,q:nat,r:nat} ('a * 'a -> bool) ->
             'a * 'a list(p) * 'a list(q) * 'a list(r) -> 'a list(p+q+r+1)
```

If the line below `case` is replaced with the following,

```
                [] => (quickSort cmp left) @ (quickSort cmp right)
```

```
datatype 'a dict =
  Empty (* considered black *)
| Black of 'a entry * 'a dict * 'a dict
| Red of 'a entry * 'a dict * 'a dict

typeref 'a dict of bool * nat with
  Empty <| 'a dict(true, 0)
| Black <|
  {cl:bool, cr:bool, bh:nat}
  'a entry * 'a dict(cl, bh) * 'a dict(cr, bh) -> 'a dict(true, bh+1)
| Red <|
  {bh:nat}
  'a entry * 'a dict(true, bh) * 'a dict(true, bh) -> 'a dict(false, bh)
```

Figure 9.1: The red/black tree data structure

that is, the programmer forgot to include `x` in the result returned by the function `par`, then the function could not be of the following type.

```
{p:nat,q:nat,r:nat} ('a * 'a -> bool) ->
'a * 'a list(p) * 'a list(q) * 'a list(r) -> 'a list(p+q+r+1)
```

As matter of a fact, the function `par` is of the following type after the replacement.

```
{p:nat,q:nat,r:nat} ('a * 'a -> bool) ->
'a * 'a list(p) * 'a list(q) * 'a list(r) -> 'a list(0)
```

Therefore, the above error is caught at compile-time when type-checking is performed.

We now present a more realistic example. A red/black tree is a balanced binary tree which satisfies the following conditions.

1. All leaves are marked black and all other nodes are marked either red or black.
2. Given a node in the tree, there are the same number of black nodes on every path connecting the node to a leaf. This number is called the *black height* of the node.
3. The two sons of every red node are black.

In Figure 9.1, we define a polymorphic datatype `'a dict`, which is essentially a binary tree with colored nodes. We then refine the datatype with type index objects (c, bh) drawn from the sort $bool * nat$, where c and bh are the color and the black height of the root of the binary tree. The node is black if and only if c is *true*. Therefore, the properties of a red/black tree is naturally captured with this datatype refinement. This enables the programmer to catch program errors which lead to violations of these properties when implementing an insertion or deletion operation on red/black trees. We have indeed encountered errors caught in this way in practice.

Notice that this refinement is different from the one declared in Section A.2, which is more suited for the implementation presented there.

9.2 Array Bound Check Elimination

Array bounds checking refers to determining whether the value of an expression is within the bounds of an array when it is used to index the array. Bounds violations, such as those notorious "off-by-one" errors, are among the most common programming errors.

- Pascal, Ada, SML, Java are among the programming languages which require that all bounds violations be captured.
- C, C++ are not.

However, run-time array bounds checking can be very expensive. For instance, it is observed that FoxNet written in SML (Buhler 1995) suffers up to 30% loss of throughput due to checksum operation, which is largely composed of run-time array bound checks. The SPIN kernel written in Modula-3 (Bershad, Savage, Pardyak, Sirer, Becker, Fiuczynski, Chambers, and Eggers 1995) also suffers some significant performance losses from run-time array bounds checking. The traditional *ad hoc* approaches to eliminating run-time array bound checks are based on flow analysis (Gupta 1994; Kolte and Wolfe 1995). A significant advantage of these approaches is that they can be made fully automatic, requiring no programmer supplied annotations. On the other hand, these approaches in general have very limited power. For instance, they cannot eliminate array bound checks involved with an array index whose value is not monotonic during the execution. Also they all rely on whole program analysis, having some fundamental difficulty crossing over module boundaries. Another serious criticism of these approaches is that they in general do not provide the programmer with feedback on why some array bound checks cannot be eliminated (if there are still some left after the flow analysis). In other words, these approaches, though may enhances the performance of the programs, cannot lead to more robust programs. Therefore, they offer virtually no software engineering benefits.

In this section, we show that dependent types can facilitate the elimination of run-time array bound checks. Our approach requires that the programmer supply type annotations in the code. In return, it is much more powerful than traditional approaches. For instance, we will show how to completely eliminate array bound checks in a binary search function, which seems beyond the reach of any practical approach based on flow analysis. In addition, our approach can provide the programmer with the feedback on why certain array bound checks cannot be eliminated. This enhances not only the performance of the programs but also their robustness. Therefore, our approach offers some software engineering benefits. Since our approach is orthogonal to the traditional ones, it seems straightforward to adopt our approach at type-checking stage and then use one based on flow analysis at code generation stage, combining the benefits of dependent types and flow analysis together.

In the standard basis we have refined the types of many common functions on integers such as addition, subtraction, multiplication, division, and the modulo operation. Please refer to Figure 8.1 in Chapter 8 for more details.

In order to eliminate array bound checks at compile-time, we assume that the array operations *sub* and *update* have been assigned the following types.

```
sub <| {n:nat} {i:nat | i < n} 'a array(n) * int(i) -> 'a
update <| {n:nat} {i:nat | i < n} 'a array(n) * int(i) * 'a -> unit
```

```
fun{size:nat}
    dotprod(v1, v2) =
      let
          fun loop(i, n, sum) =
              if i = n then sum
              else loop(i+1, n, sum + sub(v1, i) * sub(v2, i))
          where loop <| {i:nat | i <= size} int(i) * int(size) * int  -> int
      in
          loop(0, length v1, 0)
      end
where dotprod <| int array(size) * int array(size) -> int
```

Figure 9.2: The dot product function

Clearly, we are sure that the array accesses through *sub* or *update* cannot result in array bound violations, and therefore there is no need for inserting array bound checks when we compile the code.

Similarly, we can assign *nth* the following type, where *nth*, when given a list and a nonnegative integer i, returns the ith element in the list.

```
sub <| {n:nat} {i:nat | i < n} 'a list(n) * int(i) -> 'a
```

This can eliminate list tag checks in the implementation of *nth*.

The code in Figure 9.2 is an implementation of the dot product function. We use {`n:nat`} as an explicit universal quantifier or *dependent function type constructor*. Conditions may be attached, so they can be used to describe certain forms of *subset types*, such as {`n:nat | i < n`} in the types of *sub* and *update*. The two "`where`" clauses are present in the code for type-checking purposes, giving the dependent type of the local tail-recursive function `loop` and the function `dotprod` itself.

This could be a simple example for some approaches based on flow analysis since the index i in the code is always increasing. Now let us see an example which is challenging for approaches based on flow analysis. The code in Figure 1.3 is an implementation of binary search on an array. We have listed in Figure 3.4 some sample constraints generated from type-checking the code. All of these can be solved easily.

Note that if we program binary search in C, the array bound check cannot be hoisted out of loops using the algorithm presented in (Gupta 1994) since it is neither increasing nor decreasing in terms of the definition given there. On the other hand, the method in (Susuki and Ishihata 1977) could eliminate this array bound check by synthesizing an induction hypothesis similar to our annotated type for `look`. Unfortunately, synthesizing induction hypotheses is often prohibitively expensive in practice. In future work we plan to investigate extensions of the type-checker which could infer certain classes of generalizations, thereby relieving the programmer from the need for certain kinds of "obvious" annotations.

9.2.1 Experiments

We have performed some experiments on a small set of programs. Note that three of them (bcopy, binary search, and quicksort) were written by others and just annotated, providing evidence that

	constraints			type annotations		
Program	number	SML of NJ	MLWorks	total number	total lines	code size
bcopy	187	0.59/1.17	0.72/1.37	13	50	281 lines
binary search	13	0.07/0.02	0.10/0.04	2	2	33 lines
bubble sort	15	0.08/0.03	0.11/0.06	3	3	37 lines
matrix mult	18	0.10/0.04	0.16/0.06	5	10	50 lines
queen	18	0.11/0.03	0.14/0.04	9	9	81 lines
quick sort	135	0.29/0.58	0.37/0.68	16	40	200 lines
hanoi towers	29	0.10/0.09	0.13/0.13	4	10	45 lines
list access	4	0.07/0.01	0.08/0.01	2	3	18 lines

Table 9.1: Constraint generation/solution, time in secs

a natural ML programming style is amenable to our type refinements.

The first set of experiments were done on a Dec Alpha 3000/600 using SML of New Jersey version 109.32. The second set of experiments were done on a Sun Sparc 20 using MLWorks version 1.0. Sources of the programs can be found in (Xi 1997).

Table 9.1 summarizes some characteristics of the programs. We show that the number of constraints generated during type-checking and the time taken for generating and solving them using SML of New Jersey and MLWorks. Also we indicate the number of total type annotations in the code, the number of lines they occupy, and the code size. Note that some of the type annotations are already present in non-dependent form in ML, depending on programming style and module interface to the code. A brief description of the programs is given below.

bcopy This is an optimized implementation of the byte copy function used in the Fox project. We used this function to copy 1M bytes of data 10 times in a byte-by-byte style.

binary search This is the usual binary search function on an integer array. We used this function to look for 2^{20} randomly generated numbers in a randomly generated array of size 2^{20}.

bubble sort This is the usual bubble sort function on an integer array. We used this function to sort a randomly generated array of size 2^{13}.

matrix mult This is a direct implementation of the matrix multiplication function on two-dimensional integer arrays. We applied this function to two randomly generated arrays of size 256×256.

queen This is a variant of the well-known eight queens problem which requires positioning eight queens on a 8×8 chessboard without one being captured by another. We used a chessboard of size 12×12 in our experiment.

quick sort This implementation of the quick sort algorithm on arrays is copied from the SML of New Jersey library. We sorted a randomly generated integer array of size 2^{20}.

hanoi towers This is a variant of the original problem which requires moving 64 disks from one pole to another without stacking a larger disk onto a smaller one given the availability of a third pole. We used 24 disks in our experiments.

Program	with checks	without checks	gain	checks eliminated
bcopy	6.52	4.40	32%	20,971,520
binary search	40.40	30.10	25%	19,072,212
bubble sort	58.90	34.25	42%	134,429,940
matrix mult	30.62	16.79	45%	33,619,968
queen	15.85	11.06	30%	77,392,496
quick sort	29.85	25.32	15%	64,167,588
hanoi towers	11.34	8.28	27%	50,331,669
list access	2.24	1.24	45%	1,048,576

Table 9.2: Dec Alpha 3000/600 using SML of NJ working version 109.32, time unit = sec.

Program	with checks	without checks	gain	checks eliminated
bcopy	9.75	2.01	79%	20,971,520
binary search	31.78	25.00	21%	19,074,429
bubble sort	46.78	25.84	45%	134,654,868
matrix mult	60.43	51.27	15%	33,619,968
queen	29.81	14.81	50%	77,392,496
quick sort	79.95	70.28	12%	63,035,841
hanoi towers	9.59	7.20	25%	50,331,669
list access	1.58	0.77	51%	1,048,576

Table 9.3: Sun Sparc 20 using MLWorks version 1.0, time unit = sec.

list access We accessed the first sixteen elements in a randomly generated list at total of 2^{20} times.

We used the standard, safe versions of `sub` and `update` for array access when compiling the programs into the code with array bound checks. These versions always perform run-time array bound checks according to the semantics of Standard ML. We used unsafe versions of `sub` and `update` for array access when generating the code containing no array bound checks. These functions can be found in the structure `Unsafe.Array` (in SML of New Jersey), and in `MLWorks.Internal.Value` (in MLWorks). Our unsafe version of the `nth` function used `cast` for list access without tag checking.

Notice that unsafe versions of `sub`, `update` and `nth` can be used in our implementation only if they are assigned the corresponding types mentioned previously.

In Table 9.2 and Table 9.3, we present the effects of eliminating array bound checks and list tag checks. Note that the difference between the number of eliminated array bound checks in Table 9.2 and Table 9.3 reflects the difference between randomly generated arrays used in two experiments.

We also present two diagrams in Figure 9.3 and Figure 9.4. The height of a bar stands for the time spent on the experiment. The gray ones are for the experiments in which all array bound checks are eliminated at compile-time and the dark ones for the others.

It is clear that the gain is significant in all cases, rewarding the work of writing type annotations. In addition, type annotations can be very helpful for finding and fixing certain program errors, and

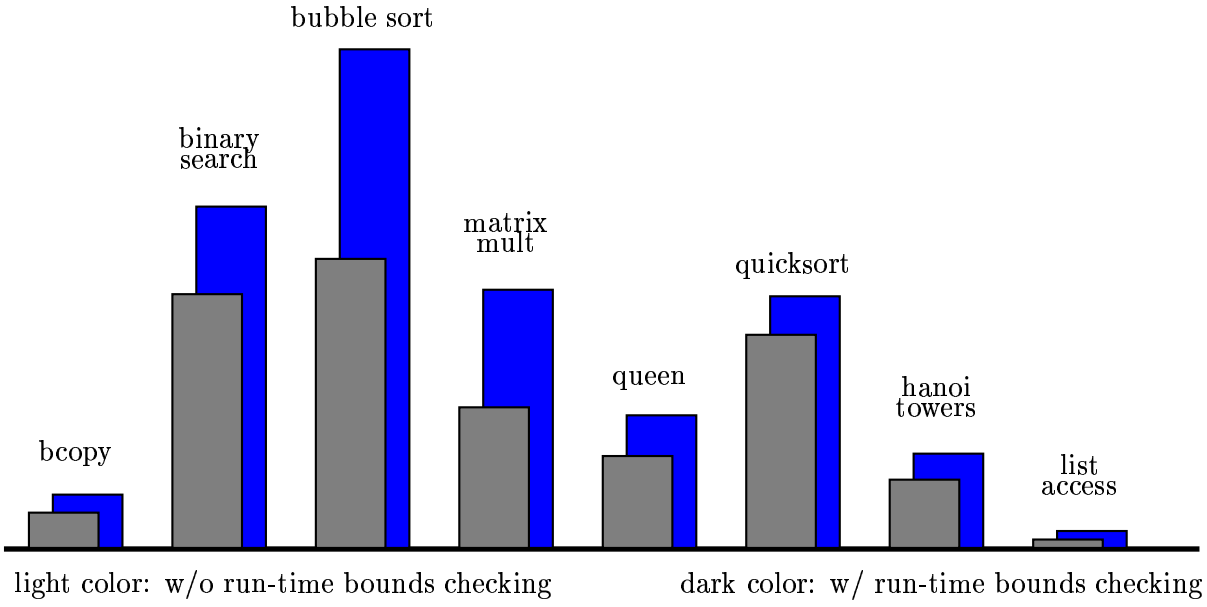

Figure 9.3: Dec Alpha 3000/600 using SML of NJ working version 109.32

for maintaining a software system since they provide the user with informative documentation. We feel that these factors yield a strong justification for our approach.

9.3 Potential Applications

In this section we present some potential applications of dependent types, which have yet to be implemented. We also outline some approaches to realizing these applications. We refer the reader to (Xi 1999) for further details regarding the subject on dead code elimination.

9.3.1 Dead Code Elimination

The following function *zip* zips two lists together. If the clause `zip(_, _) = raise zipException` is missing, then some ML compiler will issue a warning message stating that *zip* may result in a match exception to be raised. For instance, this happens if two arguments of *zip* are of different lengths.

```
exception zipException
fun('a, 'b)
    zip(nil, nil) = nil
  | zip(cons(x, xs), cons(y, ys)) = cons((x,y), zip(xs, ys))
  | zip(_, _) = raise zipException
```

However, this function is meant to zip two lists of *equal* length. If we declare that *zip* is of the following dependent type,

```
{n:nat} 'a list(n) * 'b list(n) -> ('a * b') list(n)
```

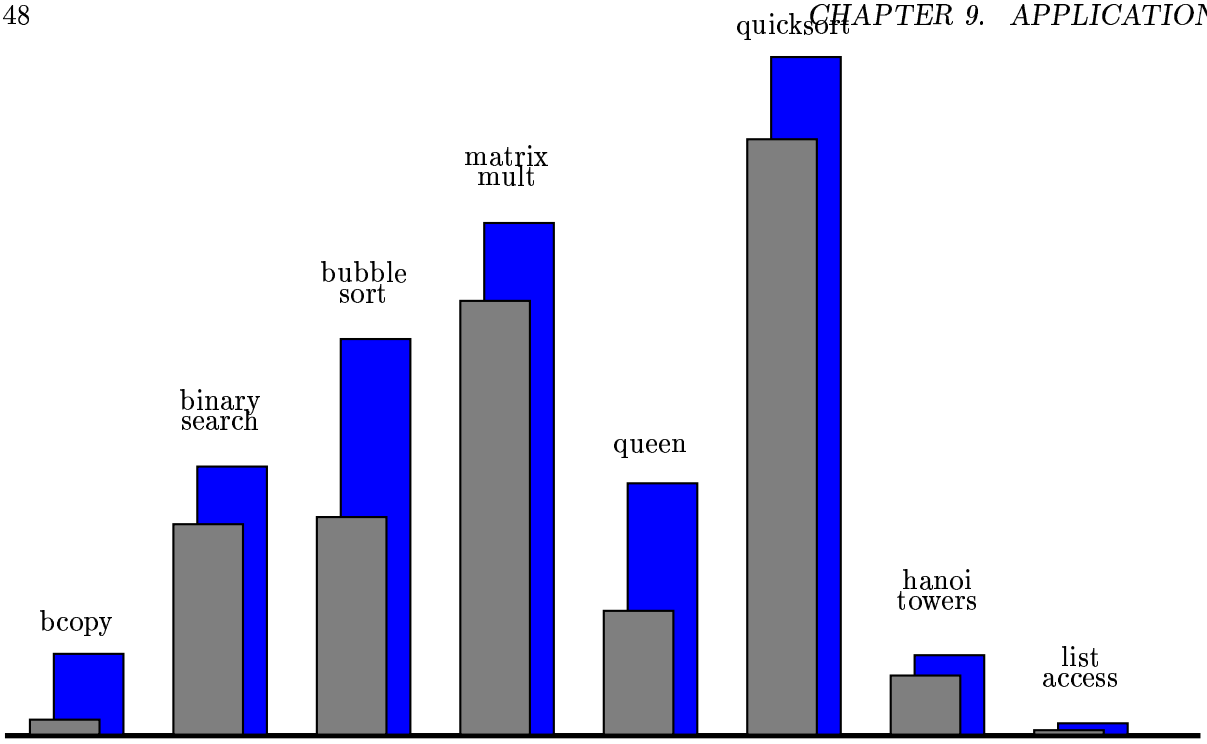

Figure 9.4: Sun Sparc 20 using MLWorks version 1.0

then the clause `zip(_, _) = raise zipException` in the definition of *zip* can never be reached, and therefore can be safely removed. In other words, we can declare the function *zip* as follows.

```
fun('a, 'b)
    zip(nil, nil) = nil
  | zip(cons(x, xs), cons(y, ys)) = cons((x,y), zip(xs, ys))
where {n:nat} 'a list(n) * 'b list(n) -> ('a * b') list(n)
```

This leads to not only more compact but also possibly more efficient code. For instance, if it has been checked that the first argument of *zip* is *nil*, then it can return the result *nil* immediately since it is redundant to check whether the second argument is *nil* (it must be).

We now prove a lemma, which provides the key to eliminating redundant matching clauses.

Lemma 9.3.1 *Given a pattern p and a type τ in $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ such that $p \downarrow \tau \rhd (\phi; \Gamma)$ is derivable. If $\cdot;\cdot \vdash v : \tau$ and $\mathbf{match}(v, p) \vdash \theta$ are derivable, then $\phi \models \bot$ is* not *satisfiable. In other words, if $\phi \models \bot$ is derivable, then there is no closed value of type τ which can match the pattern p.*

Proof If $\phi \models \bot$ is satisfiable, then $(\phi)\bot$ holds in the constraint domain C. It can be readily verified that a counterexample to $(\phi)\bot$ can be given if we let a be $\theta(a)$ for all $a \in \mathbf{dom}(\phi)$. If $\phi \models \bot$ is derivable, then $\phi \models \bot$ is satisfiable by definition. Therefore, there is no closed value v of type τ which matches the pattern p if $\phi \models \bot$ is derivable. ■

Let us call an index variable context ϕ inconsistent if $\phi \models \bot$ is satisfiable. Lemma 9.3.1 simply implies that no closed value of type τ can match a pattern p if checking p against τ yields an inconsistent index variable context.

Therefore, when the following rule is applied during elaboration,

$$\frac{p \downarrow \tau_1 \Rightarrow (p^*; \phi_1; \Gamma_1) \quad \phi, \phi_1^{\psi}; \Gamma, \Gamma_1 \vdash e \downarrow \tau_2 \Rightarrow [\psi]\ \Phi \quad \phi, \psi \vdash \tau_1 \Rightarrow \tau_2 : * \quad \phi, \psi \vdash \Gamma[\mathbf{ctx}]}{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow [\psi]\ \forall(\phi_1^{\psi}).\Phi}\ \textbf{(constr-match)}$$

we verify whether $\phi, \phi_1^{\psi} \models \bot$ is derivable. If it is, then the matching clause $p \Rightarrow e$ can never be reached. We can either issue a warning message at this point or safely remove the matching clause.

However, there is a serious issue which must be dealt with before we can apply this strategy to pattern matching in ML. The operational semantics of ML requires that pattern matching be done sequentially. For instance, if the third clause $zip(\_, \_)$ in the first declaration of zip is chosen to evaluate $zip(v)$, then v must not match either pattern (nil, nil) or $(cons(x, xs), cons(y, ys))$. Therefore, v matches either pattern $(cons(x, xs), nil)$ or $(nil, cons(y, ys))$. If v is of type $(\alpha)list(n) * (\beta)list(n)$ for some n, this is clearly impossible. This example suggests that we transform overlapped matching clauses into disjoint ones before detecting whether some of them are redundant. In the above case, this amounts to transforming the first declaration of zip into the following one.

```
exception zipException
fun('a, 'b)
    zip(nil, nil) = nil
  | zip(cons(x, xs), cons(y, ys)) = cons((x,y), zip(xs, ys))
  | zip(nil, cons(y, ys)) = raise zipException
  | zip(cons(x, xs), nil) = raise zipException
```

Let us assign zip the type $\Lambda\alpha.\Lambda\beta.\Pi n : nat.(\alpha)list(n) * (\beta)list(n) \rightarrow (\alpha * \beta)list(n)$. Notice that we have

$$(nil, cons(y, ys)) \downarrow (\alpha)list(n) * (\beta)list(n) \rhd (0 \doteq n, a : nat, a + 1 \doteq n; y : \beta, ys : (\beta)list(a))$$

Since $n : nat, 0 \doteq n, a : nat, a + 1 \doteq n \models \bot$ is derivable, the third clause is redundant by Lemma 9.3.1. Similarly, the fourth clause is also unreachable.

This approach seems to be straightforward, but it can lead to code size explosion when applied to certain examples. Therefore, we are still in search of a better solution to detecting unreachable matching clauses.

9.3.2 Loop Unrolling

In this subsection we present another potential application of dependent types, following some observation in Subsection 9.3.1. The following declared function $sumArray$ sums up all the elements in a given integer array.

```
fun{n:nat}
    sumArray(arr) =
       let
```

```
        fun loop(i, n, s) = if i = n then s else loop(i+1, n, sub(arr, i)+s)
        where loop <| {i:nat | i <= n} int(i) * int(n) * int -> int
    in
        loop(0, length(arr), 0)
    end
where sumArray <| int array(n) -> int
```

Note that `if i = n then s else loop(i+1, n, sub(arr, i)+s)` is a variant of the following case statement.

```
case i = n of true => s | false => loop(i+1, n, sub(arr, i)+s)
```

We now declare another function *sumArray*8 as follows, that is, *sumArray*8 can only be applied to an integer array of size 8.

```
fun sumArray8(arr) = sumArray(arr)
where sumArray <| int array(8) -> int
```

Then it seems reasonable that we can expand the declaration to the following through partial evaluation. We give some informal explanation.

```
fun sumArray8(arr) =
      sub(arr, 7) + (sub(arr, 6) + (sub(arr, 5) + (sub(arr, 4)
      (sub(arr, 3) + (sub(arr, 2) + (sub(arr, 1) + (sub(arr, 0) + 0)))))))
where sumArray <| int array(8) -> int
```

If *arr* is of type $(int)array(8)$, then $length(arr)$ is of type $int(8)$ since *length* is given the type $\Lambda\alpha.\Pi n : nat.(\alpha)array(n) \rightarrow int(n)$. After expanding $loop(0, length(arr), 0)$ to **let** $n = length(arr)$ **in** $loop(0, n, 0)$ **end** (this is a call-by-value language!), the type of n must be $int(8)$. We now expand $loop(0, n, 0)$ to

$$\textbf{case } 0 = n \textbf{ of } true \Rightarrow 0 \mid false \Rightarrow loop(0 + 1, n, sub(arr, 0) + 0)$$

Notice that the type of $0 = n$ is $bool(0 = 8)$ since $=$ is of the following type.

$$\Pi m : int.\Pi n : int.int(m) * int(n) \rightarrow bool(m = n)$$

Therefore, according to the reasoning in Section 9.3.1, the matching clause $true \Rightarrow 0$ is unreachable. This allows the simplification of the above case statement to $loop(0 + 1, n, sub(arr, 0) + 0)$. By repeating this process eight times, we reach the expanded declaration of *sumArray*8. This can lead to more efficient code without sacrificing clarity.

However, if the size of an integer array *arr* is a large natural number, it may not be advantageous to expand $sumArray(arr)$ since this can result in unexpected instruction cache behavior and thus slow down the code execution. We *propose* a possible solution as follows.

A significant problem with currently available programming languages is that there exist few approaches to improving the efficiency of code without overhauling the entire code. With the help of partial evaluation, this situation can be somewhat ameliorated as follows. We assume that the programmer decides to write the function *sumArray_unroll* in Figure 9.5 to replace *sumArray* for the sake of efficiency. Though much more involved than the example about *sumArray*8, we expect that *loop_8_times* can specialize to the following function with partial evaluation.

```
fun{n:nat}
    sumArray_unroll(arr) =
       let
          fun loop(i, n, s) = if i = n then s else loop(i+1, n, sub(arr, i)+s)
          where loop <| {i:nat | i <= n} int(i) * int(n) -> int

          fun loop_8_times(i, n, s) = loop(i, n, s)
          where loop_8_times <|
          {i:nat | i <= n /\ n mod 8 = 0} int(i) * int(n) -> int
       in
          let
             val n = length(arr)
             and r = n % 8
          in
             loop(n-r, r, loop_8_times(0, n-r, 0))
          end
       end
where sumArray_unroll <| int array(n) -> int
```

Figure 9.5: loop unrolling for sumArray

```
fun loop_8_times(i, n, s) =
        if i = n then s
        else loop_8_times(i+8, n,
                          sub(arr, i+7)+(sub(arr, i+6)+
                          (sub(arr, i+5)+(sub(arr, i+4)+
                          (sub(arr, i+3)+(sub(arr, i+2)+
                          (sub(arr, i+1)+s)))))))
where loop_8_times <| {i:nat | i <= n /\ n mod 8 = 0} int(i) * int(n) -> int
```

This roughly corresponds to loop-unrolling, a well-known technique in compiler optimization. Though we have not shown that loop unrolling done above preserve the operational semantics, we think that this is a straightforward matter. Now it seems reasonable to gain some performance by expanding *sumArray_unroll*(*arr*) for *arr* of large known size. The interested reader is referred to (Draves 1997) for some realistic and interesting examples which may be handled in this way.

Combining dependent types with partial evaluation, we hope to find an approach to improving the efficiency of existing code with only moderate amount of modification. This is currently an exciting but highly speculative research direction.

9.3.3 Dependently Typed Assembly Language

The studies on the use of types in compilation have been highly active recently. For instance, the work in (Morrisett 1995; Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996; Tolmach and Oliva 1998; Morrisett, Walker, Crary, and Glew 1998) has demonstrated convincing evidence to support the use of typed intermediate and assembly languages for various purposes such as data

```
int dotprod(int A[], int B[], int n) {
  int i, sum;
  sum = 0;
  for(i = 0; i < n; i++) { sum += A[i] * B[i]; }
  return sum;
}
```

Figure 9.6: The C version of dotprod function

layout, tag-free garbage collection, compiler error detection, etc. This immediately indicates that it would be beneficial if we could pass dependent types to lower level languages during compilation. Many compiler optimizations involving *code motion* may then benefit from the use of dependent types. Array bound check elimination through dependent types in Section 9.2 is a solid support of this argument.

We have started to formulate a dependently typed assembly language, which is mainly inspired by (Morrisett, Walker, Crary, and Glew 1998). The theory of this language is yet to be developed. We now use an example to informally present some ideas behind this research. The following code in Figure 9.6 is an implementation of dot product function in C. It is written in this way so that it can be directly compared with the code in Figure 9.7, which is an implementation of dot product function in DTAL, a dependently typed assembly language. Note that "\\" starts a line of comment.

In DTAL, each label is associated with a type. For instance, the label `dotprod` is associated with the following type.

```
{n: nat} [r0: int array(n), r1: int array(n), r3: int(n)]
```

Roughly speaking, this type means that when the execution of the code reaches the label `dotprod`, the registers $r0$ and $r1$ must point to integer arrays of size n for some natural number n and $r3$ stores an integer equal to n.

The DTAL code has been type-checked in a prototype implementation. Notice that the type system guarantees that there is no memory violation when the command `load r4, r0(r2)` is executed since the value in $r2$ is a natural number less than the size of the array to which $r0$ points. Therefore, if the code is downloaded from an untrusted source and type-checked locally, no run-time checks are needed for preventing possible memory violations. This opens an exciting avenue to eliminating array bound checks for programming languages such as Java, which run on networks. More examples of DTAL code can be found in (Xi 1998).

9.4 Summary

We have so far presented some applications of dependent types. The uses of dependent types in program error detection and array bound check elimination have been put into practice. Though it seems relatively straightforward to use dependent types for eliminating unreachable matching clauses or issuing more accurate warning messages about inexhausitive pattern matching, but this is yet to be implemented. Also we have speculated that it could be beneficial to combine partial

```
dotprod:{n: nat} \\ n is universally quantified
        [r0: int array(n), r1: int array(n), r3: int(n)]
        \\ r0 and r1 point to integer arrays A and B of size n, respectively
        \\ and n is stored in r3

        mov     r31, 0 \\ set r31 to 0
        mov     r2, 0 \\ set r2 to
        jmp     loop  \\ start the loop

loop:   {n:nat, i:int | 0 <= i <= n}
        \\ n and i are universally quantified and 0 <= i <= n

        [r0: int array(n), r1: int array(n), r2:int(i), r3: int(n), r31: int]
        \\ r2 = i and r3 = n

        cmp     r2, r3 \\ compare r2 and r3
        ifnz           \\ r2 is not equal to r3
                load    r4, r0(r2) \\ load A[i] into r4
                load    r5, r1(r2) \\ load B[i] into r5
                mul     r4, r4, r5 \\ r4 = r4 * r5
                add     r31, r31, r4 \\ r31 = r31 + r4
                add     r2, r2, 1 \\ increase r2 by 1
                jmp     loop \\ repeat the loop
        else           \\ r2 is equal to n
                jmp     finish \\ done
        endif

finish: [r31: int] \\ r31 stores the result, which is an integer
        halt
```

Figure 9.7: The DTAL version of dotprod function

evaluation with dependent types, demonstrating informally that loop-unrolling may be controlled by the programmer with dependent types.

It is both promising and highly desirable to spot more concrete opportunities in compiler optimization which could benefit from dependent types.

Chapter 10

Conclusion and Future Work

The dependent type inference developed in this thesis has demonstrated convincing signs of being a viable system for practical use. Compared to ML-types, dependent types can more accurately capture program invariants and therefore lead to detecting more program errors at compile-time. Also, the use of dependent types in array bound check elimination is encouraging since this can enhance not only the robustness but also the efficiency of programs.

As with any programming language, DML has many weak points. Some of the weak points result from the trade-offs made to ensure the practicality of dependent type inference, and some can be remedied through further experiment and research. In this chapter we summarize the current research status on incorporating dependent types into ML and point out some directions to pursue in the future to make DML a better programming language.

10.1 Current Status

We briefly mention the current status of DML in terms of both language design and language implementation.

10.1.1 Language Design

We have so far finished extending the core of ML with a notion of dependent types, that is, combining dependent types with language features such as datatype declarations, higher-order functions, let-polymorphism, references and exception mechanism. The extended language is given the name DML (for Dependent ML). Strictly speaking, DML is really a language parameterized over a given constraint domain C and thus should be denoted by DML(C). We may omit writing the constraint domain C in the following presentation, and if we do so then we mean that the omitted C is the integer constraint domain presented in Section 3.3, or C is simply irrelevant.

We have proven the soundness of the enriched type system and then constructed a practical type-checking algorithm for it. Furthermore, the correctness of the type-checking algorithm is also established. This has placed our work on a solid theoretical foundation.

DML is a conservative extension of ML in the sense that a DML program which uses no dependent types is simply a valid ML program. In order to make DML fully compatible with the core of ML, we designed a two-phase type-checking algorithm for DML. This guarantees that an ML program (written in some external language for ML) can always pass type-checking for DML

if it passes the type-checking for ML. Therefore, the programmer can use sparingly the features related to dependent types when writing (large) programs.

10.1.2 Language Implementation

We have finished a prototype implementation of a type-checker for a substantial part of DML(C), where C is the integer constraint domain in Section 3.3. This part roughly corresponds to the language $\mathrm{ML}^{\forall,\Pi,\Sigma}_{0,\mathrm{exc},\mathrm{ref}}(C)$ introduced in Section 7.4, including most features in the core of ML such as higher-order functions, datatypes, let-polymorphism, references and exception mechanism. However, records have yet to be implemented. It should be straightforward to include records in a future implementation since they are simply a variant of products. All examples in Chapter A have been verified in this implementation.

The constraint solver for the integer domain is based on a variant of the Fourier-Motzkin variable elimination approach (Dantzig and Eaves 1973). This is an intuitive and clean approach, which we think is more promising than those based on SUP-INF or the simplex method to report comprehensible and accurate type error or warning messages on unsatisfiable constraints, a vital component for type-checking in DML(C). The weak aspect of this approach is that it seems less promising to handle large contraints than the Simplex method, but this issue needs to be further investigated.

10.2 Future Research in Language Design

In this section, we present some future research directions for improving DML.

10.2.1 Modules

Since we have finished adding dependent types to the core of ML, namely, ML without module level constructs, the next move is naturally to study the interaction between the module system of ML and dependent types. There are many intricate issues which can only be answered in practice. An immediate question is how to export dependent types in signature. Since there is no notion of principal types in DML, a function can be assigned two dependent types neither of which coerces into the other. For instance, the following declared function *tail* can be assigned types $\forall\alpha.(\Sigma n : nat.(\alpha)list(n)) \rightarrow \Sigma n : nat.(\alpha)list(n)$ and $\forall\alpha.\Pi n : nat.(\alpha)list(n+1) \rightarrow (\alpha)list(n)$, respectively.

```
fun tail(cons(x, xs)) = x
```

The second type cannot be coerced into the first one since a function of the first type can be applied to any list while a function of the second type can only be applied to a non-empty list. If the length of a list l cannot be inferred from static type-checking, then only the first assigned type can be used if we need to type-check $tail(l)$. However, if l is inferred to be not empty at compile-time, the use of the second type can lead to potentially more efficient code as explained in Section 9.3.1. At this moment, we contemplate introducing a notion of *top-level* conjunction types into DML. In the above case, we would like to assign *tail* the following conjunction types

$$(\forall\alpha.(\Sigma n : nat.(\alpha)list(n)) \rightarrow \Sigma n : nat.(\alpha)list(n)) \wedge (\forall\alpha.\Pi n : nat.(\alpha)list(n+1) \rightarrow (\alpha)list(n))$$

Then the programmer is allowed to choose which type is needed for an occurrence of *tail*. There are yet many details to be filled in and some experience to be gained on this issue.

10.2.2 Combination of Different Refinements

We currently require that a datatype be refined *at most once*. However, there are also cases where a datatype may need different refinements for different purposes. For instance, we encountered a case where we needed to refine the datatype $((\alpha)list)list$ with a pair of index objects (i, j) to represent the length of a list of lists and the sum of the lengths of the lists in this list of lists. It is not clear how this refinement could be done since the datatype $(\alpha)list$ has already been refined with an index which stands for the length of a list. Instead, we declared the following datatype, refined it and then substituted it for $((\alpha)list)list$.

```
datatype 'a listlist = Nil | Cons of 'a list * 'a listlist
typeref 'a listlist of nat * nat
with Nil <| 'a listlist(0,0)
   | Cons <| {l:nat,m:nat,n:nat}
             'a list(l) * 'a listlist(m,n) -> 'a listlist(m+1,n+l)
```

This resulted in substituting Nil and $Cons$ for *nil* and *cons* in many places of a program, respectively. More details can be found in the example on merge sort presented in Section A.4. It is a future research topic to study how to combine several different refinements of a datatype.

10.2.3 Constraint Domains

The general constraint language in Section 3.1 allows the programmer to declare the constraint domain C over which the language DML(C) is parameterized. Then, by Theorem 5.2.7, the type-checking in DML(C) can be reduced to constraint satisfaction in C. Unfortunately, there is no method available to enable the programmer to supply a constraint solver for C.

Therefore, it is highly desirable to provide the programmer with a language in which a constraint solver can be written. A programmer-supplied constraint solver for constraint domain C can then be combined with elaboration so that type-checking for DML(C) can be performed.

10.2.4 Other Programming Languages

Another research direction is to apply the language design approach in this thesis to other (strongly typed) programming languages such as Haskell(Hudak, Peyton Jones, and Wadler 1992) and Java(Sun Microsystems 1995). Array bound check elimination in Java, however, requires some special care, as we explain now. A program in Java is often compiled into Java Virtual Machine Language (JVML) code and shipped through networks. Since JVML code can be downloaded by a local host which does not trust the source of the code, there must be some evidence attached to the code in order to convince the local host that it is safe to eliminate array bound checks in the code. An approach presented in (Necula 1997) is to make the code carry a proof of certain properties of the code which can be verified by the local host, leading to the notion of proof-carrying code. In practice, the proof carried by code may tend to be difficult to construct and large when compared to the size of the code. Another approach, following (Morrisett, Walker, Crary, and Glew 1998), is to make the compiled code explicitly typed with dependent types so that code properties can

be verified by the local host equipped with a type-checker for dependent types. This leads to the notion of dependently typed assembly language.

10.2.5 Denotational Semantics

We are also interested in constructing a categorical model for the language $\mathrm{ML}_0^{\Pi,\Sigma}(C)$. Various denotational models based on the notion of locally closed cartesian categories have already been constructed for λ-calculi with fully dependent type systems such as the one which underlies LF (Harper, Honsell, and Plotkin 1993). However, $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ is essentially different from these λ-calculi because of the separation between type index objects and language expressions. We expect that a model tailored for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ would yield some semantic explanation on index erasure, which simply cannot exist in a fully dependent type setting.

10.3 Future Implementations

The present prototype implementation exhibits many aspects for immediate improvement. For instance, we have observed that a large percentage of the constraints can be solved immediately after their generation. However, we currently collect *all* constraints generated during elaboration in a constraint store before we call a constraint solver. This practice often leads to inflating the number of constraints significantly at the stage where all constraints are transformed into some standard form. Therefore, it seems promising that elaboration can be done much more efficiently if we intertwine constraint generation with constraint solution.

Another observation is that an overwhelming majority of integer constraints generated during elaboration are trivial and can be solved with a constraint solver which is highly efficient but incomplete, such as a constraint solver based the simplex method for real numbers. After filtering out the trivial constraints, we can then use a complete constraint solver such as the one mentioned in (Pugh and Wonnacott 1992) to solve the rest of constraints. A similar strategy has been adopted in the constraint logic programming community for efficiently solving constraints.

A certifying compiler for *Safe C*, a programming language with similar constructs to part of C, is presented in (Necula and Lee 1998). At this stage, the compiler largely relies on synthesizing loop invariants in code in order to verify certain properties such as memory integrity. This approach, however, seems difficult to cope with large programs. On the other hand, the type system of DML is strong enough for allowing the programmer to supply loop invariants through type annotations. This gives DML a significant advantage when the scalability issue is concerned. Therefore, it is natural to consider whether a certifying compiler for DML can be implemented in the future.

Appendix A

DML Code Examples

A.1 Knuth-Morris-Pratt String Matching

The following is an implementation of the Knuth-Morris-Pratt string matching algorithm using dependent types to eliminate most array bound checks.

```
structure KMP =
  struct
     assert length <| {n:nat} 'a array(n) -> int(n)

     and sub <| (* sub requires NO bound checking *)
         {size:int, i:int | 0 <= i < size} 'a array(size) * int(i) -> 'a

     and subCK <| (* subCK requires bound checking *)
         'a array * int -> 'a

     (* notice the use of existential types *)
     type intPrefix = [i:int| 0 <= i+1] int(i)

     assert arrayPrefix <|
            {size:nat} int(size) * intPrefix -> intPrefix array(size)

     and subPrefix <| (* subPrefix requires NO bound checking *)
         {size:int, i:int | 0 <= i < size}
         intPrefix array(size) * int(i) -> intPrefix

     and subPrefixCK <| (* subPrefixCK requires bound checking *)
         intPrefix array * int -> intPrefix

     and updatePrefix <| (* updatePrefix requires NO bound checking *)
         {size:int, i:int | 0 <= i < size}
         intPrefix array(size) * int(i) * intPrefix -> unit
```

```
    (*
     * computePrefixFunction generates the prefix function
     * table for the pattern pat
     *)
    fun computePrefixFunction(pat) =
        let
           val plen = length(pat)
           val prefixArray = arrayPrefix(plen, ~1)

           fun loop(i, j) = (* calculate the prefix array *)
               if (j >= plen) then ()
               else
                   if sub(pat, j) <> subCK(pat, i+1) then
                      if (i >= 0) then loop(subPrefixCK(prefixArray, i), j)
                      else loop(~1, j+1)
                   else (updatePrefix(prefixArray, j, i+1);
                         loop(subPrefix(prefixArray, j), j+1))
           where loop <| {j:nat} intPrefix * int(j) -> unit
        in
           (loop(~1, 1); prefixArray)
        end
    where computePrefixFunction <| {p:nat} int array(p) -> intPrefix array(p)

    fun kmpMatch(str, pat) =
        let
           val strLen = length(str)
           and patLen = length(pat)

           val prefixArray = computePrefixFunction(pat)
           fun loop(s, p) =
               if s < strLen then
                 if p < patLen then
                    if sub(str, s) = sub(pat, p) then loop(s+1, p+1)
                    else
                        if (p = 0) then loop(s+1, p)
                        else loop(s, subPrefix(prefixArray, p-1)+1)
                    else (s - patLen)
                 else ~1
           where loop <| {s:nat, p:nat} int(s) * int(p) -> int
        in
           loop(0, 0)
        end
    where kmpMatch <| {s:nat, p:nat} int array(s) * int array(p) -> int
end
```

A.2 Red/Black Tree

```
(*
 * This example shows that the insert operation maps a balanced
 * red/black tree into a balanced one. Also it increases the size
 * of the tree by at most one (note that the inserted key may have
 * already existed in the tree). There 8 type annotations occupying
 * about 20 lines.
 *)

structure RedBlackTree =
  struct
      type key = int
      type answer = key option
      type 'a entry = int * 'a

      datatype order = LESS | EQUAL | GREATER
      datatype 'a dict =
        Empty (* considered black *)
      | Black of 'a entry * 'a dict * 'a dict
      | Red of 'a entry * 'a dict * 'a dict

      (*
       * We refine the datatype 'a dict with an index of type
       * (nat * nat * nat * nat). The meaning of the 4 numbers
       * is: (color, black height, red height, size). A balanced
       * tree is one such that
       * (1) for every node in it, both of its sons are of the
       *     same black height.
       * (2) the red height of the tree is 0, which means that there exist
              no consecutive red nodes.
       *)

      typeref 'a dict of nat * nat * nat * nat with
        Empty <| 'a dict(0, 0, 0, 0)
      | Black <|
        {cl:nat, cr:nat, bh:nat, sl:nat, sr:nat}
        'a entry * 'a dict(cl, bh, 0, sl) * 'a dict(cr, bh, 0, sr) ->
        'a dict(0, bh+1, 0, sl+sr+1)
      | Red <| {cl:nat, cr:nat, bh:nat, rhl:nat, rhr:nat, sl:nat, sr:nat}
        'a entry * 'a dict(cl, bh, rhl, sl) * 'a dict(cr, bh, rhr, sr) ->
        'a dict(1, bh, cl+cr+rhl+rhr, sl+sr+1)

      (* note if the root of a tree is black, then the tree is a balanced *)
```

```
fun compare (s1:int,s2:int) =
    if s1 > s2 then GREATER else if s1 < s2 then LESS else EQUAL
where compare <| int * int -> order

fun('a)
lookup dict key =
let
   fun lk (Empty) = NONE
     | lk (Red tree) = lk' tree
     | lk (Black tree) = lk' tree
   where lk <| 'a dict -> answer

   and lk' ((key1, datum1), left, right) =
       (case compare(key,key1) of
            EQUAL => SOME(key1)
          | LESS => lk left
          | GREATER => lk right)
   where lk' <| 'a entry * 'a dict * 'a dict -> answer
in
   lk dict
end
where lookup <| 'a dict -> key -> answer

fun('a)
  restore_right(e, Red lt, Red (rt as (_,Red _,_))) =
  Red(e, Black lt, Black rt)(* re-color *)

| restore_right(e, Red lt, Red (rt as (_,_,Red _))) =
  Red(e, Black lt, Black rt)(* re-color *)

| restore_right(e, l as Empty, Red(re, Red(rle, rll, rlr), rr)) =
  Black(rle, Red(e, l, rll), Red(re, rlr, rr))

| restore_right(e, l as Black _, Red(re, Red(rle, rll, rlr), rr)) =
  (* l is black, deep rotate *)
  Black(rle, Red(e, l, rll), Red(re, rlr, rr))

| restore_right(e, l as Empty, Red(re, rl, rr as Red _)) =
  Black(re, Red(e, l, rl), rr)

| restore_right(e, l as Black _, Red(re, rl, rr as Red _)) =
  (* l is black, shallow rotate *)
  Black(re, Red(e, l, rl), rr)

| restore_right(e, l, r as Red(_, Empty, Empty)) = Black(e, l, r)
```

```
| restore_right(e, l, r as Red(_, Black _, Black _)) =
  Black(e, l, r) (* r must be a red/black tree *)

| restore_right(e, l, r as Black _) =
  Black(e, l, r) (* r must be a red/black tree *)

where restore_right <|
{cl:nat, cr:nat, bh:nat, rhr:nat, sl:nat, sr:nat | rhr <= 1}
'a entry * 'a dict(cl, bh, 0, sl) * 'a dict(cr, bh, rhr, sr) ->
[c:nat | c <= 1 ] 'a dict(c, bh+1, 0, sl + sr + 1)

fun('a)
  restore_left(e, Red (lt as (_,Red _,_)), Red rt) =
  Red(e, Black lt, Black rt)(* re-color *)

| restore_left(e, Red (lt as (_,_,Red _)), Red rt) =
  Red(e, Black lt, Black rt)(* re-color *)

| restore_left(e, Red(le, ll as Red _, lr), r as Empty) =
  Black(le, ll, Red(e, lr, r))

| restore_left(e, Red(le, ll as Red _, lr), r as Black _) =
  (* r is black, shallow rotate *)
  Black(le, ll, Red(e, lr, r))

| restore_left(e, Red(le, ll, Red(lre, lrl, lrr)), r as Empty) =
  Black(lre, Red(le, ll, lrl), Red(e, lrr, r))

| restore_left(e, Red(le, ll, Red(lre, lrl, lrr)), r as Black _) =
  (* r is black, deep rotate *)
  Black(lre, Red(le, ll, lrl), Red(e, lrr, r))

| restore_left(e, l as Red(_, Empty, Empty), r) = Black(e, l, r)

| restore_left(e, l as Red(_, Black _, Black _), r) =
  Black(e, l, r) (* l must be a red/black tree *)

| restore_left(e, l as Black _, r) =
  Black(e, l, r) (* l must be a red/black tree *)

where restore_left <|
{cl:nat, cr:nat, bh:nat, rhl:nat, sl:nat, sr:nat | rhl <= 1}
'a entry * 'a dict(cl, bh, rhl, sl) * 'a dict(cr, bh, 0, sr) ->
[c:nat | c <= 1 ] 'a dict(c, bh+1, 0, sl + sr + 1)
```

```
exception Item_Is_Found

fun('a)
insert (dict, entry as (key,datum)) =
let
   (* val ins : 'a dict -> 'a dict  inserts entry
    * ins (Red _) may violate color invariant at root,
    * having red height 1
    * ins (Black _) or ins (Empty) will always be red/black
    * ins always preserves black height
    *)
   fun ins (Empty) = Red(entry, Empty, Empty)
     | ins (Red(entry1 as (key1, datum1), left, right)) =
       (case compare(key,key1) of
            EQUAL => raise Item_Is_Found
          | LESS => Red(entry1, ins left, right)
          | GREATER => Red(entry1, left, ins right))
     | ins(Black(entry1 as (key1, datum1), left, right)) =
       (case compare(key,key1) of
            EQUAL => raise Item_Is_Found
          | LESS => restore_left(entry1, ins left, right)
          | GREATER => restore_right(entry1, left, ins right))
   where ins <|
   {c:nat, bh:nat, s:nat}
   'a dict(c, bh, 0, s) ->
   [nc:nat, nrh:nat |
    ((c = 0 /\ nrh = 0 /\ nc <= 1) \/ (c = 1 /\ nrh <= 1 /\ nc = 1))]
   'a dict(nc, bh, nrh, s+1)
in
   let
      val dict = ins dict
   in
      case dict of
          Red (t as (_, Red _, _)) => Black t (* re-color *)
        | Red (t as (_, _, Red _)) => Black t (* re-color *)
        | Red (t as (_, Black _, Black _)) => dict
        | Red (t as (_, Empty, Empty)) => dict
        | Black _ => dict
   end handle Item_Is_Found => dict
end
where insert <|
{c:nat, bh:nat, s:nat}
'a dict(c, bh, 0, s) * 'a entry ->
[nc:nat, nbh:nat, ns:nat |
```

```
       (nbh = bh \/ nbh = bh + 1) /\ (ns = s \/ ns = s + 1) ]
      'a dict(nc, nbh, 0, ns)
  end
```

A.3 Quicksort on Arrays

```
(*
 * This example shows that array bounds checking is not required in
 * the following implementation of an in-place quicksort algorithm
 * on arrays. The code is copied from SML/NJ lib with some modification.
 * There are 16 type annotations occupying about 40 lines.
 *)

structure Array_QSort =
  struct
      datatype order = LESS | EQUAL | GREATER

      assert sub <| {n:nat, i:nat | i < n } 'a array(n) * int(i) -> 'a
      and update <| {n:nat, i:nat| i < n } 'a array(n) * int(i) * 'a -> unit
      and length <| {n:nat} 'a array(n) -> int(n)

      fun('a){size:nat}
      sortRange(arr, start, n, cmp) =
      let
         fun item i = sub(arr,i)
         where item <| {i:nat | i < size } int(i) -> 'a

         fun swap (i,j) =
             let
                val tmp = item i
             in
                update(arr, i, item j); update(arr, j, tmp)
             end
         where swap <|
         {i:nat, j:nat | i < size /\ j < size } int(i) * int(j) -> unit

         fun vecswap (i,j,n) =
             if (n = 0) then () else (swap(i,j);vecswap(i+1,j+1,n-1))
         where vecswap <|
         {i:nat, j:nat, n:nat | i+n <= size /\ j+n <= size}
         int(i) * int(j) * int(n) -> unit

         (*
          * insertSort is called if there are less than
```

```
 * eight elements to be sorted
 *)
fun insertSort (start, n) =
    let
       val limit = start+n
       fun outer i =
           if i >= limit then ()
           else
               let
                  fun inner j =
                      if j <= start then outer(i+1)
                      else
                          let
                             val j' = j - 1
                          in
                             case cmp(item j',item j) of
                                 GREATER => (swap(j,j'); inner j')
                               | _  => outer(i+1)
                          end
                  where inner <| {j:nat | j < size } int(j) -> unit
               in
                  inner i
               end
       where outer <| {i:nat} int(i) -> unit
    in
       outer(start+1)
    end
where insertSort <|
{start:nat, n:nat | start+n <= size } int(start) * int(n) -> unit

(* calculate the median of three *)
fun med3(a,b,c) =
    let
       val a' = item a
       val b' = item b
       val c' = item c
    in
       case (cmp(a', b'),cmp(b', c')) of
           (LESS, LESS) => b
         | (LESS, _) => (case cmp(a', c') of LESS => c | _ => a)
         | (_, GREATER) => b
         | _ => (case cmp(a', c') of LESS => a | _ => c)
       (* end case *)
    end
where med3 <|
```

```
{a:nat,b:nat,c:nat | a < size /\  b < size /\ c < size }
int(a) * int(b) * int(c) -> [n:nat | n < size ] int(n)

(* generate the pivot for splitting the elements *)
fun getPivot (a,n) =
    if n <= 7 then a + n div 2
    else
        let
           val p1 = a
           val pm = a + n div 2
           val pn = a + n - 1
        in
           if n <= 40 then med3(p1,pm,pn)
           else
               let
                  val d = n div 8
                  val p1 = med3(p1,p1+d,p1+2*d)
                  val pm = med3(pm-d,pm,pm+d)
                  val pn = med3(pn-2*d,pn-d,pn)
               in
                  med3(p1,pm,pn)
               end
        end
where getPivot <|
{a:nat,n:nat | 1 < n /\ a + n <= size }
int(a) * int(n) -> [p:nat | p < size] int(p)

fun quickSort (arg as (a, n)) =
    let
       (*
        * bottom was defined as a higher order
        * function in the SML/NJ library
        *)
       fun bottom(limit, arg as (pa, pb)) =
           if pb > limit then arg
           else
               case cmp(item pb,item a) of
                   GREATER => arg
                 | LESS => bottom(limit, (pa, pb+1))
                 | _ => (swap arg; bottom(limit, (pa+1,pb+1)))
       where bottom <|
       {l:nat, ppa:nat, ppb:nat |
        l < size /\ ppa <= ppb <= l+1 }
       int(l) * (int(ppa) * int(ppb)) ->
       [pa:nat, pb:nat | ppa <= pa <= pb <= l+1]
```

```
(int(pa) * int(pb))

(*
 * top was defined as a higher order
 * function in the SML/NJ library
 *)
fun top(limit, arg as (pc, pd)) =
    if limit > pc then arg
    else case cmp(item pc,item a) of
             LESS => arg
           | GREATER => top(limit, (pc-1,pd))
           | _ => (swap arg; top(limit, (pc-1,pd-1)))
where top <|
     {l:nat, ppc:nat, ppd:nat |
      0 < l <= ppc+1 /\ ppc <= ppd < size }
     int(l) * (int(ppc) * int(ppd)) ->
     [pc:nat, pd:nat | l <= pc+1 /\ pc <= pd <= ppd]
     (int(pc) * int(pd))

fun split (pa,pb,pc,pd) =
    let
       val (pa,pb) = bottom(pc, (pa,pb))
       val (pc,pd) = top(pb, (pc,pd))
    in
       if pb >= pc then (pa,pb,pc,pd)
       else (swap(pb,pc); split(pa,pb+1,pc-1,pd))
    end
where split <|
{ppa:nat, ppb:nat, ppc:nat, ppd:nat |
 0 < ppa <= ppb <= ppc+1 /\ ppc <= ppd < size }
int(ppa) * int(ppb) * int(ppc) * int(ppd) ->
[pa:nat, pb:nat, pc:nat, pd:nat |
 ppa <= pa <= pb <= pc+1 /\ pc <= pd <= ppd ]
(int(pa) * int(pb) * int(pc) * int(pd))

val pm = getPivot arg
and _ = swap(a,pm)
and pa = a + 1
and pc = a + (n-1)
and (pa,pb,pc,pd) = split(pa,pa,pc,pc)
and pn = a + n

val r = min(pa - a, pb - pa)
val _ = vecswap(a, pb-r, r)
```

```
                val r = min(pd - pc, pn - pd - 1)
                val _ = vecswap(pb, pn-r, r)

                val n' = pb - pa
                val _ = (if n' > 1 then sort(a,n') else ()) <| unit

                val n' = pd - pc
                val _ = (if n' > 1 then sort(pn-n',n') else ()) <| unit

            in () end
        where quickSort <|
        {a:nat, n:nat | 7 <= n /\ a+n <= size } int(a) * int(n) -> unit

        and sort (arg as (_, n)) =
            if n < 7 then insertSort arg
            else quickSort arg
        where sort <|
        {a:nat, n:nat |  a+n <= size } int(a) * int(n) -> unit
     in
        sort (start,n)
     end
     where sortRange <|
     {start:nat, n:nat | start+n <= size }
     'a array(size) * int(start) * int(n) * ('a * 'a -> order) -> unit

   (* sorted checks if a list is well-sorted *)
   fun('a){size:nat}
   sorted cmp arr =
   let
      val len = length arr
      fun s(v,i) =
          let
             val v' = sub(arr,i)
          in
             case cmp(v,v') of
                 GREATER => false
               | _ => if i+1 = len then true else s(v',i+1)
          end
      where s <| {i:nat | i < size } 'a * int(i) -> bool
   in
      if len <= 1 then true else s(sub(arr,0),1)
   end
   where sorted <| ('a * 'a -> order) -> 'a array(size) -> bool
end (* end of the structure *)
```

A.4 Mergesort on Lists

```
structure Merge_Sort =
    struct
        datatype 'a listlist = Nil | Cons of 'a list * 'a listlist
        typeref 'a listlist of nat * nat
        with Nil <| 'a listlist(0,0)
           | Cons <| {l:nat,m:nat,n:nat}
                     'a list(l) * 'a listlist(m,n) -> 'a listlist(m+1,n+l)

        assert not <| bool -> bool
        and    rev <| {n:nat} 'a list(n) -> 'a list(n)
        and    hd <| { n:nat | n > 0 } 'a list(n) -> 'a

        fun('a)
            sort cmp ls =
            let
                fun merge([],ys) = ys
                  | merge(xs,[]) = xs
                  | merge(x::xs,y::ys) =
                    if cmp(x,y) then y::merge(x::xs,ys)
                    else x::merge(xs,y::ys)
                where merge <|
                {m:nat, n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)

                fun mergepairs'(ls as Cons(l,Nil)) = l
                  | mergepairs'(Cons(l1,Cons(l2,ls))) =
                    mergepairs'(Cons(merge(l1,l2),ls))
                where mergepairs' <|
                {m:nat, n:nat | m > 0} 'a listlist(m,n) -> 'a list(n)

                fun mergepairs(ls as Cons(l,Nil), k) = ls
                  | mergepairs(Cons(l1,Cons(l2,ls)),k) =
                    if k mod 2 = 1 then Cons(l1,Cons(l2,ls))
                    else mergepairs(Cons(merge(l1,l2),ls), k div 2)
                where mergepairs <|
                {m:nat, n:nat | m > 0}
                'a listlist(m,n) * int -> [m:nat | m > 0] 'a listlist(m,n)

                fun nextrun(run,[]) = (rev run,[])
                  | nextrun(run,x::xs) =
                    if cmp(x,hd(run)) then nextrun(x::run,xs)
                    else (rev run,x::xs)
                where nextrun <|
                {m:nat, n:nat | m > 0 }
```

```
                'a list(m) * 'a list(n) ->
                [p:nat, q:nat | p+q = m+n] ('a list(p) * 'a list(q))


                fun samsorting([], ls, k) = mergepairs'(ls)
                  | samsorting(x::xs, ls, k) =
                    let
                        val (run,tail) = nextrun([x],xs)
                    in
                        samsorting(tail, mergepairs(Cons(run,ls),k+1), k+1)
                    end
                where samsorting <|
                {l:nat,m:nat,n:nat | m+l > 0}
                'a list(l) * 'a listlist(m,n) * int -> 'a list(n+l)
            in
                case ls of [] => [] | _::_ => samsorting(ls, Nil, 0)
            end
        where sort <| {n:nat} ('a * 'a -> bool) -> 'a list(n) -> 'a list(n)

        fun('a)
            sorted (cmp) =
            let
                fun s (x::(rest as (y::_))) = not(cmp(x, y)) andalso s rest
                  | s l = true
                    where s <| 'a list -> bool
            in s end
        where sorted <| ('a * 'a -> bool) -> 'a list -> bool

    end (* end of mergeSort *)
```

A.5 A Byte Copy Function

This implementation of a byte copy function is used in the Fox project.

```
(* This is an optimized version of byte copy function used in the Fox
 * project. All the array bound checks can be eliminated. There are
 * 13 type annotations, which consists of roughly 20% of the code
 *)

structure BCopy =
  struct
    assert sub1 <| {n:nat, i:nat| i < n } array(n) * int(i) -> byte1
    and update1 <|
        {n:nat, i:nat| i < n } array(n) * int(i) * byte1 -> unit
```

```
assert sub2 <| {n:nat, i:nat| i + 1 < n } array(n) * int(i) -> byte2
and update2 <|
    {n:nat, i:nat| i + 1 < n } array(n) * int(i) * byte2 -> unit

assert sub4 <| {n:nat, i:nat| i + 3 < n } array(n) * int(i) -> byte4
and update4 <|
    {n:nat, i:nat| i + 3 < n } array(n) * int(i) * byte4 -> unit

assert << <| byte4 * int -> byte4
and    || <| byte4 * byte4 -> byte4
and    >> <| byte4 * int -> byte4

fun{m:nat, n:nat, endsrc:nat}
unaligned(src, srcpos, endsrc, dest, destpos) =
let
   fun loop(i,j) =
       if (i >= endsrc) then ()
       else (update1(dest, j, sub1(src, i)); loop(i+1, j+1))
   where loop <|
   {i:nat, j:nat | j + endsrc - i <= n } int(i) * int(j) -> unit
in
   loop(srcpos, destpos)
end
where unaligned <|
{srcpos:nat, destpos:nat | endsrc <= m /\ destpos + endsrc - srcpos <= n }
array(m) * int(srcpos) * int(endsrc) * array(n) * int(destpos) -> unit


fun{m:nat, n:nat, endsrc:nat}
common(src, srcpos, endsrc, dest, destpos) =
case endsrc - srcpos of
    1 => (update1(dest, destpos, sub1(src, srcpos)))

  | 2 => (update1(dest, destpos, sub1(src, srcpos));
          update1(dest, destpos+1, sub1(src, srcpos+1)))

  | 4 => (update1(dest, destpos, sub1(src, srcpos));
          update1(dest, destpos+1, sub1(src, srcpos+1));
          update1(dest, destpos+2, sub1(src, srcpos+2));
          update1(dest, destpos+3, sub1(src, srcpos+3)))

  | 8 => (update1(dest, destpos, sub1(src, srcpos));
          update1(dest, destpos+1, sub1(src, srcpos+1));
          update1(dest, destpos+2, sub1(src, srcpos+2));
          update1(dest, destpos+3, sub1(src, srcpos+3));
```

```
            update1(dest, destpos+4, sub1(src, srcpos+4));
            update1(dest, destpos+5, sub1(src, srcpos+5));
            update1(dest, destpos+6, sub1(src, srcpos+6));
            update1(dest, destpos+7, sub1(src, srcpos+7)))

    | 16 => (update1(dest, destpos, sub1(src, srcpos));
            update1(dest, destpos+1, sub1(src, srcpos+1));
            update1(dest, destpos+2, sub1(src, srcpos+2));
            update1(dest, destpos+3, sub1(src, srcpos+3));
            update1(dest, destpos+4, sub1(src, srcpos+4));
            update1(dest, destpos+5, sub1(src, srcpos+5));
            update1(dest, destpos+6, sub1(src, srcpos+6));
            update1(dest, destpos+7, sub1(src, srcpos+7));
            update1(dest, destpos+8, sub1(src, srcpos+8));
            update1(dest, destpos+9, sub1(src, srcpos+9));
            update1(dest, destpos+10, sub1(src, srcpos+10));
            update1(dest, destpos+11, sub1(src, srcpos+11));
            update1(dest, destpos+12, sub1(src, srcpos+12));
            update1(dest, destpos+13, sub1(src, srcpos+13));
            update1(dest, destpos+14, sub1(src, srcpos+14));
            update1(dest, destpos+15, sub1(src, srcpos+15)))

    | 20 => (update1(dest, destpos, sub1(src, srcpos));
            update1(dest, destpos+1, sub1(src, srcpos+1));
            update1(dest, destpos+2, sub1(src, srcpos+2));
            update1(dest, destpos+3, sub1(src, srcpos+3));
            update1(dest, destpos+4, sub1(src, srcpos+4));
            update1(dest, destpos+5, sub1(src, srcpos+5));
            update1(dest, destpos+6, sub1(src, srcpos+6));
            update1(dest, destpos+7, sub1(src, srcpos+7));
            update1(dest, destpos+8, sub1(src, srcpos+8));
            update1(dest, destpos+9, sub1(src, srcpos+9));
            update1(dest, destpos+10, sub1(src, srcpos+10));
            update1(dest, destpos+11, sub1(src, srcpos+11));
            update1(dest, destpos+12, sub1(src, srcpos+12));
            update1(dest, destpos+13, sub1(src, srcpos+13));
            update1(dest, destpos+14, sub1(src, srcpos+14));
            update1(dest, destpos+15, sub1(src, srcpos+15));
            update1(dest, destpos+16, sub1(src, srcpos+16));
            update1(dest, destpos+17, sub1(src, srcpos+17));
            update1(dest, destpos+18, sub1(src, srcpos+18));
            update1(dest, destpos+19, sub1(src, srcpos+19)))

    | _ => unaligned(src, srcpos, endsrc, dest, destpos)
  where common <|
```

```
{srcpos:nat, destpos:nat |
 endsrc <= m /\ destpos + endsrc - srcpos <= n }
array(m) * int(srcpos) * int(endsrc) * array(n) * int(destpos) -> unit

fun{m:nat, n:nat, endsrc:nat}
sixteen(src, srcpos, endsrc, dest, destpos) =
let
   fun loop(i, j) =
       if i >= endsrc then ()
       else
           (update4(dest, j, sub4(src, i));
            update4(dest, j+4, sub4(src, i+4));
            update4(dest, j+8, sub4(src, i+8));
            update4(dest, j+12, sub4(src, i+12));
            loop(i+16, j+16))
   where loop <|
   {i:nat, j:nat | (endsrc - i) mod 16 = 0 /\ j + endsrc - i <= n }
   int(i) * int(j) -> unit
in
   loop(srcpos, destpos)
end
where sixteen <|
{srcpos:nat, destpos:nat |
 endsrc <= m /\ (endsrc - srcpos) mod 16 = 0 /\
 destpos + endsrc - srcpos <= n }
array(m) * int(srcpos) * int(endsrc) * array(n) * int(destpos) -> unit

fun{srcalign:nat}
aligned(src, srcpos, endsrc, dest, destpos, srcalign, bytes) =
let
   val front =
       (case srcalign of
            0 => 0
          | 1 => 3
          | 2 => 2
          | 3 => 1) <| [i:nat | (srcalign = 0 /\ i = 0) \/
                                (srcalign = 1 /\ i = 3) \/
                                (srcalign = 2 /\ i = 2) \/
                                (srcalign = 3 /\ i = 1)
                       ] int(i)

   val rest = bytes - front
   val tail = rest mod 16
   val middle = rest - tail
```

```
   val midsrc = srcpos + front
   val middest = destpos + front

   val backsrc = midsrc + middle
   val backdest = middest + middle
in
   unaligned(src, srcpos, midsrc, dest, destpos);
   sixteen(src, midsrc, backsrc, dest, middest);
   unaligned(src, backsrc, endsrc, dest, backdest)
end
where aligned <|
{m:nat, n:nat, srcpos:nat, endsrc:nat, destpos:nat, bytes:nat |
 endsrc <= m /\ srcpos + bytes = endsrc /\
 destpos + bytes <= n /\ 16 <= bytes }
array(m) * int(srcpos) * int(endsrc) *
array(n) * int(destpos) * int(srcalign) * int(bytes) -> unit


fun{m:nat, n:nat, endsrc:nat}
eightlittle(src, srcpos, endsrc, dest, destpos) =
let
   assert makebyte2 <| byte4 -> byte2
   and    makebyte4 <| byte2 -> byte4

   fun loop(i, j, carry) =
   if i >= endsrc then update2(dest, j, makebyte2(carry))
   else
       let
          val srcv = sub4(src, i)
       in
          update4(dest, j, ||(carry, <<(srcv, 16)));
          let
             val i = i + 4
             val j = j + 4
             val carry = >>(srcv, 16)
             val srcv = sub4(src, i)
          in
             update4(dest, j, ||(carry, <<(srcv, 16)));
             loop(i+4, j+4, >>(srcv, 16))
          end
       end
   where loop <|
   {i:nat, j:nat |
    i <= endsrc /\ (endsrc - i) mod 8 = 0 /\ j + endsrc - i + 2 <= n }
   int(i) * int(j) * byte4 ->  unit
```

```
in
   loop(srcpos+2, destpos, makebyte4(sub2(src, srcpos)))
end
where eightlittle <|
{srcpos:nat, destpos:nat |
 endsrc <= m /\ srcpos <= endsrc /\
 (endsrc - srcpos) mod 8 = 2 /\  destpos + endsrc - srcpos <= n }
array(m) * int(srcpos) * int(endsrc) * array(n) * int(destpos) -> unit

fun{m:nat, n:nat, endsrc:nat}
eightbig(src, srcpos, endsrc, dest, destpos) =
let
   assert makebyte2 <| byte4 -> byte2
   and    makebyte4 <| byte2 -> byte4

   fun loop(i, j, carry) =
       if i >= endsrc then update2(dest, j, makebyte2(>>(carry, 16)))
       else
           let
               val srcv = sub4(src, i)
           in
               update4(dest, j, ||(carry, >>(srcv, 16)));
               let
                  val i = i + 4
                  val j = j + 4
                  val carry = <<(srcv, 16)
                  val srcv = sub4(src, i)
               in
                  update4(dest, j, ||(carry, >>(srcv, 16)));
                  loop(i + 4, j + 4, <<(srcv, 16))
               end
           end
   where loop <|
   {i:nat, j:nat |
    i <= endsrc /\ (endsrc - i) mod 8 = 0 /\ j + endsrc - i + 2 <= n }
   int(i) * int(j) * byte4 ->  unit
in
   loop(srcpos + 2, destpos, <<(makebyte4(sub2(src, srcpos)), 16))
end
where eightbig <|
{srcpos:nat, destpos:nat | endsrc <= m /\ srcpos <= endsrc /\
 (endsrc - srcpos) mod 8 = 2 /\  destpos + endsrc - srcpos <= n }
array(m) * int(srcpos) * int(endsrc) * array(n) * int(destpos) -> unit

assert endian <| int and Little <| int
```

```
fun eight(src, srcpos, endsrc, dest, destpos) =
    if endian = Little then eightbig(src, srcpos, endsrc, dest, destpos)
    else eightlittle(src, srcpos, endsrc, dest, destpos)
where eight <|
{m:nat, n:nat, endsrc:nat, srcpos:nat, destpos:nat |
 endsrc <= m /\ srcpos <= endsrc /\
 (endsrc - srcpos) mod 8 = 2 /\ destpos + endsrc - srcpos <= n }
array(m) * int(srcpos) * int(endsrc) * array(n) * int(destpos) -> unit

fun{srcalign:nat}
semialigned(src, srcpos, endsrc, dest, destpos, srcalign, bytes) =
let
   val front =
       (case srcalign of
            0 => 2
          | 2 => 0
          | 1 => 1
          | 3 => 3) <| [i:nat | (srcalign = 0 /\ i = 2) \/
                                (srcalign = 2 /\ i = 0) \/
                                (srcalign = 1 /\ i = 1) \/
                                (srcalign = 3 /\ i = 3)
                       ] int(i)
   val rest = bytes -front
   val tail = (rest - 2) mod 8
   val middle = rest - tail
   val midsrc = srcpos + front
   val middest = destpos + front
   val backsrc = midsrc + middle
   val backdest = middest + middle
in
   unaligned(src, srcpos, midsrc, dest, destpos);
   eight(src, midsrc, backsrc, dest, middest);
   unaligned(src, backsrc, endsrc, dest, backdest)
end
where semialigned <|
{m:nat, n:nat, srcpos:nat, endsrc:nat, destpos:nat, bytes:nat |
 endsrc <= m /\ srcpos + bytes = endsrc /\
 destpos + bytes <= n /\ 16 <= bytes }
array(m) * int(srcpos) * int(endsrc) *
array(n) * int(destpos) * int(srcalign) * int(bytes) -> unit


fun copy(src, srcpos, bytes, dest, destpos) =
    if (bytes < 25) then
```

```
          common(src, srcpos, srcpos + bytes, dest, destpos)
        else
            let
               val srcalign = srcpos mod 4
               val destalign = destpos mod 4
               val endsrc = srcpos + bytes
            in
               if srcalign = destalign then
                 aligned(src, srcpos, endsrc, dest, destpos, srcalign, bytes)
               else if (srcalign + destalign) mod 2 = 0 then
                      semialigned(src, srcpos, endsrc, dest,
                                  destpos, srcalign, bytes)
                    else unaligned(src, srcpos, endsrc, dest, destpos)
            end
    where copy <|
    {m:nat, n:nat, srcpos:nat, bytes:int, destpos:nat |
     srcpos + bytes <= m /\ destpos + bytes <= n }
    array(m) * int(srcpos) * int(bytes) * array(n) * int(destpos) -> unit
  end (* end of the structure BCopy *)
```

Bibliography

Andrews, P., M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi (1996, June). TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning 16*(3), 321–353.

Augustsson, L. (1998). Cayenne – a language with dependent types. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, pp. 239–250.

Augustsson, L., T. Coquand, and B. Nordström (1990). A short description of another logical framework. In *Proceedings of the First Workshop on Logical Frameworks*, pp. 39–42.

Barendregt, H. P. (1992). Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. Maibaum (Eds.), *Handbook of Logic in Computer Science*, Volume II, pp. 117–441. Oxford: Clarendon Press.

Bershad, B., S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers (1995). Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pp. 267–284.

Bird, R. J. (1990). A calculus of functions for program derivation. In D. Turner (Ed.), *Topics in Functional Programming*.

Buhler, J. (1995, September). The Fox project: A language-structured approach to networking software. *ACM Crossroads 2.1*. Electronic Publication available as `http://www.acm.org/crossroads/xrds2-1/foxnet.html`.

Burstall, R. M. and J. L. Darlington (1977, January). A transformation system for developing recursive programs. *Journal of ACM 24*(1), 44–67.

Church, A. (1940). A formulation of the simple type theory of types. *Journal of Symbolic Logic 5*, 56–68.

Church, A. and J. B. Rosser (1936). Some properties of conversion. *Transactions of the American Mathematical Society 39*, 472–482.

Clément, D., J. Despeyroux, T. Despeyroux, and G. Kahn (1986). A simple applicative language: Mini-ml. In *Proceedings of 1986 Conference on LISP and Functional Programming*, pp. 13–27.

Constable, R. L. et al. (1986). *Implementing Mathematics with the NuPrl Proof Development System*. Englewood Cliffs, New Jersey: Prentice-Hall.

Constable, R. L. and S. F. Smith (1987, June). Partial objects in constructive type theory. In *Proceedings of Symposium on Logic in Computer Science*, Ithaca, New York, pp. 183–193.

Coquand, T. (1991). An algorithm for testing conversion in type theory. In G. Plotkin and G. Huet (Eds.), *Logical Frameworks*, pp. 255–279. Cambridge University Press.

Coquand, T. (1992). Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pp. 85–92.

Coquand, T. and G. Huet (1985). Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger (Ed.), *EUROCAL '85*, Volume 203 of *Lecture Notes in Computer Science*, Berlin, pp. 151–184. Springer-Verlag.

Coquand, T. and G. Huet (1986, May). The calculus of constructions. Rapport de Recherche 530, INRIA, Rocquencourt, France.

Coquand, T. and G. Huet (1988, February–March). The calculus of constructions. *Information and Computation 76*(2–3), 95–120.

Dantzig, G. and B. Eaves (1973). Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A) 14*, 288–297.

de Bruijn, N. G. (1980). A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley (Eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 579–606. London: Academic Press.

Dijkstra, E. W. (1975, August). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*(8), 453–457.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Englewood Cliffs, New Jersey: Prentice-Hall.

Draves, S. (1997). *Automatic Program Specialization for Interactive Media*. Ph. D dissertation, Carnegie Mellon University. Available as Technical Report No. CMU-CS-97-159.

Feferman, S. (1979). Constructive theories of functions and classes. In M. Boffa, D. van Dalen, and K. MacAloon (Eds.), *Logic Colloquium '78*. North-Holland.

Floyd, R. W. (1967). Assigning meanings to programs. In J. T. Schwartz (Ed.), *Mathematical Aspects of Computer Science*, Volume 19 of *Proceedings of Symposia in Applied Mathematics*, Providence, Rhode Island, pp. 19–32. American Mathematical Society.

Freeman, T. (1994, March). *Refinement Types for ML*. Ph. D. dissertation, Carnegie Mellon University. Available as Technical Report CMU-CS-94-110.

Freeman, T. and F. Pfenning (1991). Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Ontario, pp. 268–277.

Frühwirth, T. (1992). Constraint simplification rules. Technical Report ECRC-92-18, European Computer-Industry Center, ECRC GMBH, Arabellastr, 17 D-8000 München 81, Germany.

Gupta, R. (1994). Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems 2*(1–4), 135–150.

Harper, R., P. Lee, and F. Pfenning (1998, January). The Fox project: Advanced language technology for extensible systems. Technical Report CMU-CS-98-107, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. (Also published as Fox Memorandum CMU-CS-FOX-98-02).

Harper, R., J. C. Mitchell, and E. Moggi (1990). Higher-order modules and the phase distinction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pp. 341–354.

Harper, R. W., F. Honsell, and G. D. Plotkin (1993, January). A framework for defining logics. *Journal of the ACM 40*(1), 143–184.

Hayashi, S. (1990). An introduction to PX. In G. Huet (Ed.), *Logical Fundation of Functional Programming*. Addison-Weysley.

Hayashi, S. (1991). Singleton, union and intersection types for program extraction. In A. R. Meyer (Ed.), *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pp. 701–730.

Hayashi, S. and H. Nakano (1988). *PX: A Computational Logic*. The MIT Press.

Hoare, C. A. R. (1969, October). An axiomatic basis for computer programming. *Communications of the ACM 12*(10), 576–580 and 583.

Honsell, F., I. A. Mason, S. Smith, and C. Talcott (1995, 15 May). A variable typed logic of effects. *Information and Computation 119*(1), 55–90.

Hudak, P., S. L. Peyton Jones, and P. Wadler (1992, May). Report on the programming language Haskell, a non-strict purely-functional programming language, Version 1.2. *SIGPLAN Notices 27*(5).

Hughes, J., L. Pareto, and A. Sabry (1996). Proving the correctness of reactive systems using sized types. In *Conference Record of 23rd ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 410–423.

Jackson, D., J. Somesh, and C. A. Damon (1996). Faster checking of software specifications. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles of Programming Languages*.

Jaffar, J. and M. J. Maher (1994, May/July). Constraint logic programming: a survey. *Journal of Logic Programming 19/20*, 503–581. Special 10th Anniversary Issue.

Jay, C. and M. Sekanina (1996). Shape checking of array programs. Technical Report 96.09, University of Technology, Sydney, Australia.

Kahn, G. (1987). Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pp. 22–39. Springer-Verlag LNCS 247.

Kahrs, S., D. Sannella, and A. Tarlecki (1994). Deferred compilation: The automation of runtime code generation. Report ECS-LFCS-94-300, University of Edinburgh.

Kolte, P. and M. Wolfe (1995, June). Elimination of redundant array subscript ckecks. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*. ACM Press.

Lou, Z. (1989). ECC: an extended Calculus of Constructions. In R. Parikh (Ed.), *Proceeding of Fourth Annual Symposium on Logic in Computer Science*, pp. 386–395. IEEE computer Society Press.

Lou, Z. (1991). A unifying theory of dependent types: the schematic approach. Technical Report LFCS-92-202, University of Edinburgh.

Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Naples, Italy: Bibliopolis.

Martin-Löf, P. (1985). Constructive mathematics and computer programming. In C. R. A. Hoare (Ed.), *Mathematical Logic and Programming Languages*. Prentice-Hall.

Mendler, N. (1987, June). Recursive types and type constraints in second-order lambda calculus. In *Proceedings of Symposium on Logic in Computer Science*, pp. 30–36. The Computer Society of the IEEE.

Michaylov, S. (1992, August). *Design and Implementation of Practical Constraint Logic Programming Systems*. Ph. D. thesis, Carnegie Mellon University. Available as Technical Report CMU-CS-92-168.

Milner, R. (1978, December). A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*(3), 348–375.

Milner, R. and M. Tofte (1991). *Commentary on Standard ML*. Cambridge, Massachusetts: MIT Press.

Milner, R., M. Tofte, and R. W. Harper (1990). *The Definition of Standard ML*. Cambridge, Massachusetts: MIT Press.

Milner, R., M. Tofte, R. W. Harper, and D. MacQueen (1997). *The Definition of Standard ML*. Cambridge, Massachusetts: MIT Press.

Moggi, E. (1989). Computational lambda-calculus and monads. In *Proceedings Fourth Annual Symposium on Logic in Computer Science*, pp. 14–23.

Morrisett, G. (1995). *Compiling with Types*. Ph. D dissertation, Carnegie Mellon University. Available as Technical Report No. CMU-CS-95-226.

Morrisett, G., D. Walker, K. Crary, and N. Glew (1998, January). From system F to typed assembly language. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 85–97.

Nakano, H. (1994, Octorber). A constructive logic behind the catch and throw mechanism. *Annals of Pure and Applied Logic 69*(2–3), 269–301.

Naur, P. (1966). Proof of algorithms by general snapshots. *BIT 6*, 310–316.

Necula, G. (1997). Proof-carrying code. In *Conference Record of 24th Annual ACM Symposium on Principles of Programming Languages*, pp. 106–119. ACM press.

Necula, G. and P. Lee (1998, June). The design and implementation of a certifying compiler. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 333–344. ACM press.

Nordström, B. (1993). The ALF proof editor. In *Proceedings of the Workshop on Types for proofs and programs*, pp. 253–266.

Parent, C. (1995). Synthesizing proofs from programs in the calculus of inductive constructions. In *Proceedings of the International Conference on Mathematics for Programs Constructions*, pp. 351–379. Springer-Verlag LNCS 947.

Paulin-Mohring, C. (1993). Inductive definitions in the system Coq: rules and properties. In M. Bezem and J. de Groote (Eds.), *Proceedings of the International Conference on Typed*

Lambda Calculi and Applications, Volume 664 of *Lecture Notes in Computer Science*, pp. 328–345.

Pfenning, F. (1989). Elf: A language for logic definition and verified metaprogramming. In *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, pp. 313–322.

Pfenning, F. (1993). On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae 19(1/2)*, 185–199.

Pfenning, F. and C. Paulin-Mohring (1989). Inductively defined types in the Calculus of Constructions. In *Proceedings of fifth International Conference on Mathematical Foundations of Programming Semantics*, Volume 442 of *Lecture Notes in Computer Science*, pp. 209–228.

Pollack, R. (1994). *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. Ph. D. dissertation, University of Edinburgh.

Pugh, W. and D. Wonnacott (1992). Eliminating false data dependences using the Omega test. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 140–151. ACM Press.

Pugh, W. and D. Wonnacott (1994, November). Experience with constraint-based array dependence analysis. Technical Report CS-TR-3371, University of Maryland.

Sabry, A. and M. Felleisen (1993). Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation 6*(3/4), 289–360.

Sannella, D. and A. Tarlecki (1989, February). Toward formal development of ML programs: Foundations and methodology. Technical Report ECS-LFCS-89-71, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

Schümann, C. and F. Pfenning (1998). Automated theorem proving in a simple meta-logic for lf. In *Proceedings of the 15th International Conference on Automated Deduction (CADE)*, pp. 286–300. Springer-Verlag LNCS 1421.

Shankar, N. (1996, May). Steps toward mechanizing program transformations using PVS. *Science of Computer Programming 26*(1–3), 33–57.

Shostak, R. E. (1977, October). On the SUP-INF method for proving Presburger formulas. *Journal of the ACM 24*(4), 529–543.

Sun Microsystems (1995). The Java language specification. Available as `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`.

Susuki, N. and K. Ishihata (1977). Implementation of array bound checker. In *4th ACM Symposium on Principles of Programming Languages*, pp. 132–143.

Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee (1996, June). A type-directed optimizing compiler for ML. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 181–192.

Tolmach, A. and D. P. Oliva (1998, July). From ML to Ada(!?!): Strongly-typed language interoperability via source translation. *Journal of Functional Programming 8*(4). (to appear).

Wright, A. (1995). Simple imperative polymorphism. *Journal of Lisp and Symbolic Computation 8*(4), 343–355.

Xi, H. (1997, November). Some examples of DML programming. Available at `http://www.cs.cmu.edu/~hwxi/DML/examples/`.

Xi, H. (1998, February). Some examples in DTAL. Available at `http://www.cs.cmu.edu/~hwxi/DTAL/examples/`.

Xi, H. (1999, January). Dead code elimination through dependent types. In *The First International Workshop on Practical Aspects of Declarative Languages*, San Antonio. (to appear).

Xi, H. and F. Pfenning (1998, June). Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, pp. 249–257.

Xi, H. and F. Pfenning (1999, January). Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, San Antonio. (to appear).

Zenger, C. (1997). Indexed types. *Theoretical Computer Science 187*, 147–165.

Zenger, C. (1998). *Indizierte Typen*. Ph. D. thesis, Fakultät für Informatik, Universität Karlsruhe. Forthcoming.

Index