Polymorphic Type Inference and Abstract Data Types

by

Konstantin Läufer

A dissertation submitted in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Department of Computer Science
New York University
July, 1992

Approved
Professor Benjamin F. Goldberg
Research Advisor

Abstract

Many statically-typed programming languages provide an *abstract data type* construct, such as the package in Ada, the cluster in CLU, and the module in Modula2. However, in most of these languages, instances of abstract data types are not first-class values. Thus they cannot be assigned to a variable, passed as a function parameter, or returned as a function result.

The higher-order functional language ML has a strong and static type system with parametric polymorphism. In addition, ML provides type reconstruction and consequently does not require type declarations for identifiers. Although the ML module system supports abstract data types, their instances cannot be used as first-class values for type-theoretic reasons.

In this dissertation, we describe a family of extensions of ML. While retaining ML's static type discipline, type reconstruction, and most of its syntax, we add significant expressive power to the language by incorporating first-class abstract types as an extension of ML's free algebraic datatypes. In particular, we are now able to express

- multiple implementations of a given abstract type,
- heterogeneous aggregates of different implementations of the same abstract type, and
- dynamic dispatching of operations with respect to the implementation type.

Following Mitchell and Plotkin, we formalize abstract types in terms of existentially quantified types. We prove that our type system is semantically sound with respect to a standard denotational semantics.

We then present an extension of Haskell, a non-strict functional language that uses type classes to capture systematic overloading. This language results from incorporating existentially quantified types into Haskell and gives us first-class abstract types with type classes as their interfaces. We can now express heterogeneous structures over type classes. The language is statically typed and offers comparable flexibility to object-oriented lan-

guages. Its semantics is defined through a type-preserving translation to a modified version of our ML extension.

We have implemented a prototype of an interpreter for our language, including the type reconstruction algorithm, in Standard ML.

In memory of my grandfather.

Acknowledgments

First and foremost, I would like to thank my advisors Ben Goldberg and Martin Odersky. Without their careful guidance and conscientious reading, this thesis would not have been possible. My work owes a great deal to the insights and ideas Martin shared with me in numerous stimulating and productive discussions. Special thanks go to Fritz Henglein, who introduced me to the field of type theory and got me on the right track with my research.

I would also like to thank my other committee members, Robert Dewar, Malcolm Harrison, and Ed Schonberg, for their support and helpful suggestions; the students in the NYU Griffin group for stimulating discussions; and Franco Gasperoni, Zvi Kedem, Bob Paige, Marco Pellegrini, Dennis Shasha, John Turek, and Alexander Tuzhilin for valuable advice.

My work has greatly benefited from discussions with Martin Adabi, Stefan Kaes, Tobias Nipkow, Ross Paterson, and Phil Wadler.

Lennart Augustsson promptly incorporated the extensions presented in this thesis into his solid Haskell implementation. His work made it possible for me to develop and test example programs.

I would sincerely like to thank all my friends in New York, who made this city an interesting, inspiring, and enjoyable place to live and work. This circle of special friends is the part of New York I will miss the most.

My sister Julia, my parents, my grandmother, and numerous friends came to visit me from far away. Their visits always made me feel close to home, as did sharing an apartment with my old friend Ingo during my last year in New York.

Elena has given me great emotional support through her love, patience, and understanding. She has kept my spirits up during critical phases at work and elsewhere.

Finally, I would like to thank my parents, who have inspired me through their own achievements, their commitment to education, their constant encouragement and support, and their confidence in me.

I dedicate this thesis to my grandfather, Heinrich Viesel. Although it has now been eight years that he is no longer with me, I want to thank him for giving me an enthusiasm to learn and for spending many wonderful times with me.

This research was supported in part by the Defense Advanced Research Project Agency under Office of Naval Research grants N00014-90-J1110 and N00014-91-5-1472.

Contents

1 Introduction

Many statically-typed programming languages provide an *abstract data type* construct, such as the package in Ada, the cluster in CLU, and the module in Modula2. In these languages, an abstract data type consists of two parts, *interface* and *implementation*. The implementation consists of one or more *representation types* and some *operations* on these types; the interface specifies the *names* and *types* of the operations accessible to the user of the abstract data type. However, in most of these languages, instances of abstract data types are not first-class values in the sense that they cannot be assigned to a variable, passed to a function as a parameter or returned by a function as a result. Besides, these languages require that types of identifiers be declared explicitly.

1.1 Objectives

This dissertation seeks to answer the following question:

> *Is it feasible to design a high-level programming language that satisfies the following criteria*:

1. *Strong and static typing*: If a program is type-correct, no type errors occur at runtime.

2. *Type reconstruction*: Programs need not contain any type declarations for identifiers; rather, the typings are implicit in the program and can

be reconstructed at compile time.

3. *Higher-order functional programming*: Functions are first-class values; they may be passed as parameters or returned as results of a function, and an expression may evaluate to a function.

4. *Parametric polymorphism*: An expression can have different types depending on the context in which it is used; the set of allowable contexts is determined by the unique *most general type* of the expression.

5. *Extensible abstract types with multiple implementations*: The specification of an abstract type is separate from its (one or more) implementations; code written in terms of the specification of an abstract type applies to any of its implementations; more implementations may be added later in the program.

6. *First-class abstract types*: Instances of abstract types are also first-class values; they can be combined to heterogeneous aggregates of different implementations of the same abstract type.

From a language design point of view, criterion 1 is important for programming safety, criteria 2, 3, 4, and 6 are desirable for conciseness and flexibility of programming, and criterion 5 is crucial for writing reusable libraries and extensible systems.

1.2 Approach

The functional language ML [MTH90] already satisfies criteria 1 through 4 fully, and criteria 5 and 6 in a limited, mutually exclusive way. For this reason and for the extensive previous work on the type theory of ML and related languages, we choose ML as a starting point for our own work.

In this dissertation, we describe a family of extensions of ML. While retaining ML's static type discipline and most of its syntax, we add significant expressive power to the language by incorporating first-class abstract types as an extension of ML's free algebraic datatypes[1]. The extensions described

are independent of the evaluation strategy of the underlying language; they apply equally to strict and non-strict languages. In particular, we are now able to express

- multiple implementations of a given abstract type,
- heterogeneous aggregates of different implementations of the same abstract type, and
- dynamic dispatching of operations with respect to the implementation type.

Note that a limited form of heterogenicity may already be achieved in ML by building aggregates over a free algebraic datatype. However, this approach is not satisfactory because all implementations, corresponding to the alternatives of the datatype, have to be fixed when the datatype is defined. Consequently, such a datatype is not extensible and hence useless for the purpose of, for example, writing a library function that we expect to work for any future implementation of an abstract type.

ML also features several constructs that provide some form of data abstraction. The limitations of these constructs are further discussed in Chapter 2.

1.3 Dissertation Outline

The chapters in this dissertation are organized as follows:

- **Chapter 2. Preliminaries.** In this chapter, we review the preliminary notions and concepts used in the course of the dissertation. First, we give an overview of the functional languages ML and Haskell and discuss the shortcomings of data abstraction in ML. Then, we describe the untyped and several typed λ-calculi and existentially quantified types as a formal basis for our type-theoretic considerations. Further, we discuss standard and order-sorted unification algorithms, which are used

[1] ML's version of a variant record in Pascal or Ada.

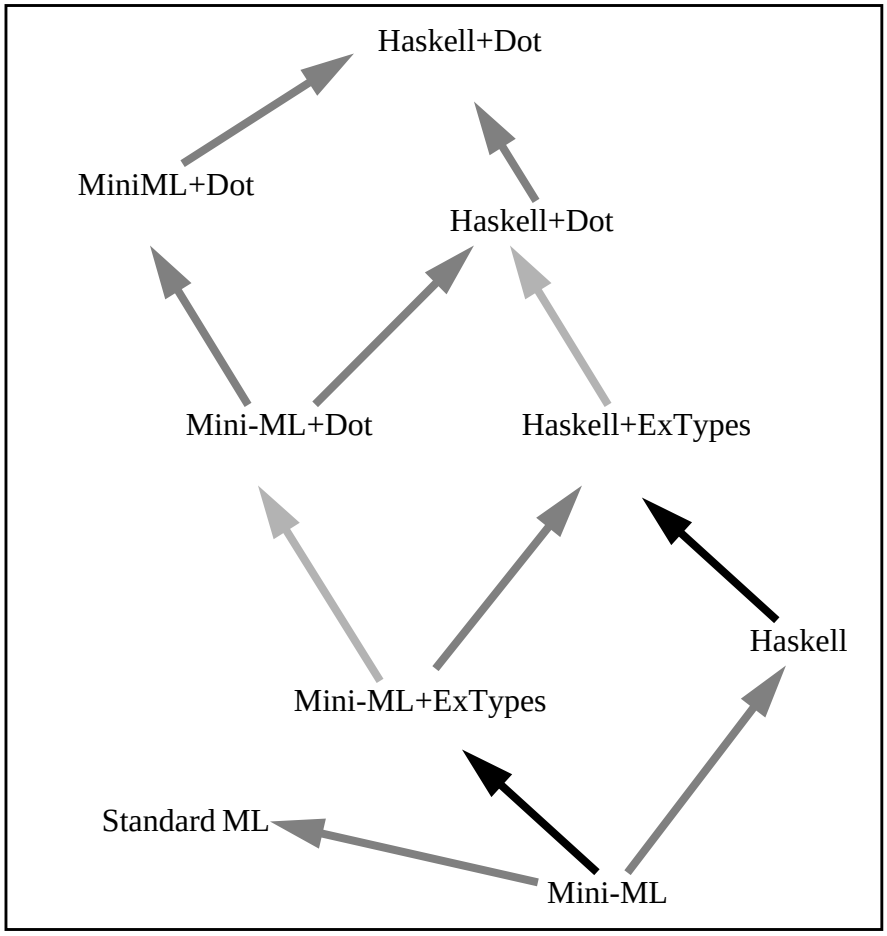
in type reconstruction algorithms. Finally, we give a review of domains and ideals, which we use as a semantic model for the languages we discuss.

- **Chapter 3. An Extension of ML with First-Class Abstract Types.** This chapter presents a semantic extension of ML, where the component types of a datatype may be existentially quantified. We show how datatypes over existential types add significant flexibility to the language without even changing ML syntax. We then describe a deterministic Damas-Milner type inference system [DM82] [CDDK86] for our language, which leads to a syntactically sound and complete type reconstruction algorithm. Furthermore, the type system is shown to be semantically sound with respect to a standard denotational semantics.
- **Chapter 4. An Extension of ML with a Dotless Dot Notation.** In this chapter, we describe a further extension of our language. The use of existential types in connection with an elimination construct (**`open`** or **`abstype`**) is impractical in certain programming situations; this is discussed in [Mac86]. A formal treatment of the dot notation, an alternative used in actual programming languages, is found in [CL90]. This notation assumes the same representation type each time a value of existential type is accessed, provided that each access is via the same identifier. We describe an extension of ML with an analogous notation. A type reconstruction algorithm is given, and semantic soundness is shown by translating into the language from Chapter 3.
- **Chapter 5. An Extension of Haskell with First-Class Abstract Types.** This chapter introduces an extension of the functional language Haskell [HPJW$^+$92] with existential types. Existential types combine well with the systematic overloading polymorphism provided by Haskell type classes [WB89]; this point is first discussed in [LO91]. Briefly, we extend Haskell's **`data`** declaration in a similar way as the

ML datatype declaration above. In Haskell, it is possible to specify what type class a (universally quantified) type variable belongs to. In our extension, we can do the same for existentially quantified type variables. This lets us use type classes as signatures of abstract data types; we can then construct heterogeneous aggregates over a given type class.

- **Chapter 6. Related Work, Future Work, and Conclusions.** This chapter concludes with a comparison with related work. Most previous work on existential types does not consider type reconstruction; other work that includes type reconstruction seems to be semantically unsound. We apparently are the first to permit polymorphic instantiation of variables of existential type in the body of the elimination construct. In our system, such variables are `let`-bound and therefore polymorphic, whereas other work treats them monomorphically. We give an outlook of future work, which includes further extensions with mutable state and a practical implementation.

The figure below illustrates the relationship between ML, Haskell, the languages introduced in this dissertation, and other possible extensions.


Haskell+Dot
MiniML+Dot
Haskell+Dot
Mini-ML+Dot
Haskell+ExTypes
Haskell
Mini-ML+ExTypes
Standard ML
Mini-ML
existential types
dot notation
type classes
mutable state

2 Preliminaries

In this chapter, we review the preliminary notions and concepts used in the course of the dissertation. First, we give an overview of the functional languages ML and Haskell and discuss in detail the shortcomings of data abstraction in ML. Then, we describe the untyped and several typed λ-calculi and existentially quantified types as a formal basis for our type-theoretic work below. Further, we discuss standard and order-sorted unification algorithms, which are used in type reconstruction algorithms for implicitly typed languages. Finally, we give a brief review of domains and ideals, which we use as a semantic model for the languages we discuss.

2.1 The Languages ML and Haskell

This section gives an overview of the functional languages ML and Haskell and discusses the shortcomings of the data abstraction constructs provided by ML. We assume some general background in programming languages; prior exposure to a statically typed functional language is helpful.

2.1.1 ML

We present a few programming examples that illustrate the relevant core of ML [MTH90] and its type system. For a full introduction, see [Har90]. The syntax of core expressions is defined recursively as constants, identifiers, and three constructs:

Constants	c	::= **0** | **1** | …
Identifiers	x	::= **x** | **y** | …
Abstractions	f	::= **fn** x **=>** e
Applications	a	::= e e'
Bindings	b	::= **let val** x = e **in** e'
Expressions	e	::= c | x | f | a | b

We also assume that a conditional construct **if** and a fixed-point operator **fix** are predefined.

To bind an identifier, we can just write

```
val x = e
```

which corresponds to an implicit **let** binding whose body encompasses the rest of the program. For functions, we can write

```
fun f x = e
```

instead of

```
val rec f = fn x => e
```

If the function is not recursive, that is, **f** is not called in *e*, the keyword **rec** may be omitted. The simplest polymorphic function is the identity function, given by

```
fn x => x
```

which simply returns its argument. Its semantics is clearly independent of the type of its argument. The following is an example of a higher-order function definition:

```
fun compose f g = fn x => f(g(x))
```

For the expression **f(g(x))** to be well-typed, the following assumptions about the types of **f**, **g**, and **x** must hold for some types **'a**, **'b**, and **'c**[1].

```
x : 'a
f : 'b -> 'c
g : 'a -> 'b
```

Under these assumptions, **compose** has the type

```
compose : ('b -> 'c) -> ('a -> 'b) -> ('a -> c)
```

where **->** is the function type constructor. Since this assumption holds for any types **'a**, **'b**, and **'c**, we can think of this type as universally quantified *type scheme* over the type variables, written as

$$\forall\alpha\forall\beta\forall\gamma.\ (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

We can now define a function that composes another function with itself:

```
fun twice f = compose f f
```

The type inferred for **twice** is

```
twice : ('a -> 'a) -> ('a -> 'a)
```

and we can apply **twice** as follows:

```
fun succ x = x + 1
(twice succ) 3
```

evaluating to **5**. It is important to note that in the definition of **twice**, both occurrences of the argument **f** are required to have the same type. Consequently, **'a** = **'b** = **'c** in this instance of **compose**.

The parameters of a function abstraction, henceforth called λ-bound identifiers, behave differently from **let**-bound identifiers:

- All occurrences of a λ-bound identifier have to have the same type.

[1] ML uses quoted letters to represent the Greek letters often used in type expressions.

- Each occurrence of a **let**-bound identifier may have a different type, which has to be an instance of the *most general* or *principal type* inferred for that identifier.

Furthermore, ML has a built-in type constructor for (homogeneous) lists, which is parameterized by the element type. Predefined constants and functions on lists include:

```
nil : 'a list
::  : 'a * 'a list -> 'a list
hd  : 'a list -> 'a
tl  : 'a list -> 'a list
null: 'a list -> bool
```

Lists are written in the form

[e_1**,** $\ldots$**,** e_n**]**

For instance,

```
(compose hd hd) [[1,2],[3,4,5]]
```

is type-correct and evaluates to **1**, while

```
(twice hd) [[1,2],[3,4,5]]
```

is not type-correct, since the type of **hd** is not of the form **'a -> 'a**.

Lastly, ML offers user-defined free algebraic datatypes. A datatype declaration of the form

datatype [*arg*] T **=** K_1 **of** τ_1 **|** $\ldots$ **|** K_n **of** τ_n

declares a type (or a type constructor, if arguments are present) T, where K_i's are value constructor functions of types $\tau_i \rightarrow (arg\ T)$. Value constructors can also lack the argument, in which case they are constants. The predefined **list** type can actually be written as a datatype:

```
datatype α list = nil | cons of 'a * 'a list
```

Values whose type is such a datatype can be constructed by applying a value constructor to an argument of appropriate type. They can be deconstructed by means of a *pattern-matching* **`let`** expression of the (simplified) form

`let val` *K x* **`=`** *e* **`in`** *e'*

For example,

```
val cons(x,xs) = [1,2,3]
```

would decompose the list on the right-hand side, binding **`x`** to **`1`** and **`xs`** to **`[2,3]`**.

2.1.2 Shortcomings of Abstract Type Constructs in ML

ML already provides three distinct constructs that can be used to describe abstract data types:

- The **`abstype`** mechanism is used to declare an abstract data type with a single implementation. It has been partially superseded by the module system.
- The ML module system provides signatures, structures, and functors. Signatures act as interfaces of abstract data types and structures as their implementations; functors are essentially parametrized structures. Several structures may share the same signature, and a single structure may satisfy several signatures. However, structures are not first-class values in ML for type-theoretic reasons discussed in [Mac86] [MH88]. This leads to considerable difficulties in a number of practical programming situations. The following example illustrates

how an abstract type **STACK** is programmed in the ML module system:

```
signature ELEM = sig
    type elem
    val init : elem
end

signature STACK = sig
    type elem
    type stack
    val empty   : stack
    val push    : elem -> stack -> stack
    val pop     : stack -> stack
    val top     : stack -> elem
    val isempty : stack -> bool
end

functor ListStack(Elem : ELEM) : STACK = struct
    type elem  = Elem.elem
    type stack = elem list
    val empty   = []
    fun push x xs = x :: xs
    val pop     = tl
    val top     = hd
    val isempty = null
end

functor ArrayStack(Elem : ELEM) : STACK = struct
    type elem = Elem.elem
    type stack = int ref * elem array
    val maxElem = 100
    val empty =
            (ref 0,Array.array(maxElem,Elem.init))
    fun push x (i,s) =
            (inc i; Array.update(s,!i,x); (i,s))
    fun pop(i,s) = (dec i; (i,s))
    fun top(i,s) = Array.sub(s,!i)
```

```
    fun isempty(i,s) = !i = 0
end

structure IntElem = struct
    type elem = int
    val init = 0
end

structure IntListStack = ListStack(IntElem)
structure IntArrayStack = ArrayStack(IntElem)
```

Note that two different implementations of **`STACK`** are given. However, the types **`IntListStack.stack`** and **`IntArrayStack.stack`** are different; thus we cannot construct, for example, the following list:

```
[IntListStack.empty,IntArrayStack.empty]
```

- Abstract data types can be implemented as a tuple (or record) of closures; the hidden bindings shared between the closures correspond to the representation, and the closures themselves correspond to the operations. The type of the tuple corresponds to the interface. A discussion of this approach is found in [Ode91]. The following example illustrates a use of a heterogeneous list of **`int Stack`**'s.

```
datatype 'a Stack =
    stack of {empty   : unit -> 'a Stack,
              push    : 'a -> 'a Stack,
              pop     : unit -> 'a Stack,
              top     : unit -> 'a,
              isempty : unit -> bool}

fun makeListStack xs =
    stack{empty  = fn() => makeListStack [],
          push   = fn x => makeListStack(x::xs),
          pop    = fn() => makeListStack(tl xs),
          top    = fn() => hd xs,
          isempty= fn() => null xs}
```

```
fun makeArrayStack xs = ...

fun empty  (stack{empty=e,...})   = e()
fun push y (stack{push=pu,...})   = pu y
fun pop    (stack{pop=po,...})    = po()
fun top    (stack{top=t,...})     = t()
fun isempty(stack{isempty=i,...}) = i()

map (push 8) [makeListStack [2,4,6],
              makeArrayStack [3,5,7]]
```

The shortcoming of this approach is that the internal representation of an instance of an abstract type is completely encapsulated; consequently, the extensibility of the abstract type is severely limited. The next example of an abstract type **`Mult`** supporting a **`square`** operation illustrates this limitation:

```
datatype Mult = mult of {square: unit -> Mult}

fun makeMult(i,f) =
    mult{square = fn() => makeMult(f(i,i),f)}

fun square(mult{square=s}) = s()

map square
    [makeMult(3,   op *: int * int -> int),
     makeMult(7.5, op *: real * real -> real)]
```

The problem arises when we want to define an additional operation on **`Mult`**, say **`cube`**. In this case, we need to add another field to the record component type of **`Mult`**, and we even need to change the definitions

of **makeMult** and **square**, although the latter was defined outside of **makeMult**:

```
datatype Mult = mult of {square: unit -> Mult,
                         cube  : unit -> Mult}

fun makeMult(i,f) =
    mult{square = fn() => makeMult(f(i,i),f),
         cube   = fn() => makeMult(f(i,f(i,i)),f)}

fun square(mult{square=s,...}) = s()
```

It is possible to work around this limitation using nested records as described in [Ode91].

Another, more serious limitation of the encapsulation imposed by the closure approach becomes apparent when we model abstract types with operations involving another argument of the same abstract type. Consider the following attempt at describing an abstract type **Tree**:

```
datatype Tree = tree of {eq    : Tree -> bool,
                         right : unit -> Tree,
                         left  : unit -> Tree,
                         ...}
```

The **eq** function could then be implemented by converting two trees to a common representation and comparing them. Suppose now that we want to compare two subtrees of the same tree. There is no obvious way to take advantage of the knowledge that both subtrees have the same representation; they still need to be converted before the comparison.

2.1.3 Haskell

The functional programming language Haskell [HPJW+92] has a polymorphic type discipline similar to ML's. In addition, it uses *type classes* as a systematic approach to operator overloading. Type classes capture common sets of operations, for example multiplication, which is common to both **int** and

real types. A particular type may be an instance of a type class and has an operation corresponding to each operation defined in the type class. Further, type classes may be arranged in a class hierarchy, in the sense that a derived type class captures all operations of its superclasses and may add new ones. Type classes were first introduced in the article [WB89], which also gives additional motivating examples and shows how Haskell programs are translated to ML programs.

The syntax of the Haskell core consists of essentially the same expressions as the ML core, with the addition of class and instance declarations of the following form:

class *C a* **where**
    op_1 **::** τ_1
    …
    op_n **::** τ_n

instance *C t* **where**
    op_1 **=** e_1
    …
    op_n **=** e_n

To motivate the type class approach, consider the overloading of mathematical operators in ML. Although **4*4** and **4.7*4.7** are valid ML expressions, we cannot define a function such as

```
fun square x = x * x
```

in ML, as the overloading of the operator * cannot be resolved unambiguously. In Haskell, we first declare a class **Num** to capture the operations **Int** and **Float** have in common:

```
class Num a where
    (-) :: a -> a
    (+) :: a -> a -> a
    (*) :: a -> a -> a
```

At this point, we can already type the **square** function, although we cannot use it yet, since we do not have any instances of **Num**. The typing is

```
square :: Num a => a -> a
```

which reads, "for any **a** that is an instance of **Num**, **square** has type **a -> a**." We then declare two instances of **Num**, assuming the existence of some predefined functions on **Int** and **Float**:

```
instance Num Int where
    (-) = intUMinus
    (+) = intAdd
    (*) = intMult

instance Num Float where
    (-) = floatUminus
    (+) = floatAdd
    (*) = floatMult
```

When we now write **square 4.0**, the type reconstructor finds out that **4.0** is of type **Float**, which in turn is an instance of **Num**. The multiplication used is **floatMult**, as specified in the instance declaration for **Float**. Given a definition of the function **map**, we can write the function

```
squarelist xs = map square xs
```

which squares each element in a list. It has type

```
squarelist :: Num a => [a] -> [a]
```

where **[a]** is the Haskell version of **'a list**.

Haskell also provides algebraic datatypes, which differ from the ones in ML only in that the formal arguments of the type constructor can be specified to be instances of a certain type class.

It should also be mentioned that Haskell is a *pure*, *non-strict* functional language, whereas ML is a *strict* language and provides mutable state in the form of *references*.

2.2 The Lambda Calculus, Polymorphism, and Existential Quantification

In this section, we describe the untyped, the simply typed, and the first-order polymorphic λ-calculi, which constitute the type-theoretic basis for functional languages such as ML and Haskell. We also give an introduction to existentially quantified types, which provide a type-theoretic description of abstract data types.

2.2.1 The Untyped λ-calculus

The untyped λ-calculus is a formal model of computation. While the λ-calculus is equivalent to Turing machines in computational power, its simple, functional structure lends itself as a useful model for reasoning about programs, in particular, functional programs. We give a brief introduction to the λ-calculus; a comprehensive reference is [Bar84].

λ-terms are defined as follows:

Constants[1]	c
Identifiers	x
Terms	$e ::= c \mid x \mid \lambda x.e \mid (e\ e')$

In a λ-abstraction a of the form $\lambda x.e$, where e is some λ-term, the variable x is said to be *bound* in a and is called a *bound variable*. Any variable y in e other than x that is not bound in a λ-abstraction inside e is said to occur *free* in a and is called a *free variable*. We assume that no free variable is identical to any bound variable within a λ-term.

The λ-calculus provides several *conversion rules* for transforming one λ-term into an equivalent one. The conversion rules are defined as follows:

[1] Constants are not actually part of the *pure* λ-calculus, but are a useful enrichment.

- β-*conversion*:

  $$(\lambda x.e)\, e' \Leftrightarrow e\,[e'/x]$$

  This rule models function application. It states that a λ-abstraction $\lambda x.e$ is applied to a term e' by replacing each free occurrence of x in e by a copy of e'. In addition, bound variables in e have to be renamed to avoid name conflict with variables that are free in e'. $e\,[e'/x]$ stands for this new term.

- α-*conversion*:

  $$\lambda x.e \Leftrightarrow \lambda y.e\,[y/x] \qquad y \notin FV(e)$$

  This rule states that the bound variable of a λ-abstraction may be renamed, provided that the renamed variable does not occur free in e.

- η-*conversion*:

  $$\lambda x.\,(e\ x) \Leftrightarrow e \qquad x \notin FV(e)$$

  This rule can be used to eliminate a redundant λ-abstraction, provided that the bound variable does not occur free in e.

- δ-*conversion*:

  The δ-rules define conversion of built-in constants and functions, for example,

  $$(\mathit{times}\ 3\ 4) \Leftrightarrow 12$$

We view the set of λ-terms as divided into α-*equivalence classes*; this means that any two λ-terms that can be transformed into one another via α-conversion are in the same equivalence class, and any one term is viewed as a representant of its α-equivalence class.

While conversion rules express that two terms are equivalent, *reduction rules* are used to evaluate a term. There are two reduction rules, β-reduction and δ-reduction; the most important rule, β-reduction, is given by

$$(\lambda x.e)\, e' \Rightarrow e\,[e'/x]$$

Semantically, a λ-term is evaluated by repeatedly applying reduction rules until no more reductions can be applied; the resulting term is said to be in *normal form*. A given λ-term may have several subterms to which β-reduction can be applied; such subterms are called reducible terms or *redexes*. An evaluation strategy where β-reduction is always applied to the leftmost outermost redex first is called is called *normal order* evaluation. A strategy where β-reduction is always applied to the leftmost innermost redex is called *applicative order* evaluation. In programming languages, normal order evaluation is often implemented by lazy or call-by-name evaluation, and applicative order evaluation is a special case of eager (call-by-value) evaluation. Normal order evaluation is *normalizing*, which means that it terminates for every term that has a normal form. Although applicative order evaluation does not guarantee termination, it is sometimes preferred in practice for efficiency reasons.

In the λ-calculus, recursion is expressed by the Y combinator, which is defined by the equation $Yf = f(Yf)$. The Y combinator can be defined by the following λ-abstraction:

$$Y = \lambda h.\,((\lambda x.\,(h\,(xx)))\,(\lambda x.\,(h\,(xx))))$$

A recursive function can then be expressed as a λ-term containing Y, for example the factorial,

$$Y\,(\lambda f.\lambda n.\,(\mathit{if}\ \ (\mathit{equal}\ n\ 0)\ 1\ (\mathit{times}\ n\ \ (f\,(\mathit{minus}\ n\ 1)))))$$

assuming suitable δ-rules for the built-in functions used.

2.2.2 The Simply Typed λ-Calculus

Typed λ-calculi are like the untyped λ-calculus, except that every bound identifier is given a *type*. The simply typed λ-calculus describes languages that have a notion of type. Informally, types are subsets of the set of all values that share a certain common structure, for example all integers, or all Booleans. An important difference between typed and untyped calculi is that typed calculi introduce the notion of (static) *type correctness* of a term, which one would like to check before trying to evaluate the term. The untyped λ-calculus could be regarded as a typed λ-calculus in which each identifier or constant has the same type *general* and all terms are type-correct. An comprehensive survey of typing in programming languages is [CW85].

As an example, consider the successor function, which we could define as

$$succ = \lambda n : int . n + 1$$

Assuming the typing $+ : int \times int \rightarrow int$, we would obtain the typing

$$succ : int \rightarrow int$$

where $\rightarrow$ is the function type constructor and $\times$ the tuple type constructor used for multiple function arguments. We could then define

$$twice = \lambda f : int \rightarrow int . \lambda x : int . f\,(f\ x)$$

and the term

$$(twice\ \ succ)\,4$$

would be type-correct and result in 6. On the other hand,

$$twice\ \ 7$$

would not be type-correct, since the type of 7 is int rather than $int \rightarrow int$.

We would like to formalize the notion of type correctness. For example, to guarantee that a function application is type-correct, it is enough to know that the argument term is of the same type as the domain type of the function. Then the type of the resulting term is the range type of the function. Such a rule is formally expressed by an *inference rule* consisting of zero or more *antecedents* and one *conclusion*. Each antecedent or conclusion is a *type judgment* of the form $A \vdash e : \tau$ where e is a well-formed term, τ a *well-formed*[1] type, and A a set of assumptions of the form $x : \tau$ stating that the identifier or constant x has type τ; $\vdash$ reads as "entails." For example, the rule governing function application is written as

$$\frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e\ e') : \tau}$$

and is read as: "If assumption set A entails type $\tau' \to \tau$ for expression e and if A entails type τ' for expression e', then A entails type τ for the application $(e\ e')$.

The type system of a typed λ-calculus is described by a system of such inference rules. Type-correct terms are those for which a type judgment can be derived within the given inference system.

The following inference system describes the simply typed λ-calculus:

(TAUT) $A \vdash x : A(x)$

(APP) $$\frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e\ e') : \tau}$$

(ABS) $$\frac{A[\tau'/x] \vdash e : \tau}{A \vdash \lambda x : \tau'.e : \tau' \to \tau}$$

[1] For our purposes, types are well-formed iff they are composed from the basic types *int*, *bool*, etc., by application of the type constructors $\to$ and $\times$.

where $A[\tau'/x]$ stands for the assumption set A extended with the assumption $x : \tau$.

One possible proof for the typing $twice : (int \to int) \to (int \to int)$ looks as follows[1]:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\left[\begin{smallmatrix} int/x \\ int \to int/f \end{smallmatrix}\right] \vdash x : int} \quad \dfrac{}{\left[\begin{smallmatrix} int/x \\ int \to int/f \end{smallmatrix}\right] \vdash f : int \to int}}{\left[\begin{smallmatrix} int/x \\ int \to int/f \end{smallmatrix}\right] \vdash f\ x : int} \quad \dfrac{}{\left[\begin{smallmatrix} int/x \\ int \to int/f \end{smallmatrix}\right] \vdash f : int \to int}}{\left[\begin{smallmatrix} int/x \\ int \to int/f \end{smallmatrix}\right] \vdash f(f\ x) : int}}{[int \to int/f] \vdash \lambda x : int.f(f\ x) : int \to int}}{\varnothing \vdash (\lambda f : int \to int.\lambda x : int.f(f\ x)) : (int \to int) \to (int \to int)}$$

When designing a statically typed programming language, we generally want type correctness to be *decidable*. That is, we would like to have an algorithm that decides, given a type judgment, whether there is a proof for this type judgment. We would also like the type system to be semantically *sound*, meaning that a type-correct program can be evaluated without, for example, trying to apply an argument to a term that is not a function.

2.2.3 The Typed λ-Calculus with `let`-Polymorphism

The typed λ-calculus with **`let`**-polymorphism is a formalization of the idea that there are λ-abstractions that have many different types depending on their argument terms. It provides a type-theoretic model for the language ML described in Section 2.1.1. As a motivating example, consider that the result of the (untyped) λ-abstraction

$$id = \lambda x.x$$

[1]The horizontal bars are read in top-down order.

always has the same type as its argument. We can express this by giving *id* the universally quantified type $id : \forall\alpha.\alpha \to \alpha$, where α is a *type variable* representing any well-formed type; this typing is read as: "for every type α, *id* has type $\alpha \to \alpha$." While $\forall\alpha.\alpha \to \alpha$ is not a well-formed type, it makes sense to think of α as a type parameter, which can be instantiated to the type of the argument passed to *id*. We therefore call constructs such as $\forall\alpha.\alpha \to \alpha$ *type schemes* or *polymorphic types*, whereas types that are not universally quantified are called *monomorphic types*. In our typed calculus, we can think of *id* as first parameterized by the argument type and then by the argument itself. This is expressed by a λ-abstraction enclosed by a Λ-abstraction, which denotes abstraction over a type argument:

$$id = \Lambda\alpha.\lambda x : \alpha.x$$

and its application to an argument has the form

$$id\ [int]\ 3,$$

where type arguments to a Λ-abstraction are enclosed in square brackets. Terms are called polymorphic or monomorphic depending on their type. In the typed λ-calculus with **let**-polymorphism, we do not allow arguments of λ-abstractions to be polymorphic. Consequently, a Λ-abstraction can only occur at the outermost level of a term or on the right side of a special binding construct that expresses the binding of a term to an identifier. This construct is called **let**-expression and is of the form **let** $x = e$ **in** e'.

The following inference system describes the typed λ-calculus with **let**-polymorphism, where τ's stand for types and σ's for type schemes.

(TAUT) $A \vdash x : A(x)$

(APP) $$\frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e\ e') : \tau}$$

$$\text{(ABS)} \quad \frac{A\,[\tau'/x] \;\vdash\; e : \tau}{A \;\vdash\; \lambda x : \tau'.e : \tau' \to \tau}$$

$$\text{(LET)} \quad \frac{A \;\vdash\; e : \sigma \qquad A\,[\sigma/x] \;\vdash\; e' : \tau}{A \;\vdash\; \textbf{let}\;\; x = e \;\; \textbf{in}\;\; e' : \tau}$$

$$\text{(INST)} \quad \frac{A \;\vdash\; e : \forall\alpha.\sigma}{A \;\vdash\; e\,[\tau] \;:\; \sigma\,[\tau/\alpha]}$$

$$\text{(GEN)} \quad \frac{A \;\vdash\; e : \sigma \qquad \alpha \notin FV(A)}{A \;\vdash\; \Lambda\alpha.e : \forall\alpha.\sigma}$$

Note that the ABS rule requires the expression from which the abstraction is constructed to be monomorphic, and the APP rule enforces that in an application the function and its argument have to be monomorphic.

The following is a sample proof in this system:

$$\dfrac{\dfrac{\dfrac{}{[\alpha/x] \;\vdash\; x : \alpha}}{\varnothing \;\vdash\; \lambda x : \alpha.x : \alpha \to \alpha}}{\varnothing \;\vdash\; \Lambda\alpha.\lambda x : \alpha.x : \forall\alpha.\alpha \to \alpha} \qquad \dfrac{\dfrac{}{[\forall\alpha.\alpha \to \alpha/id] \;\vdash\; id\,[int] \;:\; int \to int} \qquad \dfrac{}{[\forall\alpha.\alpha \to \alpha/id] \;\vdash\; 3 : int}}{[\forall\alpha.\alpha \to \alpha/id] \;\vdash\; id\,[int]\,3 : int}$$

$$\varnothing \;\vdash\; \textbf{let}\;\; id = \Lambda\alpha.\lambda x : \alpha.x \;\; \textbf{in}\;\; id\,[int]\,3 : int$$

The ML core language can be thought of as an implicitly typed version of the typed λ-calculus with **let**-polymorphism; this is discussed in detail in [MH88]. ML uses *type reconstruction* to compute the explicit type annotations of an implicitly typed expression. The problem of polymorphic type reconstruction was first discussed in [Mil78] and further developed in [DM82] and [Dam85].

2.2.4 Higher-Order Typed λ-Calculi

A considerable amount of research has focused on the second-order typed λ-calculus and higher-order systems [REFS]. Since all languages presented in this dissertation are extensions of the typed λ-calculus with **`let`**-polymorphism, we do not further discuss higher-order calculi here.

2.2.5 Existential Quantification

Existentially quantified types, or in short, existential types, are a type-theoretic formalization of the concept of abstract data types, which are featured in different forms by various programming languages.

In Ada, abstract types are expressed via *private* types. Consider as an example the following package specification, which describes an stack of integers:

```
package STACK_PKG is
    type STACK_TYPE is private;
    procedure PUSH(in out S: STACK_TYPE;
                   A : INTEGER);
    procedure POP(in out S: STACK_TYPE);
    ...
private
    type STACK_TYPE is ...;
    ...
end STACK_PKG;
```

We can then write in our program

```
use STACK_PKG;
```

and have access to the entities defined in the package specification without knowing or wanting to know how **`STACK_TYPE`** is defined in the package body. Since the program using the package works independently of the implementation of the package, we might wonder what type **`STACK_TYPE`** stands for in the program. An informal answer is, “some new type that is different from any other type in the program.”

Existential quantification is a formalization of the notion of abstract types; it is described in [CW85] and further explored in [MP88]. By stating that an expression e has existential type $\exists\alpha.\tau$, we mean that for some fixed, unknown type $\hat{\tau}$, e has type $\tau[\hat{\tau}/\alpha]$. e can thus be viewed as a pair consisting of a type component $\hat{\tau}$ and a value component of type $\tau[\hat{\tau}/\alpha]$. The components are accessed through an *elimination construct* of the form

open e **as** $\langle t, x\rangle$ **in** e'

In e', the type t stands for the hidden representation type of e, such that x can be used in e' with type $\tau[t/\alpha]$. To guarantee static typing, the type of e' must not contain t.

Values of existential type are created using the construct

pack $\langle \alpha = \hat{\tau}, e : \tau\rangle$

where α may occur free in τ. The type of this expression is $\exists\alpha.\tau$, and at his point we no longer know that the expression we packed originally had type $\tau[\hat{\tau}/\alpha]$.

A different formulation of existential quantification called the *dot notation*, closer to actual programming languages, is described in [CL90].

2.3 Type Reconstruction

In this section, we describe the Damas-Milner approach to type reconstruction in ML [Mil78] [DM82] [Dam85] and its application to type reconstruction in Haskell [NS91].

2.3.1 Type Reconstruction for ML

Before we present the type inference system and the type reconstruction algorithm for the ML core, we need to define the following terms:

- A *substitution* is a finite mapping from type variables to types. It is of-

ten written in the form $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ and applied as a postfix operator; it can also be given a name, for example, S, and applied as prefix operator. If $\sigma = \forall\beta_1 \ldots \beta_m.\tau$ is a type scheme, then $S\sigma$ is the type scheme obtained from σ by replacing each free occurrence of α_i in σ by τ_i, renaming the bound variables of σ if necessary. Let *Id* denote the identity substitution $[\ ]$.

- Type τ is a *principal type* of expression e under assumption set A if $A \vdash e : \tau$ and whenever $A \vdash e : \tau'$ then there is a substitution S such that $S\tau = \tau'$,

We now give a type inference system that describes the type system of the implicitly typed first-order polymorphic λ-calculus underlying the ML core. This type system is deterministic in that there is exactly one rule for each kind of expression. It was shown in [CDDK86] to be equivalent to the original nondeterministic system from [DM82].

(TAUT) $$\frac{A(x) \geq \tau}{A \vdash x : \tau}$$

(APP) $$\frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e\ e') : \tau}$$

(ABS) $$\frac{A[\tau'/x] \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau}$$

(LET) $$\frac{A \vdash e : \tau \qquad A[gen(A, \tau)/x] \vdash e' : \tau'}{A \vdash \texttt{let}\ \ x = e\ \ \texttt{in}\ \ e' : \tau'}$$

The following auxiliary definitions are needed:

- In the *generic instantiation* of a type scheme to a type, each *generic*

(universally quantified) type variable is replaced by a type.

$\forall\alpha_1 \ldots \alpha_n.\tau \geq \tau'$ iff there are types such that

$$\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

- The *generalization* of a type τ under an assumption set is the type scheme obtained from τ by universally quantifying over those type variables that are free in τ but not in the assumption set A.

$$gen(A, \tau) = \forall (FV(\tau) \setminus FV(A)).\tau$$

For instance, $\forall\alpha.\alpha \to \alpha \geq int \to int$ but not $\forall\alpha.\alpha \to \alpha \geq int \to real$, and $gen([\beta/x], \alpha \to \beta) = \forall\alpha.\alpha \to \beta$.

Now we have a type inference system that defines what is a valid typing judgment in the ML core. However, we are actually interested in an algorithm that tells us whether a given (implicitly typed) core-ML expression is type correct, and if so, what its principal type is. Given an assumption set A and an expression e, it returns $W(A, e) = (S, \tau)$, where is S a substitution and τ a type. We want this algorithm to be *syntactically sound* and *complete*:

- Syntactic soundness: If $W(A, e) = (S, \tau)$, then $SA \vdash e : \tau$ is a valid typing judgment, that is, we can prove it in the inference system.
- Syntactic completeness and *principal typing*: Whenever $A \vdash e : \tau$, then $W(A, e) = (S, \tau')$ terminates and $\tau' \geq \tau$ is a principal type for e under A.

The following algorithm from [DM82] has the desired properties, as proved in [Dam85]. $inst_\forall(\forall\alpha_1 \ldots \alpha_n.\tau)$ replaces each occurrence of α_i in τ with a fresh type variable, and *gen* is defined as in the inference rules.

$W(A, x) =$

$$(Id, inst_\forall(A(x)))$$

$W(A, e\ e') =$

let $(S, \tau) = W(A, e)$
$(S', \tau') = W(SA, e')$
β be a fresh type variable
$U = mgu(S'\tau, \tau' \to \beta)$
in $(US'S, U\beta)$

$W(A, \lambda x.e) =$

let β be a fresh type variable
$(S, \tau) = W(A\,[\beta/x]\,, e)$
in $(S,\ (S\beta) \to \tau)$

$W(A,$ **let** $x = e$ **in** $e') =$

let $(S, \tau) = W(A, e)$
$(S', \tau') = W(\,(SA)\ [gen(SA, \tau)/x]\,, e')$
in $(S'S, \tau')$

The function $mgu(\tau_1, \tau_2)$ computes a *most general unifier* U for τ_1 and τ_2, which is a the most general substitution such that $U\tau_1 = U\tau_2$, if one exists, otherwise *mgu* fails. The idea is that we substitute actual types for the fresh type variables generated by applications of $inst_\forall(\forall\alpha_1 \ldots \alpha_n.\tau)$, and that when the algorithm W terminates, we have constructed a proof in our inference system whose structure corresponds to the structure of the expression itself.

2.3.2 Order-Sorted Unification for Haskell

In Haskell, we have a three-level world consisting of values, types, and type classes. While types in core-ML are not classified[1], Haskell type classes classify types into partially ordered *sorts*. This is in contrast to those type

[1] Actually, in Standard ML types are classified in types with and without an equality operation defined for them.

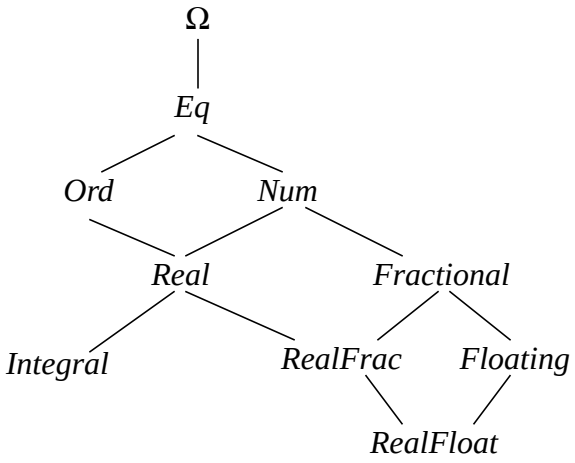
systems where types themselves are partially ordered, for example the one of OBJ [FGJM85]. *Order-sorted unification* [MGS89] can be used to obtain a type reconstruction algorithm in an order-sorted type system such as Haskell's; this is described in [NS91].

An *order-sorted signature* C consists of three parts, a set of *sort symbols*, a *sort hierarchy*, and a set of *arity declarations*. The sort hierarchy is simply an partial order on the sorts. Arity declarations are of the form $\chi : (\gamma_1, \ldots, \gamma_n)\,\gamma$, where χ is a type constructor and $\gamma, \gamma_1, \ldots, \gamma_n$ are sorts. The set of *order-sorted* type expressions is the least set satisfying the following two conditions:

- If τ has sort γ' and $\gamma' \leq \gamma$ in the sort hierarchy, then τ also has sort γ.
- If the type expressions $\tau_1, \ldots, \tau_n$ have sorts $\gamma_1, \ldots, \gamma_n$, respectively, and $\chi : (\gamma_1, \ldots, \gamma_n)\,\gamma$ is in the set of arity declarations, then the application $\chi(\tau_1, \ldots, \tau_n)$ of the type constructor has sort γ.

Substitutions are defined as in Section 2.3.1, but in addition, they must be *sort-correct*: If type variable α has sort γ, expressed by writing α_γ, then $S(\alpha)$ must also have sort γ.

The following example of a sort hierarchy shows the Haskell numeric class hierarchy:



As an example for a set of arity declarations, consider the following declarations for the type constructors *pair* and *list*:

$$pair : (\Omega, \Omega)\,\Omega$$

$$pair : (Eq, Eq)\,Eq$$

$$pair : (Ord, Ord)\,Ord$$

$$list : (\Omega)\,\Omega$$

$$list : (Eq)\,Eq$$

$$list : (Ord)\,Ord$$

$$list : (Num)\,Num$$

$$list : (\Omega)\,\Omega$$

Given a set Γ of equations over type expressions constructed from C, a *unifier* of Γ is a substitution θ such that $\theta(\tau_1) = \theta(\tau_2)$ for all equations $\tau_1 = \tau_2$ in Γ. An order-sorted signature is called *unitary* if for all such equation sets Γ there is a complete set of unifiers containing at most one element. Since unitary signatures guarantee principal types, we give the following conditions from [SS85] to guarantee that a *finite* signature is unitary:

- *Regularity*: Each type has a least sort.
- *Downward completeness*: Any two sorts have either no lower bound or an infimum.
- *Injectivity*: $\chi : (\gamma_1, \ldots, \gamma_n)\,\gamma$ and $\chi : (\gamma'_1, \ldots, \gamma'_n)\,\gamma$ imply $\gamma_i = \gamma'_i$ for all $i = 1, \ldots, n$.
- *Subsort reflection*: $\chi : (\gamma'_1, \ldots, \gamma'_n)\,\gamma'$ and $\gamma' \leq \gamma$ imply $\chi : (\gamma_1, \ldots, \gamma_n)\,\gamma$ for some $\gamma_1 \geq \gamma'_1, \ldots, \gamma_n \geq \gamma'_n$.

Haskell imposes context conditions to guarantee that the signatures that arise in Haskell programs are unitary; this is further discussed in Chapter 5.

2.4 Semantics

It is often convenient to use a *denotational semantics* to reason about the evaluation of λ-expressions. A denotational semantics is given in terms of an evaluation function that maps *syntactic terms* to *semantic values* in a *semantic domain*. The evaluation function $E[\![e]\!]\ \rho$ interprets an expression e in the *environment* ρ and returns a value in the *domain* V. An evaluation environment is a finite mapping from identifiers to semantic values. A semantic domain is an algebraic structure that allows us to represent (semantic) values corresponding to the (syntactic) entities in our calculus.

2.4.1 Recursive Domains

The notion of domains goes back to [SS71]. To illustrate this notion, we recall that in the untyped λ-calculus we start out with the built-in constants (integers, Booleans, etc.) and are able to define functions over the constants. We can further define functions that range over these functions and so on. This structure is reflected in the definition of the domain V that satisfies the following isomorphism:

$$V \cong B + N + (V \to V) + \{\mathsf{wrong}\}_{\perp}$$

Here $+$ stands for the coalesced sum, so that all types over V share the same least element $\perp$. In other words, V is isomorphic to the sum of the Boolean values B, the natural numbers N, the continuous functions from V to V, and a value wrong representing runtime type errors.

Solutions of equations of this kind can be found in the class of continuous functions over complete partial orders. A *complete partial order* (cpo) consists of a set D and a partial order $\leq$ on D such that

- there is a least element $\perp$ in D, and
- each increasing sequence $x_0 \leq \ldots \leq x_n \leq \ldots$ has a least upper bound (lub) $\bigsqcup_{n \geq 0} x_n$.

A function f is continuous iff it preserves lubs of increasing sequences, that is,

$$f(\bigsqcup_{n \geq 0} x_n) = \bigsqcup_{n \geq 0} f(x_n)$$

An element of a cpo is called ω-*finite* iff whenever it is less than the lub of an increasing sequence it is less than some element in the sequence. Finally, a *domain* is defined as a cpo satisfying the following conditions:

- *Consistently complete*: Any consistent subset of V has a least upper bound, where $X \subseteq V$ is *consistent* if it has an upper bound in V.
- ω-*algebra*ic: V has countably many ω-finite elements, and given any $x \in V$, the set of ω-finite elements less than x is directed and has x as its least upper bound.

The ω-finite elements in any subset X of a cpo are denoted by $X°$.

Our domain V can be constructed via a limiting process described in [Smy77].

2.4.2 Weak Ideals

Ideals [MPS86] capture the notion of sets of structurally similar values and have proven to be a useful model for types. As a detailed treatment of the ideal model goes beyond the scope of this dissertation, we confine ourselves to a summary of the properties relevant to our work.

A subset I of a domain D is a *(weak) ideal* iff it satisfies the following conditions:

- $I \neq \emptyset$,
- for all $y \in I$ and $x \in D$, $x \leq y$ implies $x \in I$, and
- for all increasing sequences $\langle x_n \rangle$, $x_i \in I$ for all $i \geq 0$ implies $\bigsqcup x_n \in I$.

Ideals have the pleasant property that they form a complete lattice with their greatest lower bounds given by set-theoretic intersection and their least up-

per bounds given by the following formula, stating that finite lubs are given by set-theoretic union:

$$(\bigsqcup_n I_n)^\circ = \bigcup_n (I_n{}^\circ)$$

The ideals over domain V form a *complete metric space*, on which a Banach fixed-point theorem holds. This allows us to model recursively defined types as fixed points of *contractive maps* on ideals. The maps on ideals corresponding to the type constructors in our type model (see Section 3.6.3) are contractive and consequently, our recursively defined algebraic data types[1] have a well-defined semantics.

[1] Algebraic data types in our language are a restricted version of ML datatypes.

3 An Extension of ML with First-Class Abstract Types

This chapter presents a semantic extension of ML, where the component types of a datatype may be existentially quantified. We show how datatypes over existential types add significant flexibility to the language without even changing ML syntax. We then describe a deterministic Damas-Milner type inference system [DM82] [CDDK86] for this language, which leads to a syntactically sound and complete type reconstruction algorithm. Furthermore, the type system is shown to be semantically sound with respect to a standard denotational semantics.

3.1 Introduction

In ML, datatype declarations are of the form

datatype [*arg*] *T* **=** K_1 **of** τ_1 **|** ... **|** K_n **of** τ_n

where the *K*'s are value constructors and the optional prefix argument *arg* is used for formal type parameters, which may appear free in the component types τ_i. The types of the value constructor functions are universally quantified over these type parameters, and no other type variables may appear free in the τ_i's.

An example for an ML datatype declaration is

```
datatype 'a Mytype = mycons of 'a * ('a -> int)
```

Without altering the syntax of the datatype declaration, we now give a meaning to type variables that appear free in the component types, but do not occur in the type parameter list. We interpret such type variables as existentially quantified.

For example,

```
datatype Key = key of 'a * ('a -> int)
```

describes a datatype with one value constructor whose arguments are pairs of a value of type **'a** and a function from type **'a** to **int**. The question is what we can say about **'a**. The answer is, nothing, except that the value is of the same type **'a** as the function domain. To illustrate this further, the type of the expression

```
key(3,fn x => 5)
```

is **Key**, as is the type of the expression

```
key([1,2,3],length)
```

where **length** is the built-in function on lists. Note that no argument types appear in the result type of the expression. On the other hand,

```
key(3,length)
```

is not type-correct, since the type of **3** is different from the domain type of **length**.

We recognize that **Key** is an abstract type comprised by a value of some type and an operation on that type yielding an **int**. It is important to note that values of type **Key** are first-class; they may be created dynamically and passed around freely as function parameters. The two different values of type **Key** in the previous examples may be viewed as two different implementations of the same abstract type.

Besides constructing values of datatypes with existential component types, we can decompose them using the **let** construct. We impose the restriction that no type variable that is existentially quantified in a **let** expression appears in the result type of this expression or in the type of a global identifier. Analogous restrictions hold for the corresponding **open** and **abstype** constructs described in [CW85] [MP88].

For example, assuming **x** is of type **Key**, then

```
let val key(v,f) = x in
    f v
end
```

has a well-defined meaning, namely the **int** result of **f** applied to **v**. We know that this application is type-safe because the pattern matching succeeds, since **x** was constructed using constructor **key**, and at that time it was enforced that **f** can safely be applied to **v**. On the other hand,

```
let val key(v,f) = x in
    v
end
```

is not type-correct, since we do not know the type of **v** statically and, consequently, cannot assign a type to the whole expression.

Our extension to ML allows us to deal with existential types as described in [CW85] [MP88], with the further improvement that decomposed values of existential type are **let**-bound and may be instantiated polymorphically. This is illustrated by the following example,

```
datatype 'a t = k of ('a -> 'b) * ('b -> int)
let val k(f1,f2) = k(fn x => x,fn x => 3) in
    (f2(f1 7),f2(f1 true))
end
```

which results in **(3,3)**. In most previous work, the value on the right-hand side of the binding would have to be bound and decomposed twice.

3.2 Some Motivating Examples

3.2.1 Minimum over a Heterogeneous List

Extending on the previous example, we first show how we construct heterogeneous lists over different implementations of the same abstract type and define functions that operate uniformly on such heterogeneous lists. A heterogeneous list of values of type **`Key`** could be defined as follows:

```
val hetlist = [key(3,fn x => x),
               key([1,2,3,4],length),
               key(7,fn x => 0),
               key(true,fn x => if x then 1 else 0),
               key(12,fn x => 3)]
```

The type of **`hetlist`** is **`Key list`**; it is a homogeneous list of elements each of which could be a different implementation of type **`Key`**. We define the function **`min`**, which finds the minimum of a list of **`Key`**'s with respect to the integer value obtained by applying the second component (the function) to the first component (the value).

```
fun min [x] = x
  | min ((key(v1,f1))::xs) =
        let val key(v2,f2) = min xs in
            if f1 v1 <= f2 v2 then
                key(v1,f1)
            else
                key(v2,f2)
        end
```

Then **`min hetlist`** returns **`key(7,fn x => 0)`**, the third element of the list.

3.2.2 Stacks Parametrized by Element Type

The preceding example involves a datatype with existential types but without polymorphic type parameters. As a practical example involving both ex-

istential and universal quantification, we show an abstract stack parametrized by element type.

```
datatype 'a Stack =
    stack of {value   : 'b,
              empty   : 'b,
              push    : 'a -> 'b -> 'b
              pop     : 'b -> 'b
              top     : 'b -> 'a,
              isempty : 'b -> bool}
```

An on-the-fly implementation of an **`int Stack`** in terms of the built-in type **`list`** can be given as

```
stack{value = [1,2,3], empty = [],
      push = fn x => fn xs => x :: xs,
      pop = tl, top = hd, isempty = null}
```

An alternative implementation of **`Stack`** could be given, among others, based on arrays. We provide a constructor for each implementation:

```
fun makeListStack xs = stack{value = xs, empty = [],
        push = fn x => fn xs => x :: xs, pop = tl,
        top = hd, isempty = null}

fun makeArrayStack xs = stack{...}
```

To achieve dynamic dispatching, one can provide stack operations that work uniformly across implementations. These "outer" wrappers work by opening the stack, applying the intended "inner" operations, and encapsulating the stack again, for example:

```
fun push a (stack{value = v, push = pu, empty = e,
                  pop = po, top = t, isempty = i}) =
           stack{value = pu a v, push = pu,
                 empty = e, pop = po,
                 top = t, isempty = i}
```

Different implementations could then be combined in a list of stacks, and we can uniformly apply the wrapper **push** to each element of the list:

```
map (push 8) [makeListStack [2,4,6],
              makeArrayStack [3,5,7]]
```

3.2.3 Squaring a Heterogeneous List of Numbers

The next example shows that datatypes with existential component types provide high extensibility. The following type describes an abstract data type consisting of a number and a multiplication function that can be used on the number:

```
datatype Mult = mult of 'a * ('a * 'a -> 'a)
```

We define a function that squares an abstract number:

```
fun square(mult(x,f)) = mult(f(x,x),f)
```

Now we can square each element of a heterogeneous list of numbers in the following fashion:

```
map square
    [mult(3,   op * : int * int -> int),
     mult(7.5, op + : real * real -> real)]
```

New functions using the abstract type **Mult** can be added easily without modifying the previous definitions. This provides high extensibility in comparison with the closure approach; see also the example in Section 2.1.2. For example, we can add a function **cube** and raise each element of a list to its cube:

```
fun cube(mult(x,f)) = mult(f(x,f(x,x)),f)

map cube [mult(8, op * : int * int -> int),
          mult([1,2,3], op @)]
```

3.2.4 Abstract Binary Trees with Equality

This example shows that abstract data types with binary operations can be modeled conveniently and naturally using existential types. This is in contrast to the tree example in Section 2.1.2. We start with the following datatype declaration:

```
datatype Tree = tree of {value : 'b,
                         empty : 'b
                         eq    : 'b * 'b -> bool,
                         left  : 'b -> 'b,
                         right : 'b -> 'b,
                         join  : 'b * 'b -> 'b}
```

Assuming that **t** has type **Tree**, we can now check whether the left and right subtrees of **t** are equal:

```
let val tree{value=v,left=l,right=r,eq=eq,...} = t
in
    eq(l v,r v)
end
```

As opposed to the closure approach, where we would have to convert both subtrees to a common representation, we can take advantage of the fact that two subtrees of a tree already have the same representation.

3.3 Syntax

3.3.1 Language Syntax

Identifiers x

Constructors K

Expressions $e ::= x \mid (e_1, e_2) \mid e\ e' \mid \lambda x.e \mid$
let $x = e$ **in** $e' \mid$
data $\forall\alpha_1 \ldots \alpha_n.\chi$ **in** $e \mid K \mid$ **is** $K \mid$

let $K\ x = e$ **in** e'

In addition to the usual constructs (identifiers, applications, λ-abstractions, and **let** expressions), we introduce desugared versions of the ML constructs that deal with datatypes. A **data** declaration defines a new datatype; values of this type are created by applying a constructor K, their tags can be inspected using an **is** expression, and they can be decomposed by a pattern-matching **let** expression. Further, we require each identifier bound by a λ or **let** expression to be unique[1].

The following example shows a desugared definition of ML's list type and the associated length function; μ introduces a recursive type as described below.

```
data ∀α. (μβ.Nil unit + Cons α×β) in
    let length = fix λlength.λxs.
                    if (is Nil xs)
                       0
                       (let Cons ab = xs in
                            + (length(snd ab)) 1)
    in
        length(Cons(3,Cons(7,Nil())))
```

3.3.2 Type Syntax

Type variables	α	
Skolem functions	κ	
Types	τ ::=	$unit \mid bool \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau \rightarrow \tau' \mid$ $\kappa(\tau_1, \ldots, \tau_n) \mid \chi$
Recursive types	χ ::=	$\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$ where $K_i \neq K_j$ for

[1]Of course, one would use a static, block-structured scoping discipline in practice.

$$i \neq j$$

Existential types $\eta ::= \exists\alpha.\eta \mid \tau$

Type schemes $\sigma ::= \forall\alpha.\sigma \mid \tau$

Assumptions $a ::= \sigma/x \mid \forall\alpha_1 \ldots \alpha_n.\chi/K$

Our type syntax includes recursive types χ and Skolem type constructors κ; the latter are used to type identifiers bound by a pattern-matching **`let`** whose type is existentially quantified. Explicit existential types arise only as domain types of value constructors. Assumption sets serve two purposes: they map identifiers to type schemes and constructors to the recursive type schemes they belong to. Thus, when we write $A(K)$, we mean the σ such that $\sigma = \forall\alpha_1 \ldots \alpha_n.\ldots + K\eta + \ldots$. Further, let $\Sigma[K\eta]$ stand for sum type contexts such as $K_1\eta_1 + \ldots + K_m\eta_m$, where $K_i = K$ and $\eta_i = \eta$ for some i.

3.4 Type Inference

3.4.1 Instantiation and Generalization of Type Schemes

$\forall\alpha_1 \ldots \alpha_n.\tau \geq \tau'$ iff there are types $\tau_1, \ldots \tau_n$ such that

$$\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

$\exists\alpha_1 \ldots \alpha_n.\tau \leq \tau'$ iff there are types $\tau_1, \ldots \tau_n$ such that

$$\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

gen $(A, \tau) = \forall(FV(\tau) \setminus FV(A)).\tau$

$$\textit{skolem}\,(A, \exists\gamma_1 \ldots \gamma_n.\tau) = \tau[\kappa_i(\alpha_1, \ldots \alpha_k)/\gamma_i] \text{ where } \kappa_1 \ldots \kappa_n \text{ are new}$$
$$\text{Skolem type constructors such that}$$
$$\{\kappa_1, \ldots, \kappa_n\} \cap FS(A) = \varnothing, \text{ and}$$
$$\{\alpha_1, \ldots, \alpha_k\} = FV(\exists\gamma_1 \ldots \gamma_n.\tau) \setminus FV(A)$$

The first three auxiliary functions are standard. The function *skolem* replaces each existentially quantified variable in a type by a unique type constructor whose actual arguments are those free variables of the type that are not free in the assumption set; this reflects the "maximal" knowledge we have about the type represented by an existentially quantified type variable. In addition to *FV*, the set of free type variables in a type scheme or assumption set, we use *FS*, the set of Skolem type constructors that occur in a type scheme or assumption set.

3.4.2 Inference Rules for Expressions

The first five typing rules are essentially the same as in [CDDK86].

(VAR) $$\frac{A(x) \geq \tau}{A \vdash x : \tau}$$

(PAIR) $$\frac{A \vdash e_1 : \tau_1 \qquad A \vdash e_2 : \tau_2}{A \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

(APPL) $$\frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash e\ e' : \tau}$$

(ABS) $$\frac{A[\tau'/x] \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau}$$

(LET) $$\frac{A \vdash e : \tau \qquad A[gen(A, \tau)/x] \vdash e' : \tau'}{A \vdash \textbf{let}\ \ x = e\ \ \textbf{in}\ \ e' : \tau'}$$

The new rules DATA, CONS, TEST, and PAT are used to type datatype declarations, value constructors, **is** expressions, and pattern-matching **let** expressions, respectively.

$$\text{(DATA)}\quad \frac{\begin{array}{c}\sigma = \forall\alpha_1 \ldots \alpha_n.\,\mu\beta.\,K_1\eta_1 + \ldots + K_m\eta_m \\ FV(\sigma) = \emptyset \qquad A\,[\sigma/K_1, \ldots, \sigma/K_m] \vdash e : \tau\end{array}}{A \vdash \textbf{data}\ \ \sigma\ \ \textbf{in}\ \ e : \tau}$$

The DATA rule elaborates a declaration of a recursive datatype. It checks that the type scheme is closed and types the expression under the assumption set extended with assumptions about the constructors.

$$\text{(CONS)}\quad \frac{A(K) \geq \mu\beta.\,\Sigma\,[K\eta] \qquad \eta\,[\mu\beta.\,\Sigma\,[K\eta]/\beta] \leq \tau}{A \vdash K : \tau \rightarrow \mu\beta.\,\Sigma\,[K\eta]}$$

The CONS rule observes the fact that existential quantification in argument position means universal quantification over the whole function type; this is expressed by the second premise.

$$\text{(TEST)}\quad \frac{A(K) \geq \mu\beta.\,\Sigma\,[K\eta]}{A \vdash \textbf{is}\ \ K : (\mu\beta.\,\Sigma\,[K\eta]) \rightarrow \mathit{bool}}$$

The TEST rule ensures that **is** K is applied only to arguments whose type is the same as the result type of constructor K.

$$\text{(PAT)}\quad \frac{\begin{array}{c}A \vdash e : \mu\beta.\,\Sigma\,[K\eta] \qquad FS(\tau') \subseteq FS(A) \\ A\,[\mathit{gen}\,(A, \mathit{skolem}\,(A, \eta\,[\mu\beta.\,\Sigma\,[K\eta]/\beta]))/x] \vdash e' : \tau'\end{array}}{A \vdash \textbf{let}\ \ K\ x = e\ \ \textbf{in}\ \ e' : \tau'}$$

The last rule, PAT, governs the typing of pattern-matching **let** expressions. It requires that the expression e be of the same type as the result type of the

constructor K. The body e' is typed under the assumption set extended with an assumption about the bound identifier x. By definition of the function *skolem*, the new Skolem type constructors do not appear in A; this ensures that they do not appear in the type of any identifier free in e' other than x. It is also guaranteed that the Skolem constructors do not appear in the result type τ'.

3.4.3 Relation to the ML Type Inference System

We compare our system with Mini-ML', an extension of Mini-ML with recursive datatypes, but without existential quantification. Mini-ML' has the same syntax as our language. The type inference system of Mini-ML' consists of the rules VAR, PAIR, APPL, ABS, and LET, and the following modified versions of the remaining rules[1]:

$$\text{(DATA')}\quad \frac{\sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\tau_1 + \ldots + K_m\tau_m \qquad FV(\sigma) = \emptyset \qquad A[\sigma/K_1, \ldots, \sigma/K_m] \vdash e : \tau}{A \vdash \mathbf{data}\ \sigma\ \mathbf{in}\ e : \tau}$$

$$\text{(CONS')}\quad \frac{A(K) \geq \mu\beta.\Sigma[K\tau]}{A \vdash K : \tau \rightarrow \mu\beta.\Sigma[K\tau]}$$

$$\text{(TEST')}\quad \frac{A(K) \geq \mu\beta.\Sigma[K\tau]}{A \vdash \mathbf{is}\ K : (\mu\beta.\Sigma[K\tau]) \rightarrow \mathit{bool}}$$

$$\text{(PAT')}\quad \frac{A \vdash e : \mu\beta.\Sigma[K\tau] \qquad A[\mathit{gen}(A, \tau[\mu\beta.\Sigma[K\tau]/\beta])/x] \vdash e' : \tau'}{A \vdash \mathbf{let}\ K\ x = e\ \mathbf{in}\ e' : \tau'}$$

[1] Theoretically, it is sufficient to modify only the DATA rule to preclude that existential quantifiers arise in the inference system; however, it is more illustrative to present modified versions of the CONS, TEST, and PAT rules as well.

Theorem 3.1 [Conservative extension] For any Mini-ML' expression e, $A \vdash e : \tau$ iff $A \vdash_{\text{Mini-ML'}} e : \tau$.

Proof: By structural induction on e.

Corollary 3.2 [Conservative extension] Our type system is a conservative extension of the Mini-ML type system described in [CDDK86], in the following sense: For any Mini-ML expression e, $A \vdash e : \tau$ iff $A \vdash_{\text{Mini-ML}} e : \tau$.

Proof: Follows immediately from Theorem 3.1.

3.5 Type Reconstruction

The type reconstruction algorithm is a straightforward translation from the deterministic typing rules, using a standard unification algorithm [Rob65] [MM82]. We conjecture that its complexity is the same as that of algorithm W.

3.5.1 Auxiliary Functions

In our algorithm, we need to instantiate universally quantified types and generalize existentially quantified types. Both are handled in the same way.

$$inst_{\forall}(\forall\alpha_1 \ldots \alpha_n.\tau) = \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n] \text{ where } \beta_1, \ldots, \beta_n \text{ are fresh type variables}$$

$$inst_{\exists}(\exists\alpha_1 \ldots \alpha_n.\tau) = \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n] \text{ where } \beta_1, \ldots, \beta_n \text{ are fresh type variables}$$

The functions *skolem* and *gen* are the same as in the inference rules, with the additional detail that *skolem* always creates fresh Skolem type constructors.

3.5.2 Algorithm

Our type reconstruction function takes an assumption set and an expression, and it returns a substitution and a type expression. There is one case for each typing rule.

$$TC\,(A, x) \;=$$
$$\quad (Id,\ inst_{\forall}\ (A\,(x))\,)$$

$$TC\,(A,\ (e_1, e_2)\,) \;=$$
$$\quad \textbf{let}\ \ (S_1, \tau_1) = TC\,(A, e_1)$$
$$\qquad\ \ (S_2, \tau_2) = TC\,(S_1 A, e_2)$$
$$\quad \textbf{in}\ \ (S_2 S_1, S_2\tau_1 \times \tau_2)$$

$$TC\,(A, e\ e') \;=$$
$$\quad \textbf{let}\ \ (S, \tau) = TC\,(A, e)$$
$$\qquad\ \ (S', \tau') = TC\,(SA, e')$$
$$\qquad\ \ \beta \text{ be a fresh type variable}$$
$$\qquad\ \ U = mgu\ (S'\tau, \tau' \to \beta)$$
$$\quad \textbf{in}\ \ (US'S, U\beta)$$

$$TC\,(A, \lambda x.e) \;=$$
$$\quad \textbf{let}\ \ \beta \text{ be a fresh type variable}$$
$$\qquad\ \ (S, \tau) = TC\,(A\,[\beta/x]\,, e)$$
$$\quad \textbf{in}\ \ (S, S\beta \to \tau)$$

$$TC\,(A, \textbf{let}\ \ x = e\ \ \textbf{in}\ \ e') \;=$$
$$\quad \textbf{let}\ \ (S, \tau) = TC\,(A, e)$$
$$\qquad\ \ (S', \tau') = TC\,(SA\,[gen\ (SA, \tau)\,/x]\,, e')$$
$$\quad \textbf{in}\ \ (S'S, \tau')$$

$$TC(A, \textbf{data}\ \sigma\ \textbf{in}\ e) =$$
$$\textbf{let}\ \ \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m = \sigma\ \textbf{in}$$
$$\textbf{if}\ \ FV(\sigma) = \varnothing\ \textbf{then}$$
$$TC(A[\sigma/K_1, \ldots, \sigma/K_m], e)$$

$$TC(A, K) =$$
$$\textbf{let}\ \ \tau = inst_{\forall}(A(K))$$
$$\mu\beta.\ldots + K\eta + \ldots = \tau$$
$$\textbf{in}\ \ (Id, (inst_{\exists}(\eta[\tau/\beta])) \rightarrow \tau)$$

$$TC(A, \textbf{is}\ K) =$$
$$\textbf{let}\ \ \tau = inst_{\forall}(A(K))$$
$$\textbf{in}\ \ (Id, \tau \rightarrow bool)$$

$$TC(A, \textbf{let}\ K\ x = e\ \textbf{in}\ e') =$$
$$\textbf{let}\ \ (S, \tau) = TC(A, e)$$
$$U = mgu(\tau, inst_{\forall}(A(K)))$$
$$\mu\beta.\ldots + K\eta + \ldots = U\tau$$
$$\tau_\kappa = skolem(USA, \eta[U\tau/\beta])$$
$$(S', \tau') = TC(USA[gen(USA, \tau_\kappa)/x], e')$$
$$\textbf{in}$$
$$\textbf{if}\ \ FS(\tau') \subseteq FS(S'USA) \wedge$$
$$(FS(\tau_\kappa) \setminus FS(\eta[U\tau/\beta])) \cap FS(S'USA) = \varnothing$$
$$\textbf{then}\ \ (S'US, \tau')$$

3.5.3 Syntactic Soundness and Completeness of Type Reconstruction

Since any two type schemes that differ only by renaming of bound variables instantiate to the same set of types, it is convenient to treat them as equivalent. This expressed by the following lemma:

Lemma 3.3 [Equivalence under renaming] Let $\sigma_1 = \forall\alpha_1 \ldots \alpha_n.\tau_0$ and $\sigma_2 = \forall\beta_1 \ldots \beta_n.(\tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n])$. Then $\sigma_1 \geq \tau$ iff $\sigma_2 \geq \tau$ for any type τ.

Proof: Follows immediately from the definition of instantiation.

Lemma 3.4 [Stability of $\geq$] If $\sigma \geq \tau$, then $S\sigma \geq S\tau$.

Proof: By definition, $\sigma = \forall\alpha_1 \ldots \alpha_n.\tau_0$ and $\tau = \tau_0[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ for some types $\tau_1, \ldots, \tau_n$. Since the α_i's may be renamed consistently, we assume that $\{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{Dom}\, S \cup FV(\mathrm{Rng}\, S)) = \emptyset$. Consequently, $S\sigma = \forall\alpha_1 \ldots \alpha_n.S\tau_0 \geq (S\tau_0)[S\tau_1/\alpha_1, \ldots, S\tau_n/\alpha_n] = S\tau$, by definition of $\geq$.

Lemma 3.5 [Stability of *gen*] $Sgen(A, \tau) = gen(SA, S\tau)$.

Proof: Follows from the definition of *gen*, assuming that $(FV(\tau) \setminus FV(A)) \cap (\mathrm{Dom}\, S \cup FV(\mathrm{Rng}\, S)) = \emptyset$.

Lemma 3.6 [Stability of *skolem*] $Sskolem(A, \eta) = skolem(SA, S\eta)$.

Proof: Similar, using the definition of *skolem* and an appropriate renaming.

Lemma 3.7 [Stability of $\vdash$] If $A \vdash e : \tau$ and S is a substitution, then $SA \vdash e : S\tau$ also holds. Moreover, if there is a proof tree for $A \vdash e : \tau$ of height n, then there is also a proof tree for $SA \vdash e : S\tau$ of height less or equal to n.

Proof: By induction on the height n of the proof tree for $A \vdash e : \tau$. We have one case for each type inference rule, but include only the nonstandard cases.

$A \vdash$ **data** $\forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$ **in** $e : \tau$

The premise is $A[\sigma/K_1, \ldots, \sigma/K_n] \vdash e : \tau$, where

$\sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$. Since $FV(\sigma) = \emptyset$, we have

$S\sigma = \sigma$. By the inductive assumption,

$S(A[\sigma/K_1, \ldots, \sigma/K_n]) \vdash e : S\tau$, where

$S(A[\sigma/K_1, \ldots, \sigma/K_n]) = (SA)[\sigma/K_1, \ldots, \sigma/K_n]$. Finally, we apply the DATA rule and obtain $SA \vdash$ **data** σ **in** $e : S\tau$.

$A \vdash K : \tau \to \mu\beta.\Sigma[K\eta]$ and

$A \vdash$ **is** $K : (\mu\beta.\Sigma[K\eta]) \to bool$

The claim follows from stability of $\geq$ under substitution.

$A \vdash$ **let** $K\ x = e$ **in** $e' : \tau'$

Assuming that $\beta \notin \mathrm{Dom}\,S \cup FV(\mathrm{Rng}\,S)$, we have

$S(\mu\beta.\Sigma[K\eta]) = \mu\beta.\Sigma[K(S\eta)]$ and

$S(\eta[\mu\beta.\Sigma[K\eta]/\beta]) = (S\eta)[\mu\beta.\Sigma[K(S\eta)]/\beta]$.

We apply the inductive assumption to the first premise, obtaining $SA \vdash e : \mu\beta.\Sigma[K(S\eta)]$, and to the last premise, obtaining $S(A[gen(A, skolem(A, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x]) \vdash e' : S\tau'$. Further,

$S(A[gen(A, skolem(A, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x]) =$

$(SA)[Sgen(A, skolem(A, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x] =$

$(SA)[gen(SA, skolem(SA, (S\eta)[\mu\beta.\Sigma[K(S\eta)]/\beta]))/x]$, using Lemma 3.5 and Lemma 3.6.

Finally, we observe that $FS(\tau') \subseteq FS(A)$ implies $FS(S\tau') \subseteq FS(SA)$, and the claim follows by applying the PAT rule.

■

Theorem 3.8 [Syntactic soundness] If $TC(A, e) = (S, \tau)$, then $SA \vdash e : \tau$.

Proof: A straightforward application of Lemma 3.7. We show the only tricky case:

$TC(A,$ **let** $K\ x = e$ **in** $e') = (S'U, \tau')$

By applying the inductive assumption to the first recursive call to TC, we have $SA \vdash e : \tau$. Since $mgu(\tau, inst_{\forall}(A(K)))$ succeeds with U, we know that $A(K) \geq U\tau$, whence $U\tau$ is of the form $\mu\beta.\Sigma[K\eta]$. By Lemma 3.7, $S'USA \vdash e : S'U\tau$. We now apply the inductive assumption to the second recursive call and get
$S'(USA[gen(USA, skolem(USA, \eta[U\tau/\beta]))/x]) \vdash e' : \tau'$.
We use Lemma 3.5 and Lemma 3.6 to obtain
$(S'USA)[gen(S'USA, skolem(S'USA, S'(\eta[U\tau/\beta])))/x] \vdash e' : \tau'$,
where $S'(\eta[(U\tau)/\beta]) = (S'\eta)[S'U\tau/\beta]$. The subsequent **if** statement ensures that none of the fresh Skolem constructors escape the scope of the **let** expression. Hence, the PAT rule applies and our claim is proved.

■

Definition 3.1 [Principal type] τ is a principal type of expression e under assumption set A if $A \vdash e : \tau$ and whenever $A \vdash e : \tau'$ then there is a substitution S such that $S\tau = \tau'$.

Theorem 3.9 [Syntactic completeness] If $\hat{S}A \vdash e : \hat{\tau}$, then $TC(A, e)$ succeeds with $TC(A, e) = (S, \tau)$ and there is a substitution R such that $\hat{S}A = RSA$ and $\hat{\tau} = R\tau$.

Proof: Analogous to the completeness proof given in [Dam85]. We show only the new cases:

$\hat{S}A \vdash$ **data** $\forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$ **in** $e : \hat{\tau}$

Let $\sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots$. Since $FV(\sigma) = \emptyset$, we have $\hat{S}\sigma = \sigma$.

The other premise is $\hat{S}A' \vdash e : \hat{\tau}$, where $A' = A[\sigma/K_1, \ldots, \sigma/K_n]$.

By the inductive assumption, $TC(A', e) = (S, \tau)$, where there is a substitution R such that $\hat{S}A' = RSA'$ and $\hat{\tau} = R\tau$. Hence $TC(A,$ **data** $\ldots)$ succeeds with (S, τ). Since $\hat{S}A' = RSA'$, $\hat{S}A = RSA$ also holds, whence R is our desired substitution.

$\hat{S}A \vdash K : \hat{\tau} \to \mu\beta.\Sigma[K\hat{\eta}]$

It is clear from the premises that $\hat{S}A(K)$ is of the form $\sigma = \forall\alpha_1 \ldots \alpha_n.\rho$, where $\rho = \mu\beta.\Sigma[K\exists\beta_1 \ldots \beta_k.\tau_0]$; further, $FV(\sigma) = \emptyset$, since σ must have been declared in a surrounding **data** expression. Therefore, $\hat{S}A(K) = A(K)$, and the instantiations $inst_\forall$ and $inst_\exists$ succeed, such that $\tau = (\tau_0 \to \rho)[\alpha'_i/\alpha_i, \beta'_j/\beta_j]$, where the α'_i and β'_i are fresh type variables. By definition of $\geq$ and $\leq$, there are types $\tau_1, \ldots, \tau_n$ and $\tau'_1, \ldots, \tau'_k$ such that $\hat{\tau} \to \mu\beta.\Sigma[K\hat{\eta}] = (\tau_0 \to \rho)[\tau_i/\alpha_i, \tau'_j/\beta_j]$.

Finally, by choosing $R = \hat{S} + [\tau_i/\alpha'_i, \tau'_j/\beta'_j]$, we have

$R\tau = \hat{\tau} \to \mu\beta.\Sigma[K\hat{\eta}]$ and $RIdA = \hat{S}A$, since
$FV(\tau) \subseteq \{\alpha'_1, \ldots, \alpha'_n, \beta'_1, \ldots, \beta'_k\}$ and
$FV(A) \cap \{\alpha'_1, \ldots, \alpha'_n, \beta'_1, \ldots, \beta'_k\} = \emptyset$.

$\hat{S}A \vdash$ **is** $K : (\mu\beta.\Sigma[K\hat{\eta}]) \to bool$

This case is analogous to the preceding one.

$\hat{S}A \vdash$ **let** $K\ x = e$ **in** $e' : \hat{\tau}'$

Again, $\hat{S}A(K) = A(K)$ must be of the form $\sigma = \forall\alpha_1 \ldots \alpha_n.\rho$, where $\rho = \mu\beta.\Sigma[K\eta_0]$. Therefore, $inst_\forall(\sigma) = \rho[\alpha'_i/\alpha_i]$, where the α'_i are new variables. We apply the inductive assumption to the first premise, $\hat{S}A \vdash e : \mu\beta.\Sigma[K\hat{\eta}]$, whence $TC(A, e) = (S, \tau)$ and there is an R such that $\hat{S}A = RSA$ and $R\tau = \mu\beta.\Sigma[K\hat{\eta}] = \mu\beta.\Sigma[K(\eta_0[\hat{\tau}_i/\alpha_i])]$.

In the first case, $\tau = \alpha$, thus $R = [\mu\beta.\Sigma[K(\eta_0[\hat{\tau}_i/\alpha_i])]/\alpha]$. Further, *mgu* succeeds with $U = [\rho[\alpha'_i/\alpha_i]/\alpha]$ and $R = [\hat{\tau}_i/\alpha'_i]U$, therefore $RU = R$ and $\eta = \eta_0[\alpha'_i/\alpha_i]$. Consequently,

$\hat{S}A = RSA = RUSA$ and $\hat{\eta}[\mu\beta.\Sigma[K\hat{\eta}]/\beta] = R(\eta[U\tau/\beta])$.

In the second case, $\tau = \mu\beta.\Sigma[K(\eta_0[\tau_i/\alpha_i])]$, where $R\tau_i = \hat{\tau}_i$. Therefore, *mgu* succeeds with $U = [\tau_i/\alpha'_i]$. Since the α'_i occur only in $inst_\forall(\sigma)$, we have $U\tau = \tau$ and $\eta = \eta_0[\tau_i/\alpha_i]$. Further, $\hat{S}A = RSA = RUSA$ and $\hat{\eta}[\mu\beta.\Sigma[K\hat{\eta}]/\beta] = R(\eta[U\tau/\beta])$.

In either case, we can apply Lemma 3.7 to the last premise, obtaining

$((RUSA)[gen(RUSA, skolem(RUSA, R(\eta[U\tau/\beta])))/x]) \vdash e' : \hat{\tau}'$.

By applying Lemma 3.5 and Lemma 3.6, we get

$R((USA)[gen(USA, skolem(USA, \eta[U\tau/\beta]))/x]) \vdash e' : \hat{\tau}'$.

Next, the inductive hypothesis gives us $TC(A', e') = (S', \tau')$, where $A' = USA[gen(USA, skolem(USA, \eta[U\tau/\beta]))/x]$, and an R' such that $RA' = R'S'A'$ and $\hat{\tau}' = R'\tau'$. Then $R'S'USA = RUSA = RSA = \hat{S}A$ also holds.

Finally, $FS(\hat{\tau}') = FS(R'\tau') \subseteq FS(\hat{S}A) = FS(R'S'USA)$ implies $FS(\tau') \subseteq FS(S'USA)$. This, together with the definition of *skolem* guar-

antees that the **if** statement succeeds. Consequently, $TC(A, \textbf{let}\ \ldots)$ succeeds with $(S'US, \tau')$, and R' is the desired substitution.

■

Corollary 3.10 [Principal type] If $TC(A, e) = (S, \tau)$, then τ is a principal type for e under A.

3.6 Semantics

We give a standard denotational semantics. The evaluation function E maps an expression $e \in Exp$ to some semantic value v, in the context of an evaluation environment $\rho \in Env$. An evaluation environment is a partial mapping from identifiers to semantic values. Runtime type errors are represented by the special value wrong. Tagged values are used to capture the semantics of algebraic data types.

We distinguish between the three error situations, runtime type errors (wrong), nontermination, and a mismatch when an attempt is made to decompose a tagged value whose tag does not match the tag of the destructor. Both nontermination and mismatch are expressed by $\perp$.

Our type inference system is sound with respect to the evaluation function; a well-typed program never evaluates to wrong. The formal proof for semantic soundness is given below.

It should be noted that we do not commit ourselves to a strict or non-strict evaluation function. Therefore, our treatment of existential types applies to languages with both strict and non-strict semantics. In either case, appropriate conditions would have to be added to the definition of the evaluation function for pair expressions, function applications, **let** expressions, and pattern-matching **let** expressions: the strict evaluation function returns $\perp$ whenever a subexpression evaluates to $\perp$, while the non-strict evaluation function retains $\perp$ as the value of that subexpression.

3.6.1 Semantic Domain

Unit value $U = \{\mathsf{unit}\}_\perp$

Boolean values $B = \{\mathsf{false}, \mathsf{true}\}_\perp$

Constructor tags C

Semantic domain

$$V \cong U + B + (V \to V) + (V \times V) + (C \times V) + \{\mathsf{wrong}\}_\perp$$

In the latter definition of V, $+$ stands for the coalesced sum, so that all types over V share the same $\perp$.

3.6.2 Semantics of Expressions

The semantic function for expressions,

$$E : Exp \to Env \to V,$$

is defined as follows:

$$E[\![x]\!]\,\rho = \rho(x)$$

$$E[\![(e_1, e_2)]\!]\,\rho = \langle E[\![e_1]\!]\,\rho, E[\![e_2]\!]\,\rho \rangle$$

$$E[\![e\,e']\!]\,\rho = \text{if } E[\![e]\!]\,\rho \in V \to V \text{ then } (E[\![e]\!]\,\rho)\,(E[\![e']\!]\,\rho) \text{ else wrong}$$

$$E[\![\lambda x.e]\!]\,\rho = \lambda v \in V.E[\![e]\!](\rho[v/x])$$

$$E[\![\textbf{let } x = e \textbf{ in } e']\!]\,\rho = E[\![e']\!]\,(\rho[E[\![e]\!]\,\rho/x])$$

$$E[\![\,\textbf{data}\ \sigma\ \textbf{in}\ e\,]\!]\,\rho = E[\![\,e\,]\!]\,\rho$$

$$E[\![\,K\,]\!]\,\rho = \lambda v \in V.\langle K,v\rangle$$

$$E[\![\,\textbf{is}\ K\,]\!]\,\rho = \lambda v \in V.\text{if } v \in \{K\} \times V \text{ then true else false}$$

$$E[\![\,\textbf{let}\ K\ x = e\ \textbf{in}\ e'\,]\!]\,\rho = E[\![\,e'\,]\!]\,(\rho[\text{if } E[\![\,e\,]\!]\,\rho \in \{K\} \times V \text{ then } \text{snd}\,(E[\![\,e\,]\!]\,\rho) \text{ else } \bot/x])$$

3.6.3 Semantics of Types

Following [MPS86], we identify types with *weak ideals* over the semantic domain V. A type environment $\psi \in TEnv$ is a partial mapping from type variables to ideals and from Skolem type constructors to functions between ideals. The semantic interpretation of types,

$$T : TExp \to TEnv \to \Im(V)$$

is defined as follows.

$$T[\![\,unit\,]\!]\,\psi = U$$

$$T[\![\,bool\,]\!]\,\psi = B$$

$$T[\![\,\alpha\,]\!]\,\psi = \psi(\alpha)$$

$$T[\![\,\tau_1 \times \tau_2\,]\!]\,\psi = T[\![\,\tau_1\,]\!]\,\psi \times T[\![\,\tau_2\,]\!]\,\psi$$

$$T[\![\,\tau \to \tau'\,]\!]\,\psi = T[\![\,\tau\,]\!]\,\psi \to T[\![\,\tau'\,]\!]\,\psi$$

$$T[\![\,\kappa(\tau_1, \ldots, \tau_n)\,]\!]\,\psi = (\psi(\kappa))\,(T[\![\,\tau_1\,]\!]\,\psi, \ldots, T[\![\,\tau_n\,]\!]\,\psi)$$

$$T[\![\mu\beta.\sum K_i\eta_i]\!]\psi = \mu(\lambda I \in \Im(V).\sum\{K_i\} \times T[\![\eta_i]\!](\psi[I/\beta]))$$

$$T[\![\forall\alpha.\sigma]\!]\psi = \bigcap_{I \in \Re} \lambda I \in \Im(V).T[\![\sigma]\!](\psi[I/\alpha])$$

$$T[\![\exists\alpha.\eta]\!]\psi = \bigsqcup_{I \in \Re} \lambda I \in \Im(V).T[\![\eta]\!](\psi[I/\alpha])$$

The universal and existential quantifications range over the set $\Re \subseteq \Im(V)$ of all ideals that do not contain wrong. Note that the sum in the definition of recursive types is actually a union, since the constructor tags are assumed to be distinct. It should also be noted that our interpretation does not handle ML's nonregular, mutually recursive datatypes. An adequate model can be given by extending the semantics described in [MPS86] to handle full ML datatypes [Aba92]; the machinery for this model is given in [Plo83]. An adequate semantics can also be found in the PER model described in [BM92].

Theorem 3.11 The semantic function for types is well-defined.

Proof: As in [MPS86]. We observe that

$\lambda I \in \Im(V).\sum\{K_i\} \times T[\![\eta_i]\!](\psi[I/\alpha])$ is always contractive, since cartesian product and sum of ideals are contractive; therefore, the fixed point of such a function exists.

Lemma 3.12 Let ψ be a type environment such that for every $\alpha \in \mathrm{Dom}\,\psi$, $\mathsf{wrong} \notin \psi(a)$. Then for every type scheme σ, $\mathsf{wrong} \notin T[\![\sigma]\!]\psi$.

Proof: By structural induction on σ.

Lemma 3.13 [Substitution]
$T[\![\sigma\,[\sigma'/\alpha]]\!]\psi = T[\![\sigma]\!](\psi[T[\![\sigma']\!]\psi/\alpha])$.

Proof: Again, by structural induction on σ.

Definition 3.2 [Semantic type judgment] Let A be an assumption set, e an expression, and σ a type scheme. We define $\models_{\rho,\psi} A$ as meaning that $\mathrm{Dom}A \subseteq \mathrm{Dom}\rho$ and for every $x \in \mathrm{Dom}A$, $\rho(x) \in T[\![A(x)]\!]\psi$; further, we say $A \models_{\rho,\psi} e : \sigma$ iff $\models_{\rho,\psi} A$ implies $E[\![e]\!]\rho \in T[\![\sigma]\!]\psi$; and finally, $A \models e : \sigma$ means that for all $\rho \in \mathit{Env}$ and $\psi \in \mathit{TEnv}$ we have $A \models_{\rho,\psi} e : \sigma$.

Theorem 3.14 [Semantic soundness] If $A \vdash e : \tau$ then $A \models e : \tau$.

Proof: By induction on the size of the proof tree for $A \vdash e : \tau$. We need to consider each of the cases given by the type inference rules. Applying the inductive assumption and the typing judgments from the preceding steps in the type derivation, we use the semantics of the types of the partial results of the evaluation. In each of the cases below, choose ψ and ρ arbitrarily, such that $\models_{\rho,\psi} A$. We include only the nonstandard cases. Lemma 3.13 will be used with frequency.

$A \vdash$ **data** $\forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$ **in** $e : \tau$

The premise in the type derivation is $A[\sigma/K_1, \ldots, \sigma/K_m] \vdash e : \tau$, where $\sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$. Since by definition, $\models_{\rho,\psi} A[\sigma/K_1, \ldots, \sigma/K_m]$, we can use the inductive assumption to obtain $E[\![$**data** $\forall\alpha_1 \ldots \alpha_n.\chi$ **in** $e]\!]\rho = E[\![e]\!]\rho \in T[\![\tau]\!]\psi$.

$A \vdash K : \tau \to \mu\beta.\Sigma[K\eta]$

The last premise in the type derivation is $\eta\,[\mu\beta.\Sigma[K\eta]/\beta] \leq \tau$, where $\eta = \exists\gamma_1 \ldots \gamma_n.\hat{\tau}$. By definition of instantiation of existential types,

$\tau = \hat{\tau}[\tau_j/\gamma_j, \mu\beta.\Sigma[K\eta]/\beta]$ for some types $\tau_1, \ldots, \tau_n$.

First, choose an arbitrary $v \in T[\![\tau]\!]\psi$ and a finite $a \leq v$. Now,

$$\begin{aligned}
a \in \quad & (T[\![\hat{\tau}[\tau_j/\gamma_j, \mu\beta.\Sigma[K\eta]/\beta]]\!]\psi)^\circ \\
= \quad & (T[\![\hat{\tau}[\mu\beta.\Sigma[K\eta]/\beta]]\!](\psi[T[\![\tau_j]\!]\psi/\gamma_j]))^\circ \\
\subseteq \quad & \bigcup_{J_1, \ldots, J_n \in \Re} (T[\![\hat{\tau}[\mu\beta.\Sigma[K\eta]/\beta]]\!](\psi[J_j/\gamma_j]))^\circ \\
= \quad & (\bigsqcup_{J_1, \ldots, J_n \in \Re} T[\![\hat{\tau}[\mu\beta.\Sigma[K\eta]/\beta]]\!](\psi[J_j/\gamma_j]))^\circ \\
= \quad & (T[\![\eta[\mu\beta.\Sigma[K\eta]/\beta]]\!]\psi)^\circ.
\end{aligned}$$

Hence, $v = \bigsqcup \{a \mid a \text{ finite and } a \leq v\} \in T[\![\eta[\mu\beta.\Sigma[K\eta]/\beta]]\!]\psi$, by closure of ideals under limits. Consequently,

$$\begin{aligned}
\langle K, v\rangle \in \quad & \{K\} \times T[\![\eta[\mu\beta.\Sigma[K\eta]/\beta]]\!]\psi \\
\subseteq \quad & \ldots + \{K\} \times T[\![\eta[\mu\beta.\Sigma[K\eta]/\beta]]\!]\psi + \ldots \\
= \quad & \ldots + \{K\} \times T[\![\eta]\!](\psi[T[\![\mu\beta.\Sigma[K\eta]]\!]\psi/\beta]) + \ldots \\
= \quad & T[\![\mu\beta.\Sigma[K\eta]]\!]\psi.
\end{aligned}$$

Hence $E[\![K]\!]\rho \in T[\![\tau \to \mu\beta.\Sigma[K\eta]]\!]\psi$.

$A \vdash$ **is** $K : (\mu\beta.\Sigma[K\eta]) \to bool$

Choose an arbitrary $v \in T[\![\mu\beta.\Sigma[K\eta]]\!]\psi$. Clearly,

$(E[\![$**is** $K]\!]\rho)v \in B$, whence

$E[\![$**is** $K]\!]\rho \in T[\![(\mu\beta.\Sigma[K\eta]) \to bool]\!]\psi$.

$A \vdash$ **let** $K\ x = e$ **in** $e' : \tau'$

We follow the proofs in [Dam85] and [MPS86]. The first premise in the type derivation is $A \vdash e : \tau$, where $\tau = \mu\beta.\Sigma[K\eta]$ and

$\eta = \exists\gamma_1 \ldots \gamma_n.\hat{\tau}$. Let $\{\alpha_1, \ldots, \alpha_k\} = FV(\tau) \setminus FV(A)$. Then, for arbi-

trary $I_1, \ldots, I_k \in \mathfrak{R}$, $\models_{\rho,\, \psi\,[I_i/\alpha_i \mid\, i = 1 \ldots k]} A$ holds, since none of the α_i's are free in A.

Let $v = E[\![e]\!]\,\rho$; by the inductive assumption, $v \in T[\![\tau]\!]\ (\psi\,[I_i/\alpha_i])$. Consequently,

$$
\begin{aligned}
v \in \quad & \bigcap_{I_1, \ldots, I_k \in \mathfrak{R}} T[\![\tau]\!]\ (\psi\,[I_i/\alpha_i]) \\
= \quad & \bigcap_{I_1, \ldots, I_k \in \mathfrak{R}} T[\![\mu\beta.\Sigma\,[K\eta]]\!]\ (\psi\,[I_i/\alpha_i]) \\
= \quad & \ldots + \\
& \{K\} \times \bigcap_{I_1, \ldots, I_k \in \mathfrak{R}} T[\![\eta]\!]\ (\psi\,[I_i/\alpha_i,\ T[\![\tau]\!]\ (\psi\,[I_i/\alpha_i])\,/\beta]) \\
& + \ldots .
\end{aligned}
$$

First, consider the more interesting case, $\mathsf{fst}\,(v) = K$. Then

$$
\mathsf{snd}\,(v) \in \bigcap_{I_1, \ldots, I_k \in \mathfrak{R}} \bigsqcup_{J_1, \ldots, J_n \in \mathfrak{R}} T[\![\hat{\tau}]\!]\ (\psi\,[I_i/\alpha_i, J_j/\gamma_j, T[\![\tau]\!]\ (\psi\,[I_i/\alpha_i])\,/\beta])
$$

Let $\alpha_1, \ldots, \alpha_h$, $h \le k$, be those variables among $\alpha_1, \ldots, \alpha_k$ that are free in $\hat{\tau}\,[\tau/\beta]$.

We now choose a finite a such that $a \le \mathsf{snd}\,(v)$, thus

$$
a \in \bigcap_{I_1, \ldots, I_h \in \mathfrak{R}} \bigcup_{J_1, \ldots, J_n \in \mathfrak{R}} (\, T[\![\hat{\tau}\,[\tau/\beta]]\!]\ (\psi\,[I_i/\alpha_i, J_j/\gamma_j])\,)^{\circ}.
$$

By definition of set union and intersection, there exist functions

$f_1, \ldots, f_n \in \Im(V)^h \to \Im(V)$,

such that

$$
\begin{aligned}
a \in \quad & \bigcap_{I_1, \ldots, I_h \in \Re} (T[\![\hat{\tau}\,[\tau/\beta]]\!]\ (\psi\,[I_i/\alpha_i, f_j(I_1, \ldots, I_h)/\gamma_j]))^\circ \\
\subseteq \quad & \bigcap_{I_1, \ldots, I_h \in \Re} T[\![\hat{\tau}\,[\tau/\beta]]\!]\ (\psi\,[I_i/\alpha_i, f_j(I_1, \ldots, I_h)/\gamma_j]) \\
= \quad & \bigcap_{I_1, \ldots, I_h \in \Re} T[\![\hat{\tau}\,[\kappa_j(\alpha_1, \ldots, \alpha_h)/\gamma_j, \tau/\beta]]\!]\ (\psi\,[I_i/\alpha_i, f_j/\kappa_j]) \\
= \quad & T[\![\forall\alpha_1 \ldots \alpha_h.\hat{\tau}\,[\kappa_j(\alpha_1, \ldots, \alpha_h)/\gamma_j, \tau/\beta]]\!]\ (\psi\,[f_j/\kappa_j]) \\
= \quad & T[\![\mathit{gen}\,(A, \mathit{skolem}\,(A, \eta\,[\tau/\beta]))]\!]\ (\psi\,[f_j/\kappa_j]),
\end{aligned}
$$

assuming that the κ_j's are the ones generated by $\mathit{skolem}\,(A, \eta\,[\tau/\beta])$. Since by definition of *skolem*, none of the κ_j's are free in A, $\models_{\rho, \psi\,[f_j/\kappa_j]} A$ holds and we can extend A and ρ, obtaining

$\models_{\rho\,[a/x], \psi\,[f_j/\kappa_j]} A\,[\mathit{gen}\,(A, \mathit{skolem}\,(A, \eta\,[\tau/\beta]))/x]$.

We now apply the inductive assumption to the last premise,

$A\,[\mathit{gen}\,(A, \mathit{skolem}\,(A, \eta\,[\tau/\beta]))/x] \vdash e' : \tau'$,

and obtain

$E[\![e']\!]\ (\rho\,[a/x]) \in T[\![\tau']\!]\ (\psi\,[f_j/\kappa_j]) = T[\![\tau']\!]\psi$,

since $FS(\tau') \subseteq FS(A)$. Finally,

$$
\begin{aligned}
& E[\![\,\mathbf{let}\ \ K\ x = e\ \ \mathbf{in}\ \ e'\,]\!]\,\rho \\
= \quad & E[\![e']\!]\ (\rho\,[\mathsf{snd}\,(E[\![e]\!]\,\rho)/x]) \\
= \quad & \bigsqcup \{E[\![e']\!]\ (\rho\,[a/x]) \mid a \text{ finite and } a \leq \mathsf{snd}\,(E[\![e]\!]\,\rho)\},
\end{aligned}
$$

by the continuity of E. The latter expression is in $T[\![\tau']\!]\psi$ by the closure of ideals under limits.

In the second case, $\mathsf{fst}(v) \neq K$. For any functions $f_1, \ldots, f_n \in \Im(V)^h \to \Im(V)$ we have $\bot \in T[\![gen(A, skolem(A, \eta[\tau/\beta]))]\!](\psi[f_j/\kappa_j])$. Again, since none of the κ_j's are free in A, $\models_{\rho, \psi[f_j/\kappa_j]} A$ holds and we can extend A and ρ, obtaining

$$\models_{\rho[\bot/x], \psi[f_j/\kappa_j]} A[gen(A, skolem(A, \eta[\tau/\beta]))/x] .$$

By applying the inductive assumption to the last premise,

$$A[gen(A, skolem(A, \eta[\tau/\beta]))/x] \vdash e' : \tau',$$

we obtain

$$E[\![e']\!](\rho[\bot/x]) \in T[\![\tau']\!](\psi[f_j/\kappa_j]) = T[\![\tau']\!]\psi .$$

This concludes our proof of semantic soundness.

■

Corollary 3.15 [Semantic soundness] Let ψ be a type environment such that for every $\alpha \in \mathrm{Dom}\,\psi$, $\mathsf{wrong} \notin \psi(a)$. If $A \vdash e : \tau$ and $\models_{\rho, \psi} A$, then $E[\![e]\!]\rho \neq \mathsf{wrong}$.

Proof: We apply Lemma 3.12 to Theorem 3.14.

4 An Extension of ML with a Dotless Dot Notation

In this chapter, we describe a extension of our language that allows more flexible use of existential types. Following notations used in actual programming languages, this extension assumes the same representation type each time a value of existential type is accessed, provided that each access is through the same identifier. We give a type reconstruction algorithm and show semantic soundness by translating into the language from Chapter 3.

4.1 Introduction

MacQueen [Mac86] observes that the use of existential types in connection with an elimination construct (**`open`**, **`abstype`**, or our **`let`**) is impractical in certain programming situations; often, the scope of the elimination construct has to be made so large that some of the benefits of abstraction are lost. In particular, the lowest-level entities have to be opened at the outermost level; these are the traditional disadvantages of block-structured languages.

We present an extension of ML that provides the same flexibility as the dot notation described in [CL90]. In this extension, abstract types are again modeled by ML datatypes with existentially quantified component types. Values of abstract type are created by applying a datatype constructor to a

value, and they are decomposed in a pattern-matching **let** expression. However, we allow existentially quantified type variables to escape the scope of the identifier in whose type they appear, as long as the expression decomposed is an identifier and the existentially quantified type variables do not escape the scope of that identifier. Each decomposition of an identifier, using the same constructor, produces identical existentially quantified type variables. We call our notation a "dotless" dot notation, since it uses decomposition by pattern-matching instead of record component selection.

4.2 Some Motivating Examples

We assume the type declaration

```
datatype Key = key of 'a * ('a -> int)
```

in the following examples. In the first example,

```
let val x = key(3,fn x => x + 2) in
    (let val key(_,f) = x in f end)
        (let val key(v,_) = x in v end)
end
```

the existential type variable in the type of **f** is the same as the one in the type of **v**, and the function application produces a result of type **int**. This follows from the fact that both **f** and **v** are bound by decomposition of the same[1] identifier, **x**. Consequently, they must hold the same value and the whole expression is type-correct.

[1] We assume the ML scoping discipline, which uses **let** statements as scope boundaries; alternatively, one could require each bound identifier to be unique.

In a language with the traditional dot notation, for example Ada, abstract types can be modeled as packages, and an example corresponding to the previous one would look as follows:

```
package KEY_PKG is
    type KEY is private;
    X : constant KEY;
    function F(X : KEY) return INTEGER;
private
    type KEY is INTEGER;
    X : constant KEY := 3;
end KEY_PKG;

package body KEY_PKG is
    function F(X : KEY) return INTEGER is
    begin
        return X + 2;
    end;
end KEY_PKG;

var Z : INTEGER;
...

Z := KEY_PKG.F(KEY_PKG.X);
```

The components of the abstract type **KEY_PKG** are selected using the dot notation.

The following are examples of incorrect programs. For instance,

```
let val x = key(3,fn x => x + 2) in
    let val key(_,f) = x in
        f
    end
end
```

is not type-correct, since the existential type variable in the type of **f** escapes the scope of **x**. Neither is the following program,

```
let val x = key(3,fn x => x + 2)
    val y = x
in
    (let val key(_,f) = x in f end)
        (let val key(v,_) = y in v end)
end
```

since different identifiers produce different existential type variables, although they hold the same values in this case. As the latter cannot be determined statically, we must assume that the values have different types. Similarly,

```
val z = (3,fn x => x + 2)
let val key(_,f) = key z in
    let val key(v,_) = key z in
        f v
    end
end
```

is not type-correct. Since the expressions that are decomposed are not even identifiers, we cannot assume statically that **f** can be applied to **v**.

4.3 Syntax

4.3.1 Language Syntax

Syntactically, our underlying formal language is almost unchanged, except that pattern-matching **let** expressions only allow an identifier to be decomposed, not a general expression. This is not a significant restriction, since we can always bind the expression in an enclosing **let** before decomposing it. Again, we assume that each identifier bound by a λ or **let** expression is unique.

Identifiers x

Constructors	K	
Type constructors	T	
Expressions	e ::=	**()** | **true** | **false** | x | (e_1, e_2) | $e\ e'$ | $\lambda x.e$ | **let** $x = e$ **in** e' | **data** $\forall\alpha_1 \ldots \alpha_n.\chi$ **in** e | K | **is** K | **let** $K\ x = x'$ **in** e'

4.3.2 Type Syntax

Type variables	α	
Skolem functions	κ	
Types	τ ::=	*unit* | *bool* | α | $\tau_1 \times \tau_2$ | $\tau \rightarrow \tau'$ | $\kappa_{x,K,i}(\tau_1, \ldots, \tau_n)$ | χ
Recursive types	χ ::=	$\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$ where $K_i \neq K_j$ for $i \neq j$
Existential types	η ::=	$\exists\alpha.\eta$ | τ
Type schemes	σ ::=	$\forall\alpha.\sigma$ | τ
Assumptions	a ::=	σ / x | $\forall\alpha_1 \ldots \alpha_n.\chi$

Our type syntax is almost unchanged. However, Skolem type constructors are now uniquely associated with an identifier x by using the symbol κ, indexed by x, the constructor K used in the decomposition, and the index i of the existentially quantified variable γ_i to be replaced.

4.4 Type Inference

4.4.1 Instantiation and Generalization of Type Schemes

$\forall\alpha_1 \ldots \alpha_n.\tau \geq \tau'$ iff there are types $\tau_1, \ldots \tau_n$ such that

$$\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

$\exists\alpha_1 \ldots \alpha_n.\tau \leq \tau'$ iff there are types $\tau_1, \ldots \tau_n$ such that

$$\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

$$gen(A, \tau) = \forall(FV(\tau) \setminus FV(A)).\tau$$

$$skolem^{\cdot}(A, x, K, \exists\gamma_1 \ldots \gamma_n.\tau) =$$

$$\tau[\kappa_{x,K,i}(\alpha_1, \ldots \alpha_k)/\gamma_i] \text{ where}$$

$$\{\alpha_1, \ldots, \alpha_k\} = FV(\exists\gamma_1 \ldots \gamma_n.\tau) \setminus FV(A)$$

Instantiation and generalization are unchanged. The modified function $skolem^{\cdot}$ replaces each existentially quantified variable in a type by a unique type constructor whose actual arguments are those free variables of the type that are not free in the assumption set. Since identifiers are unique, we obtain Skolem constructors uniquely associated with an identifier x by using the symbol κ, indexed by x, the constructor K used in the decomposition, and the index i of the existentially quantified variable γ_i to be replaced. In addition to FV, the set of free type variables in a type scheme or assumption set, we use FS_x, the set of those Skolem type constructors that occur in a type scheme or assumption set and are associated with identifier x.

4.4.2 Inference Rules for Expressions

The first three typing rules are the same as in the original system.

$$\text{(VAR}^{\cdot}\text{)} \quad \frac{A(x) \geq \tau}{A \vdash^{\cdot} x : \tau}$$

$$\text{(PAIR}^{\cdot}\text{)} \quad \frac{A \vdash^{\cdot} e_1 : \tau_1 \qquad A \vdash^{\cdot} e_2 : \tau_2}{A \vdash^{\cdot} (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\text{(APPL}^{\cdot}\text{)} \quad \frac{A \vdash^{\cdot} e : \tau' \to \tau \qquad A \vdash^{\cdot} e' : \tau'}{A \vdash^{\cdot} e\ e' : \tau}$$

The ABS$^{\cdot}$ and LET$^{\cdot}$ rules are modified to prevent Skolem constructors associated with a bound variable to escape the scope of that variable.

$$\text{(ABS}^{\cdot}\text{)} \quad \frac{A\,[\tau'/x] \vdash^{\cdot} e : \tau \qquad FS_x(A) \cup FS_x(\tau) = \varnothing}{A \vdash^{\cdot} \lambda x.e : \tau' \to \tau}$$

$$\text{(LET}^{\cdot}\text{)} \quad \frac{A \vdash^{\cdot} e : \tau \qquad A\,[gen(A, \tau)/x] \vdash^{\cdot} e' : \tau' \qquad FS_x(A) \cup FS_x(\tau') = \varnothing}{A \vdash^{\cdot} \textbf{let}\ \ x = e\ \ \textbf{in}\ \ e' : \tau'}$$

The rules DATA$^{\cdot}$, CONS$^{\cdot}$, TEST$^{\cdot}$ remain unchanged.

$$\text{(DATA}^{\cdot}\text{)} \quad \frac{\sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m \qquad FV(\sigma) = \varnothing \qquad A\,[\sigma/K_1, \ldots, \sigma/K_m] \vdash^{\cdot} e : \tau}{A \vdash^{\cdot} \textbf{data}\ \ \sigma\ \ \textbf{in}\ \ e : \tau}$$

$$\text{(CONS}^{\cdot}\text{)} \quad \frac{A(K) \geq \mu\beta.\Sigma\,[K\eta] \qquad \eta\,[\mu\beta.\Sigma\,[K\eta]/\beta] \leq \tau}{A \vdash^{\cdot} K : \tau \to \mu\beta.\Sigma\,[K\eta]}$$

$$\text{(TEST}^{\cdot}\text{)} \quad \frac{A(K) \geq \mu\beta.\Sigma[K\eta]}{A \vdash^{\cdot} \textbf{is}\ \ K : (\mu\beta.\Sigma[K\eta]) \rightarrow \mathit{bool}}$$

$$\text{(PAT}^{\cdot}\text{)} \quad \frac{A(K) \geq \mu\beta.\Sigma[K\eta] \qquad A(x) \geq \mu\beta.\Sigma[K\eta] \qquad A[gen(A, \mathit{skolem}^{\cdot}(A, x, K, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x'] \vdash^{\cdot} e : \tau}{A \vdash^{\cdot} \textbf{let}\ \ K\ x' = x\ \ \textbf{in}\ \ e : \tau}$$

The new PAT$^{\cdot}$ rule does not enforce any restriction on occurrence of Skolem constructors. It only requires that the variable x be of the same type as the result type of the constructor K. The body e is typed under the assumption set extended with an assumption about the bound identifier x'.

4.5 Type Reconstruction

Again, the type reconstruction algorithm is a straightforward translation from the deterministic typing rules.

4.5.1 Auxiliary Functions

While $\mathit{inst}_{\forall}$ and $\mathit{inst}_{\exists}$ are as in the preceding chapter, the other auxiliary functions are the same as in the inference rules.

4.5.2 Algorithm

Our type reconstruction function takes an assumption set and an expression, and it returns a substitution and a type expression. There is one case for each typing rule.

$$TC^{\cdot}(A, x) =$$
$$(Id, \mathit{inst}_{\forall}(A(x)))$$

$TC^{\cdot}(A,\ (e_1, e_2))\ =$

let $(S_1, \tau_1) = TC^{\cdot}(A, e_1)$

$(S_2, \tau_2) = TC^{\cdot}(S_1 A, e_2)$

in $(S_2 S_1, S_2 \tau_1 \times \tau_2)$

$TC^{\cdot}(A, e\ e')\ =$

let $(S, \tau) = TC^{\cdot}(A, e)$

$(S', \tau') = TC^{\cdot}(SA, e')$

β be a fresh type variable

$U = mgu\ (S'\tau, \tau' \to \beta)$

in $(US'S, U\beta)$

$TC^{\cdot}(A, \lambda x.e)\ =$

let β be a fresh type variable

$(S, \tau) = TC^{\cdot}(A\,[\beta / x]\,, e)$

in

if $FS_x(SA) \cup FS_x(\tau) = \varnothing$ **then**

$(S, S\beta \to \tau)$

$TC^{\cdot}(A,$ **let** $x = e$ **in** $e')\ =$

let $(S, \tau) = TC^{\cdot}(A, e)$

$(S', \tau') = TC^{\cdot}(SA\,[gen\ (SA, \tau) / x]\,, e')$

in

if $FS_x(S'SA) \cup FS_x(\tau') = \varnothing$ **then**

$(S'S, \tau')$

$$TC^{\cdot}(A, \textbf{data}\ \sigma\ \textbf{in}\ e) =$$
$$\textbf{let}\ \ \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m = \sigma\ \textbf{in}$$
$$\textbf{if}\ \ FV(\sigma) = \varnothing\ \textbf{then}$$
$$TC^{\cdot}(A\,[\sigma/K_1, \ldots, \sigma/K_m], e)$$

$$TC^{\cdot}(A, K) =$$
$$\textbf{let}\ \ \tau = inst_{\forall}\,(A\,(K))$$
$$\mu\beta.\ldots + K\eta + \ldots = \tau$$
$$\textbf{in}\ \ (Id,\ (inst_{\exists}\,(\eta\,[\tau/\beta]))) \rightarrow \tau)$$

$$TC^{\cdot}(A, \textbf{is}\ K) =$$
$$\textbf{let}\ \ \tau = inst_{\forall}\,(A\,(K))$$
$$\textbf{in}\ \ (Id, \tau \rightarrow bool)$$

$$TC^{\cdot}(A, \textbf{let}\ \ K\ x' = x\ \textbf{in}\ \ e') =$$
$$\textbf{let}\ \ \tau = inst_{\forall}(A(x))$$
$$U = mgu(\tau, inst_{\forall}(A(K)))$$
$$\mu\beta.\ldots + K\eta + \ldots = U\tau$$
$$\tau_{\kappa} = skolem^{\cdot}(UA, x, K,\ (\eta\,[U\tau/\beta]))$$
$$(S, \tau') = TC^{\cdot}(UA\,[gen\,(UA, \tau_{\kappa})/x'], e')$$
$$\textbf{in}$$
$$(SU, \tau')$$

4.5.3 Syntactic Soundness and Completeness of Type Reconstruction

Lemma 4.1 [Stability of $\vdash^{\cdot}$] If $A \vdash^{\cdot} e : \tau$ and S is a substitution, then $SA \vdash^{\cdot} e : S\tau$ also holds. Moreover, if there is a proof tree for $A \vdash^{\cdot} e : \tau$ of height n, then there is also a proof tree for $SA \vdash^{\cdot} e : S\tau$ of height less or equal to n.

Theorem 4.2 [Syntactic soundness] If $TC^{\cdot}(A, e) = (S, \tau)$, then $SA \vdash^{\cdot} e : \tau$.

Definition 4.1 [Principal Type] τ is a principal type of expression e under assumption set A if $A \vdash^{\cdot} e : \tau$ and whenever $A \vdash^{\cdot} e : \tau'$ then there is a substitution S such that $S\tau = \tau'$.

Theorem 4.3 [Syntactic completeness] If $\hat{S}A \vdash^{\cdot} e : \hat{\tau}$, then $TC^{\cdot}(A, e) = (S, \tau)$ and there is a substitution R such that $\hat{S}A = RSA$ and $\hat{\tau} = R\tau$.

Corollary 4.4 [Principal type] If $TC^{\cdot}(A, e) = (S, \tau)$, then τ is a principal type for e under A.

Proof: We modify the proofs given in Chapter 3.

4.6 A Translation Semantics

We retain our original semantic interpretation $E[\![\ ]\!]$. Following [CL90], we prove semantic soundness by giving a type- and semantics-preserving translation to our original language. The idea is that we can enclose an expression e with subexpressions of the form **let** $K\ x' = x$ **in** e' by an outer expres-

sion that defines x' and replace **let** $K\ x' = x$ **in** e' by e'. That is, we replace e by

$$\textbf{let}\ K\ x_K = x\ \textbf{in}\ e\,[e'\,[x_K / x']\,/\,\textbf{let}\ K\ x' = x\ \textbf{in}\ e']$$

We chose the enclosing **let** expression defining x' large enough so that no existentially quantified type variables arising through the inner **let** expressions escape this outer definition. Since the ABS$^{\cdot}$ and LET$^{\cdot}$ rules guarantee that no existentially quantified variables emerging from the decomposition of x escape the scope of x, it is safe to enclose the whole body of the λ or **let** expression.

However, we must be careful, since the outer decomposition in the translation might fail, while the inner decomposition in the original expression might not necessarily have been reached; this is possible if the value held by x does not have the constructor tag K. Therefore, we need to replace e by an **if** expression with branches for each constructor tag in the datatype that x has. This is reflected in the definition of the auxiliary translation function $\| \ \|$ below.

4.6.1 Modified Original Language

Type judgments in a modified version of the original language are of the form $A \vdash^{\circ} e : \tau$. We modify the *skolem* function and the PAT rule of our original language:

$$skolem^{\circ}(A, x, \exists\gamma_1 \ldots \gamma_n.\tau) = \tau\,[\kappa_{x,\,i}\,(\alpha_1, \ldots \alpha_k)\,/\,\gamma_i] \ \text{where}$$
$$\{\alpha_1, \ldots, \alpha_k\} = FV(\exists\gamma_1 \ldots \gamma_n.\tau) \setminus FV(A)$$

Unique Skolem type constructors can be generated by using the symbol κ, indexed by the unique name x of the bound identifier and the index i of the existentially quantified type variable γ_i to be replaced.

$$\text{(PAT}^\circ\text{)}\quad \frac{A \vdash^\circ e : \mu\beta.\Sigma\,[K\eta] \qquad FS_x(A) \cup FS_x(\tau') = \varnothing \qquad A\,[gen(A, skolem^\circ(A, x, \eta\,[\mu\beta.\Sigma\,[K\eta]\,/\beta]))/x'] \vdash^\circ e' : \tau'}{A \vdash^\circ \textbf{let}\ \ K\ x = e\ \ \textbf{in}\ \ e' : \tau'}$$

Using this modified *skolem*° function, the PAT° rule can enforce that newly generated Skolem constructors escape their scope by the condition $FS_x(A) \cup FS_x(\tau') = \varnothing$, which expresses that no Skolem constructor associated with x may escape the scope of x.

It is easy to see that this language has the same properties as the original one, in particular, semantic soundness.

4.6.2 Auxiliary Translation Function

The bodies of λ and **`let`** expressions are translated by the auxiliary function given below. It moves all pattern-matching **`let`** expressions that decompose the variable bound by the enclosing λ or **`let`** expression to the outermost level possible.

We use a *conformity check* in form of a nested **`if`** expression with **`is`** expressions to determine the constructor tag of the value held by x. This requires us to evaluate[1] x; consequently, the resulting expression is always strict in x. Therefore, this translation is not semantics-preserving if the original expression was non-strict in x. We need to distinguish between the translation of the strict and the non-strict version of our language:

- In the strict language, the expression bound to x is already evaluated at binding time, and evaluating it again leaves the semantics unchanged.
- In the non-strict language, the expression bound to x might not be eval-

[1] It actually suffices to evaluate the argument to *weak head normal form*, so that the top-level constructor of the argument can be inspected; see [PJ87] for details. Nevertheless, the resulting translation is not semantics-preserving.

uated at all; to be semantics-preserving, the translation must not introduce additional evaluations of x.

As described in [PJ87], the only patterns for which a conformity check can be omitted safely are the *irrefutable* patterns involving datatypes with a single constructor. We therefore restrict the *non-strict* version of our language in the following way:

Existentially quantified type variables may occur only in the component types of datatypes with a single constructor.

The auxiliary translation function for the strict version of the language is defined as follows:

$$\|e\|_{x,\, K_1\eta_1 + \ldots + K_n\eta_n} =$$

$$\begin{array}{l}
\textbf{if is } K_1\ x \textbf{ then} \\
\quad \textbf{let } K_1\ x_{K_1} = x \textbf{ in } e\left[\begin{array}{l} e'\,[x_{K_1}/x'] \,/\, \textbf{let } K_1\ x' = x \textbf{ in } e' \\ \textbf{fail} \,/\, \textbf{let } K_{i\neq 1}\ x' = x \textbf{ in } e' \end{array}\right] \\
\textbf{else if is } K_2\ x \textbf{ then} \\
\qquad \ldots \\
\textbf{else if is } K_n\ x \textbf{ then} \\
\quad \textbf{let } K_n\ x_{K_n} = x \textbf{ in } e\left[\begin{array}{l} e'\,[x_{K_n}/x'] \,/\, \textbf{let } K_n\ x' = x \textbf{ in } e' \\ \textbf{fail} \,/\, \textbf{let } K_{i\neq n}\ x' = x \textbf{ in } e' \end{array}\right] \\
\textbf{else} \\
\quad e\,[\textbf{fail} \,/\, \textbf{let } K_i\ x' = x \textbf{ in } e']
\end{array}$$

In the non-strict case, there can be only a single constructor with an existential component type, and the auxiliary translation function reduces to:

$$\|e\|_{x,\, K\eta} = \textbf{let } K\ x_K = x \textbf{ in } e\,[e'\,[x_K/x'] \,/\, \textbf{let } K\ x' = x \textbf{ in } e']$$

$$\|e\|_{x,\, K_1\tau_1 + \ldots + K_n\tau_n} = e$$

4.6.3 Inference-guided Translation

We give a translation guided by the type inference rules, along the lines of [NS91]. Let e_0 be a closed, well-typed term. The translation is defined along with the type inference rules for each subterm of e_0.

$$\text{(VAR}^{\cdot}\text{)} \quad \frac{A(x) \geq \tau}{A \vdash^{\cdot} x : \tau \Rightarrow x}$$

$$\text{(PAIR}^{\cdot}\text{)} \quad \frac{A \vdash^{\cdot} e_1 : \tau_1 \Rightarrow \tilde{e}_1 \qquad A \vdash^{\cdot} e_2 : \tau_2 \Rightarrow \tilde{e}_2}{A \vdash^{\cdot} (e_1, e_2) : \tau_1 \times \tau_2 \Rightarrow (\tilde{e}_1, \tilde{e}_2)}$$

$$\text{(APPL}^{\cdot}\text{)} \quad \frac{A \vdash^{\cdot} e : \tau' \to \tau \Rightarrow \tilde{e} \qquad A \vdash^{\cdot} e' : \tau' \Rightarrow \tilde{e}'}{A \vdash^{\cdot} e\ e' : \tau \Rightarrow \tilde{e}\ \tilde{e}'}$$

$$\text{(ABS}^{\cdot}\text{)} \quad \frac{A[\tau'/x] \vdash^{\cdot} e : \tau \Rightarrow \tilde{e} \qquad \tau' \neq \mu\beta.\Sigma[K\eta]}{A \vdash^{\cdot} \lambda x.e : \tau' \to \tau \Rightarrow \lambda x.\tilde{e}}$$

$$\text{(ABS}^{\cdot\prime}\text{)} \quad \frac{\begin{array}{c} A[\mu\beta.\Sigma[K\eta]/x] \vdash^{\cdot} e : \tau \qquad FS_x(A) \cup FS_x(\tau) = \varnothing \\ A[\mu\beta.\Sigma[K\eta]/x] \vdash^{\cdot} \|e\|_{x,\,\Sigma[K\eta]} : \tau \Rightarrow \tilde{e} \end{array}}{A \vdash^{\cdot} \lambda x.e : (\mu\beta.\Sigma[K\eta]) \to \tau \Rightarrow \lambda x.\tilde{e}}$$

$$\text{(LET}^{\cdot}\text{)} \quad \frac{\begin{array}{c} A \vdash^{\cdot} e : \tau \Rightarrow \tilde{e} \qquad \tau \neq \mu\beta.\Sigma[K\eta] \\ A[gen(A, \tau)/x] \vdash^{\cdot} e' : \tau' \Rightarrow \tilde{e}' \end{array}}{A \vdash^{\cdot} \textbf{let}\ \ x = e\ \ \textbf{in}\ \ e' : \tau' \Rightarrow \textbf{let}\ \ x = \tilde{e}\ \ \textbf{in}\ \ \tilde{e}'}$$

$$(\text{LET}^{\cdot\prime}) \quad \frac{\begin{array}{c} A \vdash^{\cdot} e : \mu\beta.\Sigma[K\eta] \Rightarrow \tilde{e} \qquad FS_x(A) \cup FS_x(\tau') = \varnothing \\ A[gen(A, \mu\beta.\Sigma[K\eta])/x] \vdash^{\cdot} e' : \tau' \\ A[gen(A, \mu\beta.\Sigma[K\eta])/x] \vdash^{\cdot} \|e'\|_{x, \Sigma[K\eta]} : \tau' \Rightarrow \tilde{e}' \end{array}}{A \vdash^{\cdot} \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau' \Rightarrow \mathbf{let}\ x = \tilde{e}\ \mathbf{in}\ \tilde{e}'}$$

$$(\text{DATA}^{\cdot}) \quad \frac{\begin{array}{c} \sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m \\ FV(\sigma) = \varnothing \qquad A[\sigma/K_1, \ldots, \sigma/K_m] \vdash^{\cdot} e : \tau \Rightarrow \tilde{e} \end{array}}{A \vdash^{\cdot} \mathbf{data}\ \sigma\ \mathbf{in}\ e : \tau \Rightarrow \mathbf{data}\ \sigma\ \mathbf{in}\ \tilde{e}}$$

$$(\text{CONS}^{\cdot}) \quad \frac{A(K) \geq \mu\beta.\Sigma[K\eta] \qquad \eta[\mu\beta.\Sigma[K\eta]/\beta] \leq \tau}{A \vdash^{\cdot} K : \tau \rightarrow \mu\beta.\Sigma[K\eta] \Rightarrow K}$$

$$(\text{TEST}^{\cdot}) \quad \frac{A(K) \geq \mu\beta.\Sigma[K\eta]}{A \vdash^{\cdot} \mathbf{is}\ K : (\mu\beta.\Sigma[K\eta]) \rightarrow bool \Rightarrow \mathbf{is}\ K}$$

$$(\text{PAT}^{\cdot}) \quad \frac{\begin{array}{c} A(K) \geq \mu\beta.\Sigma[K\eta] \qquad A(x) \geq \mu\beta.\Sigma[K\eta] \\ A[gen(A, skolem^{\cdot}(A, x, K, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x'] \vdash^{\cdot} e : \tau \\ \Rightarrow \tilde{e} \end{array}}{A \vdash^{\cdot} \mathbf{let}\ K\ x' = x\ \mathbf{in}\ e : \tau \Rightarrow \mathbf{let}\ K\ x' = x\ \mathbf{in}\ \tilde{e}}$$

4.6.4 Translation of Type Schemes and Assumption Sets

After applying $\| \ \|$ to the body of a λ or **let** expression, the only pattern-matching **let** expressions left in the body are of the form **let** $K\ x_K = x$ **in** e. In the following translation, the Skolem constructors associated with x become associated with x_K. This is reflected by the following translations:

$$\lfloor \sigma \rfloor = \sigma\left[\kappa_{x_K, i}/\kappa_{x, K, i}\right]$$

$$\lfloor A \rfloor \quad = \quad [\lfloor \sigma \rfloor / x \mid A(x) = \sigma]$$

4.6.5 Properties of the Translation

Lemma 4.5 Let $\bar{A} = A[\forall\alpha_1 \ldots \alpha_k.\mu\beta.K_1\eta_1 + \ldots + K_n\eta_n / x]$. If $\bar{A} \vdash^{\cdot} e : \tau$, then $\bar{A} \vdash^{\cdot} \|e\|_{x,\, K_1\eta_1 + \ldots + K_n\eta_n} : \tau$.

Proof: In the strict case, let $1 \le i \le n$ be arbitrary. We are free to extend $\bar{A}$, assuming that x_{K_i} is not free in $\bar{A}$, hence

$\bar{A}\left[gen(\bar{A}, skolem^{\cdot}(\bar{A}, x, K_i, \eta_i)) / x_{K_i} \right] \vdash^{\cdot} e : \tau$.

Then, any subexpression **let** $K_i\ x' = x$ **in** e' of e is well-typed and we have a subproof for $A' \vdash^{\cdot}$ **let** $K_i\ x' = x$ **in** $e' : \tau'$, where $A'(x_{K_i}) = gen(\bar{A}, skolem^{\cdot}(\bar{A}, x, K_i, \eta_i))$. A premise of this judgment is $A'\left[gen(A', skolem^{\cdot}(A', x, K_i, \eta_i)) / x_{K_i} \right] \vdash^{\cdot} e' : \tau'$. Therefore, $A' \vdash^{\cdot} e'[x_{K_i} / x'] : \tau'$, since we may drop the assumption about x' after substituting x_{K_i} for it.

By replacing the proof tree for the **let** subexpression by this latter one and by observing that **fail** has any type, we can prove

$$\bar{A}\left[gen(\bar{A}, skolem^{+}(\bar{A}, x, K_i, \eta_i)) / x_{K_i} \right] \vdash^{\cdot} e \begin{bmatrix} e'[x_{K_i} / x'] / \textbf{let}\ K_i\ x' = x\ \textbf{in}\ e' \\ \textbf{fail} / \textbf{let}\ K_{j \ne i}\ x' = x\ \textbf{in}\ e' \end{bmatrix} : \tau .$$

Thus,

$$\bar{A} \vdash^{\cdot} \textbf{let}\ K_i\ x_{K_i} = x\ \textbf{in}\ e \begin{bmatrix} e'[x_{K_i} / x'] / \textbf{let}\ K_i\ x' = x\ \textbf{in}\ e' \\ \textbf{fail} / \textbf{let}\ K_{j \ne i}\ x' = x\ \textbf{in}\ e' \end{bmatrix} : \tau$$

Using a suitable typing for the **if** expressions, we conclude that

$\bar{A} \vdash^{\cdot} \| e \|_{x, K_1\eta_1 + \ldots + K_n\eta_n} : \tau.$

The claim for the non-strict case follows analogously.

■

Theorem 4.6 [Type preservation] If $A \vdash^{\cdot} e : \tau \Rightarrow \tilde{e}$, then $\lfloor A \rfloor \vdash^{\circ} \tilde{e} : \lfloor \tau \rfloor$.

Proof: By structural induction on e. We show the only interesting case, all others are straightforward.

$A \vdash^{\cdot}$ **let** $K\ x' = x$ **in** $e : \tau \Rightarrow$ **let** $K\ x' = x$ **in** $\tilde{e}$

Since our expression is a subexpression of a well-typed expression, x is bound either in a λ or in a **let** expression. Thus, it must be a subexpression of an expression of the form $\| e' \|_{x, \Sigma[K\eta]}$, where

$A'[\forall\alpha_1 \ldots \alpha_k.\mu\beta.\Sigma[K\eta]/x] \vdash^{\cdot} \| e' \|_{x, \Sigma[K\eta]} : \tau'$ and

$FS_x(A') \cup FS_x(\tau') = \varnothing$ for some A' and τ'. By definition of $\| \ \|$, the only subexpressions of $\| e' \|_{x, \Sigma[K\eta]}$ of the form **let** $K\ x' = x$ **in** e are the branches of the **if** expression, each of the form **let** $K\ x_K = x$ **in** e, where

$A'[\forall\alpha_1 \ldots \alpha_k.\mu\beta.\Sigma[K\eta]/x] \vdash^{\cdot}$ **let** $K\ x_K = x$ **in** $e : \tau'$ and

$FS_x(A') \cup FS_x(\tau') = \varnothing$; therefore $\tau = \tau'$ and $A = A'$ in the subproof. As a premise, we have

$$A[gen(A, skolem^{\cdot}(A, x, K, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x_K] \vdash^{\cdot} e : \tau \Rightarrow \tilde{e};$$

by the inductive assumption,

$$\lfloor A[gen(A, skolem^{\cdot}(A, x, K, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x_K] \rfloor \vdash^{\circ} \tilde{e} : \lfloor \tau \rfloor,$$

hence

$$\lfloor A \rfloor\,[gen(A, \lfloor skolem^{+}(A, x, K, \eta\,[\mu\beta.\Sigma\,[K\eta]\,/\beta])\rfloor)/x_K] \;\vdash^{\circ}\; \tilde{e} : \lfloor \tau \rfloor.$$

Since

$$\lfloor skolem^{\cdot}(A, x, K, \eta\,[\mu\beta.\Sigma\,[K\eta]\,/\beta])\rfloor$$
$$= skolem^{\circ}(A, x_K, \eta\,[\mu\beta.\Sigma\,[K\eta]\,/\beta])$$

we have

$$\lfloor A \rfloor\,[gen(A, skolem^{\circ}(A, x_K, \eta\,[\mu\beta.\Sigma\,[K\eta]\,/\beta]))/x_K] \;\vdash^{\circ}\; \tilde{e} : \lfloor \tau \rfloor.$$

We translate the other two premises and obtain $\lfloor A \rfloor(K) \geq \lfloor \mu\beta.\Sigma\,[K\eta] \rfloor$ and $\lfloor A \rfloor(x) \geq \lfloor \mu\beta.\Sigma\,[K\eta] \rfloor$. We further observe that $\lfloor FS_x(A) \rfloor = FS_{x_K}(\lfloor A \rfloor)$ and $\lfloor FS_x(\tau) \rfloor = FS_{x_K}(\lfloor \tau \rfloor)$, thus $FS_{x_K}(\lfloor A \rfloor) \cup FS_{x_K}(\lfloor \tau \rfloor) = \emptyset$.

Finally, we can apply the PAT° rule and conclude that

$$\lfloor A \rfloor \vdash^{\circ} \mathbf{let}\;\; K\, x_K = x \;\; \mathbf{in}\;\; \tilde{e} : \lfloor \tau \rfloor.$$

■

Lemma 4.7 $E[\![e]\!]\,\rho = E[\![\|e\|_{x, K_1\eta_1 + \ldots + K_n\eta_n}]\!]\,\rho$ for arbitrary ρ defined for x.

Proof: By definition of E, any subexpression of e is evaluated in an environment $\rho' \supseteq \rho$, whence $\rho'(x) = \rho(x)$. We identify two cases:

$\rho(x) \in \{K_i\} \times V$ for some i. Then, in the strict case only,

$$E[\![\mathbf{let}\;\; K_{j \neq i} x' = x \;\; \mathbf{in}\;\; e']\!]\,\rho' = \bot$$

and in both cases,

$$E[\![\mathbf{let}\;\; K_i\, x' = x \;\; \mathbf{in}\;\; e']\!]\,\rho' = E[\![e'\,[x_{K_i}/x']]\!]\,(\rho'\,[\mathsf{snd}(\rho(x))/x_{K_i}])$$

Consequently, in the strict case,

$$
\begin{aligned}
& E[\![e]\!]\,\rho \\
&= E[\![e \begin{bmatrix} e'[x_{K_i}/x'] \,/\, \textbf{let}\ \ K_i\ x' = x\ \ \textbf{in}\ \ e' \\ \textbf{fail} \,/\, \textbf{let}\ \ K_{j\neq i}\ x' = x\ \ \textbf{in}\ \ e' \end{bmatrix}]\!]\ \ (\rho\,[\mathsf{snd}(\rho(x))/x_{K_i}]) \\
&= E[\![\textbf{let}\ \ K_i\ x_{K_i} = x\ \ \textbf{in}\ \ e \begin{bmatrix} e'[x_{K_i}/x'] \,/\, \textbf{let}\ \ K_i\ x' = x\ \ \textbf{in}\ \ e' \\ \textbf{fail} \,/\, \textbf{let}\ \ K_{j\neq i}\ x' = x\ \ \textbf{in}\ \ e' \end{bmatrix}]\!]\,\rho \\
&= E[\![\|e\|_{x,\,K_1\eta_1 + \ldots + K_n\eta_n}]\!]\,\rho\,,
\end{aligned}
$$

since the **if** branch for i gets selected.

In the non-strict case for $i = 1$,

$$
\begin{aligned}
& E[\![e]\!]\,\rho \\
&= E[\![e\,[e'[x_K/x'] \,/\, \textbf{let}\ \ K\ x' = x\ \ \textbf{in}\ \ e']]\!]\ \ (\rho\,[\mathsf{snd}(\rho(x))/x_K]) \\
&= E[\![\textbf{let}\ \ K\ x_K = x\ \ \textbf{in}\ \ e\,[e'[x_K/x'] \,/\, \textbf{let}\ \ K\ x' = x\ \ \textbf{in}\ \ e']]\!]\,\rho \\
&= E[\![\|e\|_{x,\,K\eta}]\!]\,\rho\,,
\end{aligned}
$$

and for $i > 1$,

$$E[\![e]\!]\,\rho \;=\; E[\![\|e\|_{x,\,K_1\tau_1 + \ldots + K_n\tau_n}]\!]\,\rho\,.$$

$\rho(x) \notin \{K_i\} \times V$ for any i.

Then, in the strict case, $E[\![\textbf{let}\ \ K_i\ x' = x\ \ \textbf{in}\ \ e']\!]\,\rho' = \bot$, whence

$$
\begin{aligned}
& E[\![e]\!]\,\rho \\
&= E[\![e\,[\textbf{fail} \,/\, \textbf{let}\ \ K_i\ x' = x\ \ \textbf{in}\ \ e']]\!]\,\rho \\
&= E[\![\|e\|_{x,\,K_1\eta_1 + \ldots + K_n\eta_n}]\!]\,\rho\,,
\end{aligned}
$$

since the last **else** branch gets selected.

In the non-strict case for $i = 1$,

$$E[\![\textbf{let}\ \ K\ x' = x\ \ \textbf{in}\ \ e']\!]\,\rho'$$

$= E[\![e']\!]\ (\rho'[\bot/x'])$

$= E[\![e'[x_K/x']]\!]\ (\rho'[\bot/x_K])$.

Therefore,

$E[\![e]\!]\ \rho$

$= E[\![e[e'[x_K/x']/\mathbf{let}\ \ K\ x' = x\ \ \mathbf{in}\ \ e']]\!]\ (\rho[\bot/x_K])$

$= E[\![\|e\|_{x, K\eta}]\!]\ \rho$.

■

Theorem 4.8 [Preservation of semantics] If $A \vdash' e : \tau \Rightarrow \tilde{e}$, then $E[\![e]\!]\ \rho\ = E[\![\tilde{e}]\!]\ \rho$ for arbitrary ρ.

Proof: By structural induction on e.

Corollary 4.9 [Semantic soundness] If e is a closed term, $A \vdash' e : \tau$ and $\models_{\rho, \psi} A$ as defined previously, then $E[\![e]\!]\ \rho \in T[\![\tau]\!]\ \psi$.

Proof: Follows immediately from the two theorems, observing that $\lfloor \tau \rfloor = \tau$ and $\lfloor A \rfloor = A$, since neither τ nor A contain any κ's.

■

5 An Extension of Haskell with First-Class Abstract Types

This chapter introduces an extension of the functional language Haskell with existential types. Existential types combine well with the systematic overloading polymorphism provided by Haskell type classes. Briefly, we extend Haskell's **`data`** declaration in a similar way as the ML datatype declaration above. In Haskell, it is possible to specify what type class a (universally quantified) type variable belongs to. In our extension, we can do the same for existentially quantified type variables. This lets us use type classes as signatures of abstract data types; we can then construct heterogeneous aggregates over a given type class. A type reconstruction algorithm is given, and semantic soundness is shown by translating into an extension of the language from Chapter 3.

5.1 Introduction

Haskell [HPJW+92] uses type classes as a systematic approach to ad-hoc polymorphism, otherwise known as overloading. Type classes capture common sets of operations. A particular type may be an instance of a type class, and has an operation corresponding to each operation defined in the type class. Type classes may be arranged hierarchically.

In [WB89], Wadler and Blott called for a closer exploration of the relationship between type classes and abstract data types. After an initial exploration described in [LO91], we now present an extension of Haskell with datatypes whose component types may be existentially quantified.

In Haskell, an algebraic datatype declaration is of the form

$$\textbf{data}\ \ [c \Rightarrow] T\ a_1 \ldots a_n\ \ \textbf{=}\ \ K_1\ t_{11} \ldots t_{1k_1} \mid \ldots \mid K_m\ t_{m1} \ldots t_{mk_m}$$

It introduces a new type constructor T with value constructors $K_1, \ldots, K_m$. The optional context c specifies of which type classes the type variables $a_1, \ldots, a_n$ are instances. The constructors are used in two ways: as functions to construct values, and in patterns to decompose values already constructed. The types of the constructors are universally quantified over the type variables $a_1, \ldots, a_n$; no other type variable may appear free in the component types t_{ij}.

We describe an extension of Haskell analogous to the extension of ML described above. Type variables that appear free in the component types are interpreted as existentially quantified. In addition to the "global" context for the universally quantified parameters of the type constructor, we introduce "local" contexts for each value constructor. The local context specifies of which type classes the existentially quantified type variables in the component types are instances. The extended datatype declaration is of the form

$$\begin{aligned} \textbf{data}\ \ [c \Rightarrow] T\ a_1 \ldots a_n\ \ \textbf{=}\ \ & [c_1 \Rightarrow] K_1\ t_{11} \ldots t_{1k_1} \\ \mid\ \ & \ldots \\ \mid\ \ & [c_m \Rightarrow] K_m\ t_{m1} \ldots t_{mk_m} \end{aligned}$$

When constructing a value using a constructor with an existentially quantified component type, the existential type variables instantiate to the actual types of the corresponding function arguments, and we lose any information on the actual types. However, we know that these types are instances of the same type classes as the corresponding existential type variables. This

means that we have types whose identity is unknown but which support the operations specified by their type classes. Therefore we regard type classes as signatures of abstract types.

5.2 Some Motivating Examples

5.2.1 Minimum over a Heterogeneous List

This example is the extended Haskell version of the example given in Section 3.2.1. We first define a type class **Key** defining the operation **whatkey** needed to obtain an integer value from the value to be compared.

```
class Key a where
    whatkey :: a -> Int
```

We now define a datatype **KEY** with a single constructor **key**. The component type of **key** is the type variable **a**, which is existentially quantified and is required to be an instance of type class **Key**.

```
data KEY = (Key a) => key a
```

We further define several instances of **Key** along with their implementations of the function **whatkey**.

```
instance Key Int   where whatkey = id
instance Key Float where whatkey = round
instance Key [a]   where whatkey = length
instance Key Bool  where whatkey =
                           \x -> if x then 1 else 0
```

A heterogeneous list of values of type **KEY** could be defined as follows:

```
hetlist = [key 3,key [1,2,3,4],key 7,
           key True,key 12]
```

The **min** function finds the minimum over a list of **KEY**'s by decomposing the elements of the list and comparing their corresponding integer values obtained by applying **whatkey**.

```
min [x] = x
min ((key v1):xs) =
    case min xs of
        key v2 ->
            if whatkey v1 <= whatkey v2 then
                key v1
            else
                key v2
```

Then **min hetlist** evaluates to **key True**, as this is the element for which **whatkey** returns the smallest number.

5.2.2 Abstract Stack with Multiple Implementations

We also give the extended Haskell version of the stack example from Section 3.2.2. However, these stacks have a fixed element type, since Haskell type classes cannot be parameterized. An extension of Haskell with parameterized type classes is found in [CHO92]; it could in turn be extended with existential types, which would allow us to have polymorphic abstract stacks.

An integer stack is described by the following type class:

```
class Stack a where
    empty   :: a
    push    :: Int -> a -> a
    pop     :: a -> a
    top     :: a -> Int
    isempty :: a -> Bool
```

To achieve abstraction, we define the corresponding datatype of "encapsulated" stacks:

```
data STACK = (Stack a) => Stack a
```

We define two stack implementations, one based on a list of integers:

```
instance Stack [Int] where
    empty   = []
    push    = (:)
    pop     = tail
    top     = head
    isempty = null
```

and one based on an integer array:

```
maxIndex :: Int
maxIndex = 100

data FixedArray = Fixarr Int (Array Int Int)

instance Stack FixedArray where
    empty = Fixarr 0 (listArray(1,maxIndex)[])
    push a (Fixarr i s) =
        if i >= maxIndex then
            error "stack size exceeded"
        else
            Fixarr(i+1)(s // [(i+1) := a])
    pop(Fixarr i s) =
        if i <= 0 then
            error "stack empty"
        else
            Fixarr(i-1) s
    top(Fixarr i s) =
        if i <= 0 then
            error "stack empty"
        else
            s!i
    isempty(Fixarr i s) = i <= 0

arrayStack xs = Stack(Fixarr(length xs)
                     (listArray(1,maxIndex) xs)
```

As we saw in Section 3.2.2, it is convenient to define wrapper functions that apply the functions operating on instances of the type class **`Stack`** to an encapsulated value of type **`STACK`**; these "outer" wrappers open the encapsulated stack, apply the corresponding "inner" operations, and close the stack again. This provides dynamic dispatching of operations across different implementations of **`STACK`**. The wrapper **`wpush`** is defined as follows:

```
wpush a (Stack s) = Stack(push a s)
```

We can define the following list, which is a homogeneous list of two different implementations of **`STACK`**:

```
stackList = [Stack([1,2,3] :: [Int]),
             arrayStack([5,6,7] :: [Int])]
```

Using the wrapper **`wpush`** and the built-in function **`map`**, we can uniformly push an integer onto each element of the list:

```
map (wpush 8) stackList
```

5.3 Syntax

The formal treatment of our extension of Haskell builds on the article [NS91] by Nipkow and Snelting, who are the first to give an accurate treatment of type inference in Haskell. Our language is an extension of theirs with algebraic data types.

5.3.1 Language Syntax

Identifiers	x
Constructors	K
Type constructors	t
Expressions	e ::= **`()`** | **`true`** | **`false`** |
	x | (e_1, e_2) | $e\ e'$ | $\lambda x.e$ |

$$\texttt{let}\ x = e\ \texttt{in}\ e' \mid$$
$$K \mid \texttt{is}\ K \mid \texttt{let}\ K\ x = e\ \texttt{in}\ e'$$

Declarations d ::= $\texttt{data}\ t = \forall\alpha_{\gamma_1} \ldots \alpha_{\gamma_n}.\chi\ \texttt{in}\ e \mid$

$$\texttt{class}\ \gamma \leq \gamma_1, \ldots, \gamma_n\ \texttt{where}$$
$$x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k \mid$$
$$\texttt{inst}\ t : (\gamma_1, \ldots, \gamma_n)\,\gamma\ \texttt{where}$$
$$x_1 = e_1, \ldots, x_k = e_k$$

Programs p ::= $d_1 \ldots d_n e$

5.3.2 Type Syntax

Type variables α

Skolem functions κ

Type constructors t

Types τ ::= $unit \mid bool \mid \alpha_\gamma \mid \tau_1 \times \tau_2 \mid \tau \rightarrow \tau' \mid$
$\kappa \mid t(\tau_1, \ldots, \tau_n) \mid \chi$

Recursive types χ ::= $\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$ where $K_i \neq K_j$ for $i \neq j$

Existential types η ::= $\exists\alpha_\gamma.\eta \mid \tau$

Type schemes σ ::= $\forall\alpha_\gamma.\sigma \mid \eta \rightarrow \tau \mid \tau$

Assumptions a ::= $\sigma/x \mid \sigma/K$

Our type syntax includes recursive types χ and Skolem type constructors κ; the latter are used to type identifiers bound by a pattern-matching **let**

whose type is existentially quantified. Explicit existential types arise only as domain types of value constructors. Further, let $\Sigma[K\eta]$ stand for sum type contexts such as $K_1\eta_1 + \ldots + K_m\eta_m$, where $K_i = K$ and $\eta_i = \eta$ for some i. Our type syntax also includes explicit type constructors t; this makes it possible to extend the order-sorted signature with arities for user-defined type constructors.

5.4 Type Inference

5.4.1 Instantiation and Generalization of Type Schemes

$\forall\alpha_{\gamma_1}\ldots\alpha_{\gamma_n}.\tau \geq_C \tau'$ iff there are types $\tau_1, \ldots\tau_n$ of sorts $\gamma_1, \ldots, \gamma_n$, respectively, such that $\tau' = \tau\left[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}\right]$

$\exists\alpha_{\gamma_1}\ldots\alpha_{\gamma_n}.\tau \geq_C \tau'$ iff there are types $\tau_1, \ldots\tau_n$ of sorts $\gamma_1, \ldots, \gamma_n$, respectively, such that $\tau' = \tau\left[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}\right]$

In addition to *FV*, the set of free type variables in a type scheme or assumption set, we use *FS*, the set of those Skolem type constructors that occur in a type scheme or assumption set, and *FT*, the set of defined type constructors in a type scheme.

5.4.2 Inference Rules for Expressions

The first five typing rules are the same as in the system described in [NS91].

$$(\mathrm{VAR}^+) \quad \frac{A(x) \geq_C \tau}{(A, C) \vdash^+ x : \tau}$$

$$(\text{PAIR}^+)\quad \frac{(A, C) \vdash^+ e_1 : \tau_1 \qquad (A, C) \vdash^+ e_2 : \tau_2}{(A, C) \vdash^+ (e_1, e_2) : \tau_1 \times \tau_2}$$

$$(\text{APPL}^+)\quad \frac{(A, C) \vdash^+ e : \tau' \to \tau \qquad (A, C) \vdash^+ e' : \tau'}{(A, C) \vdash^+ e\ e' : \tau}$$

$$(\text{ABS}^+)\quad \frac{(A\,[\tau'/x], C) \vdash^+ e : \tau}{(A, C) \vdash^+ \lambda x.e : \tau' \to \tau}$$

$$(\text{LET}^+)\quad \frac{\begin{array}{c} FV(\tau) \setminus FV(A) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_n}\} \\ (A, C) \vdash^+ e : \tau \qquad (A\left[\forall\overline{\alpha_{\gamma_n}}.\tau/x\right], C) \vdash^+ e' : \tau' \end{array}}{(A, C) \vdash^+ \mathbf{let}\ \ x = e\ \ \mathbf{in}\ \ e' : \tau'}$$

The new rules CONS$^+$, TEST$^+$, and PAT$^+$ are used to type value constructors, **is** expressions, and pattern-matching **let** expressions, respectively.

$$(\text{CONS}^+)\quad \frac{A(K) \geq_C \eta \to t(\overline{\tau_n}) \qquad \eta \leq_C \tau}{(A, C) \vdash^+ K : \tau \to t(\overline{\tau_n})}$$

The CONS$^+$ rule observes the fact that existential quantification in argument position means universal quantification over the whole function type; this is expressed by the second premise.

$$(\text{TEST}^+)\quad \frac{A(K) \geq_C \eta \to t(\overline{\tau_n})}{(A, C) \vdash^+ \mathbf{is}\ \ K : t(\overline{\tau_n}) \to \mathit{bool}}$$

The TEST$^+$ rule ensures that **is** K is applied only to arguments whose type is the same as the result type of constructor K.

$$(\mathrm{PAT}^{+}) \quad \frac{\begin{array}{c} A(K) \geq_C (\exists \overline{\beta_{\delta_k}}.\tau) \rightarrow t(\overline{\tau_n}) \qquad (A, C) \vdash^{+} e : t(\overline{\tau_n}) \\ \{\kappa_1, \ldots, \kappa_k\} \cap (FS(\tau') \cup FS(A)) = \varnothing \\ \begin{pmatrix} A\left[\tau\left[\kappa_i / \beta_{\delta_i} \middle| i = 1 \ldots k\right] / x\right] \\ C\left[\kappa_i : \delta_i \middle| i = 1 \ldots k\right] \end{pmatrix} \vdash^{+} e' : \tau' \end{array}}{(A, C) \vdash^{+} \textbf{let}\ \ K\ x = e\ \ \textbf{in}\ \ e' : \tau'}$$

The last rule, PAT^{+}, governs the typing of pattern-matching **let** expressions. It requires that the expression e be of the same type as the result type of the constructor K. The body e' is typed under the assumption set extended with an assumption about the bound identifier x. The new Skolem type constructors must not appear in A; this ensures that they do not appear in the type of any identifier free in e' other than x. It is also guaranteed that the Skolem type constructors do not appear in the result type τ'. The Skolem type constructors $\kappa_1, \ldots, \kappa_k$ replace the existentially quantified type variables of sorts $\delta_1, \ldots, \delta_k$. Thus the body of the **let** expression is typed under the extended signature containing appropriate arities for $\kappa_1, \ldots, \kappa_k$. The pattern-matching **let** expression is monomorphic in the sense that the type of the bound variable x is not generalized. This restriction is sufficient to guarantee a type-preserving translation into a target language (see Section 5.6.5). The **case** expression in Haskell syntax corresponds to a nested **if** with an **is** and a pattern-matching **let** expression for each case.

5.4.3 Inference Rules for Declarations and Programs

The rules for class and instance declarations, and programs are the same as in [NS91]. We add the DATA^{+} rule to elaborate a recursive datatype declaration.

$$(\mathrm{CLASS}^+)\ \frac{FT(\tau_1) \cup \ldots \cup FT(\tau_k) \subseteq \mathrm{Dom}(C)}{\begin{array}{l}(A, C)\ \vdash^+ \mathbf{class}\ \ \gamma \leq \gamma_1, \ldots, \gamma_n\ \ \mathbf{where} \\ \qquad\qquad x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k : \\ (A\,[\forall\alpha_\gamma.\tau_i/x_i \mid i = 1\ldots k]\,, C\,[\gamma \leq \gamma_1, \ldots, \gamma_n]\,)\end{array}}$$

$$(\mathrm{INST}^+)\ \frac{\begin{array}{c}t \in \mathrm{Dom}(C) \qquad A(x_i) = \forall\alpha_\gamma.\tau_i \\ (A, C)\ \vdash^+ e_i : \tau_i\left[t(\overline{\alpha_{\gamma_n}})/\alpha_\gamma\right] \qquad i = 1\ldots k\end{array}}{\begin{array}{c}(A, C)\ \vdash^+ \mathbf{inst}\ \ t : (\overline{\gamma_n})\,\gamma\ \ \mathbf{where}\ \ x_1 = e_1, \ldots, x_k = e_k : \\ (A, C\,[t : (\overline{\gamma_n})\,\gamma]\,)\end{array}}$$

$$(\mathrm{PROG}^+)\ \frac{\begin{array}{c}(A_{i-1}, C_{i-1})\ \vdash^+ d_i : (A_i, C_i) \qquad i = 1\ldots n \\ (A_n, C_n)\ \vdash^+ e : \tau\end{array}}{(A_0, C_0)\ \vdash^+ d_1 \ldots d_n e : \tau}$$

$$(\mathrm{DATA}^+)\ \frac{\begin{array}{c}\sigma = \forall\overline{\alpha_{\gamma_n}}.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m \\ FT(\sigma) \subseteq \mathrm{Dom}(C) \qquad t \notin \mathrm{Dom}(C)\end{array}}{\begin{array}{l}(A, C)\ \vdash^+ \mathbf{data}\ \ t = \sigma : \\ \left(\begin{array}{c}A\left[\forall\overline{\alpha_{\gamma_n}}.\eta_i\left[t(\overline{\alpha_{\gamma_n}})/\beta\right] \to t(\overline{\alpha_{\gamma_n}})/K_i \,\middle|\, i = 1\ldots m\right] \\ C\,[t : (\overline{\gamma_n})\,\Omega]\end{array}\right)\end{array}}$$

The DATA$^+$ rule adds assumptions about the value constructors to the assumption set, and extends the signature with an appropriate arity for the new type constructor. Whereas recursive datatypes were anonymous in the two preceding chapters, they are now represented by named type constructors. This is necessary since the order-sorted signature C may contain arity declarations for user-defined type constructors. We avoid using a separate type

constructor environment; therefore, in an assumption σ/K about a value constructor, σ now is the type scheme for K when regarded as a function, as opposed to the type scheme describing the entire recursive datatype K belongs to.

5.4.4 Relation to the Haskell Type Inference System

Theorem 5.1 [Conservative extension] Let Mini-Haskell' be an extension of Mini-Haskell with recursive datatypes and a monomorphic pattern-matching `let` expression, but without existential quantification. Then, for any Mini-Haskell' program p, $(A, C) \vdash^{+} p : \tau$ iff $(A, C) \vdash_{\mathrm{MH'}} p : \tau$.

Proof: By structural induction on p.

Corollary 5.2 [Conservative extension] Our type system is a conservative extension of the Mini-Haskell type system described in [NS91], in the following sense: For any Mini-Haskell program p, $(A, C) \vdash^{+} p : \tau$ iff $(A, C) \vdash_{\mathrm{MH}} p : \tau$.

Proof: Follows immediately from Theorem 5.1.

5.5 Type Reconstruction

The type reconstruction algorithm is a translation from the deterministic typing rules, using order-sorted unification [SS85][MGS89] instead of standard unification.

5.5.1 Unitary Signatures for Principal Types

The article [NS91] describes several conditions necessary to guarantee unitary signatures, which are sufficient to guarantee principal types. First, to make a signature C regular and downward compete, we perform the following two steps to obtain a new signature C_R:

- For any two incomparable classes $\gamma_1, \gamma_2 \in \mathrm{Dom}(C)$, we introduce a new class declaration **class** $\gamma \leq \gamma_1, \gamma_2$ with an empty **where** part combining the operations of γ_1 and γ_2.
- Then, for each type constructor with instance declarations

  inst $t : (\overline{\gamma_{1n}})\,\gamma_1$ **where** ...

  inst $t : (\overline{\gamma_{2n}})\,\gamma_2$ **where** ...

  introduce another instance declaration of the form

  inst $t : (\gamma_{11} \wedge \gamma_{21}, \ldots, \gamma_{1n} \wedge \gamma_{2n})\ (\gamma_1 \wedge \gamma_2)$

  where $\gamma \wedge \delta$ is simply the additionally declared class if γ and δ are incomparable, or otherwise the lower one in the class hierarchy.

Note that Haskell uses multiple class assertions for type variables to express this conjunction of classes.

Since regular signatures alone do not guarantee the existence of principal types, we impose the following two conditions on C_R, which are also present in Haskell:

- Injectivity: A type constructor may not be declared as an instance of a particular class more than once in the same scope
- Subsort reflection: If $\gamma_1, \ldots, \gamma_m$ are the immediate superclasses of δ, a declaration **inst** $t : (\overline{\delta_n})\,\delta$ **where** ... must be preceded by declarations **inst** $t : (\overline{\gamma_n^i})\,\gamma_i$ **where** ... such that δ_j is a subclass of γ_j^i for al $i = 1 \ldots m$ and $j = 1 \ldots n$.

As discussed in [NS91], a Haskell signature that satisfies these conditions is unitary.

5.5.2 Auxiliary Functions

In our algorithm, we need to instantiate universally quantified types and generalize existentially quantified types. Both are handled in the same way.

$inst_{\forall}(\forall\alpha_{\gamma_1}\ldots\alpha_{\gamma_n}.\tau) = \tau\left[\beta_{\gamma_1}/\alpha_{\gamma_1}, \ldots, \beta_{\gamma_n}/\alpha_{\gamma_n}\right]$ where $\beta_{\gamma_n}, \ldots, \beta_{\gamma_n}$ are fresh type variables

$inst_{\exists}(\exists\alpha_{\gamma_1}\ldots\alpha_{\gamma_n}.\tau) = \tau\left[\beta_{\gamma_1}/\alpha_{\gamma_1}, \ldots, \beta_{\gamma_n}/\alpha_{\gamma_n}\right]$ where $\beta_{\gamma_n}, \ldots, \beta_{\gamma_n}$ are fresh type variables

$osu_C(\tau, \tau')$ the most general unifier of τ and τ' under order-sorted signature C

5.5.3 Algorithm

Our type reconstruction function takes an assumption set, an order-sorted signature, and an expression, and it returns a substitution and a type expression. There is one case for each typing rule.

$TE(A, C, x) =$

$(Id, inst_{\forall}(A(x)))$

$TE(A, C, (e_1, e_2)) =$

let $(S_1, \tau_1) = TE(A, C, e_1)$

$(S_2, \tau_2) = TE(S_1A, C, e_2)$

in $(S_2S_1, S_2\tau_1 \times \tau_2)$

$TE(A, C, ee') =$

let $(S, \tau) = TE(A, C, e)$
$(S', \tau') = TE(SA, C, e')$
β be a fresh type variable
$U = osu_C(S'\tau, \tau' \to \beta)$
in $(US'S, U\beta)$

$TE(A, C, \lambda x.e) =$

let β be a fresh type variable
$(S, \tau) = TE(A[\beta/x], C, e)$
in
$(S, S\beta \to \tau)$

$TE(A, C,$ **let** $x = e$ **in** $e') =$

let $(S, \tau) = TE(A, C, e)$
$(S', \tau') = TE(SA[gen(SA, \tau)/x], C, e')$
in
$(S'S, \tau')$

$TE(A, C, K) =$

let $\eta \to \tau = inst_\forall(A(K))$
in $(Id, (inst_\exists(\eta)) \to \tau)$

$TE(A, C,$ **is** $K) =$

let $\eta \to \tau = inst_\forall(A(K))$
in $(Id, \tau \to bool)$

$TE(A, C,$ **let** $K\ x = e$ **in** $e') =$

let $(S, \tau) = TE(A, C, e)$

$(\exists\overline{\beta_{\delta_k}}.\tau_0) \rightarrow t(\overline{\alpha_{\gamma_n}}) = inst_{\forall}(A(K))$

$U = osu_C(\tau, t(\overline{\alpha_{\gamma_n}}))$

$\kappa_1, \ldots, \kappa_k$ fresh type constructors

$\tau_\kappa = (U\tau_0)\left[\kappa_i/\beta_{\delta_i} \middle| i = 1 \ldots k\right]$

$C' = C\,[\kappa_i : \delta_i \mid i = 1 \ldots k]$

$(S', \tau') = TE(USA\,[\tau_\kappa/x], C', e')$

in

if $\{\kappa_1, \ldots, \kappa_k\} \cap (FS(S'USA) \cup FS(\tau')) = \varnothing$ **then**

$(S'US, \tau')$

$TD(A, C,$ **data** $t = \sigma) =$

let $\forall\alpha_{\gamma_1} \ldots \alpha_{\gamma_n}.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m = \sigma$ **in**

if $FV(\sigma) = \varnothing \wedge$

$t \notin \mathrm{Dom}(C) \wedge FT(\sigma) \subseteq \mathrm{Dom}(C)$

then

$$\begin{pmatrix} A\left[\forall\overline{\alpha_{\gamma_n}}.\eta_i\left[t(\overline{\alpha_{\gamma_n}})/\beta\right] \rightarrow t(\overline{\alpha_{\gamma_n}})/K_i \middle| i = 1 \ldots m\right] \\ C\,[t : (\overline{\gamma_n})\,\Omega] \end{pmatrix}$$

$TD(A, C,$ **class** $\gamma \leq \gamma_1, \ldots, \gamma_n$ **where** $x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k) =$

$(A\,[\forall\alpha_\gamma.\tau_i/x_i \mid i = 1 \ldots k], C\,[\gamma \leq \gamma_1, \ldots, \gamma_n])$

$TD(A, C,$ **inst** $t : (\overline{\gamma_n})\,\gamma$ **where** $x_1 = e_1, \ldots, x_k = e_k) =$

$(A, C\,[t : (\overline{\gamma_n})\,\gamma])$

$TD(A, C, d_1 \ldots d_n) =$

let $(A', C') = TD(A, C, d_1)$ **in**
$TD(A, C', d_2 \ldots d_n)$

$TP(A, C, d_1 \ldots d_n e) =$

let $(A', C') = TD(A, C, d_1 \ldots d_n)$ **in**
$TE(A', C', e)$

5.5.4 Syntactic Soundness and Completeness of Type Reconstruction

Lemma 5.3 [Stability of $\vdash^+$] If $(A, C) \vdash^+ e : \tau$ and S is a substitution, then $(SA, C) \vdash^+ e : S\tau$ also holds. Moreover, if there is a proof tree for $(A, C) \vdash^+ e : \tau$ of height n, then there is also a proof tree for $(SA, C) \vdash^+ e : S\tau$ of height less or equal to n.

Theorem 5.4 [Syntactic soundness] If $TC(A, C, e) = (S, \tau)$, then $(SA, C) \vdash^+ e : \tau$.

Definition 5.1 [Principal type] τ is a principal type of expression e under assumption set A and signature C if $(A, C) \vdash^+ e : \tau$ and whenever $(A, C) \vdash^+ e : \tau'$ then there is a substitution S such that $S\tau = \tau'$.

Theorem 5.5 [Syntactic completeness] If $(\hat{S}A, C) \vdash e : \hat{\tau}$, then $TC(A, C, e) = (S, \tau)$ and there is a substitution R such that $\hat{S}A = RSA$ and $\hat{\tau} = R\tau$.

Proof: We extend Nipkow's recent work on type classes and order-sorted unification and extend it with existential types.

We assume that the signature built from the global **`class`** and **`inst`** declarations is unitary. Clearly, the extended signature used to type the body of a pattern-matching **`let`** expression is also unitary, since the Skolem type constructors κ_i are unique, and each κ_i appears in only one arity declaration. The latter trivially guarantees injectivity and subsort reflection.

■

5.6 Semantics

As in [NS91] [WB89], we give an inference-guided translation to the target language, an enhanced version of our extension of ML with existential types described in Chapter 3. Type classes and instances are replaced by *(method) dictionaries*, which contain all the operations associated with a particular instance of a type class. The translation rules are of the form $(A, C) \vdash^{+} e : \tau \Rightarrow \tilde{e}$ and mean "in the context (A, C), e is assigned type τ and translates to $\tilde{e}$."

5.6.1 Target Language

Our extension of Mini-Haskell is translated into an extended version of the language presented in Chapter 3. As a generalization of pair types, the language contains all n-ary product types $\alpha_1 \times \ldots \times \alpha_n$ with expressions $(e_1, \ldots, e_n)$ and projection functions π_i^n of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_i$. The PAIR rule is superseded by the TUPLE rule:

$$\text{(TUPLE)}\ \frac{A \vdash e_1 : \tau_1 \qquad \ldots \qquad A \vdash e_n : \tau_n}{A \vdash (e_1, \ldots, e_n) : \tau_1 \times \ldots \times \tau_n}$$

Semantically, expressions of the form

`let` $K(x_1, \ldots, x_n) = e$ **`in`** e'

are regarded as short forms for nested **let** expressions of the form

$$\begin{array}{l}\textbf{let}\ \ K\ z = e\ \ \textbf{in}\\ \quad\textbf{let}\ \ x_1 = \pi_1^n z, \ldots, x_n = \pi_n^n z\ \ \textbf{in}\ \ e'\end{array}$$

and are typed by following PAT'' rule:

$$\text{(PAT'')}\quad \frac{\begin{array}{c}A \vdash e : \mu\beta.\Sigma\,[K\eta] \qquad \eta = \exists\beta_1 \ldots \beta_k.\tau_1 \times \ldots \times \tau_n\\ \tau'_1 \times \ldots \times \tau'_n = \mathit{skolem}(A, \eta) \qquad \mathit{FS}(\tau') \subseteq \mathit{FS}(A)\\ \sigma_1 = \mathit{gen}(A, \tau'_1) \qquad \ldots \qquad \sigma_n = \mathit{gen}(A, \tau'_n)\\ A\,[\sigma_1/x_1, \ldots, \sigma_n/x_n] \vdash^+ e' : \tau'\end{array}}{A \vdash \textbf{let}\ \ K\,(x_1, \ldots, x_n) = e\ \ \textbf{in}\ \ e' : \tau'}$$

This rule is semantically sound, since the translation of the short form to the full form is type-preserving: an application of the PAT'' rule is replaced by an application of the PAT rule followed by n successive applications of the LET rule, using appropriate typings for the tuple projections.

5.6.2 Dictionaries and Translation of Types

We call the translated types "ML-types" to distinguish them from the original ones. ML-types introduce a method dictionary for each sorted type variable in the original type; each sorted type variable is then replaced by an ordinary type variable.

A class declaration

$$(A, C) \vdash^+ \textbf{class}\ \ \gamma \leq \gamma_1, \ldots, \gamma_n\ \ \textbf{where}\ \ x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k$$

introduces a new ML-type for method dictionaries of this class,

$$\gamma(\alpha) = \tau_1\,[\alpha/\alpha_\gamma] \times \ldots \times \tau_k\,[\alpha/\alpha_\gamma] \times \gamma_1(\alpha) \times \ldots \times \gamma_n(\alpha)$$

where the type parameter α stands for the type of the instance. The first k components of $\gamma(\alpha)$ are the operations corresponding to class γ. The next n components are the dictionaries for all immediate superclasses $\gamma_1, \ldots, \gamma_n$ of γ. Note that the dictionary $\Omega(\alpha)$ for the top of the class hierarchy is the empty product type.

Instead of defining $\gamma(\alpha)$ directly, the dictionary type is defined in terms of access functions $x_1, \ldots, x_k$ to extract operations from a dictionary, and access functions $\gamma_{1_\gamma}, \ldots, \gamma_{n_\gamma}$ to extract the dictionaries for the immediate superclasses from a dictionary.

Coercion functions are needed to convert a dictionary α_γ of a class γ into a dictionary of a superclass of γ; they are defined the same way as in [NS91]:

$$cast_C(\alpha_{\gamma'}, \gamma') = \begin{cases} \alpha_\gamma & \text{if } \gamma = \gamma' \\ (\gamma'_\delta \; cast_C(\alpha_{\gamma'}, \delta)) & \text{if } \gamma \leq \delta \wedge \gamma' \in super_C(\delta) \end{cases}$$

If there is more than one path from γ to γ' with respect to $\geq_C$, $cast_C$ chooses an arbitrary fixed path. The immediate superclasses of a class γ are defined as:

$$super_C(\gamma) = \{\gamma' \mid \gamma < \gamma' \wedge \neg\exists\delta.\, \gamma < \delta < \gamma'\}$$

The method dictionary for an instance τ of a class γ within the signature C is defined as

$$dict_C(\alpha_{\gamma'}, \gamma) = cast_C(\alpha_{\gamma'}, \gamma)$$

$$dict_C(t(\tau_1, \ldots, \tau_n), \gamma) = \gamma_t\,(dict_C(\tau_1, \gamma_1)) \ldots (dict_C(\tau_n, \gamma_n)) \text{ if according to } C,\ t(\tau_1, \ldots, \tau_n) \text{ is sort-correct with resulting sort } \gamma$$

Note that on the left hand side, $\alpha_{\gamma'}$ is a type variable, and on the right hand side, an identifier that stands for a dictionary of class γ'. As we will see below, the function γ_t is the dictionary defined for the type constructor *t* in the translation of the corresponding instance declaration. Its arguments are the dictionaries for the actual type parameters in an application of *t*.

We define the translation function for type schemes as follows:

$$ML(\forall\overline{\alpha_{\gamma_n}}.\tau) = \forall\overline{\alpha_n}.\gamma_1(\alpha_1) \to \ldots \to \gamma_n(\alpha_n) \to \tau\left[\alpha_1/\alpha_{\gamma_1}, \ldots, \alpha_n/\alpha_{\gamma_n}\right]$$

where each α_{γ_i} uniquely maps to an α_i. For existential component types of user-defined datatypes, the corresponding ML-types need to include the dictionaries for the existentially quantified type variables. This reflects the way operations on existential types are explicitly included in the datatype components in Chapter 3.

$$ML(\exists\overline{\beta_{\delta_k}}.\tau) = \exists\overline{\beta_k}.\tau\left[\beta_1/\beta_{\delta_1}, \ldots, \beta_k/\beta_{\delta_k}\right] \times \delta_1(\beta_1) \times \ldots \times \delta_k(\beta_k)$$

The resulting translation function for user-defined recursive type schemes is

$$ML(\forall\overline{\alpha_{\gamma_n}}.\mu\beta.\Sigma\,[K\eta]) = \forall\overline{\alpha_n}.\mu\beta.\Sigma\,[K\ ML(\eta)]\left[\alpha_1/\alpha_{\gamma_1}, \ldots, \alpha_n/\alpha_{\gamma_n}\right]$$

Note that the dictionaries for the universally quantified type variables are *not* included in the component types, as they are determined by the actual instance types substituted for the type variables. Since user-defined datatypes are anonymous in the target language, the translation function for the type of a value constructor *K* is given by the entire recursive type scheme to which *K* belongs:

$$ML(A(K)) = ML(\forall\overline{\alpha_{\gamma_n}}.\mu\beta.\Sigma\,[K\eta]) \text{ where } A(K) = \forall\overline{\alpha_{\gamma_n}}.\eta \to \tau$$

The function *ML* extends to assumption sets as follows:

$$ML(A) \;=\; \{ML(A(x))/x \mid x \in \mathrm{Dom}A\}$$

5.6.3 Translation Rules for Declarations and Programs

The first three translation rules concern class declarations, instance declarations, and programs. They are the same as in [NS91]. Declarations translate to **let** expressions without bodies.

$$(\mathrm{CLASS}^+)\ \frac{FT(\tau_1) \cup \ldots \cup FT(\tau_k) \subseteq \mathrm{Dom}(C)}{\begin{array}{l}(A, C) \vdash^+ \textbf{class}\ \ \gamma \le \gamma_1, \ldots, \gamma_n\ \ \textbf{where} \\ \qquad\qquad x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k : \\ \quad (A\,[\forall\alpha_\gamma.\tau_i/x_i \mid i = 1\ldots k]\,, C\,[\gamma \le \gamma_1, \ldots, \gamma_n]\,) \Rightarrow \\ \textbf{let}\ \ x_1 = \pi_1^{k+n}, \ldots, x_k = \pi_k^{k+n}, \gamma_{1_\gamma} = \pi_{k+1}^{k+n}, \ldots, \gamma_{n_\gamma} = \pi_{k+n}^{k+n}\end{array}}$$

$$(\mathrm{INST}^+)\ \frac{\begin{array}{c}super_C(\gamma) = \{\gamma^1, \ldots, \gamma^s\} \\ t \in \mathrm{Dom}(C) \qquad A(x_i) = \forall\alpha_\gamma.\tau_i \\ (A, C) \vdash^+ e_i : \tau_i\left[t(\overline{\alpha_{\gamma_n}})/\alpha_\gamma\right] \Rightarrow \tilde{e}_i \qquad i = 1\ldots k\end{array}}{\begin{array}{l}(A, C) \vdash^+ \textbf{inst}\ \ t : (\overline{\gamma_n})\,\gamma\ \ \textbf{where}\ \ x_1 = e_1, \ldots, x_k = e_k : \\ \qquad\qquad (A, C\,[t : (\overline{\gamma_n})\,\gamma]\,) \Rightarrow \\ \quad \textbf{let}\ \ \gamma_t = \lambda\overline{\alpha_{\gamma_n}}.(\tilde{e}_1, \ldots, \tilde{e}_k, \\ \qquad\qquad (\gamma_t^1 cast_C(\alpha_{\gamma_1}, \gamma_1^1)\ldots cast_C(\alpha_{\gamma_n}, \gamma_n^1))\,, \\ \qquad\qquad \ldots \\ \qquad\qquad (\gamma_t^s cast_C(\alpha_{\gamma_1}, \gamma_1^s)\ldots cast_C(\alpha_{\gamma_n}, \gamma_n^s))\,)\end{array}}$$

$$(\mathrm{PROG}^+)\ \frac{\begin{array}{c}(A_{i-1}, C_{i-1}) \vdash^+ d_i : (A_i, C_i) \Rightarrow \tilde{d}_i \qquad i = 1\ldots n \\ (A_n, C_n) \vdash^+ e : \tau \Rightarrow \tilde{e}\end{array}}{(A_0, C_0) \vdash^+ d_1\ldots d_n e : \tau \Rightarrow \tilde{d}_1\ \ \textbf{in}\ \ \ldots\ \ \textbf{in}\ \ \tilde{d}_n\ \ \textbf{in}\ \ \tilde{e}}$$

$$\text{(DATA}^{+}\text{)}\quad \frac{\begin{array}{c}\sigma = \forall\overline{\alpha_{\gamma_n}}.\mu\beta.K_1\exists\overline{\beta_{\delta_{1k_1}}}.\tau_1 + \ldots + K_m\exists\overline{\beta_{\delta_{mk_m}}}.\tau_m \\ FT(\sigma) \subseteq \mathrm{Dom}(C) \qquad t \notin \mathrm{Dom}(C)\end{array}}{\begin{array}{l}(A, C) \vdash^{+} \textbf{data}\ \ t = \sigma : \\ \left(\begin{array}{c} A\left[\forall\overline{\alpha_{\gamma_n}}.(\exists\overline{\beta_{\delta_{ik_i}}}.\tau_1\left[t(\overline{\alpha_{\gamma_n}})/\beta\right]) \rightarrow t(\overline{\alpha_{\gamma_n}})/K_i \,\middle|\, i = 1\ldots m\right] \\ C\,[t : (\overline{\gamma_n})\,\Omega]\end{array}\right) \\ \Rightarrow \textbf{data}\ \ \forall\overline{\alpha_n}.\mu\beta. \\ \qquad K_1\exists\overline{\beta_{1k_1}}.\tau_1\left[\alpha_i/\alpha_{\gamma_i}, \beta_{1j}/\beta_{\delta_{1j}}\right] \times \\ \qquad\qquad \delta_{11}(\beta_{11}) \times \ldots \times \delta_{1k_1}(\beta_{1k_1}) \\ \qquad + \ldots + \\ \qquad K_m\exists\overline{\beta_{mk_m}}.\tau_m\left[\alpha_i/\alpha_{\gamma_i}, \beta_{mj}/\beta_{\delta_{mj}}\right] \times \\ \qquad\qquad \delta_{m1}(\beta_{m1}) \times \ldots \times \delta_{mk_m}(\beta_{mk_m})\end{array}}$$

The DATA^{+} rule translates a **data** declaration with order-sorted type variables to a **data** declaration in the target language. The component types of the translated datatype consist of the original component types together with the dictionaries for the existentially quantified type variables. This is reflected in the CONS^{+} and PAT^{+} rules below.

5.6.4 Translation Rules for Expressions

The first five translation rules are identical to the ones in [NS91].

$$\text{(VAR}^{+}\text{)}\quad \frac{A(x) = \forall\overline{\alpha_{\gamma_n}}.\tau}{\begin{array}{l}(A, C) \vdash^{+} x : \tau\left[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}\right] \Rightarrow \\ (x\ \ dict_C(\tau_1, \gamma_1)\ldots dict_C(\tau_n, \gamma_n))\end{array}}$$

$$(\text{PAIR}^+)\quad \frac{(A, C) \vdash^+ e_1 : \tau_1 \Rightarrow \tilde{e}_1 \qquad (A, C) \vdash^+ e_2 : \tau_2 \Rightarrow \tilde{e}_2}{(A, C) \vdash^+ (e_1, e_2) : \tau_1 \times \tau_2 \Rightarrow (\tilde{e}_1, \tilde{e}_2)}$$

$$(\text{APPL}^+)\quad \frac{(A, C) \vdash^+ e : \tau' \to \tau \Rightarrow \tilde{e} \qquad (A, C) \vdash^+ e' : \tau' \Rightarrow \tilde{e}'}{(A, C) \vdash^+ e\ e' : \tau \Rightarrow \tilde{e}\ \tilde{e}'}$$

$$(\text{ABS}^+)\quad \frac{(A\,[\tau'/x], C) \vdash^+ e : \tau \Rightarrow \tilde{e}}{(A, C) \vdash^+ \lambda x.e : \tau' \to \tau \Rightarrow \lambda x.\tilde{e}}$$

$$(\text{LET}^+)\quad \frac{\begin{array}{c}(A, C) \vdash^+ e : \tau \Rightarrow \tilde{e} \qquad FV(\tau) \setminus FV(A) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_n}\} \\ (A\left[\forall\overline{\alpha_{\gamma_n}}.\tau/x\right], C) \vdash^+ e' : \tau' \Rightarrow \tilde{e}'\end{array}}{\begin{array}{c}(A, C) \vdash^+ \textbf{let}\ \ x = e\ \ \textbf{in}\ \ e' : \tau' \Rightarrow \\ \textbf{let}\ \ x = \lambda\overline{\alpha_{\gamma_n}}.\tilde{e}\ \ \textbf{in}\ \ \tilde{e}'\end{array}}$$

In the LET$^+$ rule, the translation $\tilde{e}$ of the expression to be bound to x may contain free dictionary variables corresponding to the generic type variables in τ; λ-bindings for those dictionaries need to be provided.

New translation rules are added for value constructors, **is** expressions, and pattern-matching **let** expressions.

$$(\text{CONS}^+)\quad \frac{A(K) = \forall\overline{\alpha_{\gamma_n}}.\ (\exists\overline{\beta_{\delta_k}}.\tau) \to t(\overline{\alpha_{\gamma_n}})}{\begin{array}{c}(A, C) \vdash^+ K :\ (\tau \to t(\overline{\alpha_{\gamma_n}}))\left[\tau_i/\alpha_{\gamma_i}, \hat{\tau}_j/\beta_{\delta_j}\right] \Rightarrow \\ (\lambda x.K\,(x, \mathit{dict}_C(\hat{\tau}_1, \delta_1), \ldots, \mathit{dict}_C(\hat{\tau}_k, \delta_k))\,)\end{array}}$$

The translation of a value constructor is again a value constructor; this translated constructor packs dictionaries for the types substituted for the existentially quantified type variables together with the value.

$$(\text{TEST}^+)\quad \frac{A(K) \geq_C \eta \to t(\overline{\tau_n})}{(A, C) \vdash^+ \textbf{is}\ \ K : t(\overline{\tau_n}) \to \mathit{bool} \Rightarrow \textbf{is}\ \ K}$$

The **is** K expression is needed to examine the constructor tag and translates to itself.

$$(\text{PAT}^+)\quad \frac{\begin{array}{c} A(K) \geq_C (\exists\overline{\beta_{\delta_k}}.\tau) \to t(\overline{\tau_n}) \\ (A, C) \vdash^+ e : t(\overline{\tau_n}) \Rightarrow \tilde{e} \\ \{\kappa_1, \ldots, \kappa_k\} \cap (FS(\tau') \cup FS(A)) = \varnothing \\ \begin{pmatrix} A\left[\tau\left[\kappa_i/\beta_{\delta_i} \middle| i = 1\ldots k\right]/x\right] \\ C\,[\kappa_i : \delta_i \,|\, i = 1\ldots k] \end{pmatrix} \vdash^+ e' : \tau' \Rightarrow \tilde{e}' \end{array}}{\begin{array}{c}(A, C) \vdash^+ \textbf{let}\ \ K\ x = e\ \ \textbf{in}\ \ e' : \tau' \Rightarrow \\ \textbf{let}\ \ K\,(x, \delta_{\kappa_1}, \ldots, \delta_{\kappa_k}) = \tilde{e}\ \ \textbf{in}\ \ \tilde{e}'\end{array}}$$

To translate a pattern-matching **let** expression, we need to look at the way value constructors are translated. We need to provide a binding for the original bound variable x, which corresponds to the first component of the encapsulated value; we further need to retrieve the dictionaries for the Skolem type constructors from the remaining k components of the encapsulated value and bind them to variables $\delta_{\kappa_1}, \ldots, \delta_{\kappa_k}$. Any of these bound variables may occur in $\tilde{e}'$, the translation of the body of the **let** expression.

Since $\tilde{e}$, the translation of the expression to be bound, is used monomorphically, no λ-bindings for potentially free dictionary variables need to be provided.

5.6.5 Properties of the Translation

As in [NS91], the correctness of our translation scheme depends on the condition that each instance declaration lists exactly the same operations $x_1, \ldots, x_n$ as the corresponding class declaration, and in the same order. This scheme prohibits redefinition of operations listed in superclasses.

Furthermore, when translating a program $d_1 \ldots d_n e$, we require the signature resulting from elaborating $d_1 \ldots d_n$ to satisfy the injectivity and subsort reflection conditions stated in Section 5.5.1.

The next lemma says that the translation can only *introduce* free dictionary variables in an expression and is needed in the main theorem.

Lemma 5.6 [Free variables] If $(A, C) \vdash^{+} e : \tau \Rightarrow \tilde{e}$, then $FV(\tilde{e}) \supseteq FV(e)$, and $FV(\tilde{e}) \setminus FV(e)$ contains only dictionary variables.

Proof: By structural induction on e. Free dictionary variables are explicitly generated in the VAR^{+} and CONS^{+} rules. The variables that are bound in the LET^{+} rule are also dictionary variables and, by definition, are not free in the original expression.

Lemma 5.7 [Free variables] If $(A, C) \vdash^{+} e : \tau \Rightarrow \tilde{e}$ and $(A', C') \vdash^{+} e' : \tau \Rightarrow \tilde{e'}$, then

$$(FV(\tilde{e}) \cup FV(\tilde{e'})) \setminus (FV(e) \cup FV(e')) = (FV(\tilde{e}) \setminus FV(e)) \cup (FV(\tilde{e'}) \setminus FV(e')) \ .$$

Proof: Using Lemma 5.6 and the fact that $FV(e)$ and $FV(e')$ do not contain any dictionary variables.

Lemma 5.8 [Types of dictionaries] Let $d_1 \ldots d_n e$ be a program with translation $\tilde{d}_1$ **in** $\ldots$ **in** $\tilde{d}_n$ **in** $\tilde{e}$. Let C be the unitary signature obtained by elaborating $d_1 \ldots d_n$, and A a superset of the assumption set obtained by elaborating $\tilde{d}_1$ **in** $\ldots$ **in** $\tilde{d}_n$. The following type judgments hold for occurrences of *cast* and *dict* in $\tilde{e}$:

$$A\,[\gamma(\alpha)/\alpha_\gamma] \vdash cast_C(\alpha_\gamma, \gamma') : \gamma'(\alpha)$$

$$A\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \,\middle|\, i = 1 \ldots n\right] \vdash dict_C(\tau, \gamma) : \gamma(\tau\left[\alpha_i/\alpha_{\gamma_i} \,\middle|\, i = 1 \ldots n\right])$$

Proof: $\tilde{d}_1$ **in** $\ldots$ **in** $\tilde{d}_n$ **in** is a nested **let** expression without body; it results in assumptions for superclass dictionary access functions γ_{i_γ} and instance dictionaries of t for γ and its superclasses. The claim follows from the definitions of *cast* and *dict* and the conditions on C.

The following, main theorem of this section states that a well-typed expression in the original type system translates to an expression that is well typed in the type system of the target language.

Theorem 5.9 [Type preservation] If $(A, C) \vdash^+ e : \tau \Rightarrow \tilde{e}$ and $(FV(\tilde{e}) \setminus FV(e)) \cup (FV(\tau) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_n}\}$, then $ML(A)\left[\gamma_1(\alpha_1)/\alpha_{\gamma_1}, \ldots, \gamma_n(\alpha_n)/\alpha_{\gamma_n}\right] \vdash \tilde{e} : \tau\left[\alpha_1/\alpha_{\gamma_1}, \ldots, \alpha_n/\alpha_{\gamma_n}\right]$, where $\{\alpha_1, \ldots, \alpha_n\} \cap FV(ML(A)) = \emptyset$.

Proof: We first observe that the translation rules from Section 5.6.3 exactly implement the type translation scheme from Section 5.6.2 by producing the corresponding **let** expressions without bodies. We then continue by structural induction on the expression e, going through each case in turn:

$(A, C) \vdash^{+} x : \tau\left[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}\right] \Rightarrow (x\ dict_C(\tau_1, \gamma_1) \ldots dict_C(\tau_n, \gamma_n))$

The premise of this judgment is $A(x) = \forall\overline{\alpha_{\gamma_n}}.\tau$, whence

$(ML(A))(x) = \forall\overline{\alpha_n}.\gamma_1(\alpha_1) \to \ldots \to \gamma_n(\alpha_n) \to \tau\left[\alpha_1/\alpha_{\gamma_1}, \ldots, \alpha_n/\alpha_{\gamma_n}\right]$.

Observing that $FV(dict_C(\tau_i, \gamma_i)) = FV(\tau_i)$, let

$\{\alpha'_{\gamma_1}, \ldots, \alpha'_{\gamma_m}\} = FV(\tau_1) \cup \ldots \cup FV(\tau_n) = FV(x\ dict_C(\tau_1, \gamma_1)\ldots)$.

By Lemma 5.8 and extending the assumption set with assumptions for all of the α'_{γ_j} where necessary, we have for $1 \le i \le n$

$$ML(A)\left[\gamma'_i(\alpha'_i)/\alpha'_{\gamma_i} \mid i = 1\ldots m\right] \vdash dict_C(\tau_i, \gamma_i) : \gamma_i(\tau_i\left[\alpha'_1/\alpha'_{\gamma_1} \mid i = 1\ldots m\right])$$

By extending the assumption set again and using the TAUT rule, we also have

$$ML(A)\left[\gamma'_i(\alpha'_i)/\alpha'_{\gamma_i} \mid i = 1\ldots m\right] \vdash x : (\gamma_1(\alpha_{\gamma_1}) \to \ldots \to \gamma_n(\alpha_{\gamma_n}) \to \tau)\left[\tau_i\left[\alpha'_1/\alpha'_{\gamma_1} \mid i = 1\ldots m\right]/\alpha_{\gamma_i}\right]$$

We apply the APPL rule n times and obtain

$$ML(A)\left[\gamma'_i(\alpha'_i)/\alpha'_{\gamma_i} \mid i = 1\ldots m\right] \vdash (x\ dict_C(\tau_1, \gamma_1)\ldots dict_C(\tau_n, \gamma_n)) : (\tau\left[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}\right])\left[\alpha'_1/\alpha'_{\gamma_1} \mid i = 1\ldots m\right]$$

$(A, C) \vdash^{+} e\ e' : \tau \Rightarrow \tilde{e}\ \tilde{e}'$

The premises of this judgment according to the translation rules are $(A, C) \vdash^{+} e : \tau \to \tau' \Rightarrow \tilde{e}$ and $(A, C) \vdash^{+} e' : \tau' \Rightarrow \tilde{e}'$. Let

$(FV(\tilde{e}\ \tilde{e}') \setminus FV(e\ e')) \cup ((FV(\tau) \cup FV(\tau')) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_m}\}$.

Using Lemma 5.7, we choosing a suitable numbering such that

$(FV(\tilde{e}) \setminus FV(e)) \cup (FV(\tau' \to \tau) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_l}\}$ and

$(FV(\tilde{e'}) \setminus FV(e')) \cup (FV(\tau') \setminus FV(A)) = \{\alpha_{\gamma_{h+1}}, \ldots, \alpha_{\gamma_m}\}$, where $h \leq l$.

By the induction assumption, the following two judgments hold:

$$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \middle| i = 1 \ldots l\right] \vdash \tilde{e} : (\tau' \to \tau)\left[\alpha_i/\alpha_{\gamma_i} \middle| i = 1 \ldots l\right]$$

$$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \middle| i = h+1 \ldots m\right] \vdash \tilde{e'} : \tau'\left[\alpha_i/\alpha_{\gamma_i} \middle| i = h+1 \ldots m\right].$$

Since we can extend the assumption sets and substitutions in both judgments for variables that do not occur free, we obtain:

$$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \middle| i = 1 \ldots m\right] \vdash \tilde{e} : (\tau' \to \tau)\left[\alpha_i/\alpha_{\gamma_i} \middle| i = 1 \ldots m\right]$$

$$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \middle| i = 1 \ldots m\right] \vdash \tilde{e'} : \tau'\left[\alpha_i/\alpha_{\gamma_i} \middle| i = 1 \ldots m\right].$$

Our claim follows by applying the APPL rule and eliminating the superfluous variables

$\{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_m}\} \setminus ((FV(\tilde{e}\ \tilde{e'}) \setminus FV(e\ e')) \cup (FV(\tau) \setminus FV(A)))$ from the assumption set.

$(A, C) \vdash^+ (e_1, e_2) : \tau_1 \times \tau_2 \Rightarrow (\tilde{e}_1, \tilde{e}_2)$

$(A, C) \vdash^+ \lambda x.e : \tau' \to \tau \Rightarrow \lambda x.\tilde{e}$

These two cases are handled in a similar way as the previous one.

$(A, C) \vdash^+ \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau' \Rightarrow \mathbf{let}\ x = \lambda\overline{\alpha_{\gamma_n}}.\tilde{e}\ \mathbf{in}\ \tilde{e'}$

Let

$(FV(\tilde{e}\ \tilde{e'}) \setminus FV(e\ e')) \cup ((FV(\tau) \cup FV(\tau')) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_m}\}.$

Using Lemma 5.7, we choosing a suitable numbering such that

$(FV(\tilde{e}) \setminus FV(e)) \cup (FV(\tau) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_l}\}$ and

$(FV(\tilde{e'}) \setminus FV(e')) \cup (FV(\tau') \setminus FV(A)) = \{\alpha_{\gamma_{h+1}}, \ldots, \alpha_{\gamma_m}\}$, where

$n \le h \le l \le m$. By the inductive assumption,

$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \mid i = 1 \ldots l\right] \vdash \tilde{e} : \tau\left[\alpha_i/\alpha_{\gamma_i} \mid i = 1 \ldots l\right]$, and after n applications of the ABS rule, we obtain

$$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \mid i = n+1 \ldots l\right]$$
$$\vdash \lambda\overline{\alpha_{\gamma_n}}.\tilde{e} : \gamma_1(\alpha_1) \to \ldots \to \gamma_n(\alpha_n) \to \tau\left[\alpha_i/\alpha_{\gamma_i} \mid i = n+1 \ldots l\right]$$

We apply the inductive assumption to the last premise, observing that $FV(A) = FV(A\left[\forall\overline{\alpha_{\gamma_n}}.\tau/x\right])$:

$$ML(A\left[\forall\overline{\alpha_{\gamma_n}}.\tau/x\right])\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \mid i = h+1 \ldots m\right]$$
$$\vdash \tilde{e'} : \tau'\left[\alpha_i/\alpha_{\gamma_i} \mid i = h+1 \ldots m\right]$$

Finally, we extend the assumption sets of this and the preceding judgment to include $\alpha_{\gamma_{n+1}}, \ldots, \alpha_{\gamma_m}$ and are ready to apply the LET rule.

$(A, C) \vdash^+ K : (\tau \to t(\overline{\alpha_{\gamma_n}}))\left[\tau_i/\alpha_{\gamma_i}, \hat{\tau}_j/\beta_{\delta_j}\right] \Rightarrow$

$(\lambda x.K\,(x, dict_C(\hat{\tau}_1, \delta_1), \ldots, dict_C(\hat{\tau}_k, \delta_k))\,)$

Let $ML(A(K)) = \forall\overline{\alpha_n}.\mu\beta.\Sigma\,[K\eta] = \forall\overline{\alpha_n}.\chi$, where

$\eta = \exists\overline{\beta_k}.\tau\left[\alpha_i/\alpha_{\gamma_i}, \beta_j/\beta_{\delta_j}\right] \times \delta_1(\beta_1) \times \ldots \times \delta_k(\beta_k)$, and let

$FV(\tau_1) \cup \ldots \cup FV(\tau_n) \cup FV(\hat{\tau}_1) \cup \ldots \cup FV(\hat{\tau}_k) = \{\alpha'_{\gamma_1}, \ldots, \alpha'_{\gamma_m}\}$.

Furthermore, let $\tau' = \tau\left[\tau_i/\alpha_{\gamma_i}, \hat{\tau}_j/\beta_{\delta_j}\right]$.

We first apply the CONS rule to derive

$$ML(A)\left[\gamma'_1(\alpha'_1)/\alpha'_{\gamma_1}, \ldots, \gamma'_m(\alpha'_m)/\alpha'_{\gamma_m}, \tau'\left[\alpha'_i/\alpha'_{\gamma_i}\right]/x\right] \vdash K :$$
$$(\tau' \times \delta_1(\hat{\tau}_1) \times \ldots \times \delta_k(\hat{\tau}_k) \to \chi\,[\tau_i/\alpha_i])\left[\alpha'_i/\alpha'_{\gamma_i}\right]$$

and the TUPLE rule together with Lemma 5.7 to derive

$$ML(A)\left[\gamma'_1(\alpha'_1)/\alpha'_{\gamma_1}, \ldots, \gamma'_m(\alpha'_m)/\alpha'_{\gamma_m}, \tau'\left[\alpha'_i/\alpha'_{\gamma_i}\right]/x\right]$$
$$\vdash (x, dict_C(\hat{\tau}_1, \delta_1), \ldots, dict_C(\hat{\tau}_k, \delta_k)) :$$
$$(\tau' \times \delta_1(\hat{\tau}_1) \times \ldots \times \delta_k(\hat{\tau}_k))\left[\alpha'_i/\alpha'_{\gamma_i}\right]$$

for the argument supplied to K. We then use the APPL rule to derive

$$ML(A)\left[\gamma'_1(\alpha'_1)/\alpha'_{\gamma_1}, \ldots, \gamma'_m(\alpha'_m)/\alpha'_{\gamma_m}, \tau'\left[\alpha'_i/\alpha'_{\gamma_i}\right]/x\right]$$
$$\vdash K\ (x, dict_C(\hat{\tau}_1, \delta_1), \ldots, dict_C(\hat{\tau}_k, \delta_k)) : (\chi\,[\tau_j/\alpha_j])\left[\alpha'_i/\alpha'_{\gamma_i}\right]$$

and finally the ABS rule, which gives us

$$ML(A)\left[\gamma'_1(\alpha'_1)/\alpha'_{\gamma_1}, \ldots, \gamma'_m(\alpha'_m)/\alpha'_{\gamma_m}\right]$$
$$\vdash (\lambda x.K\,(x, dict_C(\hat{\tau}_1, \delta_1), \ldots, dict_C(\hat{\tau}_k, \delta_k))) :$$
$$(\tau' \to \chi\,[\tau_j/\alpha_j])\left[\alpha'_i/\alpha'_{\gamma_i}\right]$$

$(A, C) \vdash^+ \textbf{is}\ \ K : t(\overline{\tau_n}) \to bool \Rightarrow \textbf{is}\ \ K$

This case is a straightforward application of the TEST rule in the target language.

$(A, C) \vdash^+ \textbf{let}\ \ K\ x = e\ \ \textbf{in}\ \ e' : \tau' \Rightarrow \textbf{let}\ \ K\,(x, \delta_{\kappa_1}, \ldots, \delta_{\kappa_k}) = \tilde{e}\ \ \textbf{in}\ \ \tilde{e}'$

Let

$$(FV(\tilde{e}\ \tilde{e}') \setminus FV(e\ e')) \cup ((FV(t(\overline{\tau_n})) \cup FV(\tau')) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_m}\}$$

choosing a suitable numbering such that

$(FV(\tilde{e}) \setminus FV(e)) \cup (FV(t(\overline{\tau_n})) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_l}\}$ and

$(FV(\tilde{e'}) \setminus FV(e')) \cup (FV(\tau') \setminus FV(A)) = \{\alpha_{\gamma_{h+1}}, \ldots, \alpha_{\gamma_m}\}$, where

$h \le l \le m$.

Let $\forall\overline{\alpha'_n}.\mu\beta.\Sigma[K\exists\overline{\beta_k}.\tilde{\tau}] = ML(\sigma)$, where

$\tilde{\tau} = \tau\left[\beta_1/\beta_{\delta_1}, \ldots, \beta_k/\beta_{\delta_k}\right] \times \delta_1(\beta_1) \times \ldots \times \delta_k(\beta_k)$.

By the induction assumption,

$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \middle| i = 1 \ldots l\right] \vdash \tilde{e} : \tilde{t}\left[\alpha_i/\alpha_{\gamma_i} \middle| i = 1 \ldots l\right]$, where

$\tilde{t} = (\mu\beta.\Sigma[K\exists\overline{\beta_k}.\tilde{\tau}])\left[\tau_1/\alpha'_{\gamma_1}, \ldots, \tau_n/\alpha'_{\gamma_n}\right]$. We further apply the induction assumption to the last premise to obtain

$$ML(A)\left[\begin{array}{c} \gamma_i(\alpha_i)/\alpha_{\gamma_i} \,\middle|\, i = h+1 \ldots m \\ \tau\left[\kappa_j/\beta_{\delta_j}\right]/x,\ \delta_1(\kappa_1)/\delta_{\kappa_1}, \ldots, \delta_k(\kappa_k)/\delta_{\kappa_k} \end{array}\right]$$
$$\vdash \tilde{e'} : \tau'\left[\alpha_i/\alpha_{\gamma_i} \middle| i = h+1 \ldots m\right]$$

Note that $dict_{C'}(\kappa_j, \delta_j) = \delta_{\kappa_j}$ in $\tilde{e'}$. We now extend the assumption sets of this and the preceding judgment to include $\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_m}$, and apply the PAT'' rule. Our claim follows after restricting the final assumption set to $(FV(\tilde{e}\ \tilde{e'}) \setminus FV(e\ e')) \cup (FV(\tau') \setminus FV(A))$.

■

The following corollary is a deterministic version of the type preservation theorem in [NS91]. It covers the case of unambiguous resulting expressions, that is, expressions whose translations do not contain free dictionary variables not free in their types.

Corollary 5.10 [Type preservation] Let all types in the range of A be closed. If $(A, C) \vdash^{+} e : \tau \Rightarrow \tilde{e}$ and $FV(\tilde{e}) \setminus FV(e) \subseteq FV(\tau)$, then $ML(A) \vdash \lambda\overline{\alpha_{\gamma_n}}.\tilde{e} : \gamma_1(\alpha_1) \to \ldots \to \gamma_n(\alpha_n) \to \tau\left[\alpha_1/\alpha_{\gamma_1}, \ldots, \alpha_n/\alpha_{\gamma_n}\right]$, where $\{\alpha_1, \ldots, \alpha_n\} \cap FV(ML(A)) = \emptyset$.

Proof: We use the preceding theorem and apply the ABS rule from Chapter 3 n successive times.

Corollary 5.11 [Semantic soundness] Let all types in the range of A be closed, and let ψ be a type environment such that for every $\alpha \in \mathrm{Dom}\,\psi$, $\mathsf{wrong} \notin \psi(\alpha)$. If $(A, C) \vdash^{+} e : \tau \Rightarrow \tilde{e}$ and $\models_{\rho, \psi} ML(A)$, then $E[\![\tilde{e}]\!]\,\rho \neq \mathsf{wrong}$.

Proof: By type preservation and semantic soundness of the target language.

6 Related Work, Future Work, and Conclusions

6.1 Related Work

The following table compares our work with other programming languages with similar features or objectives. The design criteria used as a basis for our comparison are taken from Section 1.1:

1. Strong and static typing,
2. type reconstruction,
3. higher-order functions,
4. parametric polymorphism,
5. extensible abstract types with multiple implementations, and
6. first-class abstract types.

In the table, ✔ means the feature is supported, ❍ means it is not fully supported, and a blank entry means it is not supported at all.

	Design Criterion					
Language	1.	2.	3.	4.	5.	6.
Our work	✔	✔	✔	✔	✔	✔
ML/Haskell	✔	✔	✔	✔	❍	❍
SOL	✔		✔	✔	✔	✔
Hope+C	❍	✔	✔	❍	❍	✔
XML+	✔		✔	✔	✔	✔
Dynamics	❍	✔	✔	✔	❍	❍
OOL	❍			❍	❍	✔

6.1.1 SOL

SOL is based on the full second-order polymorphic λ-calculus. It is not known whether there is a type reconstruction algorithm for this language.

6.1.2 Hope+C

The only other work known to us that deals with Damas-Milner-style type reconstruction for existential types is [Per90]. However, the typing rules given there are not sufficient to guarantee the absence of runtime type errors, even though the Hope+C compiler seems to impose sufficient restrictions. The following unsafe program, here given in ML syntax, is well-typed according to the typing rules, but rejected by the compiler:

```
datatype T = K of ''a
fun f x = let val K z = x in z end
f(K 1) = f(K true)
```

In addition, an identifier bound in a pattern-matching **`let`** expression is not polymorphic according to the typing rules. This restriction does not apply to our work.

6.1.3 XML+

The possibility of making ML structures first-class by implicitly hiding their type components is discussed in [MMM91] without addressing the issue of type inference. By hiding the type components of a structure, its type is implicitly coerced from a strong sum type to an existential type. Detailed discussions of sum types can be found in [Mac86] [MH88].

6.1.4 Dynamics in ML

An extension of ML with objects that carry dynamic type information is described in [LM91]. A dynamic is a pair consisting of a value and the type of the value. Such an object is constructed from a value by applying the constructor **`dynamic`**. The object can then be dynamically coerced by pattern matching on both the value and the runtime type. Existential types are used to match dynamic values against dynamic patterns with incomplete type information. Dynamics are useful for typing functions such as **`eval`**. However, they do not provide type abstraction, since they give access to the type of an object at runtime. It seems possible to combine their system with ours, extending their existential patterns to existential types. We are currently investigating this point.

6.1.5 Object-Oriented Languages

Most statically typed object-oriented languages identify subclassing with subtyping (C++ [Str86], Modula-3 [CDG$^+$89]) at the expense of severely restricting the expressive power of the language. Due to the *contravariance rule* for function subtyping, not even simple algebraic structures can be described in C++; this is discussed in detail in [CHC90] [HL91].

Recognizing and attempting to overcome this restriction, other languages sacrifice static typing (Eiffel [Mey92], Ada 9X [dod91]) and rely on run time checks to guarantee compatibility of function arguments.

Furthermore, most object-oriented languages do not support type reconstruction; a recent advance in type reconstruction for a Smalltalk-like language is presented in [PS91].

6.2 Current State of Implementation

We have implemented a Standard ML prototype of an interpreter with type reconstruction for our core language, Mini-ML [CDDK86] extended with recursive datatypes over existentially quantified component types. The ML-style examples from this thesis have been developed and tested using our interpreter.

Technically, the interpreter consists of the following components:

- Lexer and parser were built using the tools ML-Lex [AMT89] and ML-Yacc [TA91], respectively.
- The type reconstruction phase is based on [Han87].
- The evaluator directly implements the denotational semantics presented in Section 3.6.2.

We plan further to develop this prototype towards an interpreter of a full language based on our extension of SML.

The latest releases of the Lazy ML [AJ92] and Haskell B. [Aug92] systems feature datatypes with existentially quantified component types. Both systems were developed at the Chalmers University of Technology; they provide full compilers and interpreters capable of dealing with larger programs. The Haskell examples from this thesis have been tested using the Chalmers Haskell B. interpreter.

6.3 Conclusions

The question we had set out to answer in this dissertation was:

Is it feasible to design a high-level programming language that satisfies criteria 1. *through* 6.*?*

We showed that such a design is feasible from a type-theoretic, a language design, and an implementation perspective:

- *Type-theoretic view*: Static typing and semantic soundness of the type systems hold for all three languages presented. Furthermore, we extended the Damas-Milner type reconstruction algorithm used in ML to cope with our languages.
- *Language design view*: Our examples demonstrated that we gain considerable expressiveness and flexibility by adding first-class abstract types to ML and Haskell while retaining the syntactic and semantic "look and feel" of the original languages.
- *Implementation view*: Our prototype implementation shows that our languages can be implemented using standard techniques as the ones described in [Han87] or used in the Standard ML of New Jersey implementation [AM92]. The Chalmers LML and HBC systems demonstrate that it is feasible to implement our extensions in practical compilers and interpreters.

6.4 Future Work

Our work leads off to a number of future research directions, some of which are discussed below.

6.4.1 Combination of Modules and Existential Quantification in ML

We demonstrated in Chapter 5 how Haskell type classes can be used as signatures of abstract data types. The ML module system also provides signatures, which are strong sum types. One could imagine using these signatures to describe interfaces of abstract types. First-class abstract types could then

be achieved by applying an injection that makes the type components of the signature existentially quantified, along the lines of [MMM91].

6.4.2 A Polymorphic Pattern-Matching `let` Expression

An identifier bound using the pattern-matching **`let`** expression from Chapter 5 is monomorphic, whereas an identifier bound by the corresponding expression from Chapter 3 can be used polymorphically. It would be desirable to overcome this restriction by exploring an extended target language, where a function depending on some method dictionaries can be decomposed before being applied to any arguments. While unsound in the general case, we conjecture that this is sound in our case, since the arguments are the same whenever the bound identifier is used with the same type.

6.4.3 Combination of Parameterized Type Classes and Existential Types in Haskell

Type classes in Haskell are not parameterized, thus we cannot model abstract container classes. This shortcoming was discussed in [LO91] and is also present in our extension of Haskell described in Chapter 5; thus the stack example from Section 5.2.2 is not polymorphic. An extension of Haskell with parameterized type classes was recently presented in [CHO92]; it would be desirable to apply the same extension to our language. We conjecture that parameterized type classes are an orthogonal extension and combine well with existential quantification.

Another interesting extension of Haskell is one with a dotless dot notation analogous to the ML extension from Chapter 4; it appears that such a language could be translated into the language described in Chapter 5.

6.4.4 Existential Types and Mutable State

Since the full ML language also provides polymorphic references, an extension of this language with existential types would depend on the coexistence of existential types and polymorphic references. Similar considerations hold for other forms of mutable state such as linear types [Ode91] [Wad90].

6.4.5 Full Implementation

Whereas implementations of Lazy ML and Haskell B. extended with existential types are now available, further implementation work could be envisioned both at the ML level and at the Haskell level.

At the ML level, the language would be strict and include datatypes with existentially quantified component types, polymorphic references, and possibly modules.

At the Haskell level, the language could be strict or non-strict and include existential quantification over parameterized type classes. Alternative implementation strategies for Haskell or similar languages with type classes could be explored; instead of translating to an ML-like language, type classes could be mapped to C++ templates [Ode92]. A possible starting point for further exploration could be an explicitly typed version of Mini-Haskell in the spirit of [MH88].

Bibliography

[Aba92] M. Abadi. Private communication, June 1992.

[AJ92] L. Augustsson and T. Johnsson. Lazy ML user manual. July 1992.

[AM92] A. Appel and D. MacQueen. Standard ML of New Jersey. In *Proc. 19th ACM Symp. on Principles of Programming Languages (POPL)*, January 1992.

[AMT89] A. Appel, J. Mattson, and D. Tarditi. A lexical analyzer generator for Standard Ml. Part of the Standard ML of NJ Distribution, December 1989.

[Aug92] L. Augustsson. Haskell B. user manual. July 1992.

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[BM92] K. Bruce and J. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proc. 19th ACM Symp. on Principles of Programming Languages (POPL)*, pages 316–327, January 1992.

[CDDK86] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 13–27, 1986.

[CDG+89] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. *Modula-3 Report (revised)*, Oct. 1989.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 125–135, Jan. 1990.

[CHO92] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Proc. ACM Conf. Lisp and Functional Programming*, 1992.

[CL90] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Proc. IFIP Working Conference on Programming Concepts and Methods*, pages 466–491, Sea of Gallilee, Israel, April 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[Dam85] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[dod91] Ada 9x mapping rationale, 1991. United States Department of Defense.

[FGJM85] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. 12th ACM Symp. on Principles of Programming Languages (POPL)*, pages 52–66, January 1985.

[Han87] P. Hancock. Polymorphic type checking. In S. Peyton-Jones, editor, *The Implementation of Functional Programming Languages*, chapter 8. Prentice-Hall, 1987.

[Har90] R. Harper. Introduction to Standard ML. Technical report, Carnegie Mellon University, September 1990.

[HL91] F. Henglein and K. Läufer. Programming with structures, functions, and objects. In *Proc. XVII Latin American Informatics Conference (PANEL '91)*, pages 333–352. USB, 1991.

[HPJW+92] P. Hudak, S. Peyton-Jones, P. Wadler, et al. Report on the programming language Haskell A non-strict, purely functional language Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[LM91] X. Leroy and M. Mauny. Dynamics in ML. In *Proc. Functional Programming Languages and Computer Architecture*, pages 406–426. ACM, 1991.

[LO91] K. Läufer and M. Odersky. Type classes are signatures of abstract types. In *Proc. Phoenix Seminar and Workshop on Declarative Programming*, November 1991.

[Mac86] D. MacQueen. Using dependent types to express modular structure. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 277–286. ACM, Jan. 1986.

[Mey92] B. Meyer. *Eiffel The Language*. Prentice-Hall, 1992.

[MGS89] J. Meseguer, J. Goguen, and G. Smolka. Order-sorted unification. *J. Symbolic Computation*, (8):383–413, 1989.

[MH88] J. Mitchell and R. Harper. The essence of ML. In *Proc. Symp. on Principles of Programming Languages*. ACM, Jan. 1988.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, Apr. 1982.

[MMM91] J. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1991.

[MP88] J. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988.

[MPS86] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71, 1986.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NS91] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In *Proc. Functional Programming Languages and Computer Architecture*, pages 1–14. ACM, 1991.

[Ode91] M. Odersky. Objects and subtyping in a functional perspective. IBM Research Report RC 16423, 1991.

[Ode91b] M. Odersky. How to make destructive updates less destructive. In *Proc. 18th ACM Symp. on Principles of Programming Languages (POPL)*, January 1991.

[Ode92] M. Odersky. Translating type classes to C++ templates. Private communication, March 1992.

[Per90] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, 1990.

[PJ87] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Plo83] G. Plotkin. Domains. Course notes, 1983. TeX-ed edition.

[PS91] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proc. ACM Conf. Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, pages 146–165, 1991.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12(1):23–41, 1965.

[Smy77] M. Smyth. Effectively given domains. *Theoretical Computer Science*, 5:257–274, 1977.

[SS71] D. Scott and Ch. Strachey. Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, pages 19–46, Brooklyn, New York, 1971. MRI Symp. Proc‘s, Vol. XXI.

[SS85] M. Schmidt-Schauss. A many-sorted calculus with polymorphic functions based on resulution and paramodulation. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 1162–1168, 1985.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[TA91] D. Tarditi and A. Appel. ML-Yacc Version 2.1. Part of the Standard ML of NJ Distribution, March 1991.

[Wad90] P. Wadler. Linear types can change the world. In *IFIP TC2 Working Conference on Programming Concepts and Methods*, April 1990.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.