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.

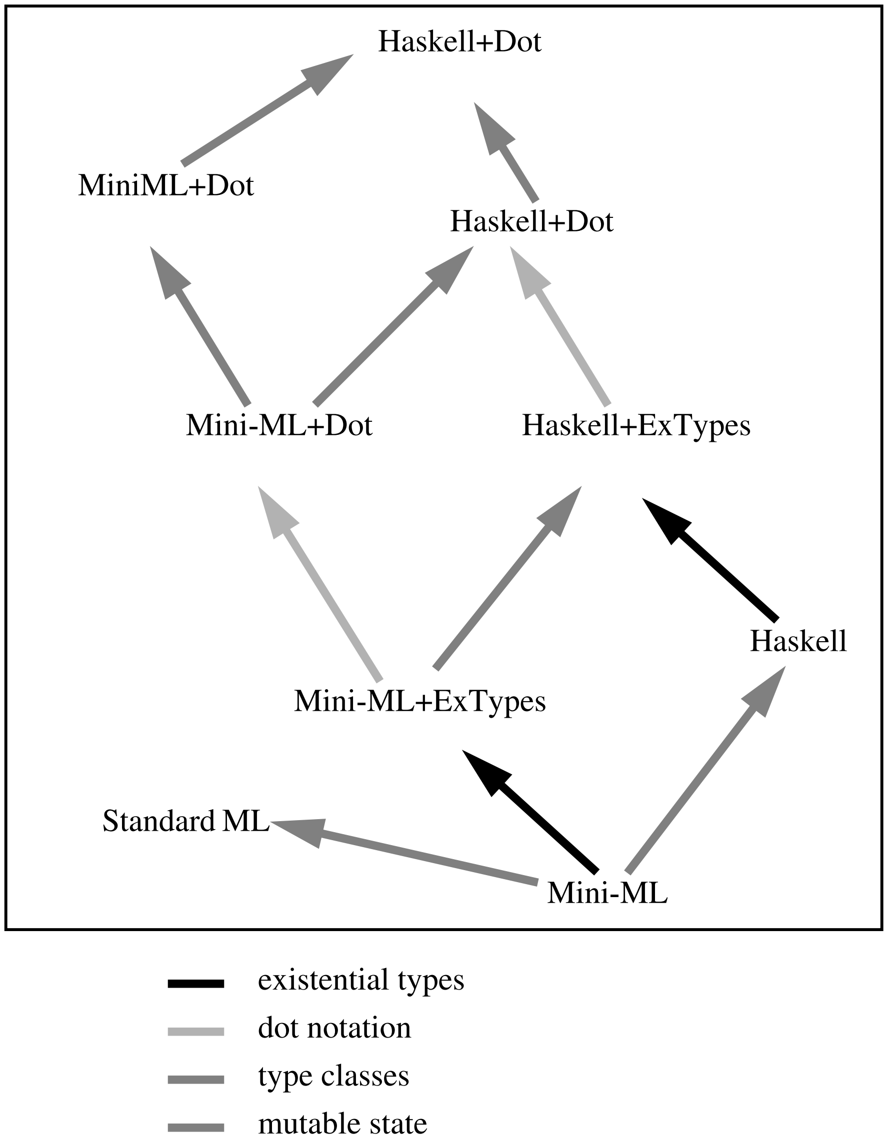
in type reconstruction algorithms. Finally, we give a review of domains and ideals, which we use as a semantic model for the languages we discuss.

- **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 ::=	$c \mid x \mid f \mid a \mid b$

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**,** ... **,** 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 **|** ... **|** 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 $+ : int \times int \to int$, we would obtain the typing

$$succ : int \to int$$

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

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

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 $int \to int$.

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

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

and is read as: "If assumption set A entails type $\tau' \rightarrow \tau$ 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) $\quad A \vdash x : A(x)$

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

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

[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 $\rightarrow$ 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]:

$$
\frac{
\frac{
\begin{bmatrix} int/x \\ int \to int/f \end{bmatrix} \vdash f : int \to int
\qquad
\frac{
\begin{array}{c}
\begin{bmatrix} int/x \\ int \to int/f \end{bmatrix} \vdash x : int \\
\begin{bmatrix} int/x \\ int \to int/f \end{bmatrix} \vdash f : int \to int
\end{array}
}{
\begin{bmatrix} int/x \\ int \to int/f \end{bmatrix} \vdash f\ x : int
}
}{
\frac{
\begin{bmatrix} int/x \\ int \to int/f \end{bmatrix} \vdash f(f\ x) : int
}{
[int \to int/f] \vdash \lambda x : int . f(f\ x) : int \to int
}
}
}{
\emptyset \vdash (\lambda f : int \to int . \lambda x : int . f(f\ x)) : (int \to int) \to (int \to int)
}
$$

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.

$$\text{(TAUT)} \quad A \vdash x : A(x)$$

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

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

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

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

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

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:

$$\dfrac{\dfrac{\dfrac{[\alpha/x] \vdash x : \alpha}{\varnothing \vdash \lambda x : \alpha.x : \alpha \to \alpha}}{\varnothing \vdash \Lambda\alpha.\lambda x : \alpha.x : \forall\alpha.\alpha \to \alpha} \qquad \dfrac{\begin{array}{c}[\forall\alpha.\alpha \to \alpha/id] \vdash id\,[int] : int \to int \\ [\forall\alpha.\alpha \to \alpha/id] \vdash 3 : int\end{array}}{[\forall\alpha.\alpha \to \alpha/id] \vdash id\,[int]\,3 : int}}{\varnothing \vdash \textbf{let}\ \ id = \Lambda\alpha.\lambda x : \alpha.x\ \ \textbf{in}\ \ id\,[int]\,3 : int}$$

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

$$\textbf{open}\ e\ \textbf{as}\ \langle t, x\rangle\ \textbf{in}\ e'$$

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

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

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 often 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].

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

$$\text{(APP)} \quad \frac{A \vdash e : \tau' \rightarrow \tau \qquad A \vdash e' : \tau'}{A \vdash (e\ e') : \tau}$$

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

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

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.

$\forall\alpha_1\ldots\alpha_n.\tau \ge \tau'$ 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.

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

For instance, $\forall\alpha.\alpha \to \alpha \ge int \to int$ but not $\forall\alpha.\alpha \to \alpha \ge int \to real$, 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 $\tau' \ge \tau$ 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.

$$W(A, x) =$$
$$\qquad (Id, inst_\forall\,(A\,(x)))$$

$$
\begin{aligned}
&W(A, e\ e') = \\
&\qquad \texttt{let}\ \ (S, \tau) = W(A, e) \\
&\qquad\qquad (S', \tau') = W(SA, e') \\
&\qquad\qquad \beta \text{ be a fresh type variable} \\
&\qquad\qquad U = mgu(S'\tau, \tau' \to \beta) \\
&\qquad \texttt{in}\ \ (US'S, U\beta)
\end{aligned}
$$

$$
\begin{aligned}
&W(A, \lambda x.e) = \\
&\qquad \texttt{let}\ \ \beta \text{ be a fresh type variable} \\
&\qquad\qquad (S, \tau) = W(A\,[\beta/x], e) \\
&\qquad \texttt{in}\ \ (S, (S\beta) \to \tau)
\end{aligned}
$$

$$
\begin{aligned}
&W(A, \texttt{let}\ \ x = e\ \ \texttt{in}\ \ e') = \\
&\qquad \texttt{let}\ \ (S, \tau) = W(A, e) \\
&\qquad\qquad (S', \tau') = W((SA)\ [gen(SA, \tau)/x], e') \\
&\qquad \texttt{in}\ \ (S'S, \tau')
\end{aligned}
$$

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.

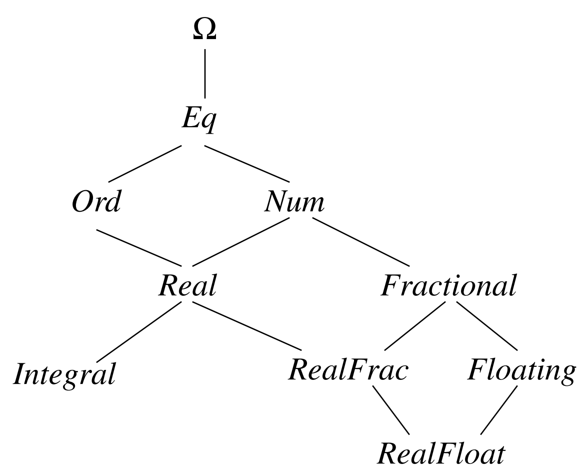
systems where types themselves are partially ordered, for example the one of OBJ [FGJM85]. *Order-sorted unification* [MGS89] can be used to obtain a type reconstruction algorithm in an order-sorted type system such as Haskell's; this is described in [NS91].

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 $\gamma' \le \gamma$ imply $\chi : (\gamma_1, \ldots, \gamma_n)\,\gamma$ for some $\gamma_1 \ge \gamma'_1, \ldots, \gamma_n \ge \gamma'_n$.

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 upper 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

$$\texttt{datatype}\ [arg]\ T\ \texttt{=}\ K_1\ \texttt{of}\ \tau_1\ \texttt{|}\ \dots\ \texttt{|}\ K_n\ \texttt{of}\ \tau_n$$

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 existential 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 $e \; ::= \; x \mid (e_1, e_2) \mid e \; e' \mid \lambda x.e \mid$

$\mathbf{let} \; x = e \; \mathbf{in} \; e' \mid$

$\mathbf{data} \; \forall\alpha_1 \ldots\alpha_n.\chi \; \mathbf{in} \; e \mid K \mid \mathbf{is} \; K \mid$

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

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 ::=	$unit \mid bool \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau \to \tau' \mid$
		$\kappa(\tau_1, \ldots, \tau_n) \mid \chi$
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	η	::=	$\exists\alpha.\eta \mid \tau$
Type schemes	σ	::=	$\forall\alpha.\sigma \mid \tau$
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

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

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

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

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].

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

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

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

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

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

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

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

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.

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

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.

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

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

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

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]:

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

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

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

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

[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.

$TC(A, x) =$

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

$TC(A, (e_1, e_2)) =$

let $(S_1, \tau_1) = TC(A, e_1)$

$(S_2, \tau_2) = TC(S_1 A, e_2)$

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

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

let $(S, \tau) = TC(A, e)$

$(S', \tau') = TC(SA, e')$

β be a fresh type variable

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

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

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

let β be a fresh type variable

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

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

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

let $(S, \tau) = TC(A, e)$

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

in $(S'S, \tau')$

$TC(A, \mathbf{data}\ \ \sigma\ \ \mathbf{in}\ \ e) =$

$\quad\mathbf{let}\ \ \forall\alpha_1\ldots\alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m = \sigma\ \ \mathbf{in}$

$\qquad\mathbf{if}\ \ FV(\sigma) = \varnothing\ \ \mathbf{then}$

$\qquad\quad TC(A[\sigma/K_1, \ldots, \sigma/K_m], e)$

$TC(A, K) =$

$\quad\mathbf{let}\ \ \tau = inst_{\forall}(A(K))$

$\qquad\mu\beta.\ldots + K\eta + \ldots = \tau$

$\quad\mathbf{in}\ \ (Id, (inst_{\exists}(\eta[\tau/\beta])) \to \tau)$

$TC(A, \mathbf{is}\ \ K) =$

$\quad\mathbf{let}\ \ \tau = inst_{\forall}(A(K))$

$\quad\mathbf{in}\ \ (Id, \tau \to bool)$

$TC(A, \mathbf{let}\ \ K\ x = e\ \ \mathbf{in}\ \ e') =$

$\quad\mathbf{let}\ \ (S, \tau) = TC(A, e)$

$\qquad U = mgu(\tau, inst_{\forall}(A(K)))$

$\qquad\mu\beta.\ldots + K\eta + \ldots = U\tau$

$\qquad\tau_\kappa = skolem(USA, \eta[U\tau/\beta])$

$\qquad(S', \tau') = TC(USA[gen(USA, \tau_\kappa)/x], e')$

$\quad\mathbf{in}$

$\qquad\mathbf{if}\ \ FS(\tau') \subseteq FS(S'USA)\ \wedge$

$\qquad\quad(FS(\tau_\kappa) \setminus FS(\eta[U\tau/\beta])) \cap FS(S'USA) = \varnothing$

$\qquad\mathbf{then}\ \ (S'US, \tau')$

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 $\{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{Dom}\,S \cup FV(\mathrm{Rng}\,S)) = \varnothing$. 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*] $S\,gen(A, \tau) = gen(SA, S\tau)$.

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

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

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.

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

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 $SA \vdash \mathbf{data}\ \ \sigma\ \ \mathbf{in}\ \ e : S\tau$.

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

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

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

$A \vdash \mathbf{let}\ \ K\ x = e\ \ \mathbf{in}\ \ e' : \tau'$

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 $S(A[\mathit{gen}(A, \mathit{skolem}(A, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x]) \vdash e' : S\tau'$. Further,

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

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

$(SA)[\mathit{gen}(SA, \mathit{skolem}(SA, (S\eta)[\mu\beta.\Sigma[K(S\eta)]/\beta]))/x]$, 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:

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

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* guarantees 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,

$$E : \mathit{Exp} \to \mathit{Env} \to V,$$

is defined as follows:

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

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

$$E[\![e\, e']\!]\, \rho \quad = \quad \begin{array}{l} \mathsf{if}\ E[\![e]\!]\, \rho \in V \to V\ \mathsf{then} \\ \quad (E[\![e]\!]\, \rho)\ (E[\![e']\!]\, \rho) \\ \mathsf{else}\ \mathsf{wrong} \end{array}$$

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

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

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

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

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

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

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.

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

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 $E[\![\texttt{data}\ \ \forall\alpha_1\ldots\alpha_n.\chi\ \ \texttt{in}\ \ e]\!]\rho = E[\![e]\!]\rho \in T[\![\tau]\!]\psi$.

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

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,

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

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,

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

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,

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

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

$A \vdash$ **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 arbitrary $I_1, \ldots, I_k \in \Re$, $\models_{\rho,\, \psi\, [I_i/\alpha_i |\, i = 1\ldots k]} A$ 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,

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

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

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

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

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

By definition of set union and intersection, there exist functions

$f_1, \ldots, f_n \in \mathfrak{I}(V)^h \to \mathfrak{I}(V)$,

such that

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

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,

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

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 $f_1, \ldots, f_n \in \mathfrak{I}(V)^h \to \mathfrak{I}(V)$ 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.