Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 136 (2005) 57–102

www.elsevier.com/locate/entcs

Programming Examples Needing Polymorphic Recursion

J. J. Hallett[1]

Department of Computer Science
Boston University
Boston, USA

A. J. Kfoury[2]

Department of Computer Science
Boston University
Boston, USA

Abstract

Inferring types for polymorphic recursive function definitions (abbreviated to *polymorphic recursion*) is a recurring topic on the mailing lists of popular typed programming languages. This is despite the fact that type inference for polymorphic recursion using ∀-types has been proved undecidable. This report presents several programming examples involving polymorphic recursion and determines their typability under various type systems, including the Hindley-Milner system, an intersection-type system, and extensions of these two. The goal of this report is to show that many of these examples are typable using a system of intersection types as an alternative form of polymorphism. By accomplishing this, we hope to lay the foundation for future research into a decidable intersection-type inference algorithm.

We do not provide a comprehensive survey of type systems appropriate for polymorphic recursion, with or without type annotations inserted in the source language. Rather, we focus on examples for which types may be inferred without type annotations, with an emphasis on systems of intersection-types.

Keywords: polymorphic recursion, intersection types, finitary polymorphism , examples

[1] Email: `jhalllett@cs.bu.edu`

[2] Email: `kfoury@cs.bu.edu`

1571-0661
doi:10.1016/j.entcs.2005.06.014

1 Introduction

Background and Motivation

Type inference in the presence of polymorphic recursion using ∀-types (the familiar "type schemes" of SML) is undecidable [10,11,4]. Attempts to work around this limitation include explicit type annotations by the user [8] and user-tunable iteration limits [17]. However, both of these approaches require the programmer to be actively engaged in the type checking process, thereby defeating the goal of automatic type inference and transparent type checking. There is also an implementation of SML that allows the user to switch between the standard type system (which is restricted to monomorphic recursion) and a type system augmented with polymorphic recursion using ∀-types, in an attempt to prove that "hard" examples of polymorphic recursion do not arise in practice [1]. Yet, practical examples of programs requiring polymorphic recursion continually appear in discussions on the mailing lists of programming languages such as SML, Haskell, and OCaml.

Contribution of the Report

This document attempts to lay the foundation for further research into the typability of implicit polymorphic recursion by discussing several examples which fail to type under the standard type system of SML – also called the Hindley-Milner system. The examples are written (mostly) in SML syntax (one example is presented in Haskell syntax) and are accompanied by the corresponding error found by the SML/NJ type checker. A few of the examples are also shown in Haskell syntax with its corresponding GHC error message for the side purpose of comparing the error reporting of the SML/NJ and GHC compilers.

We also discuss examples which remain untypable using the Hindley-Milner system augmented with polymorphic recursion with ∀-types – also called the Milner-Mycroft system – but are typable using an intersection-type system. These examples support the use of intersection types as an alternative to ∀-types to represent polymorphism.

In addition, we elucidate the need for what we call "infinite-width" intersection types by examples. However, we do not extend our standard (finite-width) intersection type system in this way, because we do not know a straightforward extension of the standard system and developing one is beyond the scope of this report. Consequently, we resort to polymorphic recursion with ∀-types for these examples; i.e., we present examples which are not typable using our intersection-type system, but are with $\forall$-types. An example is also given which requires both intersection types and $\forall$-types. Lastly, we present a polymorphic recursive program that is not typable with either intersection types or $\forall$-types.

Organization of the Report

The paper is organized as follows. First we define the types that we deal with, and then we present the rules of several type systems we consider later in the report, starting with the Hindley-Milner system which we here denote **HM**; this is done in Section 2. In the remaining sections, we introduce several simple and natural examples of polymorphic recursion to motivate the augmentation of system **HM**. We develop type systems that allow polymorphic recursion using only $\forall$-types ($\mathbf{HM}^{\forall}$), only intersection types (**S**), and both universal and intersection types ($\mathbf{S}^{\forall}$). Using these systems we show that we can construct valid typing derivations for most examples. The following chart summarizes the typability of the examples developed in this report with respect to the four type systems we define.[3] The last column in the chart, with the heading "Minor Alteration", indicates whether an example can be "easily" altered to make it typable under system **HM**.

[3] System **S** is called "**S**" for lack of a better name.

Example	**HM**	$\mathbf{HM}^{\forall}$	**S**	$\mathbf{S}^{\forall}$	Minor Alteration
Double		✓	✓	✓	✓
Mycroft		✓	✓	✓	✓
Sum List		✓	✓	✓	✓
Composition		✓	✓	✓	✓
Compiler Pass		✓	✓	✓	✓
Confusing		✓	✓	✓	
Matrix Transpose			✓	✓	✓
Vector Addition			✓	✓	✓
Collect		✓		✓	
Bar		✓		✓	
Construct List				✓	
Delay					

The above table is a little misleading in the following respect. The table indicates that certain examples are typable in our system of intersection types (**S**) but not in our system of ∀-types ($\mathbf{HM}^{\forall}$). Whereas $\mathbf{HM}^{\forall}$ restricts ∀-quantifiers to appear only in the outermost position of type expressions, **S** imposes no similar restriction on occurrences of $\wedge$ in type expressions. See Section 8 for further discussion of this matter.

Related Work

For other examples of polymorphic recursive programs, specifically nested recursive data types similar to the Collect example, see Chris Okasaki's book [16]. Simon Peyton-Jones and Mark Shields have written a paper describing the approach taken by GHC when inferring arbitrary high rank types via explicit user-defined type annotations [9].

Future Work

In the future we plan to explore the possibility of a decidable (and hopefully feasible) type inference and checking algorithm for a system of intersection types under which most, if not all, of the examples in this report can be typed. We would also like to investigate whether introducing expansion variables, a technology developed in conjunction with System $\mathbb{I}$, into our intersection type system will yield any benefits [14].

Acknowledgments

Joe Wells was a continual source of encouragement and technical advice. We would also like to thank Simon Peyton-Jones for his valuable feedback, particularly with regards to higher-rank $\forall$-types.

2 Types and Type Systems

The syntax of types is specified by the following grammar:

$$\begin{aligned} \tau \in \mathsf{Type} &::= \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \tau\ \mathsf{list} \mid \tau\ \wedge\ \tau \mid \mathsf{int} \mid \mathsf{bool} \mid\ \ldots \\ \sigma \in \mathsf{Scheme} &::= \tau \mid \forall\alpha.\sigma \end{aligned}$$

Note that we use τ as a metavariable ranging over the set Type which comprises simple types combined with intersection types, and σ as a metavariable ranging over the set Scheme which comprises all members of Type each preceded by zero or more $\forall$ quantifiers. In particular, Type is a proper subset of Scheme.

We list the four different type systems considered in the rest of the report. The basic type system, **HM**, is analogous to the type system of SML, Haskell, and OCaml, which allows let-polymorphism and only monomorphic recursion. System $\mathbf{HM}^\forall$ is an extension of system **HM** that allows polymorphic recursion with $\forall$-types, and $\forall$-types in general as long as the $\forall$ quantifiers are outside all type constructors. System **S** allows intersection types; **S** provides polymorphic recursion via intersection types. The last system that we develop is called $\mathbf{S}^\forall$. System $\mathbf{S}^\forall$ allows intersection types and $\forall$-types together; $\mathbf{S}^\forall$ also requires that $\forall$ quantifiers are kept outside all type constructors.

We now outline the conventions for reading the following tables. We assume there exists a function, type, from term constants to types, such that the $\mathsf{type}(c)$ is the type of constant c. We use Δ as a context in our typing judgement. Δ is a sequence of bindings between term variables and types. However, we also allow Δ to act as a function from term variables to types, such that $\Delta(x)$ is the type bound to variable x. Lastly, we use the function FTV from contexts to sets of type variables, such that $\mathsf{FTV}(\Delta)$ is the set of free type variables that occur in context Δ. First we define system **HM**.

System **HM** *Typing Rules:* (all types are $\wedge$-free)

$$\frac{\mathsf{type}(c) = \sigma}{\Delta \vdash c : \sigma}\ (\forall\text{-}\mathsf{Const}) \qquad (\sigma \text{ closed}) \qquad \frac{\Delta(x) = \sigma}{\Delta \vdash x : \sigma}\ (\forall\text{-}\mathsf{Var})$$

$$\frac{\Delta, x : \tau \vdash M : \tau'}{\Delta \vdash \mathsf{fn}\ x => M : \tau \to \tau'}\ (\mathsf{Abs}) \qquad \frac{\Delta \vdash M : \tau \to \tau' \quad \Delta \vdash N : \tau}{\Delta \vdash MN : \tau'}\ (\mathsf{App})$$

$$\frac{\Delta \vdash M : \sigma \quad \Delta, x : \sigma \vdash N : \tau}{\Delta \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N\ \mathsf{end} : \tau}\ (\forall\text{-}\mathsf{Let})$$

$$\frac{\begin{array}{c}\Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash N : \tau \\ \Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M_p : \tau_p\end{array}}{\begin{array}{l}\Delta \vdash \mathsf{let\ val\ rec}\ x_1 = M_1\ \mathsf{and} \ldots \\ \qquad \mathsf{and}\ x_n = M_n\ \mathsf{in}\ N\ \mathsf{end} : \tau\end{array}}\ (\mathsf{Rec}) \qquad (1 \leq p \leq n)$$

$$\frac{\Delta \vdash M : \sigma}{\Delta \vdash M : \forall\alpha.\sigma}\ (\mathsf{Gen}) \qquad (\alpha \notin \mathsf{FTV}(\Delta)) \qquad \frac{\Delta \vdash M : \forall\alpha.\sigma}{\Delta \vdash M : \sigma[\alpha := \tau]}\ (\mathsf{Inst})$$

$$\frac{\Delta \vdash M_1 : \tau_1 \quad \Delta \vdash M_2 : \tau_2}{\Delta \vdash (M_1, M_2) : \tau_1 \times \tau_2}\ (\times)$$

$$\frac{\Delta \vdash M : \tau_1 \times \tau_2}{\Delta \vdash \mathsf{fst}(M) : \tau_1}\ (\mathsf{Fst}) \qquad \frac{\Delta \vdash M : \tau_1 \times \tau_2}{\Delta \vdash \mathsf{snd}(M) : \tau_2}\ (\mathsf{Snd})$$

$$\frac{\Delta \vdash M_1 : \mathsf{bool} \quad \Delta \vdash M_2 : \tau \quad \Delta \vdash M_3 : \tau}{\Delta \vdash \mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 : \tau}\ (\mathsf{If})$$

To define system $\mathbf{HM}^{\forall}$ we simply augment **HM** with the rule ($\forall$-Rec).

System $\mathbf{HM}^{\forall}$ *Typing Rules:* (all types are $\wedge$-free)

All the typing rules of system **HM** are typing rules of system $\mathbf{HM}^{\forall}$ in addition to the following.

$$\frac{\begin{array}{l}\Delta, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash N : \tau \\ \Delta, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M_p : \sigma_p\end{array}}{\begin{array}{l}\Delta \vdash \mathsf{let\ val\ rec}\ x_1 = M_1\ \mathsf{and} \ldots \\ \qquad \mathsf{and}\ x_n = M_n\ \mathsf{in}\ N\ \mathsf{end} : \tau\end{array}}\ (\forall\text{-}\mathsf{Rec}) \qquad (1 \leq p \leq n)$$

Note that we allow both rules ($\forall$-Rec) and (Rec) to co-exist within system $\mathbf{HM}^{\forall}$. This is acceptable because (Rec) is simply a special case of ($\forall$-Rec). System **S** uses only intersection types.

System **S** *Typing Rules:* (all types are $\forall$-free)

$$\frac{\mathsf{type}(c) = \tau}{\Delta \vdash c : \tau}\ (\wedge\text{-Const}) \qquad (\tau \text{ closed}) \qquad \frac{\Delta(x) = \tau}{\Delta \vdash x : \tau}\ (\wedge\text{-Var})$$

$$\frac{\Delta, x : \tau \vdash M : \tau'}{\Delta \vdash \mathsf{fn}\ x \Rightarrow M : \tau \to \tau'}\ (\mathsf{Abs}) \qquad \frac{\Delta \vdash M : \tau \to \tau' \quad \Delta \vdash N : \tau}{\Delta \vdash MN : \tau'}\ (\mathsf{App})$$

$$\frac{\Delta \vdash M : \tau' \quad \Delta, x : \tau' \vdash N : \tau}{\Delta \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N\ \mathsf{end} : \tau}\ (\wedge\text{-Let})$$

$$\frac{\begin{array}{l}\Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash N : \tau \\ \Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M_p : \tau_p\end{array}}{\begin{array}{l}\Delta \vdash \mathsf{let\ val\ rec}\ x_1 = M_1\ \mathsf{and} \ldots \\ \qquad \mathsf{and}\ x_n = M_n\ \mathsf{in}\ N\ \mathsf{end} : \tau\end{array}}\ (\wedge\text{-Rec}) \qquad (1 \leq p \leq n)$$

$$\frac{\Delta \vdash M_1 : \tau_1 \quad \Delta \vdash M_2 : \tau_2}{\Delta \vdash (M_1, M_2) : \tau_1 \times \tau_2}\ (\times)$$

$$\frac{\Delta \vdash M : \tau_1 \times \tau_2}{\Delta \vdash \mathsf{fst}(M) : \tau_1}\ (\mathsf{Fst}) \qquad \frac{\Delta \vdash M : \tau_1 \times \tau_2}{\Delta \vdash \mathsf{snd}(M) : \tau_2}\ (\mathsf{Snd})$$

$$\frac{\Delta \vdash M : \tau_i \quad i \in I}{\Delta \vdash M : \bigwedge_{i \in I} \tau_i}\ (\wedge) \qquad (\mathsf{size}(I) \geq 2), (\mathrm{size}(I) \text{ is finite})$$

$$\frac{\Delta \vdash M : \tau \quad \tau \leq \tau'}{\Delta \vdash M : \tau'}\ (\mathsf{Sub})$$

$$\frac{}{\tau \leq \tau}\ (\mathsf{S\text{-}Refl}) \qquad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ (\mathsf{S\text{-}Trans})$$

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2' \leq \tau_2}{\tau_1' \to \tau_2' \leq \tau_1 \to \tau_2}\ (\mathsf{S\text{-}Fun}) \qquad \frac{\tau_1' \leq \tau_1 \quad \tau_2' \leq \tau_2}{\tau_1' \times \tau_2' \leq \tau_1 \times \tau_2}\ (\mathsf{S\text{-}Pair})$$

$$\frac{\tau_i \leq \tau_i' \quad i \in I \quad I \subseteq J}{\bigwedge_{i \in J} \tau_i \ \leq\ \bigwedge_{i \in I} \tau_i'}\ (\mathsf{S\text{-}}\wedge)$$

This system has been proved sound. The proof can be found in appendix B. Lastly, we define system $\mathbf{S}^{\forall}$.

System $\mathbf{S}^{\forall}$ *Typing Rules:*

All the typing rules of system $\mathbf{S}$ are typing rules of system $\mathbf{S}^{\forall}$ in addition to the following.

$$\frac{\mathsf{type}(c) = \sigma}{\Delta \vdash c : \sigma}\ (\forall\text{-}\mathsf{Const}) \qquad (\sigma \text{ closed}) \qquad \frac{\Delta(x) = \sigma}{\Delta \vdash x : \sigma}\ (\forall\text{-}\mathsf{Var})$$

$$\frac{\Delta \vdash M : \sigma \quad \Delta, x : \sigma \vdash N : \tau}{\Delta \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N\ \mathsf{end} : \tau}\ (\forall\text{-}\mathsf{Let})$$

$$\frac{\begin{array}{l}\Delta, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash N : \tau \\ \Delta, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M_p : \sigma_p\end{array}}{\begin{array}{l}\Delta \vdash \mathsf{let\ val\ rec}\ x_1 = M_1\ \mathsf{and} \ldots \\ \qquad \mathsf{and}\ x_n = M_n\ \mathsf{in}\ N\ \mathsf{end} : \tau\end{array}}\ (\forall\text{-}\mathsf{Rec}) \qquad (1 \leq p \leq n)$$

$$\frac{\Delta \vdash M : \sigma}{\Delta \vdash M : \forall \alpha.\sigma}\ (\mathsf{Gen}) \qquad (\alpha \notin \mathsf{FTV}(\Delta)) \qquad \frac{\Delta \vdash M : \forall \alpha.\sigma}{\Delta \vdash M : \sigma[\alpha := \tau]}\ (\mathsf{Inst})$$

Note that ($\wedge$-Const), ($\wedge$-Var), ($\wedge$-Let), and ($\wedge$-Rec) in system $\mathbf{S}$ are special cases of ($\forall$-Const), ($\forall$-Var), ($\forall$-Let), and ($\forall$-Rec) in system $\mathbf{S}^{\forall}$.

3 Typable in $\mathbf{HM}^{\forall}$ and S

3.1 Double

3.1.1 Double - Coupled

The following is a simple example that exposes the untypability of polymorphic recursion in SML.

```
let val rec double = fn f => fn y => f (f y)
    and foo = fn v => double (fn x => x + 1) v
    and goo = fn w => double Math.sqrt w
in (foo 3, goo 16.0) end
```

SML Type Checker Reports:

```
Error: operator and operand don't agree [literal]
  operator domain: real -> real
  operand:         int -> int
  in expression:
    double (fn x => x + 1)
```

The definitions of `double`, `foo`, and `goo` are mutually recursive. Therefore the calls to `double` within the definition of `foo` and `goo` are recursive calls. Hence, the Hindley-Milner typing derivation breaks down with the realization that each of these recursive calls is on an argument of a different type.

This example is not typable under system **HM**. However, we can use either $\mathbf{HM}^{\forall}$ or **S** to type it. Using $\mathbf{HM}^{\forall}$ we can write a typing derivation for this example, where the final types assigned are:

$$\begin{aligned}
\texttt{double} &: \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha \\
\texttt{foo} &: \mathsf{int} \to \mathsf{int} \\
\texttt{goo} &: \mathsf{real} \to \mathsf{real}.
\end{aligned}$$

Using **S** we can also type this example. If `double` is given the following intersection type:

$$((\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}) \wedge ((\mathsf{real} \to \mathsf{real}) \to \mathsf{real} \to \mathsf{real})$$

then the call to `double` within the body of `foo` would be able to utilize the first component of the intersection type and the call to `double` within the body of `goo` would be able to use the second component. We hold off on a typing derivation in **S** until the next, more complicated, example.

3.1.2 An Aside: SML/NJ vs. GHC

As an aside we translate a couple of the examples in this report into Haskell syntax and compare the SML/NJ error messages with the GHC error messages (which uses Algorithm M in contrast to Algorithm W of SML/NJ - for more discussion see [3]). We choose to translate only those examples which will yield an interesting and different error message. Most of the following examples, when translated, offer error messages that are very similar to the SML/NJ error messages, but differ occasionally in the program location which the compiler targets as problematic. This example, when translated, is no different.

```
intFunc :: Int -> Int
```

```
intFunc x = x + 1

doubleFunc :: Double -> Double
doubleFunc x = sqrt x

myPair = let (double, foo, goo) =
             (\f -> \y -> f (f y),
              \v -> double intFunc v,
              \w -> double doubleFunc w)
         in (foo 3, goo 16.0)
```

GHC Type Checker Reports:

```
Couldn't match 'Double' against 'Int'
    Expected type: Int -> Int
    Inferred type: Double -> Double
In the first argument of 'double', namely 'doubleFunc'
In a lambda abstraction: \ w -> double doubleFunc w
```

Both the SML/NJ and the GHC compiler detect the same error but SML/NJ assigns the type:

$$\texttt{double} : (\mathsf{real} \to \mathsf{real}) \to \mathsf{real} \to \mathsf{real},$$

while GHC assigns the type:

$$\texttt{double} : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}.$$

Although this difference is not enormous, it does show an operational disparity between the two compilers.

3.1.3 Double - Uncoupled

The problem exhibited in the `double` example above can be alleviated by a technique that we call "uncoupling". Namely, we make use of the Hindley-Milner let-polymorphism by removing double from the mutual recursive definition and defining it in an outer let.

```
let val double = fn f => fn y => f (f y)
in let val rec foo = fn v => double (fn x => x + 1) v
       and goo = fn w => double Math.sqrt w
   in (foo 3, goo 16.0) end
end
```

SML Type Checker Reports:

```
No Errors
```

3.2 Mycroft

3.2.1 Mycroft - Coupled

The following is the canonical example of polymorphic recursion as discovered by Alan Mycroft [15].

```
let val rec myMap = fn f => fn l =>
                    if (null l)
                    then l
                    else cons (f(hd l)) (myMap f (tl l))
    and sqList = fn l => myMap (fn (x:int) => x * x) l
    and compList = fn l => myMap not l
in (sqList [2,4], compList [true,false]) end
```

SML Type Checker Reports:

```
Error: operator and operand don't agree [tycon mismatch]
  operator domain: bool -> bool
  operand:         int -> int
  in expression:
    myMap (fn x : int => x * x)
```

As before, we have three mutually recursive function definitions and two recursive calls with arguments of different types. This example, though untypable in system **HM**, can be typed in a system of polymorphic recursion with ∀-types or intersection types. To witness this either system must be able to handle lists. For the purposes of brevity we will consider `hd`, `tl`, `cons`, and `nil` to all be primitive constants within our language. With these constants we will be able to handle expressions with list types. Also, we note that the expressions `[1,2]` and `[true, false]` are simply syntactic sugar for `cons 1 (cons 2 nil)` and `cons true (cons false nil)` respectively. We are now able to assign the following types under $\mathbf{HM}^{\forall}$:

$$\begin{aligned} \texttt{myMap} &: \forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha\ \mathsf{list} \to \beta\ \mathsf{list} \\ \texttt{sqList} &: \mathsf{int\ list} \to \mathsf{int\ list} \\ \texttt{compList} &: \mathsf{bool\ list} \to \mathsf{bool\ list}. \end{aligned}$$

With **S** we can assign the following rank-1 types:

$$\begin{aligned} \texttt{myMap} &: ((\mathsf{int} \to \mathsf{int}) \to \mathsf{int\ list} \to \mathsf{int\ list}) \wedge \\ &\quad ((\mathsf{bool} \to \mathsf{bool}) \to \mathsf{bool\ list} \to \mathsf{bool\ list}) \\ \texttt{sqList} &: \mathsf{int\ list} \to \mathsf{int\ list} \\ \texttt{compList} &: \mathsf{bool\ list} \to \mathsf{bool\ list}. \end{aligned}$$

In both systems the final type assigned to Mycroft's example is:

$$\mathsf{int\ list} \times \mathsf{bool\ list}.$$

For the full typing derivation under **S** see appendix A.

3.2.2 Mycroft - Uncoupled

As before, uncoupling is possible. This is shown in a slightly different form below.

```
let val rec myMap = fn f => fn l =>
                    if (null l)
                    then l
                    else cons (f(hd l)) (myMap f (tl l))
    val rec sqList = fn l => myMap (fn x => x * x) l
    and compList = fn l => myMap not l
in (sqList [2,4], compList [true,false]) end
```

SML Type Checker Reports:

```
No Errors
```

3.3 Sum List

The example below finds the sum of the elements of a list, but also applied the polymorphic identity function to each element and sublist in the process. The idea here is that we may want to record some information about each element and its corresponding sublist (possibly via side effects).

```
let val rec id = fn x => x
    and sumList = fn l =>
                  if (null l)
                  then 0
                  else (id (hd l)) + (sumList (id (tl l)))
in sumList [1,2,3] end
```

SML Type Checker Reports:

```
Error: operator and operand don't agree [circularity]
  operator domain: 'Z
  operand:         'Z list
  in expression:
    id (tl l)
```

Using $\mathbf{HM}^{\forall}$ we assign the following types:

$$\texttt{id} : \forall\alpha.\alpha \to \alpha$$
$$\texttt{sumList} : \mathsf{int\ list} \to \mathsf{int}.$$

Using $\mathbf{S}$ we assign the following types:

$$\texttt{id} : (\mathsf{int} \to \mathsf{int}) \wedge (\mathsf{int\ list} \to \mathsf{int\ list})$$
$$\texttt{sumList} : \mathsf{int\ list} \to \mathsf{int}.$$

The final type assigned to this example is: int. This example can be uncoupled in the same way as the previous two examples. A natural question at this point would be to ask why `id` needs to be defined mutually recursive to `sumList`. To avoid such a question we could pass `id` as an argument to `sumList` and then motivate this move by demonstrating a need to pass two different functions to `sumList`. We show this for the Matrix Transpose example so we do not show it here.

3.4 Isomorphic Compositions

This example uses the composition function as the polymorphic recursive function. The order of two composed functions are switched and applied to different arguments. The results of both applications are then compared.

```
let val createList = fn x => [x]
    val removeList = fn l => hd l
    val rec comp = fn f => fn g => f o g
    and appComp = fn v1 => fn v2 =>
                  (comp removeList createList v1) =
                  hd (comp createList removeList v2)
in appComp 5 [5] end
```

SML Type Checker Reports:

```
Error: operator and operand don't agree [circularity]
  operator domain: 'Z list -> 'Z
  operand:         'Z list -> 'Z list list
  in expression:
```

```
    comp createList
```

Using $\mathbf{HM}^{\forall}$ we assign the following types:

$$\begin{aligned}
\texttt{createList} &: \mathsf{int} \to \mathsf{int\ list} \\
\texttt{removeList} &: \mathsf{int\ list} \to \mathsf{int} \\
\texttt{comp} &: \forall\alpha.\forall\beta.\forall\eta.(\beta \to \eta) \to (\alpha \to \beta) \to \alpha \to \eta \\
\texttt{appComp} &: \mathsf{int} \to \mathsf{int\ list} \to \mathsf{bool}.
\end{aligned}$$

Using $\mathbf{S}$ we assign the following types:

$$\begin{aligned}
\texttt{createList} &: \mathsf{int} \to \mathsf{int\ list} \\
\texttt{removeList} &: \mathsf{int\ list} \to \mathsf{int} \\
\texttt{comp} &: ((\mathsf{int} \to \mathsf{int\ list}) \to (\mathsf{int\ list} \to \mathsf{int}) \to \mathsf{int\ list} \to \mathsf{int\ list}) \wedge \\
&\quad ((\mathsf{int\ list} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int\ list}) \to \mathsf{int} \to \mathsf{int}) \\
\texttt{appComp} &: \mathsf{int} \to \mathsf{int\ list} \to \mathsf{bool}.
\end{aligned}$$

In both systems the final type assigned to this example is: bool. This example can also be uncoupled.

3.5 Compiler Pass

This example is very similar to the previous examples and is due to Simon Peyton Jones [6,7], who states that this is a program that he "really wanted to write". The author was writing a compiler pass which made use of two data types and three functions written in continuation passing style. It is presented in Haskell syntax.

```
data Exp = Let Bind Exp
data Bind = MkBind String Exp

doBinds (b:bs) = doBindAndScope b (\b' -> b' : doBinds bs)

doExp (Let b e) = doBindAndScope b (\b' -> Let b' (doExp e))

doBindAndScope (MkBind s e) cont = cont (MkBind s (doExp e))
```

GHC Type Checker Reports:

```
Couldn't match '[Bind]' against 'Exp'
    Expected type: [Bind]
    Inferred type: Exp
In the application 'doBinds bs'
In the second argument of '(:)', namely 'doBinds bs'
```

The trouble with this program is that `doExp` and `doBindAndScope` are defined mutually recursive to one another. This means that the call to `doBindAndScope` is a recursive call and can not be polymorphic. However, `doBinds` and `doExp` each call `doBindAndScope` with arguments of different types. The author goes on to describe a way to alleviate this problem by encapsulating the polymorphism inside a data type structure and adding constructors to the arguments of `doBindAndScope`. However, he points out that this fix is not only "obscure", but also "inefficient at runtime".

This example can be typed by either $\mathbf{HM}^{\forall}$ or $\mathbf{S}$. Under system $\mathbf{HM}^{\forall}$ we can assign the following types:

$$\begin{aligned} \texttt{doBinds} &: \mathsf{Bind\ list} \to \mathsf{Bind\ list} \\ \texttt{doExp} &: \mathsf{Exp} \to \mathsf{Exp} \\ \texttt{doBindAndScope} &: \forall\alpha.\mathsf{Bind} \to (\mathsf{Bind} \to \alpha) \to \alpha. \end{aligned}$$

Under system $\mathbf{S}$ we can assign these types:

$$\begin{aligned} \texttt{doBinds} &: \mathsf{Bind\ list} \to \mathsf{Bind\ list} \\ \texttt{doExp} &: \mathsf{Exp} \to \mathsf{Exp} \\ \texttt{doBindAndScope} &: (\mathsf{Bind} \to (\mathsf{Bind} \to \mathsf{Bind\ list}) \to \mathsf{Bind\ list}) \wedge \\ & \ (\mathsf{Bind} \to (\mathsf{Bind} \to \mathsf{Exp}) \to \mathsf{Exp}). \end{aligned}$$

Besides the method for alleviating this example already discussed, we can uncouple this program in the usual way.

3.6 Confusing

3.6.1 Confusing - Unalleviated

The following example is not very intuitive but serves a purpose.

```
let val rec f = fn n => fn x => fn y =>
                if x > y orelse n = 0
                then n
                else if n >= 100
                     then if n < 200
                          then n
                          else f (n div 2) (x * y) y
                     else if x < y
                          then f (n*n) 0.03 1.0
                          else f (n*n) 1 1
in f 3 5 6 end
```

SML Type Checker Reports:

```
Error: operator and operand don't agree [literal]
  operator domain: real
  operand:         int
  in expression:
    (f (n * n)) 1

Error: operator and operand don't agree [literal]
  operator domain: real
  operand:         int
  in expression:
    (f 3) 5
```

This example requires the second and third arguments of `f` to be of types int and real. The example makes use of the overloaded operators <, >, and * which are defined for both these types. Notice that if we give `f` the appropriate type then this example is well-typed within both $\mathbf{HM}^\forall$ and $\mathbf{S}$.

Under $\mathbf{HM}^\forall$ we assign the following type:

$$\mathtt{f} : \forall\alpha.\mathsf{int} \to \alpha \to \alpha \to \mathsf{int}.$$

Under $\mathbf{S}$ we assign the following type:

$$\mathtt{f} : (\mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}) \wedge (\mathsf{int} \to \mathsf{real} \to \mathsf{real} \to \mathsf{int}).$$

In both systems, the final type assigned to the example is: int.

This example differs from all the previous examples. The preceding examples all make use of a polymorphic function that is defined mutually recursive to another function. The polymorphic function is then used twice on arguments of different types. This example is designed to show that it is possible to define a polymorphic recursive function that is inherently so, without the aid of an external polymorphic function. As a result, this example is difficult to alleviate. In the next sections we will see other polymorphic recursive functions that share this same property but are impossible to type without extensions to $\mathbf{HM}^\forall$ and $\mathbf{S}$.

3.6.2 An Aside: SML/NJ vs. GHC

It is worth noting that when translated into Haskell syntax this example can be typed by the GHC compiler. The reason for this is that the GHC compiler converts the integers in this example to doubles and assigns the following type:

$$\texttt{f} : \mathsf{double} \to \mathsf{double} \to \mathsf{double} \to \mathsf{double}.$$

3.6.3 Confusing - Alleviated

We can alleviate this example by duplication. Consider the following program.

```
let val rec f1 = fn n => fn x => fn y =>
                 if x > y orelse n = 0
                 then n
                 else if n >= 100
                      then if n < 200
                           then n
                           else f1 (n div 2) (x * y) y
                      else if x < y
                           then f2 (n*n) 0.03 1.0
                           else f1 (n*n) 1 1
    and f2 = fn n => fn x => fn y =>
             if x > y orelse n = 0
             then n
             else if n >= 100
                  then if n < 200
                       then n
                       else f2 (n div 2) (x * y) y
                  else if x < y
                       then f2 (n*n) 0.03 1.0
                       else f1 (n*n) 1 1
in f1 3 5 6 end
```

This program is now typable under **HM**. We can assign the following types:

$$\texttt{f1} : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}$$
$$\texttt{f2} : \mathsf{int} \to \mathsf{real} \to \mathsf{real} \to \mathsf{int}.$$

However, alleviating the example in this way differs from all the previous attempts in that we must duplicate the entire program. Since duplication defeats the purpose of polymorphism this alteration cannot be recommended.

4 Typable in S Only

4.1 Matrix Transpose

4.1.1 Matrix Transpose - Unalleviated

This examples shows a concise and elegant formulation of the matrix transpose operation.

```
let val map1 = map
    val rec map2 = fn f => fn l =>
                   if (null l)
                   then nil
                   else if (null (hd l))
                        then nil
                        else cons (f hd l) (map2 f (f tl l))
in map2 map1 [[1,2],[3,4]] end
```

SML Type Checker Reports:

```
Error: operator and operand don't agree [circularity]
  operator domain: 'Z list -> 'Z
  operand:         'Z list -> 'Z list
  in expression:
    f tl
```

This example, unlike the previous examples, cannot be typed by polymorphic recursion with ∀-types. The problem arises when trying to type the first argument to `map2`, `f`. To see this, we need only look at the else-branch of the nested if-expression.

Notice that from `cons (f hd l) (map2 f (f tl l))` the type of the first occurrence of `f` must be of the form:

$$\mathtt{f} : (\alpha\ \mathsf{list} \to \alpha) \to \alpha\ \mathsf{list}\ \mathsf{list} \to \alpha\ \mathsf{list}.$$

Yet, the second occurrence of `f` requires the form:

$$\mathtt{f} : (\alpha\ \mathsf{list} \to \alpha\ \mathsf{list}) \to \alpha\ \mathsf{list}\ \mathsf{list} \to \alpha\ \mathsf{list}\ \mathsf{list}.$$

Thus, `f` must have a polymorphic type. However, since we restrict ∀-quantifiers to be only on the outer most portion of the type, a ∀-type for `map2` is impossible.

Fortunately, using **S** we are able to assign this example a rank-2 type:

$$\begin{aligned}
\texttt{map1} :\ & ((\textsf{int list} \to \textsf{int}) \to \textsf{int list list} \to \textsf{int list}) \wedge \\
& ((\textsf{int list} \to \textsf{int list}) \to \textsf{int list list} \to \textsf{int list list}) \\
\texttt{map2} :\ & (((\textsf{int list} \to \textsf{int}) \to \textsf{int list list} \to \textsf{int list}) \wedge \\
& ((\textsf{int list} \to \textsf{int list}) \to \textsf{int list list} \to \textsf{int list list})) \\
& \to \textsf{int list list} \to \textsf{int list list}.
\end{aligned}$$

The final type assigned to this example is: $\textsf{int list list}$.

An objection made in a preliminary presentation of this work is that this example (Matrix Transpose) and the next (Vector Addition) are not cases of truly polymorphic recursive functions, because the polymorphism is not at the outermost position of the type expression, as in the previous examples. However, such a definition of polymorphic recursion is arguably too restrictive, as it disallows function types whose argument type (i.e., expressions to the left of the arrow constructor) are polymorphic.

4.1.2 Matrix Transpose - Alleviated

Similar to the previous example, uncoupling is impossible. However, we can side-step this dilemma with another crafty trick.

```
let val map1 = map
    val rec map2 = fn f1 => fn f2 => fn l =>
                   if (null l)
                   then nil
                   else if (null (hd l))
                        then nil
                        else cons (f1 hd l)
                             (map2 f1 f2 (f2 tl l))
in map2 map1 map1 [[1,2],[3,4]] end
```

SML Type Checker Reports:

```
No Errors
```

By simply passing the `map2` function two different `map1` functions so that each one is used with only one type, our example becomes typable. Although, this technique yields a well-typed program the process for transforming untypable polymorphic recursive programs has become ad-hoc. No longer, can the programmer use a simple uncoupling scheme. Instead, the programmer must come up with, possibly very complex, fixes for each circumstance. A better programming language would not require these efforts from the programmer, but rather allow the program to be typed as the programmer wrote it. With this as our goal we reject the alleviated example as our ultimate solution and determine to type the original, unalleviated example.

As an alternative alleviation, one could simply remove the first argument, `f`, of `map2` and replace each `f` in the body of `map2` with the standard `map` function. However, there may be cases where passing `map1` as an argument is advantageous. For example, consider the following. Suppose that given a matrix M, one wishes to compare the transpose of M with the transpose of $\bar{M}$, where $\bar{M}$ is defined as follows: if $\bar{m}_{ij}$ is an element (i, j) of $\bar{M}$ then $\bar{m}_{ij} = m_{ij} + i$ where m_{ij} is an element (i, j) of M. Then we can compute the pair $(M^{\mathsf{T}}, \bar{M}^{\mathsf{T}})$ as follows.

```
let val map1 = fn f => fn l => map f (map (map (fn x => x+1)) l)
    val rec map2 = fn f => fn l =>
                     if (null l)
                     then nil
                     else if (null (hd l))
                          then nil
                          else cons (f hd l) (map2 f (f tl l))
in (map2 map [[1,2],[3,4]], map2 map1 [[1,2],[3,4]]) end
```

Otherwise, the programmer would have to compute the transpose of M and separately compute $\bar{M}^{\mathsf{T}}$ from M^{T}. A program that was implemented in this way would require significant code duplication.

4.2 Vector Addition

This example computes the addition of equal-length vectors represented as list.

```
let val addList = fn l => foldr (op +) 0 l
    val rec addVecs = fn f => fn l =>
                        if (null (hd l))
                        then nil
                        else cons (addList (f hd l))
                             (addVecs f (f tl l))
in addVecs map [[1,2,3],[4,5,6]] end
```

SML Type Checker Reports:

```
Error: operator and operand don't agree [circularity]
  operator domain: 'Z list -> 'Z
  operand:         'Z list -> 'Z list
  in expression:
```

```
  f tl
```

This example is very similar to the Matrix Transpose example. Just has before, the `f` argument of `addVecs` requires a polymorphic type. However, since we disallow ∀-quantification within a function type, system $\mathbf{HM}^{\forall}$ is not sufficient to type `addVecs`.

Again using **S** we are able to assign this example a rank-2 type:

$$\begin{aligned}\texttt{addList} &: \mathsf{int\ list} \to \mathsf{int}\\ \texttt{addVecs} &: (((\mathsf{int\ list} \to \mathsf{int}) \to \mathsf{int\ list\ list} \to \mathsf{int\ list}) \wedge\\ &\quad\ ((\mathsf{int\ list} \to \mathsf{int\ list}) \to \mathsf{int\ list\ list} \to \mathsf{int\ list\ list}))\\ &\quad\ \to \mathsf{int\ list\ list} \to \mathsf{int\ list}.\end{aligned}$$

The final type assigned to this example is: $\mathsf{int\ list}$.

And again, we can alleviate this example using the alternative techniques to uncoupling described for the Matrix Transpose example alleviation.

5 Typable in $\mathbf{HM}^{\forall}$ Only

5.1 Collect

This example from the ML mailing list was already discussed by Trevor Jim [5]. This function collects all the data from the defined data type and stores them in a list.

```
datatype 'a T = EMPTY
              | NODE of 'a * ('a T) T

let val rec collect = fn t =>
                      case t of
                          EMPTY = nil
                        | NODE(n,t) =
                          cons n
                          (flatmap collect (collect t))
in collect EMPTY end
```

SML Type Checker Reports:

```
Error: operator and operand don't agree [circularity]
  operator domain: 'Z T
  operand:         'Z T T
  in expression:
    collect t
```

Here `flatmap` is a function similar to the `map` function. The type of `flatmap` is:

$$\texttt{flatmap} : (\alpha \to \beta\ \mathsf{list}) \to \alpha\ \mathsf{list} \to \beta\ \mathsf{list}.$$

Obviously this example is not typable in **HM**, however, using system $\mathbf{HM}^{\forall}$ we can give this example the following types:

$$\begin{aligned}&\texttt{flatmap} : \forall\alpha.\forall\beta.(\alpha \to \beta\ \mathsf{list}) \to \alpha\ \mathsf{list} \to \beta\ \mathsf{list}\\ &\texttt{collect} : \forall\alpha.\alpha\ \mathsf{T} \to \alpha\ \mathsf{list}.\end{aligned}$$

Under system **S** this example is not typable. To see why let's try to assign `collect` the following reasonable type:

$$\texttt{collect} : \alpha\ \mathsf{T} \to \alpha\ \mathsf{list}.$$

We have no trouble deriving this type for the Empty-branch of the case-expression. However, from the program fragment: `collect t`, of the Node-branch, `collect` must have the following type:

$$\texttt{collect} : \alpha\ \mathsf{T}\ \mathsf{T} \to \alpha\ \mathsf{T}\ \mathsf{list},$$

since `t` has the following type:

$$\texttt{t} : \alpha\ \mathsf{T}\ \mathsf{T}.$$

Therefore collect must have a polymorphic type. Unfortunately, using intersection types, it is not possible to assign the type:

$$\texttt{collect} : (\alpha\ \mathsf{T} \to \alpha\ \mathsf{list}) \wedge (\alpha\ \mathsf{T}\ \mathsf{T} \to \alpha\ \mathsf{T}\ \mathsf{list}),$$

because when deriving the type $\alpha\ \mathsf{T}\ \mathsf{T} \to \alpha\ \mathsf{T}\ \mathsf{list}$ for `collect` we will require:

$$\texttt{collect} : \alpha\ \mathsf{T}\ \mathsf{T}\ \mathsf{T} \to \alpha\ \mathsf{T}\ \mathsf{T}\ \mathsf{list}.$$

This cyclic dilemma will continue indefinitely.

If we were to extend system **S** with infinite width intersection types such as the following:

$$\texttt{collect} : \ \bigwedge\nolimits_{i \in N}\ \tau_{i+1} \to \tau_i\ \mathsf{list},$$

where

$$\tau_i = \begin{cases}\alpha & \text{if } i = 0,\\ \tau_{i-1}\mathsf{T} & \text{otherwise},\end{cases}$$

then we could derive a typing derivation for this example. However, we since we do not know how to deal with infinite width intersection types we reject this idea and resort to system $\mathbf{HM}^{\forall}$ and polymorphic recursion with $\forall$-types.

Uncoupling this examples is impossible.

5.2 BAR

This example is a bit contrived but displays an interesting form of polymorphic recursion that is impossible to alleviate by uncoupling. Assuming the second argument to `BAR` is the `f` defined in the example, `BAR` can be understood by the following mathematical formula:

$$\mathtt{BAR}\; x\; (\lambda x.x \times 2)\; Z = Z * 2^{2^{\#\text{ of recursive calls}}} = Z * 2^{2^{\log_2(4/x)}}.$$

Below we show the example program.

```
let val r = fn i => i >= 4
    val f = fn i => i * 2
    val a = 5
    val rec BAR = fn x => fn F => fn Z =>
                  if r x
                  then F Z
                  else BAR (f x) (fn v => fn w => v (v w)) F Z
in BAR 1 f a end
```

SML Type Checker Reports:

```
Error: right-hand-side of clause doesn't agree with
       function result type [circularity]
  expression:  (('Z -> 'Z) -> 'Z -> 'Z) ->
               ('Z -> 'Z) -> 'Z -> 'Z
  result type:  (('Z -> 'Z) -> 'Z -> 'Z) ->
                (('Z -> 'Z) -> 'Z -> 'Z) ->
                ('Z -> 'Z) -> 'Z -> 'Z
  in declaration:
    BAR = (fn x => (fn <pat> => <exp>))

Error: operator and operand don't agree [literal]
  operator domain: ('Z -> 'Z) -> 'Z -> 'Z
  operand:         int -> int
  in expression:
    (BAR 1) f
```

This example, much like the previous, requires an infinite width intersection type. To see why, observe that both sides of the if-expression in the body of BAR are required to of the same type by the rule (If). Assume, without a loss of generality, that the arguments to BAR have the following types:

$$\begin{aligned}&\mathtt{x} : \mathsf{int}\\&\mathtt{F} : \mathsf{int} \to \mathsf{int}\\&\mathtt{Z} : \mathsf{int}.\end{aligned}$$

then the then-branch has type: int. As a result, BAR must have the following type:

$$\mathtt{BAR} : \mathsf{int} \to (\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}.$$

Also as a result, the else-branch must have type: int. If this is to occur then the result of BAR applied to its three arguments in the else-branch must be type: $\mathsf{int} \to \mathsf{int}$. This can only happen if the occurrence of BAR within the else-branch has the following type:

$$\mathtt{BAR} : \mathsf{int} \to ((\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})) \to (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}).$$

Just as we saw in the last example this issue can be resolved if we give BAR the type:

$$\begin{aligned}\mathtt{BAR} :&(\mathsf{int} \to (\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}) \wedge\\&(\mathsf{int} \to ((\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})) \to (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})).\end{aligned}$$

However, now the rule $(\wedge)$ requires us to type BAR as both components of the above intersection. Typing it as the second component will require us to expand the type of BAR even more. This cycle makes an infinite intersection type for BAR imperative.

We will now show such an infinite intersection type. Consider the following type:

$$\tau_i = \begin{cases}\alpha & \text{if } i = 0,\\ \tau_{i-1} \to \tau_{i-1} & \text{otherwise.}\end{cases}$$

We can use this to define an infinite intersection type for BAR as follows:

$$\text{BAR} : \ \wedge_{i \in N} \ \mathsf{int} \to \tau_{i+1} \to \tau_i \to \tau_i.$$

However, for the same reasons as before we choose to use $\forall$-types for this example. Under $\mathbf{HM}^{\forall}$ we assign the following type:

$$\mathtt{BAR} : \forall\alpha.\mathsf{int} \to (\alpha \to \alpha) \to \alpha \to \alpha.$$

The final type assigned to this example is: int.

6 Typable in $\mathbf{S}^{\forall}$ Only

6.1 Construct List

The following example presents a function, constList, that takes an input x and a number n. constList then constructs a list of 2^{2^n} elements, all equal to x. Here is the program.

```
let val rec constList = fn x => fn n =>
                        if (n = 0)
                        then [x,x]
                        else cons x
                             (tl (concat
                                 (constList (constList x (n-1))
                                            (n-1))))
    val applyCL = fn l1 => fn l2 => fn f =>
                  ((constList l1 (f l1)),
                   (constList l2 (f l2)))
in applyCL [1,2,3] [true,false,true] length end
```

SML Type Checker Reports:

```
Error: operator and operand don't agree [circularity]
  operator domain: 'Z list list * 'Z list list list
  operand:         'Z list list * 'Z list
  in expression:
    x :: tl (concat ((constList <exp>) (<exp> - <exp>)))

Error: operator and operand don't agree [literal]
  operator domain: _ list list
  operand:         int list
  in expression:
    applyCL (1 :: 2 :: 3 :: nil)
```

The above program is composed of one main function (constList), and one auxillary function (applyCL). The applyCM function makes two calls to constList (one for each input list) after applying an input function to each input list.

constList is a simple formulation of a function that constructs a list of the length described above without the use of arithmetical operations to explicitly calculate 2^{2^n}. Notice that a more concise formulation is not immediately evident.

This example is unique in that it requires both $\forall$-types and intersection types. The need for $\forall$-types stems from the clause:

```
constList (constList x (n-1)) (n-1)
```

This statement requires that the result of `constList` be the same type as the first argument to `constList`. Suppose the first argument to `constList` is of type α. We also know that the return type of `constList` must be of type α list from the then-branch of the conditional. If we try to assign `constList` the type: $(\alpha \to \mathsf{int} \to \alpha\ \mathsf{list}) \wedge (\alpha\ \mathsf{list} \to \mathsf{int} \to \alpha\ \mathsf{list\ list})$ then we run into the same cyclic dilemma that was described in the Collect example. Therefore the type of `constList` must be: $\forall\alpha.\alpha \to \mathsf{int} \to \alpha\ \mathsf{list}$ (infinite width intersection types are another option but, again, we choose $\forall$-types). Note that this example uses the same mechanism to require $\forall$-types as the Collect example. Yet, this example does not involve a recursive data type as the Collect example does. Instead this example uses only lists.

The need for intersection types arises when we inspect `applyCL`. Notice that we would like `f` to be a polymorphic argument to `applyCL` (this because we apply `applyCL` to two lists of different types). Since `f` is an argument it is impossible assign it a $\forall$-type since we have restricted our $\forall$-types such that quantifiers are not allowed inside a type. Therefore our only option is to assign `f` an intersection type.

Under $\mathbf{S}^\forall$ the following types can be assigned:

$$\begin{aligned}
\texttt{constMatrix} &: \forall\alpha.\alpha \to \mathsf{int} \to \alpha\ \mathsf{list} \\
\texttt{applyCL} &: \mathsf{int\ list} \to \mathsf{bool\ list} \to \\
&\quad ((\mathsf{int\ list} \to \mathsf{int\ list}) \wedge (\mathsf{bool\ list} \to \mathsf{bool\ list})) \to \\
&\quad (\mathsf{int\ list\ list} \times \mathsf{bool\ list\ list})
\end{aligned}$$

Uncoupling is not immediately evident for this example due to the fragment of the `constList` function that requires a $\forall$-type.

6.1.1 An Aside: SML/NJ vs. GHC

We now return to our comparison of SML/NJ and GHC error reporting. The BAR example, this example, and the following example (Delay) all demonstrate a difference between the error reporting of the two compilers that we have not yet seen. Here we show the Haskell translation and GHC error message of this example.

```
constList x 0 = [x,x]
```

```
constListx n = (x:(tail (concat (constList
                                  (constList x (n-1))
                                  (n-1)))))

applyCL l1 l2 f = ((constList l1 (f l1)), (constList l2 (f l2)))
```

GHC Type Checker Reports:

```
Occurs check: cannot construct the infinite type: a = [a]
    Expected type: [[a]]
    Inferred type: [a]
In the application ‘constList (constList x (n - 1)) (n - 1)’
In the first argument of ‘concat’, namely
    ‘(constList (constList x (n - 1)) (n - 1))’
```

Notice that the error message reported by GHC consists of only one message while SML/NJ reports two messages. This suggests that GHC may get to the heart of the error while SML/NJ reports numerous superfluous messages. On the other hand, perhaps SML/NJ error reporting is more precise, exposing every relevant error location. Since this is not the main objective of this report we leave this issue for future inquiry. However, the interested reader is advised to see [3] for more discussion.

7 Untypable

7.1 Delay Evaluation

The following example shows some of the limitations of polymorphic recursion using intersection types and ∀-types.

```
let val delay = fn x => fn () => x
    val rec nDelays = fn n => fn x =>
                      if n=0
                      then x
                      else nDelays (n-1) (delay x)
in nDelays 3 (fn x => x + 1) end
```

SML Type Checker Reports:

```
Error: right-hand-side of clause doesn’t agree with
       function result type [circularity]
  expression:  ’Z -> ’Z
  result type:  (unit -> ’Z) -> ’Z
  in declaration:
    nDelays = (fn n => (fn <pat> => <exp>))
```

```
Error: operator and operand don't agree [literal]
  operator domain: unit -> 'Z
  operand:         int -> int
  in expression:
    (nDelays 3) (fn x => x + 1)
```

Polymorphic recursion with ∀-types is not powerful enough to type this example. To see why there is no ∀-type let us inspect the example. First, it is easy to see that the type of `delay` is:

$$\texttt{delay} : \forall\alpha.\alpha \to \mathsf{unit} \to \alpha.$$

It is apparent that `n` has type int. Suppose next, that we give `x` type α. From the then-branch we see that the return type of the function must be of type α. So far we have assigned `nDelays` the following type:

$$\texttt{nDelays} : \forall\alpha.\mathsf{int} \to \alpha \to \alpha.$$

Next, according to the rule (If), we will make sure that the else-branch also has type α. This is where the problem manifests. The first argument to `nDelays`, `n-1`, clearly has type int. However, the second argument to `nDelays`, `delay x`, has type $\mathsf{unit} \to \alpha$ which, according to the type previously assigned to `nDelays`, means the else-branch has type unit $\to \alpha$.

Polymorphic recursion with intersection types is also not sufficient to type this example. To see why, first observe that to type:

```
fn n => fn x => if n=0
                then x
                else nDelays (n-1) (delay x),
```

we require `x` to have an intersection type. In the then-branch, `x` must have the same type as the result of `nDelays` which we will call τ. In the else-branch, we require `x` to have a type with strictly fewer units than τ has, since the call to `delay` will add one unit and `nDelays` does not accept arguments with a greater number of units than its return type. Therefore by assigning the following type to x:

$$\texttt{x} : \alpha \wedge (\mathsf{unit} \to \alpha),$$

we are able to derive the same type in both branches of the if-expression.

However, this presents a different problem. In order to type:

```
nDelays 3 (fn x => x + 1),
```

we require `nDelays` to have the type:

$$\mathsf{int} \to (\mathsf{int} \to \mathsf{int}) \to \tau.$$

but as a result of the subtyping relation rules, this type is not attainable if we require the second argument of `nDelays` to have an intersection type. We can see this from the following failed subtype derivation (where the boxed judgement is the failure point).

$$\dfrac{\dfrac{}{\mathsf{int} \le \mathsf{int}}\ (\text{S-Refl}) \qquad \dfrac{\boxed{\mathsf{int} \to \mathsf{int} \le ((\mathsf{int} \to \mathsf{int})\ \wedge\ \ldots)} \qquad \dfrac{}{\sigma \le \sigma}\ (\text{S-Refl})}{((\mathsf{int} \to \mathsf{int})\ \wedge\ \ldots) \to \sigma \le (\mathsf{int} \to \mathsf{int}) \to \sigma}\ (\text{S-Fun})}{\mathsf{int} \to (((\mathsf{int} \to \mathsf{int})\ \wedge\ \ldots) \to \sigma) \le \mathsf{int} \to ((\mathsf{int} \to \mathsf{int}) \to \sigma)}\ (\text{S-Fun})$$

Therefore we cannot derive an intersection type for this example using our system.

8 High-Order ∀-Polymorphism

In this section we describe the difference between our construction of **S** and $\mathbf{HM}^{\forall}$. We have chosen to disallow ∀-quantifiers anywhere inside a type. However, we allow ∧ to occur freely inside a type. At first glance, these choices may seem biased toward the intersection type system. The rationalization behind these choices was a decision to investigate the typability of programs for which there is a known type inference algorithm that does not rely on any type annotations. It is known how to infer types for high-rank uses of intersection types [14], but this is not the case for high-rank uses of ∀-types. This being said, if we were to consider a system of ∀-types that allowed arbitrary rank uses of ∀-types, then under this system we could type every example in this report that **S** types.

9 Conclusion

In summary, we have shown several examples of programs that require polymorphic recursion. Each program is not typable in the traditional Hindley-Milner system (**HM**). Some of the examples require ∧-types and others require ∀-types. Still others are not typable even with a combination of the two. We have seen that an intersection type system (**S**) can type many of our examples including Mycroft's example. To the best of our knowledge System **S** is the first type system that has been able to achieve this. Therefore, although a finite width intersection type system is not able to type all possible polymorphic recursive programs, it can type a significant subset with the possibility of decidable type inference.

References

[1] M. Emms, H. Leiss. Standard ml with polymorphic recursion, 1998. http://www.cis.uni-muenchen.de/projects/polyrec.html.

[2] C. Haack, J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In *Programming Languages & Systems, 12th European Symp. Programming*, vol. 2618 of *LNCS*. Springer-Verlag, 2003. Superseded by [3].

[3] C. Haack, J. B. Wells. Type error slicing in implicitly typed, higher-order languages. *Sci. Comput. Programming*, 50, 2004. Supersedes [2].

[4] F. Henglein. Type inference with polymorphic recursion. *ACM Trans. on Prog. Langs. & Systs.*, 15(2), 1993.

[5] T. Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[6] S. P. Jones. Haskell mailing list - subject: Polymorphic recursion, 1993. http://www.mail-archive.com/haskell@haskell.org/msg00517.html.

[7] S. P. Jones. Haskell mailing list - subject: Re: Polymorphic recursion, 1994. http://www.mail-archive.com/haskell@haskell.org/msg00492.html.

[8] S. P. Jones, J. Hughes. Haskell 98: A non-strict, purely functional language. Technical report, The Haskell 98 Committee, 1999. Currently available at http://haskell.org.

[9] S. P. Jones, M. Shields. Practical type inference for arbitrary-rank types. *Under Consideration for the Journal of Functional Programming*, 2004.

[10] A. J. Kfoury, J. Tiuryn, P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. on Prog. Langs. & Systs.*, 15(2), 1993.

[11] A. J. Kfoury, J. Tiuryn, P. Urzyczyn. The undecidability of the semi-unification problem. *Inform. & Comput.*, 102(1), 1993.

[12] A. J. Kfoury, J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, 1999. Superseded by [14].

[13] A. J. Kfoury, J. B. Wells. Principality and type inference for intersection types using expansion variables. Supersedes [12], 2003.

[14] A. J. Kfoury, J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3), 2004. Supersedes [12]. For omitted proofs, see the longer report [13].

[15] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings, 6th International Conference on Programming*. Springer-Verlag, 1984.

[16] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[17] Z. Somogyi. Comparing mercury and haskell, 2003. http://www.cs.mu.oz.au/research/mercury/information/comparison_with_haskell.html.

A Mycroft Typing Derivation in System S

Suppose we have the following types:

$$\begin{aligned}
\tau_{\mathsf{int}} &= \mathsf{int} \to \mathsf{int}\\
\tau_{\mathsf{bool}} &= \mathsf{bool} \to \mathsf{bool}\\
\tau_{\mathsf{int\ list}} &= \mathsf{int\ list} \to \mathsf{int\ list}\\
\tau_{\mathsf{bool\ list}} &= \mathsf{bool\ list} \to \mathsf{bool\ list}\\
\tau_{\times} &= \mathsf{int\ list} \times \mathsf{bool\ list}\\
\tau_{\wedge} &= (\tau_{\mathsf{int}} \to \tau_{\mathsf{int\ list}}) \wedge (\tau_{\mathsf{bool}} \to \tau_{\mathsf{bool\ list}}).
\end{aligned}$$

Also suppose we have the following context:

$$\Gamma = \mathtt{myMap} : \tau_{\wedge}, \mathtt{sqList} : \tau_{\mathsf{int\ list}}, \mathtt{compList} : \tau_{\mathsf{bool\ list}}.$$

Finally, suppose we have the following terms:

$$\begin{aligned}
&\mathtt{M} = \mathtt{fn}\ f \Rightarrow \mathtt{fn}\ l \Rightarrow \mathtt{if}(\mathtt{null}\ l)\ \mathtt{then}\ l\ \mathtt{else}\ \mathtt{cons}\ (f\ (\mathtt{hd}\ l))\ (\mathtt{myMap}\ f\ (\mathtt{tl}\ l))\\
&\mathtt{S} = \mathtt{fn}\ l \Rightarrow \mathtt{myMap}\ (\mathtt{fn}\ x \Rightarrow x * x)\ l\\
&\mathtt{C} = \mathtt{fn}\ l \Rightarrow \mathtt{myMap}\ \mathtt{not}\ l\\
&\mathtt{E} = (\mathtt{sqList}\ (\mathtt{cons}\ 2\ (\mathtt{cons}\ 4\ \mathtt{nil})), \mathtt{compList}\ (\mathtt{cons}\ \mathtt{true}\ (\mathtt{cons}\ \mathtt{false}\ \mathtt{nil}))).
\end{aligned}$$

Then we have the following typing derivation:

98.		$\tau_{\mathsf{bool}} \to \tau_{\mathsf{bool\ list}} \leq \tau_{\mathsf{bool}} \to \tau_{\mathsf{bool\ list}}$	(S-Refl)
97.		$\tau_{\wedge} \leq \tau_{\mathsf{bool}} \to \tau_{\mathsf{bool\ list}}$	(S-$\wedge$) from 98
96.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{myMap} : \tau_{\wedge}$	($\wedge$-Var)
95.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$l : \mathsf{bool\ list}$	($\wedge$-Var)
94.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{hd} : \mathsf{bool\ list} \to \mathsf{bool}$	($\wedge$-Const)
93.		$\tau_{\mathsf{int}} \to \tau_{\mathsf{int\ list}} \leq \tau_{\mathsf{int}} \to \tau_{\mathsf{int\ list}}$	(S-Refl)
92.		$\tau_{\wedge} \leq \tau_{\mathsf{int}} \to \tau_{\mathsf{int\ list}}$	(S-$\wedge$) from 93
91.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{myMap} : \tau_{\wedge}$	($\wedge$-Var)
90.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$l : \mathsf{int\ list}$	($\wedge$-Var)
89.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{hd} : \mathsf{int\ list} \to \mathsf{int}$	($\wedge$-Const)

88.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$l : \mathsf{bool\ list}$	$(\wedge\text{-Var})$
87.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{tl} : \tau_{\mathsf{bool\ list}}$	$(\wedge\text{-Const})$
86.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$f : \tau_{\mathsf{bool}}$	$(\wedge\text{-Var})$
85.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{myMap} : \tau_{\mathsf{bool}} \to \tau_{\mathsf{bool\ list}}$	(Sub) from 96, 97
84.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{hd}\ l : \mathsf{bool}$	(App) from 94, 95
83.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$f : \tau_{\mathsf{bool}}$	$(\wedge\text{-Var})$
82.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$l : \mathsf{int\ list}$	$(\wedge\text{-Var})$
81.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{tl} : \tau_{\mathsf{int\ list}}$	$(\wedge\text{-Const})$
80.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$f : \tau_{\mathsf{int}}$	$(\wedge\text{-Var})$
79.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{myMap} : \tau_{\mathsf{int}} \to \tau_{\mathsf{int\ list}}$	(Sub) from 91, 92
78.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{hd}\ l : \mathsf{int}$	(App) from 89, 90
77.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$f : \tau_{\mathsf{int}}$	$(\wedge\text{-Var})$
76.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{tl}\ l : \mathsf{bool\ list}$	(App) from 87, 88
75.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{myMap}\ f : \tau_{\mathsf{bool\ list}}$	(App) from 85, 86
74.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$f\ (\mathtt{hd}\ l) : \mathsf{bool}$	(App) from 83, 84
73.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{cons} : \mathsf{bool} \to \tau_{\mathsf{bool\ list}}$	$(\wedge\text{-Const})$
72.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{tl}\ l : \mathsf{int\ list}$	(App) from 81, 82
71.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{myMap}\ f : \tau_{\mathsf{int\ list}}$	(App) from 79, 80
70.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$f\ (\mathtt{hd}\ l) : \mathsf{int}$	(App) from 77, 78
69.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{cons} : \mathsf{int} \to \tau_{\mathsf{int\ list}}$	$(\wedge\text{-Const})$
68.	$\Gamma, l : \mathsf{int\ list}, x : \mathsf{int} \vdash$	$x : \mathsf{int}$	$(\wedge\text{-Var})$
67.	$\Gamma, l : \mathsf{int\ list}, x : \mathsf{int} \vdash$	$* : \mathsf{int} \to \tau_{\mathsf{int}}$	$(\wedge\text{-Const})$
66.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{myMap}\ f\ (\mathtt{tl}\ l) : \mathsf{bool\ list}$	(App) from 75, 76
65.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{cons}\ (f\ (\mathtt{hd}\ l)) : \tau_{\mathsf{bool\ list}}$	(App) from 73, 74

64.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$l : \mathsf{bool\ list}$	($\wedge$-Var)
63.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{null} : \mathsf{bool\ list} \to \mathsf{bool}$	($\wedge$-Const)
62.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{myMap}\ f\ (\mathtt{tl}\ l) : \mathsf{int\ list}$	(App) from 71, 72
61.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{cons}\ (f\ (\mathtt{hd}\ l)) : \tau_{\mathsf{int\ list}}$	(App) from 69, 70
60.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$l : \mathsf{int\ list}$	($\wedge$-Var)
59.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{null} : \mathsf{int\ list} \to \mathsf{bool}$	($\wedge$-Const)
58.	$\Gamma, l : \mathsf{int\ list}, x : \mathsf{int} \vdash$	$x : \mathsf{int}$	($\wedge$-Var)
57.	$\Gamma, l : \mathsf{int\ list}, x : \mathsf{int} \vdash$	$*x : \tau_{\mathsf{int}}$	(App) from 67, 68
56.	$\Gamma \vdash$	$4 : \mathsf{int}$	($\wedge$-Const)
55.	$\Gamma \vdash$	$\mathtt{cons} :$	($\wedge$-Const)
		$\mathsf{int} \to \mathsf{int\ list} \to \mathsf{int\ list}$	
54.	$\Gamma \vdash$	$\mathtt{false} : \mathsf{bool}$	($\wedge$-Const)
53.	$\Gamma \vdash$	$\mathtt{cons} :$	($\wedge$-Const)
		$\mathsf{bool} \to \mathsf{bool\ list} \to \mathsf{bool\ list}$	
52.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{cons}\ (f\ (\mathtt{hd}\ l))$	
		$(\mathtt{myMap}\ f\ (\mathtt{tl}\ l)) : \mathsf{bool\ list}$	(App) from 65, 66
51.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$l : \mathsf{bool\ list}$	($\wedge$-Var)
50.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$(\mathrm{null}\ l) : \mathsf{bool}$	(App) from 63, 64
49.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{cons}\ (f\ (\mathtt{hd}\ l))$	(App) from 61, 62
		$(\mathtt{myMap}\ f\ (\mathtt{tl}\ l)) : \mathsf{int\ list}$	
48.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$l : \mathsf{int\ list}$	($\wedge$-Var)
47.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$(\mathrm{null}\ l) : \mathsf{bool}$	(App) from 59, 60
46.	$\Gamma, l : \mathsf{int\ list}, x : \mathsf{int} \vdash$	$x * x : \mathsf{int}$	(App) from 57, 58
45.		$\tau_{\mathsf{int}} \to \tau_{\mathsf{int\ list}} \leq$	(S-Refl)

		$\tau_{\mathsf{int}} \longrightarrow \tau_{\mathsf{int\ list}}$	
44.		$\tau_{\wedge} \leq \tau_{\mathsf{int}} \longrightarrow \tau_{\mathsf{int\ list}}$	(S-$\wedge$) from 45
43.	$\Gamma, l : \mathsf{int\ list} \vdash$	$\mathtt{myMap} : \tau_{\wedge}$	($\wedge$-Var)
42.		$\tau_{\mathsf{bool}} \longrightarrow \tau_{\mathsf{bool\ list}} \leq$	(S-Refl)
		$\tau_{\mathsf{bool}} \longrightarrow \tau_{\mathsf{bool\ list}}$	
41.		$\tau_{\wedge} \leq \tau_{\mathsf{bool}} \longrightarrow \tau_{\mathsf{bool\ list}}$	(S-$\wedge$) from 42
40.	$\Gamma, l : \mathsf{bool\ list} \vdash$	$\mathtt{myMap} : \tau_{\wedge}$	($\wedge$-Var)
39.	$\Gamma \vdash$	$\mathtt{nil} : \mathsf{int\ list}$	($\wedge$-Const)
38.	$\Gamma \vdash$	$\mathtt{cons}\ 4 :$	(App) from 55, 56
		$\mathsf{int\ list} \rightarrow \mathsf{int\ list}$	
37.	$\Gamma \vdash$	$2 : \mathsf{int}$	($\wedge$-Const)
36.	$\Gamma \vdash$	$\mathtt{cons} :$	($\wedge$-Const)
		$\mathsf{int} \rightarrow \mathsf{int\ list} \rightarrow \mathsf{int\ list}$	
35.	$\Gamma \vdash$	$\mathtt{nil} : \mathsf{bool\ list}$	($\wedge$-Const)
34.	$\Gamma \vdash$	$\mathtt{cons}\ \mathtt{false} :$	(App) from 53, 54
		$\mathsf{bool\ list} \rightarrow \mathsf{bool\ list}$	
33.	$\Gamma \vdash$	$\mathtt{true} : \mathsf{bool}$	($\wedge$-Const)
32.	$\Gamma \vdash$	$\mathtt{cons} :$	($\wedge$-Const)
		$\mathsf{bool} \rightarrow \mathsf{bool\ list} \rightarrow \mathsf{bool\ list}$	
31.	$\Gamma, f : \tau_{\mathsf{bool}}, l : \mathsf{bool\ list} \vdash$	$\mathtt{if}(\mathtt{null}\ l)\ \mathtt{then}\ l$	(If) from 50, 51, 52
		$\mathtt{else}\ \mathtt{cons}\ (f\ (\mathtt{hd}\ l))$	
		$(\mathtt{myMap}\ f\ (\mathtt{tl}\ l)) : \mathsf{bool\ list}$	
30.	$\Gamma, f : \tau_{\mathsf{int}}, l : \mathsf{int\ list} \vdash$	$\mathtt{if}(\mathtt{null}\ l)\ \mathtt{then}\ l$	(If) from 47, 48, 49
		$\mathtt{else}\ \mathtt{cons}\ (f\ (\mathtt{hd}\ l))$	

		$(\texttt{myMap}\ f\ (\texttt{tl}\ l))$: int list	
29.	Γ, l : int list $\vdash$	$\texttt{fn}\ x \Rightarrow x * x : \tau_{\text{int}}$	(Abs) from 46
28.	Γ, l : int list $\vdash$	$\texttt{myMap} : \tau_{\text{int}} \to \tau_{\text{int list}}$	(Sub) from 43, 44
27.	Γ, l : bool list $\vdash$	$\texttt{not} : \tau_{\text{bool}}$	($\wedge$-Const)
26.	Γ, l : bool list $\vdash$	$\texttt{myMap} : \tau_{\text{bool}} \to \tau_{\text{bool list}}$	(Sub) from 40, 41
25.	$\Gamma \vdash$	cons 4 nil : int list	(App) from 38, 39
24.	$\Gamma \vdash$	cons 2 : int list $\to$ int list	(App) from 36, 37
23.	$\Gamma \vdash$	cons false nil : bool list	(App) from 34, 35
22.	$\Gamma \vdash$	cons true :	(App) from 32, 33
		bool list $\to$ bool list	
21.	$\Gamma, f : \tau_{bool} \vdash$	$\texttt{fn}\ l \Rightarrow \texttt{if}(\texttt{null}\ l)\ \texttt{then}\ l$	(Abs) from 31
		$\texttt{else cons}\ (f\ (\texttt{hd}\ l))$	
		$(\texttt{myMap}\ f\ (\texttt{tl}\ l)) : \tau_{\text{bool list}}$	
20.	$\Gamma, f : \tau_{int} \vdash$	$\texttt{fn}\ l \Rightarrow \texttt{if}(\texttt{null}\ l)\ \texttt{then}\ l$	(Abs) from 30
		$\texttt{else cons}\ (f\ (\texttt{hd}\ l))$	
		$(\texttt{myMap}\ f\ (\texttt{tl}\ l)) : \tau_{\text{int list}}$	
19.	Γ, l : int list $\vdash$	l : int list	($\wedge$-Var)
18.	Γ, l : int list $\vdash$	$\texttt{myMap}\ (\texttt{fn}\ x \Rightarrow x * x) : \tau_{\text{int list}}$	(App) from 28, 29
17.	Γ, l : bool list $\vdash$	l : bool list	($\wedge$-Var)
16.	Γ, l : bool list $\vdash$	$\texttt{myMap not} : \tau_{\text{bool list}}$	(App) from 26, 27
15.	$\Gamma \vdash$	cons 2 (cons 4 nil) : int list	(App) from 24, 25
14.	$\Gamma \vdash$	$\texttt{sqList} : \tau_{\text{int list}}$	($\wedge$-Var)
13.	$\Gamma \vdash$	cons true (cons false nil) :	(App) from 22, 23
		bool list	

12.	$\Gamma \vdash$	`compList` : $\tau_{\mathsf{bool\ list}}$	($\wedge$-Var)
11.	$\Gamma \vdash$	`fn` f => `fn` l =>	(Abs) from 21
		`if`(`null` l) `then` l	
		`else cons` (f (`hd` l))	
		(`myMap` f (`tl` l)) : $\tau_{\mathsf{bool}} \to \tau_{\text{bool list}}$	
10.	$\Gamma \vdash$	`fn` f => `fn` l =>	(Abs) from 20
		`if`(`null` l) `then` l	
			`else cons` (f (`hd` l))
		(`myMap` f (`tl` l)) : $\tau_{\mathsf{int}} \to \tau_{\mathsf{int\ list}}$	
9.	Γ, l : int list $\vdash$	`myMap` (`fn` x => $x * x$) l : int list	(App) from 18, 19
8.	Γ, l : bool list $\vdash$	`myMap not` l : bool list	(App) from 16, 17
7.	$\Gamma \vdash$	`sqList`	(App) from 14, 15
		(`cons` 2 (`cons` 4 `nil`)) : int list	
6.	$\Gamma \vdash$	`compList`	(App) from 12, 13
		(`cons true` (`cons false nil`)) :	
		bool list	
5.	$\Gamma \vdash$	`M` : $\tau_{\wedge}$	($\wedge$) from 10, 11
4.	$\Gamma \vdash$	`S` : $\tau_{\mathsf{int\ list}}$	(Abs) from 9
3.	$\Gamma \vdash$	`C` : $\tau_{\mathsf{bool\ list}}$	(Abs) from 8
2.	$\Gamma \vdash$	`E` : $\tau_{\times}$	(Pair) from 6, 7
1.	$\vdash$	`let val rec myMap` = `M`	($\wedge$-Rec) from 2, 3, 4, 5
		`and sqList` = `S`	
		`and compList` = `C in E end` : $\tau_{\times}$	

B Proof of Soundness of a Subsystem of S

In this section our goal is to show the soundness of a subsystem of **S**. We choose to eliminate the pair and conditional rules of **S** for the simplicity of the proof. We do not anticipate any difficulties in the proof of soundness if these additional rules were included. To achieve soundness we first define the operational semantics of our system. After this we prove the Inversion and Substitution Lemmas which allow us to show Subject Reduction holds.

Before we define the operational semantics of our system let us define the expressions and values of our system.

$$\begin{aligned} M, N \in \mathsf{Expressions} &::= x \mid c \mid \texttt{fn}\, x \mathrel{=>} M \mid M\ N \mid \texttt{let}\ \texttt{val}\ x = M\ \texttt{in}\ N\ \texttt{end} \mid \\ &\qquad \texttt{let}\ \texttt{val}\ \texttt{rec}\ x_1 = M_1\ \texttt{and}\ \ldots\ \texttt{and}\ x_n = M_n\ \texttt{in}\ N\ \texttt{end} \\ V \in \mathsf{Values} &::= x \mid \texttt{fn}\, x \mathrel{=>} M \end{aligned}$$

Now we review the static semantics of our system.

Subsystem of System **S** *Typing Rules:*

$$\frac{\mathsf{type}(c) = \tau}{\Delta \vdash c : \tau}\ (\wedge\text{-}\mathsf{Const}) \quad (\tau \text{ closed}) \qquad \frac{\Delta(x) = \tau}{\Delta \vdash x : \tau}\ (\wedge\text{-}\mathsf{Var})$$

$$\frac{\Delta, x : \tau \vdash M : \tau'}{\Delta \vdash \mathtt{fn}\ x \mathtt{=>} M : \tau \to \tau'}\ (\mathsf{Abs}) \qquad \frac{\Delta \vdash M : \tau \to \tau' \quad \Delta \vdash N : \tau}{\Delta \vdash MN : \tau'}\ (\mathsf{App})$$

$$\frac{\Delta \vdash M : \tau' \quad \Delta, x : \tau' \vdash N : \tau}{\Delta \vdash \mathtt{let}\ x = M\ \mathtt{in}\ N\ \mathtt{end} : \tau}\ (\wedge\text{-}\mathsf{Let})$$

$$\frac{\begin{array}{c}\Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash N : \tau \\ \Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M_p : \tau_p\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{let\ val\ rec}\ x_1 = M_1\ \mathsf{and} \ldots \\ \qquad \mathsf{and}\ x_n = M_n\ \mathtt{in}\ N\ \mathtt{end} : \tau\end{array}}\ (\wedge\text{-}\mathsf{Rec}) \qquad (1 \le p \le n)$$

$$\frac{\Delta \vdash M : \tau_i \quad i \in I}{\Delta \vdash M : \wedge_{i \in I}\, \tau_i}\ (\wedge) \quad (\mathsf{size}(I) \ge 2),\ (\mathsf{size}(I) \text{ is finite})$$

$$\frac{\Delta \vdash M : \tau \quad \tau \le \tau'}{\Delta \vdash M : \tau'}\ (\mathsf{Sub})$$

$$\frac{}{\tau \le \tau}\ (\mathsf{S\text{-}Refl}) \qquad \frac{\tau_1 \le \tau_2 \quad \tau_2 \le \tau_3}{\tau_1 \le \tau_3}\ (\mathsf{S\text{-}Trans})$$

$$\frac{\tau_1 \le \tau_1' \quad \tau_2' \le \tau_2}{\tau_1' \to \tau_2' \le \tau_1 \to \tau_2}\ (\mathsf{S\text{-}Fun}) \qquad \frac{\tau_i \le \tau_i' \quad i \in I \quad I \subseteq J}{\wedge_{i \in J}\, \tau_i \ \le\ \wedge_{i \in I}\, \tau_i'}\ (\mathsf{S\text{-}}\wedge)$$

Below are the dynamic semantics our subsystem.

Subsystem of System **S** *Operational Semantics:*

$$\frac{M \Rightarrow M'}{M\ N \Rightarrow M'\ N}\ (\mathsf{E\text{-}App1})$$

$$\frac{N \Rightarrow N'}{V\ N \Rightarrow V\ N'}\ (\mathsf{E\text{-}App2})$$

$$\frac{}{(\texttt{fn}\ x \texttt{=>}\ M)\ V \Rightarrow M[x := V]}\ (\mathsf{E\text{-}AppAbs})$$

$$\frac{M \Rightarrow M'}{\texttt{let val}\ x = M\ \texttt{in}\ N\ \texttt{end} \Rightarrow \texttt{let val}\ x = M'\ \texttt{in}\ N\ \texttt{end}}\ (\mathsf{E\text{-}Let1})$$

$$\frac{}{\texttt{let val}\ x = V\ \texttt{in}\ N\ \texttt{end} \Rightarrow N[x := V]}\ (\mathsf{E\text{-}Let2})$$

$$\frac{M_p[x_1 := M_1]\ldots[x_n := M_n] \Rightarrow M'_p}{\begin{array}{l}\texttt{let val rec}\ x_1 = V_1\ \texttt{and}\ldots\texttt{and}\ x_p = M_p\ \texttt{and}\ldots\\ \texttt{and}\ x_n = M_n\ \texttt{in}\ N\ \texttt{end} \Rightarrow\\ \texttt{let val rec}\ x_1 = V_1\ \texttt{and}\ldots\texttt{and}\ x_p = M'_p\ \texttt{and}\ldots\\ \texttt{and}\ x_n = M_n\ \texttt{in}\ N\ \texttt{end}\end{array}}\ (\mathsf{E\text{-}Rec1}) \qquad (1 \leq p \leq n)$$

$$\frac{}{\begin{array}{l}\texttt{let val rec}\ x_1 = V_1\ \texttt{and}\ldots\texttt{and}\ x_n = V_n\ \texttt{in}\ N\ \texttt{end} \Rightarrow\\ N[x_1 := V'_1]\ldots[x_n := V'_n]\end{array}}\ (\mathsf{E\text{-}Rec2})$$

Lemma B.1 (Inversion of the Subtype Relation)
If $\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2$, *then* $\tau'_1 \leq \tau_1$ *and* $\tau_2 \leq \tau'_2$.

Proof. There are three possible subtyping rules which may have been the last rule applied in the subtyping derivation of the judgement $\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2$. If the rule (S-Fun) was last applied then the result is obvious. If the rule (S-Refl) rule was last applied then the result can be obtained by straightforward induction on the premise of the rule. If the rule (S-Trans) was last applied then again we proceed by induction on the premises of the rule, but we must also apply the (S-Trans) rule to these results. □

Lemma B.2 (Inversion) *If* $\Delta \vdash \texttt{fn}\ x \texttt{=>} M : \tau_1 \to \tau_2$, *then* $\Delta, x : \tau_1' \vdash M : \tau_2$ *and* $\tau_1 \leq \tau_1'$.

Proof. By inspection of the inference rules we observe that the last rule applied in the typing derivation of the judgement $\Delta \vdash \texttt{fn}\ x \texttt{=>} M : \tau_1 \to \tau_2$ can only be one of two possibilities. We proceed by case analysis.

case: $D = \dfrac{\Delta, x : \tau_1 \vdash M : \tau_2}{\Delta \vdash \texttt{fn}\ x \texttt{=>} M : \tau_1 \to \tau_2}$ (Abs)

Then we have $\Delta, x : \tau_1 \vdash M : \tau_2$ where $\tau_1' = \tau_1$ and $\tau_1 \leq \tau_1$ by (S-Refl).

case: $D = \dfrac{\Delta \vdash \texttt{fn}\ x \texttt{=>} M : \tau_1' \to \tau_2' \quad \tau_1' \to \tau_2' \leq \tau_1 \to \tau_2}{\Delta \vdash \texttt{fn}\ x \texttt{=>} M : \tau_1 \to \tau_2}$ (Sub)

$\tau_1 \leq \tau_1'$ and $\tau_2' \leq \tau_2$	Subtype Inversion Lemma on $\tau_1' \to \tau_2' \leq \tau_1 \to \tau_2$
$\Delta, x : \tau_1'' \vdash M : \tau_2'$ and $\tau_1' \leq \tau_1''$	I.H. on $\Delta \vdash \texttt{fn}\ x \texttt{=>} M : \tau_1' \to \tau_2'$
$\tau_1 \leq \tau_1''$	(S-Trans) applied to $\tau_1 \leq \tau_1'$ and $\tau_1' \leq \tau_1''$
$\Delta, x : \tau_1'' \vdash M : \tau_2$	(Sub) applied to $\Delta, x : \tau_1'' \vdash M : \tau_2'$ and $\tau_2' \leq \tau_2$

□

Lemma B.3 (Weakening) *If* $\Delta \vdash M : \tau$, *then* $\Delta, \Delta' \vdash M : \tau$, *provided that* Δ, Δ' *is a valid context.*

Proof. The proof proceeds by straightforward induction on the structure of the derivation $D :: \Delta \vdash M : \tau$. The only case in which the context is examined is when the rule (Var) is the last rule applied in the derivation. It should be clear that (Var) is only applicable if the context Δ contains the assignment $x : \tau$. And by extending the context with additional, non-conflicting assignments we do not alter this property. □

Lemma B.4 (Substitution) *If* $\Delta \vdash N : \tau$ *and* $\Delta, x : \tau, \Delta' \vdash M : \tau'$, *then* $\Delta, \Delta' \vdash M[x := N] : \tau'$.

Proof. By structural induction on the derivation $D :: \Delta, x : \tau, \Delta' \vdash M : \tau'$.

We show only a few cases, as the rest follow the same pattern.

$$\textit{case: } D = \dfrac{\Delta, x : \tau, \Delta'(y) = \tau'}{\Delta, x : \tau, \Delta' \vdash y : \tau'}\ (\wedge\text{-}\mathsf{Var})$$

Depending on whether $x = y$ we have two subcases.

subcase: $x = y$ and $\tau = \tau'$

$x[x := N] = N$	Definition of Substitution
$\Delta, \Delta' \vdash N : \tau$	Weakening Lemma on assumption $\Delta \vdash N : \tau$

subcase: $x \neq y$

$y[x := N] = y$	Definition of Substitution
$\Delta, \Delta' \vdash y : \tau'$	Assumptions $\Delta, x : \tau, \Delta' \vdash y : \tau'$ and $x \neq y$

$$\textit{case: } D = \dfrac{\Delta, x : \tau, \Delta' \vdash M_1 : \tau_1' \to \tau' \quad \Delta, x : \tau, \Delta' \vdash M_2 : \tau_1'}{\Delta, x : \tau, \Delta' \vdash M_1 M_2 : \tau'}\ (\mathsf{App})$$

Depending on whether $x \in \mathsf{FV}(M_1)$ we have two subcases.

subcase: $x \in \mathsf{FV}(M_1)$
Depending on whether $x \in \mathsf{FV}(M_2)$ we have two subsubcases.

subsubcase: $x \in \mathsf{FV}(M_2)$

$\Delta, \Delta' \vdash M_1[x := N] : \tau_1' \to \tau'$	I.H. on $\Delta, x : \tau, \Delta' \vdash M_1 : \tau_1' \to \tau'$
$\Delta, \Delta' \vdash M_2[x := N] : \tau_1'$	I.H. on $\Delta, x : \tau, \Delta' \vdash M_2 : \tau_1'$
$\Delta, \Delta' \vdash M_1[x := N]M_2[x := N] : \tau'$	(App) applied to $\Delta, \Delta' \vdash M_1[x := N] : \tau_1' \to \tau'$ and $\Delta, \Delta' \vdash M_2[x := N] : \tau_1'$
$\Delta, \Delta' \vdash (M_1 M_2)[x := N] : \tau'$	Definition of Substitution

subsubcase: $x \notin \mathsf{FV}(M_2)$ and $\Delta, \Delta' \vdash M_2 : \tau_1'$

$\Delta, \Delta' \vdash M_1[x := N] : \tau'_1 \to \tau'$	I.H. on $\Delta, x : \tau, \Delta' \vdash M_1 : \tau'_1 \to \tau'$
$\Delta, \Delta' \vdash (M_1[x := N])M_2 : \tau'$	(App) applied to $\Delta, \Delta' \vdash M_1[x := N] : \tau'_1 \to \tau'$ and $\Delta, \Delta' \vdash M_2 : \tau'_1$
$\Delta, \Delta' \vdash (M_1 M_2)[x := N] : \tau'$	Definition of Substitution

subcase: $x \notin \mathtt{FV}(M_1)$ and $\Delta, \Delta' \vdash M_1 : \tau'_1 \to \tau'$
Depending on whether $x \in \mathtt{FV}(M_2)$ we have two subsubcases.

subsubcase: $x \in \mathtt{FV}(M_2)$

$\Delta, \Delta' \vdash M_2[x := N] : \tau'_1$	I.H. on $\Delta, x : \tau, \Delta' \vdash M_2 : \tau'_1$
$\Delta, \Delta' \vdash M_1(M_2[x := N]) : \tau'$	(App) applied to $\Delta, \Delta' \vdash M_1 : \tau'_1 \to \tau'$ and $\Delta, \Delta' \vdash M_2[x := N] : \tau'_1$
$\Delta, \Delta' \vdash (M_1 M_2)[x := N] : \tau'$	Definition of Substitution

subsubcase: $x \notin \mathtt{FV}(M_2)$ and $\Delta, \Delta' \vdash M_2 : \tau'_1$

$(M_1 M_2)[x := N] = M_1 M_2$	Definition of Substitution
$\Delta, \Delta' \vdash M_1 M_2 : \tau'$	Assumptions $\Delta, x : \tau, \Delta' \vdash M_1 M_2 : \tau'$, $x \notin \mathtt{FV}(M_1)$, and $x \notin \mathtt{FV}(M_2)$

The remaining cases are similar. □

Theorem B.5 (Subject Reduction) *If $\Delta \vdash M : \tau$ and $M \Rightarrow M'$, then $\Delta \vdash M' : \tau$.*

Proof. By structural induction on the derivation of $D :: \Delta \vdash M : \tau$.

case: $D = \dfrac{\mathsf{type}(c) = \tau}{\Delta \vdash c : \tau}\ (\wedge\text{-}\mathsf{Const})$
Can't happen because there are no evaluation rules for constants.

case: $D = \dfrac{\Delta(x) = \tau}{\Delta \vdash x : \tau}\ (\wedge\text{-}\mathsf{Var})$
Can't happen because there are no evaluation rules for variables.

case: $D = \dfrac{\Delta, x : \tau \vdash M : \tau'}{\Delta \vdash \texttt{fn}\, x \texttt{ =>} M : \tau \to \tau'}$ (Abs)

Can't happen because there are no evaluation rules for abstractions.

case: $D = \dfrac{\Delta \vdash M : \tau \to \tau' \quad \Delta \vdash N : \tau}{\Delta \vdash MN : \tau'}$ (App)

From the operational semantics there are three ways we can derive $M \Rightarrow M'$. We proceed by cases.

subcase: $M \Rightarrow M'$

$M\ N \Rightarrow M'\ N$	(E-App1)
$\Delta \vdash M' : \tau \to \tau'$	I.H. on $\Delta \vdash M : \tau \to \tau'$ and $M \Rightarrow M'$
$\Delta \vdash M'N : \tau'$	(App) applied to $\Delta \vdash M' : \tau \to \tau'$ and $\Delta \vdash N : \tau$

subcase: M is a value and $N \Rightarrow N'$

$M\ N \Rightarrow M\ N'$	(E-App2)
$\Delta \vdash N' : \tau$	I.H. on $\Delta \vdash N : \tau$ and $N \Rightarrow N'$
$\Delta \vdash MN' : \tau'$	(App) applied to $\Delta \vdash M : \tau \to \tau'$ and $\Delta \vdash N' : \tau$

subcase: $M = \texttt{fn}\ x \texttt{ =>} M'$ and N is a value

$(\texttt{fn}\ x \texttt{ =>} M')\ N \Rightarrow M'[x := N]$	(E-AppAbs)
$\Delta, x : \tau'' \vdash M' : \tau'$, where $\tau \leq \tau''$	Inversion Lemma on
	$\Delta \vdash \texttt{fn}\ x \texttt{ =>} M' : \tau \to \tau'$
$\Delta \vdash N : \tau''$	(Sub) applied to $\Delta \vdash N : \tau$
	and $\tau \leq \tau''$
$\Delta \vdash M'[x := N] : \tau'$	Substitution Lemma on
	$\Delta, x : \tau'' \vdash M' : \tau'$ and $\Delta \vdash N : \tau''$

case: $D = \dfrac{\Delta \vdash M : \tau' \quad \Delta, x : \tau' \vdash N : \tau}{\Delta \vdash \texttt{let}\ x = M\ \texttt{in}\ N\ \texttt{end} : \tau}$ ($\wedge$-Let)

From the operational semantics there are two ways we can derive $M \Rightarrow M'$. We proceed by cases.

subcase: $M \Rightarrow M'$

$\texttt{let val } x = M \texttt{ in } N \texttt{ end} \Rightarrow$	
$\texttt{let val } x = M' \texttt{ in } N \texttt{ end}$	$(\mathsf{E\text{-}Let1})$
$\Delta \vdash M' : \tau'$	I.H. on $\Delta \vdash M : \tau'$ and $M \Rightarrow M'$
$\Delta \vdash \texttt{let val } x = M' \texttt{ in } N \texttt{ end} : \tau$	$(\wedge\text{-}\mathsf{Let})$ applied to $\Delta \vdash M' : \tau'$ and $\Delta, x : \tau' \vdash N : \tau$

subcase: M is a value

$\texttt{let val } x = M \texttt{ in } N \texttt{ end} \Rightarrow N[x := M]$	$(\mathsf{E\text{-}Let2})$
$\Delta \vdash N[x := M] : \tau$	Substitution Lemma on $\Delta, x : \tau' \vdash N : \tau$ and $\Delta \vdash M : \tau'$

case: $D = \dfrac{\Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash N : \tau \quad \Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M_p : \tau_p}{\Delta \vdash \texttt{let val rec } x_1 = M_1 \texttt{ and } \ldots \texttt{ and } x_n = M_n \texttt{ in } N \texttt{ end} : \tau}$ $(\wedge\text{-}\mathsf{Rec})$

From the operational semantics there are two ways we can derive $M \Rightarrow M'$. We proceed by cases.

subcase: $M_p \Rightarrow M'_p$, where $1 \leq p \leq n$

$\texttt{let val rec } x_1 = M_1 \texttt{ and} \ldots$ (E-Rec1)

$\texttt{and } x_p = M_p \texttt{ and} \ldots$

$\texttt{and } x_n = M_n \texttt{ in } N \texttt{ end} \Rightarrow$

$\texttt{let val rec } x_1 = M_1 \texttt{ and} \ldots$

$\texttt{and } x_p = M'_p \texttt{ and} \ldots$

$\texttt{and } x_n = M_n \texttt{ in } N \texttt{ end}$

$\Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M'_p : \tau_p$ I.H. on $\Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M_p : \tau_p$ and $M_p \Rightarrow M'_p$

$\Delta \vdash \texttt{let val rec } x_1 = M_1 \texttt{ and} \ldots$ ($\wedge$-Rec) applied to $\Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M'_p : \tau_p$ and

$\texttt{and } x_p = M'_p \texttt{ and} \ldots$ $\Delta, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash N : \tau$

$\texttt{and } x_n = M_n \texttt{ in } N \texttt{ end} : \tau$

subcase: $M_1 \ldots M_n$ are all values.

$\texttt{let val rec } x_1 = M_1 \texttt{ and} \ldots$ (E-Rec2)

$\texttt{and } x_n = M_n \texttt{ in } N \texttt{ end} \Rightarrow$

$N[x := M_1] \ldots [x := M_n]$

$\Delta \vdash N[x := M_1] \ldots [x := M_n] : \tau$ By n applications of the Substitution Lemma

case: $D = \dfrac{\Delta \vdash M : \tau_i \quad i \in I}{\Delta \vdash M : \wedge_{i \in I} \tau_i} (\wedge)$

$\Delta \vdash M' : \tau_i \quad i \in I$ I.H. on $\Delta \vdash M : \tau_i \quad i \in I$ and $M \Rightarrow M'$

$\Delta \vdash M' : \wedge_{i \in I} \tau_i$ $(\wedge)$ applied to $\Delta \vdash M' : \tau_i \quad i \in I$

case: $D = \dfrac{\Delta \vdash M : \tau \quad \tau \leq \tau'}{\Delta \vdash M : \tau'} (\mathsf{Sub})$

$\Delta \vdash M' : \tau$ I.H. on $\Delta \vdash M : \tau$ and $M \Rightarrow M'$

$\Delta \vdash M' : \tau'$ (Sub) applied to $\Delta \vdash M' : \tau$ and $\tau \leq \tau'$

$\square$